Distributed Decision-Making and TaskCoordination in Dynamic, Uncertain andReal-Time Multiagent Environments

Sébastien Paquet

SÉBASTIEN PAQUET

Distributed Decision-Making and Task Coordination in Dynamic, Uncertain and Real-Time Multiagent Environments

Thèse présentée
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de doctorat en informatique
pour l'obtention du grade de Philosophiæ Doctor, (Ph.D.)

Faculté de Sciences et Génie
UNIVERSITÉ LAVAL
QUÉBEC

Janvier 2006

Résumé

La prise de décision dans l'incertain et la coordination sont au coeur des systèmes multiagents. Dans ce type de systèmes, les agents doivent être en mesure de percevoir leur environnement et de prendre des décisions en considérant les autres agents. Lorsque l'environnement est partiellement observable, les agents doivent être en mesure de gérer cette incertitude pour prendre des décisions les plus éclairées possible en considérant les informations incomplètes qu'ils ont pu acquérir. Par ailleurs, dans le contexte d'environnements multiagents coopératifs, les agents doivent être en mesure de coordonner leurs actions de manière à pouvoir accomplir des tâches demandant la collaboration de plus d'un agent.

Dans cette thèse, nous considérons des environnements multiagents coopératifs complexes (dynamiques, incertains et temps-réel). Pour ce type d'environnements, nous proposons une approche de prise de décision dans l'incertain permettant une coordination flexible entre les agents. Plus précisément, nous présentons un algorithme de résolution en ligne de processus de décision de Markov partiellement observables (POMDPs).

Par ailleurs, dans de tels environnements, les tâches que doivent accomplir les agents peuvent devenir très complexes. Dans ce cadre, il peut devenir difficile pour les agents de déterminer le nombre de ressources nécessaires à l'accomplissement de chacune des tâches. Pour résoudre ce problème, nous proposons donc un algorithme d'apprentissage permettant d'apprendre le nombre de ressources nécessaires à l'accomplissement des tâches selon les caractéristiques de celles-ci. Dans un même ordre d'idée, nous proposons aussi une méthode d'ordonnancement permettant d'ordonner les différentes tâches des agents de manière à maximiser le nombre de tâches pouvant être accomplies dans un temps limité.

Toutes ces approches ont pour but de permettre la coordination d'agents pour l'accomplissement efficace de tâches complexes dans un environnement multiagent partiellement observable, dynamique et incertain. Toutes ces approches ont démontré leur efficacité lors de tests effectués dans l'environnement de simulation de la RoboCup-Rescue.

Abstract

Decision-making in uncertainty and coordination are at the heart of multiagent systems. In this kind of systems, agents have to be able to perceive their environment and take decisions while considering the other agents. When the environment is partially observable, agents have to be able to manage this uncertainty in order to take the most enlightened decisions they can based on the incomplete information they have acquired. Moreover, in the context of cooperative multiagent environments, agents have to coordinate their actions in order to accomplish complex tasks requiring more then one agent.

In this thesis, we consider complex cooperative multiagent environments (dynamic, uncertain and real-time). In this kind of environments, we propose an approach of decision-making in uncertainty that enable the agents to flexibly coordinate themselves. More precisely, we present an online algorithm for partially observable Markov decision processes (POMDPs).

Furthermore, in such complex environments, agent's tasks can also become quite complex. In this context, it could be complicated for the agents to determine the required number of resources to accomplish each task. To address this problem, we propose a learning algorithm to learn the number of resources necessary to accomplish a task based on the characteristics of this task. In a similar manner, we propose a scheduling approach enabling the agents to schedule their tasks in order to maximize the number of tasks that could be accomplish in a limited time.

All these approaches have been developed to enable the agents to efficiently coordinate all their complex tasks in a partially observable, dynamic and uncertain multiagent environment. All these approaches have demonstrated their effectiveness in tests done in the RoboCupRescue simulation environment.

Avant-propos

J'aimerais profiter de ces quelques lignes pour remercier les personnes qui ont collaboré à l'aboutissement de cette thèse. Dans un premier temps, j'aimerais remercier mon directeur de recherche, M. Brahim Chaib-draa, pour ses précieux conseils, sa disponibilité et ses encouragements. Il a toujours été présent pour m'aider à faire avancer mes recherches.

J'aimerais également remercier tous les étudiants stagiaires qui ont travaillé avec moi sur ce projet: Nicolas Bernier, Ludovic Tobin et Stéphane Ross. Ils m'ont beaucoup aidé lors de la programmation en vue des compétitions et pour la rédaction de certains articles. Sans le très bon travail, nous n'aurions pas pu terminer 6^{e} et 2^{e} lors des compétitions internationales de la RoboCupRescue simulation de 2003 et 2004.

De plus, j'aimerais remercier le CRSNG, le RDDC-Valcartier et le laboratoire DAMAS pour leur support financier tout au long de mes études graduées.

Par ailleurs, il m'est important de mentionner la collaboration des membres du DAMAS sans qui mes études n'auraient pas été aussi plaisantes: Frédérick Asselin, Patrick Beaumont, Mathieu Bergeron, Nicolas Bernier, Étienne Bolduc, Charles Desjardins, Vincent Dumouchel, Simon Hallé, Marc-André Labrie, Jean-Claude Lacombe, Julien Laumonier, Philippe Lefebvre, Ève Levesque, Thierry Moyaux, Jean-François Morissette, Philippe Pasquier, Mathieu Pelletier, Pierrick Plamondon, Stéphane Ross, Martin Soucy et Ludovic Tobin.

J'aimerais également remercier ma compagne Sara pour sa gentillesse, sa patience, sa compréhension et son support tout au long de mes études. Finalement, j'aimerais remercier et dédier cette thèse à mes parents et à ma soeur pour leurs extraordinaires amour, compréhension, générosité et attention qu'ils ont toujours eus à mon égard.

Sébastien Paquet

À mes parents Yvon et Gloria.

Nul ne peut solutionner le monde!
Mais localement, tout problème peut
être amenuisé!

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Distributed Decision-making and task coordination are at the heart of multiagent systems. These systems are composed of many interacting autonomous software entities (agents) that have to perceive their environment, reason on these perceptions and choose some actions in order to achieve their goals. These actions' choices are complicated by the fact that agents are not alone in the considered environment and this forces them to consider the other agents in their decisions. Moreover, agents may have to coordinate themselves to accomplish complex tasks that need more than one agent to be accomplished. These tasks may be so complicated that the agents may not know the number of agents required to accomplish the tasks or the time they have before the tasks become obsolete.

Similarly, the distributed decision-making process may be even more complex if the environment is partially observable, dynamic, uncertain and real-time. Partially observable means that an agent can only perceive a small part of the environment, which forces the agent to take its decision with incomplete information. Dynamic and uncertain means that the environment is in constant evolution and that an agent cannot know with certainty how the world will evolve or how its actions will impact the world. And finally, real-time means that the agent has to respect some time constraints when making its decisions.

Studying such multiagent systems in so complex environments may sound quite difficult and in deed it is, but this reflects the reality, since the real world has all these characteristics. One should therefore overcome difficulties sustained by such systems in order for autonomous agents to be effective in real life applications like Mars rovers (Estlin et al. (2005)), unmanned vehicles (Karim and Heinze (2005)), rescue robots (Nourbakhsh et al. (2005)), etc.

This thesis considers complex multiagent systems in which agents evolve in dynamic, uncertain and real-time environments. In this context, it proposes some approaches to

enable the agents to make good decisions and to coordinate themselves in order to accomplish their tasks as efficiently as possible. Briefly, this thesis proposes:

- an online POMDP algorithm that allows agents to choose efficient actions in large partially observable environments;
- a learning algorithm that allows agents to learn the required number of resources to accomplish a complex task;
- a scheduling approach that allows agents to optimize the number of tasks accomplished in a short time.

1.1 RoboCupRescue Simulation

The test-bed environment that motivated us and that has been used to test our approaches is the RoboCupRescue simulation (Kitano (2000)). This environment consists of a simulation of an earthquake happening in a city. The goal of the agents (representing firefighters, policemen and ambulance teams) consists in minimizing the damages caused by a big earthquake, such as civilians buried, buildings on fire and roads blocked. The RoboCupRescue simulation environment has all the complex characteristics mentioned previously and it is thus a complex test-bed for cooperative multiagent systems.

In the RoboCupRescue simulation there are three main types of agents: *FireBrigade*, *PoliceForce* and *AmbulanceTeam*. The *FireBrigade* agents have to extinguish fires, the *PoliceForce* agents have to clear the roads and the *AmbulanceTeam* agents have to rescue civilians. In a simulation, there can be up to 38 of these rescue agents that have to cooperate with each other in order to maximize the number of survivors and to minimize the damages caused by the fires.

All rescue agents only have really limited perceptions, therefore the environment is highly partially observable. In addition, all rescue agents have to respect a hard real-time constraint since they have to return their actions in less than a second after they received their perceptions. Rescue agents have limited communication capabilities and they have limited amount of resources. Moreover, rescue agents have to deal with an highly dynamic environment in which the fires can spread fast and the civilians health can deteriorate rapidly.

A complex cooperative multiagent environment like the RoboCupRescue simulation demands flexible algorithms that can cope with this complexity. The following subsections give an overview of the algorithms that are presented in detail in the following chapters of this thesis.

1.2 Online POMDP Algorithm

Agents evolving in the RoboCupRescue simulation environment have to choose an action at each time step even though they only have an incomplete representation of their environment. When faced with a partially observable environment like the RoboCupRescue simulation, a general model for sequential decision problems consists in using Partially Observable Markov Decision Processes (POMDPs).

A lot of problems can be modelled with POMDPs, but very few can be solved because of their computational complexity (POMDPs are PSPACE-complete (Papadimitriou and Tsitsiklis (1987))). The main problem with POMDPs is that their complexity makes them applicable only on small environments. However, most problems of interest have a huge state space, which motivates the search for approximation methods (Hauskrecht (2000); Pineau et al. (2003); Smith and Simmons (2005); Spaan and Vlassis (2005)). This is especially the case for multiagent systems where there is often a huge state space with autonomous agents interacting with each other.

Most recent algorithms for solving POMDPs learn a complete policy offline, defining which action to take in all possible situation. While these approximation approaches are quite efficient on small problems, they often cannot deal with larger environments. In this thesis, instead of computing a complete policy offline, we present an online approach based on a look-ahead search in the belief state space to find the best action to execute, at each cycle in the environment (Paquet et al. (2005b,c)). Our algorithm, called RTBSS (Real-Time Belief Space Search), only explores reachable belief states starting from the agent's current belief state. By doing an online search, we avoid the overwhelming complexity of computing a policy for every possible situation the agent could encounter. Since there is no computation offline, the algorithm is immediately applicable to previously unseen environments, if the environments' dynamics are known. On the other hand, if the agent has some time offline and online, we have also developed some hybrid algorithms that can improve the offline policies with an online search.

With the RTBSS algorithm, presented in Chapter 3, we have developed a local reward function that dynamically defines the rewards as they are needed for the POMDP algorithm. This enables the rewards to be defined only for reachable states and not for the whole state space. In addition, this reward function is used as a coordination mechanism between the *PoliceForce* agents. All these agents are executing their own instance of our RTBSS algorithm and the coordination among these agents is controlled by the reward function. In fact, this function defines sort of smooth boundaries between the *PoliceForce* agents so that they can efficiently divide the different tasks among themselves. This coordination mechanism is quite flexible and it can adjust to environment changes.

With the POMDP algorithm and the local reward function, *PoliceForce* agents can be coordinated in order to divide simple tasks that can be achieved by only one agent. However, if the tasks are more complex and if they need to be accomplished by more than one agent, then the coordination process needs to regroup the agents in order to achieve the tasks. Moreover, if there is some uncertainty, the agents may even not know how many agents are required to accomplish each task. In this case, the first step in order to achieve good performances consists in learning the required number of agents (considered here as resources) to accomplish each task. In the next sub-section, we briefly present the selective perception reinforcement learning algorithm that has been developed to this end.

1.3 Task Allocation Learning

As mentioned previously, if agents are faced with complex tasks, it might be hard for them to determine how many agents are required to accomplish each task, which is important information for the coordination process. Without this information, agents would not be able to divide up the different tasks efficiently.

To learn the required number of agents for each task, we have developed a selective perception reinforcement learning algorithm (Paquet et al. (2004b)), which is useful to manage a large set of possible task descriptions with discrete or continuous variables. It enables us to regroup common task descriptions together, thus greatly diminishing the number of different task descriptions. Starting from this, the reinforcement learning algorithm can work with a relatively small state space.

The algorithm that we developed to this end is presented in Chapter 4 and has been applied to the *FireBrigade* agents. Notice that these agents need to estimate the number of *FireBrigade* agents necessary to extinguish a fire. These tasks of extinguishing fires are quite complex because the number of required agents depends on many factors like: the fire's intensity, the building's size, the building's composition, etc.

Our tests in the RoboCupRescue simulation environment showed that the *FireBrigade* agents are able to learn a compact representation of the state space, facilitating their task of learning good expected rewards. Furthermore, agents were also able to use those expected rewards to choose the right number of agents to assign to each task. With this information, agents coordinate themselves on the different tasks in an effective way, thus improving the group performance.

In the next sub-section, we introduce another approach that not only has to learn an estimation of a task characteristic, but also has to carefully schedule the tasks, because the order in which the tasks are accomplish influences a lot the performances of the rescuing agents.

1.4 Task Scheduling in Complex Multiagent Environments

In complex multiagent systems like the RoboCupRescue, the agents could faced with many tasks to accomplish. Moreover, when the tasks have different deadlines, the order in which the tasks are accomplished becomes quite important. In such settings, agents have to decide how many agents to assign to each task and in which order they should accomplish the tasks. To achieve that, agents need a good scheduling algorithm that can maximize the number of tasks accomplished before their deadline.

The scheduling approach, presented in Chapter 5, has been applied to the *AmbulanceTeam* agents. These agents have to rescue the civilians, but the number of civilians that can be rescued depends a lot on the order in which they are rescued. Furthermore, the scheduler agents have to be able to adapt the schedules frequently to take the dynamic changes of the environment into consideration. Another challenge is that the RoboCupRescue environment is partially observable, therefore the tasks are not known at the beginning of the simulation. Agents thus have to explore the environment to find the tasks and then incorporate them in their schedule. Notice that since we are considering uncertain environments, the tasks' parameters could even change between two observations. In this case, the system's performance depends not only on the maximization of the optimization criterion, but also on the agents' capacity to adapt their schedule efficiently.

In Chapter 5, we analyze the advantages and the disadvantages of distributing or not the scheduling process in a complex multiagent system. More precisely, we study the impact on the agents' efficiency and on the amount of information transmitted when using centralized and decentralized scheduling. We also study the usefulness of distributing the execution of the tasks in a scheduling problem and thus accomplishing goals in parallel, compared to the strategy of concentrating all resources to accomplish one goal at a time.

In similarity with the approach presented in the previous section, agents also had to learn to estimate one of the characteristics of the tasks. One important parameter that the *AmbulanceTeam* agents have to learn is the expected death time of the civilians. This parameter is quite important for the *AmbulanceTeam* agents to make good schedules. To this end, a K-Nearest-Neighbors (KNN) approach has been used to learn the damage progression of the civilians which is used to estimate the expected death time.

Now that the main approaches of this thesis have been briefly presented, we emphasize in the next section the contributions of these approaches to the multiagent research community.

1.5 Contributions

The problems of decision-making and task coordination are really important to the field of multiagent systems. When the environment is only partially observable, the decision process of the agents may become quite hard. Agents then have to choose their actions based on incomplete information. It becomes difficult for the agents to stay coordinated when they cannot perceive the other agents and when the environment is in constant changes. The main contributions of this thesis is to propose coordination and decision-making algorithms capable of dealing with such complex cooperative multiagent systems. To be more precise, here is a list of the main contributions of this thesis:

An online POMDP algorithm. We have conceived a real online POMDP algorithm. Most POMDP algorithms try to solve the problem offline by defining a policy for all possible situations the agent could encounter. This offline process is quite complex and this limits the applicability of most offline approaches to small problems. Many claimed online algorithms need in fact a lot of executions in the environment to learn a good policy. Our algorithm is different, because it does not need any calculation time offline and it is immediately efficient, even in previously unseen configurations of the environment.

Pruning strategy. We have defined a pruning strategy to accelerate the search in the belief state space. We have combined a limited depth first search strategy with a pruning strategy that uses dynamically updated bounds based on the solutions found at the maximal depth of the search. The pruning of the tree is also accelerated by sorting the actions in order of their expected efficiency. Since the more interesting actions are tried first, there is more chance that the first branches developed have better values, thus better bounds for the pruning condition.

Theoretical bound. The algorithm has a theoretical bound, thus we can guaranty that the distance between the policy defined by our algorithm and the optimal policy is bounded. The error gets smaller as the agent evolves in the environment because of the discount factor.

Hybrid approaches. We developed some hybrid approaches that use the RTBSS online search strategy mixed with approximate offline strategies. We present three new algorithms: RTBSS-QMDP, RTBSS-PBVI-QMDP and RTDPBSS. The results show that the performances of the hybrid approaches are often better than the performances of the online approach or the offline approach taken alone.

Belief state for dynamic environments. We have conceived an approach to maintain a belief state based on the real agent's observations. This helps the agent to manage the highly dynamic and unpredictable parts of the environment. During the search in the belief state space, the agent considers some variables fixed and concentrate only on the most important parts of the environment to choose its actions. This approach is possible with the RTBSS algorithm because it is an online algorithm that can readjust its belief states between each execution in order to stay up to date with the agent's observations.

Local reward function. We defined a local reward function enabling an agent using RTBSS to redefine a reward function before each action's choice. This enables to define the reward function only for the current situation, which is really useful when there are a lot of possible situations. This again is possible because the agent's policy is dynamically defined thus the reward function can be modified before the action's search.

Flexible coordination approach. Our local reward function can be used to dynamically coordinate many agents in an environment without any coordination related messages. This new multiagent POMDP coordination approach has shown to be effective and quite flexible to control many agents in a highly dynamic environment.

Learning required resources. Most coordination learning approaches consider that the number of required resources to accomplish a task is known or that they have enough information to have a probability distribution over the number of required resources. In our approach, we consider that the tasks are complex and that the agents have to learn this information since it is not available. We developed, to this end, an algorithm to learn the required number of resources for each task.

Reduction of the task description space. The tasks considered in our simulations are described with discrete and continuous attributes. Therefore, there are a lot of possible task descriptions. To manage this complexity, we have adapted a selective perception reinforcement learning algorithm to the problem of learning the required number of resources to accomplish a task. With this algorithm we can find a generalization of the task description space, thus allowing the reinforcement learning algorithm to work on smaller task description spaces.

Coordination algorithm. We proposed a coordination algorithm using the information learned about the number of resources needed for a task. This algorithm uses really few messages between the agents, which is interesting in environments with limited and/or unreliable communications.

Links between multiagent and scheduling systems. We show some possible links between multiagent systems and task scheduling systems. We present a design model defining in a structured way the main steps necessary to extract from a multiagent system the scheduling problem and mostly how to structure the solution to this scheduling problem.

Reduction of the communication. We show that a decentralized scheduling system can offer the same performances as a centralized one, while diminishing the amount of information transmitted between the agents. This is done in the objective of being more robust to constraints on the communications.

K-Nearest-Neighbors algorithm. We have presented a K-Nearest-Neighbors algorithm to estimate the value of an uncertain parameter of a task. We have presented results showing the efficiency of the predictions in the RoboCupRescue simulation. We have also presented a strategy to manage the missing attribute values by simply ignoring them when calculating the distances between the instances. We have presented results showing that ignoring the missing values is a more efficient approach than trying to estimate their values when many values are missing.

RoboCupRescue approaches. All the approaches presented in this thesis are also good contributions to the RoboCupRescue simulation community. These approaches can be reused and improved by other participants in the RoboCupRescue simulation competition. This can contribute to improve the approaches specifically developed for rescue operations.

1.6 Outline

This thesis is structured in 6 chapters, which are briefly presented in the following:

Chapter 1 , i.e. the current chapter, introduces the context of this research. It then briefly presents the approaches presented in this thesis and it emphasizes the contributions of this thesis.

Chapter 2 presents our test-bed environment: the RoboCupRescue simulation. The most important characteristics of this environment are presented, followed by a discussion on some of its challenges that make it a really interesting and hard testbed for multiagent algorithms. This environment is described at the beginning of this thesis to define the context in which our approaches have been developed.

Chapter 3 presents our online POMDP algorithm. In the first part of this chapter, we review the literature on POMDP algorithms. Then we describe the formalism of our online algorithm and some hybrid algorithms, followed by some results on standard POMDPs. Afterwards, we present the adaptation of our basic online POMDP algorithm to the RoboCupRescue environment and some results showing its efficiency in such an environment. In addition, we explain our local reward function and how it is used to coordinate the *PoliceForce* agents.

Chapter 4 proposes a reinforcement learning algorithm to learn the required number of agents (considered as resources) to accomplish a complex task. This chapter considers more complex tasks than Chapter 3. These tasks take more time to be accomplished and some of their characteristics are unknown, like the required number of resources. The coordination mechanism thus has to estimate the unknown characteristics before the agents could be coordinated. At the beginning of this chapter, we briefly review the literature on coordination learning algorithms. We then describe in detail our selective perception reinforcement learning algorithm. Finally, we present the results obtained by testing our algorithm in the RoboCupRescue simulation.

Chapter 5 considers tasks with varying deadlines that need to be schedule in order to maximize the number of tasks accomplished before the end of the simulation. The first part of this chapter presents different scheduling approaches. Afterwards, we describe how the *AmbulanceTeam* agents estimate the deadlines of the tasks using a new application of the K-Nearest-Neighbors learning algorithm. Finally, we present results showing the performances of this scheduling approach in the RoboCupRescue simulation environment.

Chapter 6 summarizes this thesis and presents some open problems for future work.

Chapter 2

RoboCupRescue

This chapter presents the RoboCupRescue simulation environment which has been used as a testbed for most of the algorithms presented in this thesis. The most important characteristics of this environment are presented, followed by a discussion on some of its challenges that make it a really interesting and hard testbed for multiagent algorithms.

The simulation project of the RoboCupRescue is one of the activities of the RoboCup Federation (RoboCup (2003)), which is an international organization, registered in Switzerland, to organize international effort to promote science and technology using soccer games and rescue operations by robots and software agents. RoboCup is an international joint project to promote artificial intelligence, robotics, and related fields. It is an attempt to foster artificial intelligence and intelligent robotics research by providing standard problems where wide range of technologies can be integrated and examined.

RoboCup chose to use soccer game as a central topic of research, aiming at innovations to be applied for socially significant problems and industries. The ultimate goal of the RoboCup project is by 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer. The first international RoboCupSoccer competition took place in 1997 in Nagoya, Japan. Since then, the activities of the RoboCup Federation have been diversified. In 2001, the RoboCup Federation initiated the RoboCupRescue project in order to specifically promote research in socially significant issues. Currently, the RoboCup Federation has 12 leagues regrouped in three major domains:

- RoboCupSoccer
  - Simulation League
  - Small Size Robot League (f-180)
  - Middle Size Robot League (f-2000)

  - Four-Legged Robot League
  - Humanoid League
  - E-League
  - RoboCup Commentator Exhibition
- RoboCupRescue
  - Rescue Simulation League
  - Rescue Robot League
- RoboCupJunior
  - Soccer Challenge
  - Dance Challenge
  - Rescue Challenge

As mentioned before, our work has been done in the RoboCupRescue Simulation League. The RoboCupRescue simulation environment consists of a simulation of an earthquake happening in a city (Kitano et al. (1999); Kitano (2000)). The goal of the agents (representing firefighters, policemen and ambulance teams) is to minimize the damages caused by a big earthquake, such as buried civilians, buildings on fire and blocked roads. Figure 2.4 on page 18 shows an illustration of a RoboCupRescue simulation.

In the next sections, we first describe the RoboCupRescue simulator, then we present the different agents that have to be implemented and finally we explain why this environment is an interesting testbed environment for multiagent algorithms.

2.1 RoboCupRescue Simulator

In this section, the RoboCupRescue simulator is presented. First, we present all the modules constituting the simulator. Then we present how the time is managed and how the different modules interact between each other during the simulation.

2.1.1 RoboCupRescue Simulator Modules

The RoboCupRescue simulation is composed of a number of modules that communicate between each other using the TCP protocol. These modules consist of: the kernel, the rescue agents (FireBrigade, PoliceForce, etc.), the civilian agents, the simulators

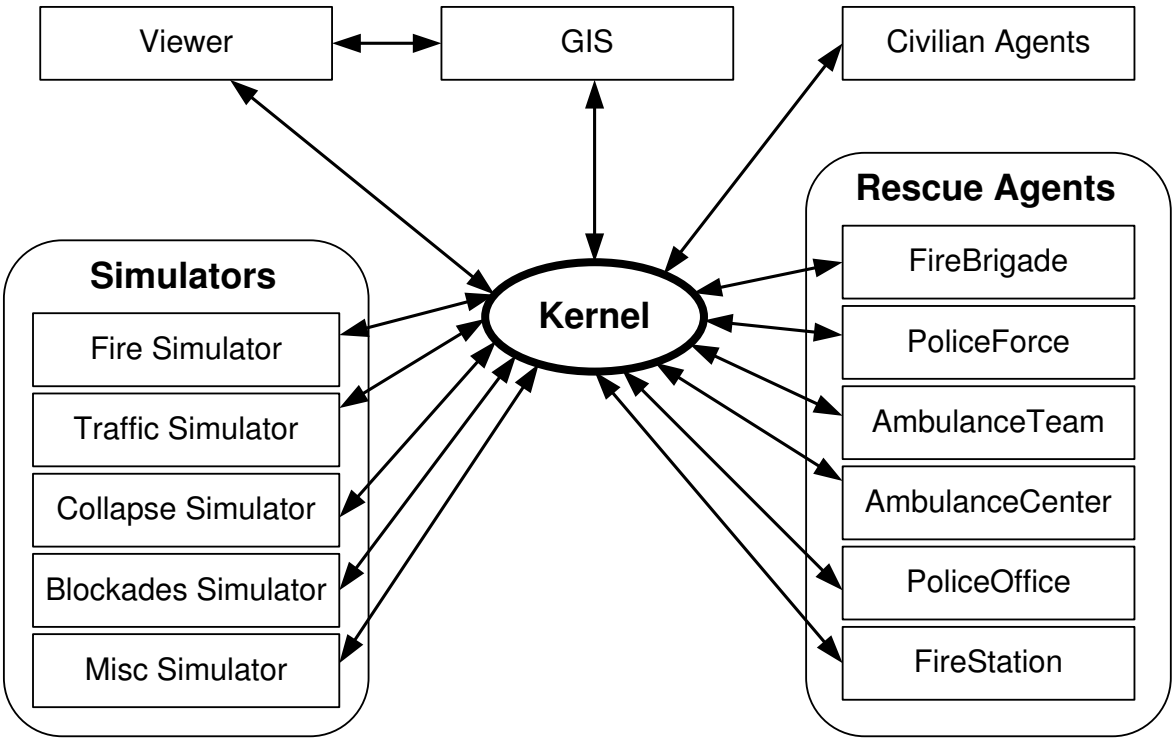


Figure 2.1: RoboCupRescue simulator architecture.

(FireSimulator, TrafficSimulator, etc.), the GIS (Geographical Information System) and the viewers. All these modules can be distributed on different computers. Figure 2.1 presents the relations between these modules. More precisely, these modules can be described as:

Kernel: The kernel is at the heart of the RoboCupRescue simulator. It controls the simulation process and it manages the communications between the modules. For example, all messages between the agents have to go trough the kernel. There are no direct communications between the agents. When the kernel receives the messages, it verifies them to make sure that they respect some predefined rules. Then it sends the valid messages to their intended recipients. The kernel is also responsible for the integration of all the simulation results returned by the simulators. Moreover, the kernel controls the time steps of the simulation and it manages the synchronization between the modules.

GIS: The GIS module provides the initial configuration of the world at the beginning of the simulation, i.e. locations of roads, buildings and agents. During the simulation, this module is responsible for providing all the geographical information to the simulators and the viewers. It also records the simulation logs, so that the simulation can be analyzed afterwards.

Simulators: The simulator modules are responsible for the dynamic aspects of the world. They manage all dynamic parts of the world, including the effects of the agents' actions. To achieve that, they manipulate the environment information provided by the GIS module. There are five simulators:

Fire simulator: The fire simulator controls the fire propagation, which depends on the buildings' compositions, the wind, the amount of water thrown by the *FireBrigade* agents and the distance between the buildings. This is the most important simulator module because the fires have a huge impact on the simulation.

Traffic simulator: The traffic simulator is in charge of simulating the agents' movements in the city. It has to deal with blockades and traffic jam.

Collapse simulator: The collapse simulator simulates the impact of the earthquake on the buildings. It controls how badly a building is damaged by the earthquake and how deeply the agents are trapped in the buildings.

Blockades simulator: The blockade simulator simulates the impact of the earthquake on the roads. It generates all the blockades on the roads.

Misc simulator: The misc simulator simulates the agents' injuries and the agents' actions: load, unload, rescue and clear.

Civilian Agents: This module controls the civilian agents, which have to be rescued by the rescue agents. These agents have really simple behaviors. They scream for help if they are trapped in buildings or they simply move around in the city, trying to reach a refuge.

Viewer: The viewer module graphically presents the information about the world provided by the GIS module. It is possible to have more than one viewer connected to the kernel at the same time. There are 2D and 3D viewers available.

Rescue agents: These modules control the rescue agents. For the competition, participants have to develop these modules. There are six different modules or type of agents that have to be developed:

FireBrigade: There are between 0 to 15 agents of this type that have to extinguish fires.

PoliceForce: There are between 0 to 15 agents of this type that have to clear the roads.

AmbulanceTeam: There are between 0 to 8 agents of this type that have to rescue agents or civilians that are trapped in collapse buildings.

FireStation: This is the *FireBrigade's* control station which is responsible for the communications between the *FireBrigade* agents and the other type of agents.

PoliceOffice: This is the *PoliceForce's* control station which is responsible for the communications between the *PoliceForce* agents and the other type of agents.

AmbulanceCenter: This is the *AmbulanceTeam's* control station which is responsible for the communications between the *AmbulanceTeam* agents and the other type of agents.

2.1.2 Time Management

The RoboCupRescue system simulates 5 hours after the earthquake has happened. It is a discrete simulation in which each time step corresponds to one minute. Therefore, the simulation is executed in 300 time steps. It is the kernel that is responsible for managing the time of the simulation. To achieve that, it should impose a real-time constraint for all the modules; the kernel does not wait for the modules' responses. If the kernel receives an information from a module too late, this information is discarded. This is an important constraint that the participants have to keep in mind, because it limits the time allowed for an agent to reason about its next action. Therefore, all the algorithms developed for the agents have to be fast and efficient. In the current settings of the environment, the complete loop of the kernel takes two seconds, which leaves approximately one second for all the modules to compute their actions.

2.1.3 Simulation's Progress

The first step of a simulation consists for all the modules to connect to the kernel. Figure 2.2 illustrates the initialization process. At the beginning of the connection process, the GIS module sends the initial configuration of the world to the kernel. The kernel then forwards this information to the simulator modules and it sends to the rescue agents only their perceptual information. At the same time, the viewer ask the GIS for the graphical information about the world. Afterwards, the simulation starts. As mentioned before, the simulation is composed of 300 cycles and each one of them is composed of the following steps (see Figure 2.3):

1. At the beginning of every cycle, the kernel sends to the rescue agents all their sensors information (visual and auditive). The visual information of an agent contains all the objects that are in a 10 meters radius around the agent. The auditive

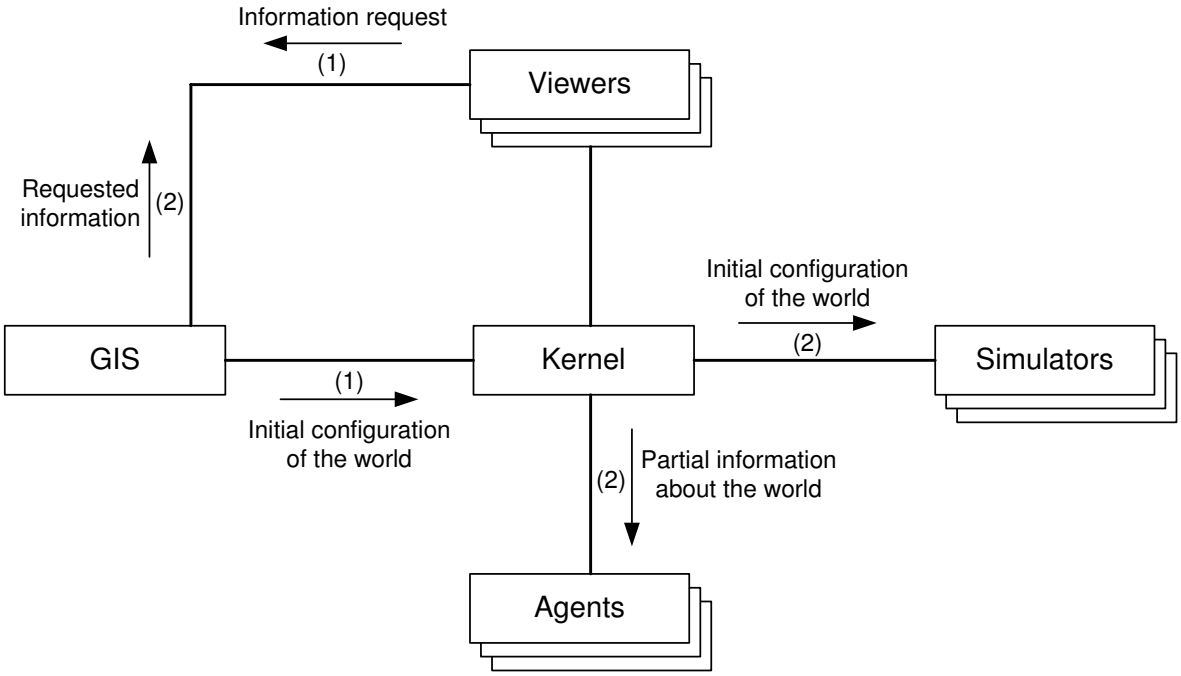


Figure 2.2: Communication between the simulator modules during the initialization phase.

information contains all voice messages and radio messages. The communication between the agents is explained in section 2.2.

2. Each agent module then uses the sensors information received to decide which actions it should do. Notice that the actions include the physical actions and the communication actions.

3. The kernel gathers all the actions received from the agents and it sends them to the simulator modules. The actions received are sometimes filtered by the kernel. For example, if the kernel receives an action from a dead agent, the action would not be considered. Also, since the simulation proceeds in real-time, the kernel ignores all the actions that do not arrive in time. Only accepted actions are sent to the simulator modules.

4. The simulator modules individually compute how the world will change based upon its internal state and the actions received from the kernel. Then, each simulator modules sends its simulation results to the kernel.

5. The kernel integrates the results received from the simulator modules and it sends them to the GIS module and to the simulator modules. The kernel only integrates the results that are received on time.

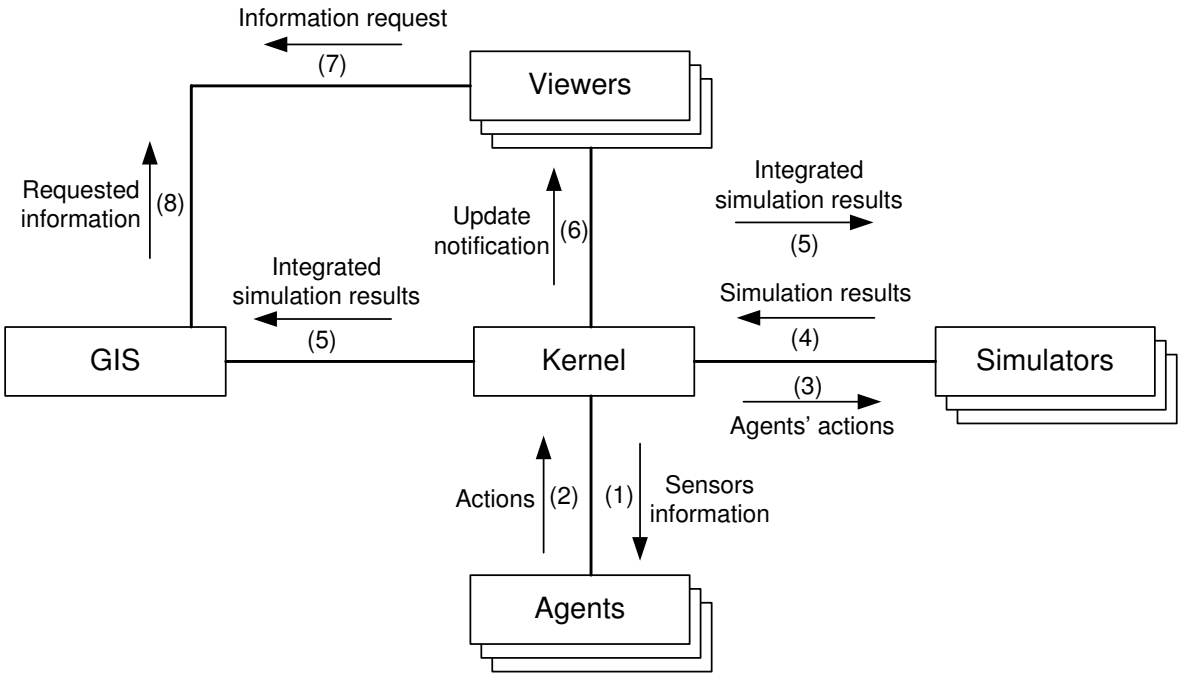


Figure 2.3: Communication between the simulator modules during one cycle of the simulation.

6. The kernel increases the simulation clock and it notifies the viewers about the update.

7. The viewers request the GIS to send the updated information of the world.

8. The GIS keeps track of the simulation results and it sends to the viewers the information they requested. Finally, the viewers visually display the information received from the GIS.

2.1.4 Evaluation Function

In the RoboCupRescue simulation, the performance of the rescue agents is evaluated by considering the number of agents that are still alive, the healthiness of the survivors and the unburned area. As we can see in the following equation, the most important aspect is the number of survivors. Therefore, agents should try to prioritize the task of rescuing civilians.

$$Score = \left(nA + \frac{H}{H_{ini}} \right) \sqrt{\frac{B}{B_{ini}}} \tag{2.1}$$

where nA is the number of living agents, H is the remaining number of health points (HP) of all agents, H_{ini} is the total number of HP of all agents at the beginning, B_{ini} is

Fierceness	Meaning
0	Intact building.
1	Small fire.
2	Medium fire.
3	Huge fire.
4	Not on fire, but damage by the water.
5	Extinguished, but slightly damage.
6	Extinguished, but moderately damage.
7	Extinguished, but severely damage.
8	Completely burned down.

Table 2.1: Meaning of the building's fierceness attribute values.

Fierceness	Score rules
0	No penalty.
1 or 5	$\frac{1}{3}$ of the building's area is considered destroyed.
4	Water damage, also $\frac{1}{3}$ of the building's area is considered destroyed.
2 or 6	$\frac{2}{3}$ of the building's area is considered destroyed.
3, 7 or 8	The whole building is considered destroyed.

Table 2.2: Score rules used to evaluate the area burned based on the building's fierceness attribute.

total buildings' area at the beginning and B is the undestroyed area which is calculated using the fierceness value of all buildings. The fierceness attribute indicate the intensity of the fire and how badly the building has been damaged by the fire. This attribute can take values from 0 to 8, as presented in Table 2.1. Using these fierceness values, Table 2.2 presents the rules used to evaluate the unburned area of each building.

2.1.5 Graphical Representation of the Simulation

In order to see the evolving simulation, the RoboCupRescue simulator has a viewer module responsible for the graphical representation of the simulation. Figure 2.4 presents an example of a RoboCupRescue situation. Buildings are represented as polygons. Gray polygons means that the buildings are not on fire. If the building is on fire, then it is yellow, or orange. If the building was on fire and then extinguished, it is blue. Green buildings represent refuges where the injured agents have to be sent. White buildings represent the three center agents (*FireStation*, *PoliceOffice* and *AmbulanceTeam*). The darker a building is, the more damage it is.

Figure 2.4: Example of a RoboCupRescue situation.

Agents are represented as circles: FireBrigade (red), PoliceForce (yellow), AmbulanceTeam (white) and Civilians (green). Again, the darker an agent is, the more injured it is. The blue lines represent the water thrown by the *FireBrigade* agents on the fires.

A little "x" on a road means that this road is blocked. When a *PoliceForce* agent clears a road, the "x" disappears.

2.2 Rescue Agents

The objective of the RoboCupRescue simulation project is to study rescue strategies, and also collaboration and coordination strategies between rescue teams (Takahashi et al. (2002)). Participants in the RoboCupRescue championship have to develop software agents representing teams of firefighters, polices and paramedics, in order to

manage the disaster the best way they can. These agents have to:

- determine where are the emergencies with the highest priorities,
- choose which roads to clear so that strategic places can be reached,
- choose where to dig in order to rescue the most civilians,
- carry injured civilians to the refuges,
- choose which fires to extinguish in priority,
- etc.

For the current testbed, there are approximately 100 agents representing groups of people (civilian families, firefighter teams, police forces, ambulance teams). This grouping has been done to simplify the simulation. However, the objective of the RoboCupRescue committee is to have more than 10 000 agents in the simulation to make it more realistic (Tadkoro et al. (2000)). The number of agents will be increased when the computer hardware will support that many deliberative agents in one simulation.

In the simulation, agents can accomplish different actions that can be divided in two classes (Koch (2002)): actions shared by all agents and actions specialized and available to only some types of agents.

- *Shared actions*:
  - Move (except for building agents);
  - Speak to near agents;
  - Communicate by radio with all the agents of the same type and their center agent;
  - Do nothing.
- *Specialized actions*:
  - *FireBrigade* agents can extinguish fires;
  - *PoliceForce* agents can clear roads;
  - *AmbulanceTeam* agents can dig to rescue civilians and they can transport other agents (civilians or rescue agents);
  - Center agents (*FireStation*, *PoliceOffice* and *AmbulanceCenter*) can communicate with the other center agents.

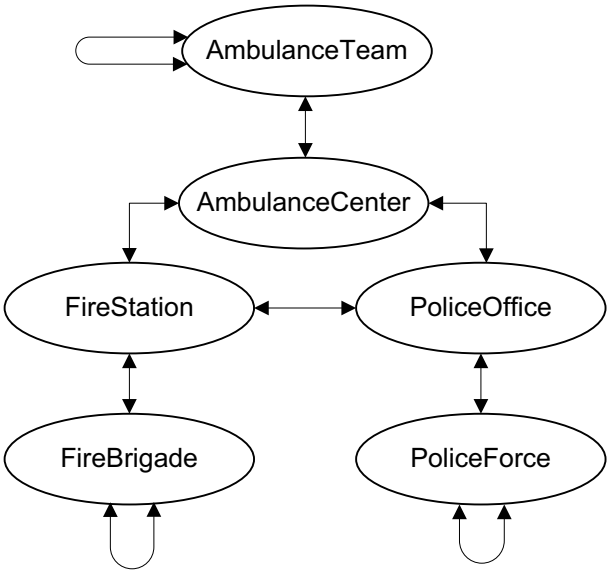


Figure 2.5: Communication organization. Links between different types of agents indicate that a message can be sent by radio between these two types of agents.

The coordination and the collaboration between the agents are really important, because the agents' efficiency can be improved if the agents collaborate with each other. The firefighter agents, the police agents and the paramedic agents work faster if they work in teams. For example, if there are many *FireBrigade* agents that cooperate to extinguish the same fire, then the fire will be extinguished mush faster than if only one agent tries to extinguish it (Ohta et al. (2001)).

There are two different communication actions: *Say* and *Tell*. With the *Say* action, an agent can speak to all agents in a 30 meters radius around it. With the *Tell* action, agents communicate by radio. All radio messages are broadcasted to the other agents following the communication organization presented on Figure 2.5. For example, if a *FireBrigade* agent sends a message by radio, it will be received at the next time step by all the other *FireBrigade* agents and by the *FireStation* agent.

One should note that this communication organization limits the ability to communication between heterogeneous agents. For example, a *FireBrigade* agent cannot directly send a message to a *PoliceForce* agent. The message has to go from the *FireBrigade* agent to the *FireStation* agent, to the *PoliceOffice* agent and finally to the *PoliceForce* agent. As we can see, it needs at least three time steps for a message to go from a *FireBrigade* agent to a *PoliceForce* agent. This communication constraint is only one of the many constraints imposed by the RoboCupRescue simulation environment. In the next section, we present why it is such a complex problem.

2.3 Environment Complexity

The RoboCupRescue simulation is a complex environment that imposes many constraints like:

- A real-time constraint on the agents' response time. All agents have to return their action in less than a second after they received their perceptions.
- The agents' perceptions are limited to a 10 meters radius.
- The length and the number of messages that an agent can send or received are limited.
- The FireBrigade agents have a limited amount of water available.
- The civilians die if they are not saved on time.
- The time left before a civilian dies is unknown.
- The fires are spreading fast if they are not controlled rapidly.
- Rescue agents can easily create traffic jam.

One of the most important problems in the RoboCupRescue simulation is the partial observability of the environment. In the simulation, agents have only a local perception of their surroundings. Agents only perceive the objects that are in a 10 meters radius around them. Consequently, there is no agent that has a complete view of the environment state. Even more than that, the RoboCupRescue simulation is a collectively partially observable environment (Nair et al. (2003)). This means that even if all the agents' perceptions are regrouped, these agents would not have a perfect vision of the situation.

This uncertainty complicates the problem greatly. Agents have to explore the environment, they cannot just work on the visible problems. Therefore, one major problem for the agents is to acquire useful information in a reasonable time (Kitano et al. (1999)). Agents also have to communicate with each other to improve their local perceptions, even though they will never have a perfect knowledge about the environment. Communications are quite restricted, but they are still really important, because the coordination between the agents depends a lot on the efficiency of the communications between them.

As mentioned before, agents have to communicate to compensate for their restrictive local perceptions. However, agents have to be really careful about the messages they send, because it is really easy to loose a message due to the limitations on the number of

Agent's type	**Receive**	**Send**
Mobile agents	4	4
Center agents	$2n$	$2n$

Table 2.3: Maximal number of messages per time step that an agent can send or receive. n is the number of mobile agents of the same type as the center agent. The center agents are: *FireStation*, *PoliceOffice* and *AmbulanceCenter*. And the mobile agents are: *FireBrigade*, *PoliceForce* and *AmbulanceTeam*.

messages that can be sent or received and because of the communication organization presented in Figure 2.5. The maximum number of messages that can be sent or received during one time step of the simulation are presented in Table 2.3. As we can see, center agents have better communication capabilities because they can receive and send more messages than the mobile agents. Each center agent can receive and send $2n$ messages per time step where n is the number of mobile agents of the same type as the center agent. For example, if there are 10 *FireBrigade* agents, then the *FireStation* agent can send and receive 20 messages per time step. Since center agents can receive more messages, they normally have a better knowledge of the global situation. Therefore, center agents are the best agents to serve as the center of coordination for the mobile agents of the same type.

With such communication constraints, there is a good chance that a message gets lost and that it does not reach its intended recipient. For example, consider the case where 10 FireBrigade agents each sends one message during one time step. This is really under the limitation of the agents, because they could each send 4 messages in one time step. However, even with only one message sent per agent, each agent will receive 9 messages, which is more than twice the number of messages an agent can receive in one time step. Consequently, each agent will lost 5 messages. The situation can be much worst if the agents have more than one message to send or if there are messages coming from other types of agents trough the center agent. It then becomes really important for the agents to have a good strategy to choose which messages they should send or listen to.

Moreover, the communications in the RoboCupRescue simulation are situated communications (Noda (2001)), which means that the information contained in a message depends a lot on the position of this information on the map. For example, an information about a fire is useless if the agent does not transmit the position of the fire. For the communication between the agents, the complexity happens when agents have to choose which are the most important messages to listen to. For example, a message coming from a near agent has more chance to be useful than a message coming from a far agent, because normally we need more coordination messages for agents working

on the same problem. Consequently, to efficiently choose which messages to listen to, each agent has to estimate the position of the other agents in the city. This could be quite hard since agents are always moving.

Another difficulty of the RoboCupRescue environment is that agents are heterogeneous. They have different capabilities and there is no agent that can do everything by itself. Consequently, agents have to collaborate with each other if they want to accomplish their tasks efficiently (Paquet et al. (2004a)). Agents have to coordinate their actions in order to profit from each other's capabilities.

In the simulation, it is also really important to efficiently manage the resources, because there is a lot of work to do with few resources. Logistic and more particularly distributed logistic become then a complex problem. There are a lot of problematic situations in the simulated city and the agents have to be assigned to the problems that will maximize their actions' results.

2.4 Multiagent Testbed

The RoboCupRescue simulation environment is a good testbed for multiagent algorithms, because it has some really interesting characteristics for research in this domain. Here are some of its advantages as a testbed environment:

- The environment is complex enough to be realistic.
- The testbed is easily accessible.
- The testbed covers a lot of different multiagent problems.
- The testbed enables to compare the approaches developed with the other participants at the competition.

The RoboCupRescue simulation environment offers a complex testbed allowing many multiagent research opportunities or more generally many artificial intelligence research opportunities (Kitano et al. (1999)). These opportunities are present in domains like:

Multiagent planning. There are many heterogeneous agents that have to plan and act in an hostile and dynamic environment.

Anytime and real-time planning. Agents have to plan while following some real-time constraints.

Robust planning. Planning has to be done with incomplete information. The planning system has to be able to efficiently replan if some information changes.

Resources management. Resources are really limited in the simulation, thus it becomes important to manage them efficiently.

Learning. Tasks are also quite complicated, thus agents have to learn how to assign the resources to the different tasks. They also have to learn some dynamic aspects of the environment in order to estimate its evolution.

Information gathering. Agents have to explicitly plan for information gathering actions in order to improve the agents' global vision of the environment.

Coordination. Agents have to coordinate their actions because more than one agent is usually needed to accomplish the tasks.

Decision-making in large scale systems. Agents have to analyze many possibilities and choose an action to accomplish in a really huge partially observable state space.

Scheduling. There are many civilians that have to be rescued and each of them has a different estimated death time and a different rescue time. These dynamic tasks have to be schedule in order to maximize the number of civilians rescued.

In the next chapters, we present some ideas and some algorithms that we have developed in order to tackle some of these challenges.

Chapter 3

Online POMDP Algorithm

As described in Chapter 2, the RoboCupRescue simulation environment is partially observable. Agents evolving in this environment have to choose an action at each time step even though they only have an incomplete representation of their environment. When faced with a partially observable environment like the RoboCupRescue simulation, a general model for sequential decision problems is to use the Partially Observable Markov Decision Processes (POMDPs).

A lot of problems can be modelled with POMDPs, but very few can be solved because of their computational complexity (POMDPs are PSPACE-complete (Papadimitriou and Tsitsiklis (1987))). The main problem with POMDPs is that their complexity makes them applicable only on small environments. However, most problems of interest have a huge state space, which motivates the search for approximation methods (Hauskrecht (2000)). This is especially the case for multiagent systems where there is often a huge state space with autonomous agents interacting with each other.

POMDPs have generated a lot of interest in the AI community and many approximation algorithms have been developed recently (Pineau et al. (2003); Braziunas and Boutilier (2004); Poupart (2005); Smith and Simmons (2005); Spaan and Vlassis (2005)). They all share in common the fact that they solve the problem offline. This means that they specify, prior to the execution, the action to execute for all possible situations the agent could encounter in the environment. This plan, linking an environment state with the chosen action, is called a *policy*. While these approximation algorithms can achieve very good performances, they are still not applicable on large problems, where there are too many possible situations to completely solve the problem offline.

In this chapter, instead of computing a complete policy offline, we present an online approach based on a look-ahead search in the belief state space to find the best action to execute at each cycle in the environment (Paquet et al. (2005b,c)). Our algorithm,

called RTBSS (Real-Time Belief Space Search), only explores reachable belief states starting from the agent's current belief state. This online exploration has to be as fast as possible, since our algorithm has to work under some real-time constraints in the RoboCupRescue simulation. To achieve that, we opted for a factored POMDP representation and a branch and bound strategy. By pruning some branches of the search tree, our algorithm is able to search deeper, while still respecting the real-time constraints.

By doing an online search, we avoid the overwhelming complexity of computing a policy for every possible situation the agent could encounter. Since there is no computation offline, the algorithm is immediately applicable to previously unseen environments, if the environments' dynamics are known. Other approaches have used an online search for POMDPs, but they were not immediately efficient without any offline computations. For example, the BI-POMDP algorithm needs the underlying MDP to be solved offline to choose in which order to expand the search (Washington (1997)). Similarly, the RTDP-BEL algorithm (Geffner and Bonet (1998)) needs successive trials in the environment in addition to the solution for the underlying MDP, thus it needs an offline training before becoming efficient.

In the first part of this chapter, we present a more detailed literature review on POMDP algorithms. Then we describe the formalism of our online algorithm and some hybrid algorithms, followed by some results on standard POMDPs and finally we present an adaptation of our method for a complex multiagent environment and some results showing its efficiency in such environments. The results show that it is possible to achieve relatively good performances by using a very short amount of time online. The tradeoff between the solution quality and the computing time is very interesting.

3.1 Literature Review

In this section, we present a literature review covering many methods for solving partially observable Markov decision problems. Firstly, we present the POMDP model, followed by an example of the optimal value iteration algorithm. Afterwards, we present offline approximation methods, which construct a complete policy offline, before the agent has to be effective in the environment. Then we present online methods, which need to be executed in the environment in order to define the agent's policy. Finally, we present how the POMDP model can be factorized by representing it with random variables and how this can help to find approximated solutions.

3.1.1 POMDP Model

Partially Observable Markov Decision Processes (POMDPs) provide a general framework for acting in partially observable environments (Astrom (1965); Smallwood and Sondik (1973); Kaelbling et al. (1998)). A POMDP is a model for planning under uncertainty, which gives the agent the ability to effectively estimate the outcome of its actions even though it cannot exactly observe the environment. Formally, a POMDP is represented as a tuple $\langle S, A, T, R, \gamma, \Omega, O \rangle$ where:

- S is the set of all the environment states. A state is a description of the environment at a specific moment and it should capture all information relevant to the agent's decision-making process.
- A is the set of all possible actions.
- $T(s, a, s')$ is the transition function, which gives the probability of ending in state s' if the agent performs action a in state s, $Pr(s'|s, a)$.
- $R(s, a)$ is the reward function which gives the reward associated with doing action a in state s.
- γ is the discount factor $(0 < \gamma \leq 1)$.
- Ω is the set of all possible observations.
- $O(s', a, o)$ is the observation function which gives the probability of observing o if action a is performed and the resulting state is s', $Pr(o|a, s')$.

In a POMDP, the states are not directly observable. At any given time, the agent only has access to some observation $o \in \Omega$ that give some incomplete information about the current state. Since the states are not observable, the agent cannot choose its actions based on the states. It rather has to consider a complete history of its past actions and observations to choose its current action. The history at time t is defined as:

$$h_t := \{a_0, o_1, \ldots, o_{t-1}, a_{t-1}, o_t\} \tag{3.1}$$

This explicit representation of the past is really expensive in memory. To reduce the length of the past actions and observations considered, some researchers have worked on approaches to learn a more compact representation of the past (McCallum (1996); Dutech and Samuelides (2003)). Instead of memorizing all actions and observations, they learn smaller histories for different situations.

However, it is not necessary to explicitly represent histories, because it is possible to summarize all relevant information from previous actions and observations in a probability distribution over the state space S, which is called a belief state (Astrom (1965)). A belief state at time t is defined as the posterior probability distribution that gives the probability of being in each state knowing the complete history:

$$b_t(s) := Pr(s_t = s|h_t) \tag{3.2}$$

It has been shown that the belief state b_t is a sufficient statistic for the history h_t (Smallwood and Sondik (1973)), therefore the agent can choose its actions based on the current belief state b_t instead of all past actions and observations. Furthermore, the belief state b_t can be computed from the previous belief state b_{t-1}, the previous action a_{t-1} and the current observation o_t. This is done with the belief state update function $\tau(b, a, o)$: if $b_t = \tau(b_{t-1}, a_{t-1}, o_t)$, then

$$b_t(s') = \eta O(s', a_{t-1}, o_t) \sum_{s \in S} T(s, a_{t-1}, s') b_{t-1}(s) \tag{3.3}$$

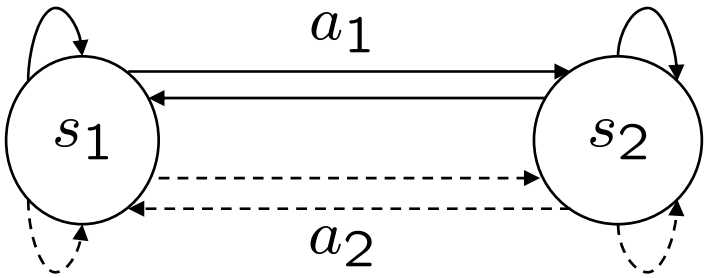
where η is a normalizing constant.

Now that the agent has a way to estimate in which state it is, it needs to choose an action based on its belief state. This action is determined by the agent's policy π, which is a function that maps a belief state to the action the agent should execute in this belief state. Therefore, it defines the agent's strategy for all the possible situations it could be faced with. This strategy should maximize the amount of reward earned over a finite or infinite time horizon. More precisely, the optimal policy π^* is the policy that maximizes the expected sum of discounted rewards:

$$\pi^* = \underset{\pi \in \Gamma}{\text{argmax}}\ E\left[\sum_{t=0}^{\infty} \gamma^t \sum_{s \in S} b_t(s) R(s, \pi(b_t))\right] \tag{3.4}$$

Where γ is the discount factor ($0 < \gamma \leq 1$), $b_t(s)$ is the probability that the agent is in state s according to the belief state b_t and $\pi(b_t)$ is the action prescribed by the policy π in the belief state b_t.

Graphically, a policy can be represented as a tree structure. For example, Figure 3.2 shows policies for different horizons for a problem with two actions and two observations which is described in Figure 3.1. At the horizon of 1, the agent can only do one action, therefore there are only two possible policies: doing action a_1 or doing action a_2. From the horizon 2, the agent has to consider the possible observations. The policy has to contain actions for all possible observations. For example, the policy P_{121} specifies that the agent executes action a_1, then if it observes o_1, it does policy P_2 and if it observes



States: $\{s_1, s_2\}$
Actions: $\{a_1, a_2\}$
Observations: $\{o_1, o_2\}$
Discount factor: $\gamma = 0.9$

Reward Function

- $R(s_1, a_1) = 2$
- $R(s_1, a_2) = 1$
- $R(s_2, a_1) = 1$
- $R(s_2, a_2) = 3$

Transition Function

- $T(s_1, a_1, s_1) = 0.3$
- $T(s_1, a_1, s_2) = 0.7$
- $T(s_1, a_2, s_1) = 0.1$
- $T(s_1, a_2, s_2) = 0.9$
- $T(s_2, a_1, s_1) = 0.6$
- $T(s_2, a_1, s_2) = 0.4$
- $T(s_2, a_2, s_1) = 0.8$
- $T(s_2, a_2, s_2) = 0.2$

Observation Function

- $O(s_1, a_1, o_1) = 0.9$
- $O(s_1, a_1, o_2) = 0.1$
- $O(s_1, a_2, o_1) = 0.9$
- $O(s_1, a_2, o_2) = 0.1$
- $O(s_2, a_1, o_1) = 0.5$
- $O(s_2, a_1, o_2) = 0.5$
- $O(s_2, a_2, o_1) = 0.5$
- $O(s_2, a_2, o_2) = 0.5$

Figure 3.1: POMDP Example.

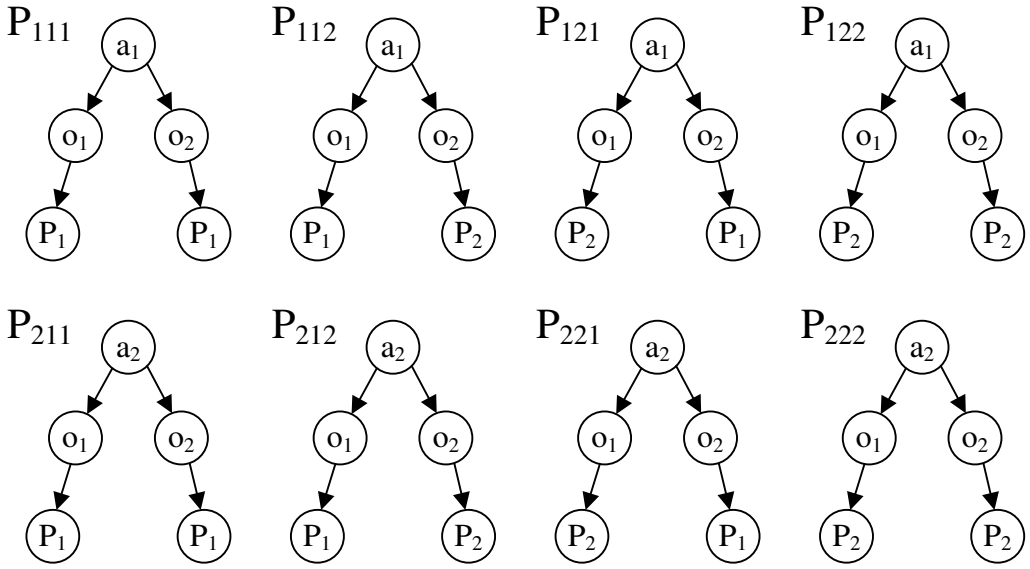
o_2, it does policy P_1. For our example problem at a horizon of 2, there are 8 possible policies, illustrated in Figure 3.2. For a horizon of 3, there are 128 possible policies and Figure 3.2 presents three examples of these. Table 3.1 presents the number of possible policies for horizons of 1 trough 5. As we can see, the number of possible policies grows rapidly. In fact, the number of policies $|\Gamma_t|$ for a horizon of t is given by:

$$|\Gamma_t| = |A|^{\frac{|\Omega|^t - 1}{|\Omega| - 1}} \tag{3.5}$$

This last equation shows that the number of states does not have an impact on the number of possible policies. However, it has a big impact on the time needed to evaluate each policy, which is in $O(|S|^2)$.

For a horizon of 1, there are 2 possible policies (P_1 and P_2):

For a horizon of 2, there are 8 possible policies:



For a horizon of 3, there are 128 possible policies:

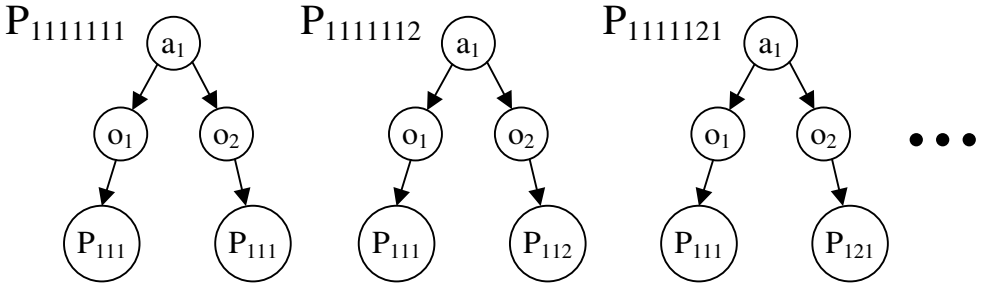


Figure 3.2: Tree representation of some policies at three different horizons.

Horizon	Number of Policies
1	2
2	8
3	128
4	32 768
5	2 147 483 648

Table 3.1: Number of policies for a problem with two actions and two observations.

3.1.2 Optimal Value Iteration Algorithm

There are many optimal POMDP algorithms: the Enumeration algorithm (Sondik (1971)), the algorithm of Monahan (1982), the One-Pass algorithm (Smallwood and Sondik (1973)), the linear support algorithm (Cheng (1988)), the Witness algorithm (Littman (1996)), the Incremental Pruning algorithm (Zhang and Liu (1996); Cassandra et al. (1997)), the algorithm of Zhang and Zhang (2001), etc. All these algorithms exactly solve POMDPs for a fixed horizon. Some of them are faster than the others, but all of them are still completely impractical for problems with more than a few dozen states.

In this section, we present the Enumeration algorithm (Sondik (1971)) which is a simple exact value iteration algorithm that can find an optimal policy for a specified horizon. This algorithm uses dynamic programming to compute increasingly more accurate values for each belief state b. The basic idea behind the value iteration algorithm is to construct policies gradually, one horizon at a time, and to reuse the calculation made for the preceding horizon. For example, in Figure 3.2, we can see that the policies at the horizon of 3 are reusing the policies created at the horizon of 2.

To construct those policies, the value iteration algorithm needs to evaluate the value of a belief state. Formally, let V be a value function that takes a belief state as parameter and returns a numerical value in $\mathbb{R}$ of this belief state. The initial value function is:

$$V_0(b) = \max_{a \in A} \sum_{s \in S} b(s) R(s, a) \tag{3.6}$$

Afterwards, the value function at the horizon of t is constructed from the value function at the horizon $t-1$ by using the following recursive equation:

$$V_t(b) = \max_{a \in A} \left[\sum_{s \in S} b(s) R(s, a) + \gamma \sum_{o \in \Omega} Pr(o|b, a) V_{t-1}(\tau(b, a, o)) \right] \tag{3.7}$$

Where $\tau(b, a, o)$ is the belief update function defined in Equation 3.3 and $Pr(o|b, a)$ is the probability of observing o if action a is performed in belief state b, which is defined

as (for a detailed proof see Littman (1994b)):

$$Pr(o|b,a) = \sum_{s' \in S} O(s',a,o) \sum_{s \in S} T(s,a,s')b(s) \tag{3.8}$$

The value function in Equation 3.7 returns the maximum expected sum of discounted rewards that the agent can receive in the next t time steps, for any belief state b. Therefore, the optimal policy for a horizon of t is simply to choose the action that maximizes $V_t(b)$:

$$\pi_t^*(b) = \underset{a \in A}{\text{argmax}} \left[\sum_{s \in S} b(s)R(s,a) + \gamma \sum_{o \in \Omega} Pr(o|b,a)V_{t-1}(\tau(b,a,o)) \right] \tag{3.9}$$

This last equation associates an action to a specific belief state, which has to be done for all possible belief states in order to define a complete policy. The problem is that there is an infinite and uncountable number of belief states. It is thus impossible to calculate a policy for all individual belief states. Consequently, the idea is to define the value of a specific policy over the entire belief state space. To do so, we cannot simply calculate a numerical value for a policy, since it is not the same for all belief states. We thus have to find a function for each policy that takes a belief state as parameter and returns the value of the policy for this belief state.

3.1.2.1 α-vectors

A value function for a policy is represented as a linear function with a term for each state. The variable for each term is the probability of being in the corresponding state and the coefficient is the expected reward of being in this state. Figure 3.3 presents an example of such a function for the policy P_1 of our POMDP example (Figure 3.1 and Figure 3.2). The value function for the policy P_1 is: $V(b) = 2b(s_1) + 1b(s_2)$. This means that the expected reward of the policy P_1 if the agent is in state s_1 is 2, and if the agent is in state s_2 it is 1. This function can be represented more compactly with a vector containing only the coefficients for all states: $\alpha = [2, 1]$. There is one α-vector for each policy and each α-vector has $|S|$ terms, i.e. as many coefficients as there are states. Thus, an α-vector represents an $|S|$-dimensional hyperplane that defines the expected reward of a policy.

A key result by Smallwood and Sondik (1973) shows that the optimal value function for a finite-horizon POMDP is piecewise-linear and convex. Therefore, the value function $V_t(b)$ at any horizon t can be represented by a set of α-vectors: $\Gamma_t = \{\alpha_0, \alpha_1, \ldots, \alpha_m\}$. With this representation, the value of a belief state is the maximum value returned by one of the α-vectors for this belief state and the best policy is the one associated with

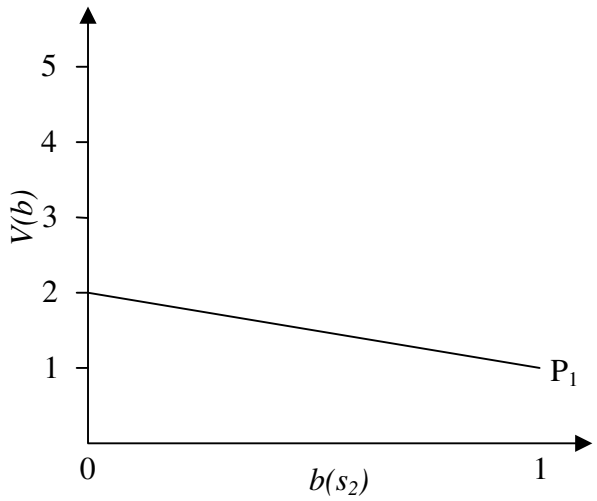


Figure 3.3: Example of a value function of a policy.

the α-vector that returned the best value.

$$V_t(b) = \max_{\alpha \in \Gamma_t} \sum_{s \in S} \alpha(s) b(s) \tag{3.10}$$

Algorithm 3.1 presents the exact value iteration algorithm showing how the α-vectors are created. The first step is to create one set $\Gamma_t^{a,*}$ for each action. Each of these set contains only one α-vector representing the immediate reward if the agent execute action a (lines 2-5). Then, there is an initialization step that corresponds to the horizon $t = 0$. Since the agent cannot do any actions, the value function equals zero for all belief states (lines 6 and 7). For all succeeding horizons t, we iterate over all actions to generate the Γ_t^a sets. To do so, for each observation, we create a new α-vector for each α-vector in the Γ_{t-1} set (lines 12 and 13). We thus have one $\Gamma_t^{a,o}$ set for each observation and each $\Gamma_t^{a,o}$ set has $|\Gamma_{t-1}|$ α-vectors. Afterwards, we calculate the set Γ_t^a by doing the cross-sum[1] over all the α-vector sets for the current action a (line 16). When we have all the Γ_t^a sets for all the actions, we can calculate the Γ_t set by doing the union over all the Γ_t^a sets (line 18).

The Γ_t set contains all the α-vectors representing the exact value function for a horizon of t. However, at first, Γ_t often contains many unnecessary α-vectors. This is why all the dominated α-vectors are removed at line 19. A dominated α-vector is an α-vector that does not return the best value for any belief state, thus it is never used to calculate the value function. Removing dominated α-vectors helps to keep to a minimum the number of α-vectors representing the value function.

[1]The symbol $\oplus$ denotes the cross-sum operator. A cross-sum operation is defined over two sets, $A = \{a_1, a_2, \ldots, a_m\}$ and $B = \{b_1, b_2, \ldots, b_n\}$, and produces a third set, $C = \{a_1 + b_1, a_1 + b_2, \ldots, a_1 + b_n, a_2 + b_1, a_2 + b_2, \ldots, a_m + b_n\}$

Function EXACT-VALUE-ITERATION($Horizon$)

Returns: A set of α-vectors representing the value function.
Inputs: $Horizon$: The maximal horizon.
Statics: The POMDP model.

for all $a \in A$ **do**
  $\alpha^{a,*}(s) = R(s,a)$
  $\Gamma^{a,*} \leftarrow \{\alpha^{a,*}\}$
end for
$\alpha_0 = [0, 0, \ldots, 0]$ {Initialization with a vector containing only zeros.}
$\Gamma \leftarrow \{\alpha_0\}$
for $t = 1$ to $t = Horizon$ **do**
  for all $a \in A$ **do**
    for all $o \in O$ **do**
      for all $\alpha^{'} \in \Gamma$ **do**
        $\alpha^{a,o}(s) = \gamma \sum_{s' \in S} T(s,a,s')O(s',a,o)\alpha^{'}(s')$
        $\Gamma_t^{a,o} \leftarrow \Gamma_t^{a,o} \bigcup \{\alpha^{a,o}\}$
      end for
    end for
    $\Gamma_t^a = \Gamma^{a,*} \oplus \Gamma_t^{a,o_1} \oplus \Gamma_t^{a,o_2} \oplus \cdots \oplus \Gamma_t^{a,o_{|\Omega|}}$ {The cross-sum over all sets of α-vectors.}
  end for
  $\Gamma_t = \bigcup_{a \in A} \Gamma_t^a$
  $\Gamma_t =$ REMOVE-DOMINATED-VECTORS(Γ_t)
  $\Gamma = \Gamma_t$
end for
return Γ

Algorithm 3.1: Exact value iteration algorithm.

3.1.2.2 Example

This section presents an example of an execution of the exact value iteration algorithm as described in Algorithm 3.1. The problem defined in Figure 3.1 is used for this example. The first step of the algorithm is to create a set $\Gamma^{a,*}$ for each action.

$$\Gamma^{a_1,*} = \begin{bmatrix} R(s_1,a_1) \\ R(s_2,a_1) \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \tag{3.11}$$

$$\Gamma^{a_2,*} = \begin{bmatrix} R(s_1, a_2) \\ R(s_2, a_2) \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \tag{3.12}$$

Afterwards, the algorithm begins at a horizon of $t = 0$, which means that the agent cannot make any action, thus each belief state has a value of 0, which is represented with one α-vector, in which all coefficients for all states equal zero. The α-vector P_0 represents the values for the policy consisting of doing nothing.

$$\Gamma_0 = \underbrace{\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)}_{P_0} \tag{3.13}$$

For the horizon $t = 1$, we begin by constructing the α-vector sets for the action a_1 (lines 12-13). There is one α-vector set for each observation and each set contains one vector for each vector in Γ_0.

$$\begin{aligned} \Gamma_1^{a_1,o_1} &= \begin{bmatrix} \gamma \sum_{s' \in S} T(s_1, a_1, s') O(s', a_1, o_1) P_0(s') \\ \gamma \sum_{s' \in S} T(s_2, a_1, s') O(s', a_1, o_1) P_0(s') \end{bmatrix} \\ &= \begin{bmatrix} 0.9 \times (0.3 \times 0.9 \times 0 + 0.7 \times 0.5 \times 0) \\ 0.9 \times (0.6 \times 0.9 \times 0 + 0.4 \times 0.5 \times 0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned} \tag{3.14}$$

$$\begin{aligned} \Gamma_1^{a_1,o_2} &= \begin{bmatrix} \gamma \sum_{s' \in S} T(s_1, a_1, s') O(s', a_1, o_2) P_0(s') \\ \gamma \sum_{s' \in S} T(s_2, a_1, s') O(s', a_1, o_2) P_0(s') \end{bmatrix} \\ &= \begin{bmatrix} 0.9 \times (0.3 \times 0.1 \times 0 + 0.7 \times 0.5 \times 0) \\ 0.9 \times (0.6 \times 0.1 \times 0 + 0.4 \times 0.5 \times 0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned} \tag{3.15}$$

Then, we combine all the α-vector sets for the action a_1 with the cross-sum operator.

$$\begin{aligned} \Gamma_1^{a_1} &= \Gamma^{a_1,*} \oplus \Gamma_1^{a_1,o_1} \oplus \Gamma_1^{a_1,o_2} \\ &= \begin{bmatrix} 2 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \underbrace{\begin{bmatrix} 2 \\ 1 \end{bmatrix}}_{P_1} \end{aligned} \tag{3.16}$$

By doing the same calculations for the action a_2, we obtain:

$$\Gamma_1^{a_2} = \underbrace{\begin{bmatrix} 1 \\ 3 \end{bmatrix}}_{P_2} \tag{3.17}$$

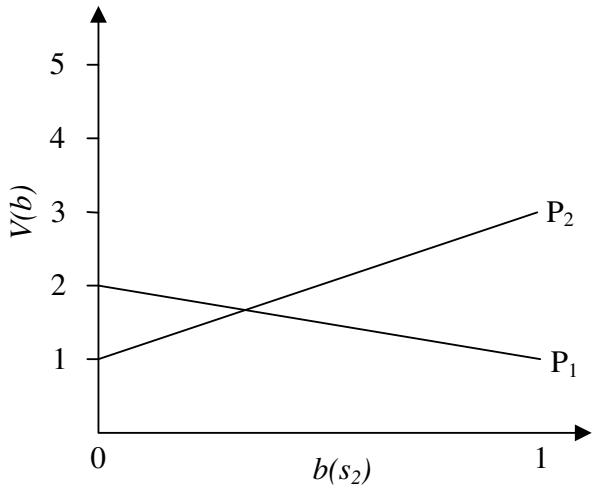


Figure 3.4: Value iteration example at a horizon of 1.

Finally, the solution set for the horizon of 1 is obtained by doing the union of the α-vector sets of all the actions (see Figure 3.4 for a graphical representation).

$$\Gamma_1 = \Gamma_1^{a_1} \cup \Gamma_1^{a_2} = \left(\underbrace{\begin{bmatrix} 2 \\ 1 \end{bmatrix}}_{P_1} \underbrace{\begin{bmatrix} 1 \\ 3 \end{bmatrix}}_{P_2} \right) \tag{3.18}$$

Now, for the horizon $t = 2$, we begin by constructing the α-vector sets for the action a_1. To achieve this, for the observation o_1, we need to construct two vectors, the first one if the policy P_1 is executed afterwards and the second one if the policy P_2 is executed afterwards.

$$\begin{aligned}
\Gamma_2^{a_1,o_1} &= \begin{bmatrix} \gamma \sum_{s' \in S} T(s_1, a_1, s') O(s', a_1, o_1) P_1(s') \\ \gamma \sum_{s' \in S} T(s_2, a_1, s') O(s', a_1, o_1) P_1(s') \end{bmatrix} \\
&\quad \begin{bmatrix} \gamma \sum_{s' \in S} T(s_1, a_1, s') O(s', a_1, o_1) P_2(s') \\ \gamma \sum_{s' \in S} T(s_2, a_1, s') O(s', a_1, o_1) P_2(s') \end{bmatrix} \\
&= \begin{bmatrix} 0.9 \times (0.3 \times 0.9 \times 2 + 0.7 \times 0.5 \times 1) \\ 0.9 \times (0.6 \times 0.9 \times 2 + 0.4 \times 0.5 \times 1) \end{bmatrix} \\
&\quad \begin{bmatrix} 0.9 \times (0.3 \times 0.9 \times 1 + 0.7 \times 0.5 \times 3) \\ 0.9 \times (0.6 \times 0.9 \times 1 + 0.4 \times 0.5 \times 3) \end{bmatrix} \\
&= \begin{bmatrix} 0.801 \\ 1.152 \end{bmatrix} \begin{bmatrix} 1.188 \\ 1.026 \end{bmatrix}
\end{aligned} \tag{3.19}$$

The same calculations can be done for the observation o_2, resulting in:

$$\Gamma_2^{a_1,o_2} = \begin{bmatrix} 0.369 \\ 0.288 \end{bmatrix} \begin{bmatrix} 0.972 \\ 0.594 \end{bmatrix} \tag{3.20}$$

Then, we combine all the α-vector sets for the action a_1 with the cross-sum operator.

$$\begin{aligned} \Gamma_2^{a_1} &= \Gamma^{a_1,*} \oplus \Gamma_2^{a_1,o_1} \oplus \Gamma_2^{a_1,o_2} \\ &= \begin{bmatrix} 2 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 0.801 \\ 1.152 \end{bmatrix} \begin{bmatrix} 1.188 \\ 1.026 \end{bmatrix} \oplus \begin{bmatrix} 0.369 \\ 0.288 \end{bmatrix} \begin{bmatrix} 0.972 \\ 0.594 \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 3.17 \\ 2.44 \end{bmatrix}}_{P_{111}} \underbrace{\begin{bmatrix} 3.773 \\ 2.746 \end{bmatrix}}_{P_{112}} \underbrace{\begin{bmatrix} 3.557 \\ 2.314 \end{bmatrix}}_{P_{121}} \underbrace{\begin{bmatrix} 4.16 \\ 2.62 \end{bmatrix}}_{P_{122}} \end{aligned} \tag{3.21}$$

By doing the same calculations for the action a_2, we obtain:

$$\Gamma_2^{a_2} = \underbrace{\begin{bmatrix} 1.99 \\ 4.62 \end{bmatrix}}_{P_{211}} \underbrace{\begin{bmatrix} 2.791 \\ 4.728 \end{bmatrix}}_{P_{212}} \underbrace{\begin{bmatrix} 2.719 \\ 4.152 \end{bmatrix}}_{P_{221}} \underbrace{\begin{bmatrix} 3.52 \\ 4.26 \end{bmatrix}}_{P_{222}} \tag{3.22}$$

Finally, the solution set for the horizon of 2 is obtained by doing the union of the α-vector sets of all the actions (see Figure 3.5 for a graphical representation).

$$\Gamma_2 = \Gamma_2^{a_1} \cup \Gamma_2^{a_2} \tag{3.23}$$

$$\Gamma_2 = \left(\underbrace{\begin{bmatrix} 3.17 \\ 2.44 \end{bmatrix}}_{P_{111}} \underbrace{\begin{bmatrix} 3.773 \\ 2.746 \end{bmatrix}}_{P_{112}} \underbrace{\begin{bmatrix} 3.557 \\ 2.314 \end{bmatrix}}_{P_{121}} \underbrace{\begin{bmatrix} 4.16 \\ 2.62 \end{bmatrix}}_{P_{122}} \underbrace{\begin{bmatrix} 1.99 \\ 4.62 \end{bmatrix}}_{P_{211}} \underbrace{\begin{bmatrix} 2.791 \\ 4.728 \end{bmatrix}}_{P_{212}} \underbrace{\begin{bmatrix} 2.719 \\ 4.152 \end{bmatrix}}_{P_{221}} \underbrace{\begin{bmatrix} 3.52 \\ 4.26 \end{bmatrix}}_{P_{222}} \right)$$

As we can see in Figure 3.5, some of the α-vectors are dominated, which means that for all belief states, there is an α-vector that has a bigger value. All dominated α-vectors can be removed from the Γ set because they do not contribute to the definition of the value function. An α-vector can be dominated by one α-vector or by a group of α-vectors. It is easier to determine when an α-vector is dominated by one α-vector. For example, in Figure 3.5, P_{211} is dominated by P_{212}. However, it is harder to determine if an α-vector is dominated by a group of α-vectors. To do so, we have to use a linear programming algorithm, which can take a lot of time. In Figure 3.5, P_{112} is dominated by the α-vectors P_{122} and P_{212}. After removing all dominated α-vectors in our example, we are left with only three α-vectors to define the value function at a horizon of 2 (see Figure 3.6).

$$\Gamma_2 = \left(\underbrace{\begin{bmatrix} 4.16 \\ 2.62 \end{bmatrix}}_{P_{122}} \underbrace{\begin{bmatrix} 2.791 \\ 4.728 \end{bmatrix}}_{P_{212}} \underbrace{\begin{bmatrix} 3.52 \\ 4.26 \end{bmatrix}}_{P_{222}} \right) \tag{3.24}$$

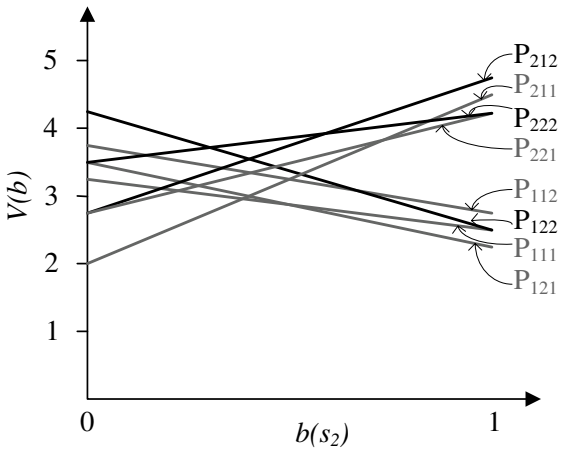


Figure 3.5: Value function at a horizon of 2.

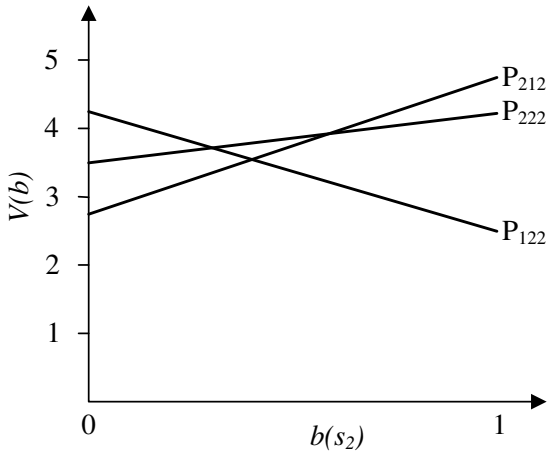


Figure 3.6: Value function at a horizon of 2 without dominated α-vectors.

When the Γ_t set as been created for the desired horizon t, the agent can use it to determine its best policy. To present how it is done, we will use an example. Let $b = [0.2, 0.8]$ be the agent current belief state, as illustrated in Figure 3.7. By using Equation 3.10, the agent can calculate the value of its belief state, which is the maximum value return by one of the α-vectors. The optimal policy for the horizon of t is then the α-vector that returned this best value. In our example, we can see that the agent should execute the policy P_{212}, which means that the agent should execute the action a_2, then if it observes o_1, it should execute a_1 and if it observes o_2, it should execute a_2.

Keeping a complete policy tag for all α-vectors takes a lot of memory, because these tags grow really fast. A more convenient way that takes less memory is to keep only the first action of the policy as illustrated in Figure 3.8. By doing so, the agent does not have a direct access to the optimal policy anymore. The agent has to find each action individually. It finds the first action, then it recalculates its belief state considering the new observation to find the next action. In our example in Figure 3.8, if the agent perceives o_1, it ends up in belief state b_1 (best action a_1) and if it perceives o_2, it ends up in belief state b_2 (best action a_2). As we can see, the two approaches return the same overall policy, but by keeping only the first action for each α-vectors, the size of the tags stays constant.

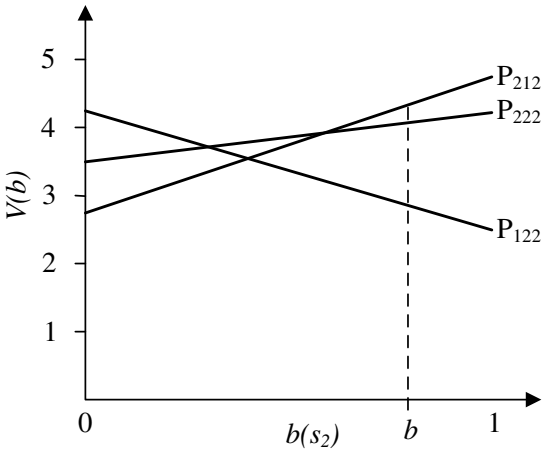


Figure 3.7: The optimal policy for a belief state.

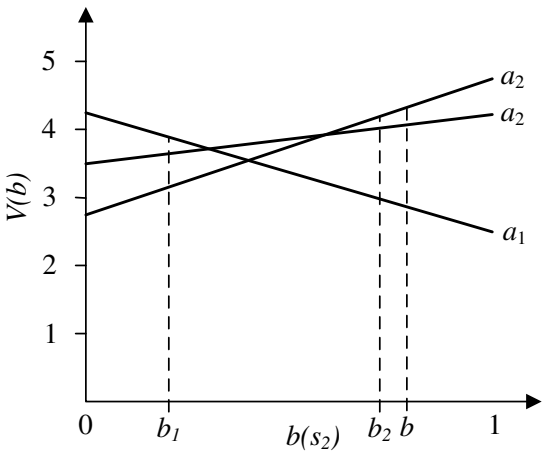


Figure 3.8: The optimal action for a belief state.

3.1.2.3 Complexity of the Optimal Value Iteration Algorithm

For every horizon, the first step is to generate all the α-vectors (or policies) for all set of actions and observations with the following equation:

$$\Gamma_t^{a,o} \leftarrow \alpha_i^{a,o}(s) = \gamma \sum_{s' \in S} T(s,a,s')O(s',a,o)\alpha_i^{'}, \forall \alpha_i^{'} \in \Gamma_{t-1} \tag{3.25}$$

In the worst case, this generates $O(|A||\Omega||\Gamma_{t-1}|)$ α-vectors. Then, we have to apply the cross-sum operator which generates in the worst case: $O(|A||\Gamma_{t-1}|^{|\Omega|})$ α-vectors. Finally, the time needed to calculate each vector depends on the number of states. We have to update all the entries of the α-vector ($|S|$ entries) by iterating over all states. Thus, the complexity for each α-vector is of $O(|S|^2)$. Therefore, the complexity in the worst case to generate the Γ_t set from the Γ_{t-1} set is:

$$O(|S|^2|A||\Gamma_{t-1}|^{|\Omega|}) \tag{3.26}$$

$$|\Gamma_{t-1}| = |A|^{\frac{|\Omega|^{t-1}-1}{|\Omega|-1}}$$

Consequently, the complexity depends a lot on the number of α-vectors conserved at each horizon. This is why it is important to remove all dominated α-vectors in order to keep the Γ_t set as small as possible.

3.1.3 Offline Approximation Algorithms

Due to their complexity, optimal POMDP algorithms are quite useless. They can only be applied to really small problems of only ten to twenty states. In con-

sequence, many researchers have worked on improving the applicability of POMDP approaches by developing approximation approaches that can be applied to bigger problems (Hauskrecht (2000); Murphy (2000); Aberdeen (2003a)). In this section, we present offline algorithms, i.e. algorithms that calculate the agent's policy for all possible situations before the agent has to perform in the environment. There are two main categories of offline approaches: value iteration approaches and policy iteration approaches. Value iteration approaches try to approximate the value function in order to extract the agent's policy from this approximate value function. On the other hand, policy iteration approaches work directly on a representation of the agent's policy. These last approaches often represent the policy as a finite state controller that is improved over time.

3.1.3.1 Value Iteration Approaches

In this section, we present examples of approximation POMDP algorithms that approximate the value function by updating it only for some selected belief states. Thus, not all the α-vectors are updated, but only the α-vectors that define the value function at specific belief states. The idea is that instead of planning over the complete belief space of the agent (which is intractable for large state spaces), planning is carried out only on a limited set of belief states that are chosen or sampled by using the POMDP model.

Some methods choose some belief states at the beginning (Lovejoy (1991); Littman et al. (1995); Hauskrecht (1997)). They can choose belief states at some given interval, or randomly, or at the position of the states of the underlying MDP. These methods are called fixed grid-based methods because their grid of chosen belief states over the belief state space stays constant. Some other methods prefer to use variable problem dependant sets of belief states and consequently, they can be seen as variable grid-based methods (Cheng (1988); Brafman (1997); Hauskrecht (1997); Zhou and Hansen (2001); Bonet (2002)). This enables them to have more belief states to represent the value function in the most important regions of the the belief state space.

Finally, we present PBVI (Pineau et al. (2003)), HSVI (Smith and Simmons (2004)) and Perseus (Spaan and Vlassis (2005)), which are three recent variations of the point-based approach. These algorithms consider a starting belief state from which they try to predict the belief states that will be reachable by the agent. The point-based approach consists in updating not only the values of the chosen belief states, but also their gradient. The value function is thus improved for all the belief state space and not only for the chosen belief states. These three methods are distinguished by the approach they use for choosing belief states and by the method they use to update the value function at these chosen belief states.

Fixed Grid-based Methods It is well known (Sondik (1978)) that an optimal policy for a POMDP with n states can be obtained by solving the belief-space MDP whose state space consists of all probability distributions over the state space of the POMDP, i.e., an n-dimensional simplex. Grid approximations attempt to solve the belief-space MDP directly by placing a discrete, finite grid on this n-dimensional space and restricting their calculations to the points of this grid. The computational effort required for approximating the value function on a k point grid is roughly equivalent to that of solving a k state MDP.

It is possible to use a fixed regular grid (Lovejoy (1991)) or a fixed random grid (Hauskrecht (1997)). With the regular grid, all the grid points are equally spaced in the belief space. The grid construction is simple and fast, and the regularity of the grid facilitates simple value interpolation algorithms to estimate the value function between the grid points. With a random grid, it takes more time to do the interpolation for estimating the value of new points.

The Q_{MDP} method of Littman et al. (1995) can be viewed as a fixed grid based method in which the grid consists of the states of the underlying MDP. The first step is to solve the POMDP as if it was completely observable. The solution to the underlying MDP gives a value $V(s)$ to each state. Thus the grid points are at the position of each state. For a non-grid belief state b, its value is estimated as the sum of the values of all the states pondered by the probability of being in each of these states:

$$V(b) = \sum_{s \in S} b(s)V(s) \tag{3.27}$$

This method performs relatively well, but it causes the agent to act as if it will end up in a state of perfect information, i.e., a belief state corresponding to one of the underlying MDP's states. Consequently, the agent does not learn how to efficiently use the information gathering actions. For example, the agent would not learn how to use a sensing action if the only effect of a sensing action is to clarify its belief state, because during the learning phase, there are no uncertain belief states considered.

Variable Grid-based Methods On the other hand, variable grid methods use a problem dependent grid that may change during the solution process. These methods concentrate the effort where it is most needed and they obtain good approximations with a much smaller grid than the fixed methods. However, interpolation is harder to perform, and thus some variable grid construction methods are quite complex, requiring considerable computation time and space (e.g., Cheng (1988)).

Brafman (1997) has developed a variable grid algorithm for obtaining approximate solutions to POMDPs in the infinite horizon case. The first points of the grid are

the ones corresponding to the underlying MDP. These first points are used to get an initial estimate of the value function. Then, it uses this estimation to generate new grid points that could improve the value function. The value function is then estimated using these new grid points. This algorithm can loop between the steps of generating new grid points and calculating the new estimate of the value function. At the beginning, the grid points are the states of the underlying MDP. When looking for new grid points, for each pair of grid points, the algorithm generates a grid point in the middle and if this grid point has a good chance to be visited, then it is added to the grid point set.

Hauskrecht (1997) has also developed a variable grid algorithm. It starts by adding all the points corresponding to the states of the underlying MDP, then it adds the successor belief points, which are generated using a one-step stochastic simulation (Equation 3.3).

Another variable grid approach is the algorithm developed by Zhou and Hansen (2001). These authors tried to combine the advantages of the fixed and variable grid based methods by sub-sampling the fixed grid proposed by Lovejoy (1991). Such algorithm allows both fast interpolation and increased resolution in the most useful areas of the belief space. However, it still needs a lot of grid points to achieve good performances.

Bonet (2002) has developed a variable grid-based algorithm which is ϵ-optimal. However, to attain such guaranty, his algorithm needs a lot of grid points. In fact, grid-based methods in general do not scale well for large state spaces since the size of the grid tends to grow exponentially with the number of states.

Point-based Approaches The point-based approach is an improvement of the variable grid-based approach. In the point-based approach, some representative belief states are sampled like in the variable grid-based approach. However, in the point-based approach, the belief states are sampled by starting in the initial belief state and by simulating some random interactions of the agent with the POMDP environment. By doing so, the belief states sampled have more chance of being reached during the execution of the agent. Another important improvement is that the point-based approach does not just update the value at the sampled belief states, but also updates their gradient. This means that the value function is defined for all the belief state space and not just for the sampled belief states. In the following, we present three examples of point-based approaches.

The first point-based approach is the Point-Based Value Iteration (PBVI) algorithm (Pineau et al. (2003); Pineau (2004)). PBVI maintains a set B of reachable belief states from the starting belief state b_0. To do so, PBVI iteratively expands its set B by adding a new belief state for each belief state already in B, using a one step stochastic exploration strategy. For each belief state $b \in B$ and for each action $a \in A$, PBVI samples a state

s from the distribution b, a resulting state s' from the distribution $T(s, a, *)$ and an observation o from the distribution $O(s', a, *)$. From these samples, it generates a new belief state b_a by using the belief update function: $b_a = \tau(b, a, o)$. Finally, it keeps only the new belief state which is the farthest away from any point already in B. To sum up, PBVI interleaves phases of belief state set expansion and value iteration, that is why it is considered as an anytime algorithm. The value iteration phase only tries to improve the values at the sampled belief states. Consequently, PBVI only keeps one α-vector for each sampled belief states. In fact, PBVI can keep less α-vectors than sampled belief states if one α-vector maximizes more than one belief state. After each value iteration phase, PBVI proposes a solution that improves as the number of belief states in B grows.

Another algorithm based on the value iteration approach is the HSVI algorithm (Smith and Simmons (2004, 2005)). Instead of keeping only one lower bound like PBVI, HSVI also maintain an upper bound. The lower bound is a set of α-vectors like PBVI and the upper bound is represented as a convex hull defined with a set of belief points. At the beginning, the lower bound is initialized with only one α-vector representing the worst possible case if the same action is applied indefinitely. The upper bound is initialized with the solution of the underlying MDP. Afterwards, the bounds are updated at specified belief points. For the lower bound, the update at a belief state b consists at adding a new α-vector defining the value function at b, as in the PBVI algorithm. For the upper bound, the update at a belief state b consists at adding a new belief point in the set defining the convex hull. The belief point added is the best belief point obtained after trying all possible actions from b. The belief points are chosen by doing a search in the tree of the belief states attainable from the initial belief state b_0. The search is directed by using the lower and upper bounds.

Spaan and Vlassis have developed an algorithm called Perseus (Spaan and Vlassis (2004); Vlassis and Spaan (2004); Spaan and Vlassis (2005)), which is an adaptation of the PBVI algorithm. Perseus operates on a large set of belief states which are gathered by simulating random interactions of the agent with the POMDP environment. Then, a number of value function updates are performed on this belief state set. The algorithm ensures that in each value function update the value of all belief states in the belief state set is improved (or at least does not decrease). Contrary to PBVI, Perseus updates only a random subset of belief states. The key idea is that, in each value iteration step, it is possible to improve the value of all points in the belief set by only updating the value and its gradient of a subset of the points. This allows to compute value functions that consist of only a small number of vectors (relative to the belief set size), leading to significant speedups.

The point-based approaches are quite interesting because they can concentrate the computation on the attainable belief states. Therefore, attainable belief states have

more chance to be optimized. Even if they are not optimized, all the other belief states are also defined in the value function, because point-based approaches keep an α-vector for each sampled belief state and not just its value. One disadvantage of these methods is that they only optimize over a relatively small number of belief points, which is sometimes too small to give a good solution. Another drawback is that the α-vectors can become really hard to manage in big state spaces, because they have as many elements as there are states in the environment.

3.1.3.2 Policy Iteration Approaches

Instead of learning a value function and then extracting a policy, some methods directly try to optimize the policy. Most methods in this case restrict the space of policies to policies that can be represented as finite state controllers (FSCs). Starting from that, their approach consists in searching for the best policy represented by a finite state controller of some limited size. Policies represented by FSCs are defined by a (possibly cyclic) directed graph $\pi = \langle N, E \rangle$, where each node $n \in N$ is labeled by an action a and each edge $e \in E$ by an observation z. Each node has one outward edge per observation. A policy is executed by taking the action associated with the "current node" and updating the current node by following the edge labeled by the observation made. Many algorithms have been developed to search in the space of FSCs:

- policy iteration (PI) (Hansen (1997, 1998); Hansen and Zhou (2003); Poupart and Boutilier (2003a)),
- gradient ascent (GA) (Aberdeen and Baxter (2002); Aberdeen (2003b); Meuleau et al. (1999a)),
- branch and bound (B&B) (Meuleau et al. (1999b)),
- belief-based stochastic local search (BBSLS) (Braziunas and Boutilier (2004)).

In our experiments on the Tag problem (see Table 3.2 on page 78), our approach is compared with two of these algorithms (BBSLS and BPI), which are briefly presented here.

BBSLS Braziunas and Boutilier developed an algorithm called Belief-Based Stochastic Local Search (BBSLS) (Braziunas and Boutilier (2004)). This algorithm searches in the space of finite-state controllers (FSCs) in order to find an approximate policy for a POMDP problem. One of the drawbacks of traditional gradient ascent methods is that they have a tendency to get trapped in local optima. In order to have a method that

can escape from the local optima, BBSLS adds local moves inspired from the dynamic programming approach. These local moves enable to choose policies at unreachable belief states that would not be chosen by the local search. These moves are those that would give good rewards if the pre-condition belief state was met. To make sure these new moves are considered, they are added in a tabu list, which prevent them from being removed. This is to let some time to the local search to adjust to these new moves. The local search is done by doing global moves that correspond to direct stochastic hill-climbing, and are designed to increase controller value immediately, often taking advantage of earlier local moves. To sum up, at each step, the BBSLS algorithm performs one or more local moves to get out of local optima, followed by a sequence of global moves to improve the controller.

BPI Poupart and Boutilier developed an algorithm called bounded policy iteration (BPI) (Poupart and Boutilier (2003a); Poupart (2005)). Traditional policy iteration algorithms are guaranteed to converge to an optimal policy, however the size of the controller often grows intractably. In contrast, gradient ascent methods restrict their search to controllers of a bounded size, but may get trapped in local optima. The BPI algorithm improves the policy much like policy iteration algorithms (Hansen (1998)), but while keeping the size of the controller fixed. However, in order to get out of a local optimum, the controller is allowed to slightly grow by adding one (or a few) node(s) to escape the local optimum.

Policy iteration methods are quite interesting because they often converge rapidly. However, it might be hard to find the best policy representation for each problem. Also, gradient search methods can get trapped in local minima.

3.1.3.3 Value Function Approximations

Some methods are trying to directly approximate the POMDP's value function. Bertsekas and Tsitsiklis (1996) developed a technique called *neuro-dynamic programming* in which they train a neural network by dynamic programming to approximate the Q-functions. Another approach is the one of Parr and Russel (1995) in which they use a smooth and differentiable function that is optimized by gradient descent. Another way to approximate a POMDP's value function is to use particle filtering to do approximate tracking of the belief state and using a nearest-neighbor function approximation for the value function (Thrun (2000)).

3.1.3.4 Reusable Trajectories

Kearns et al. have developed an algorithm to evaluate the quality of POMDP policies by doing an exploration of trajectories in a tree structure (Kearns et al. (2000)). Their algorithm is not used to construct a POMDP policy. These authors consider that they already have a finite set of policies and, from this set, they are trying to choose the best policy. To evaluate each policy, they suppose that they have a generative model of the POMDP to generate m trajectory trees. The best policy is the policy that maximizes the average reward returned by applying the policy in the trajectory trees. Each node of the tree contains a state and an observation. From a node, there is a branch for each possible action. When an action is tried in a state, the generative model is called and it returns the resulting state, observation and reward.

To generate the trajectory trees, they need to know the underlying states, which is not normally accessible in a POMDP. To take the partial observability into consideration, they have developed another algorithm in which they generate m observation histories instead of m trajectory trees. With these observation histories, they estimate the expected value of a belief state for a specific policy π as the average return of the histories that are accepted by the policy π. An history is accepted by a policy if the actions returned by the policy would have been the same as the actions recorded in the history. The observation history approach needs a less powered generative model then the trajectory trees approach. However, both approaches need similar amounts of experiences given by the generative model of the POMDP in order to obtain good estimates.

3.1.3.5 Belief State Bounds

Varakantham et al. (2005) have worked on using belief state bounds to speedup exact and approximate POMDP algorithms. The key idea was to notice that in some problems, a large part of the belief state space is unreachable. Thus, it would be more efficient to concentrate the computation on the reachable belief states. Their techniques for exploiting belief region reachability exploit three key domain characteristics: (i) not all states are reachable at each decision epoch, because of limitations of physical processes or progression of time; (ii) not all observations are obtainable, because not all states are reachable; (iii) the maximum probability of reaching specific states can be tightly bounded. These authors introduce polynomial time techniques based on Lagrangian analysis to compute tight bounds on belief state probabilities. These bounds can then be used to speedup most existing exact and approximate POMDP algorithms.

3.1.4 Online Approximation Algorithms

Usually, with offline approaches, the algorithm returns a complete policy defining which action to execute in every possible belief state. Such an approach is not applicable for problems having a very large belief state space, because there are too many situations to consider. In large POMDPs, a more fruitful way of thinking might be an online view, in which the policy is calculated only for belief states that have been reached online. In this section, we present some online search approaches that are using local search algorithms to construct a policy online based on the current belief state.

These kind of approaches are also known as agent-centered search (Koenig (2001)). Agent-centered search methods usually do not plan all the way from the start state to a goal state. Instead, they decide on the local search space, search it, and determine which actions to execute within it. Then, they execute these actions (or only the first action) and repeat the overall process from their new state, until they reach a goal state.

We also present in this section some history-based algorithms which are classified here as online algorithms because they need to interact with the environment to gather some experiences in order to define their policies based on the agent observation history. These methods do not consider the availability of the POMDP model, therefore their policies are not based on the belief states because they would not be able to maintain such belief states without the POMDP model.

3.1.4.1 Online Search Approaches

An online algorithm takes as input the current belief state and return the single action that seems to be the best for this particular belief state. In this online view, the online POMDP algorithm is itself simply a policy, but one that may need to perform some non trivial computation at each belief state in order to return the best action (Kearns et al. (2002)). Consequently, online POMDP algorithms are really interesting to manage large belief state spaces, because they concentrate only on the most important belief states. However, they may take a lot of time choosing an action to ensure some optimality guaranties. Here, we present three examples of online search methods.

BI-POMDP Washington developed the BI-POMDP algorithm (Washington (1997)), which expands an AND/OR tree to find an approximate policy for the POMDP. The nodes of the tree are belief states. The actions form the OR branches, since the optimal action is a choice among the set of actions. The observations form the AND branches, since the utility of an action is a sum of the utility of the belief state implied by each

possible observation multiplied by the probability of this observation. An iterative AO* algorithm (Nilsson (1980)) is then used to expand the tree.

However, the AO* algorithm does not specify in which order the AND nodes are expanded. The BI-POMDP algorithm uses a strategy that chooses the node that presents the greatest potential to change the estimated value of the overall path. The chosen node is the one with the largest difference between the lower and the upper bounds. For the lower bound, the BI-POMDP algorithm uses the solution of the underlying MDP as an estimation of the value function for the POMDP. For the upper bound, the BI-POMDP uses the worst case MDP, which consists in minimizing the expected rewards instead of maximizing them. Normally, the solution to the underlying MDP would be the upper bound and the worst case MDP would be the lower bound, but Washington uses disutilities instead of utilities, thus the bounds are reversed. When the bounds are equal, it means that the algorithm has found the optimal policy since the optimal policy has a value between the lower and upper bounds.

The BI-POMDP can be executed online where to choose each action, it does a search for a given time. However, the BI-POMDP algorithm also needs some time offline to calculate the solution of the underlying MDP, which is used to calculate the bounds.

RTDP-BEL The RTDP-BEL algorithm learns some heuristic values for the belief states visited by successive trials in the environment (Geffner and Bonet (1998)). At each belief state visited, the agent evaluates all possible actions. For each action a, the agent estimates the expected reward of taking action a in the current belief state b with the following equation:

$$Q(a,b) = R(a,b) + \gamma \sum_{o \in O} P(o|b,a)V(b') \tag{3.28}$$

Where $V(b')$ is the value learned for the belief state b'. If the belief state b' has no value in the table, then it is initialized to the Q_{MDP} value. Consequently, the underlying MDP has to be solved before the execution of the RTDP-BEL algorithm.

The agent then executes the action that returned the greatest $Q(a,b)$ value. Afterwards, the value $V(b)$ in the table is updated with the $Q(a,b)$ value of the best action. Finally, the agent executes the chosen action and it makes the new observation, ending up in a new belief state. This approach is repeated until the goal is reached by the agent.

The RTDP-BEL algorithm learns a heuristic value for each belief state visited. Therefore, in order to have an estimated value for each belief state in memory, it needs to discretize the belief state space to have a finite number of belief states. It might be

difficult to find the best discretization for a given problem. In practice, this algorithm needs a lot of memory to store all the learned belief state values.

Planning for factored POMDP Another online algorithm is the one presented by McAllester and Singh (1999), where these authors used an online exploration of the belief state space, similar to the one used by the BI-POMDP algorithm. The difference is that they do not explore all observations for each possible action in a given belief state, they rather sample C observations from a generative model for each action. Also, at the leaves of the tree, they did not use any estimation of the value function, they simply gave a value of zero for the leaves. Their search algorithm is a complete depth limited search. Their algorithm is an adaptation of the online MDP algorithm presented in Kearns et al. (2002). They have also used a belief state factorization approach taken from Boyen and Koller (1998) to simplify the belief state calculations.

3.1.4.2 History-Based Approaches

When the model of the environment is unknown, it is possible to construct POMDP's policies based on the past actions and observations instead of the belief state (Chrisman (1992); McCallum (1996); Dutech (2000); Littman et al. (2001)). In such settings, the agent does not know the transition function, the observation function and the reward function. These algorithms are able to construct good policies based on the agent's past, but they normally need a lot of experiences.

The UTree algorithm (McCallum (1996)) uses the agent's past experiences to construct a simplified tree representation of the state space. This tree regroups past experiences that have similar rewards. Afterwards, based on its current past actions and observations, the agent can go down the tree and find the corresponding abstract state (a leaf of the tree), which contains the Q-values defining the agent's policy.

Another approach based on past observations and actions is the Predictive State Representation (PSR) (Littman et al. (2001); Rudary and Singh (2003); Singh et al. (2003); James and Singh (2004); Singh et al. (2004); Rosencrantz et al. (2004)). However, instead of basing its action choice on past actions and observations, it base its action on predictive tests. This algorithm learns predictive tests from past experiences. These predictive tests are arrays of future actions and observations (for example, $a_1 o_1 a_2 o_2$). The correct prediction for this example test given past experiences until time k is the probability of these observations occurring (in order) given that these actions are taken (in order) (i.e. $P(O_k = o_1, O_{k+1} = o_2 | A_k = a_1, A_{k+1} = a_2)$). A PSR is a set of tests that is sufficient information to determine the prediction for all possible tests (a sufficient statistic).

3.1.5 Factored POMDP

The traditional POMDP model is not necessarily suited for large environments because it requires enumerating explicitly all the states for the transition, observation and reward functions. The planning problem becomes quite hard when the number of states increases. This is called the *curse of dimensionality*: in a problem with n physical states, the policy π is defined over all belief states in an $(n-1)$-dimensional continuous space (Pineau (2004)). Consequently, the complexity of a POMDP problem is highly dependant on the number of states.

To tackle the curse of dimensionality, many researchers have tried to exploit the structure of the problems to factorize the state space. Most environments are structured and can thus be described as a set of different features which allows representing the states much more compactly. The states can then be defined with a set of random variables. Let $X = \{X_1, \ldots, X_M\}$ be the set of M random variables that fully describe a state, and D_i be the set of all possible values for the random variable X_i. We suppose that each variable has a finite number of possible values. Therefore, a state is defined by assigning a value to each variable: $s = \{X_1 = x_1, \ldots, X_M = x_M\}$ where $x_i \in D_i$. Consequently, the number of states is: $|S| = D_1 D_2 \cdots D_M$. We also use a more compact representation for factored states: $s = \{x_i\}_{i=1}^{M}$, as suggested by Sallans (2002).

When the states are factorized, it is then possible to represent the POMDP components compactly and still be able to optimally solve the POMDP (Boutilier and Poole (1996)). The transition and observation functions of a POMDP can typically be compactly represented as a dynamic Bayesian network (DBN) (Dean and Kanazawa (1989)), which is a graphical representation for stochastic processes that exploits conditional independence. Conditional independence refers to the fact that some variables are probabilistically independent of each other when the values of other variables are held fixed (Pearl (1988)).

Figure 3.9 presents an example of a dynamic Bayesian network that specifies a transition function for a specific action. In this example, variables are considered boolean variables. The acyclic graph represents the dependencies between the variables' values at time t and their values at time $t+1$. For example, the value of X_2 at time $t+1$ depends on the values of X_1 and X_2 at time t. The transition probabilities can then be specified in conditional probability tables (CPT). These CPTs can be represented even more compactly with a decision tree (Boutilier et al. (1996, 2000)), algebraic decision diagrams (Hansen and Feng (2000); Hoey et al. (1999)) or horn rules (Poole (1993, 1997)).

The reward function can also be represented using the same compact representations (see Figure 3.10). The reward function can be represented in a structured fashion using

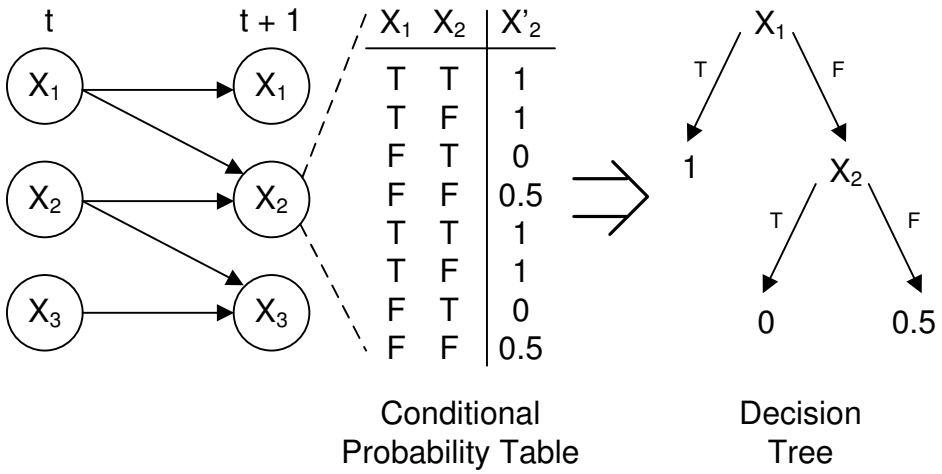


Figure 3.9: Dynamic Bayesian network.

Figure 3.10: Reward function network.

a value node and a decision tree describing the influence of various combinations of variables on rewards (as with decision tree CPTs) (Boutilier and Poole (1996)). Leaves of the tree represent the reward associated with the states consistent with the labeling of the corresponding branch.

When a policy is represented as a mapping from belief states to actions, the agent's belief state has to be quickly updated at each time step during the execution of the policy. As mentioned before, the belief state updates are done using the belief state update function presented in Equation 3.3. In a factored POMDP setting, it might be possible to factor the belief states into products of marginals (unconditional probabilities) by exploiting the conditional independence of some variables describing an environment state.

According to the factored representation of states, the belief state definition has to be slightly modified. In this context, a belief state b is defined as a full joint probability distribution over all random variables: $b = \mathbf{P}(X_1, \ldots, X_M)$. Without anymore assumptions, we are not gaining anything since the full joint probability table has as many cells as there are states in the environment. It is however really useful if we consider

some independence between the variables. Then, the full joint probability table can be split into smaller tables. At the extreme, if all environment variables are considered independent from one another, then the belief state can be reformulated as a product of the variables' probability distributions: $b = \mathbf{P}(X_1) \cdots \mathbf{P}(X_M)$. It follows that the probability of being in a state can easily be computed by doing the product of the variables' probabilities. In general, if the variables can be partitioned into probabilistically independent subsets, the joint probability distribution can be compactly represented by the product of the marginal distributions of each subset.

If at every time-step belief states can be factored into sets of marginals, then belief state monitoring algorithms can focus only on factored belief states which have a reduced dimensionality equal to the sum of the size of each marginal. The size of a marginal is exponential in the size of the subset of variables it corresponds to, however for small subsets, this effectively reduces dimensionality (Poupart (2005)). Unfortunately, as observed by Boyen and Koller (1998), even in the extreme case in which the initial belief state is completely factored (all state variables are mutually independent), correlations introduced by the transition and observation functions tend to render most (if not all) state variables correlated after some time.

Nevertheless, many of those correlations are weak in practice and this suggests an approximation scheme where we force some subsets of variables to remain independent by breaking at each time step any correlation that could creep in. Boyen and Koller (1998) proposed to project at each time step the exact joint distribution of the belief state onto a predetermined set of marginals that partition state variables into mutually independent subsets. They also showed theoretically that this approximation technique can significantly speed up belief state monitoring, while ensuring that the KL divergence[2] between exact and approximate belief states remains bounded irrespective of the number of projection operations performed.

It has also been shown theoretically that planning, in a rapidly mixing POMDP, based on an accurate belief state simplication results in a bounded deviation from the optimal rewards (McAllester and Singh (1999)). These last authors have developed an online POMDP algorithm, based on the factorization approach of Boyen and Koller (1998), that expands a search tree of future actions and observations to find the actions that maximizes the expected reward.

During the execution of a POMDP policy, belief state monitoring is conducted only to facilitate action selection. Thus, although the KL divergence between exact and

[2]Given two probability mass functions $p(x)$ and $q(x)$, $D(p\|q)$, the Kullback-Leibler divergence (or relative entropy) between p and q is defined as: $D(p\|q) = \sum_x p(x) log \frac{p(x)}{q(x)}$. Even though it is not a true distance between distributions (because it is not symmetric and does not satisfy the triangle inequality), it is still often useful to think of the KL-divergence as a "distance" between distributions (Cover and Thomas (1991)).

approximate belief states remains bounded, the policy may be altered since the action selected for every approximate belief state may be different from the one prescribed by the exact belief state (Poupart (2005)). Therefore, for the agent's performances, it is more important to select the best actions than to have a perfect belief state at each time step. Consequently, it might be more interesting to evaluate the belief state factorization based on the decision quality compared to a distance from the perfect belief state. To do so, Poupart and Boutilier have developed algorithms to select sets of marginals that directly minimize the impact on decision quality rather than KL divergence (Poupart and Boutilier (2001)).

The algorithm of Poupart and Boutilier (2003b) finds a linear projection of the belief state, which facilitates the planning phase. However, it is also possible to consider non-linear projections (Roy and Gordon (2003)). With their E-PCA algorithm, Roy and Gordon uses Exponential-family Principal Component Analysis to project the beliefs onto a non-linear projection. The belief state compression is often better, but the planning phase becomes harder with non-linear projections. Roy has shown that it is possible to adapt a grid-based approach to work with non-linear projections (Roy (2003)), however his approach does not offer any theoretical guarantees about its distance from the optimal performance.

Another approach using belief state factorization is the work of Sallans (2000) in which he used factored representations of belief states to reduce the number of parameters for learning the POMDP dynamics as well as an optimal value function. Finally, it is worth mentioning the work of Doshi and Gmytrasiewicz (2005), which uses particle filters to estimate the agent's belief states in a multiagent environment. To efficiently take the other agents into consideration, these last authors have incorporated the beliefs of the other agents in the POMDP model (Gmytrasiewicz and Doshi (2005)).

Now that we have presented many POMDP algorithms, we present in the next section the motivations that motivated us to develop another POMDP algorithm.

3.2 Motivations

As previously explained, our main motivation was the RoboCupRescue simulation environment (see Chapter 2) in which we needed a decision-making approach for the *PoliceForce* agents. These agents are evolving in a highly dynamic and uncertain environment in which they have to choose efficient actions while maintaining an effective coordination with all other rescue agents. Here is a list of the principal constraints that our algorithm has to manage:

- The environment is partially observable;

- The environment is highly dynamic;
- The algorithm has to be efficient in large state spaces;
- The algorithm has to be efficient in previously unknown instances of the environment (in new cities in the RoboCupRescue simulation);
- The agent's response time has to respect a real-time constraint;
- The algorithm should take the other agents into consideration to maintain a good coordination.

These constraints are not only present in the RoboCupRescue simulation environment, but also in many other environments. For example, autonomous robots that are conceived to work in the real world have to manage these constraints, because the real world is highly dynamic and uncertain. The real world has an infinite state space in which it is really easy for a robot to end-up in unknown places. In order to stay efficient in a fast changing world, a robot has to react rapidly to changes in its environment. Moreover, the robot is often not alone, it has to interact with other robots or humans.

In real world applications, a POMDP algorithm should guarantee a fast agent response time. Moreover, an agent should be immediately efficient in any configuration of the modeled environment. This last constraint eliminates all offline approaches, because the agent does not have time to learn a complete policy before its execution. Most offline POMDP algorithms are not flexible enough to manage little changes in the environment. Offline POMDP algorithms normally have to recompute a complete policy after each changes which can take hours or days for complex POMDPs. In our case, we need an algorithm that can manage environment changes while limiting the time needed to compute the agent's policy. In such settings, an online algorithm seems the most appropriate approach.

To sum up, those constraints motivated us to develop our online POMDP algorithm that can ensure a quick response time in a huge state space. In the following, we describe in detail how our online POMDP algorithm works, but first we describe how we have incorporated the factorization approach in our algorithm.

3.3 Using the Factored Representation

In this thesis, we are interested in POMDPs with large state spaces. In such environments, it might become hard for the agent to find the best actions because of the curse of dimensionality. However, as explained in section 3.1.5, factorizing the state space can help to reduce the impact of the curse of dimensionality. Consequently, we

$$b = \left(\begin{array}{ccc} X_1 & X_2 & X_3 \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} & \begin{bmatrix} 0.25 \\ 0.05 \\ 0.50 \\ 0 \\ 0.10 \\ 0.10 \end{bmatrix} & \begin{bmatrix} 0.10 \\ 0.30 \\ 0.15 \\ 0.45 \end{bmatrix} \end{array} \right) \qquad b' = \left(\begin{array}{cc} X_1 & X_2X_3 \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} & \begin{bmatrix} P(X_2 = x_1 \wedge X_3 = x_1) \\ P(X_2 = x_1 \wedge X_3 = x_2) \\ \dots \\ \dots \\ \dots \\ P(X_2 = x_6 \wedge X_3 = x_4) \end{bmatrix} \end{array} \right)$$

Figure 3.11: Examples of factored belief states with all independent variables (left) or with two dependent variables (right).

consider that our approach is applied in POMDP environments that can be specified using a factored representation. This means that an environment state is described as a set of random variables each of which representing a characteristic of a state. Consequently, the transition, observation and reward functions can be represented with dynamic Bayesian networks and decision trees (as presented in section 3.1.5).

For the belief state factorization, we consider that we can identify independent subsets of variables via domain knowledge or via an algorithm that can find such subsets automatically (Boyen and Koller (1998); Poupart (2005)). These algorithms are quite interesting, because even if some variables are dependent, they are still able to factorize the belief state with minimal degradation of the solution's quality.

To illustrate how the factorization can be useful, let's suppose an environment that can be described by 3 variables X_1, X_2 and X_3. Each of these variables can respectively take 5, 6 and 4 different values, which means that the environment has 120 states. In Figure 3.11, we can see two examples of belief states. On the left, all variables are considered independent and there is one vector representing the probability distribution of each variable. On the right, the variables X_2 and X_3 are considered dependant, which means that there is one vector for the two variables. This vector represents the combined probability distribution of these two dependant variables. In general, there will be one vector for each independent subsets of variables identified by the factorization algorithm.

By maintaining the probabilities on variables instead of states, it is much easier to update the belief state and the expected value of a belief state can be calculated much faster. For example, let's suppose that we want to know the reward of being in the belief state shown in Figure 3.11(left) for a specific action a ($R_B(b, a)$). This current reward is the first step when evaluating a belief state and, as shown by Equation 3.7, it involves a summation on all states:

$$R_B(b, a) = \sum_{s \in S} b(s) R(s, a) \tag{3.29}$$

However, when looking at our example, we see that it might not be necessary to sum on all 120 states since many of them are not possible. For example, the agent cannot be in any of the 24 states where the variable X_1 has the first value, because there is a zero probability for this to happen. In fact, if we remove all impossible states, we end up with only 20 states. To calculate the possible number of states, we only have to multiply the number of possible values for each variable ($1 \times 5 \times 4$ in our example). More formally, the number of possible states for a specified belief state $\omega(b)$ is defined as:

$$\omega(b) = \{\{x_i\}_{i=1}^{M} \mid (\forall x_i)\ Pr(X_i = x_i|b) \neq 0\} \tag{3.30}$$

This function returns all the states the agent could be in, according to a belief state. We know that a state is impossible if one of the variables has a probability of zero according to b. In the worst case, this function returns the set S, i.e. all states. If the variables are ordered approximately according to their certainty, this subset of states can be constructed quite rapidly because each time we encounter a variable with a zero probability, we can immediately exclude all the corresponding states. The following equation can then be computed much more rapidly than Equation 3.29:

$$R_B(b, a) = \sum_{s \in \omega(b)} b(s)R(s, a) \tag{3.31}$$

The only difference in this equation is that the summation is defined on a subset of states ($\omega(b)$) instead of the whole state space. The less uncertainty the agent has, the smaller the subset of possible states is and the faster the computation of Equation 3.31 is, compared to Equation 3.29. If the agent directly perceives some characteristics of the environment, then the probability distribution of the corresponding variables would be represented by a vector containing only zeros except for the perceived value which would have a value of 1. Therefore, if all these observable variables are explored first when constructing the set $\omega(b)$ then a lot of states can be eliminated early on, which accelerates the construction of $\omega(b)$. Often, there are also different degrees of partial observability for some variables in a POMDP and that can be used by ordering the variables according to their certainty in order to further accelerate the construction of $\omega(b)$.

Now that we consider only possible states, we would also like to have a function that returns the states that are reachable from a certain belief state. For this, we define a new function $\alpha(a, b, o)$ that takes as parameters the current belief state b, the action performed a and the observation perceived o, and returns all reachable states.

$$\alpha(a, b, o) = \{s' \mid (\forall s \in \omega(b))\ T(s, a, s') \neq 0 \wedge\ O(s', a, o) \neq 0\} \tag{3.32}$$

In other words, the α function returns the set of subsequent states with transition and observation probabilities greater than zero. It follows that the probability of making

an observation can also be calculated more efficiently using α and ω, because it iterates on less states:

$$Pr(o \mid a, b) = \sum_{s' \in \alpha(a,b,o)} O(s', a, o) \sum_{s \in \omega(b)} T(s, a, s') b(s) \tag{3.33}$$

The belief state update function $\tau(b, a, o)$ can also be computed more efficiently with a factored representation (Boyen and Koller (1998); Poupart (2005)). If we use our ω and α functions, we can rewrite the belief state update function. If $b' = \tau(b, a, o)$, then $\forall s' \in \alpha(a, b, o)$:

$$b'(s') = \eta O(s', a, o) \sum_{s \in \omega(b)} T(s, a, s') b(s) \tag{3.34}$$

3.4 Online Decision Making

As mentioned above, offline POMDP algorithms are still mostly limited to small problems. Even state of the art approximation algorithms can at best be applied on medium sized problems. The main problem of such approaches is that they have to construct and represent a policy for all possible situations. Therefore, the policy construction step takes normally a lot of time before the agent can be efficiently executed in the environment.

On the other side, an online POMDP algorithm does not have to construct a policy for all possible situations. The policy is constructed online while the agent is evolving in the environment. In other words, the policy construction steps and the execution steps are interleaved with one another as shown in Figure 3.12. Normally, online approaches need a little more execution time because the policy might not be optimal in long term, since the policy is locally constructed. However the policy construction time is normally smaller, because online approaches only consider reachable belief states. Consequently, the overall time for the policy construction and execution is normally less for online approaches (Koenig (2001)). In some situations, the overall time is important. For instance, if someone is asking a robot to get him a coffee, he do not really care if the robot plans motionless for a moment before getting the coffee. However, he would want to have its coffee as fast as possible.

In our work, instead of computing a policy offline, we adopted an online approach where the agent only explores belief states that can be reached from the current belief state. This is also called an agent-centered search approach (Koenig (2001)). This allows avoiding searching for a complete policy, thus avoiding a lot of computations. The advantage of such a method is that it can be applied on very large problems. It also allows having a model for decision making in large stochastic environments. In this section, we explain in detail how the algorithm for online decision-making works.

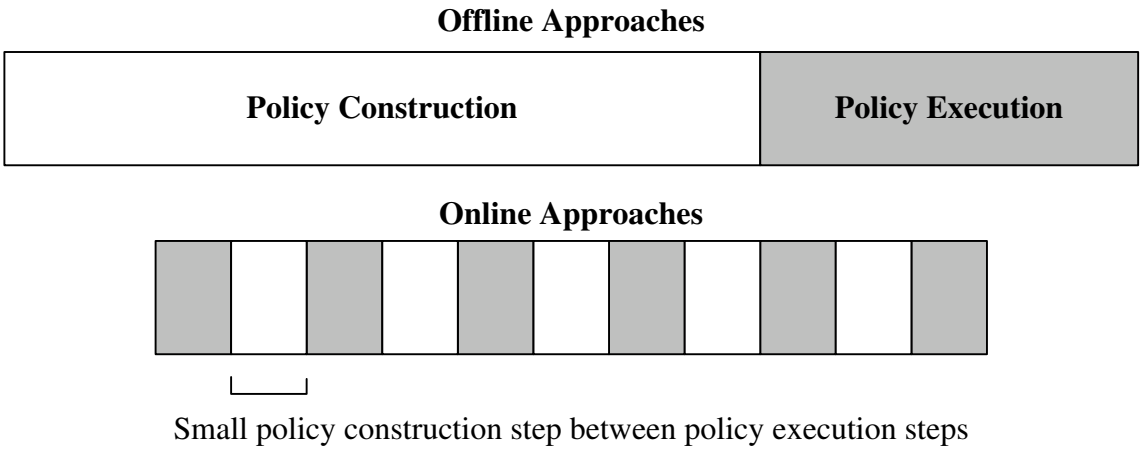


Figure 3.12: Comparison between offline and online approaches.

3.4.1 Belief State Value Approximation

In Section 3.1.1, we have described how it was possible to exactly compute the value of a belief state using dynamic programming (Equation 3.7). In this section, we instead explain how we estimate the value of a belief state for our online approach by using a look-ahead search. The main idea is to construct a tree where the nodes are belief states and where the branches are a combination of actions and observations (see Figure 3.13). During the execution, a new tree is calculated each time the agent has to choose an action. In our approach, we use a tree with estimated values at the fringe to estimate the value of the current belief state b_0. As the tree is expanded, the estimates become more accurate, which is guaranteed by the discount factor (Washington (1997)). In other words, the tree estimates the value of the current belief state b_0, by exploring all reachable belief states for a given horizon.

Formally, to describe how the tree is built, we have defined a new function that takes as parameters a belief state b and a depth d and returns an estimation of the value of b by performing a search of depth d. For the first call, d is initialized at D, the maximum depth allowed for the search.

$$\delta(b,d) = \begin{cases} U(b) & \text{, if } d = 0 \\ \max\limits_{a \in A} \left[R_B(b,a) + \gamma \sum\limits_{o \in \Omega} \left(Pr(o \mid b,a) \times \delta(\tau(b,a,o), d-1) \right) \right] & \text{, if } d > 0 \end{cases} \tag{3.35}$$

where $R_B(b,a)$ is computed using Equation 3.31, $Pr(o|b,a)$ using Equation 3.33 and $\tau(b,a,o)$ using Equation 3.34.

When $d = 0$, we are at the bottom of the search tree. In this situation, the value of a belief state is given by a utility function $U(b)$. This function gives an idea of the real

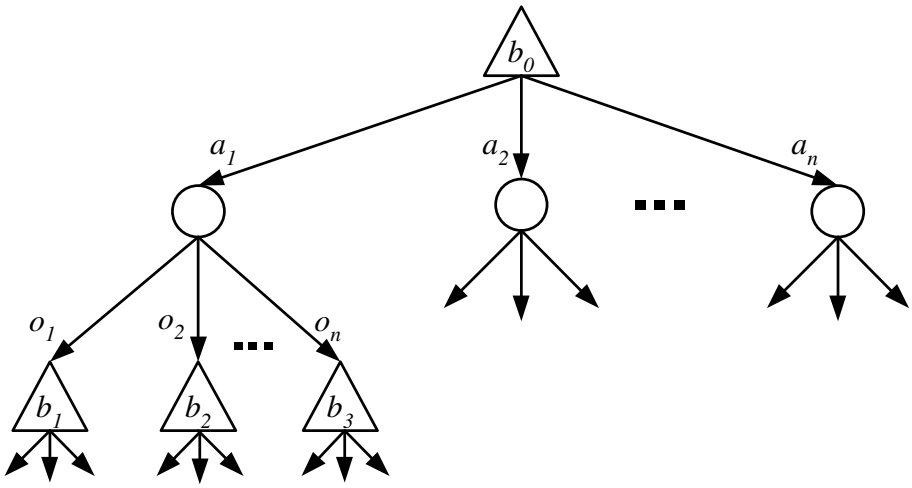


Figure 3.13: A search tree.

value of this belief state (if the function $U(b)$ was perfect, there would be no need for a search). This utility function has to be defined for each problem.

When $d > 0$, the value of a belief state at a depth of $D - d$ is simply the immediate reward for being in this belief state added to the maximum discounted reward of the subtrees underneath this belief state.

Finally, the agent's policy is dynamically calculated each time the agent has to choose an action. The action a to perform in a certain belief state b is simply the action that returns the best expected value evaluated with a search of depth D:

$$\pi(b, D) = \underset{a \in A}{\operatorname{argmax}} \left[R_B(b, a) + \gamma \sum_{o \in \Omega} (Pr(o \mid b, a) \times \delta(\tau(b, a, o), D - 1)) \right] \tag{3.36}$$

It is important to notice that this equation is not used to define the policy for all possible belief states. This function is called only with the current agent's belief state when the agent has to choose an action online.

3.4.2 RTBSS Algorithm

Now that we have formally given an overview of our approach, in this section we go into detail to describe our online POMDP algorithm, called RTBSS (Real-Time Belief State Search). Its name is due to the fact that it does a search in the belief state space online while satisfying a real-time constraint. Since it is an online algorithm, it must be applied each time the agent has to make a decision. RTBSS is used to construct the search tree, defined by Equation 3.35, and to find the best action the agent should perform in the current belief state (see Algorithm 3.2).

1: **Function** RTBSS(b, d) **returns** the estimated value of b.

Inputs: b: The current belief state.
d: The current depth.
Statics: D: The maximal search depth.
action: The best action.

2: **if** $d = 0$ **then**
3: **return** $U(b)$
4: **end if**
5: $actionList \leftarrow \text{SORT}(b, A)$
6: $max \leftarrow -\infty$
7: **for all** $a \in actionList$ **do**
8: $curReward \leftarrow R_B(b, a)$
9: $uBound \leftarrow curReward + \text{HEURISTIC}(b, a, d)$
10: **if** $uBound > max$ **then**
11: **for all** $o \in \Omega$ **do**
12: $curReward \leftarrow curReward + \gamma Pr(o|a, b)\text{RTBSS}(\tau(b, a, o), d - 1)$
13: **end for**
14: **if** $curReward > max$ **then**
15: $max \leftarrow curReward$
16: **if** $(d = D)$ **then**
17: $action \leftarrow a$
18: **end if**
19: **end if**
20: **end if**
21: **end for**
22: **return** max

Algorithm 3.2: The RTBSS algorithm.

The RTBSS algorithm does not completely develop the tree defined by the Equation 3.35. In order to speed up the search, our algorithm uses a "Branch and Bound" strategy to cut some useless sub-trees. To achieve that, the algorithm needs a lower bound on the maximal expected value. The bound is defined empirically during the search in order to have an accurate bound. During the search, the value of the bound is always the value of the best complete branch searched.

More precisely, here is how the bound is calculated and used. In any given belief state, the RTBSS algorithm checks if it would be possible to improve the value return for one of the actions in this belief state. This means that for each belief state visited during the search, the RTBSS algorithm has to explore at least one action. Since the algorithm only has to return the maximum value for all possible actions, it can prune some actions if these actions have no chance to improve the current maximum value.

At line 9 of the algorithm, the upper bound is evaluated as the immediate reward plus the heuristic value. The value returned by the HEURISTIC function has to be an upper bound. In other words, the HEURISTIC function returns the maximal value that the algorithm can find if it explores the current action up to the maximal depth D. Afterwards, at line 10, if the upper bound value is not greater than the current maximum value, then the action is pruned and the following subtree will not be expanded. The current maximum value serves as a lower bound, since the algorithm is guarantied to return at least this current maximum value. In order for the current maximum value to be a lower bound, the utility function ($U(b)$) at the leaves of the tree has to return a lower bound on the expected value of the belief state b.

A default lower bound heuristic might be set to the value we would get with a blind policy, that is the best policy consisting of always doing the same action until we reach a terminal state. Depending on the problem, this can often be improved by a smarter heuristic. Also, one can always set the lower bound to 0 for all belief states which has the effect of finding the best action for the finite horizon d if the rewards are positive. For the upper bound, the maximum possible discounted reward one can get in an environment is $R_{max}(s,a)/(1-\gamma)$, although setting the upper bound to this value will generally yield no pruning. For problems where only the terminal state receives a positive reward, a good heuristic might be to consider that we will always reach the terminal state from any belief state, therefore setting the upper bound to the reward one gets by doing an action that reach the terminal state. But again, depending on the problem, it might be possible to generate smarter heuristics. In some problems where there is no evident way of conceiving good heuristics, it is always possible to not use the action pruning by always returning $R_{max}(s,a)/(1-\gamma)$ or even infinity for all belief states. It is also not necessary to sort the actions when the pruning is not used (line 5 of Algorithm 3.2).

3.4.2.1 Detailed Description

Now, let's look more closely at how the algorithm works. The first call to the algorithm is $RTBSS(b,D)$, where b is the current belief state and D is the maximal depth for the search.

Lines 2-4 are executed when the algorithm reaches the maximal depth allowed for the search. At this point, the algorithm returns the utility value of the current belief state ($U(b)$). If the pruning is on, then this value has to be a lower bound on the expected value of b.

If the current node is not a leaf of the tree, the next step of the algorithm is to sort the actions according to the current belief state at line 5. This is done in order to try

the actions that are the most promising first because it generates more pruning early in the search tree. If we do not want to hinder the algorithm, we have to sort the actions really fast. This sort function is not mandatory, but if it is possible for a given problem to roughly sort the actions, then it helps the depth first algorithm to choose the most promising actions first, which improve the chance to find good bounds early during the search, which would help to prune more subtrees.

Afterwards, for each action a, we calculate at line 8 the immediate reward if the action a is executed in the current belief state b. Then, at line 9, we evaluate the upper bound as the immediate reward plus the heuristic value.

At line 10, the pruning condition is verified. Thus, if the upper bound is not greater than the maximum value, the action is not explored and the algorithm tries another action for the current belief state. If there is no pruning, then the reward of the current action a is calculated by summing the expected reward of each observation (lines 11-13). To achieve that, at line 12, the RTBSS algorithm is recursively called.

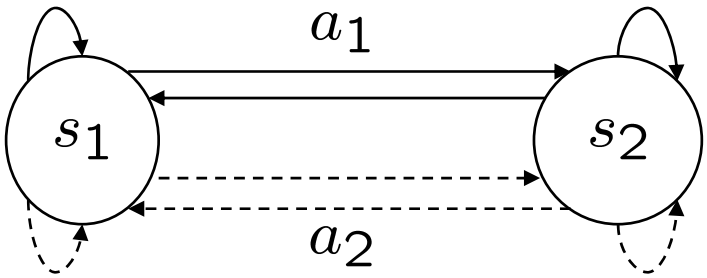
Lines 14-19 are used to record the best value for the current node among all the actions tried. It is also used to specify the policy by recording the action that returned the best value if we are at the root of the tree (lines 16-18). At the end at line 22, the maximal value for the current node is returned to the parent node.

To clearly illustrate how the RTBSS algorithm works, we show in the next section an example of its execution in a simple environment.

3.4.2.2 Example

This example uses the same environment that was used at the beginning of this chapter for an example of the value iteration algorithm. The example is represented again in Figure 3.14. For the purpose of this example, let's suppose that the current belief state of the agent is $b = [0.1, 0.9]$ and that the maximal depth of the search is $D = 2$. For this problem, the best action in state s_1 is a_1 and in state s_2 is a_2. Therefore, the SORT function returns the action a_1 first if the most probable state is the state s_1 and returns the action a_2 first if the most probable state is the state s_2. If both states are equally probable, the SORT function uniformly randomly chooses between a_1 and a_2. Also, let's suppose that the HEURISTIC function considers that it is possible to get the maximal reward (R_{max}) after each action. R_{max} is the maximal reward for all states and all actions of the underlying MDP.

$$\text{HEURISTIC}(b, a, d) = \sum_{i=1}^{d} \gamma^i R_{max} \tag{3.37}$$



States: $\{s_1, s_2\}$
Actions: $\{a_1, a_2\}$
Observations: $\{o_1, o_2\}$
Discount factor: $\gamma = 0.9$

Reward Function

- $R(s_1, a_1) = 2$
- $R(s_1, a_2) = 1$
- $R(s_2, a_1) = 1$
- $R(s_2, a_2) = 3$

Transition Function

- $T(s_1, a_1, s_1) = 0.3$
- $T(s_1, a_1, s_2) = 0.7$
- $T(s_1, a_2, s_1) = 0.1$
- $T(s_1, a_2, s_2) = 0.9$
- $T(s_2, a_1, s_1) = 0.6$
- $T(s_2, a_1, s_2) = 0.4$
- $T(s_2, a_2, s_1) = 0.8$
- $T(s_2, a_2, s_2) = 0.2$

Observation Function

- $O(s_1, a_1, o_1) = 0.9$
- $O(s_1, a_1, o_2) = 0.1$
- $O(s_1, a_2, o_1) = 0.9$
- $O(s_1, a_2, o_2) = 0.1$
- $O(s_2, a_1, o_1) = 0.5$
- $O(s_2, a_1, o_2) = 0.5$
- $O(s_2, a_2, o_1) = 0.5$
- $O(s_2, a_2, o_2) = 0.5$

Figure 3.14: POMDP Example.

Finally, let's suppose that the utility function at the leaves of the tree is:

$$U(b) = \max_{a \in A} R_B(b, a) \tag{3.38}$$

Now, let's see how the RTBSS algorithm works on this problem. First, the RTBSS algorithm starts with the current belief state at the root of the search tree, as shown in Figure 3.15. The first action tried is the action a_2, which is the first action returned by the SORT function, because the state s_2 is the most probable state in the initial belief state $[0.1, 0.9]$. Afterwards, the algorithm has to explore an observation. The choice of an observation is not important because all observations have to be explored. For the purpose of this example, we have always chosen o_1 before o_2. Therefore, the next step of the algorithm is to consider the possibility of observing o_1, and thus the search ends

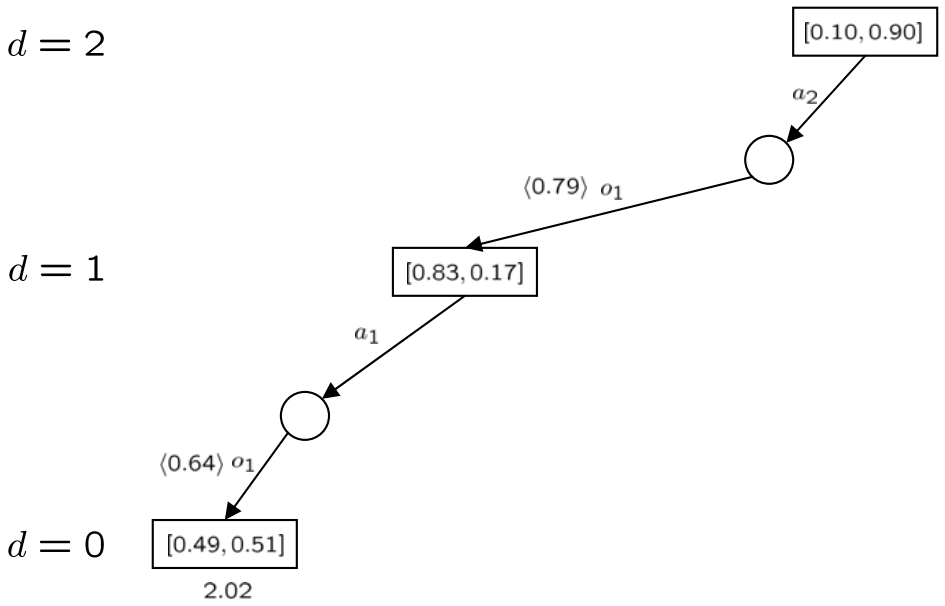


Figure 3.15: Example of an execution of the RTBSS algorithm (Step 1).

up in belief state $[0.83, 0.17]$.

$$\begin{aligned}
\tau([0.1, 0.9], a_2, o_1) &= \eta[O(s_1, a_2, o_1)(T(s_1, a_2, s_1)b(s_1) + T(s_2, a_2, s_1)b(s_2)), \\
&\quad O(s_2, a_2, o_1)(T(s_1, a_2, s_2)b(s_1) + T(s_2, a_2, s_2)b(s_2))] \\
&= \eta[0.9(0.1 \times 0.1 + 0.8 \times 0.9), 0.5(0.9 \times 0.1 + 0.2 \times 0.9)] \\
&= \eta[0.657, 0.135] \\
&= [0.83, 0.17] \qquad (3.39)
\end{aligned}$$

In the belief state $[0.83, 0.17]$, the algorithm tries the action a_1, because the state s_1 is the most probable state. Then, the algorithm explores the possibility of observing o_1 and it calculates the resulting belief state $[0.49, 0.51]$, using a similar equation as the Equation 3.39. At this point, the algorithm has attained the limit depth, thus it returns $U(b) = 2.02$.

$$\begin{aligned}
U(b) &= \max[R_B([0.49, 0.51], a_1), R_B([0.49, 0.51], a_2)] \\
&= \max[(R(s_1, a_1)b(s_1) + R(s_2, a_1)b(s_2)), (R(s_1, a_2)b(s_1) + R(s_2, a_2)b(s_2))] \\
&= \max[(2 \times 0.49 + 1 \times 0.51), (1 \times 0.49 + 3 \times 0.51)] \\
&= \max[1.49, 2.02] \\
&= 2.02 \qquad (3.40)
\end{aligned}$$

Now that the algorithm has reached the maximal depth, it backtracks and it explores the other observation, as presented in Figure 3.16. The belief state reached when

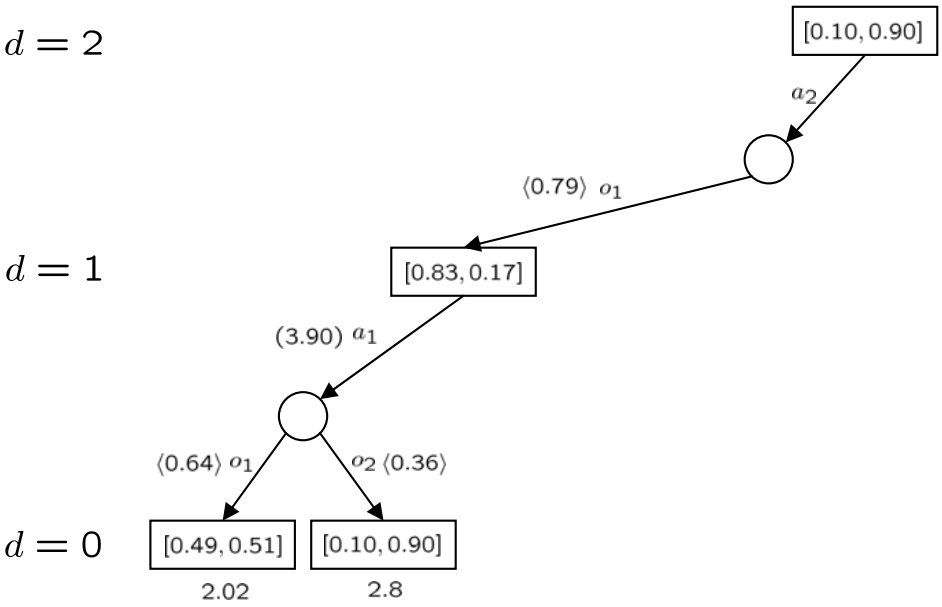


Figure 3.16: Example of an execution of the RTBSS algorithm (Step 2).

considering the observation o_2 is $[0.10, 0.90]$ (calculation similar to Equation 3.39). At this moment, the action a_1 has been completely explored, thus we can calculate its value:

$$\begin{aligned} curReward &= R_B([0.83, 0.17], a_1) + \\ &\quad \gamma \sum_{o \in O} Pr(o|a_1, [0.83, 0.17]) \text{RTBSS}(\tau([0.83, 0.17], a_1, o), 0) \\ &= 2 * 0.83 + 1 * 0.17 + 0.9 * (0.64 * 2.02 + 0.36 * 2.8) \\ &= 3.90 \end{aligned} \tag{3.41}$$

Afterwards, the RTBSS algorithm calculates the upper bound for the action a_2 with the HEURISTIC function, which returns the value: 4.04.

$$\begin{aligned} uBound &= R_B([0.83, 0.17], a_2) + \text{HEURISTIC}([0.83, 0.17], a_2, 1) \\ &= (R(s_1, a_2)b(s_1) + R(s_2, a_2)b(s_2)) + \sum_{i=1}^{1} 0.9^i \times 3 \\ &= (1 \times 0.83 + 3 \times 0.17) + 0.9 \times 3 \\ &= 4.04 \end{aligned} \tag{3.42}$$

Since this last value is greater than the value for the action a_1 (3.90), the RTBSS algorithm has to explore the action a_2, which is shown in Figure 3.17. After the same calculations as the ones done for the action a_1, the algorithm finds that the action a_2 has a value of 3.64. The estimated value for the belief state $[0.83, 0.17]$ is then 3.90, which is the greatest value among all possible actions.

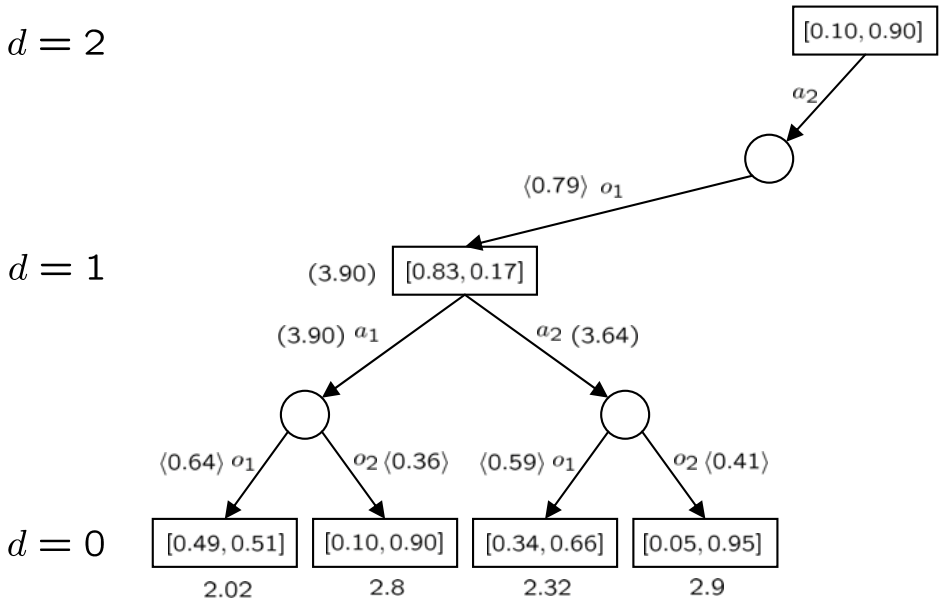


Figure 3.17: Example of an execution of the RTBSS algorithm (Step 3).

Then, the RTBSS algorithm has to backtrack to explore the observation o_2 at the top of the tree has shown in Figure 3.18. After exploring the action a_2 for the belief state $[0.35, 0.65]$, the algorithm then evaluates if it can prune the action a_1. It this case, the algorithm can prune the action a_1 because the value of the upper bound for the action a_1 (4.05) is not greater than the value of action a_2 (4.05).

The value for the belief state $[0.35, 0.65]$ is thus 4.05, as shown in Figure 3.19. The algorithm then backtracks all the way to the top of the tree and it evaluates if it can prune the action a_1. In this situation it can because the upper bound for the action a_1 (6.23) is not greater than the value of action a_2 (6.34).

$$\begin{aligned} uBound &= R_B([0.10, 0.90], a_1) + \textsc{Heuristic}([0.10, 0.90], a_1, 2) \\ &= (R(s_1, a_1)b(s_1) + R(s_2, a_1)b(s_2)) + \sum_{i=1}^{2} 0.9^i \times 3 \\ &= (2 \times 0.10 + 1 \times 0.90) + 0.9 \times 3 + 0.9^2 \times 3 \\ &= 6.23 \end{aligned} \tag{3.43}$$

The algorithm has thus finished and the value, at the root of the tree, of the current belief state [0.1,0.9] is 6.34. In fact, the algorithm does not return the value of the current belief state, but it returns the best action found, which is the action a_2 in this example.

As we can see in this example, the RTBSS algorithm has estimated the expected rewards of the agent's current belief state by doing a limited search of depth 2. However,

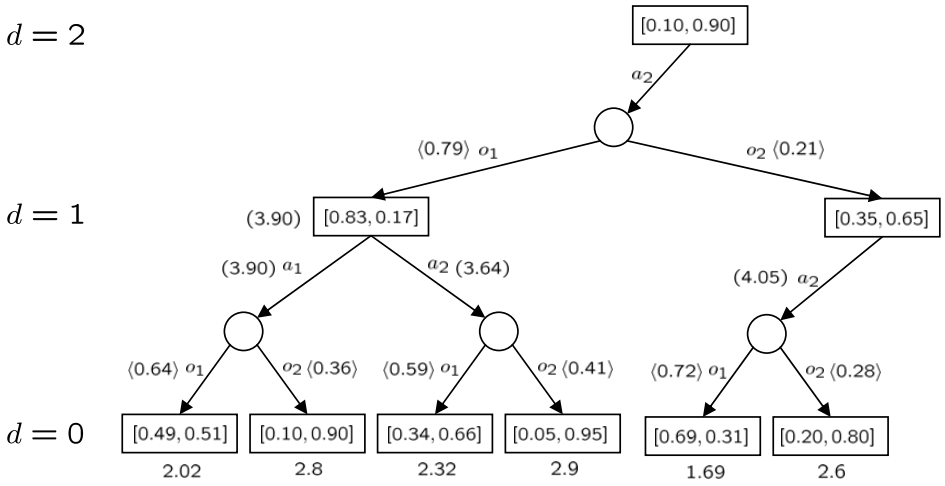


Figure 3.18: Example of an execution of the RTBSS algorithm (Step 4).

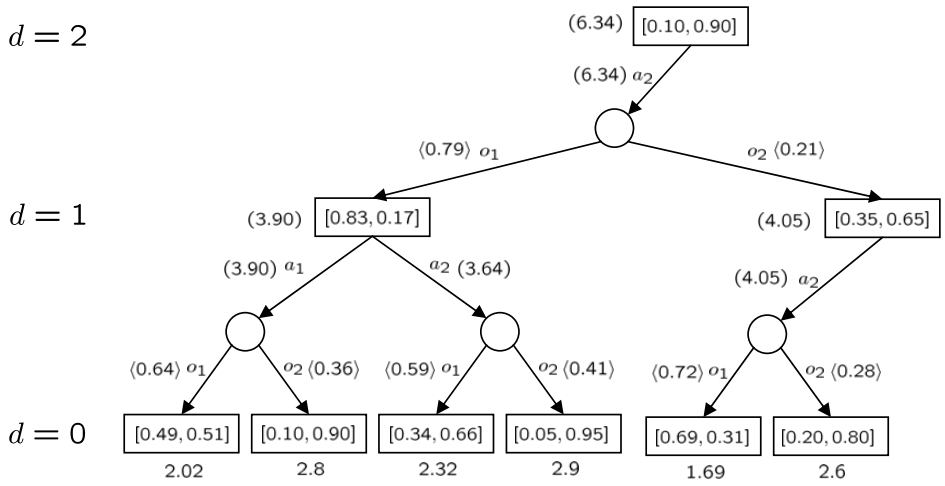


Figure 3.19: Example of an execution of the RTBSS algorithm (Step 5).

the algorithm has explored less than half of the search tree, because two actions have been pruned.

Now that the RTBSS algorithm has been presented in detail, in the next section, we evaluate the complexity of this algorithm.

3.4.2.3 Complexity

With the RTBSS algorithm the agent finds at each turn the action that has the maximal expected value up to a certain horizon of D (the maximal depth of the search). As a matter of fact, the performance of the algorithm strongly depends on the depth of the search. In the worst case, the number of nodes generated for the search is: $O((|A||\Omega|)^D)$, where $|A|$ is the number of actions and $|\Omega|$ the number of observations. This is if no pruning is done, consequently with a good pruning heuristic, it is possible to do much better in practice.

Therefore, our algorithm is efficient if the number of actions and observations is kept small. Otherwise, the search cannot be done deeply enough since the branching factor becomes too big. If there are many observations, it is possible to use a sampling of the observations in order to explore only the most probable observations (McAllester and Singh (1999)). However, in our work, we only considered exploring all the observations in order to stay optimal in our limited horizon of D.

For each node, we also have to sort the actions according to the current belief state, which has a complexity of $O(|A|log(|A|))$. Sorting the actions adds some complexity, but it also helps the pruning process to cut more subtrees. If it is possible to efficiently sort the actions according to their expected efficiency, then it is normally worth it to do so, because it removes a lot of calculations if more subtrees are pruned.

The belief state also has to be updated at each node. A complete belief state update has a complexity of $O(|S|^2)$, where $|S|$ is the number of states. However, the dependance on $|S|$ can be improved by associating each node in the search tree with a more efficiently computable simplified belief state approximating the true belief state at that node (McAllester and Singh (1999)). In sections 3.1.5 and 3.3, we have presented some factorization techniques that could be used to efficiently compute good approximations of the belief state.

Finally, if we consider the complexity of the pruning heuristic function (C_H) and the utility function used at the leaves of the tree (C_U), then the complexity of the RTBSS algorithm in the worst case is: $O((|A||\Omega|)^D\ |A|log(|A|)\ |S|^2\ C_H C_U)$. It is important to notice that this is the worst case complexity, in practice the RTBSS algorithm is much better because of the pruning and the factorization.

In the next section, we present the theoretical bound showing that our belief state value estimate remains within some bounds of the optimal value.

3.4.2.4 Error Bounds

For any belief state b and depth d, $RTBSS(b,d)$ calculates an estimate of the value of b. In this section, we show that the error between $RTBSS(b,d)$ and the optimal value function V^* is bounded. As mentioned before, RTBSS explores all reachable belief states, up to a certain depth, from the current belief state.

Lemma 3.1. *For any belief state b, any depth d and when $U(b) = R_B(b)$ ($R_B(b)$ is the reward for being in the belief state b), the error of the RTBSS algorithm is bounded by:*

$$|RTBSS(b,d) - V^*(b)| \leq \gamma^d \max(|R_{max} - V_{min}|, |R_{min} - V_{max}|)$$

Where, R_{max} and R_{min} are the maximum and the minimum possible rewards respectively, and:

$$V_{max} = \sum_{i=0}^{\infty} \gamma^i R_{max} = \frac{R_{max}}{1-\gamma} \qquad V_{min} = \sum_{i=0}^{\infty} \gamma^i R_{min} = \frac{R_{min}}{1-\gamma} \tag{3.44}$$

Proof. If $U(b) = R_B(b)$, RTBSS is equivalent to using a dynamic programming approach to obtain the optimal value for a horizon of d (V_d^*), thus: $RTBSS(b,d) = V_d^*(b)$. So,

$$\begin{aligned}
|RTBSS(b,d) - V^*(b)| = {} & |RTBSS(b,d) - V_d^*(b)| + \\
& |V_d^*(b) - V^*(b)| && \text{divide in two parts} \\
= {} & |V_d^*(b) - V^*(b)| && RTBSS(b,d) = V_d^*(b) \\
\leq {} & \gamma^d |V_0^*(b) - V^*(b)| && \text{Bertsekas (2001)} \\
= {} & \gamma^d |R_B(b) - V^*(b)| && V_0^*(b) = R_B(b) \\
\leq {} & \gamma^d \max(|R_{\max} - V_{\min}|, |R_{\min} - V_{\max}|) && \text{see text}
\end{aligned}$$

The last line is due to the fact that the bound is maximized when $R_B(b) = R_{\max}$ and $V^*(b) = V_{\min}$ or when $R_B(b) = R_{\min}$ and $V^*(b) = V_{\max}$. □

This value of $V_{\max}$ is valid only if the agent can get $R_{\max}$ each turn. If we are in a goal searching problem, where the agent receives $R_{\min}$ until it reaches the goal where it gets $R_{\max}$, then:

$$V_{\max} = \gamma^G R_{\max} + \sum_{i=0}^{G-1} \gamma^i R_{\min} \tag{3.45}$$

Where G represents the shortest distance to the goal. Even in these settings, $V_{\min}$ does not change, since the agent may never find the goal.

Theorem 3.1. *In the context of a discounted reward problem, the error of the RTBSS algorithm gets smaller as the agent acts in the environment. More precisely, the error at a certain time t_1 is bounded by:*

$$\left|\widehat{V}^{t_1}(b_{t_1}) - V^{*,t_1}(b_{t_1})\right| \leq \gamma^{t_1+d} \max(\left|R_{max} - V_{min}\right|, \left|R_{min} - V_{max}\right|)$$

Where,

$$V^{*,t_1}(b_{t_1}) = \gamma^{t_1} V^*(b_{t_1}) + \sum_{i=0}^{t_1-1} \gamma^i r_i \qquad \widehat{V}^{t_1}(b_{t_1}) = \gamma^{t_1} RTBSS(b_{t_1}, d) + \sum_{i=0}^{t_1-1} \gamma^i r_i \quad (3.46)$$

Proof. When the rewards are discounted, the total reward of the agent is defined as: $R = \sum_{t=0}^{\infty} \gamma^t r_t$, where r_t is the reward received at time t. Consequently, the optimal and the estimated expected total reward at a certain time t_1 are:

Therefore, the error bound at a certain time t_1 is:

$$\begin{aligned}
\left|\widehat{V}^{t_1}(b_{t_1}) - V^{*,t_1}(b_{t_1})\right| &= \left|\gamma^{t_1} RTBSS(b_{t_1}, d) + \sum_{i=0}^{t_1-1} \gamma^i r_i - \right. \\
&\quad \left. \left(\gamma^{t_1} V^*(b_{t_1}) + \sum_{i=0}^{t_1-1} \gamma^i r_i\right)\right| && \text{Substitution} \\
&= \gamma^{t_1} \left|RTBSS(b_{t_1}, d) - V^*(b_{t_1})\right| && \text{Simplification} \\
&\leq \gamma^{t_1+d} \max(\left|R_{\max} - V_{\min}\right|, \left|R_{\min} - V_{\max}\right|) && \text{Lemma 3.1} \quad \square
\end{aligned}$$

Theorem 3.1 shows that RTBSS gets closer to the optimal as the agent progresses, because of the discount factor. Moreover, Bertsekas and Tsitsiklis (1996) showed that if the estimated value is sufficiently close to the optimal value, then the greedy policy based on the estimated value must be an optimal policy. More precisely, they showed that if $|\widehat{V} - V^*| = \epsilon$ and if π is a greedy policy based on $\widehat{V}$, then:

$$\left|\widehat{V}^{\pi} - V^*\right| \leq \frac{2\gamma\epsilon}{1-\gamma} \quad (3.47)$$

Finally, according to the same authors, there exists some ϵ_0 such that if $\epsilon < \epsilon_0$, then π is an optimal policy. In our case, since RTBSS can be seen as a greedy policy and that the error of RTBSS is reduced as the agent acts in the environment, then the policy also gets closer to the optimal policy, because of the discount factor.

3.5 Hybrid Approaches

Even though the algorithm we propose is an online approach, which has unique properties over offline approaches, we can see that it is possible to combine this online approach with existing offline approaches, if we do not need those properties for a specific problem. For example, in the RoboCupRescue, it would not be possible to use hybrid approaches, because the state space is too big and we need the agents to be efficient in unknown cities. However, in simpler problems, it might be possible to use hybrid approaches. The three offline algorithms that we have used are: the Q_{MDP} algorithm (section 3.1.3.1), the PBVI algorithm (section 3.1.3.1) and the RTDP-BEL algorithm (section 3.1.4.1). Using these three offline algorithms with our online RTBSS algorithm, we propose three new hybrid approaches that combine offline and online computation: RTBSS-QMDP, RTBSS-PBVI-QMDP and RTDPBSS.

3.5.1 RTBSS-QMDP

The RTBSS-QMDP approach uses the Q_{MDP} algorithm to compute an approximation of the value function offline. Then, RTBSS-QMDP algorithm uses the Q_{MDP} approximation as the leaf value function in the search tree of the RTBSS algorithm. In other words, the only modification to the RTBSS algorithm is that the $U(b)$ function at the leaves of the tree returns $V_{MDP}(b)$. Since we know that $V_{MDP}(b)$ introduces a certain error ϵ on the value of the belief state b, the advantage of using the RTBSS algorithm in combination with Q_{MDP} is to reduce this error. Since the RTBSS algorithm returns an exact value up to depth d, it will multiply the error ϵ by a factor of γ^d, where d is the depth of the search done by RTBSS. Consequently, the deeper is the search, the less is the error of the algorithm (see proof in section 3.5.4). This improved evaluation of a belief state should generate a better policy for the agent.

The advantage of using the Q_{MDP} algorithm over the other offline algorithms is that it rapidly gives a good estimate of the belief state values, even in large state spaces. However, the inconvenient is that by using Q_{MDP} as the RTBSS leaf values, we cannot do any action pruning in the tree since Q_{MDP} is an upper bound on the exact value function V^*. We recall that the action pruning can only be done if the leaf value is a lower bound on the exact value function $V^*(b)$ and if the pruning HEURISTIC function returns an upper bound on the exact value $V^*(b)$. Nevertheless, the experiments show that only a search depth of 3 or 4 can give very good results and improve the results of both algorithms when used alone.

3.5.2 RTBSS-PBVI-QMDP

In order to reintroduce the action pruning into the RTBSS-QMDP algorithm, we propose the RTBSS-PBVI-QMDP algorithm, where PBVI is used to compute a lower bound on the exact value function $V^*(b)$ and Q_{MDP} is used to compute an upper bound on the exact value function $V^*(b)$. Therefore, the only modification to the RTBSS algorithm is that the $U(b)$ function at the leaves of the tree returns $V_{PBVI}(b)$ and the pruning HEURISTIC function returns $V_{MDP}(b)$. What we want to show with this approach is that it might be possible to compute a quick policy with PBVI, then use this policy as a lower bound and use the Q_{MDP} algorithm to do some action pruning in the search tree of the RTBSS algorithm. By doing so, it might be possible to search deeper and thus reduce the error introduced by the approximate PBVI policy used at the leaves of the search tree.

3.5.3 RTDPBSS

The RTDPBSS algorithm is basically a combination of the RTDP-BEL algorithm and the RTBSS algorithm, without the action pruning. Instead of choosing an action based directly on the value of the approximation function, as done in RTDP-BEL, the RTDPBSS algorithm adds a depth search of depth d, where the leaf values are given by the approximate value function of the RTDP-BEL approach, but where parent nodes compute the exact values with the immediate rewards. This approach has the advantage of reducing the error of the approximate value function by a factor of γ^d. The RTDP-BEL approximate value function will always have an error, even after convergence, because of the belief state discretization. Moreover, by searching deeper, the RTDPBSS algorithm improves the approximate value function with a much more precise belief state value.

The only inconvenient of this approach is that it necessitates more time online to choose actions than in RTDP-BEL, since we do a full depth search. Therefore, the RTDPBSS algorithm requires more time to run a certain number of simulations during the learning phase. However, the results show that the RTDPBSS algorithm tends to converge much more rapidly toward a very good policy than the RTDP-BEL algorithm.

In the next section, we present a proof that the hybrid approach can always reduce the error of an approximate offline approach.

3.5.4 Proof of the Usefulness of a Hybrid Approach

The intuition behind a hybrid approach is that for any approximate value function $V(b)$ with error bounds $[\inf_b \epsilon(b), \sup_b \epsilon(b)]$ (where $\epsilon(b) = |V(b) - V^*(b)|$) on the exact value function $V^*(b)$, we can always get a lower error bound by computing an exact value function up to a certain horizon d and use the approximation $V(b)$ at the last horizon. This should in fact multiply the error bound of $V(b)$ by a factor of γ^d, as demonstrated in the following proof.

Theorem 3.2. *For any algorithm that computes an approximate value function $V(b)$ with error function $\epsilon(b)$, defining the error on the exact value function $V^*(b)$ such that $\epsilon(b) = |V(b) - V^*(b)|$, the error bounds of the RTBSS algorithm doing a search of depth d with $U(b) = V(b)$ will have $\sup_b \epsilon_{RTBSS}(b) <= \gamma^d \sup_b \epsilon(b)$ and $\inf_b \epsilon_{RTBSS}(b) >= \gamma^d \inf_b \epsilon(b)$ on the exact value function $V^*(b)$.*

Proof. The following proof uses many terms, which are defined as:

- $P(b^d) = P(o^0|b^0, a^0_{\max})P(o^1|b^1, a^1_{\max}) \cdots P(o^{d-1}|b^{d-1}, a^{d-1}_{\max})$
- $a^i_{\max} = \text{argmax}_a R(b^i, a) + \gamma \sum_{o^i} P(o^i|b^i, a) V_{RTBSS(d-i-1)}(b^{i+1})$
- $b^d = \tau(b^{d-1}, a^{d-1}_{\max}, o^{d-1})$
- $s(b) = -1$ when $V(b) < V^*(b)$
- $s(b) = 1$ when $V(b) \geq V^*(b)$
- $\epsilon_{\max} = \sup_b \epsilon(b)$
- $\epsilon_{\min} = \inf_b \epsilon(b)$

Where, $P(b^d)$ represents the probability of reaching the leaf belief state b^d at search depth d. In the search tree, $P(b^d)$ corresponds to the product of the observations probability encountered by following the path from the root b to the leaf b^d.

Proof of Theorem 3.2: if $U(b) = V(b)$, the error of the RTBSS algorithm ($\epsilon_{RTBSS(d)}(b)$) is bounded by:

$$
\begin{aligned}
\epsilon_{RTBSS(d)}(b) &= |V_{RTBSS(d)}(b) - V^*(b)| && \text{Definition of the error} \\
&= |V^*_{d-1}(b) + \gamma^d \sum P(b^d) U(b^d) - V^*(b)| && \text{Definition of } V_{RTBSS(d)}(b) \\
&= |V^*_{d-1}(b) + \gamma^d \sum P(b^d) V(b^d) - V^*(b)| && \text{Definition of } U(b) \\
&= |V^*_{d-1}(b) + \gamma^d \sum P(b^d)(V^*(b^d) + \\
&\quad s(b^d)\epsilon(b^d)) - V^*(b)| && \text{Definition of } V(b) \\
&= |V^*_{d-1}(b) + \gamma^d \sum (P(b^d)V^*(b^d) + \\
&\quad P(b^d)s(b^d)\epsilon(b^d)) - V^*(b)| && \text{Distributivity} \\
&= |V^*_{d-1}(b) + \gamma^d \sum (P(b^d)V^*(b^d)) + \\
&\quad \gamma^d \sum (P(b^d)s(b^d)\epsilon(b^d)) - V^*(b)| && \text{Split sum} \\
&= |V^*(b) + \gamma^d \sum (P(b^d)s(b^d)\epsilon(b^d)) - V^*(b)| && \text{Definition of } V^*(b) \\
&= |\gamma^d \sum (P(b^d)s(b^d)\epsilon(b^d))| && \text{Simplification}
\end{aligned}
$$

In the worst case:

$$
\begin{aligned}
|\gamma^d \sum (P(b^d)s(b^d)\epsilon(b^d))| &\leq |\gamma^d \sum (P(b^d)s(b^d)\epsilon_{\max})| && \epsilon(b^d) \leq \epsilon_{\max} \\
&= |\gamma^d \epsilon_{\max} \sum (P(b^d)s(b^d))| && \text{Simplification} \\
&\leq |\gamma^d \epsilon_{\max}| && \sum (P(b^d)s(b^d)) \leq 1 \\
&= \gamma^d \epsilon_{\max}
\end{aligned}
$$

In the best case:

$$
\begin{aligned}
|\gamma^d \sum (P(b^d)s(b^d)\epsilon(b^d))| &\geq |\gamma^d \sum (P(b^d)s(b^d)\epsilon_{\min})| && \epsilon(b^d) \geq \epsilon_{\min} \\
&= |\gamma^d \epsilon_{\min} \sum (P(b^d)s(b^d))| && \text{Simplification} \\
&\geq |-\gamma^d \epsilon_{\min}| && \sum (P(b^d)s(b^d)) \geq -1 \\
&= \gamma^d \epsilon_{\min}
\end{aligned}
$$

Consequently, $\epsilon_{RTBSS(d)}(b) \in [\gamma^d \epsilon_{\min}, \gamma^d \epsilon_{\max}]$ □

This theorem has a pretty strong implication. It means that for any offline algorithm computing an approximate value function with error bounds $[\epsilon_{\min}, \epsilon_{\max}]$, we can always get a better approximation by combining this offline algorithm with our RTBSS algorithm online. In the next section, we present results for the hybrid approaches that confirm this claim.

3.6 Experimentations

As mentioned before, the conception of the RTBSS algorithm has been motivated by the RoboCupRescue simulation environment. However, it is a general algorithm that can be applied to any POMDP environment. In this section, we present the results of RTBSS on two problems found in the POMDP literature: *Tag* (Pineau et al. (2003)) and *RockSample* (Smith and Simmons (2004)). Our primary goal was to develop an approach applicable in the RoboCupRescue simulation environment, but it is hard to evaluate the performances of our approach in such a complex environment. This is why we first compare our approach with state of the art POMDP algorithms in smaller environments.

It is important to notice that the *Tag* and *RockSample* problems do not totally do justice to our algorithm because we can affirm that it is better to apply an offline algorithm in order to have a better solution, even if we need to wait a few hours or days before having the solution. An inconvenient with offline approaches is that each time the environment slightly changes, we have to wait another few hours or days. Thus, if the environment is not exactly the same from one execution to another, the offline approaches become really expensive. We have used those two environments because they were popular and because they enabled us to compare the RTBSS's performances with the best offline performances. Our results show that even if they have all the necessary time, offline approaches have difficulty catching up with the performances obtained by our RTBSS algorithm on big environments.

In all the results presented in this thesis, if there is no citation beside the name of the algorithm in the tables, then the results have been obtained with our own implementation of the algorithm. Therefore, the performances may slightly differ from other implementations.

3.6.1 *Tag*

We tested our algorithm in *Tag*, introduced for the first time by Pineau et al. (2003). This environment has also been used recently in (Poupart and Boutilier (2003a); Vlassis and Spaan (2004); Pineau (2004); Spaan and Vlassis (2004); Smith and Simmons (2004); Braziunas and Boutilier (2004); Spaan and Vlassis (2005); Smith and Simmons (2005)). For this environment, we need to use an approximate POMDP algorithm, because of its medium size of 870 states. In this environment, we can compare our approach with many other recent approaches. Such a comparison would not be possible in much bigger POMDPs because the majority of the other approaches would not be applicable as we will see later. The results show that our algorithm can obtain good results with a small amount of computation time.

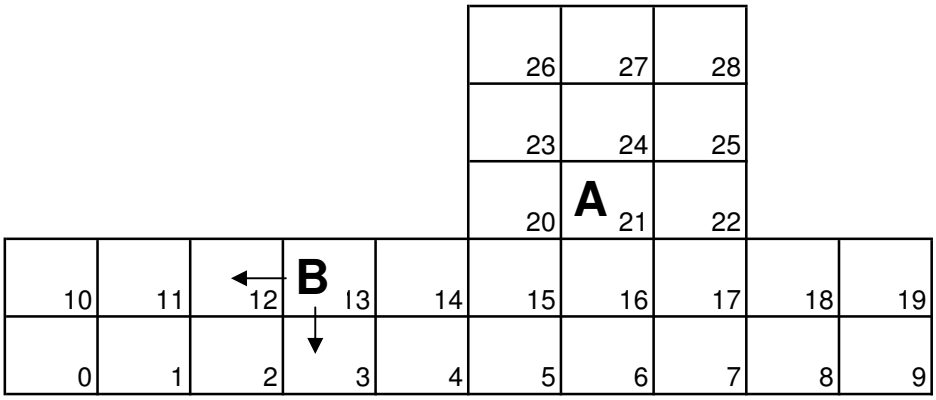


Figure 3.20: The *Tag* problem.

3.6.1.1 Description of the *Tag* Environment

The *Tag* environment consists in an agent A that has to catch another agent B. The environment configuration is presented in Figure 3.20. The agent A cannot see the agent B, except if both agents are at the same position. To find B, the agent A has to move in the environment where it thinks B should be. If it finds B, it can tag it and its goal has been achieved.

With a factored representation, a state in this problem is described with two variables: $X_1 = \{0 \dots 28\}$ is the state of the agent A and $X_2 = \{0 \dots 28 \text{ and tagged}\}$ is the state of agent B. Agent A can choose between 5 actions: *North*, *South*, *East*, *West* and *Tag*. The initial position of the agents is randomly chosen (without the trivial case where the two agents start at the same position). In this problem, the transition and observation functions are deterministic, which means that there is no uncertainty about the actions and the observations of agent A. However, the environment stays highly partially observable because the agent A does not know where the agent B is.

Agent B has a perfect vision and a simple behavior. It moves away from A with a probability of 0.8 and it stays at the same position with a probability of 0.2. For example, if B can do two actions to get away from A (see Figure 3.20), then each of these two actions would have a probability of 0.4, and always 0.2 to stay in place. Thus, even if A is practically blind, it can have a good idea of B's position, because it knows that it always tries to go away from it. Thus, B has a good chance to be in the opposite corner.

Notice that the agent A receives a reward of -1 for each move action it does in the environment, a reward of $+10$ if the *Tag* action succeeds and a reward of -10 if the *Tag* action fails. The agent tries to maximize the discounted sum of its rewards with a discount factor $\gamma = 0.95$.

To apply our RTBSS algorithm, some functions have to be defined. The first function we need is the HEURISTIC function used in the pruning condition to evaluate the best utility value possible from each belief state up to the maximal depth D. For *Tag*, this utility function simply considers that it will be possible, with the highest probability of any of the states, to tag the agent B at each action until the maximum depth D. This gives the maximum reward agent A could get. By doing so, we are sure that the HEURISTIC function always overestimates the true value.

The second function we need is the $U(b)$ function which gives an estimation of the value of the belief state b at the fringe of the search tree. For this function, we used the immediate reward $R(b)$.

The last function we need is the SORT function that sorts the actions. For Tag, this function simply sorts the move actions according to the most probable position of the agent B. The move action that goes toward the most probable position of B is tried first. For other problems, these functions have to be defined considering the problem's characteristics.

3.6.1.2 Results

On the Tag problem, the RTBSS algorithm obtains very good results with no offline computation time and with less then a second online. In Table 3.2, we can see that the RTBSS algorithm obtains the best results. This can be explained by the fact that, at a depth of 12, the RTBSS algorithm can explore all possible positions for agent B, thus it can have a good idea of the best action to take in order to catch agent B.

The RTBSS-PBVI-QMDP hybrid algorithm also obtains good results. However, it is important to notice that the result presented is only for the starting position of 0 for the agent A. This is because we have only calculated the policy of PBVI for this starting position. Since PBVI needs the starting belief state, we have to generate one policy for each starting position for the agent A. For our tests, we generated one policy in 10 hours for the starting position 0. This policy obtained an average reward of -10.61. On the other hand, the hybrid approach obtained an average reward of -5.40. This shows that the hybrid approach is quite efficient to improve a weak offline policy. Another example of that is the RTBSS-QMDP approach that obtained a much better result (-6.11) than the Q_{MDP} approach alone.

The RTDP-BEL and the RTDPBSS algorithms are not good on Tag. They were not able to converge, because they were not guided enough. The reward can be quite far in Tag, and they had some trouble finding the other agent. These methods act almost randomly at the beginning, thus they were not able to catch the agent B often enough to learn how to catch the agent B. These methods need a lot of memory to store all the

Method	Reward	Offline Time (s)	Online Time (s)
Q_{MDP}	-16,75	0.875	-
RTDP-BEL	-12.15	3645	0,001
RTDPBSS	**-9.60**	**24540**	**0,556**
PBVI Pineau et al. (2003)	-9,18	180880	-
BBSLS Braziunas and Boutilier (2004)	$\backsim$ -8.3	$\backsim$ 100000	-
BPI Poupart (2005)	-6,65	250	-
HSVI1 Smith and Simmons (2004)	-6,37	10113	-
HSVI2 Smith and Simmons (2005)	-6,36	24	-
PERSEUS Spaan and Vlassis (2004)	-6,17	1670	-
PBVI Pineau (2004)	$\backsim$ -6,12	$\backsim$ 900000	-
RTBSS-QMDP(5)	**-6.11**	**0.875**	**0,311**
RTBSS-PBVI-QMDP(4)	**-5.40***	**36000**	**0,220**
RTBSS(12)	**-5.03**	**0**	**0,750**

Table 3.2: Comparison of our approach on the *Tag* problem. For RTBSS, the reward and the computation time are averages over 5000 simulations. The number between parenthesis beside the RTBSS based methods is the depth of the search D used to obtain these results. *: For this result, the agent A always starts in position 0.

belief states encountered, thus they often ran out of memory, even with one Gigabyte of memory.

Figures 3.21 and 3.22 compare our RTBSS algorithm with a version without pruning (i.e. complete limited search of depth D). Figure 3.21 presents the performance of our algorithm in function of the depth of the search used. The rewards obtained are the same whether we use the pruning or not; the slight variation comes from randomness in the tests. We see that our algorithm does not require any pruning to work properly. However, if we are able to find a good pruning heuristic for a problem, it greatly improves the algorithm's speed, as shown in Figure 3.22. The complexity is still exponential but it grows slower than the brute force version.

Figure 3.23 compares the progression of the performances for the RTBSS and the RTBSS-QMDP approaches for different depths of the search. We can see that the RTBSS-QMDP approach starts with really bad scores. This can be explained by the fact that at low depths, the algorithm is almost the same as Q_{MDP}, which is not good on Tag. However, with higher depths, the RTBSS-QMDP approach becomes to be good, because the search gives more information about the reachable belief states. At the depths of 4 and 5, we can see that the Q_{MDP} approach is useful, because the RTBSS-QMDP algorithm obtains better results than the RTBSS algorithm. The results stop

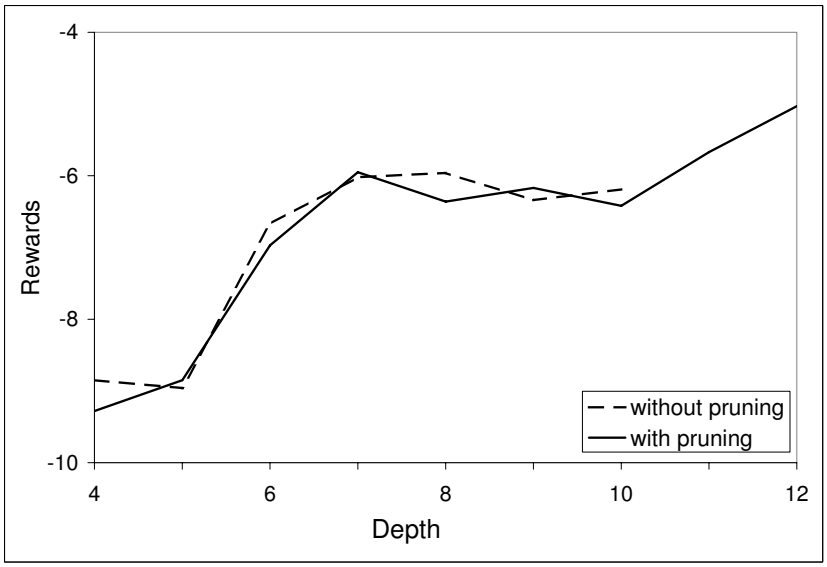


Figure 3.21: Average reward on *Tag* for different search depths D.

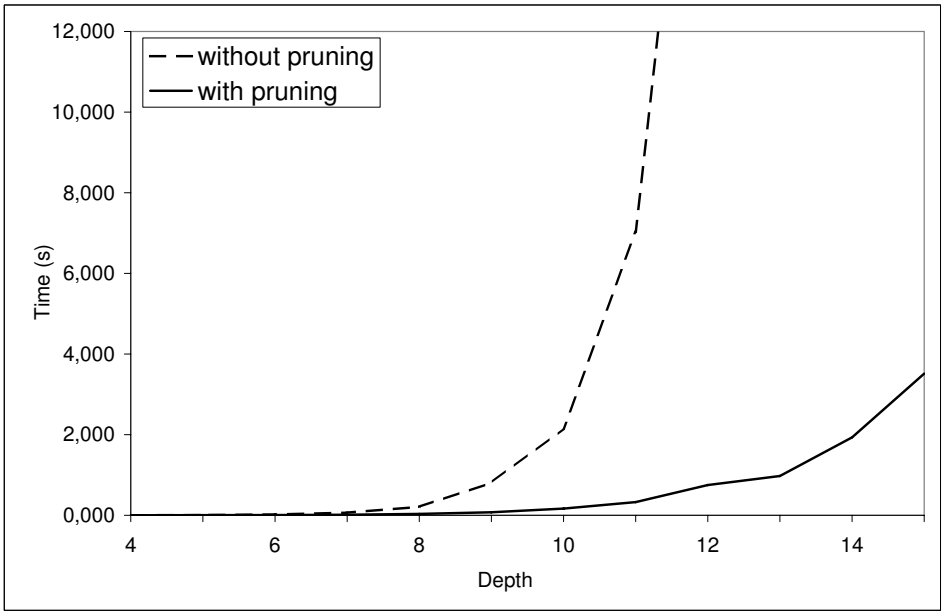


Figure 3.22: Average deliberation time on *Tag* for different search depths D.

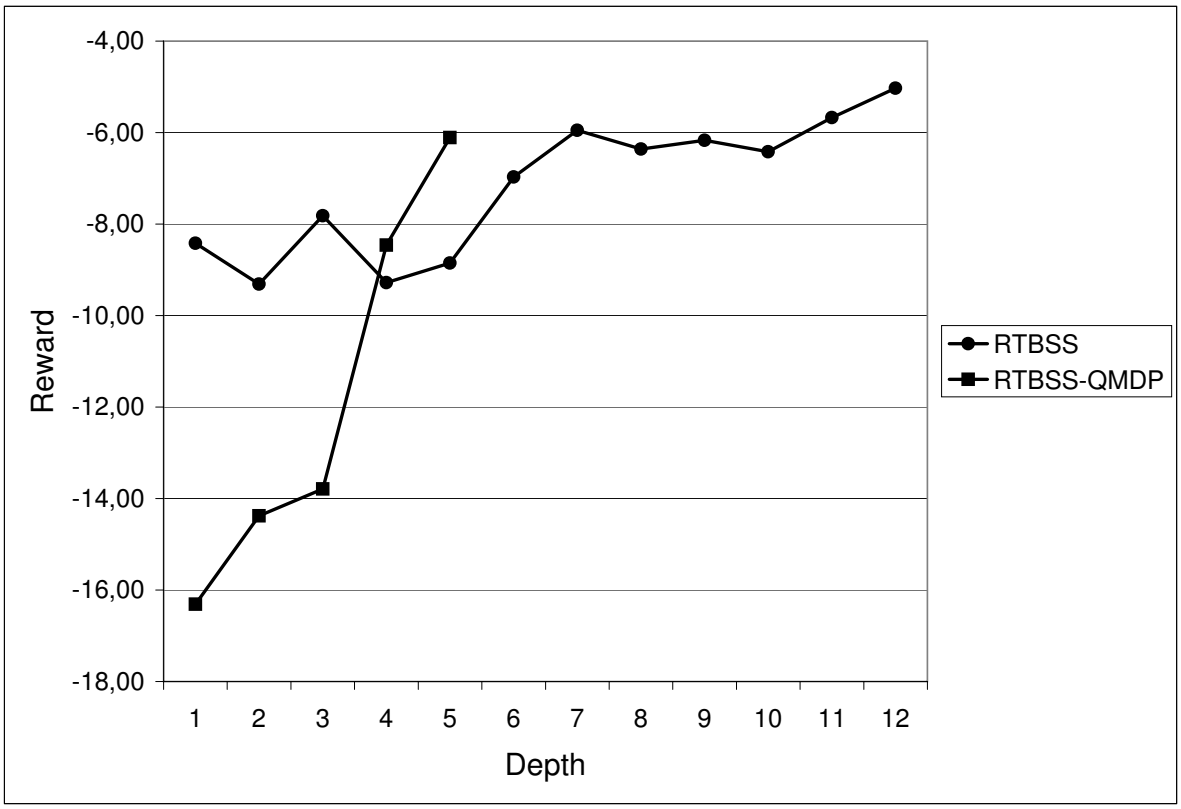


Figure 3.23: Rewards for different depth on Tag.

at depth 5, because at greater depths, the RTBSS-QMDP algorithm needs more than one second on average to choose an action.

3.6.2 *RockSample*

We have also tested our RTBSS algorithm in a bigger environment, the *RockSample* problem which was presented for the first time by Smith and Simmons (2004). For this problem, we have only results for the HSVI algorithm, however it was one of the best on the *Tag* problem, thus it should be a good comparison.

3.6.2.1 Environment Description

In the *RockSample* problem, an agent has to explore the environment and sample some rocks (see Figure 3.24), a little bit like a real robot should do on the planet Mars. The agent receives rewards by sampling rocks and by leaving the environment (by going to the extreme right of the environment). A rock can have a scientific value or not and

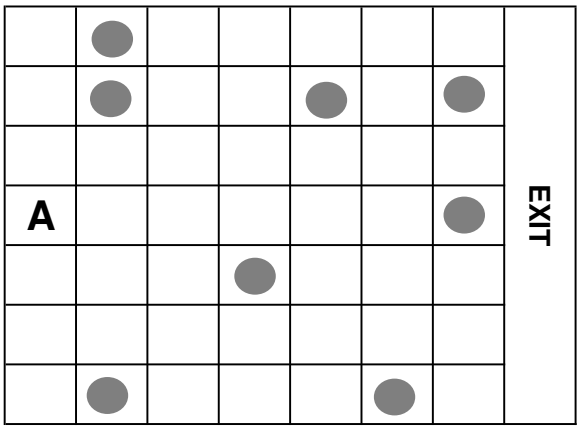


Figure 3.24: *RockSample[7,8].*

the agent has to sample only good rocks. At the beginning, the agent knows the position of each rock, but not their scientific value. In order to verify if a rock is good, the agent can use an imperfect sensor. This sensor helps the agent to see if a rock is good or not before choosing to go to this rock and sample it.

We define $RockSample[n, k]$ as an instance of the *RockSample* problem with a grid size of $n \times n$ and with k rocks. With a factored representation, the POMDP model is as follows. A state is characterized by $k + 1$ variables: $Position$, which can take the values $\{(1, 1), (1, 2), \ldots, (n, n)\}$ and k variables $Rock_i$, which can take the values $\{Good, Bad\}$. There is also an additional terminal state at the extreme right of the environment. Thus, for the $RockSample[n, k]$ environment, there are $n^2 \times 2^k + 1$ states.

Moreover, the agent can execute $k+5$ actions: $\{North, South, East, West, Sample, Check_1, \ldots, Check_k\}$. The moving actions are totally deterministic. The *Sample* action samples the rock at the agent's current location. If the rock is good, the agent receives a reward of 10 and the rock becomes bad (indicating that nothing more can be gained by sampling it). If the rock is bad, the agent receives a reward of -10. If the agent moves into the terminal state at the extreme right of the environment, it receives a reward of 10. All other moves have no rewards.

Each $Check_i$ action applies the long-range sensor to rock i, returning a noisy observation from $\{Good, Bad\}$. The sensor is more precise if the agent is closer from the rock. The noise in the long-range sensor reading is determined by the efficiency η, which decreases exponentially as a function of the Euclidean distance from the target. The efficiency of the long-range sensor is defined as being $\eta = 2^{-d/d_0}$, where d is the distance and d_0 is a tunable constant called the *half efficiency distance* ($\eta = 1/2$ when $d = d_0$). At $\eta = 1$, the sensor always returns the correct value. At $\eta = 0$, it has a 50/50 chance of returning *Good* or *Bad*. At intermediate values, these behaviors are

Problem	η	d_0
RockSample[4,4]	e^{-d}	-
RockSample[5,5]	$2^{-d/d_0}$	4
RockSample[5,7]	$2^{-d/d_0}$	20
RockSample[7,8]	$2^{-d/d_0}$	20
RockSample[10,10]	$2^{-d/d_0}$	20

Table 3.3: *RockSample* test parameters, which are the same parameters as those used by Smith and Simmons (2004, 2005).

combined linearly. The initial belief is that every rock has equal probability of being *Good* or *Bad*.

3.6.2.2 Results

We have compared many algorithms on four instances of the *RockSample* problem. For all tests, the discount factor was set to $\gamma = 0.95$. Table 3.3 shows the values for the constants: η (the efficiency) and d_0 (the half efficiency distance). For the results presented, the reward and the calculation time are averages over 5000 simulations. Moreover, the Q_{MDP} approach used in the *RockSample* environments has slightly been modified compared to the basic algorithm presented in section 3.1.3.1. The only modification is that it does not consider a check action on a rock that has a zero probability to be *Good*. Also, the convergence value for the Q_{MDP} algorithm was set to 5×10^{-15}.

RockSample[4,4] In this environment, since it is rather small, we can see in Table 3.4 that most offline and hybrid approaches succeeded in finding the optimal policy (average reward around 18). Our algorithm, RTBSS, which is done entirely online, is not so far behind and has no computation time offline. In addition, the proposed hybrid approaches developed with our algorithm performed as well as the offline and other hybrid approaches, but at a fraction of the offline time and within acceptable online time.

The Q_{MDP} approach alone did not yield very good results in this environment since it is an information gathering problem, which is not well supported by an algorithm that solves the underlying MDP. So most of the time, the agent got stuck doing a lot of check actions far from the rocks, since it did not learn to optimize its check actions, which hindered a lot its performances.

With our RTBSS approach, we got pretty good results, although the path chosen by the agents to visit the rocks was not optimal, thus getting a bit lower results than

Algorithm	Reward	Offline Time (s)	Online Time (s)
Q_{MDP}	8.6	0.188	0
Perseus	16.1	2104	0
RTBSS(4)	**16.5**	**0**	**0.001**
RTBSS-PBVI-QMDP(2)	**17.4**	**8229**	**0.014**
PBVI	17.9	10200	0
RTDP-BEL	18.0	486	0.002
RTBSS-QMDP(5)	**18.0**	**0.188**	**0.036**
HSVI Smith and Simmons (2004)	18.0	577	-
HSVI2 Smith and Simmons (2005)	18.0	0.75	-
RTDPBSS(2)	**18.1**	**1272**	**0.022**

Table 3.4: Results for the *RockSample*$[4, 4]$ problem. This problem has 257 states, 9 actions and 2 observations. The number between parenthesis beside the RTBSS based methods is the depth of the search D used to obtain these results.

the best policies. This was probably caused by our heuristic function for the leaf value which simply returned the value the agent would get with a blind policy (always move east and head straight to the exit). We also only used a heuristic action pruning in this problem, which limited our depth search to a depth of 6 to stay in respectable online time.

With the Perseus algorithm, we got pretty inconsistent policies, some were quite good and others were pretty bad and got stuck doing check indefinitely on already sampled rocks. This was probably caused by the approximation done in the algorithm that did not seem to always optimize the value function for key belief states. To get more consistent policies, we modified slightly the algorithm to be sure that it would always choose the best vector for some specific belief states at each iteration. The specific belief states we chose were belief states in which the agent was in a position where there was a good rock with a probability of 1 and a belief state where the agent was in a position adjacent to the exit. This had the effect of rapidly propagating the high rewards that the agent can get in the environment and generally yielded more consistent policies. However, even with this modification, we ended up with 3 bad policies out of 10. The bad policies had only average rewards of 4.3, 7.8 and 7.9.

The PBVI algorithm tended to yield better and more consistent policies than the Perseus algorithm. Still, in some rare cases, we got bad policies similar to those obtained with Perseus, which seemed to be caused when the value function did not successfully converge within the maximum horizon. Out of 10 policies, 2 were bad (7.6 and 7.6) and the others were all optimal or very close to the optimal policy.

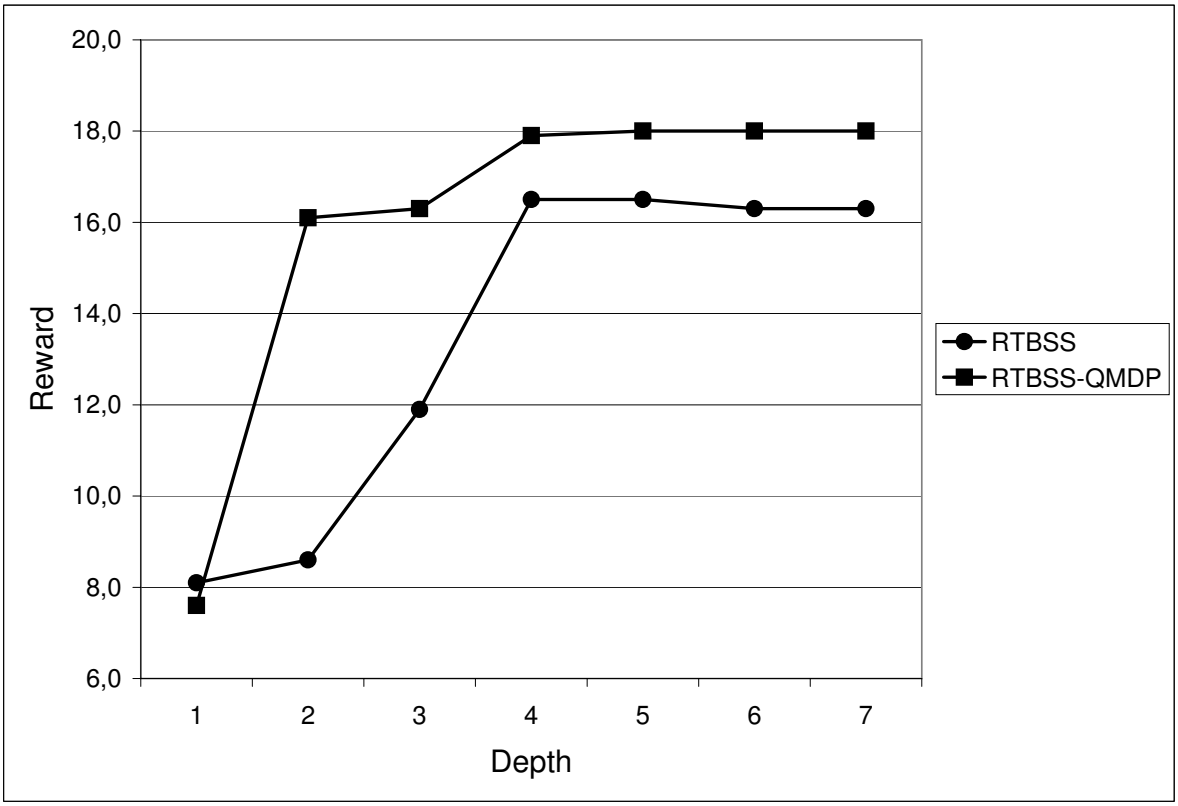


Figure 3.25: Rewards for different depth on RockSample[4,4].

With our proposed hybrid approach RTBSS-QMDP, we used the policy computed by Q_{MDP} that did not yield good results. With only a depth of 2, the RTBSS-QMDP algorithm got significantly better results than Q_{MDP} and similar results to our online RTBSS algorithm. At a depth of 4, the algorithm found the optimal policy and did not show further improvement at deeper depths. Figure 3.25 shows a comparison of the performances of the RTBSS algorithm and the RTBSS-QMDP algorithm for different depths.

With the RTDP-BEL algorithm, we used a discretization factor of 20 and a maximum number of steps of 251 per simulation. It succeeded finding the optimal policy within 5000 simulations. However, the algorithm did pretty badly in the first 2500 simulations, always taking the maximum number of steps at each simulation, which considerably slowed its learning process in these first simulations. As a result, the algorithm did not show faster convergence rate than other offline approaches such as HSVI or Perseus.

Our proposed RTDPBSS algorithm showed quite an improvement over the RTDP-BEL algorithm in the time it took to converge to the optimal policy. We used the same

Algorithm	Reward	Offline Time (s)	Online Time (s)
RTDP-BEL	7.3	1444	0.003
Q_{MDP}	13.9	0.625	0.002
Perseus	14.0	36000	0
RTBSS(6)	**18.5**	**0**	**0.131**
HSVI Smith and Simmons (2004)	19.0	10208	-
PBVI	19.1	36000	0
RTBSS-PBVI-QMDP(2)	**19.2**	**36000**	**0.020**
RTBSS-QMDP(4)	**19.3**	**0.625**	**0.627**
RTDPBSS(2)	**19.4**	**18636**	**0.054**

Table 3.5: Results for the *RockSample*$[5, 5]$ problem. This problem has 801 states, 10 actions and 2 observations. The number between parenthesis beside the RTBSS based methods is the depth of the search D used to obtain these results.

parameters as for the RTDP-BEL algorithm, that is, a discretization factor of 20 and a maximum number of steps of 251. Even at a search depth of 2, the algorithm attained the optimal policy around 3 times faster than the RTDP-BEL algorithm and showed better behaviors in the first simulations compared to RTDP-BEL algorithm.

RockSample[5,5] The Q_{MDP} approach alone did not yield very good results in this environment for the same reasons mentioned for the *RockSample*$[4, 4]$ problem. With our RTBSS approach, we got pretty good results, although the path chosen by the agents to visit the rocks was not optimal, thus getting a bit lower results than the best policies.

With the Perseus algorithm, we got pretty inconsistent policies for this environment too. The best result reported in Table 3.5 is 14.0, but the average over 10 policies was only 9.5. Again, the PBVI algorithm tended to yield better and more consistent policies than the Perseus algorithm. However, the average over ten policies was 12.1, which is way worse than the best score (19.1) reported in the table.

With our proposed hybrid approach RTBSS-QMDP, we used the policy computed by Q_{MDP} that did not yield good results. The improvement is quite good, because at a depth of 4, the RTBSS-QMDP algorithm obtained the second best score. The RTBSS-QMDP algorithm is quite interesting because in most problems, it can obtain really good results with a short amount of time offline and online.

In the RockSample[5,5] environment, the RTDP-BEL algorithm obtained surprisingly bad results. It was not able to converge at all, even after 20,000 simulations.

Algorithm	Reward	Offline Time (s)	Online Time (s)
Perseus	12.5	36000	0.001
Q_{MDP}	17.3	4	0.015
PBVI	18.9	36000	0
RTBSS-PBVI-QMDP(2)	**20.3**	**36000**	**0.177**
RTBSS(5)	**22.7**	**0**	**0.070**
HSVI Smith and Simmons (2004)	23.1	10263	-
RTBSS-QMDP(2)	**23.7**	**4**	**0.104**
RTDPBSS(2)	**24.5**	**41117**	**0.234**
RTDP-BEL	24.7	8773	0.006

Table 3.6: Results for the *RockSample*[5, 7] problem. This problem has 3201 states, 12 actions and 2 observations. The number between parenthesis beside the RTBSS based methods is the depth of the search D used to obtain these results.

Our proposed RTDPBSS algorithm showed quite an improvement over the RTDP-BEL algorithm. At only a search depth of 2, the algorithm attained very good results after only 7500 simulations. This shows that it can be quite useful to guide the search of the RTDP algorithm with a deeper search online.

RockSample[5,7] Again in this environment, the Q_{MDP} approach alone did not yield very good results. With our RTBSS approach, we got pretty good results, really close to the HSVI algorithm.

The Perseus algorithm was quite bad in this environment. The best result reported in Table 3.6 is 12.5, but the average over 10 policies was only 5.5. Again, the PBVI algorithm tended to yield better and more consistent policies than the Perseus algorithm. However, the average over ten policies was 13.5, which is way worse than the best score (18.9) reported in the table.

With our proposed hybrid approach RTBSS-QMDP, we used the policy computed by Q_{MDP} that did not yield good results. The improvement is quite good, because at a depth of only 2, the RTBSS-QMDP algorithm obtained one of the best scores.

In the RockSample[5,7] environment, the RTDP-BEL and the RTDPBSS algorithms obtained the best results. It took 30,000 simulations for the RTDP-BEL algorithm to find a good policy. For the RTDPBSS algorithm it took only 7500 simulations to find a policy as good as the one found by the RTDP-BEL algorithm. However, the time of each simulation is far bigger for the RTDPBSS algorithm. Therefore, the RTDPBSS algorithm is really interesting only if the simulations have a cost. For some problems, it might be cheaper to take more time to do a simulation and to learn with less simulations.

Algorithm	Reward	Offline Time (s)	Online Time (s)
PBVI	4.3	36000	0
Perseus	8.3	36000	0.001
RTDP-BEL	8.7	8362	0.029
RTBSS-PBVI-QMDP(2)	**13.1**	**36000**	**0.559**
HSVI Smith and Simmons (2004)	15.1	10266	-
Q_{MDP}	15.5	24	0.048
RTDPBSS(2)	**17.2**	**47007**	**0.356**
RTBSS(5)	**19.0**	**0**	**0.022**
RTBSS-QMDP(2)	**20.3**	**24**	**0.320**
HSVI2 Smith and Simmons (2005)	20.6	1003	-

Table 3.7: Results for the *RockSample*$[7, 8]$ problem. This problem has 12545 states, 13 actions and 2 observations. The number between parenthesis beside the RTBSS based methods is the depth of the search D used to obtain these results.

RockSample[7,8] In this environment, the Q_{MDP} approach was not bad at all. It obtained better results than the HSVI approach, as shown in Table 3.7. In this big environment, the HSVI algorithm did not have time to converge to a good policy. However, the HSVI2 algorithm obtained really good results in a short amount of time.

Moreover, in this environment, the RTBSS approach obtained a good result with really little computation time. With these results, we can see that the approaches that use some online calculations are less hindered by the growth of the environment size.

The Perseus and the PBVI algorithms were really bad in this environment. After 10 hours of calculations, they never obtained a good policy and they often obtained the worst policies with average rewards of 0. The environment is just too big for these two approaches to find good policies in 10 hours.

The RTBSS-QMDP approach obtained the second best result with a search depth of only 2. If we look at the results from all the problems tested, the RTBSS-QMDP approach seems to be the more consistent. It can always find good results with a really short amount of time offline and online.

In the RockSample[7,8] environment, the RTDP-BEL algorithm obtained bad results. It ran out of memory before reaching 20,000 simulations. On the other hand, the RTDPBSS algorithm showed quite an improvement over the RTDP-BEL algorithm. At only a search depth of 2, the algorithm attained a reasonably good result after only 2500 simulations. However, it took it a lot of time to do these 2500 simulations.

Algorithm	Reward	Offline Time (s)	Online Time (s)
Q_{MDP}	11.2	208	0.029
RTBSS-QMDP(2)	**19.2**	**208**	**1.234**
RTBSS(7)	**20.0**	**0**	**1.441**
HSVI2 Smith and Simmons (2005)	20.4	10014	-

Table 3.8: Results for the *RockSample*$[10, 10]$ problem. This problem has 102 401 states, 19 actions and 2 observations. The number between parenthesis beside the RTBSS based methods is the depth of the search D used to obtain these results.

Depth	Reward	Online Time (s)
1	5.1	0.000
2	17.3	0.000
3	18.8	0.000
4	18.0	0.001
5	19.6	0.011
6	18.8	0.133
7	20.0	1.441

Table 3.9: Results of the RTBSS algorithm for different search depth in the *RockSample*$[10, 10]$ problem.

RockSample[10,10] In this big environment of 102 401 states, the RTBSS and RTBSS-QMDP algorithms obtained competitive results with the HSVI2 algorithm, as shown in Table 3.8. More precisely, in Table 3.9, we can see that the RTBSS algorithm obtained good results with a short amount of time online. For example, at depth 3, the RTBSS algorithm already obtained an average reward of 18.8 with less than a millisecond on average to choose an action. This shows that the RTBSS algorithm can stay strong in big environments.

3.6.3 Offline Computation Time

Another huge advantage of our algorithm is its adaptability to environment changes. Let's suppose that we have the *RockSample* problem but at each new execution in the environment, the initial position of the rocks changes or the shape of the grid changes. With offline algorithms, it would require recomputing a new policy for the new configuration while our algorithm could be applied right away. Therefore, our RTBSS algorithm is more suited to environments in which the initial configuration can change

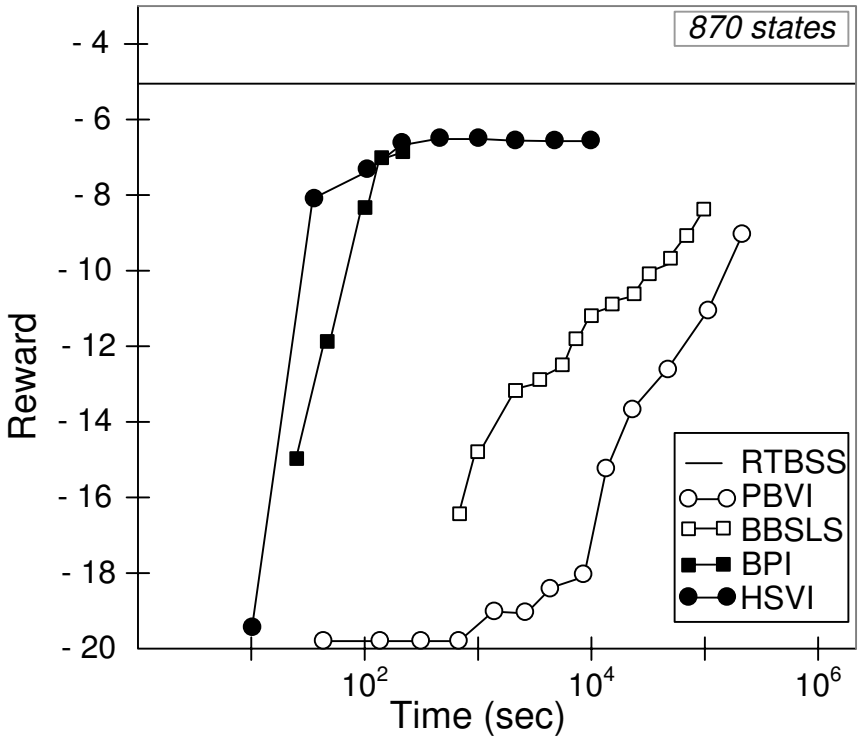


Figure 3.26: Solution quality versus offline computation time for the *Tag* environment.

and when the agent has to be deployed rapidly. For instance, in the RoboCupRescue simulation, agents have to be deployed immediately in previously unknown cities; they do not have the time to learn a good policy because the situation is deteriorating rapidly.

Figures 3.26 to 3.28 show that our RTBSS algorithm is the best one if only a small amount of time is allowed for offline computation. On this figure, the line representing the RTBSS performance is different from the others because it does not use any offline computation time to construct a policy. It shows how much time it takes for offline approaches to catch up (if they do) with the performances that RTBSS can have immediately. On big environments, offline approaches need too much time offline to obtain good results, but our online RTBSS algorithm can obtain good results immediately. This becomes really important if the agents have to be deployed rapidly in an environment that can change from one execution to the other.

3.7 Experimentations in RoboCupRescue

In this section, we present results in a much more complete environment: the RoboCupRescue simulation. This environment, presented in Chapter 2, consists of a simulation of an earthquake happening in a city. The goal of the agents (representing

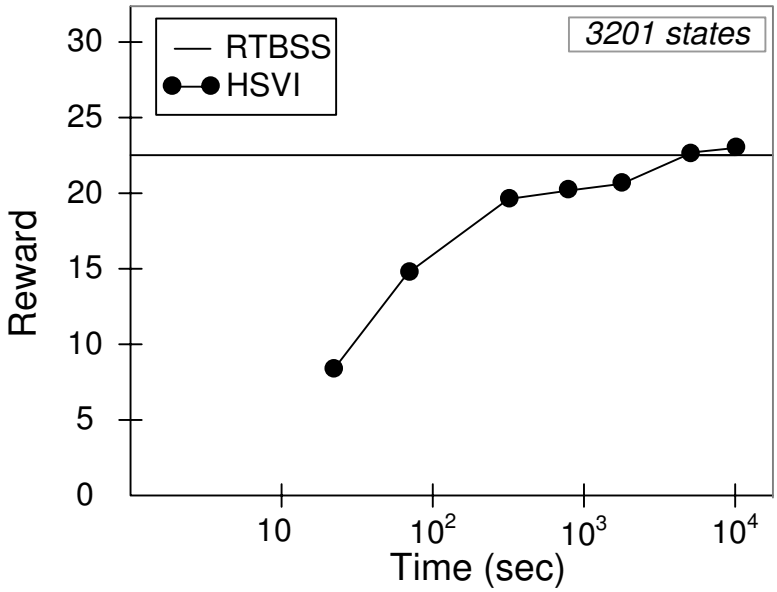


Figure 3.27: Solution quality versus offline computation time for the *RockSample*[5,7] environment.

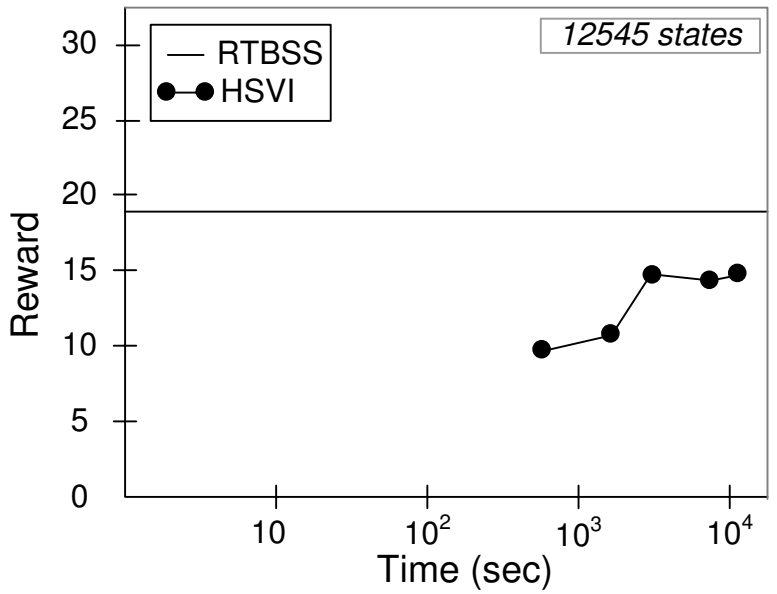


Figure 3.28: Solution quality versus offline computation time for the *RockSample*[7,8] environment.

firefighters, policemen and ambulance teams) is to minimize the damages caused by a big earthquake, such as civilians buried, buildings on fire and roads blocked. In this dynamic environment, there are a lot of uncertainties that complicate the work of the agents.

In the RoboCupRescue simulation, we have applied the RTBSS algorithm to control the policeman agents. Their task is to clear the most important roads as fast as possible, which is crucial to allow the other rescuing agents to perform their tasks. However, it is not easy to determine how the policeman agents should move in the city because they do not have a lot of information. They have to decide which road to prioritize and coordinate themselves so that they do not try to clear the same road.

In this section, we present how we applied our RTBSS algorithm in the RoboCupRescue simulation. In fact, we have been interested in only a subproblem of it which can be formulated as: Having a partial knowledge of the roads that are blocked or not, the buildings on fire and the position of other agents, which sequence of actions should a policeman agent perform?

3.7.1 RoboCupRescue viewed as a POMDP

The first task in order to apply our RTBSS algorithm is to define the RoboCupRescue simulation as a POMDP from the point of view of a policeman agent. The different actions an agent can do can be represented as four move actions (*North*, *South*, *East*, *West*) and a *Clear* action. The *Clear* action clears the road on which the agent is. A state can be described by approximately 1500 random variables, depending on the simulation:

- *Roads*: There are approximately 800 roads in a simulation and they can either be blocked or cleared. Consequently, there are 2^{800} possible configurations.
- *Buildings*: There are approximately 700 buildings in a simulation and they can either be on fire or not. Therefore, there are 2^{700} possible configurations.
- *Agents position*: An agent can be on any of the 800 roads and there are usually 30 to 40 agents. Then, in the worst case there are 800^{40} possible configurations.

This modeling is a simplification from the original RoboCupRescue simulation because we consider that the variables representing the roads and the buildings can have only two values. In reality, a road and a building can be blocked or on fire at different degrees. However, for a policeman agent it is not really important since a road has to be cleared no matter how bad it is blocked and a fire has to be extinguished no matter how big the fire is.

Consequently, if we estimate the number of states, we obtain $2^{800} \times 2^{700} \times 800^{40}$ states. This is a huge number of states, way beyond the capacity of most POMDP algorithms. However, a strong majority of them are not possible and will not ever be reached. The state space of RoboCupRescue is too important to even consider applying offline algorithms. We must therefore adopt an online method that allows finding a good solution very quickly.

3.7.2 Application of RTBSS on RoboCupRescue

Due to the complexity of the RoboCupRescue simulation, the basic RTBSS algorithm has been slightly modified to take the specificity of multiagent systems into consideration for the transition model and the maintenance of the belief state. The main idea is to abstract some of the dynamic parts of the environment in order to respect the real-time constraint. Most of the algorithm is the same as the standard RTBSS algorithm that has been presented previously. The modifications were simply done to improve its applicability in such complex, dynamic and uncertain environment.

First of all, in the RoboCupRescue simulation, the online search in the belief state space represents a search in the possible paths that an agent can take. In the tree, the probability of going from one belief state to another depends on the probability that the road used is blocked. One specificity of this problem is that we have to return a path to the simulator, thus the RTBSS algorithm has been modified to return the best branch of the tree instead of only the first action. This is a very simple modification that shows that the RTBSS algorithm can be easily modified to return a longer plan instead of only the first action. Depending of the problem considered, it might be interesting to return a longer sequence of actions.

Since the environment is real-time dynamic and it contains so many states, we had to add another simplification in order for the belief state to be updated in a timely manner. The idea is to consider factored beliefs in which some of the variables are kept fixed during the search. Therefore, the agent does not have to maintain beliefs over these variables. Consequently, the agent's belief state can be maintained more rapidly while doing the search in the reachable belief state space. In other words, the agent considers that some parts of the environment are static during its search in the tree.

For example, in the RoboCupRescue, all variables are considered static except the position of the agent and the variables about the roads, which are the most important variables for the policeman agent decisions. For the other variables, like the position of the other agents and the position of the fires, the agent considers that they keep the last value observed. Consequently, all those fixed variables are represented in the belief state by a vector containing only zeros except for the last value observed which has

a probability of one. Therefore, the function ω (Equation 3.30) only returns a small subset of states.

To sum up, the agent focuses on the most important variables for which it maintains its beliefs as precisely as possible. The other less important variables are considered fixed between the agent's observations. Which means, that the values of the fixed variables are only modified when the agent perceives a new value.

The fixed variables are not ignored during the search, but the agent does not update them. For example, if a firefighter agent is considered to be on road r_3, it will stay there during the whole search. We know that in practice it moves, but to simplify the search, we consider that it stays at the same position. We update the value of the fixed variables only when the agent perceives a new value. In our model, we consider the observations to be both the direct agent's observations and the information received by messages. We are in a cooperative multiagent system, therefore all agents have complete confidence in the information received from the other agents.

In the RoboCupRescue, we could estimate the position of all other agents by considering their position, their tasks and their speed. However, since the agents' behaviors are quite complex, the estimated probabilities would be only weak approximates. Even if it would be possible to obtain relatively good approximations, this would probably take a lot of computation time. Consequently, the price to pay is too important compared to the gain we could have made.

In complex dynamic multiagent environments, it is often more valuable to rely on observations than on predictions, because there are too many things moving in the simulation. Therefore, the agent should focus on the more important parts of the environment. To efficiently take all the unpredicted parts of the environment into consideration, the agent can shorten its loop of observation and action to keep its belief state up-to-date. This can be done because our RTBSS algorithm can find an action very quickly. Consequently, an agent using the RTBSS algorithm makes frequent observations. Therefore, it does not need a complicated model to predict the movement of the less important parts of the world, because these less important parts do not have time to move a lot between observations.

Since the RTBSS algorithm is executed online, the agent's belief state can be updated with the new agent's observations before each decision about the action to execute. Therefore, there is a less negative impact in considering some variables to be fixed during a local search than to consider them fixed for the complete planning process. Generally, in realistic environments, the agent can more precisely perceive the objects around it, therefore its belief state is normally more precise for the near objects. Since the agent is choosing its actions based on a local search, the agent will only consider well known near objects during the search. This reduces the impact of considering some

variables fixed because the far objects of the world would not be considered during the search anyway. Moreover, we can do an analogy with a robot in a room. This robot needs to have a good knowledge of the objects in the room, it is normally less important for the robot to have good beliefs about the objects in the other rooms.

3.7.3 Local Reward Function

Another advantage of using our RTBSS online POMDP algorithm is that the reward function does not have to be defined for all the states. The reward function can be adjusted before each action's decision to consider all the moving parts in the environment. This enables the algorithm to be applied in a very dynamic environment like the RoboCupRescue. This is possible because the algorithm solves a new problem each time it has to find an action, thus it does not matter if the reward function changes from one action's decision to another.

For the particular problem of the policeman agents in the RoboCupRescue simulation, we have defined a local reward function that gives a reward for clearing a road that depends on the position of the fires and on the position of the other agents. This enables the agent to efficiently compute its estimated rewards based on its current belief state without having to explicitly store all rewards for all possible states. The rewards are only defined for the current situation, which greatly simplify the reward function definition. For example, we do not have to define the rewards considering agents in the north part of the city if there are no agents at this position.

More precisely, here is how our local reward function works. A policeman agent needs to assign a reward to each road in the city, which are represented as nodes in a graph (see Figure 3.29). The reward values change in time based on the position of the other agents and the fires, therefore the RTBSS agent needs to recalculate the rewards at each turn. To calculate the reward values, the RTBSS agent propagates some rewards over the graph, starting from the rewarding roads, which are the position of the other agents and the fires. For example, if a firefighter agent is on road r_1 then this road would receive a reward of 5, the roads adjacent to r_1 in the graph would receive a reward of 4, the roads adjacent to the roads adjacent to r_1 would receive 3, and so on. For the fires, we draw concentric circles around the fires at different perimeters. The roads around the fires then receive rewards based on in which perimeter they are.

In our settings, each policeman agent is running its own RTBSS algorithm based on its own perceptions. However, policeman agents have to be coordinated, because we do not want all agents to take the same road since they could block each other. Therefore, we have to maintain some degree of dispersion among the policeman agents. This needed coordination is obtained using our local reward function. It is only with the reward function that a policeman agent considers the other policeman agents.

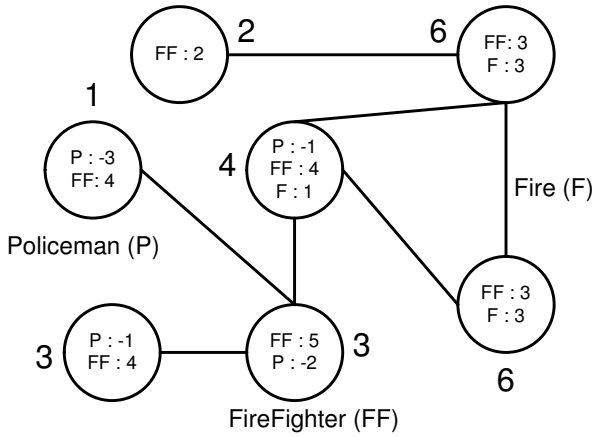


Figure 3.29: Reward function's graph.

To make sure the agents do not work on the same tasks, the RTBSS agent propagates negative rewards around the other policeman agents, thus they end up repulsing each other. The rewards are propagated exactly like the firefighter example described before, the only difference is that we propagates negative rewards. With this simple modification of the local reward function, we were able to disperse efficiently, and thus dynamically coordinate up to fifteen agents acting in a real-time dynamic environment.

Figure 3.29 shows an example of a reward graph. The nodes represent the roads and the reward source is identified in each node. The big number over a node is the total reward, which is the sum of all rewards identified in the node. As we can see, roads around the firefighter agent receive positive rewards, while roads around the policeman agent receive negative rewards. For example, there is a firefighter at the bottom center node. The rewards induced by this firefighter are represented in the road nodes as FF. We can see that the road where the agent is receives a reward of 5 because of the presence of the firefighter ($FF = 5$). The roads connected to the firefighter's road receive a reward of 4 ($FF = 4$). And so on, for all connected roads to the connected roads.

This reward propagation tells the policeman agent that it is not just the road where the firefighter is that is important to clear but also adjacent roads so that the firefighter could move more freely afterwards. It also helps the policeman agent to consider the uncertainty about the position of the other agents. Since the policeman agent is not sure about the position of the other agents, it is a good idea to enlarge the impact of the agent's position to adjacent roads because it could be the case that the firefighter is not exactly at the specified position.

If we look at the total rewards on our example, the policeman agent using this reward function would want to go to roads near the fire at the right and not necessarily go to the firefighter at the bottom because there is already a policeman agent near

it. Consequently, agents are coordinating themselves simply by propagating negative rewards. This is a nice way to coordinate agents in an online multiagent POMDP, because it gives us a very flexible coordination process. In our example, policeman agents repulse each other, but if some roads become very important, then many agents would try to clear them. In other words, if roads all have approximately the same values, then the agents would be dispersed. However, if some roads become more important, then more agents would try to clear them. And when these important roads are cleared, the agents can be dispersed again. This really flexible behavior gave us pretty goad results in the RoboCupRescue simulation, as presented in the next section.

3.7.4 Results

In such a huge problem as RoboCupRescue, it was impossible to compare our approach with other POMDP algorithms. Therefore, we compared our algorithm RTBSS with a heuristic method for the policeman agents.

To demonstrate the efficiency of RTBSS, we have compared it with our last approach for the policemen, which was an intuitive approach in which agents cleared roads according to some priorities. Each policeman agent received a sector for which it was responsible at the beginning of the simulation. Policeman agents cleared roads in this order: roads asked by the other agents, roads around refugees and fires and finally, all the roads in their sector.

The results that we have obtained on 7 different maps are presented in Figure 3.30. For our results, the maximal depth D was set to 10. We have tried many values, and it was $D = 10$ that gave us the best tradeoff between performance and calculation time. By using our RTBSS algorithm, it improved the average score by 11 points. This difference is very important because in competitions, a few tenths of a point can make a big difference. For example, at the 2004 international competition, our DAMAS-Rescue team missed the first place by 0.4 points. Furthermore, on the graph we show a 95% confidence interval that suggest that our algorithm allows more stable performances.

Figure 3.31 shows a comparison of the number of agents that are blocked at each cycle. As we mentioned above, one of the goals of the policeman agents is to clear the roads so that other agents can navigate freely in the city. The fewer agents that are blocked, the better the performances are. The results show that our method allows prioritizing the most important roads since on average, there are one or two fewer blocked agents. This means that those agents save civilians instead of waiting for policeman agents. Furthermore, Figure 3.32 shows the number of roads that are blocked at each cycle in the simulation. We see that RTBSS allows the policeman agents to clear the roads faster. Briefly, with RTBSS, agents clear the most important roads faster than with the heuristic approach.

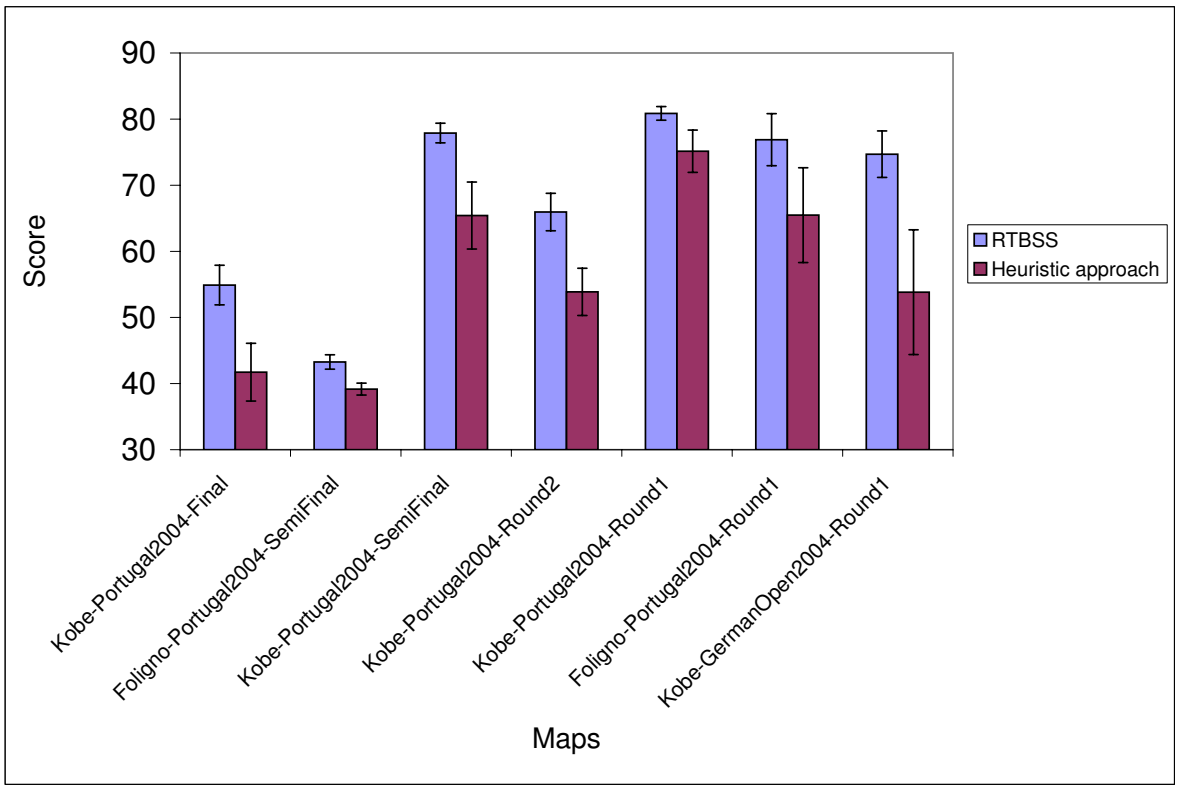


Figure 3.30: Scores obtained on seven different simulations.

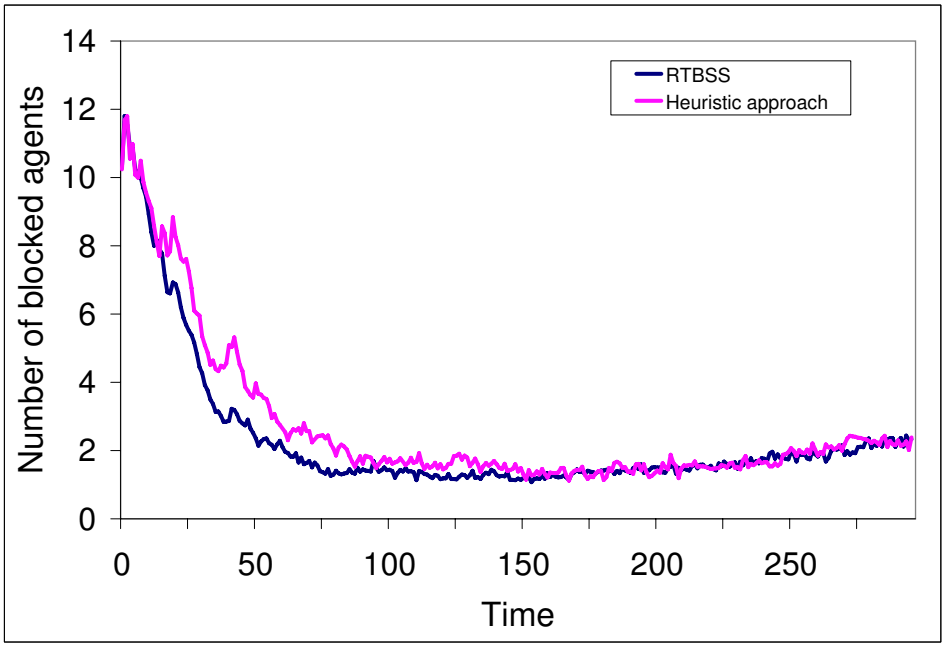


Figure 3.31: Number of agents blocked.

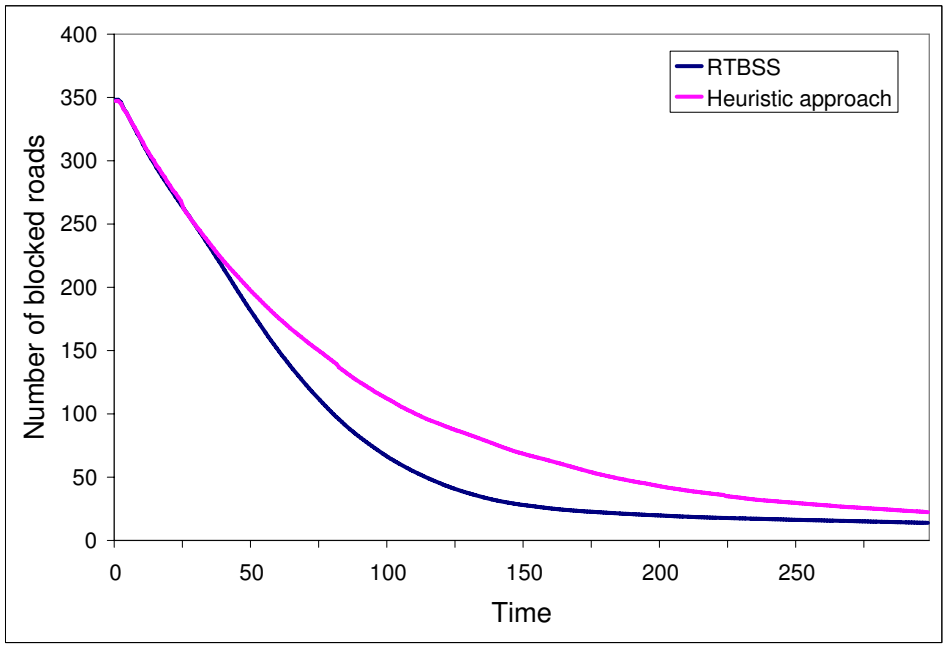


Figure 3.32: Number of roads blocked.

Another import result which supports this approach is our performance in the 2004 RoboCupRescue international competition. We finished in second place at this competition, really close to first place. Part of our success was because our *PoliceForce* agents were efficiently clearing the roads. Table 3.10 shows the percentage of cleared roads for all the maps used during the competition. In this table, the results of all the semifinalist teams are presented. We can see that our team (DAMAS) was the best team on six maps, which is the best score among the participants. In addition, we got the best percentage and by far the best standard deviation, which shows that our agents were the most consistent. With our RTBSS algorithm, our agents were able to efficiently adapt their behavior to all maps.

3.8 Discussion

Now that we have presented our online POMDP algorithm (RTBSS), we summarize in this section some of its advantages and some of its disadvantages. This should help people to see if the RTBSS algorithm is a good approach to solving one of their problems.

	ResQ	Damas	Caspian	BAM	SOS	SBC	ARK	B.Sheep
Final-VC	74,68	**82,22**	71,79	70,43	N/A	N/A	N/A	N/A
Final-Random	77,84	**86,51**	77,66	63,10	N/A	N/A	N/A	N/A
Final-Kobe	92,25	**93,74**	92,08	92,05	N/A	N/A	N/A	N/A
Final-Foligno	96,41	**97,72**	97,22	96,07	N/A	N/A	N/A	N/A
Semi-VC	67,93	79,57	68,86	57,90	67,22	57,85	53,27	**80,53**
Semi-Random	82,53	**87,44**	77,47	81,93	82,26	79,53	80,30	78,76
Semi-Kobe	92,40	93,65	92,71	92,51	92,62	92,56	93,55	**99,72**
Semi-Foligno	95,45	**97,08**	95,58	96,37	96,93	97,07	95,92	83,44
Round2-Kobe	92,52	93,52	91,46	92,46	92,78	93,45	92,25	**99,50**
Round2-Random	87,74	90,03	87,62	87,71	87,86	88,73	85,03	**99,97**
Round2-VC	91,34	91,62	90,74	89,87	91,40	90,92	N/A	**98,86**
Round1-Kobe	89,19	89,51	87,78	88,21	88,30	87,70	**91,12**	81,17
Round1-VC	91,90	92,13	91,74	91,84	N/A	91,81	91,54	**99,82**
Round1-Foligno	95,84	96,92	96,52	96,36	94,19	96,62	**97,63**	80,15
Number of wins	0	**6**	0	0	0	0	2	**6**
AVG %:	87,72	**90,83**	87,09	85,49	88,17	87,62	86,73	90,19
STD %:	8,25	**5,09**	8,59	11,25	8,93	11,59	13,63	9,96

Table 3.10: Percentage of cleared roads during the 2004 RoboCupRescue international competition. Results reported by Kleiner et al. (2006).

3.8.1 Advantages

- The RTBSS algorithm does not need any offline computation. This enables the agent to be efficient in previously unknown environments, if the model of the environment is known at runtime. For example, the agent would be effective in any *RockSample* environments or in any city of the RoboCupRescue simulation, without any loss of time. With standard approaches, the agent would have to learn a new policy for each new configuration before acting in this new configuration.

- The RTBSS algorithm is able to manage large state spaces, since it only performs local searches in the belief state space; it does not need to define a policy for all possible situations. The policy is dynamically constructed online.

- The RTBSS algorithm is applicable in real-time environments, because we can easily control the time it takes to find an action by limiting the maximal depth of the search.

- It is possible to use the agent's real observations to maintain approximate belief states in a dynamic environment, because the RTBSS algorithm is executed online.

- It is possible to define a local reward function that can be adjusted at each turn to take into consideration the dynamic parts of the environment. This means that the reward function does not have to be defined for all possible states, but only for the current situation.

3.8.2 Disadvantages

- The branching factor depends on the number of actions and observations. Thus if there are a lot of observations and/or actions, it might not be possible to search deeply enough. However, if the number of actions and/or observations increases, it has a negative impact on all existing approaches.
- If the offline time is not a problem, it might be more interesting to use an offline algorithm since they normally give better results if they have the time to converge. However, if the problem is too big, it might be better to use our RTBSS algorithm, because the offline approach would not have time to converge. A good compromise might be to use hybrid approaches that have shown to be quite efficient.
- Two problem dependant functions have to be defined which might be hard for some problems: a heuristic function for the pruning condition and a belief state utility function to evaluate the leaves of the tree.
- The RTBSS algorithm needs a model of the environment.
- The RTBSS algorithm needs some time online between each action's choice.

3.9 Contributions

In this chapter, we have presented our Real-Time Belief State Search (RTBSS) algorithm, which is a new online algorithm for partially observable Markov decision processes (POMDP). In this last section, we summarize our contributions:

An online POMDP algorithm. We have conceived a real online POMDP algorithm. Most POMDP algorithms try to solve the problem offline by defining a policy for all possible situations the agent could encounter. This offline process is quite complex and this limits the applicability of most offline approaches to small problems. Many claimed online algorithms need in fact a lot of executions in the environment to learn a good policy. Our algorithm is different, because it does not need any calculation time offline and it is immediately efficient, even in previously unseen configurations of the environment.

Pruning strategy. We have defined a pruning strategy to accelerate the search in the belief state space. We have combined a limited depth first search strategy with a pruning strategy that uses dynamically updated bounds based on the solutions found at the maximal depth of the search. The pruning of the tree is also accelerated by sorting the actions in order of their expected efficiency. Since the more interesting actions are tried first, there is more chance that the first branches developed have better values, thus better bounds for the pruning condition.

Theoretical bound. The algorithm has a theoretical bound, thus we can guaranty that the distance between the policy defined by our algorithm and the optimal policy is bounded.

Hybrid approaches. We presented some hybrid approaches that use the RTBSS online search strategy mixed with approximate offline strategies. We presented three new algorithms: RTBSS-QMDP, RTBSS-PBVI-QMDP and RTDPBSS. The results show that the performances of the hybrid approaches are often better than the performances of the online approach or the offline approach taken alone. Our results have shown that RTBSS-QMDP is the most consistent approach over all the test environments.

Experimentations on standard POMDPs. We have compared our algorithm with state of the art POMDP algorithms on two POMDP problems: *Tag* (Pineau et al. (2003)) and *RockSample* (Smith and Simmons (2004)). We obtained competitive results with much less computation time and we were much better on the biggest environments.

Belief state for dynamic environments. We have conceived an approach to maintain a belief state based on the real agent's observations. This helps the agent manage the highly dynamic and unpredictable parts of the environment. During the search in the belief state space, the agent considers some variables fixed and concentrates only on the most important parts of the environment to choose its actions. This approach is possible with the RTBSS algorithm because it is an online algorithm that can readjust its belief states between each execution in order to stay up to date with the agent's observations.

Local reward function. We defined a local reward function enabling an agent using RTBSS to redefine a reward function before each action's choice. This enabled defining the reward function only for the current situation, which is really useful when there are a lot of possible situations. This again is possible because the agent's policy is dynamically defined thus the reward function can be modified before the action's search.

Flexible coordination approach. Our local reward function can be used to dynamically coordinate many agents in an environment without any coordination related messages. This new multiagent POMDP coordination approach has shown to be effective and quite flexible in controlling many agents in a highly dynamic environment.

Stable performances. At the RoboCupRescue competition, our agents using the RTBSS algorithm obtained the best and the most stable results. We obtained by far the best standard deviation. This shows that our agents were efficient in all maps used at the competition. This proves our claim that the RTBSS algorithm can be immediately efficient in previously unseen configurations of an environment.

Chapter 4

Task Allocation Learning

4.1 Introduction

The concept of coordination is often used and thus understood by most people. People can recognize a situation where there is good coordination, but it is often easier to recognize the absence of coordination. People often only notice a lack of coordination when they are faced with the consequences, like a collision, a delay or simply the failure of a task.

The coordination can be defined as the process managing the dependencies between different activities (Malone and Crowston (1994)). Consequently, if there are no dependencies between the agents' activities, then there are no reasons to try to coordinate them. Agents can then all act independently. However, in most multiagent systems, there are many dependencies between the agents' goals, their capacities and the resources they are using. In these conditions, coordination becomes quite important.

In fact, the coordination can be seen as the process enabling the agents to act together and help each other with some positive interactions (a task can help or improve another task), instead of harming each other with some negative interactions (a task can block or diminish the efficiency of another task) (Paquet (2001)).

Furthermore, solutions to coordination problems can be divided into three general classes (Boutilier (1996)):

- Those based on communication in which agents can communicate and negotiate together to: (i) determine the allocation of the tasks; (ii) solve conflicts and (iii) share resources.
- Those based on conventions in which agents use predefined conventions imposed by the system designer to assure a joint optimal action.

- Those based on learning, in which agents can learn coordination policies (or conventions) by repetitive interactions with the other agents.

In environments where the communications are limited or uncertain, approaches highly based on communication are not really appropriate. In such environments, the quantity of information that can be sent is limited and some messages may never reach their recipients. Consequently, approaches highly based on communication may become inefficient, since the agents' coordination relies on uncertain communications.

The second approach consists of defining all the coordination conventions *a priori*. This is a good method for solving a coordination problem in the required time. It is simple and fast to apply, because the system designer only has to define the coordination rules by himself. The difficulty is not in the complexity of defining the rules, but in the quantity of the rules necessary to obtain good coordination in all possible situations. In complex environments, the number of possible situations is huge and the number of coordination rules is thus also really huge. Moreover, this approach does not offer a great deal of flexibility, because if the environment changes, the rules have to be adjusted manually. This readjustment can become really tedious if there are many rules to readjust manually. Each minor modification of the environment can then require a huge workload to readjust the rules.

The third approach enables reducing the number of rules that the system designer has to define by using learning techniques. Therefore, the system designer does not have to define all the coordination conventions for all possible situations. Moreover, an approach based on learning enables obtaining a multiagent system that can gradually adapt itself to environment changes. In this thesis, learning is considered as any process that modifies the different agent components in order to better align them to the information returned by the environment, thus improving the global performance of the agent (Russel and Norvig (2003)).

In this chapter, we focus on learning algorithms used to improve the coordination between the agents. More precisely, we consider cooperative multiagent environments in which agents have to divide up the different tasks among themselves in order to accomplish them efficiently. However, if agents are faced with complex tasks, it might be hard for them to determine how many agents are required to accomplish each task, which is important information for the coordination process. Without this information, agents would not be able to divide up the different tasks efficiently.

To learn the required number of agents for each task, we have developed a selective perception reinforcement learning algorithm (Paquet et al. (2004b)), which is useful to manage a large set of possible task descriptions with discrete or continuous variables. It enables us to regroup common task descriptions together, thus greatly diminishing the

number of different task descriptions. Starting from this, the reinforcement learning algorithm can work with a relatively small state space.

Our tests in the RoboCupRescue simulation environment showed that the agents are able to learn a compact representation of the state space, facilitating their task of learning good expected rewards. Furthermore, agents were also able to use those expected rewards to choose the right number of agents to assign to each task. This information helped the agents to efficiently coordinate themselves on the different tasks, thus improving the group performance.

In the remaining sections of this chapter, we describe our selective perception reinforcement learning algorithm used to learn the required number of resources necessary to efficiently accomplish a task. Then, we present the results obtained by testing our algorithm in the RoboCupRescue simulation. But first, we present a literature review on coordination learning algorithms in the next section.

4.2 Literature Review on Coordination Learning

Coordination learning between software agents is an important domain in multiagent research (Panait and Luke (2003)). In this section, we present some methods used for coordination learning. This state of the art is not exhaustive, but it is representative of the different approaches developed in this domain. The different methods are divided into three categories. The first category contains the methods using reinforcement learning techniques, in which agents learn some utility values for all possible actions in all possible states. The second category contains the approaches that record an execution trace during the execution of the tasks. These execution traces are then analyzed to find some explanations for the failure or the success of the tasks. These explanations are then used to modify the agents behavior in order to act more efficiently if similar situations are encountered in the future. Finally, the last category regroups all the other approaches that do not fit in the first two categories.

4.2.1 Coordination Learning via Reinforcement Learning

Reinforcement learning (RL) is used to learn which action to perform in all possible situations in order to maximize a numeric reward. The agent does not receive any information about the action it is supposed to do, as is the case in many learning techniques. Instead, the agent has to discover, by trying the actions, which is the one that gives the best reward in each situation. In the most interesting case, the actions can affect not only the immediate reward, but also the subsequent rewards.

These two characteristics (trial and error and delayed rewards) are the most important characteristics that distinguish RL from the other learning techniques (Sutton and Barto (1999)).

RL techniques are really interesting for the agents to learn optimal behaviors because they only need a scalar retroaction from the system. Moreover, these techniques can be used when there are some uncertainties about the evolution of the environment. However, the convergence of the RL algorithms (like $TD(\lambda)$ (Sutton (1988)) or *Q-Learning* (Watkins and Dayan (1992))) has only been proved for Markov decision processes (MDP). MDPs are used for sequential decision problems where the agent needs to make many decisions and where each decision can have an impact on the subsequent decisions (Cassandra (1998)). Basic MDPs can be really useful to model the agent's behavior in a specific environment.

However, basic MDPs are not well adapted to multiagent systems, because in an MDP the other agents are not considered. Furthermore, in an MDP, the environment is considered stationary, which means that the state transitions have invariant probabilities (Buffet (2000)). For example, the condition of stationarity enables proving the convergence of the Q-learning algorithm to an optimal policy (Mitchell (1997)). However, in multiagent systems where agents are learning, the stationarity condition does not hold anymore, because agents are modifying their behaviors. Some researchers have tried to adapt MDPs to multiagent systems, as for example: the work of Boutilier (1996) on MMDPs and the work of Bernstein et al. (2002) on decentralized MDPs in partially observable environments (DEC-POMDP).

As shown by Boutilier (1996), a multiagent system can be represented with an MDP. Consequently, it is possible to use classic learning algorithms for multiagent systems. In such a case, a system state is a composition of the state of all agents and an action is a joint action composed of all the individual actions of all agents. This approach works well in theory, but in practice, the number of states and actions, in this central vision of the problem, often becomes too big for reinforcement learning algorithms to be applied.

It could also be possible to use a decentralized approach (Becker et al. (2003); Bernstein et al. (2005); Beynier and Mouaddib (2004, 2005)), but it is hard to solve the problem using such an approach (Bernstein et al. (2002); Goldman and Zilberstein (2004)), because there are two main difficulties:

1. *Transitions are uncertain.* Other agents are unpredictable elements of the environment, thus the state transitions viewed from the point of view of an agent are uncertain. Agents do not know in which states they could end up after their actions.

2. *The environment is partially observable.* Since the agents perception is local, they cannot know the global state of the system. Consequently, such a problem is categorized as a partially observable Markov decision process (POMDP). However, most POMDP techniques are limited to really small environments.

As we can see, classic reinforcement learning algorithms need adaptation to make them applicable in multiagent environments. The next sub-sections present some reinforcement learning methods to learn how to coordinate agents in a multiagent system. These methods have been categorized in four main groups:

1. Methods using game theory test environments.
2. Methods where the coordination emerges without agents considering the other agents.
3. Methods where the agents exchange their perception and/or their experiences to improve their coordination.
4. Other methods that do not fit in the previous categories.

4.2.1.1 Game Theory Test Environments

All the methods presented here use utility tables to represent explicitly the situations with good coordination and those with bad coordination (Littman (1994a); Boutilier (1996); Chalkiadakis and Boutilier (2003); Kapetanakis and Kudenko (2002)). Each agent has access to the utility table defining the utility for all the agents. It is thus easy for the agents to determine the good situations from the bad ones. Good situations are simply the ones in which the agents receive the greatest rewards. The agents objective is then to coordinate their actions in order to receive the maximal reward in the utility tables considering the anticipated actions of the other agents.

In a *multiagent Markov decision process* (MMDP), the multiagent system is modelled as if there was only one agent which has the objective of producing an optimal policy for the joint MDP (Boutilier (1996)). A joint MDP is a standard MDP, but containing all possible states and actions for all the agents. An MMDP is a 5-tuple (α, S, A, T, R), in which α is a finite collection of N agents, $S = S^1 \times \ldots \times S^N$ is the joint state set and $A = A^1 \times \ldots \times A^N$ is the joint action set. A joint action $(a^1, \ldots, a^N)$ represents the concurrent execution of each action a^i by the agent i. The transition function T and the reward function R are defined over joint states and joint actions.

In an MMDP, the agents have to coordinate themselves, because the actions are chosen in a distributed manner. In general, there is more than one optimal policy in

an MMDP. Since each agent can choose its policy individually based on one optimal joint policy, there is no guarantee that all agents will choose the same optimal joint policy. For example, suppose a problem with two agents in which the agents have to choose the same action to receive the greatest reward. If there are two possible actions, then there are two optimal policies: both agents taking the first action, or both agents taking the second action. In such problems, agents have to be coordinated to choose the same optimal joint policy. In his approach, Boutilier (1996) supposed that each agent has some a priori knowledge of the other agents' policies and that this knowledge is updated during the execution. At each step, each agent records the actions executed by the other agents. After many experiments, each agent can obtain a probability distribution for each possible action of the other agents. The agents can then use these probability distributions to choose the most probable joint action.

Reinforcement learning algorithms need a good exploration of the possible strategies in order to converge on a stable solution. Chalkiadakis and Boutilier (2003) have developed an approach to try to solve the exploration problem in multiagent settings. They use bayesian models to weight the explorations with the expected gain using the notion of information value. This method requires that each agent has a model of the other agents, because each agent has to estimate the value of an action by considering the influence of this action on the future action choices of the other agents. Agents use bayesian networks to maintain beliefs on the world model and on the other agents' strategies. This method has only been tested on small utility tables.

One of the problems with the majority of the reinforcement learning algorithms is that they do not guarantee the convergence to the optimal joint action in scenarios where there are big penalties associated with bad coordination situations. Even approaches where the agents build a predictive model of the other agents have not proved the convergence to an optimal joint action (Claus and Boutilier (1998)). By modifying the action selection strategy of the Q-learning algorithm, Kapetanakis and Kudenko (2002) have shown that it is possible to improve the probability of convergence to the optimal joint action. In fact, they showed that one can converge to the optimal joint action with a probability which is near to 100%. However, this approach has only been tested on a really simple problem with only two agents with three actions each.

4.2.1.2 Emergence of the Coordination

This part presents five reinforcement learning methods where the agents do not try to model the other agents. The other agents are only seen as environment components like any other component and there is no communication between the agents. In these kinds of methods, the coordination can emerge because the rewards received are generally global rewards.

The first approach is called incremental reinforcement learning and it consists of progressively increasing the problem complexity. By doing so, it can use the solutions of the simpler tasks to find better solutions for the more complex tasks (Dutech et al. (2001)). This incremental learning is done along two axes: gradually increasing the number of agents and gradually increasing the tasks' complexity. The authors have shown that they can obtain better results with the incremental algorithm than with a standard reinforcement learning algorithm. Another similar approach called *behavior transfer*, developed by Taylor and Stone (2005), uses the information learned in simpler tasks to accelerate the learning of more complex tasks. One drawback of these two approaches is that they are highly dependant on the problem, because the system designer has to define all the progressive steps. In some problems, it might not be easy to define such progressive tasks.

Another approach to enable the agents to learn a cooperative task is to give the agents a share description of the environment and a global reinforcement (Crites and Barto (1998)). In this approach, the coordination emerges because they are learning from the same rewards. The rewards may seem to contain noise for an agent because it does not know the behavior of the other agents. However, the authors have demonstrated that it is still possible to learn a cooperative task in these conditions. One difficulty with this approach is the credit assignment problem, that is, the problem of properly assigning rewards for an overall performance change to each agent in the system that contributed to that change (Sen and Weiss (2000)).

A similar approach to the preceding one is the isolated and concurrent reinforcement learners approach (Sen and Weiss (2000)). In this approach, each agent maximizes the rewards received from the environment without considering the other agents. This method has many limitations, because it does not give a good coordination: (i) when the actions of the agents are highly coupled (i.e. the actions of an agent has a big impact on the other agents plans); (ii) when the rewards are delayed and; (iii) when there are many optimal behaviors.

Sen and his colleagues have shown that it was possible to obtain a good coordination between the agents without using any communication (Sen et al. (1994); Sen and Sekaran (1998)). These authors have used the Q-learning algorithm to make two agents learn how to push a block to a specified position. Agents were not communicating, but they still needed to be coordinated to push in the right direction. In order for this approach to work, the agents have to be able to perceive each other's action all the time. In more complex systems, this condition generally does not hold.

In this same vein, Abul et al. (2000) have presented two coordination mechanisms for agents using reinforcement learning. In the first mechanism, called *perceptual coordination mechanism*, the other agents are included in the state description and the

coordination information is learned from the state transitions. In the second mechanism, called *observation coordination mechanism*, the other agents are also included in the state description, but with the rewards obtained by the nearby agents. The rewards observed are used to construct an optimal policy. This approach has the same problem as Sen's approach: agents have to perceive all other agents at all times.

The emergence of the coordination is quite interesting, because the agents can obtain a good coordination without any communication and the communications can often be limited or costly. For the coordination to emerge, these methods normally suppose that the agents can perceive each other. Consequently, in partially observable environments, the coordination is harder to obtain because the agents cannot perceive each other all the time. It is harder to stay coordinated when the other agents are not visible. In the next section, we present methods in which the agents can communicate their perceptions or experiences in order to improve their coordination.

4.2.1.3 Sharing Perceptions and Experiences

In this section, five approaches, using information sharing in the context of coordination learning, are presented. The information shared can simply be the agent's perceptions so that all agents can profit from the encountered situations. By doing so, each agent has more examples from which it can learn.

In a prey-predator domain, Tan (1993) has studied the impact of sharing the agents' perceptions, the agents' policies and the agents' episodes. His results show that the agents that share their learned policies are more efficient than independent agents. However, it is important to notice that the coordination is somehow easy in this simplified domain. Notice that, in this domain, agents are homogeneous and consequently the coordination is facilitated.

Another way to share information is to share learned values to hold all the knowledge learned in common. In this optic, Berenji and Vengerov (1999, 2000) have developed an approach in which the agents can share their experiences by sharing their learned values from the Q-learning algorithm. They have shown that K cooperative agents learning in separate worlds during N time steps were more efficient than K independent agents learning during $K * N$ time steps.

Another approach is the one developed by Mataric (1994) where the global behavior of an agent is represented as a group of many basic behaviors. The behaviors that are of the most interest here are the social behaviors. Mataric (1997) has shown that three types of reinforcement are important when learning social behaviors. The first type is the individual perception of the progression toward the goal, i.e. that the agent receives a reward for each action getting it closer to the goal. The second type is the perception

of the other agents, i.e. that the observed behavior of the other agents is seen as positive reinforcements. In practice, this means that the agent receives a reward if it repeats the behavior of an agent it just has seen. The third type is the observation of the rewards received by the other agents. A shared reward is given to all agents participating in a local social interaction. For example, if an agent reaches its goal because another agent has moved out of the way, then both agents would receive a reward.

Bonarini and Trianni (2001) have done reinforcement learning using fuzzy classifier systems. Each agent has a set of rules in fuzzy logic which are used to choose the best behavior. Their behaviors are similar than the ones described by Mataric (1994). To improve the learning process, agents share their rewards. When an agent receives a reward, it communicates this reward to the agents that have helped. In fact, in their problem, agents communicated rewards to all agents in a short perimeter, determined by the communication range, around the agent that received the reward.

Another approach proposed by Ghavamzadeh and Mahadevan (2002) is to use the dynamic fusion of individual solutions, represented as MDPs, to build the global solution. Each MDP represents the individual solution if the agent was acting alone in the environment. The fusion of all the individual solutions from all the agents gives the solution for the global multiagent MDP in which all agents act together. The authors have developed a new temporal difference algorithm called MAPLE (MultiAgent Policy LEarning). This algorithm uses Q-learning and the dynamic fusion to build global solutions which are efficient in the complete multiagent problem. The main drawback of this approach is that each agent needs an individual solution of the problem at the beginning. They gave an example, where it is the system designer that gives each agent an individual optimal policy to solve the problem alone. Another limitation is that they suppose that the system is completely observable by all agents.

4.2.1.4 Other Approaches

In the approach of Prasad et al. (1996), the coordination is reached by giving each agent one or more roles which structures the agents' interactions. To improve the coordination, agents learn which role to undertake in all situations. To learn which role to assume, agents learn the three following values by reinforcement. First, they learn an estimation of the final state's value if the agent undertakes a certain role in the current situation. Secondly, they learn the probability of reaching a good final state if the agent undertakes a certain role in the current situation. And finally, they learn a cost value representing the computation time needed if a certain role is undertaken. With these values, the agents can use a reinforcement learning algorithm to learn which role to undertake in all situations in order to maintain good coordination between the agents.

In another approach, Stone and Veloso (1999) have developed an algorithm called TPOT-RL in which the states are factorized. TPOT-RL can learn a set of effective policies (one for each team member) with very few training examples. It relies on learned action-dependent features which coarsely generalize the state space. This algorithm is applicable in environments which are complex, non-Markovien, multiagent, with a big state space and where the learning opportunities are limited. Results in the RoboCup Soccer show that this algorithm enables a team of agents to learn to cooperate in order to reach a specific goal.

Tumer et al. (2002) and Agogino and Tumer (2005) have studied how to define the rewards so that if each agent is maximizing its own rewards, then the whole group of agents would reached a desired global solution. To do so, the authors have used the concept of collective intelligence presented by Wolpert and Tumer (2000). For the learning process to be effective, all agents need to see how their behavior has influenced the rewards received. To achieve that, each agent A uses a utility function which is the sum of the rewards received by the all the agents minus the sum of the rewards if agent A had not have been there. This kind of utility function helps to obtain a more cooperative behavior from the agents and thus it helps the group to be more efficient.

Makar et al. (2001) and Ghavamzadeh et al. (2005) have used a hierarchical structure to accelerate the learning process of the cooperative behaviors. Each agent uses the $MAXQ$ decomposition (Dietterich (1998)) to decompose the main task in sub-tasks. The coordination is learned using the joint action in the highest levels of the hierarchy. The hierarchical approach enables the agents to learn to coordinate themselves faster by sharing information at the sub-task level instead of trying to coordinate themselves at the primitive action level. The authors of this approach have also proposed a new model MSMDP (Multiagent Semi-Markov Decision Process) to deal with cooperative actions in the hierarchy that may take some time to be completed (Ghavamzadeh and Mahadevan (2004)). This model is used in their *COM-Cooperative HRL* algorithm in which the agents' objective aims to learn a policy to optimize the communication needed for proper coordination, given the communication cost. The hierarchical decomposition enables their approaches to be applicable on big problems. Furthermore, since the agents in a hierarchical approach are only communicating at the highest levels and since actions at these levels normally take more time, then the agents are communicating less.

Now that we have presented some coordination learning algorithms based on reinforcement learning, in the next section, we explore other kinds of learning algorithms that interpret the agents' past experiences in order to improve their future experiences.

4.2.2 Coordination Learning Using Execution Traces

In the previous section, we presented reinforcement learning approaches in which agents learn by modifying the probability of taking a specific action in the given situation. In fact, the learning process is a statistical process using the rewards received.

Other types of learning methods try to interpret the obtained results in order to improve the agents' future performances. With such methods, agents analyze the past situations in order to find the causes of success or failure of their actions. To achieve that, agents use execution traces containing a lot of information recorded during the execution of their tasks. During the learning process, these execution traces are cleaned and generalized. They are then used to improve the agents' behavior.

Sugawara and Lesser have developed one of these methods. In this method, agents learn new behavioral rules from the failure situations, in order not to repeat them next time (Sugawara and Lesser (1995, 1998)). Agents record execution traces and when an undesirable situation happens, they analyze their traces to find the cause of the failure. Then, they add new behavioral rules so that this bad situation never happens again. Each agent individually learns its own set of rules. The rules specify what non-local information is needed in each situation to obtain a good coordination. By doing so, the agents learn what and when to communicate in order to optimize their success.

Another method using execution traces has been developed by Garland and Alterman in which agents learn *coordinated procedures* (Garland and Alterman (2001); Garland (2000)). In their approach, agents learn from the successful situations. In fact, their approach is a case-based approach where agents use the past cases to improve the coordination between them. The past cases, called *coordinated procedures*, are organized around *coordination points*. These coordination points represent moments during an activity when an agent cannot progress without the help of the other agents.

In fact, Garland and Alterman have used two coordination learning techniques in their work: coordinated procedures and operator probabilities. The coordinated procedures are partial plans constituted of coordination points and individual actions that have shown to be successful in the past. The operator probabilities give the success probability of each action. These probabilities are used during the planning phase to decide when the agents should cooperate and when they should adapt the coordinated procedures to the current situation.

In the next section, we present other learning methods that try to improve the agents' coordination. These methods do not really fit in the previous categories, and thus they are presented separately.

4.2.3 Other Learning Methods

Ahmadi et al. (2002) have developed an approach, applied in the RoboCupRescue, which learns iteratively an approximation of the value of a message. Agents then use these messages values to choose which messages to listen to. Their approach gave good results for the *PoliceForce* agents in the RoboCupRescue, but it really depends on the application; it uses a lot of constants set empirically.

Prasad and Lesser have worked on a method describing an instance based learning approach (COLLAGE) to learn to coordinate software agents (Prasad and Lesser (1999); Prasad (1997)). In their approach, agents start with a set of coordination strategies, and their objective is to learn to choose the best coordination strategy for all possible situations. To do this, the multiagent system is executed on many coordination problems and the agents record the performances of their coordination strategies on each situation presented. In their results, the authors show that agents improve their performances by choosing better coordination strategies. This method of learning to choose coordination strategies from a predefined set has also been explored by other researchers (Excelente-Toledo and Jennings (2002)). Notice that these methods are only applicable if it is possible to predefine a set of coordination procedures, which is not necessarily the case for complex coordination problems.

Horling and Lesser (1999) have worked on a way to diagnose a coordination problem to improve the learning process. Agents have a set of coordination rules which are represented using the task specification language TAEMS (Decker and Lesser (1993)). The authors use a causal model to link the coordination problem (e.g. exceeding time, wrong utilization of the resources, etc.) with the cause (e.g. broken resource, wrong estimation of a task duration, etc.). This enables the agent to have a more precise return from the environment, which helps to find the coordination rules to adjust.

Similarly, Jensen et al. (1999) have developed an approach using TAEMS which enables agents to use the task structure to learn the relations between different tasks. The objective is to learn the effects of one agent's actions on the other agents. This knowledge could be quite useful to coordinate agents by preventing conflicts and by taking advantage of the beneficial relations between the actions. In this context, agents can learn the following relations between the actions:

- *Enables*: the execution of a task enables another task to be executed;
- *Disables*: the execution of a task disables another task to be executed;
- *Facilitates*: the execution of a task facilitates or improves the execution of another task;
- *Hinders*: the execution of a task hinders the execution of another task;

In another approach, Bui et al. (1998) have developed a framework to deal with incomplete information. In this framework, agents learn a probability distribution for each source of uncertainty via repeated interactions. Bui and his colleagues have applied their approach in an appointment application. In this application, each agent has to learn a probability distribution on its user's preferences. Afterwards, each agent sends its learned information about its user to all other agents. These communications help the agents to have a better vision of the situation, and consequently a better coordination.

It is also possible, as shown by Haynes and Sen (1998), to use a case-based multiagent learning approach to learn complementary behaviors. In the Haynes and Sen approach, each agent starts with a set of basic behaviors that can be modified based on its interactions with the environment. When an action cannot be executed, it is considered as a negative case and the agent modifies its behavior rules so that it never happens again.

4.3 Tasks Allocation Learning: Our Motivations

Now that we have presented many coordination learning algorithms, in this section we present the motivations that led us to develop another coordination learning algorithm. Our main motivation was the RoboCupRescue simulation environment (see Chapter 2) in which we needed a coordination approach for the *FireBrigade* agents. These agents evolve in a complex environment with complex tasks and consequently they have to learn the right number of resources to assign to a task. Here is a list of the principal constraints that our algorithm has to manage:

- A learning algorithm is interesting because the dynamics of the RoboCupRescue environment can change;
- The algorithm has to be efficient with complex tasks described with many attributes;
- The algorithm has to deal with tasks having discrete and continuous variables;
- The information learned has to be general enough so that it could be applied to previously unseen task descriptions;
- The algorithm has to be as close as possible to the optimal, because there are few resources and many tasks.

In the cooperative multiagent learning domain, most researchers have focused on coordinating agents' actions, but most of them consider that the characteristics of the

tasks are known (Shehory and Kraus (1998); Excelente-Toledo and Jennings (2004)). Other approaches supposed that they have access to a probability distribution over the amount of resources needed for a task (Beynier and Mouaddib (2004)).

In our case, the required number of resources for each task is completely unknown. Thus we focused our attention on developing an algorithm enabling agents to learn this important task's characteristic. Some researchers considered that the number of resources is unknown, but for very simplified tasks requiring only one or two resources (Garland and Alterman (2004)). Here, we present a more general approach that allows us to deal with many resources in a complex task description space.

4.4 Application Domain

The resource allocation problem that motivated this work requires an efficient allocation of *FireBrigade* agents to the tasks of extinguishing fires. Therefore, throughout this chapter, we consider *FireBrigade* agents to be resources that are allocated to fires. These *FireBrigade* agents evolve in the RoboCupRescue simulation environment.

In this chapter, only the agents responsible for extinguishing fires are considered, i.e. the *FireBrigade* agents which extinguish fires and the *FireStation* agent which constitutes their communication center. All those agents are in contact by radio, but they are limited on the number of messages they can send as well as the length of those messages. Furthermore, in the simulation, each individual agent receives visual information of only the region surrounding it. Therefore, agents rely on the communication to acquire some knowledge about the environment. One should note that, since the center agent has more communication capabilities, it normally has a better global view of the situation than the *FireBrigade* agents.

As mentioned before, the task of the *FireBrigade* agents consists in extinguishing fires. Therefore, at each step in time, each *FireBrigade* agent has to choose which burning building to extinguish. However, in order to be effective, *FireBrigade* agents have to coordinate their choices about the burning buildings to extinguish, because more than one agent is often required to extinguish a building on fire. In our problem, we do not consider the importance of a task, and we suppose that the agents have a utility function allowing them to order the different fires according to the fires' priorities. Evidently, the agents' choices about the different burning buildings to extinguish depends on: (i) the number of available *FireBrigade* agents, (ii) their distance from those fires.

As previously stated in chapter 2, the *FireStation* agent has a better global view of the situation. Therefore it can suggest good fire areas to *FireBrigade* agents. On the other hand, the *FireBrigade* agents have a more accurate local view, consequently they

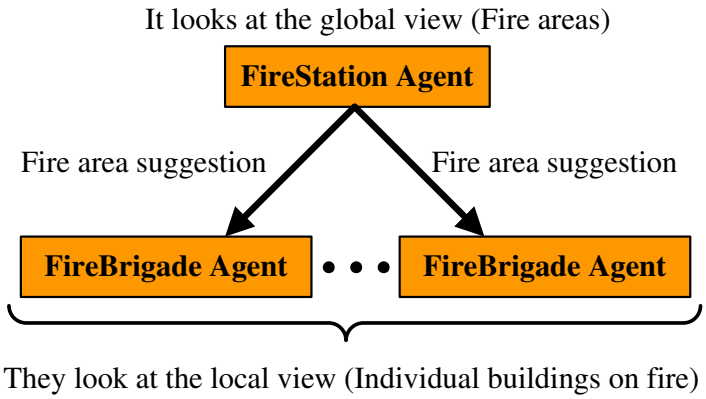


Figure 4.1: Collaboration between the *FireStation* and the *FireBrigade* agents.

can choose more efficiently which particular building on fire to extinguish in the given area (see Figure 4.1). By doing so, we can take advantage of the better global view of the *FireStation* agent and the better local view of the *FireBrigade* agent at the same time.

The main decision for the *FireBrigade* agents is to choose how many agents to send to the most important fires. However, the problem is that they do not know how many agents are required for each building on fire. The required number of agents depends on the characteristics of the building. For example, a small building on fire may require only two agents to extinguish it, but a bigger one may require 10 agents. In order to be effective, *FireBrigade* agents have to learn the number of agents required for each task (i.e., the task of extinguishing a building on fire) which is described as:

- the fire's fierceness (3 possible values),
- the building's composition (3 possible values),
- the building's size (continuous value),
- the building's damage (4 possible values).

In these conditions, there are many possible task descriptions. In fact, with the continuous attributes, there is an infinite number of task descriptions. With such a huge number of task descriptions, it is necessary to find an algorithm that can generalize the information learned on one task to similar tasks. In the next sections, we present our selective perception reinforcement learning algorithm which enables the *FireBrigade* agents to learn the best number of resources to extinguish each fire.

4.5 Problem Definition

In this chapter, we consider agents accomplishing tasks in an uncertain and dynamic environment where most tasks require more than one agent to be accomplished. In this case, agents are forced to coordinate their tasks' choices and they need to know the required number of agents to accomplish each task. This can be hard to estimate if subtle changes in a task description can change the required number of agents.

To make this estimation, we propose a new approach that uses a selective perception reinforcement learning algorithm to learn the expected reward if a certain number of agents tries to accomplish a certain type of task. An advantage of learning expected rewards instead of directly learning the number of agents is that the rewards can encapsulate the time needed to accomplish a task. A task taking more time to be accomplished has a smaller expected reward, due to the discount factor.

In our approach, an agent dynamically learns a tree representation of the task description space in order to reduce the number of task descriptions considered. This approach has been used before to find a compact representation of the state space to facilitate the definition of the agent's policy (McCallum (1996); Uther and Veloso (1998); Ron et al. (1994); Moore (1993); Chapman and Kaelbling (1991)).

In our work, we use approximately the same tree structure as the U-Tree algorithm (McCallum (1996)) to which we have made some modifications. Firstly, we do not use the tree to calculate Q-values for every possible basic action that an agent can take. We use the tree to calculate the expected reward of a particular goal decision of the agent. It still has to find the actions to accomplish this goal. In other words, the tree is used at the goal decision level, not at the action decision level.

To be more precise, we do not consider states, but task descriptions and our objective is not to find a policy for the agent, but to evaluate the capability of a given group of agents to accomplish a task. Therefore, the only implicit action is to accomplish a task, but it is never explicitly considered in the model. Our model can be described as a tuple $< D, N, R, T >$ where:

- D is the set of all possible task descriptions.
- N is the number of available agents.
- R is a reward function that gives the reward if a task is accomplished.
- T is a transition function that gives the probability to go from one task description to another. In other words, it gives the probability that the task description changes while some agents are accomplishing it.

The transition function is useful to take the dynamic aspects of the environment into consideration. It takes time to accomplish a task and, during this time, some characteristics of the current task may change and this may have an impact on the required number of agents.

Moreover, a task description is described in a factored way by a set of discrete or continuous attributes: $\{A_1, A_2, \ldots, A_n\}$. As previously stated, the number of different task descriptions can be huge, especially if there are continuous attributes. The primary objective of building a tree representation of the task description space is to reduce the number of task descriptions considered. The next section explains how the tree is built and how it is used to estimate the required number of agents to accomplish each task.

4.6 Tree Construction

Our algorithm uses a tree structure similar to a decision tree in which each leaf of the tree represents an abstract task description that regroups many task descriptions. This compact representation is iteratively expanded when new experiences are gathered by the learning agents.

At the beginning, all tasks are considered to be the same, so there is only the root of the tree. After each simulation, agents add new experiences to the tree and the tree is expanded. Those experiences are tasks that the agents tried to accomplish in the simulation with their associated rewards. All experiences are stored in the leaves of the tree. To expand the tree, the algorithm tests for each leaf l whether it would be interesting to divide the experiences stored in l by adding a new test on a task's attribute. The addition of a new test refines the agents' view of the task description space.

An advantage of this algorithm is that it distinguishes only tasks that really need to be distinguished. Therefore, the task description space is reduced, thus facilitating the reinforcement learning process.

4.6.1 Tree Structure

The algorithm presented here is an instance-based algorithm in which a tree is used to store all agents' experiences which are kept in the leaves of the tree. The other nodes of the tree, called center nodes, are used to divide the instances with a test on a specific attribute. Each leaf of the tree also contains a Q-value indicating the expected reward if a task that belongs to this leaf is chosen. In our approach, a leaf l of the tree is considered to be a task description (a state) for the learning algorithm.

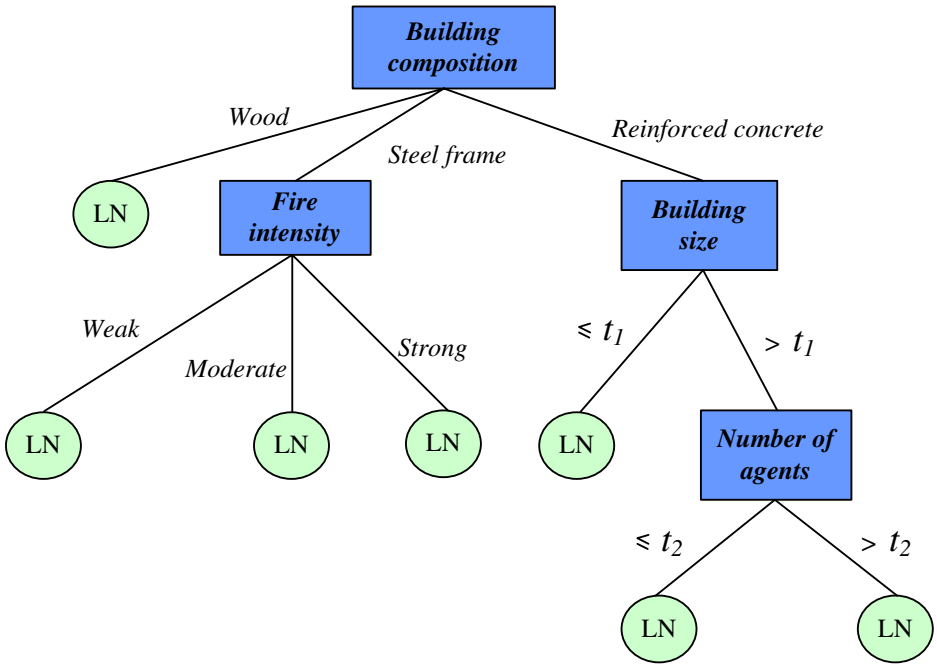


Figure 4.2: Structure of a tree.

An example of a tree is shown in Figure 4.2. Each rectangular node represents a test on the specified attribute. The words on the links represent possible values for discrete variables. The tree also contains a center node testing on a continuous attribute, the "Building size". A test on a continuous attribute always has two possible results, it is either less or equal to the threshold or greater than the threshold. The oval nodes (LN) are the leaf nodes of the tree where the agents' experiences and the Q-values are stored. Furthermore, in a complete tree, there are always many nodes "Number of agents". These nodes are used to evaluate the number of required agents for a task, as we will see later.

4.6.2 Recording the Agents' Experiences

In the RoboCupRescue, each simulation takes 300 time steps. During a simulation, each *FireBrigade* agent records, at each time step t, its experience about which fire it is trying to extinguish. More precisely, an experience is recorded as an *instance* that contains the task in consideration ($d_t \in D$), the number of agents that tried the same task (n_t) and the reward it obtained (r_t). Each instance also has a link to the preceding instance and the next one, thus making a chain of instances. Consequently, an instance at time t is defined as:

$$i_t = \langle i_{t-1}, d_t, n_t, r_t, i_{t+1} \rangle \tag{4.1}$$

In our case, d_t contains all the attributes describing a fire. Moreover, we have one chain for each fire that an agent chooses to extinguish. A chain contains all instances from the time an agent chooses to extinguish a fire until it changes to another fire. Therefore, during a simulation, each *FireBrigade* agent records many instances organized in many instance chains.

In the U-Tree algorithm (McCallum (1996)), there is only one chain of instances which links all instances in the simulation. In our case, it is better to use many instance chains because the tasks are independent. In fact, our concept of instance chains is closer to the concept of *episode* described by Xuan et al. (2004). Each task tried is seen as an independent episode.

Moreover, all those instances regrouped in many instance chains are only recorded during a simulation. Agents do not have time to learn during a simulation, because they have to act while respecting the real-time constraint of the RoboCupRescue simulation. Therefore, the learning process only takes place after a simulation, when the agents have time to learn. At this time, the *FireBrigade* agents regroup all their instances together, then the tree is updated with all those new instances and the resulting tree is returned to each agent. By regrouping their instances, agents can accelerate the learning process.

To sum up, all *FireBrigade* agents and the *FireStation* agent have the same tree learned from all the *FireBrigade* agents' experiences. Afterwards, as is explained in section 4.6.4, the *FireStation* agent uses the learned tree to assign *FireBrigade* agents to fire areas and each *FireBrigade* agent uses the learned tree to choose which fire to extinguished in the assigned area.

4.6.3 Update of the Tree

This section presents the algorithm used to update the tree using all the new recorded instances. Algorithm 4.1 shows an abstract version of the algorithm and the following subsections present each function used in more detail.

4.6.3.1 Add Instances

The first step is simply to add all the new instances, recorded by the *FireBrigade* agents, to the leaves they belong to (Algorithm 4.1, lines 2-4). To find those leaves, the algorithm starts at the root of the tree and heads down the tree choosing at each center node the branch indicated by the result of the test on the instance's attribute, which could be one of the attributes of the task description d or the number of agents n. When building the tree, the number of agents that tried to accomplish the task is

1: **Procedure** UPDATE-TREE(*Instances*)

Input: *Instances*: all instances to add to the tree.
Static: *Tree*: the tree.

2: **for all** i in *Instances* **do**
3: ADD-INSTANCE($Tree$, i)
4: **end for**
5: UPDATE-Q-VALUES($Tree$)
6: EXPAND($Tree$)
7: UPDATE-Q-VALUES($Tree$)

Algorithm 4.1: Algorithm used to update the tree.

considered as a normal task attribute. By doing so, the learning algorithm adds center nodes testing on the number of agents which will be useful later to evaluate the required number of agents to accomplish a task.

4.6.3.2 Update *Q*-values

The second step updates the Q-values of each leaf node to take into consideration the new instances which were just added (Algorithm 4.1, line 5). The objective here is to have precise Q-values when the time comes to expand the tree. The updates are done with the following equation:

$$Q'(l) \leftarrow \hat{R}(l) + \gamma \sum_{l'} \hat{T}(l, l')Q(l') \tag{4.2}$$

where $Q(l)$ is the expected reward if the agent tries to accomplish a task belonging to the leaf l, γ is the discount factor ($0 \leq \gamma \leq 1$), $\hat{R}(l)$ is the estimated immediate reward if a task that belongs to the leaf l is chosen, $\hat{T}(l, l')$ is the estimated probability that the next instance would be stored in leaf l' given that the current instance is stored in leaf l. Those values are calculated directly from the recorded instances. $\hat{R}(l)$ is the average reward of all the instances stored in leaf l. $\hat{T}(l, l')$ is the number of times that the following instance of an instance stored in leaf l is in leaf l', divided by the total number of instances in leaf l. More formally, here are the equations defining those values:

$$\hat{R}(l) = \frac{\sum_{i_t \in I_l} r_t}{|I_l|} \tag{4.3}$$

$$\hat{T}(l, l') = \frac{|\{i_t \mid i_t \in I_l \wedge L(i_{t+1}) = l'\}|}{|I_l|} \tag{4.4}$$

where $L(i)$ is a function returning the leaf l of an instance i, I_l represents the set of all instances stored in leaf l, $|I_l|$ is the number of instances in leaf l and r_t is the reward obtained at time t when n_t agents were trying to accomplish the task d_t.

To update the Q-values, the equation 4.2 is applied iteratively until the average squared error is less than a small specified threshold. The error is calculated using the following equation, which is the average squared difference between the new and the old Q-values:

$$E = \frac{\sum_{l}(Q'(l) - Q(l))^2}{nl} \tag{4.5}$$

where nl is the number of leaf nodes in the tree.

4.6.3.3 Expand the Tree

After the Q-values have been updated, the third step checks all leaf nodes to see if it would be useful to expand a leaf and replace it with a new center node (Algorithm 4.1, line 6). The objective is to divide the instances more finely and to refine the agent's representation of the task description space, in order to help the agent to predict rewards.

To find the best test to divide the instances in each leaf, the agent tries all possible tests, i.e. it tries to divide the instances according to each attribute describing a task or the number of agents. After all attributes have been tested, it chooses the attribute that maximizes the error reduction as shown in equation 4.6 (Quinlan (1993b)).

The error measure considered is the standard deviation ($sd(I_l)$) on the instances' expected rewards. Therefore, a test is chosen if, by splitting the instances, it ends up reducing the standard deviation on the expected rewards. If the standard deviation is reduced, it means that the rewards are closer to one another. Thus, the tree moves toward its objective of dividing the instances in groups with similar expected rewards, in order to help the agent to predict rewards. In fact, the test is chosen only if the expected error reduction is greater than a certain threshold, if not, it means that the test does not add enough distinction, so the leaf is not expanded.

The expected error reduction obtained when dividing the instances I_l of leaf l is calculated using the following equation where I_k denotes the subset of instances in I_l that have the k^{th} outcome for the potential test:

$$\Delta error = sd(I_l) - \sum_{k} \frac{|I_k|}{|I_l|} sd(I_k) \tag{4.6}$$

The standard deviation is calculated on the expected reward of each instance which

is defined as:

$$Q_I(i_t) = r_t + \gamma \hat{T}(L(i_t), L(i_{t+1}))Q(L(i_{t+1})) \tag{4.7}$$

where $\hat{T}(L(i_t), L(i_{t+1}))$ is calculated using equation 4.4 and $Q(L(i_{t+1}))$ using equation 4.2.

As mentioned earlier, one test is tried for each possible instance's attribute. For a discrete attribute, we divide the instances according to their value for this attribute. For instance, if an attribute has three possible values, it generates three subsets, thus adding three children nodes to the tree. We then use Equation 4.6 and record the error reduction for this test. For a continuous attribute, we have to test different thresholds to find the best one. A continuous attribute always divides the instances into two subsets, the first one is for the instances with a value less or equal to the threshold for the specified attribute and the second subset is for the instances with a value greater than the threshold.

To find the best threshold, we have used the technique described by Quinlan (1993a). The instances are first sorted according to their value for the attribute being considered. Afterwards, we examine all $m - 1$ possible splits, where m is the number of different values. For example, with an ordered list of values $\{v_1, v_2, ..., v_m\}$, we try all possible thresholds. So, we try the value v_1 as a threshold, thus dividing the instances in two subsets, those less or equal and those greater than v_1. We calculate and record the error reduction for this division. Then, we do the same thing for the other possible values, v_2 to v_{m-1}. At the end, we keep only the threshold with the best error reduction value.

There are different ways that can be used to expand the tree (McCallum (1996); Uther and Veloso (1998); Pyeatt and Howe (1995)), but we have used an approach that has been shown to be effective in decision tree algorithms (Quinlan (1993b)) and that enabled us to have a fast algorithm and to consider continuous attributes.

At the end, when the tree has been updated, the UPDATE-Q-VALUES function is called again to take the new tree structure into consideration (Algorithm 4.1, line 7). The updates are done exactly the same way as in section 4.6.3.2.

4.6.4 Use of the Tree

During a simulation, all learning agents use the same learned tree to estimate the number of agents that are required to accomplish a task. Since the number of agents is considered as an attribute when the tree is learned, if different numbers of agents are tested, different leaves and thus different rewards may be found, even with the same task description. Consequently, to find the required number of agents for a particular task, the algorithm can test different numbers of agents and look at the expected rewards returned by the tree.

1: **Function** NUMBER-AGENTS-REQUIRED(d, N) **returns** an integer

Inputs: d: a task description.
N: the number of available agents.
Statics: *Tree*: the tree learned.
Threshold: the limit to surpass.

2: **for** $n = 1$ to N **do**
3: $expReward \leftarrow$ EXPECTED-REWARD($Tree$, d, n)
4: **if** $expReward \geq Threshold$ **then**
5: **return** n
6: **end if**
7: **end for**
8: **return** ∞

Algorithm 4.2: Algorithm used to find the required number of agents for a given task description d.

Algorithm 4.2 presents the function used to estimate the required number of agents for a given task. In this algorithm, the function EXPECTED-REWARD at line 3 returns the expected reward if n agents are trying to accomplish a task described as d. To this end, it finds the corresponding leaf in the tree, considering the task description d and the number of agents n, and records the expected reward for this abstract task.

The EXPECTED-REWARD function is called for all possible numbers of agents until the expected reward returned by the tree is greater than a specified *Threshold*. If the expected reward is greater then the *Threshold*, it means that the current number of agents should be enough to accomplish the task. If the expected reward is always under the *Threshold*, even with the maximum number of agents, the function returns ∞, meaning that the task is considered impossible with the available number of agents N. The *Threshold* value is set empirically and it corresponds to the minimum expected reward needed to accomplish a task. The agent stops when the *Threshold* is exceeded because the objective here is to find the minimum number of agents for each task.

4.6.5 Algorithm Characteristics

First of all, let's look at the algorithm complexity. The first step of the algorithm used to update the tree is to add all the new instances recorded during the last simulation. In the worst case, this step takes $O(|I|D_{\max})$, where $|I|$ is the number of instances to add and $D_{\max}$ is the maximal depth of the tree. Therefore, as the tree grows, this step takes more time. However, in our RoboCupRescue experiments, the maximum depth of the tree was always relatively small (≤ 30).

The second step is to update the Q-values using Equation 4.2. The reward and the transition function (Equations 4.3 and 4.4) do not take time, because they are simply updated each time a new instance is added to a leaf. The complexity of Equation 4.2 depends on the number of subsequent leaf nodes link to the current leaf node. To update the Q-values, the algorithm has to visit all the leaves and in the worst case, all the leaves are connected together, thus the complexity is $O(|L|^2)$, where $|L|$ is the number of leaves in the tree. This complexity is multiplied by the number of iterations needed until convergence of the Q-values. Since the Q-values are normally only slightly modified when the new instances are added, it generally does not take a lot of iterations to converge. Most of the time, it took less then 10 iterations in our tests.

The third step of the algorithm is to expand the tree. To achieve that, the algorithm has to visit all the leaves of the tree one time. For each leaf, it has to evaluate all possible splits using all the attributes describing a task. For a discrete attribute, there is only one split to try. For a continuous attribute, there are as many possible splits as there are different values for this continuous attribute in the current leaf. In the worst case, there are as many attribute values as there are instances in the leaf. Therefore, the complexity in the worst case is: $O(|L|n_d n_c|I|_{\max})$, where $|L|$ is the number of leaves in the tree, n_d and n_c are the number of discrete and continuous variables used to describe a task and $|I|_{\max}$ is the maximum number of instances in a leaf.

The last step of the algorithm does another update of the Q-values. Consequently, the total complexity of the algorithm is: $O(|I|D_{\max}) + 2O(|L|^2) + O(|L|n_d n_c|I|_{\max})$. The most expensive step is the expansion step, because it manipulates all the instances stored in all the leaves of the tree. In our tests, the time to update the tree was not a big factor since it was executed offline. It always took less time to learn the tree than to run a simulation.

Moreover, for this algorithm, we suppose that all the *FireBrigade* agents have the same vision of the situation, i.e. they see the same fires. In the RoboCupRescue it is almost always the case, because the agents are close to one another when they are extinguishing fires and the fires can be seen from a far distance.

Another hypothesis is that all the agents have the same learned tree. The tree is learned offline after each simulation, using all the instances gathered by the *FireBrigade* agents during the simulation. The communications between the agents are really limited during a simulation, thus we wanted the agents to have some common ground on which to base their decisions in order to reduce the amount of communication necessary to maintain the coordination between the agents.

In the next section, we present some experiments in which we described in more detail how the learned trees are used. We also present results showing the quality of the solutions found and the speed at which the tree grows when we add new instances.

Figure 4.3: Example of fire areas. There are four active fire areas on this map, which are identified by the circles.

4.7 Experiments

In this section, we present how our learning algorithm, previously presented, can be used in the RoboCupRescue simulation to help the *FireBrigade* agents to coordinate themselves on the buildings on fire to extinguish. All tests have been made on the RoboCupRescue simulator used at the 2004 international competition. Since there could be a lot of fires, agents do not consider all fires at once. They choose separately which fire area to extinguish and which specific building in the chosen fire area to extinguish. Fire areas are simply groups of close buildings on fire. Figure 4.3 shows an example of a situation with four fire areas. To make their decision, agents use the tree created offline to estimate the required number of agents for each building on fire. All agents have the same tree and it does not change during a simulation.

As previously stated in section 4.4, the *FireStation* agent has a better global view

of the situation and therefore it can suggest fire areas to *FireBrigade* agents. The *FireBrigade* agents have however a more accurate local view, consequently they choose which particular building on fire to extinguish in the given area. By doing so, we can take advantage of the better global view of the *FireStation* agent and the better local view of the *FireBrigade* agent at the same time. In the next two sub-sections, we present how the *FireStation* agent chooses the fire areas and how the *FireBrigade* agents choose the buildings on fire to extinguish.

4.7.1 Fire Areas Allocation

To allocate the fire areas, the *FireStation* agent has a list of all fire areas; see Algorithm 4.3. For each fire area, it has to estimate the number of agents that are required to extinguish this area. To achieve that, it makes a list of all the buildings that are at the edge of the fire area (line 8). Agents only consider buildings at the edge because those are the buildings that have to be extinguished to stop the propagation of the fire. For each burning building at the edge, the *FireStation* agent finds the number of agents required to extinguish the fire (to do so, at line 12, Algorithm 4.3 calls Algorithm 4.2). The agent then estimates the required number of agents for the fire area as the maximum number of agents returned for one building in the area. The *FireStation* agent does the same thing with all fire areas, ending up with a number of agents (n_z) for each area z.

Afterwards, for each area z, the *FireStation* agent calculates the average distance of the n_z closest *FireBrigade* agents from z (line 17). Then, it chooses the fire area z with the smallest average distance. Consequently, the n_z closest *FireBrigade* agents from the chosen area z are assigned to z. The *FireStation* agent then removes the assigned area and *FireBrigade* agents from its lists (lines 25-26) and continues the process with the remaining agents and the remaining fire areas. It continues until there is no agent or fire area left. At the end, the *FireStation* agent sends each *FireBrigade* agent their assigned fire area.

4.7.2 Choice of Buildings on Fire

To choose a building to extinguish (see Algorithm 4.4), a *FireBrigade* agent builds a list of all the buildings on fire in the fire area specified by the *FireStation* agent. This list is sorted according to a utility function that gives an idea about the usefulness of extinguishing a fire (line 2). The utility function $U(f_i)$ gives a value to a fire f_i based on the buildings and the civilians in danger if f_i propagates to buildings close by. The utility function considers all buildings in danger by the given fire f_i. Buildings

```
1: Function ASSIGN-FIRE-AREAS(Agents, FireAreas) returns a fire area per agent
   Inputs: Agents: a list of all available agents.
           FireAreas: a list of all fire areas.
2: while FireAreas.size() > 0 ∧ Agents.size() > 0 do
3:   nbAgents ← Agents.size()
4:   smallestDistance ← ∞
5:   chosenArea ← null
6:   chosenAgents ← null
7:   for each fireArea in FireAreas do
8:     borderBuildingsList ← GET-BORDER-BUILDINGS(fireArea)
9:     n_z ← 0
10:    for each borderBuilding in borderBuildingsList do
11:      d ← GET-TASK-DESCRIPTION(borderBuilding)
12:      nbRequiredAgents ← NUMBER-AGENTS-REQUIRED(d, nbAgents)
13:      if n_z < nbRequiredAgents then
14:        n_z ← nbRequiredAgents
15:      end if
16:    end for
17:    ⟨averageDistance, listAgents⟩ ← AVERAGE-DISTANCE(Agents,n_z)
18:    if smallestDistance > averageDistance then
19:      smallestDistance ← averageDistance
20:      chosenArea ← fireArea
21:      chosenAgents ← listAgents
22:    end if
23:  end for
24:  agentsAssigned[chosenArea] ← chosenAgents
25:  FireAreas ← FireAreas − chosenArea
26:  Agents ← Agents − chosenAgents
27: end while
28: return agentsAssigned
```

Algorithm 4.3: Algorithm used by the *FireStation* agent to allocate a fire area to each *FireBrigade* agent.

in danger are near buildings which are not on fire, but that may catch fire if fire f_i is not extinguished. For each building in danger, the utility function returns the amount of points lost if the building in danger catches fire. The utility function uses the official score function (Equation 2.1 at page 16) and it considers that the building will completely burn and that civilians trapped in it will die. More formally, the utility

function $U(f_i)$ is calculated using the following equations:

$$\begin{aligned} U(f_i) &= \sum_{b \in D(f)} (Score_{ini} - ScoreLost(b)) \\ Score_{ini} &= \left(A + \frac{H_{ini}}{H_{ini}}\right) \sqrt{\frac{B_{ini}}{B_{ini}}} = A + 1 \\ ScoreLost(b) &= \left(A - nCiv(b) + \frac{H_{ini} - sumHP(b)}{H_{ini}}\right) \sqrt{\frac{B_{ini} - area(b)}{B_{ini}}} \end{aligned} \tag{4.8}$$

where, $D(f_i)$ is the set of all buildings in danger from the fire f_i, $Score_{ini}$ is the initial score at the beginning of the simulation, $ScoreLost(b)$ is the score lost if the building in danger b catches fire, $nCiv(b)$ returns the number of civilians trapped in b, $sumHP(b)$ returns the sum of the health points (HP) of all the civilians trapped in b and $area(b)$ returns the area of b.

All *FireBrigade* agents have approximately the same list of buildings on fire. To choose their building on fire they go through the list, one building at a time. For each building, they use the tree to find the expected required number of agents to extinguish the fire (to do so, at line 6, Algorithm 4.4 calls Algorithm 4.2).

The *FireBrigade* agents choose the burning buildings following a prefixed order given to them at the beginning of the simulation. Knowing the sorted list of buildings on fire, the order of all *FireBrigade* agents and the number of agents required for each fire, each *FireBrigade* agent can choose the fire it should extinguish. For example, suppose that there are two fires requiring 5 and 3 agents respectively and that the *FireBrigade* agent A has to choose one of them. If the agent A's rank is 3, it would choose the first fire, because the five first agents have to go to the first fire. However, if the agent A's rank is 7, it would choose the second fire, because the agents ranked 6, 7 and 8 have to go to the second building on fire. With this coordination process, if all *FireBrigade* agents actually have the same information, they should be well coordinated on which buildings to extinguish.

4.8 Results and Discussion

As mentioned before, experiments have been done in the RoboCupRescue simulation environment. We have made our tests on a situation with a lot of fires, but with all roads cleared. The simulations started with 8 fires, but the agents began to extinguish fires only after 30 simulation steps (to allow fires to propagate). Figure 4.4 shows a view of the city at time 30, just before the *FireBrigade* agents begin to work. This gave us a hard situation to handle for the *FireBrigade* agents. Those agents started with an

```
1: Function CHOOSE-FIRE(Fires, nbAgents, rank) returns a fire
   Inputs: Fires: a list of all fires.
           rank: the agent's rank in the group of FireBrigade agents.
           nbAgents: the number of agents assigned to the same fire area.
2: sortedFires ← SORT-FIRES(Fires)
3: currentIndex ← 0
4: for each fire in sortedFires do
5:    d ← GET-TASK-DESCRIPTION(fire)
6:    nbRequiredAgents ← NUMBER-AGENTS-REQUIRED(d, nbAgents)
7:    currentIndex ← currentIndex + nbRequiredAgents
8:    if currentIndex ≥ rank then
9:       return fire
10:   end if
11: end for
12: return first fire in sortedFires
```

Algorithm 4.4: Algorithm used to choose a fire in the fire area specified by the *FireStation* agent.

empty tree and they learned from one simulation to another to distinguish the tasks that had to be distinguished and the expected rewards associated with those tasks.

Our experiments have been done with the simulator used in the 2004 RoboCup-Rescue international competition. With this simulator, the attributes used to describe a task d were:

- the fire's fierceness (3 possible values),
- the building's composition (3 possible values),
- the building's size (continuous value),
- the building's damage (4 possible values).

All these attributes lead to a number of possible instances which is quite important. In fact, with the continuous attribute "building's size", there is an infinite number of instances. However, since the tests were done only in one city, the number of different buildings was only 730 in our tests. In the case where we consider 15 agents, the number of possible instances for our tests climbed up to 394 200 ($3 \times 3 \times 730 \times 4 \times 15$).

We have compared the results obtained by our agents with two other strategies, as described in Figure 4.5). The first one is the strategy of the team ResQFreiburg

Figure 4.4: Initial situation.

(Brenner et al. (2005)) which finished first at the 2004 RoboCupRescue simulation world competition. This team used data-mining techniques to evaluate the propagation of the fires and a priority function to choose which fire to extinguish. If we look at the performance of ResQFreiburg on our test map, they only obtained an average percentage of intact buildings of 59%.

The second one is our strategy, but without the learning part, thus all agents choose the first building on the list. The comparison described in Figure 4.5 shows the advantage of learning the required number of agents to accomplish a task. If all agents go to the same building, they obtained an average percentage of intact buildings of 63% and after learning, they obtained 84%. This is a substantial improvement showing that the information learned is really useful. They learned how many agents are required for all possible situations so they were able to divide the tasks efficiently between them. Notice that the substantial improvement is mainly due to the fact that agents are able to split themselves on the first two or three tasks and accomplish them all at once.

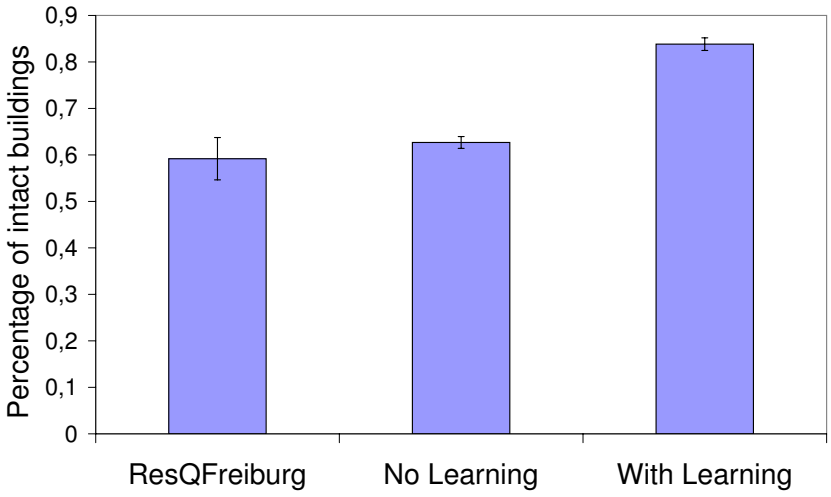


Figure 4.5: Comparison with other strategies.

Thus, the more efficient they become at estimating the required number of agents, the faster they become at accomplishing all their tasks.

Another interesting result is that our agents were able to attain such good performances with trees having less than 2000 leaves. Therefore, they just distinguished 2000 task descriptions out of the 394 200 possible task descriptions. In other words, our agents were able to perform efficiently with an internal task description space of only 0.5% of the complete task description space (the percentage is the number of leaves in the tree divided by the number of possible task descriptions). This shows a very good reduction of the task description space, enabling the learning algorithm to work on an easier problem with less possible states.

Moreover, Figure 4.6 presents results with different γ values for the equation 4.2. It shows the evolution of the percentage of intact buildings at the end of a simulation. Every point represents an average over 10 simulations. We have tested all γ values from 0 to 1 with an increment of 0.1, but here, for an improved visibility, we present only four representative learning curves. As we can see, with high γ values, the learning process is less effective. With 0.9, the agents do not get better at all, and with 1, it was even worse. With a value of 0.8, the agents have some trouble learning at the beginning, but eventually they start to catch up after approximately 70 simulations. We have observed that with a higher gamma value, agents normally need more time to learn. With the small γ values of 0.3 and 0.5, the learning curves are quite smooth. The best γ value we found was 0.5. Since we obtained better results with γ values greater than 0, it shows that it is important to consider future rewards. However, since we obtained better results with smaller γ values, it shows that it is not efficient to consider rewards too far away. This is the case in the RoboCupRescue simulation, because the simulation evolves fast and the fires have to be extinguished rapidly. With bigger γ

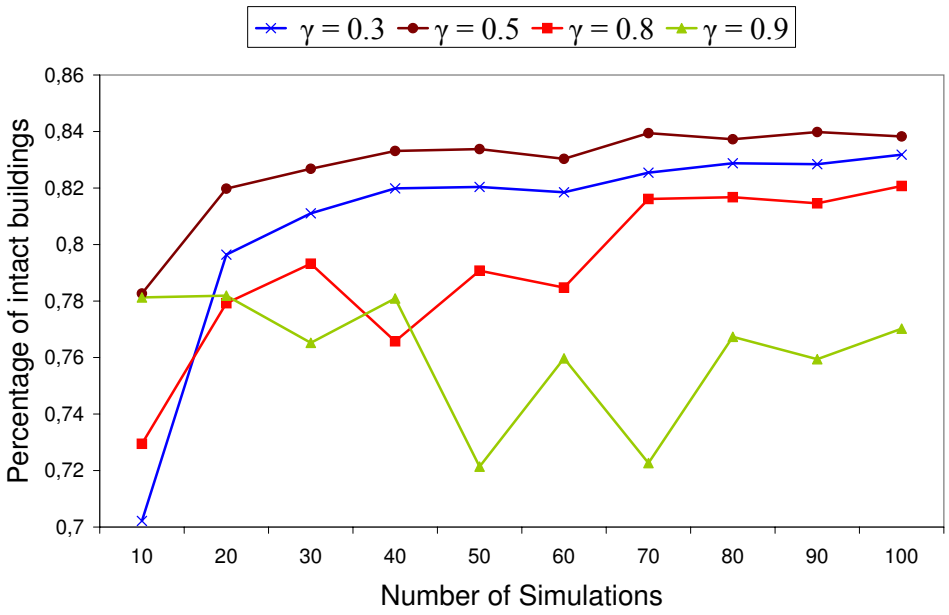


Figure 4.6: Percentage of intact buildings over 100 simulations for different γ values.

values, the agents choose buildings that take too much time to extinguish, because they consider far rewards.

In each simulation, agents were able to gather approximately 2000 instances. Of course, with our algorithm, the necessary memory always grows to store all those instances, but since they did not take too much space, this was not really a problem. Moreover, after the learning phase, the instances are not necessary anymore, therefore when the tree is used at the execution time it is really small.

Since we are expanding a tree, the growth could be exponential. However, it is not the case, because we are only expanding leaves that help to predict rewards. Therefore, the growth of the tree is controlled and in our tests it was even sub-linear (see Figure 4.7).

Another import result which supports this approach is our performance in the 2004 RoboCupRescue international competition. We finished in second place at this competition, really close to first place. One part of our success is due to the fact that we were quite good at extinguishing fires. Table 4.1 shows the percentage of saved buildings for all the maps used during the competition. In this table, the results of all the semifinalist teams are presented. We can see that our team (DAMAS) was the best team on 5 maps, which is the best score among the participants.

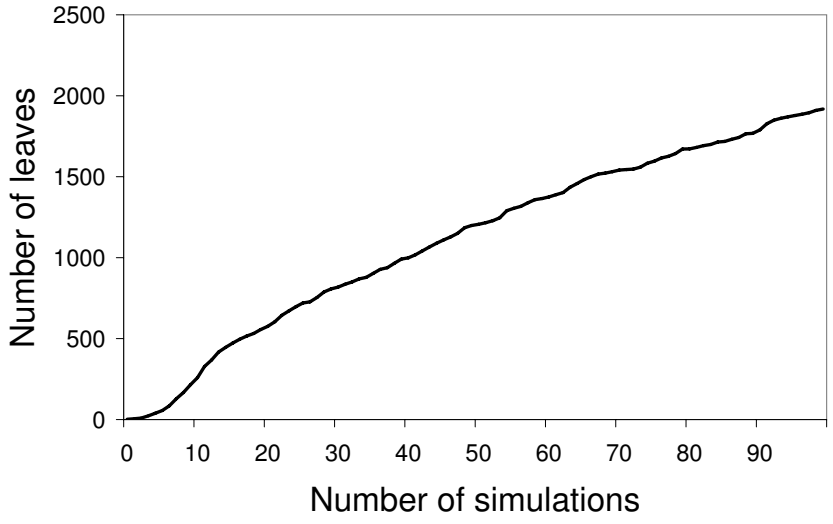


Figure 4.7: Number of leaves in the tree over 100 simulations.

	ResQ	Damas	Caspian	BAM	SOS	SBC	ARK	B.Sheep
Final-VC	47,21	54,13	**81,67**	43,19	N/A	N/A	N/A	N/A
Final-Random	24,04	**26,38**	15,03	12,35	N/A	N/A	N/A	N/A
Final-Kobe	38,24	**61,89**	38,38	13,51	N/A	N/A	N/A	N/A
Final-Foligno	**91,15**	62,77	60,92	34,56	N/A	N/A	N/A	N/A
Semi-VC	23,45	23,60	25,49	27,14	19,12	25,10	26,36	**27,22**
Semi-Random	23,18	**28,73**	18,09	19,55	22,82	21,45	17,09	18,91
Semi-Kobe	**96,49**	76,76	94,32	95,41	24,32	90,54	55,27	94,19
Semi-Foligno	36,22	**38,06**	32,72	37,79	31,89	28,48	26,82	23,23
Round2-Kobe	70,27	37,03	59,73	95,41	48,38	61,49	10,54	**95,54**
Round2-Random	99,04	60,91	54,68	99,16	63,55	97,60	80,70	**99,52**
Round2-VC	10,23	11,57	10,23	13,53	12,67	**71,99**	N/A	36,51
Round1-Kobe	99,46	98,92	**99,73**	**99,73**	99,05	98,78	67,16	91,89
Round1-VC	97,25	99,53	79,70	**99,76**	N/A	98,90	99,53	99,53
Round1-Foligno	**98,99**	**98,99**	36,13	45,99	32,53	54,29	43,59	29,86
Number of Wins	3	**5**	2	2	0	1	0	3

Table 4.1: Percentage of saved buildings during the 2004 RoboCupRescue international competition. Results reported by Kleiner et al. (2006).

4.9 Contributions

In this chapter, we have presented a learning algorithm which is useful to learn the required number of agents for each different task in a complex cooperative multiagent environment. In this last section, we summarize our contributions:

Learning required resources. Most coordination learning approaches consider that the number of required resources to accomplish a task is known or that they have enough information to have a probability distribution over the number of required resources. In our approach, we consider that the tasks are complex and that the agents have to learn this information since it is not available.

Reduction of the task description space. The tasks considered in our simulations are described with discrete and continuous attributes. Therefore, there are a lot of possible task descriptions. To manage this complexity, we have adapted a selective perception reinforcement learning algorithm to the problem of learning the required number of resources to accomplish a task. With this algorithm we can find a generalization of the task description space, which helps the reinforcement learning algorithm to work on smaller task description spaces.

Splitting criteria. Unlike the basic U-Tree algorithm, our algorithm can deal with discrete and continuous attributes. The splitting criteria is inspired from decision trees algorithms to make sure that similar task descriptions are stored in the same nodes. We also present a method to dynamically find good thresholds for the continuous variables.

Coordination algorithm. We proposed a coordination algorithm using the information learned about the number of resources needed for a task. This algorithm uses really few messages between the agents, which is interesting in environments with limited and/or unreliable communications.

Experiments. We have presented some tests in the RoboCupRescue environment showing that the agents can efficiently learn and that the learned information is really helpful to improve the agents' performances. The agents obtained good results with an internal task description space of only 0.5% of the complete task description space. We have also shown results taken during the 2004 international competition showing that we were the most efficient team in extinguishing fires.

Chapter 5

Task Scheduling in Complex Multiagent Environments

In complex multiagent systems, the agents could be faced with many tasks to accomplish. Moreover, when the tasks have different deadlines, the order in which the tasks are accomplished becomes quite important. In such settings, agents have to decide how many agents to assign to each task and in which order they should accomplish the tasks. To achieve that, agents need a good scheduling algorithm that can maximize the number of tasks accomplished before their deadline.

The common understanding of scheduling in Artificial Intelligence is that it is a special case of planning in which the actions are already chosen, leaving only the problem of determining a feasible order. This is an unfortunate trivialization of scheduling (Smith et al. (2000)). In this chapter, we define scheduling as the problem of assigning limited resources to tasks over time to optimize one or more objectives (Dean and Kambhampati (1996); Mali and Kambhampati (1999)). We restrain ourselves to scheduling problems in which all tasks have the same value and where the set of tasks is not accomplishable in the time allowed. To link the domains of multiagent systems and scheduling systems, we consider agents to be resources that can accomplish tasks. We talk about an agent accomplishing a task and about the allocation of an agent to a task. In other words, these are scheduling problems in which the agents have to schedule the tasks in order to maximize the number of tasks accomplished.

Furthermore, in multiagent systems, the scheduling can be done in a centralized or decentralized way (Durfee (1999); Paquet et al. (2005a)). Likewise, the execution can also be done in a centralized or decentralized way. Centralized scheduling means that there is one agent responsible for scheduling the tasks of all the agents. On the other hand, in decentralized scheduling each agent is responsible for the scheduling of its tasks. Moreover, an execution is considered distributed if the agents can accomplish

many tasks in parallel. If all agents always accomplish the same task together, then it is considered a centralized execution.

Moreover, in a dynamic environment, the scheduling process has to be able to adapt the schedule frequently to take the changes into consideration (Vieira et al. (2003)). Another challenge is when the environment is partially observable, because then the tasks are not known at the beginning, thus the agents have to explore the environment to find the tasks and then incorporate them in their schedule. Notice that since we are considering uncertain environments, the tasks' parameters could even change between two observations. In this case, the system's performance depends not only on the maximization of the optimization criterion, but also on the agents' capacity to adapt their schedule efficiently. The scheduling task then becomes a reactive process (Smith (1994)), because agents have to react to changes in the environment by adjusting their schedule.

In this chapter, we analyze the advantages and the disadvantages of distributing or not the scheduling process in a complex multiagent system. More precisely, we study the impact on the agents' efficiency and on the amount of information transmitted when using centralized and decentralized scheduling. We also study the usefulness of distributing the execution of the tasks in a scheduling problem and thus accomplishing goals in parallel, compared to the strategy of concentrating all resources to accomplish one goal at a time.

In similarity with the approach presented in Chapter 4, we also had to learn to estimate one of the characteristics of the tasks. In this chapter, we consider the work of the *AmbulanceTeam* agents, which are responsible to rescue the civilians in the RoboCup-Rescue simulation. One important parameter that they have to learn is the expected death time of the civilians. This parameter is quite important for the *AmbulanceTeam* agents to make good schedules. To this end, we present the K-Nearest-Neighbors (KNN) approach that has been used to learn the damage progression of the civilians which is used to estimate the expected death time.

This chapter is organized as follows. First, the scheduling approach that has been used is presented, followed by the K-Nearest-Neighbors learning algorithm. Finally, we present results showing the performances of this scheduling approach in the RoboCup-Rescue simulation environment.

5.1 A New Methodology for Task Scheduling in Complex Multiagent Environments

In a task scheduling system, we generally use a set of resources to accomplish a set of tasks in an order maximizing an optimization criterion (French (1982)). For example, we could want to accomplish the set of tasks as fast as possible or accomplish as many tasks as possible in a given time. The major problem in task scheduling systems is to decide how to distribute the resources efficiently and in which order. This thesis focuses on multiagent systems in which the work of some agents can be described as a task scheduling system. So, agents are considered as resources that can complete tasks. Evidently, not all multiagent systems can be modeled as a task scheduling system. In several cases however, such approach can be useful, particularly in environments where agents have to accomplish tasks and the order of these tasks influences the efficiency of all the multiagent system.

These kinds of task scheduling systems raise many questions: To which model does our task scheduling system correspond? Which scheduling algorithm to use? Who is responsible for the scheduling? With all these questions, it is possible to get lost in the search for answers. This is why we use a methodology to answer these questions in a structured way. This work methodology has three steps which should be handled in this order:

1. Scheduling problem definition.
2. Scheduler type definition.
3. Scheduling algorithm definition.

5.1.1 First Step: Scheduling Problem Definition

The task scheduling problem has to be defined in the first step. This means that we have to extract the scheduling problem from the multiagent system and to define it formally. Firstly, we have to identify the characteristics of the scheduling problem which consist of defining the set of tasks to execute, the task's parameters (execution cost, deadline, release time, etc.) and the optimization criterion to maximize. When the scheduling problem has been carefully analyzed, we can formalize it. This formalization is not absolutely necessary, but it facilitates the presentation and the comprehension of the problem.

A lot of notations for task scheduling problems exist because these kinds of problems have been widely studied. In particular, in the context of task allocation for systems

with one or many processors (Gonzalez (1977)) and in distributed computing (Norman and Thanisch (1993)). Among all existing notations, those used in industrial engineering seemed to be the most useful to multiagent systems. This notation defined by Lawer et al. (1982) is used in many books and articles that discuss scheduling theory (Pinedo (1995); Blazewick (2001)). We briefly present the notation to help the reader understand the description of the scheduling systems used in this thesis. We do not present the notation exhaustively, we only concentrate on the symbols that are used to defined our scheduling problems.

A scheduling problem, with the goal of managing a set of tasks T, is described with three fields separated with the character "|", as in: $\alpha \mid \beta \mid \gamma$.

- α : the machines' environment. This field tells how many machines are present and what their characteristics are. This field can be filled with the following symbols:

  1 : Mono-machine environment.
  P_m : Multi-machines environment where m represents the number of machines.

- β : the constraints and the characteristics. This field tells for example, if the tasks have deadlines or if there is a cost to change from one task to another. It is possible that this field stays empty when constraints and characteristics are implicitly defined in the field γ. This field can be filled with the following symbols:

  p_j : The execution cost of the task j.
  d_j : The deadline of the task j.
  s_{jk} : The cost to change from task j to task k.

- γ : the optimization criterion. This field defines what the scheduler is supposed to optimize, for instance the number of tasks executed before a deadline ($\sum U_j$) or the time to complete all the tasks ($\sum C_j$).

  $\sum C_j$: The sum of completion times of all tasks.
  $\sum U_j$: The sum of unit penalty of all tasks where:

$$U_j = \begin{cases} 1 & \text{if } C_j > d_j \\ 0 & \text{if not} \end{cases}$$

In other words, $\sum U_j$ means that the scheduler tries to minimize the number of tasks exceeding their deadline, i.e. tasks for which their completion time C_j is greater than their deadline d_j.

In this notation, a machine corresponds to a resource used to complete tasks. For example, a multi-machine system means that there is more than one resource to complete tasks, or many agents in a multiagent context. In our approach, the terms "resource", "agent" and "machine" are all synonyms. Because of the multiagent purpose of this thesis, we will henceforth use the word "agent" to indicate an entity that can accomplish a task.

5.1.2 Second Step: Scheduler Type Definition

The scheduler, in a scheduling system, represents the abstraction level where the tasks ordering is decided to maximize the optimization criterion. To build a good schedule, such a system has to have a good knowledge of the environment's elements describing the tasks.

We can create two main categories of scheduler: centralized and decentralized. In the centralized approach, the schedule of the tasks is done by only one agent. This agent has to schedule and distribute the tasks for all the agents. To do that, it needs a global knowledge of the environment that it can acquire by exploration or by inter-agent communication. Especially in a dynamic and partially observable environment, this can demand a lot of messages because the state of the environment has to be transferred frequently to regroup all the agents' perceptions (Xuan and Lesser (2002)). Agents have therefore to find the right level of messages to exchange in order to be well coordinated without sending everything they know (Xuan et al. (2001)).

The main problem with the centralized scheduler is that it generally depends a lot on communications. When the communications are limited, the performances of such an approach deteriorate. Furthermore, the loss of the centralized scheduler agent can be catastrophic.

In the distributed approach, the schedule of the tasks is not under the responsibility of one agent, but many agents (Jones and Rabelo (1998)). Each agent schedules its own tasks according to what it knows about the environment. However, to stay coordinated, the agents need to synchronize themselves using communications, transferring only the information pertinent to the synchronization. In theory, the distributed approach needs less communications and it is more robust to casualties or harder communication constraints, because the scheduler is distributed. However, it could be harder to implement because of the need to synchronize the agents and we can expect some loss of efficiency in the schedules. In the distributed approach, there are no agents that have a global view of the situation to generate an optimal schedule.

It is important to make a distinction between a distributed scheduler and a distributed execution. Normally, in scheduling theory, when a schedule is made for more

than one machine, it is still done by one centralized process that schedules the tasks for all machines. Here, it is really the scheduler that is distributed, which means that there is no centralized process scheduling the tasks. The goal is to use multiagent concepts to improve standard scheduling algorithms when they are used in multiagent settings.

5.1.3 Third Step: Scheduling Algorithm Definition

In this third step, we have to choose the task scheduling algorithm to solve the problem defined in the first step. Many optimal and approximation algorithms already exist in the literature to solve different types of scheduling problems (Pinedo (1995); Jones and Rabelo (1998); Blazewick (2001)). For some simpler problems, there are even optimal algorithms that can be executed in polynomial time (see Section 5.2.3). There are also some good approximation algorithms for NP-Hard problems. This shows the advantage of formalizing a multiagent problem in a scheduling formalism because, by doing so, we can search in the scheduling theory literature to find good algorithms to solve such a problem. This helps to take advantage of both multiagent and scheduling domains.

When looking at available scheduling algorithms for the problem defined in step 1, we can really see the complexity of our scheduling problem. We can then choose to relax some constraints, if possible, to reduce its complexity. In section 5.2.3, we present the algorithms that we have chosen for different scheduling problems.

5.2 Application to the RoboCupRescue Environment

We have previously explained how to extract a scheduling system from a multiagent system. In this section, we show how to use it in a multiagent system by explaining all the steps and specifically focus on the scheduler part.

We have tested our approach in the RoboCupRescue simulation environment where we focused only on the work of the *AmbulanceTeam* agents. In the RoboCupRescue simulation there can be between 0 to 8 *AmbulanceTeam* agents that are in charge of rescuing civilians. These civilians are wounded when they are buried in collapsed buildings and they can die if they are not saved fast enough. The health of injured civilians can worsen with time. Therefore, the *AmbulanceTeam* agents have to dig in the detritus of the collapsed buildings to save the civilians that are trapped. Afterwards, they have to transport them to refuges where they can be treated. The *AmbulanceTeam* agents are helped in their work by the *AmbulanceCenter* agent with which they are in contact by radio (see Figure 2.5 on page 20).

In this complex environment, cooperation between the agents is really important, because the time needed to accomplish a task depends on how many agents are working on it. Agents work faster if they work together. For example, if there are many *AmbulanceTeam* agents working together to dig in a collapsed building, they will reach the buried civilians faster than if only one agent is trying to do the same work.

More specifically, we are interested in systems where the set of tasks is not initially known because the environment is partially observable. Thus, the agents have to explore the environment to find the tasks to accomplish. In the RoboCupRescue, this means that the *AmbulanceTeam* agents have to explore the collapsed buildings to find the injured civilians. These civilians are seen as tasks and since the health state of a civilian is uncertain, the parameters of the tasks could change in time. For example, if one civilian catches on fire, its expected death time would drop rapidly. In other words, the deadline of the task would be reduced.

Notice that, there are also important constraints on the communications in the RoboCupRescue: agents are limited in the number of messages they can send or receive and the messages' length is also limited. With these limitations, it becomes primordial to manage the communications efficiently.

5.2.1 First Step: Scheduling Problem Definition

In the RoboCupRescue simulation environment, the goal of the *AmbulanceTeam* agents is to rescue as many civilians as possible. To achieve that, agents have to sort out the civilians to rescue. This problem can be modelled as a task scheduling problem in which rescuing a civilian is considered as a task and all *AmbulanceTeam* agents are considered as resources that can accomplish a task. With this approach, we can reformulate the problem by saying that the agents' goal is simply to perform as many tasks as possible. Therefore, the problem is now to find the sequence of tasks that will accomplish their goal. In the RoboCupRescue environment, the task's duration (p_j) is the time needed to save a civilian and the deadline of a task (d_j) is the civilian's estimated death time.

This scheduling problem is very complex, because the scheduler has to allocate tasks that have different costs and deadlines to many agents. The cost of a task depends on how many agents work on it at the same time. There is also a cost to change from one task to another, i.e. when an *AmbulanceTeam* agent has to move from one civilian to another. Finally, an overload of tasks often happens, i.e. it is impossible to complete all tasks before their deadline. In other words, it is impossible to save all the civilians.

Table 5.1 presents some links that we can make between our multiagent problem issued from the RoboCupRescue and the scheduling problem. With the notation pre-

Multiagent problem	**Scheduling problem**
Set of civilians to rescue	Set of tasks (T)
Set of *AmbulanceTeam* agents	Set of resources (P_m)
Time needed to dig up a civilian	Execution cost of the task (p_j)
Death time of a civilian	Deadline of the task (d_j)
Moving time between civilians	Cost to change from one task to another (s_{jk})

Table 5.1: Links between the multiagent problem and the scheduling problem.

sented in section 5.1.1, the scheduling problem can be formally expressed as:

$$P_m|s_{jk}|\sum U_j \tag{5.1}$$

This means that m agents (P_m) evolve in an environment where there is a cost to switch from one task to the other (s_{jk}). The goal is to maximize the number of tasks accomplished before their deadline ($\sum U_j$). The task execution cost (p_j) and the task deadline (d_j) are not explicitly represented because they are implicitly considered in the parameter $\sum U_j$. This problem as it is defined has been proven to be NP-Hard (Pinedo (1995)). So that it can be solvable in polynomial time, we relaxed some constraints and made some changes to the original problem. We now detail these aspects.

In the original problem, we consider that there is a cost to switch from one task to another. This is problematical because the cost could be different for each pair of tasks ($task_x$, $task_y$) and thus, it increases the complexity of the problem. We therefore relaxed this constraint by giving a value of 0 for each s_{jk}. However, even if we have relaxed this constraint, we still need a way to take the travelling time from one civilian to another into consideration. Hence, we added a unique estimation of the switching time to the task's execution cost. We say that it is a unique estimation because it does not depend on the preceding task or the agent accomplishing the task. In practice, this means that the time to rescue a civilian (p_j) is equal to the unique estimated time to move to the civilian's location plus the time to dig him up. This latest time is known, but the estimated time to move to the civilian's location is in fact the distance between the position of the civilian and the building farthest away multiplied by a constant K. This is the worst case estimation pondered by a constant determined experimentally by looking at some simulations.

In this thesis, we have considered two types of problems. The first one considers a distributed execution of the schedule and the second one considers a centralized execution of the schedule. With distributed execution and if we take into consideration that $s_{jk} = 0$, the problem can be defined as:

$$P_m||\sum U_j \tag{5.2}$$

With the centralized execution, the scheduling problem definition is again slightly modified by setting $m = 1$. In other words, it means that we consider that we have only one agent. In practice this is not true, thus it results in all agents working on the same civilian because, for the scheduler, the group of agents is only one indivisible resource.

Although this modification of considering only one resource can reduce the efficiency of the schedules, it has a substantial advantage because there is an optimal algorithm working in polynomial time for this modified problem. Notice that such modification is logical in the RoboCupRescue simulation because the execution time of a task depends on the number of agents working on it. In fact, if there are n agents working on one task, then the execution time will be n times less. Formally, the scheduling problem can now be defined as:

$$1||\sum U_j \tag{5.3}$$

This means that we consider a centralized execution in which all agents are considered to be one big resource working on one task at a time and trying to maximize the number of tasks accomplished in the time allowed.

5.2.2 Second Step: Scheduler Type Definition

In the second step, the system designer must choose between centralized and distributed scheduling. In this thesis, we compare both approaches in order to schedule the tasks of the *AmbulanceTeam* agents in the RoboCupRescue simulation. In the centralized approach there is one agent responsible for the scheduling of all tasks. In the distributed approach, all agents are in charge of the scheduling process.

5.2.2.1 Centralized Scheduler

With a centralized scheduler, there is one agent taking alone the decision about the ordering of tasks. This agent has to gather information about the environment, make a schedule for each task and each agent and send the schedule to all agents so that they can execute it.

In the RoboCupRescue simulation, the "best agent" to serve as the central scheduler is the *AmbulanceCenter* agent. This agent has better communication capabilities, so it can receive and send more messages. This means that it should have a better global view of the situation, enabling it to make good schedules. Briefly, this means that each agent sends the information it has about the civilians to the *AmbulanceCenter* agent. Afterwards, this agent schedules all the tasks, deciding in which order they should be accomplished and by how many resources (agents). Finally, it sends the assignments

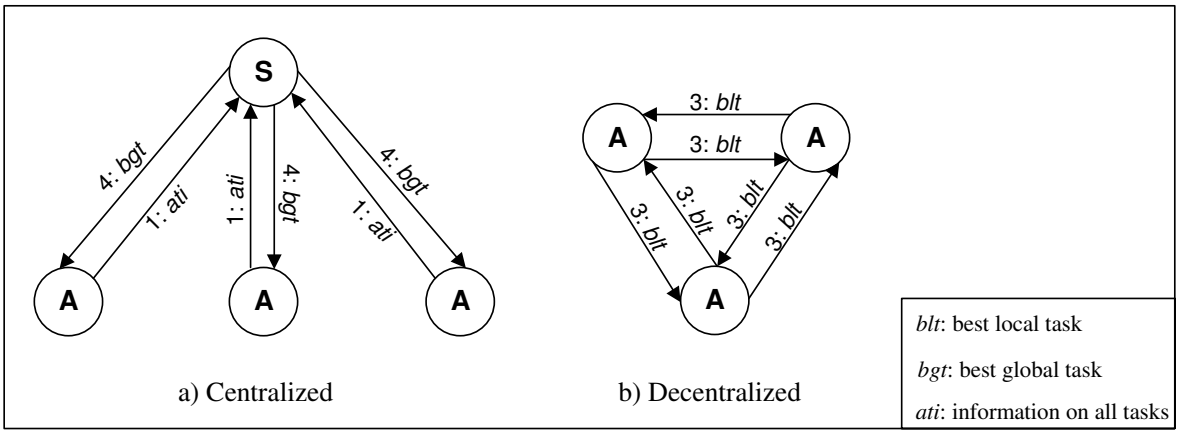


Figure 5.1: Information exchange when using the centralized (a) and the decentralized (b) scheduling approaches. On the arrows, the type of the message is identified with the number referring to the step in the scheduling process.

to all *AmbulanceTeam* agents that only have to conform to them. In brief, the steps of the scheduling process are:

1. All the *AmbulanceTeam* agents send all their perceptions about possible tasks to the scheduler agent, which is the *AmbulanceCenter*.
2. The scheduler agent combines all the information received to construct its list of possible tasks.
3. The scheduler agent applies a scheduling algorithm to schedule the tasks.
4. The scheduler agent sends the first task of the schedule to all agents.

Figure 5.1 a) presents a graphic representation of the information exchanged between the agents when using the centralized approach. As we can see, at step 1 all the agents send the information they know about all possible tasks. These messages can be quite long. Afterwards, at step 4, the scheduler agent sends the first task of the schedule to all agents, which then accomplish this most important task.

5.2.2.2 Decentralized Scheduler

In this approach, we can see the scheduler as an entity composed of many agents. Each agent has its own local perception about the environment and the tasks to accomplish. The scheduling is done in two steps. Firstly, each agent chooses locally its best task to accomplish using a scheduling algorithm and given its local knowledge.

Secondly, all agents exchange their best local task. Then each agent uses a scheduling algorithm to find the task to accomplish among the set of best tasks. In brief, the steps of the scheduling process are:

1. All agents build their own list of possible tasks.
2. All agents apply a scheduling algorithm to find the best local task to accomplish.
3. All agents broadcast their best local task to all other ambulance team agents.
4. All agents build a list with all the best local tasks received.
5. All agents apply a scheduling algorithm to find the best global task to accomplish.

Figure 5.1 b) presents a graphic representation of the information exchanged between agents when using the decentralized approach. As we can see, agents send messages only at step 3 and those messages are quite small because they contain only the information about one task. Therefore, the decentralized approach is less demanding on the communication because agents do not send big messages, like the ones send at step 1 of the centralized approach.

5.2.3 Third Step: Scheduling Algorithm Definition

For this thesis, we have mainly used two scheduling algorithms: the Earliest Due Date algorithm (EDD) (Jackson (1955)) and the Hodgson's scheduling algorithm (Moore (1968)). In the next two sections, these two algorithms are presented. Then, we present the scheduling strategies we have developed using these two basic algorithms.

5.2.3.1 Earliest Due Date Algorithm

The Earliest Due Date algorithm (EDD) is quite simple. To schedule a set of tasks, the EDD algorithm sorts all the tasks in the ascending order of their deadlines. Consequently, the first task to execute is the task with the earliest deadline. This algorithm can be executed in time $O(nlog_n)$ (Brucker (2001)), where n is the number of tasks. This algorithm is particularly interesting because it is optimal if there is no overload, i.e. if it is possible to accomplish all the tasks in the given time. Although some overload could happen in our environment, the performances of this algorithm stay good, as presented in our results, and its simplicity enabled us to demonstrate how we can distribute the decision making in a scheduling system.

Since EDD is a greedy algorithm, it is possible to just find the first task to accomplish without having to schedule all the tasks. This first task is simply the task with the

earliest feasible deadline. This property of the EDD algorithm is interesting in dynamic environments where the agents have to react to changes in the environment. With this algorithm, the scheduling can be done really fast, because the scheduler agent only has to do one iteration over the tasks to find the next task to accomplish. This type of greedy algorithm is well adapted to a problem of decentralized decision making because it is never necessary to reconsider a decision previously made. This enables agents to find the next task to accomplish in time $O(n)$, where n is the number of tasks. This property also helped agents to save a lot of messages in the decentralized approach.

5.2.3.2 Hodgson's Scheduling Algorithm

The Hodgson's scheduling algorithm is an optimal algorithm to solve our simplified problem $(1||\sum U_j)$. This algorithm can find the optimal schedule in time $O(nlog_n)$, where n is the number of tasks. This algorithm is based on the (EDD) algorithm presented in the previous section.

The Hodgson's scheduling algorithm is presented in Algorithm 5.1. In short, it begins by sorting all the tasks in increasing order of their deadlines and then it gives priority to tasks with lower execution costs. To do so, it goes through the ordered task list and when it encounters a task that could not be executed before its deadline, it removes the task from the previously scheduled task list that has the biggest execution time.

In the RoboCupRescue simulation, this algorithm receives a set of civilians to rescue and it returns the schedule that should rescue as many civilians as possible. Only civilians that could be rescued are returned and those that cannot be rescued before their deadline are discarded.

5.2.3.3 Scheduling Strategies

In this section, we present the different scheduling strategies that we have tested in the context of RoboCupRescue. We have looked at strategies to distribute the scheduler and to distribute the execution of the schedules.

Central Scheduler and Central Execution. This first strategy is used in the simplified version of the problem $(1||\sum U_j)$. In the RoboCupRescue simulation, it means that we consider all the *AmbulanceTeam* agents to be only one indivisible resource and that the objective of the scheduling system is to maximize the number of civilians rescued before their deadlines.

1: **Function** HODGSONSCHEDULING(T)
returns T', an optimal list of scheduled tasks respecting their deadline.
Input: T, a set of tasks to schedule.
Local Variable: T_k, the task in T' with the greatest execution cost.
2: $T \leftarrow EDD(T)$
3: $n \leftarrow |T|$
4: $T' \leftarrow \oslash$
5: **for** $i \leftarrow 0$ to n **do**
6: $T' \leftarrow T' \cup \{T_i\}$
7: **if** $\sum_{j \in T'} p_j > d_i$ **then**
8: $T_k \in T'$ such that $p_k = \underset{j \in T'}{max}(p_j)$
9: $T' \leftarrow T' - \{T_k\}$
10: **end if**
11: **end for**
12: return T'

Algorithm 5.1: Hodgson's scheduling algorithm.

This modification of the original problem can reduce the efficiency of the task scheduling, because the scheduler do not consider splitting the *AmbulanceTeam* agents. However, it has a big advantage because the Hodgson's scheduling algorithm (presented in section 5.2.3.2) can find an optimal schedule in polynomial time for this modified problem. Moreover, as mentioned before, this modification is logical for our problem because the time to dig out a civilian is inversely proportional to the number of *AmbulanceTeam* agents digging in the collapse building.

Central Scheduler and Distributed Execution. For this second strategy, we have conserved the original multi-machine scheduling problem (P_m). Therefore, the scheduler can now divide the agents to work on more than one civilian at a time. However, we still have some simplifying constraints. The *AmbulanceTeam* agents are divided in m groups and the number of groups m stays constant. Furthermore, two groups cannot work on the same civilian.

These simplifications enabled us to have a fixed task execution cost. Hence, for a total of n agents, the task execution cost will be of n/m times less than the cost of doing the task alone, because there are n/m agents in each group. Moreover, we now have the opportunity to work on m civilians at a time, which was impossible with the first strategy. Thus, we have the opportunity to save more than one urgent civilian. However, this modification can also reduce the efficiency, because each task will be done

more slowly. In fact, the execution cost of each task is multiplied by m compared to the first strategy.

There is no existing optimal algorithm to solve this NP-Hard problem in polynomial time (Pinedo (1995)). Therefore, after some tests, we chose the approximation algorithm EDD which seemed to be the most efficient. In short, the tasks are sorted in increasing order of their deadline time, then the first group of agents is assigned to the task with the smallest feasible deadline, the second group to the second civilian, and so on if there are more groups.

Distributed Scheduler and Central Execution. This third strategy also works with the simplified version of the problem ($1||\sum U_j$). The difference here is that the scheduling process has been distributed among the *AmbulanceTeam* agents, as presented in section 5.2.2.2. The scheduling algorithm used in this decentralized scheduling approach is also the EDD algorithm. We have chosen this algorithm because it is possible to have a distributed version that does not loose in efficiency. If we look at the centralized and decentralized schedulers presented in sections 5.2.2.1 and 5.2.2.2 respectively, it is easy to see that, with the EDD scheduling algorithm, the decentralized approach returns the same task as the centralized approach, because at the end, agents choose the task with the earliest deadline in both approaches. The difference is in the number of messages exchanged by the two approaches. In section 5.4.2, we present results comparing the centralized and the distributed schedulers.

5.2.3.4 Rescheduling Strategy

As stated in many previous places, the RoboCupRescue environment is uncertain and dynamic and consequently agents have to be able the reschedule when changes happen (Smith (1994)). One method to do that would be to reschedule each time that something changes in the environment. However, this method could make agents to change from one task to another before completing any, if each time something changes, the scheduler modifies the first task in the schedule.

To circumvent this, we have used a strategy that follows the following principle: if a task is being executed, it will be executed until the end. If during the execution of this task, a new information is received, it is stored and it is only when the task is completed that the agents take the new information into consideration to reschedule and choose the next task to accomplish.

All the algorithms presented in this section consider that the deadlines of the tasks are known. In other words, they consider that they know the exact death time of each civilian. In practice, this is not really the case. The agents do not know the exact death

time of a civilian, they can only approximate it using their partial perceptions. In the next section, we present the learning approach that we have developed to estimate the death time of a civilian.

5.3 Learning Mechanism for the Estimation of the Civilian's Death Time

In order for the schedules to be valid and efficient, the predictions about the civilian's death times have to be as precise as possible. To obtain the best possible predictions, we have used an instance based K-Nearest-Neighbors (KNN) learning algorithm. However, we have modified the basic KNN learning algorithm to adapt it to partially observable environments in which many attribute values can be missing.

In the RoboCupRescue simulation, *AmbulanceTeam* agents have to predict the civilian's death times in order to identify the civilians with the highest priorities. To achieve this, the rescue agents have to determine how the *damage* attribute of a civilian evolves during the simulation. The *damage* is the number of health points (HP) that an injured civilian looses per time step ($HP_t = HP_{t-1} - damage$). If the number of health points reaches zero, then the civilian is considered as dead. A civilian HP is initially set to 10000. An injured civilian's *damage* is always ascending, i.e. that it increases at each time step, until the civilian dies or the civilian reaches a refuge, in which case it is set to 0. Figure 5.2 shows the progression of the damage attribute for many civilians. As we can see in the figure, there are some tendencies in the damage progression of the civilians. In the following we present how we have used a KNN learning algorithm to approximate these curves for a new civilian.

As stated previously, the position and the health status of the civilians are unknown at the beginning of the simulation. All rescuing agents have to search in the collapsed buildings to find the buried civilians. When a civilian is found, the rescuing agents can see the current state of health of the civilian. A priori, the rescuing agent does not know how the health of the civilian will evolve. The rescuing agent can however come back later to take a second look at the civilian state of health. The fact that the state of health of the civilians is not known at every time step is the most important source of uncertainty that the *AmbulanceTeam* agents have to deal with when scheduling the civilians to rescue. In the next section, a KNN learning algorithm that can deal with such uncertainty is presented.

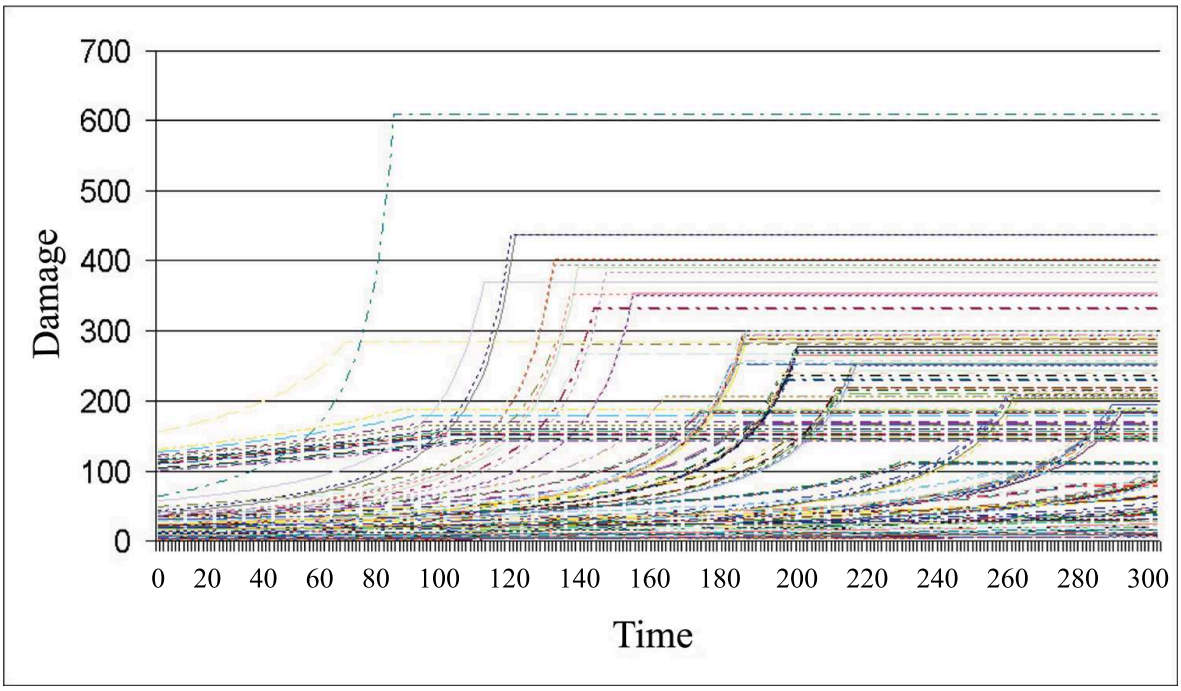


Figure 5.2: Damage progression for 250 civilians. Each line represents the progression of the damage value for one civilian.

5.3.1 K-Nearest-Neighbors: Introduction

As mentioned before, we have developed an instance based K-Nearest-Neighbors (KNN) learning algorithm to learn to predict the death time of a civilian agent. In this section, we briefly present the K-Nearest-Neighbors approach, in which an instance corresponds to a point in a n dimension space. An instance x is described by a vector of n attributes:

$$(a_1(x),\ a_1(x),\ \ldots\ a_n(x)) \tag{5.4}$$

where $a_i(x)$ is the value of the i^{th} attribute of the instance x; n is the number of attributes per instance.

Each instance is associated with a class. For example, in Figure 5.3, there are five instances with their associated class. The objective of the KNN algorithm is to find the class of a new unclassified instance by using the previously classified instances.

When all the attribute's values are present, it is possible to use the standard KNN learning algorithm. It normally consists in finding the k closest neighbors of the instance to classify by using a distance function. The most popular distance function is the euclidian distance. The distance between two instances x_i and x_j is defined by $d(x_i, x_j)$

Instance x_i	**Class of** x_i
$< 5, 8, 3, 7, 6 >$	A
$< 2, 4, 1, 3, 5 >$	A
$< 8, 3, 2, 8, 9 >$	B
$< 6, 4, 8, 2, 1 >$	A
$< 4, 5, 7, 1, 9 >$	B

Figure 5.3: Instance database.

where:

$$d(x_i, x_j) = \sqrt{\sum_{r=i}^{n} \left(a_r(x_i) - a_r(x_j)\right)^2} \tag{5.5}$$

When the k neighbors of an instance x_q have been identified, then the instance x_q can be classified by considering the values of its neighbors. In a discrete case, the equation used to classify the instance x_q is:

$$\hat{f}(x_q) \leftarrow \underset{v \epsilon V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i)) \tag{5.6}$$

where $\hat{f}(x_q)$ is the estimated class of the instance x_q, V is the set of all possible classes for the classification function and:

$$\delta(a, b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if not} \end{cases} \tag{5.7}$$

In our approach, an instance corresponds to a progression curve of the damage value for a civilian. An instance has 300 attributes, one for each time step in the RoboCupRescue simulation. The attribute value is the damage value observed at the time in question. For example, if the civilian's damage at time 84 is 120, then the 84^{th} attribute has the value 120. In the next section, we present how the instances are used to estimated the death time of a civilian, but first we present a method to deal with missing attributes in the general case.

It is necessary to consider missing attributes because the health state of a civilian is not known at each time step of the simulation, since the RoboCupRescue environment is partially observable. In our case, the instances in the instance database are completely defined, i.e. that they have a value for each attribute. The missing attribute values are only for the instances to classify, since they are dynamically built during the simulation. In other words, during the learning phase, all the attribute values are known, but when a new instance has to be classified, then a number between 0 and $n - 1$ values can be missing.

Let x_q be the instance to classify based on the instance database presented in Figure 5.3. As we can see, the value of the third attribute is missing.

$x_q =< 5, 4, ?, 2, 8 >$

If we evaluate the distance between the query instance x_q and each of the instances present in the instance database, then by omitting the third attribute, we obtain:

$d(x_q, x_1) = \sqrt{45}$
$d(x_q, x_2) = \sqrt{19}$
$d(x_q, x_3) = \sqrt{47}$
$d(x_q, x_4) = \sqrt{50}$
$d(x_q, x_5) = \sqrt{4}$

If we set $k = 3$, then the three closest neighbors are x_1, x_2 and x_5. The class of x_1 and x_2 is A and the class of x_5 is B. Using Equation 5.6, we find that the estimated class of x_q is A.

Figure 5.4: Example of an instance classification with one missing attribute.

When there are some missing attribute values, the only modification to the KNN algorithm is when choosing the k nearest neighbors. In our approach, we simply ignore the missing attribute values when calculating the distance function:

$$d(x_i, x_j) = \sqrt{\sum_{c \in C} (a_c(x_i) - a_c(x_j))^2} \tag{5.8}$$

where C is the set of available attribute values in the instance to classify.

Consequently, even if the attribute values are available in the instance database, if they are not available for the query instance, then they are not considered when evaluating the distance between the instances. Figure 5.4 shows an example of the approach. To better understand the approach, lets represent the instances in a two dimension space. The x axis represents the attributes and the y axis represents the values for these attributes. We can now see that each instance in the database is represented by a curve. As an example, if we use the instances presented in Figure 5.3, then we obtain the curves presented in Figure 5.5.

It is also possible to represent the query instance x_q in the graphical representation of the instance database. As shown in Figure 5.6, the query instance is represented as a set of points. If there are some missing attributes, then there are no points for the

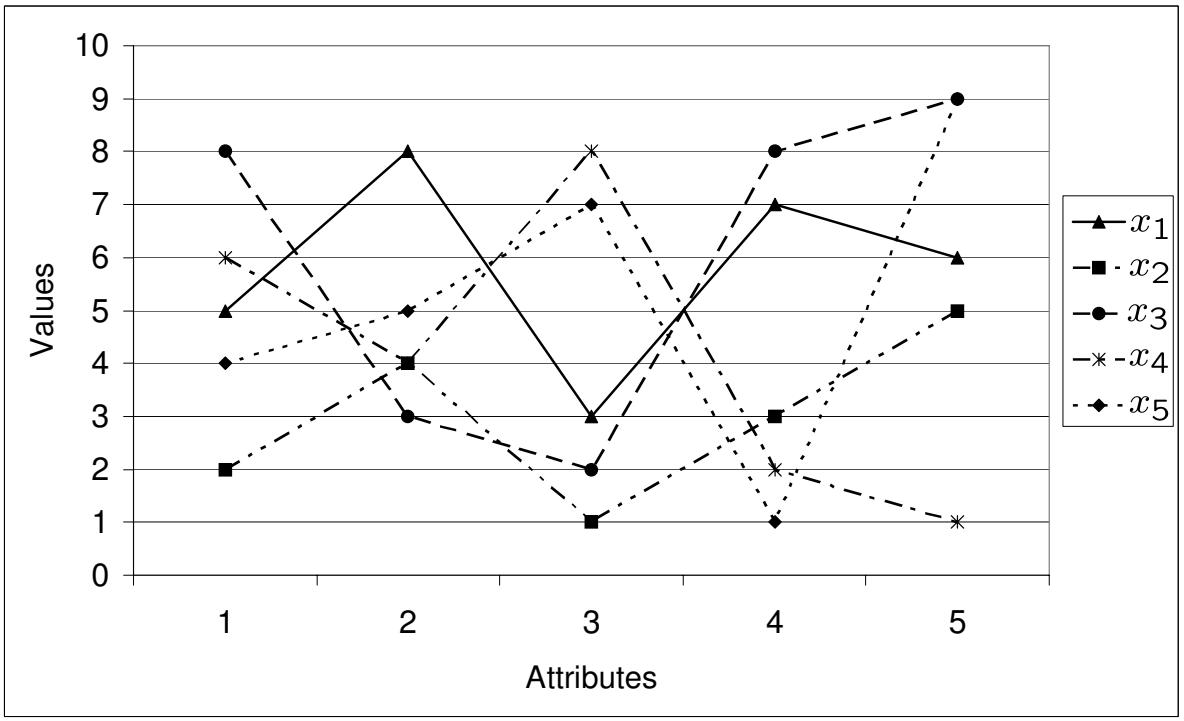


Figure 5.5: Graphical representation of the instance database.

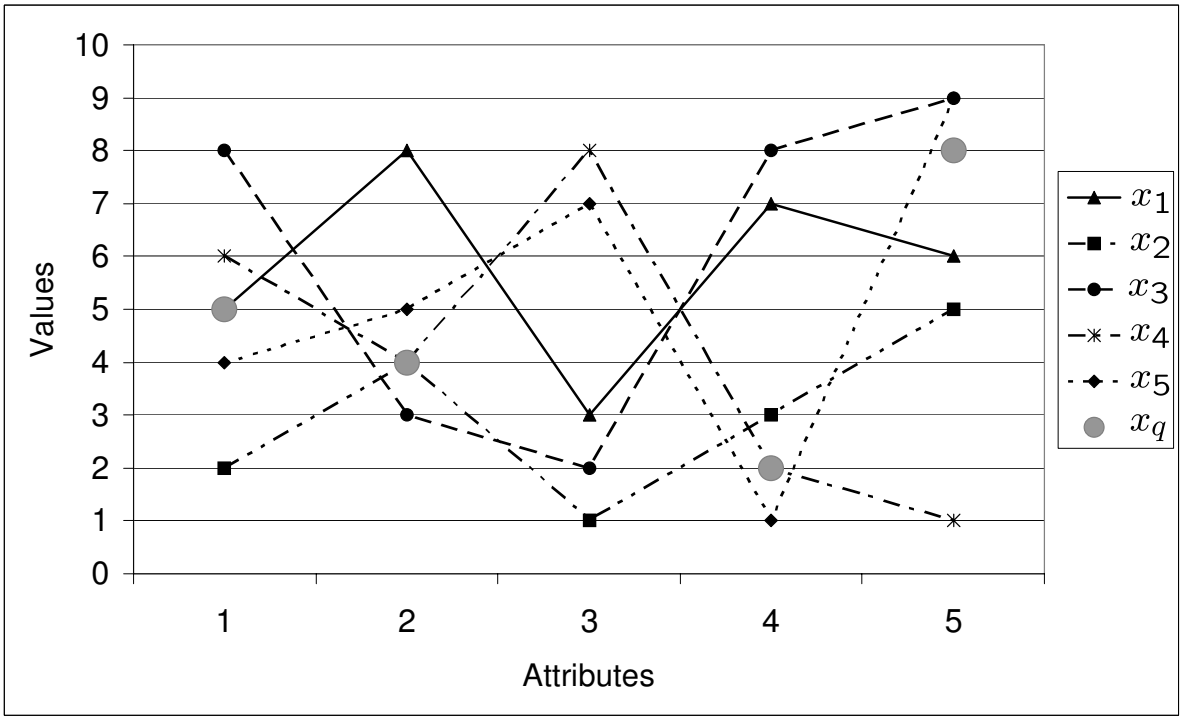


Figure 5.6: Instance database with query points.

corresponding x coordinates. The approach then consists in finding the k closest curves from the set of points representing x_q, by using the Equation 5.8. When the k closest curves have been identified, then the algorithm uses the classification equation 5.6 to classify x_q, based on the k nearest neighbors.

This technique that simply ignores missing attributes is in contrast with the approaches that try to approximate the missing attribute values like the EM and the k-means approaches (Caruana (2001); Acuna and Rodriguez (2004)). In our context, as shown in section 5.4.1, it was more efficient to ignore the missing attributes. In the next section, we present how we have applied our algorithm in the RoboCupRescue.

5.3.2 KNN for the RoboCupRescue

The proposed solution is particularly suited for problems in which there is a correlation between the attributes in an instance. This is exactly the case when we want to predict the death time of a civilian based on the damage observed at some time steps.

In the RoboCupRescue simulation, it is really important to have a good estimation of the death time of the civilians in order to have good schedules. To do this, the rescue agent has to determine how the civilian's damage will evolve in time. This task is not easy because the civilian's damage is an uncertain parameter in the simulation. It is impossible to perfectly predict the damage progression in time. The only think possible is to try to approximate it.

On the other hand, it is interesting to notice the similarities in the evolution of the damages as presented in Figure 5.2 on page 152. This figure presents the evolution of the damage for 250 civilians. We can see some tendencies based on the starting damage value. However, it is impossible to predict the direction that the damage will take since the rescue agent does not know the civilian's starting damage value. The civilians' damage values are unknown at the beginning of the simulation, and consequently the rescue agents have to find the civilians in order to see their health state. Another difficulty is that the curves are crossing each other. This complicates the problem because with few observations it is harder to identify the curve that could best predict the future observations.

In order to predict the death time of a civilian, we use the K-Nearest-Neighbors learning algorithm presented previously. In our case, an instance corresponds to a progression curve of the damage value for a civilian. An instance has 300 attributes, one for each time step in the RoboCupRescue simulation. The attribute value is the damage value observed at the time in question. For example, if the civilian's damage at time 84 is 120, then the 84^{th} attribute has the value 120.

During the learning phase, the damage values are recorded for all time steps, for all the civilians. This constitutes the instance database. Moreover, in our model, there are as many classes as there are instances in the instance database. As it can be seen on Figure 5.2, there are many instances that can be really close from one another. Therefore, it could be interesting to reduce the number of instances in the database in order to accelerate the algorithm (Grudzinski and Duch (2000); Morring and Martinez (2004)). Since there is one class for each instance, it is not necessary to have two similar curves giving similar predictions. Therefore, we only keep one instance for each possible prediction. In other words, each instance in the database has to respect a minimal distance δ between the other instances. During the learning phase, when we add an instance in the database, we make sure that the new instance is at a distance of at least δ from the other instances already present in the database. If this condition is not met, then the new instance is discarded. This is also done to improve the online efficiency of the algorithm. During the simulation, the algorithm has to be executed under a strict real-time constraint, thus if the database is smaller, then it is faster to calculate a prediction.

The value of the parameter δ depends on the problem and the targeted performances. If the δ parameter is small, then there will be less instances discarded. This may improve the precision, but it will also take more time to calculate a prediction. In our problem, we have fixed δ to 50. As our tests have shown, this was a good compromise between precision and speed for our specific problem. With this value, the number of instances in the database has been reduce by 68% (from 2700 to 860 instances). This huge reduction is mainly due to the fact that there was a lot of similar instances.

A classification of an instance is required when an agent wants to estimate the death time of an injured civilian. More precisely, the rescue agent has to create an instance based on its past observations about the civilian's damage. In these condition, the rescue agent has to determine the class to which its new instance belongs. At this step, our approach for the instance's missing attributes is quite useful. The reason is that there are many more missing attribute values than available attribute values, since it is not possible to observe all the civilian's damage values at each time step of the simulation. Each time a new observation is made about a civilian's damage, it fills one hole in the instance representing the progression of the civilian's damage. After the agent has created the instance with all the available values, it then finds the nearest instance x_n in the database:

$$x_n = \underset{x_i \in X}{\operatorname{argmin}}\ d(x_q, x_i) \tag{5.9}$$

where X is the set of instances in the database and x_n is the nearest instance of the instance x_q.

This corresponds to find the curve in the instance database that best represents the observed points for the query instance. When the closest instance has been found, the rescue agent uses it to predict the civilian's damage evolution, which then can be used to predict the death time of the civilian.

5.4 Experimentations

This section presents the experimentations that have been done to test our scheduling approach. In the first set of experiments, we present results showing the efficiency of our K-Nearest-Neighbors approach to estimate the death time of a civilian. Then, we present results comparing the efficiencies of a centralized and a decentralized scheduler. Afterwards, we present results comparing the efficiencies of a centralized and a decentralized execution of the schedules. Finally, we compare the results of our scheduling approach with the results of another RoboCupRescue team.

5.4.1 K-Nearest-Neighbors Experiments

Firstly, we have compared the prediction accuracy of the K-Nearest-Neighbors (KNN) approach with an approach that only considers the current damage value of the civilian (HP/DMG). With the latest approach, the estimated death time is simply calculated by dividing the health points (HP) value of the civilian by its damage value (DMG). With the KNN approach, we use the damage estimated progression to simulate the diminution of the HP value at each time step. In this case, the estimated death time is the time step reached when the HP value reaches 0.

The results comparing the prediction efficiency of the KNN and the HP/DMG approach are presented in Figure 5.7. These results have been obtained on 26 simulations. Each time a rescue agent had to estimate a civilian death time, we have recorded the estimation from the two approaches. Afterwards, we have compared these estimations with the real death time of the civilian. The results are average differences between the estimated and the real value. Each line on the graphic represents the average of all the predictions made within a 10 time steps interval. For example, the first line represents the average for all the predictions that have been made between the time steps 0 and 10. As we can see in Figure 5.7, the KNN approach makes much better predictions then the HP/DMG approach. For example, at time 110, the average error in the predictions for the KNN approach is around 1. This means that on average, the agents are predicting the death time of a civilian at ± 1 time step. For the same predictions, at the same time, the HP/DMG approach is only estimating the death time of a civilian at ± 26 time steps. After time step 150, the KNN approach is almost perfect in its predictions.

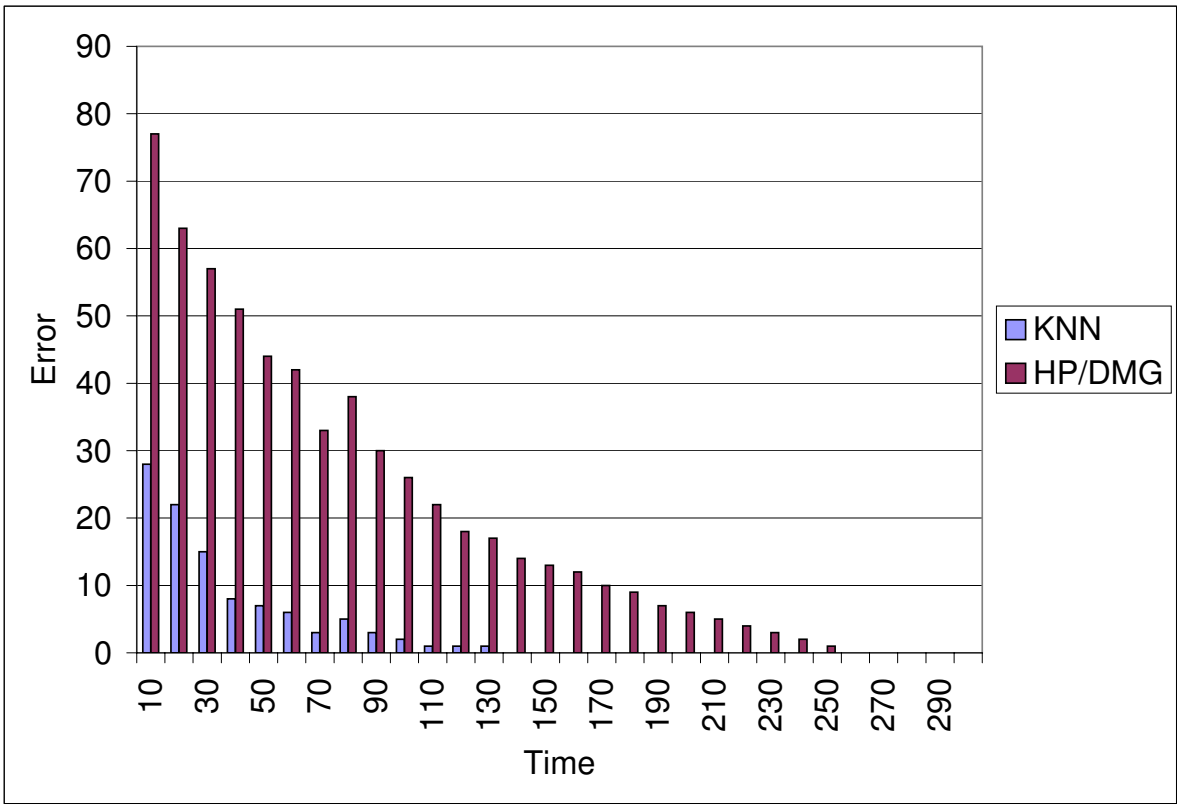


Figure 5.7: Prediction efficiency of the K-Nearest-Neighbors (KNN) approach. This graphic presents the error made by the KNN approach compared to the approach HP/DMG, which consists in dividing the current HP value of a civilian by its damage value (DMG). The error is the average difference between the prediction and the real value.

As explained in section 5.3.2, we are ignoring the missing attribute values in the query instance x_q. Another popular strategy when dealing with missing attributes is to approximate the values of all missing attribute values using a KNN algorithm (Dixon (1979)). With this kind of imputation strategy, the estimated value of each missing attribute value is the average of the same attribute for the k-nearest neighbors. In our tests, reported on Figure 5.8, we have tested two values for k: 2 and 15. We have tested with different percentage of missing attribute values. We can see that our approach of no imputation is among the best approaches with the 2-nearest neighbors imputation method. We can also see on Figure 5.9 that our approach is much faster than the other two approaches. In our case, it is thus better to do no imputation and to simply ignore all missing attribute values of the query instance.

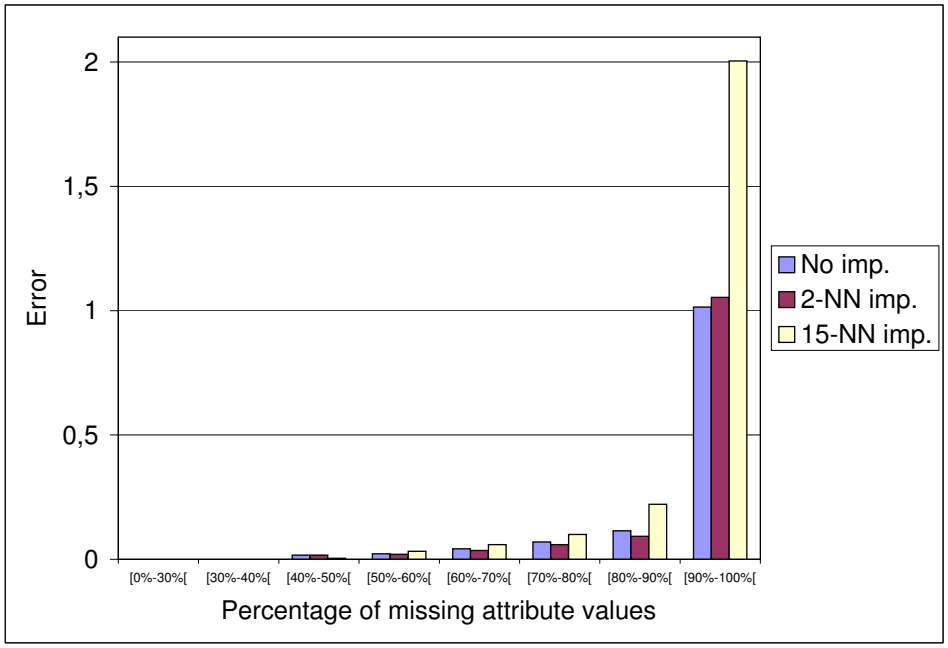


Figure 5.8: Comparison of the performances of different strategies to deal with missing attribute values. The error is the average difference between the prediction and the real value.

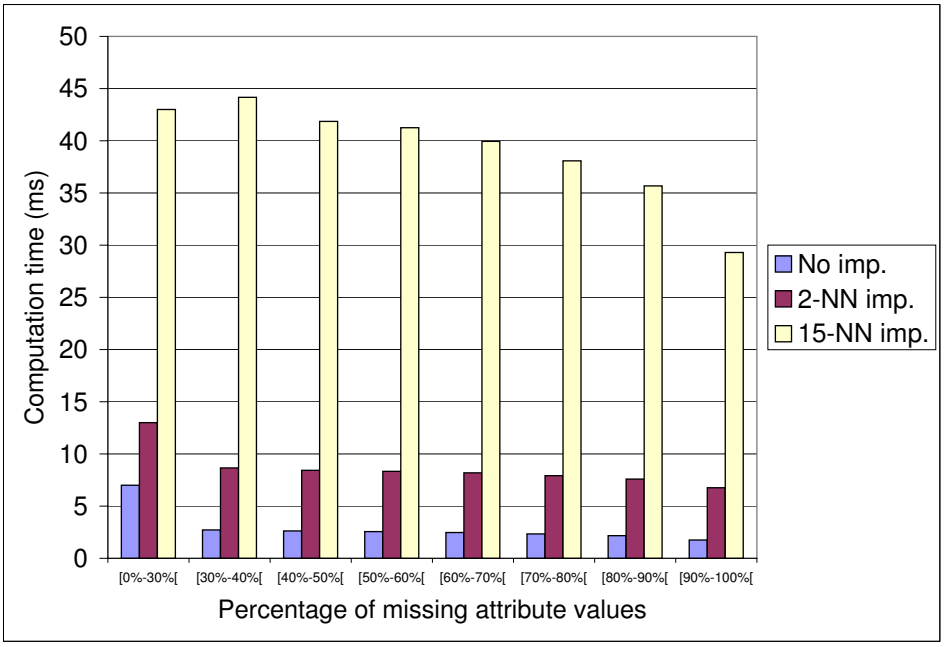


Figure 5.9: Comparison of the computation time of different strategies to deal with missing attribute values.

5.4.2 Centralized Versus Decentralized Scheduler

The objective of our experiments was to compare the performance and the robustness of a decentralized scheduling system with a centralized one. In our environment, the performance is measured based on the number of civilians saved before their death, i.e. the number of tasks accomplished before their deadline. The robustness test is used to determine the capacity of the system to maintain good performances when faced with hard communication constraints. The communication burden is evaluated by considering the amount of information (measured in bytes) transmitted by the agents.

Our first experiment shows that it is possible to implement a distributed task scheduling system that offers the same performances as a centralized approach, by generating the same tasks ordering while diminishing the communication burden. In our second experiment, we show that a decentralized scheduling system is more robust, i.e. less sensitive to hard communication constraints.

5.4.2.1 First Experiment

The goal of this first experiment is to compare the performances and the communication burden of the decentralized scheduling approach compared to the centralized approach. For our experiments, we have created six different simulation scenarios. Those scenarios were designed to bring to the fore the work of the ambulance team agents. Therefore, we have simplified the simulations to remove everything that could interact with the ambulance team agents. To be more precise, we have removed the blocked roads as well as all the fires. The scores obtained in those simulations are consequently only dependant on the number of civilians alive, that is only on the ambulance team agents work. For our six scenarios, we have used three different maps and we have used each of them twice with a different number of agents. Each of these scenarios has the same importance, i.e. all scenarios have similar complexity.

Figure 5.10 presents the comparison between the performances of each approach. The centralized approach is slightly better in five scenarios out of six. However, this difference is really subtle and if we consider the 95% confidence interval, the two approaches can be considered equal.

However, the diminution of the communication burden, as shown in Figure 5.11, is really at the advantage of the decentralized approach. In this figure which presents the comparison of the number of bytes sent by the agents, the ordinate axis represents the number of bytes sent in average during one simulation. We can see that the distributed approach enables reducing the quantity of information sent. On average, there is a 30% reduction. This is mainly because the agents do not have to send all the information

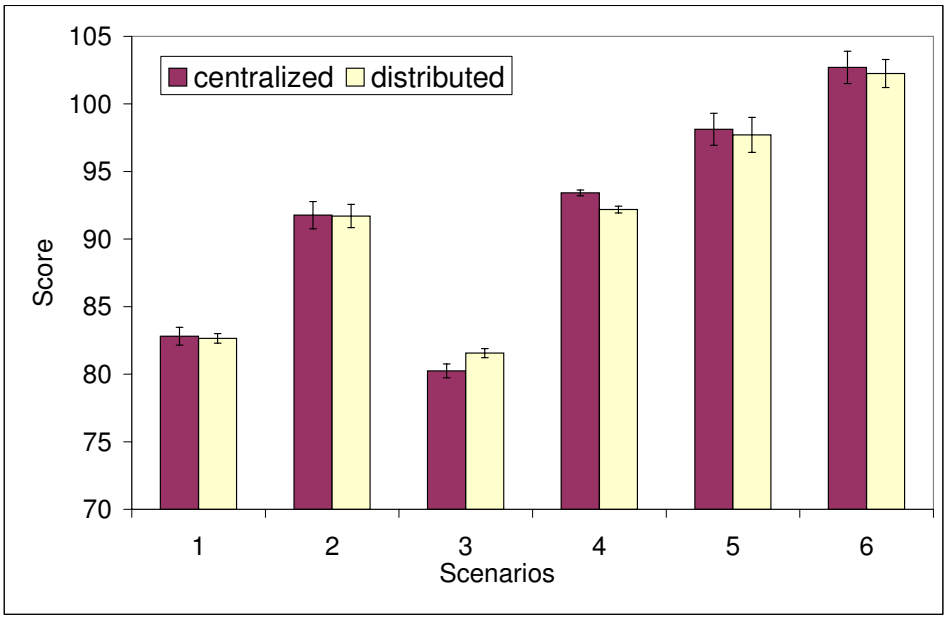


Figure 5.10: Comparison of the performance between the centralized and the distributed scheduler approaches.

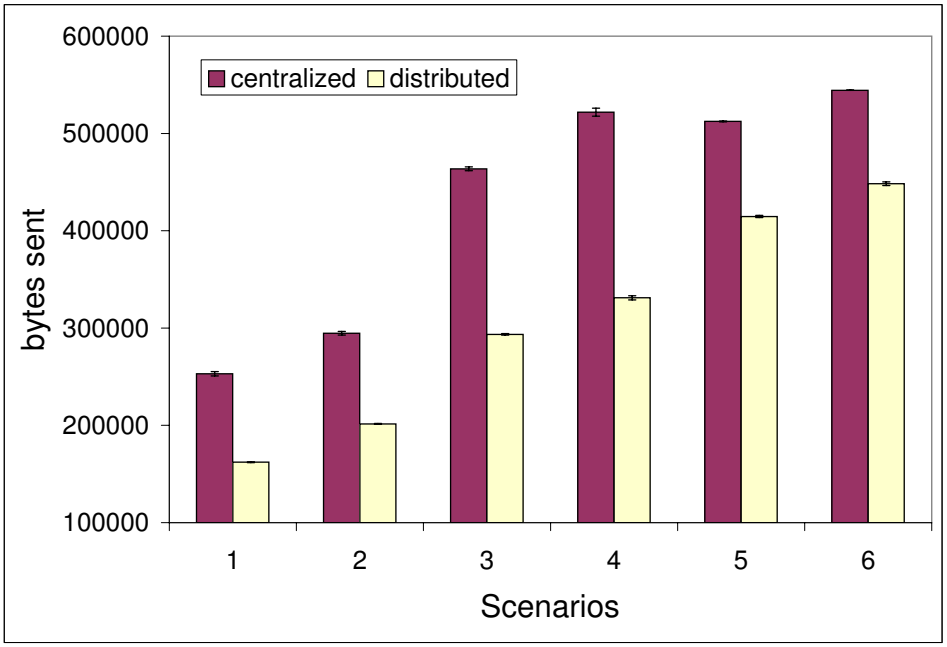


Figure 5.11: Number of bytes sent by the centralized and the decentralized approaches.

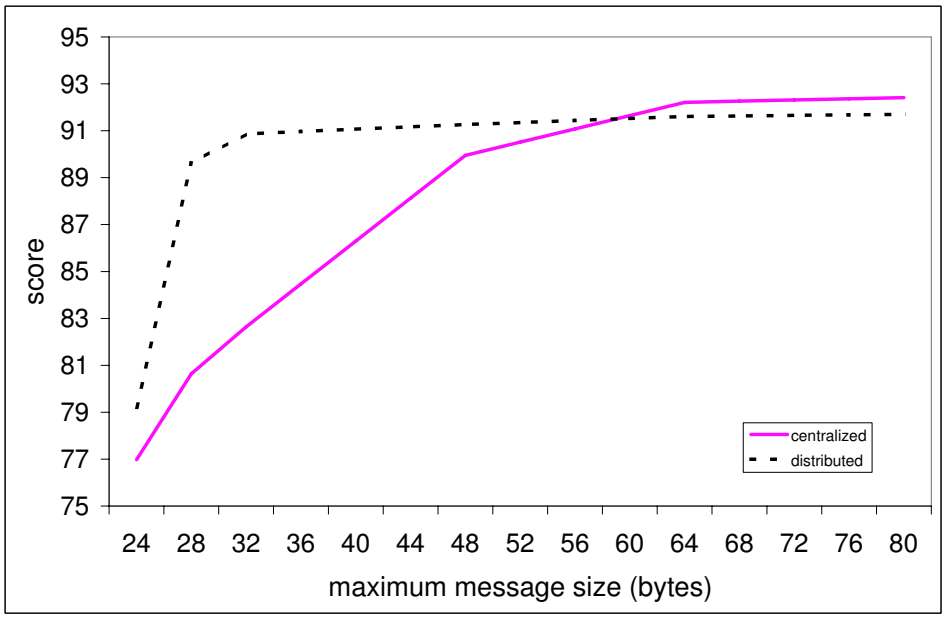


Figure 5.12: Performances when the constraint on the message's length is modified.

they have about all possible tasks, but only the information about the most interesting task. In the centralized approach, there are a lot of redundancies in the messages received by the scheduler, because in the worst case, all agents can send the same information about a given task to the central scheduler.

In short, the decentralized approach is able to obtain the same performance with 30% less information sent. In the next section, we can see the impact of this bandwidth economy on the approach's robustness.

5.4.2.2 Second Experiment

In this second experiment, we tested the robustness of the two approaches. More precisely, our goal was to know which approach can perform well even when the communications are really limited. We tested the performances of the *AmbulanceTeam* agents on different configurations in which we modified the maximum length of a message. We did six series of tests with message length limits ranging from 24 to 80 bytes.

Figure 5.12 compares the results of both approaches. Those results show that the decentralized approach is more stable. If the message length is more than 36 bytes, the performances are not affected by a diminution of the exchange capacity. The centralized approach is less stable, its performances decrease rapidly when we limit the communication. Therefore, if the agents evolve in a system in which the communication are limited, the distributed approach seems a better option, because it can obtain good performances with fewer messages.

	Strategy 1	**Strategy 2**
Step 1	$1 \mid\mid \sum U_j$	$P_m \mid\mid \sum U_j$
Step 2	centralized	centralized
Step 3	Hugdson's algorithm	EDD

Figure 5.13: Strategies compared in our tests.

5.4.3 Centralized Versus Decentralized Execution

The goal of this experiment consisted in comparing the two strategies briefly presented in Figure 5.13. We aimed to verify if dividing agents in many teams with an approximation scheduling algorithm is better or not compared to keeping the agents in one group with an optimal scheduling algorithm. For this experimentation, we have used the same six scenarios as the previous experimentations. Each scenario was used a dozen of times in three different sets of tests. The first set of tests investigates the first strategy, i.e. the one group approach. The other sets of tests investigates the second strategy in which we divide the agents in two and three groups ($m = 2$ et $m = 3$).

In theory, the first strategy is better for some sets of tasks and for others the second approach is the preferred one. In these conditions, our objective was to find which approach is the best depending on the tasks distribution in the simulation and whether there is one approach that seems to be more efficient in all scenarios.

In addition, we want to test if there is a limit when dividing the agents in groups. We can suppose that if the agents are too divided, the performances would drop because the agents would not be enough to accomplish some hard tasks.

Figure 5.14 presents the results of this first experimentation. The average obtained for each scenario is represented on the horizontal axis. One can observe that using one group or three groups do not seem to be the best strategies. For all the scenarios, these strategies have the worst score three times each. Also they only obtain the best score one time each. On the other hand, dividing the agents into two groups seems to be a more stable strategy. This solution obtains the best score four times and it is never the worst. It is thus the best strategy for this scheduling system. However, we have to accept that it is not the best strategy for all scenarios.

Besides, those results confirm the hypothesis that it is not always advantageous to divide the agents into many groups. Too many divisions can have a negative impact on the performance. In our tests, we can see that even with only three groups, the performances begin to be more unstable. This can be explained by the fact that when we divide the agents, the time necessary to accomplish a task increases, thus penalizing other urgent tasks.

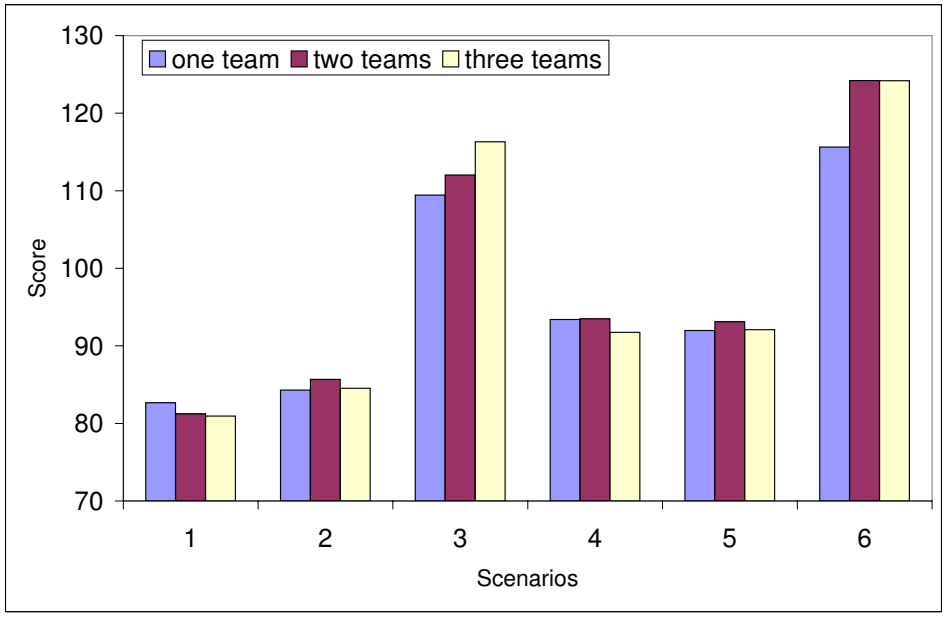


Figure 5.14: Comparison of three different scheduling strategies.

5.4.4 Comparison With Another Team

This experimentation has been done for testing concretely and objectively our global approach, which is to model the multiagent problem as a task scheduling problem. The preceding tests compared different scheduling systems, but how does our approach compare to other methods? To achieve that, we have used the same six scenarios in order to measure the performances obtained by the agents of another team. The other team is the *ResQFreiburg* team that finished first at the 2004 international competition in Portugal. In brief, their approach consisted of choosing the civilians to rescue using genetic algorithms (Kleiner et al. (2005)). We have compared their agents with our agents using the strategy of dividing the agents into two groups.

Figure 5.15 shows the results of those comparisons. The ordinate axis corresponds to the average score obtained for each scenario. The performances are quite similar, but we can see that our approach is the best in four scenarios out of six. If we look more closely, those results show that dividing the agents into two groups is not always the best division to do. If we look at the figures 5.14 and 5.15, we can see that the scheduling strategy with only one group would have been the best and would have beat *ResQFreiburg* in the first scenario.

To sum up, those results are convincing and show that our approach consisting of modelling the system with a task scheduling formalism seems to have a good potential. It helps to extract the important characteristics of the problem and afterwards to use efficient algorithms to solve the problem.

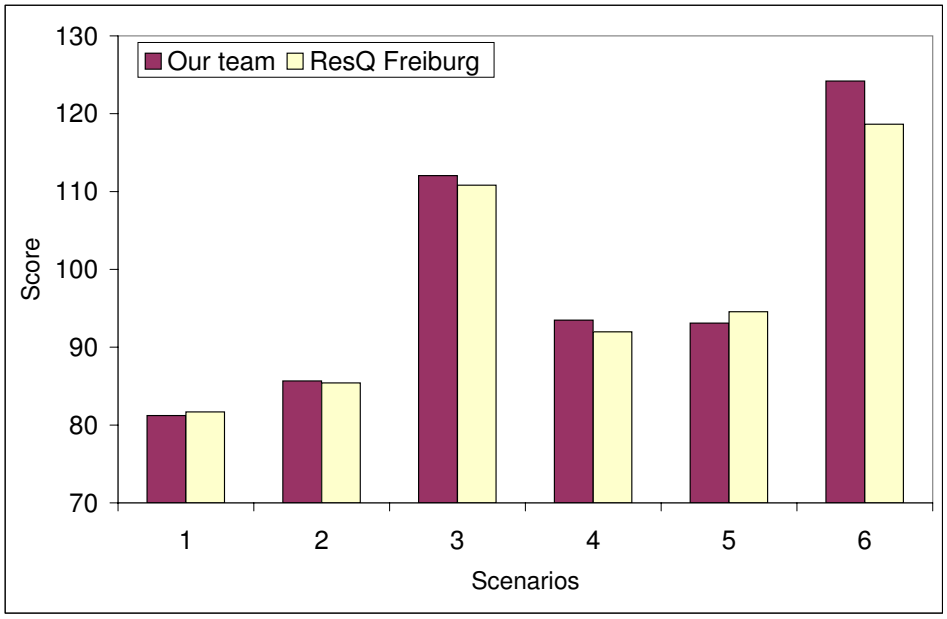


Figure 5.15: Comparison with the ResQFreiburg team.

5.5 Contributions

In this chapter, we presented a scheduling approach for the problem of rescuing civilians in the RoboCupRescue simulation. This scheduling approach uses a K-Nearest-Neighbors learning algorithm to learn one task characteristic. In this last section, we summarize our contributions:

Links between multiagent and scheduling systems. We have showed some possible links between multiagent systems and task scheduling systems. We presented a methodology defining in a structured way the main steps necessary to extract from a multiagent system the scheduling problem and mostly how to structure the solution to this scheduling problem. We have also emphasized the usefulness of keeping the domains of task scheduling and multiagent systems linked. Those two domains can help each other to find good solutions to common problems.

Reduction of the communication. We have showed that a decentralized scheduling system can offer the same performances as a centralized one, while diminishing the amount of information transmitted between the agents. This was done in the objective of being more robust to constraints on the communications.

Experimentations in the RoboCupRescue. We have demonstrated the efficiency of the decentralized approach on two experiments in a complex environment (partially observable, uncertain and real-time). We have shown that the

decentralized approach is more robust to changes in the communication capacities, while being as efficient as the centralized approach. Furthermore, we have demonstrated the efficiency of our scheduling approach by comparing our agents with the agents of the wining team of the 2004 RoboCupRescue international competition.

Distribution of the execution. We have tested different divisions of the resources in order to accomplish the tasks faster. We have tested with one, two or three groups. The strategy with one group has the advantage of using an optimal algorithm. However, when the resources are divided, we have to use an approximation scheduling algorithm.

K-Nearest-Neighbors Algorithm. We have presented a new application of the K-Nearest-Neighbors algorithm to estimate the value of an uncertain parameter of a task. We have presented results showing the efficiency of the predictions in the RoboCupRescue simulation. We have also presented a strategy to manage the missing attribute values by simply ignoring them when calculating the distances between the instances. We have presented results showing that ignoring the missing values is a more efficient approach than trying to estimate their values when many values are missing.

Chapter 6

Conclusion

The problems of decision-making and task coordination are really important to the field of multiagent systems. When the environment is only partially observable, the decision process of the agents may become quite hard. Agents then have to choose their actions based on incomplete information. It becomes difficult for the agents to stay coordinated when they cannot perceive the other agents and when the environment is in constant changes.

In such complex cooperative multiagent systems, this thesis presents some algorithms to coordinate the agents in order to accomplish complex tasks that need more than one agent to be accomplished. This thesis also presents an online POMDP algorithm that can choose efficient actions in large partially observable environments.

In this chapter, we first present a summary of the approaches developed in this thesis and we finish by presenting some open problems for future work.

6.1 Summary

In this thesis, we have addressed different issues that arises when dealing with complex cooperative multiagent systems. In Chapter 2, we described our test-bed environment: the RoboCupRescue simulation. We showed that it was a complex environment that imposes many constraints such as: real-time constraints, partial observability, limited communications, limited resources, etc. We also emphasized the fact that the RoboCupRescue simulation environment offers many research opportunities like: Multiagent planning, anytime planning, resources management, learning, information gathering, coordination, scheduling, etc.

We then presented, in the following chapters, our original approaches to deal with

different problems in cooperative multiagent systems. All these approaches have been tested in the RoboCupRescue environment to show their applicability and efficiency in such a complex cooperative multiagent environment. The next three sub-sections summarize these approaches and the results obtained.

6.1.1 Online POMDP Algorithm

In Chapter 3, we have presented our Real-Time Belief State Search (RTBSS) algorithm, which is a new online algorithm for partially observable Markov decision processes (POMDP). The RTBSS algorithm is based on a look-ahead search in the belief state space to find the best action to execute at each cycle in the environment. This algorithm only explores reachable belief states starting from the agent's current belief state.

By doing an online search, we avoid the overwhelming complexity of computing a policy for every possible situation the agent could encounter. Since there is no computation offline, the algorithm is immediately applicable to previously unseen environments, if the environments' dynamics are known.

This online exploration has to be as fast as possible, since our algorithm has to work under some real-time constraints in the RoboCupRescue simulation. To achieve that, we opted for a factored POMDP representation and a branch and bound strategy. We have combined a limited depth first search strategy with a pruning strategy that uses dynamically updated bounds based on the solutions found at the maximal depth of the search. The pruning of the tree is also accelerated by sorting the actions in order of their expected efficiency. Since the more interesting actions are tried first, there is more chance that the first branches developed have better values, thus better bounds for the pruning condition.

In addition, we presented some hybrid approaches that use the RTBSS online search strategy mixed with approximate offline strategies. We presented three new algorithms: RTBSS-QMDP, RTBSS-PBVI-QMDP and RTDPBSS. The RTBSS-QMDP algorithm uses the Q_{MDP} value function as the utility of the belief states at the leaves of the RTBSS search tree. The RTBSS-PBVI-QMDP algorithm uses the PBVI value function as a lower bound at the leaves of the tree and the Q_{MDP} value function as an upper bound for the pruning function. Finally, the RTDPBSS algorithm is exactly like the RTDP algorithm except that the RTDPBSS algorithm does a deeper search online when choosing the actions. The results of our experiments showed that the performances of the hybrid approaches are often better than the performances of the online approach or the offline approach taken alone. Our results have shown that RTBSS-QMDP is the most consistent approach over all the test environments.

We have conceived an approach to maintain a belief state based on the real agent's observations. This helps an agent to manage the highly dynamic and unpredictable parts of the environment. During the search in the belief state space, the agent considers some variables fixed and concentrate only on the most important parts of the environment to choose its actions. This approach is possible with the RTBSS algorithm because it is an online algorithm that can readjust its belief states between each execution in order to stay up to date with the agent's observations.

Moreover, we defined a local reward function enabling an agent using RTBSS to redefine a reward function before each action's choice. This enabled to define the reward function only for the current situation, which is really useful when there are a lot of possible situations. This again is possible because the agent's policy is dynamically defined at each time step by the online RTBSS algorithm. Therefore, the reward function can be modified before each decision, because the RTBSS algorithm does a new search for each action's choice. This local reward function can be used to dynamically coordinate many agents in an environment without any coordination related messages. This new multiagent POMDP coordination approach has shown to be effective and quite flexible to control many agents in a highly dynamic environment.

6.1.2 Task Allocation Learning

In Chapter 4, we have presented a learning algorithm which is useful to learn the required number of agents for each different task in a complex cooperative multiagent environment. Most coordination learning approaches considered that the number of required resources to accomplish a task is known or that they have enough information to have a probability distribution over the number of required resources. In our approach, we considered that the tasks are complex and that the agents have to learn this information since it is not available.

The tasks considered in our simulations are described with discrete and continuous attributes. Therefore, there are a lot of possible task descriptions. To manage this complexity, we have adapted a selective perception reinforcement learning algorithm to the problem of learning the required number of resources to accomplish a task. With this algorithm we can find a generalization of the task description space, thus allowing the reinforcement learning algorithm to work on smaller task description spaces.

Furthermore, we proposed a coordination algorithm using the information learned about the number of resources needed for a task. This algorithm uses really few messages between the agents, which is interesting in environments with limited and/or unreliable communications.

We have presented some tests in the RoboCupRescue environment showing that

the agents can efficiently learn and that the learned information is really helpful to improve the agents' performances. The agents obtained good results with an internal task description space of only 0.5% of the complete task description space. We have also shown results taken during the 2004 international competition showing that we were the most efficient team to extinguish fires.

6.1.3 Task Scheduling in Complex Multiagent Environments

In Chapter 5, we presented a scheduling approach for the problem of rescuing civilians in the RoboCupRescue simulation. We analyzed the advantages and the disadvantages of distributing or not the scheduling process in a complex multiagent system. More precisely, we studied the impact on the agents' efficiency and on the amount of information transmitted when using centralized and decentralized scheduling. We also study the usefulness of distributing the execution of the tasks in a scheduling problem and thus accomplishing goals in parallel, compared to the strategy of concentrating all resources to accomplish one goal at a time.

We have showed that a decentralized scheduling system can offer the same performances as a centralized one, while diminishing the amount of information transmitted between agents. This was done in the objective of being more robust relatively to constraints on the communications.

We have also tested different divisions of the resources in order to accomplish the tasks faster. We have tested with one, two or three groups. The strategy with one group has the advantage of using an optimal algorithm. However, when the resources are divided, we have to use an approximation scheduling algorithm. The results in the RoboCupRescue simulation have shown that the best approach was to divide the agents in two groups.

In similarity with the approach presented in Chapter 4, we also had to learn to estimate one of the characteristics of the tasks. To this end, we developed a K-Nearest-Neighbors (KNN) approach that has been used to learn the damage progression of the civilians which is used to estimate the expected death time of the civilians. We have presented results showing the efficiency of the predictions in the RoboCupRescue simulation. We have also presented a strategy to manage the missing attribute values by simply ignoring them when calculating the distances between the instances. We have presented results showing that ignoring the missing values is a more efficient approach than trying to estimate their values when many values are missing.

6.2 Future Work

The algorithms presented in this thesis are a contribution to the field of cooperative multiagent systems. However, there is much work to do in order to developed near optimal cooperative multiagent systems in complex environments. In this section, we present some ideas on how the approaches presented in this thesis could be extended. These ideas are divided according to the main three chapters of this thesis.

6.2.1 Online POMDP

One improvement for our RTBSS algorithm could be to reuse the information computed previously in the simulation. This would allow being able to explore in greater depth without using more computation time. During a search to choose an action, the RTBSS algorithm calculates many estimated values for the belief states encountered during the search. It would be interesting to reuse these estimations if the same belief states are encountered in future searches. The RTDPBSS algorithm, presented in this thesis, is a first step toward this goal, but this algorithm took a lot of memory. On the bigger problems, this algorithm often ran out of memory before reaching the time limit. This is one drawback of this approach, because recording belief states values takes a lot of space since there are an infinite number of possible belief states. The RTDPBSS algorithm used some discretization of the belief state space in order to limit the number of belief states considered, but this has a negative impact on the performances, because many different belief states are regrouped together. To sum up, it would be interesting to find an efficient way to record past estimated belief state values in order to accelerate the search.

Moreover, when using the factorization, it could be interesting to study the impact of removing not only values with a 0 probability, but also values with a really small probability. By doing so, the set of possible states ω would be smaller. However, it would be important to be careful not to remove useful values.

In addition, it would be interesting to have a way to automatically find the best depth for the search. This best depth would have to be dependant on the available time to search in the belief state space online.

Another future work could be to look at other hybrid approaches. In our experiments, the hybrid approaches have often shown some improvements compared to completely online or offline approaches. Therefore, we think that the hybrid approaches have the most potential to deal with complex POMDP environments.

6.2.2 Learning Task's Characteristics

In Chapter 4, we presented a reinforcement learning based on selective perception. This algorithm adds some state distinctions by growing a tree representation of the state space. The tree was grown for a fix number of simulations. It would be interesting to define a measure of performance that could automatically find the best depth of the tree. This measure of performance would have to balance the quality of the state's distinction with the time needed by the reinforcement learning algorithm if there are more states.

6.2.3 Scheduling

For the scheduling approach presented in Chapter 5, we have presented results in which the agents were divided in one, two or three groups. As our results have shown, there was no perfect division that was the best in all situations. Consequently, it would be interesting to develop a scheduling algorithm that can find the optimal division in all situations while still respecting the real-time constraints.

Another improvement of our scheduling approach would be to consider the moving time between two tasks (s_{jk}). In our experiments, we have used a constant time, but the scheduler agent could generate better schedules if the real moving time was considered. The problem with these flexible moving times is that they are different from one task pair to another. Therefore, this complicates the scheduling problem, which is already NP-Hard. Using better estimations for the moving times could be a first step to improve the schedules.

Bibliography

Aberdeen, D. (2003a). A (revised) survey of approximate methods for solving partially observable markov decision processes. Technical report, National ICT Australia.

Aberdeen, D. and Baxter, J. (2002). Scaling Internal-State Policy-Gradient Methods for POMDPs. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 3–10, Sydney, Australia.

Aberdeen, D. A. (2003b). *Policy-Gradient Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, The Australian National University.

Abul, O., Polat, F., and Alhajj, R. (2000). Multiagent Reinforcement Learning Using Function Approximation. *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Application and Reviews*, 30(4).

Acuna, E. and Rodriguez, C. (2004). The Treatment of Missing Values and its Effect in the Classifier Accuracy. In Banks, D., House, L., McMorris, F., Arabie, P., and Gaul, W., editors, *Classification, Clustering and Data Mining Applications*, pages 639–648, Berlin-Heidelberg. Springer-Verlag.

Agogino, A. K. and Tumer, K. (2005). Multi Agent Reward Analysis for Learning in Noisy Domains. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-05)*, pages 81–88, Utrecht, Netherlands.

Ahmadi, M., Sayyadian, M., and Habibi, J. (2002). A Learning Method for Evaluating Messages in Multi-Agent Systems. In *Proceedings of the Agent Communication Languages and Conversation Policies, AAMAS'02 Workshop*, Bologna, Italy.

Astrom, K. J. (1965). Optimal Control of Markov Decision Processes with Incomplete State Estimation. *Journal of Mathematical Analysis and Applications*, 10:174–205.

Becker, R., Zilberstein, S., Lesser, V., and Goldman, C. V. (2003). Transition-Independent Decentralized Markov Decision Processes. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-03)*, pages 41–48, Melbourne, Australia. ACM Press.

Berenji, H. R. and Vengerov, D. A. (1999). Cooperation and Coordination Between Fuzzy Reinforcement Learning Agents in Continuous-State Partially Observable Markov Decision Processes. In *Proceedings of the 8th IEEE International Conference on Fuzzy Systems (FUZZ-IEEE'99)*.

Berenji, H. R. and Vengerov, D. A. (2000). Learning, Cooperation, and Coordination in Multi-Agent Systems. Technical Report IIS-00-10, Intelligent Inference Systems Corp.

Bernstein, D. S., Givan, R., Immerman, N., and Zilberstein, S. (2002). The Complexity of Decentralized Control of Markov Decision Processes. *Mathematics of Operations Research*, 27(4):819–840.

Bernstein, D. S., Hansen, E. A., and Zilberstein, S. (2005). Bounded Policy Iteration for Decentralized POMDPs. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, Edinburgh, Scotland.

Bertsekas, D. P. (2001). *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Beynier, A. and Mouaddib, A. (2005). A Polynomial Algorithm for Decentralized Markov Decision Processes with Temporal Constraints. In *Proceedings of the Fourth International Autonomous Agents and Multiagent Systems Conference (AAMAS-05)*, Utrecht, Netherlands.

Beynier, A. and Mouaddib, A.-I. (2004). Non-Communicative DEC-MDP for cooperative Multi-agent systems. In *Proceedings of the ECAI Workshop on Multi-Agent Decision Processes : Theories and Models*.

Blazewick, J. (2001). *Scheduling computer and manufacturing processes*. Springer.

Bonarini, A. and Trianni, V. (2001). Learning Fuzzy Classifier Systems for Multi-Agent Coordination. *Information Sciences*, 136:215–239.

Bonet, B. (2002). An Epsilon-Optimal Grid-Based Algorithm for Partially Observable Markov Decision Processes. In *Proceedings of The Nineteenth International Conference on Machine Learning (ICML-2002)*, pages 51–58.

Boutilier, C. (1996). Planning, Learning and Coordination in Multiagent Decision Processes. In *Proceedings of TARK-96: Theoretical Aspects of Rationality and Knowledge*, De Zeeuwse Stromen, Hollande.

Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic Dynamic Programming with Factored Representations. *Artificial Intelligence*, 121:49–107.

Boutilier, C., Friedman, N., Goldszmidt, M., and Koller, D. (1996). Context-Specific Independance in Bayesian Networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI-96)*, pages 115–123, Portland, OR.

Boutilier, C. and Poole, D. (1996). Computing Optimal Policies for Partially Observable Decision Processes Using Compact Representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1168–1175, Portland, OR.

Boyen, X. and Koller, D. (1998). Tractable Inference for Complex Stochastic Processes. In *In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 33–42.

Brafman, R. I. (1997). A Heuristic Variable Grid Solution Method for POMDPs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 76–81, Providence, Rhode Island. AAAI Press / MIT Press.

Braziunas, D. and Boutilier, C. (2004). Stochastic local search for pomdp controllers. In *The Nineteenth National Conference on Artificial Intelligence (AAAI-04)*.

Brenner, M., Kleiner, A., Exner, M., Degen, M., Metzger, M., Nussle, T., and Thon, I. (2005). ResQ Freiburg: Deliberative Limitation of Damage. In Nardi, D., Riedmiller, M., and Sammut, C., editors, *RoboCup-2004: Robot Soccer World Cup VIII*, Berlin. Springer Verlag.

Brucker, P. (2001). *Scheduling Algorithms*. Springer.

Buffet, O. (2000). Apprentissage par renforcement dans un système multi-agents. Master's thesis, Université Henri Poincarré - Nancy I.

Bui, H. H., Venkatesh, S., and Kieronska, D. (1998). A Framework for Coordination and Learning among Team of Agents. *Lecture Notes in Computer Science*, 1441.

Caruana, R. (2001). A Non-Parametric EM-Style Algorithm for Imputing Missing Values. In *Proceedings of the Artificial Intelligence and Statistics*.

Cassandra, A., Littman, M. L., and Zhang, N. L. (1997). Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 54–61.

Cassandra, A. R. (1998). *Exact and approximate algorithms for partially observable markov decision processes*. PhD thesis, Brown University.

Chalkiadakis, G. and Boutilier, C. (2003). Coordination in Multiagent Reinforcement Learning: A Bayesian Approach. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)*, Melbourne, Australia.

Chapman, D. and Kaelbling, L. P. (1991). Learning from delayed reinforcement in a complex domain. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*.

Cheng, H. (1988). *Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, University of British Columbia - School of Commerce.

Chrisman, L. (1992). Reinforcement Learning with Perceptual Aliasing: The Perceptual Distinctions Approach. In *Proceedings of the National Conference on Artificial Intelligence*, pages 183–188.

Claus, C. and Boutilier, C. (1998). The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 746–752, Madison.

Cover, T. M. and Thomas, J. A. (1991). *Elements of Information Theory*. Wiley.

Crites, R. H. and Barto, A. G. (1998). Elevator Group Control Using Multiple Reinforcement Learning Agents. *Machine Learning*, 33(2-3):235–262.

Dean, T. and Kambhampati, S. (1996). Planning and Scheduling. In Tucker, A. B., editor, *The CRC Handbook of Computer Science and Engineering*, pages 614–636. CRC press.

Dean, T. and Kanazawa, K. (1989). A Model for reasoning About Persistence and Causation. *Computational Intelligence*, 5(3):142–150.

Decker, K. S. and Lesser, V. R. (1993). Quantitative Modeling of Complex Environments. *International Journal of Intelligence Systems in Accounting, Finance, and Management*, 2(4):215–234. Special issue on Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior.

Dietterich, T. G. (1998). The MAXQ Method for Hierarchical Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning*, pages 118–126, San Francisco.

Dixon, J. K. (1979). Pattern Recognition with Partly Missing Data. *IEEE Transactions on Systems, Man, and Cybernetics*, 9:617–621.

Doshi, P. and Gmytrasiewicz, P. (2005). Approximating State Estimation in Multiagent Settings using Particle Filters. In *In Proceedings of the Fourth International Autonomous Agents and Multiagent Systems Conference (AAMAS-05)*, Utrecht, Netherlands.

Durfee, E. H. (1999). Distributed Problem Solving and Planning. In Weiss, G., editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter 3, pages 121–164. The MIT Press, Cambridge, MA.

Dutech, A. (2000). Solving POMDPs Using Selected Past Events. In *Proceedings of the European Conference on Artificial Intelligence (ECAI-2000)*, Berlin.

Dutech, A., Buffet, O., and Charpillet, F. (2001). Multi-Agent Systems by Incremental Gradient Reinforcement Learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence IJCAI-01*, pages 833–838, Seattle.

Dutech, A. and Samuelides, M. (2003). Apprentissage par renforcement pour les processus décisionnels de markov partiellement observés. *Revue d'Intelligence Artificielle*, 17(4).

Estlin, T., Gaines, D., Fisher, F., and Castano, R. (2005). Coordinating Multiple Rovers with Interdependent Science Objectives. In *Proceedings of the Fourth International Autonomous Agents and Multiagent Systems Conference (AAMAS-05)*, Utrecht, Netherlands.

Excelente-Toledo, C. B. and Jennings, N. R. (2002). Learning to Select a Coordination Mechanism. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, Bologna, Italie.

Excelente-Toledo, C. B. and Jennings, N. R. (2004). The Dynamic Selection of Coordination Mechanisms. *Journal of Autonomous Agents and Multi-Agent Systems*, 9(1-2):55–85.

French, S. (1982). *Sequencing and Scheduling*. Wiley.

Garland, A. and Alterman, R. (2001). Learning Procedural Knowledge to Better Coordinate. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence IJCAI-01*, pages 1073–1083, Seattle.

Garland, A. and Alterman, R. (2004). Autonomous Agents that Learn to Better Coordinate. *Autonomous Agents and Multi-Agent Systems*, 8(3):267–301.

Garland, A. E. (2000). *Learning to Better Coordinate in Joint Activities*. PhD thesis, Brandeis University.

Geffner, H. and Bonet, B. (1998). Solving Large POMDPs Using Real Time Dynamic Programming. Working notes. Fall AAAI symposium on POMDPs.

Ghavamzadeh, M. and Mahadevan, S. (2002). A Multiagent Reinforcement Learning Algorithm by Dynamically Merging Markov Decision Processes. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, Bologna, Italie.

Ghavamzadeh, M. and Mahadevan, S. (2004). Learning to Communicate and Act using Hierarchical Reinforcement Learning. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2004)*, pages 1114–1121, New-York.

Ghavamzadeh, M., Mahadevan, S., and Makar, R. (2005). Hierarchical Multiagent Reinforcement Learning. Submitted to the Journal of Autonomous Agents and Multi-Agent Systems.

Gmytrasiewicz, P. and Doshi, P. (2005). A Framework for Sequential Planning in Multi-Agent Settings. *Journal of Artificial Intelligence Research*, 24:49–79.

Goldman, C. V. and Zilberstein, S. (2004). Decentralized Control of Cooperative Systems: Categorization and Complexity Analysis. *Journal of Artificial Intelligence Research*, 22:143–174.

Gonzalez, M. J. (1977). Deterministic processor scheduling. *ACM Computing Surveys*, 9(3):173–204.

Grudzinski, K. and Duch, W. (2000). SBL-PM: A Simple Algorithm for Selection of Reference Instances for Similarity Based Methods. In *Proceedings of Intelligent Information Systems (IIS-2000)*, pages 99–108. Physica Verlag (Springer).

Hansen, E. A. (1997). An Improved Policy Iteration Algorithm for Partially Observable MDPs. In *Tenth Neural Information Processing Systems Conference (NIPS-97)*, Denver, Colorado.

Hansen, E. A. (1998). Solving POMDPs by Searching in Policy Space. In *Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 211–219, Madison, Wisconsin.

Hansen, E. A. and Feng, Z. (2000). Dynamic Programming for POMDPs Using a Factored State Representation. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 130–139, Breckenridge, CO.

Hansen, E. A. and Zhou, R. (2003). Synthesis of Hierarchical Finite-State Controllers for POMDPs. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-03)*, Trento, Italy.

Hauskrecht, M. (1997). Incremental Methods for Computing Bounds in Partially Observable Markov Decision Processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 734–739.

Hauskrecht, M. (2000). Value-Function Approximations for Partially Observable Markov Decision Processes. *Journal of Artificial Intelligence Research*, 13:33–94.

Haynes, T. and Sen, S. (1998). Learning cases to resolve conflicts and improve group behavior. *International Journal of Human-Computer Studies*, 48:31–49.

Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (1999). SPUDD: Stochastic Planning Using Decision Diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 279–288, Stockholm.

Horling, B. and Lesser, V. (1999). Using Diagnosis to Learn Contextual Coordination Rules. In *Proceedings of the AAAI-99 Workshop on Reasoning in context for AI Applications*, pages 70–74.

Jackson, J. R. (1955). Scheduling a production line to minimize maximum tardiness. Research Report 43, Management Science, University of California, Los Angeles, CA.

James, M. and Singh, S. (2004). Learning and Discovery of Predictive State Representations in Dynamical Systems With Reset. In *Proceedings of the Twenty First International Conference on Machine Learning (ICML-2004)*, Banff, Canada.

Jensen, D., Atighetchi, M., Vincent, R., and Lesser, V. (1999). Learning Quantitative Knowledge for Multiagent Coordination. In *16th National Conference on Artificial Intelligence (AAAI-99)*, pages 24–31, Orlando.

Jones, A. and Rabelo, J. (1998). Survey of job shop scheduling techniques. Technical report, National Institute of Standards and Technology, Gaithersburg, MD.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101.

Kapetanakis, S. and Kudenko, D. (2002). Reinforcement Learning of Coordination in Cooperative Multi-Agent Systems. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-02).*

Karim, S. and Heinze, C. (2005). Experiences with the Design and Implementation of an Agent-based Autonomous UAV Controller. In *Proceedings of the Fourth International Autonomous Agents and Multiagent Systems Conference (AAMAS-05)*, pages 19–26, Utrecht, Netherlands.

Kearns, M., Mansour, Y., and Ng, A. Y. (2000). Approximate Planning in Large POMDPs via Reusable Trajectories. In Solla, S., Leen, T., and Muller, K.-R., editors, *Advances in Neural Information Processing Systems 12*. MIT Press.

Kearns, M., Mansour, Y., and Ng, A. Y. (2002). A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes. *Machine Learning*, 49(2-3):193–208.

Kitano, H. (2000). Robocup rescue: A grand challenge for multi-agent systems. In *Proceedings of ICMAS 2000*, Boston, MA.

Kitano, H., Tadokor, S., Noda, H., Matsubara, I., Takhasi, T., Shinjou, A., and Shimada, S. (1999). Robocup-rescue: Search and rescue for large scale disasters as a domain for multi-agent research. In *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics (SMC-99)*.

Kleiner, A., Brenner, M., Brauer, T., Dornhege, C., Gobelbecker, M., Luber, M., Prediger, J., Stuckler, J., and Nebel, B. (2006). Successful Search and Rescue in Simulated Disaster Areas. In Noda, I., Jacoff, A., Bredenfeld, A., and Takahashi, Y., editors, *RoboCup-2005: Robot Soccer World Cup IX*. Springer Verlag, Berlin.

Kleiner, A., Brenner, M., Bräuer, T., Dornhege, C., Göbelbecker, M., Luber, M., Prediger, J., and Stückler, J. (2005). Resq freiburg: Team description and evaluation. In Nardi, D., Riedmiller, M., and Sammut, C., editors, *RoboCup-2004: Robot Soccer World Cup VIII*. Springer Verlag.

Koch, E. (2002). Simulation multiagent de situations d'urgence dans le cadre de la RobocupRescue. Master's thesis, Facultés Universitaires Notre Dame de la Paix.

Koenig, S. (2001). Agent-Centered Search. *AI Magasine*, 22(4):109–131.

Lawer, E., Lenstra, J., and Kan, A. R. (1982). Recent developments in deterministic sequencing scheduling : A servey. *Deterministic and Stochastic Scheduling*, pages 35–74.

Littman, M. L. (1994a). Markov Games as a Framework for Multi-Agent Reinforcement Learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163, San Francisco, CA. Morgan Kaufmann.

Littman, M. L. (1994b). The witness algorithm: Solving partially observable markov decision processes. Technical Report CS-94-40, Brown University.

Littman, M. L. (1996). *Algorithms for Sequential Decision Making*. PhD thesis, Brown University.

Littman, M. L., Cassandra, A. R., and Kaelbling, L. P. (1995). Learning Policies for Partially Observable Environments: Scaling Up. In *Proceedings of the 12th International Conference on Machine Learning (ICML-95)*.

Littman, M. L., Sutton, R. S., and Singh, S. (2001). Predictive Representation of State. In *Proceedings of Advances in Neural Information Processing Systems (NIPS-2001)*, Vancouver.

Lovejoy, W. S. (1991). Computationally Feasible Bounds for POMDPs. *Operations Research*, 39(1).

Makar, R., Mahadevan, S., and Ghavamzadeh, M. (2001). Hierarchical Multi-Agent Reinforcement Learning. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-2001)*, pages 246–253, Montreal, Canada.

Mali, A. D. and Kambhampati, S. (1999). Distributed Planning. In *The Encyclopaedia of Distributed Computing*. Kluwer Academic Publishers.

Malone, T. W. and Crowston, K. (1994). The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1).

Mataric, M. J. (1994). *Interaction and Intelligent Behavior*. PhD thesis, Massachusetts Institute of Technology.

Mataric, M. J. (1997). Learning Social Behavior. *Robotics and Autonomous Systems*, 20:191–204.

McAllester, D. and Singh, S. (1999). Approximate Planning for Factored POMDPs using Belief State Simplification. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 409–416, San Francisco, CA. Morgan Kaufmann Publishers.

McCallum, A. K. (1996). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester, New-York.

Meuleau, N., Kim, K.-E., Kaelbling, L. P., and Cassandra, A. R. (1999a). Solving POMDPs by searching the space of finite policies. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 417–426, San Francisco. Morgan Kaufmann.

Meuleau, N., Peshkin, L., Kim, K.-E., and Kaelbling, L. P. (1999b). Learning Finite-State Controllers for Partially Observable Environments. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 427–436, San Francisco. Morgan Kaufmann.

Mitchell, T. M. (1997). *Machine Learning*. MIT Press and The McGraw-Hill Compagnies, Inc.

Monahan, G. E. (1982). A Survey of Partially Observable Markov Decision Processes: Theory, Models and Algorithms. *Management Science*, 28(1-16).

Moore, A. W. (1993). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. In *Proceedings of Advances of Neural Information Processing Systems (NIPS 6)*, pages 711–718. Morgan Kaufmann.

Moore, J. (1968). An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Sci.*, 15:102–109.

Morring, B. D. and Martinez, T. R. (2004). Weighted Instance Typicality Search (WITS): A Nearest Neighbor Data Reduction Algorithm. *Intelligent Data Analysis*, 8(1):61–78.

Murphy, K. P. (2000). A Survey of POMDP Solution Techniques. Technical report, U.C. Berkeley.

Nair, R., Tambe, M., and Marsella, S. (2003). Team Formation for Reformation in Multiagent Domains like RoboCupRescue. In Kaminka, G., Lima, P., and Roja, R., editors, *Proceedings of RoboCup-2002 International Symposium*, Lecture Notes in Computer Science. Springer Verlag.

Nilsson, N. (1980). *Principles of Artificial Intelligence*. Tioga Publishing.

Noda, I. (2001). Rescue Simulation and Location-based Communication Model. In *Proc. of SCI-2001*.

Norman, M. G. and Thanisch, P. (1993). Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302.

Nourbakhsh, I., Sycara, K., Koes, M., Yong, M., Lewis, M., and Burion, S. (2005). Human-Robot Teaming for Search and Rescue. In *IEEE Pervasive Computing: Mobile and Ubiquitous Systems*, pages 72–78.

Ohta, M., Takahashi, T., and Kitano, H. (2001). RoboCup-Rescue Simulation: in case of Fire Fighting Planning. In Stone, P., Balch, T., and Kraetzschmar, G., editors, *RoboCup 2000*, volume 2019 of *Lecture Notes in Artificial Intelligence*, pages 351–356. Springer-Verlag.

Panait, L. and Luke, S. (2003). Cooperative Multi-Agent Learning: The State of the Art. Technical Report GMU-CS-TR-2003-1, Department of Computer Science, George Mason University.

Papadimitriou, C. and Tsitsiklis, J. N. (1987). The complexity of markov decision processes. *Mathematics of Operations Research*, 12(3):441–450.

Paquet, S. (2001). Coordination de plans d'agents: Application à la gestion des ressources d'une frégate. Master's thesis, Université Laval.

Paquet, S., Bernier, N., and Chaib-draa, B. (2004a). Comparison of Different Coordination Strategies for the RoboCupRescue Simulation. In *Proceedings of The 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2004)*, volume 3029 of *Lecture Notes in Artificial Intelligence*, pages 987–996, Ottawa, Canada. Springer-Verlag.

Paquet, S., Bernier, N., and Chaib-draa, B. (2004b). Selective Perception Learning for Tasks Allocation. In *AAMAS-04 Workshop on Learning and Evolution in Agent Based Systems*, New York.

Paquet, S., Bernier, N., and Chaib-draa, B. (2005a). Multiagent Systems Viewed as Distributed Scheduling Systems: Methodology and Experiments. In *Proceedings of the Eighteenth Canadian Conference on Artificial Intelligence (AI-2005)*, Victoria, Canada.

Paquet, S., Tobin, L., and Chaib-draa, B. (2005b). An Online POMDP Algorithm for Complex Multiagent Environments. In *Proceedings of The fourth International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-05)*, Utrecht, The Netherlands.

Paquet, S., Tobin, L., and Chaib-draa, B. (2005c). Prise de Décision en Temps-réel pour des POMDPs de Grande Taille. *Revue d'intelligence artificielle: numéro spécial - Décision et planification dans l'incertain*.

Parr, R. and Russel, S. (1995). Approximating Optimal Policies for Partially Observable Stochastic Domains. In *Proceedings of the fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1088–1094, Montreal, Canada. Morgan Kauffman.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Pineau, J. (2004). *Tractable Planning Under Uncertainty: Exploiting Structure*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

Pineau, J., Gordon, G., and Thrun, S. (2003). Point-based value iteration: An anytime algorithm for pomdps. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1025–1032, Acapulco, Mexico.

Pinedo, M. (1995). *Scheduling: Theory, Algorithms and Systems*. Prentice Hall.

Poole, D. (1993). Probabilistic Horn Abduction and Bayesian Networks. *Artificial Intelligence*, 64(1):81–129.

Poole, D. (1997). Probabilistic Partial Evaluation: Exploiting Rule Structure in Probabilistic Inference. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1284–1291, Nagoya, Japan.

Poupart, P. (2005). *Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes*. PhD thesis, University of Toronto.

Poupart, P. and Boutilier, C. (2001). Vector-Space Analysis of Belief-State Approximation for POMDPs. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI-2001)*, pages 445–452, Seattle.

Poupart, P. and Boutilier, C. (2003a). Bounded Finite State Controllers. In *Advances in Neural Information Processing Systems 16 (NIPS-2003)*, Vancouver, Canada.

Poupart, P. and Boutilier, C. (2003b). Value-Directed Compression of POMDPs. In *Proceedings of Advances in Neural Information Processing Systems (NIPS-2003)*, volume 15.

Prasad, M. N., Lesser, V., and Lander, S. (1996). Learning Organizational Roles in a Heterogeneous Multi-Agent System. In *Proceedings of the Second International Conference on Multiagent Systems*, pages 291–298.

Prasad, M. V. N. (1997). *Learning Situation-Specific Control in Multi-Agent Systems*. PhD thesis, University of Massachusetts Amherst.

Prasad, M. V. N. and Lesser, V. R. (1999). Learning Situation-Specific Coordination in Cooperative Multi-agent Systems. *Autonomous Agents and Multi-Agent Systems*, 2(2):173–207.

Pyeatt, L. D. and Howe, A. E. (1995). Decision tree function approximation in reinforcement learning. Technical Report TR CS-98-112, Colorado State University, Fort Collins, Colorado.

Quinlan, J. R. (1993a). *C4.5 Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.

Quinlan, J. R. (1993b). Combining instance-based and model-based learning. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 236–243, Amherst, Massachusetts. Morgan Kaufmann.

RoboCup (2003). RoboCup Official Site. [Online]. http://www.robocup.org (Page visited on february 23, 2003).

Ron, D., Singer, Y., and Tishby, N. (1994). Learning probabilitic automata with variable memory length. In *Proceedings of Computational Learning Theory*. ACM press.

Rosencrantz, M., Gordon, G., and Thrun, S. (2004). Learning Low Dimentional Predictive Representations. In *Proceedings of the Twenty First International Conference on Machine Learning (ICML-2004)*, Banff, Canada.

Roy, N. (2003). *Finding Approximate POMDP Solutions Through Belief Compression*. PhD thesis, Carnegie Mellon University.

Roy, N. and Gordon, G. (2003). Exponential Family PCA for Belief Compression in POMDPs. In *Proceedings of Advances in Neural Information Processing Systems (NIPS-2003)*, volume 15, pages 1043–1049.

Rudary, M. and Singh, S. (2003). A Nonlinear Predictive State Representation. In *Proceedings of Advances in Neural Information Processing Systems (NIPS-2003)*, Vancouver.

Russel, S. and Norvig, P. (2003). *Artificial Intelligence A Modern Approach*. Pearson Education, Upper Saddle River, New Jersey, second edition.

Sallans, B. (2000). Learning Factored Representations for Partially Observable Markov Decision Processes. In *Proceedings of Advances in Neural Information Processing Systems (NIPS-2000)*, pages 1050–1056, Denver.

Sallans, B. (2002). *Reinforcement Learning for Factored Markov Decision Processes*. PhD thesis, University of Toronto.

Sen, S. and Sekaran, M. (1998). Individual Learning of Coordination Knowledge. *Journal of Experimental and Theoretical Artificial Intelligence*, 10:333–356. (special issue on Learning in Distributed Artificial Intelligence Systems).

Sen, S., Sekaran, M., and Hale, J. (1994). Learning to coordinate without sharing information. In *Proceedings of the National Conference on Artificial Intelligence*, pages 426–431.

Sen, S. and Weiss, G. (2000). Learning in Multiagent Systems. In Weiss, G., editor, *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, chapter 6, pages 259–298. MIT press.

Shehory, O. and Kraus, S. (1998). Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1-2):165–200.

Singh, S., James, M., and Rudary, M. (2004). Predictive State Representations: A New Theory for Modeling Dynamical Systems. In *Proceedings of the Twenty First International Conference on Machine Learning (ICML-2004)*, Banff, Canada.

Singh, S., Littman, M., Jong, N., Pardoe, D., and Stone, P. (2003). Learning Predictive State Representations. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*.

Smallwood, R. D. and Sondik, E. J. (1973). The Optimal Control of Partially Observable Markov Processes over a Finite Horizon. *Operations Research*, 21(5):1071–1088.

Smith, D. E., Frank, J., and Jonsson, A. K. (2000). Coordination of multiple agents in distributed manufacturing scheduling. In *The Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 61–94.

Smith, S. F. (1994). Reactive scheduling systems. *Intelligent Scheduling Systems*, pages 155–192.

Smith, T. and Simmons, R. (2004). Heuristic search value iteration for pomdps. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence(UAI-04)*, Banff, Canada.

Smith, T. and Simmons, R. (2005). Point-based POMDP Algorithms: Improved Analysis and Implementation. In *Proceedings of the 21th Conference on Uncertainty in Artificial Intelligence(UAI-05)*, Edinburgh, Scotland.

Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University.

Sondik, E. J. (1978). The Optimal Control of Partially Observable Markov Processes Over the Infinite Horizon: Discounted Costs. *Operations Research*, 26(2).

Spaan, M. T. J. and Vlassis, N. (2004). A Point-Based POMDP Algorithm for Robot Planning. In *In Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2399–2404, New Orleans, Louisiana.

Spaan, M. T. J. and Vlassis, N. (2005). Perseus: Randomized Point-based Value Iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220.

Stone, P. and Veloso, M. (1999). Team-Partitioned, Opaque-Transition Reinforcement Learning. In Asada, M. and Kitano, H., editors, *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Sugawara, T. and Lesser, V. (1995). Learning Coordination Plans in Distributed Problem-Solving Environments. In *Proceedings of the First International Conference on Multiagent Systems (ICMAS-95)*.

Sugawara, T. and Lesser, V. R. (1998). Learning to Improve Coordinated Actions in Cooperative Distributed Problem-Solving Environments. *Machine Learning*, 33(2-3):129–153.

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3:9–44.

Sutton, R. S. and Barto, A. G. (1999). *Reinforcement Learning*. MIT press.

Tadkoro, S., Kitano, H., Takahashi, T., Noda, I., Matsubara, H., Shinjoh, A., Koto, T., Takeuchi, I., Takahashi, H., Matsuno, F., Hatayama, M., Ohta, M., Tayama, M., Matsui, T., Kaneda, T., Chiba, R., Takeuchi, K., Nobe, J., Noguchi, K., and Kuwata, Y. (2000). The RoboCup-Rescue: an IT challenge to emergency response problem in disaster. In *Industrial Electronics Society, 2000. IECON 2000. 26th Annual Conference of the IEEE*. IEEE.

Takahashi, T., Tadokoro, S., Ohta, M., and Ito, N. (2002). Agent Based Approach in Disaster Rescue Simulation - From Test-Bed of Multiagent System to Practical Application. In Birk, A., Coradeschi, S., and Tadokoro, S., editors, *RoboCup 2001*, volume 2377 of *Lecture Notes in Artificial Intelligence*, pages 102–111. Springer-Verlag.

Tan, M. (1993). Multi-Agent Reinforcement Learning: Independant vs. Cooperative Agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337.

Taylor, M. E. and Stone, P. (2005). Behavior Transfer for Value-Function-Based Reinforcement Learning. In *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2005)*, pages 53–59.

Thrun, S. (2000). Monte Carlo POMDPs. In *Proceedings of Advances in Neural Information Processing Systems (NIPS-2000)*, volume 12, pages 1064–1070.

Tumer, K., Agogino, A. K., and Wolpert, D. H. (2002). Learning Sequences of Actions in Collectives of Autonomous Agents. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, Bologna, Italie.

Uther, W. T. B. and Veloso, M. M. (1998). Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 769–774, Menlo Park, CA. AAAI-Press/MIT-Press.

Varakantham, P., Maheswaran, R., and Tambe, M. (2005). Exploiting Belief Bounds: Practical POMDPs for Personal Assistant Agents. In *In Proceedings of the Fourth International Autonomous Agents and Multiagent Systems Conference (AAMAS-05)*, Utrecht, Netherlands.

Vieira, G. E., Herrmann, J. W., and Lin, E. (2003). Rescheduling manufacturing systems: a framework of strategies, policies, and methods. *Journal of Scheduling*, 6(1):35–58.

Vlassis, N. and Spaan, M. T. J. (2004). A fast point-based algorithm for POMDPs. In *Benelearn 2004: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands*, pages 170–176, Brussels, Belgium. (Also presented at the NIPS-16 workshop 'Planning for the Real-World', Whistler, Canada, Dec 2003).

Washington, R. (1997). BI-POMDP: Bounded, Incremental Partially-Observable Markov-Model Planning. In *Proceedings of the 4th European Conference on Planning*, volume 1348 of *Lecture Notes in Computer Science*, pages 440–451, Toulouse, France. Springer.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

Wolpert, D. and Tumer, K. (2000). An Introduction to Collective Intelligence. Technical Report NASA-ARC-IC-99-63, NASA Ames Research Center.

Xuan, P. and Lesser, V. (2002). Multi-agent policies: From centralized ones to decentralized ones. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1098–1105. ACM Press.

Xuan, P., Lesser, V., and Zilberstein, S. (2001). Communication decisions in multi-agent cooperation: Model and experiments. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 616–623.

Xuan, P., Lesser, V., and Zilberstein, S. (2004). Modeling Cooperative Multiagent Problem Solving as Decentralized Decision Processes. *Autonomous Agents and Multi-Agent Systems*. (under review).

Zhang, N. L. and Liu, W. (1996). Planning in Stochastic Domains: Problem Characteristics and Approximation. Technical Report HKUST-CS96-31, Department of Computer Science, Hong Kong University of Science and Technology.

Zhang, N. L. and Zhang, W. (2001). Speeding Up the Convergence of Value Iteration in Partially Observable Markov Decision Processes. *Journal of Artificial Intelligence Research*, 14:29–51.

Zhou, R. and Hansen, E. A. (2001). An Improved Grid-Based Approximation Algorithm for POMDPs. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 707–716.

Appendix A

Notations

This appendix defines the different symbols used in the equations presented in each chapter of this thesis.

A.1 Notations for Chapter 2

- nA: the number of living agents.
- H: the remaining number of health points (HP) of all agents.
- H_{ini}: the total number of HP of all agents at the beginning.
- B_{ini}: total buildings' area at the beginning of the simulation.
- B: the undestroyed buildings' area.

A.2 Notations for Chapter 3

- S: the set of all the environment states.
- A: the set of all possible actions.
- $T(s, a, s')$: the transition function, which gives the probability of ending in state s' if the agent performs action a in state s, $Pr(s'|s, a)$.
- $R(s, a)$: the reward function which gives the reward associated with doing action a in state s.
- γ: the discount factor ($0 < \gamma \leq 1$).

- Ω: the set of all possible observations.
- $O(s', a, o)$: the observation function which gives the probability of observing o if action a is performed and the resulting state is s', $Pr(o|a, s')$.
- h_t: the agent's history at time t.
- a_t: the action made by the agent at time t.
- o_t: the observation perceived by the agent at time t.
- s_t: the state of the agent at time t.
- $b_t(s)$: the belief state at time t for the state s. In other words, the probability of being in state s according to belief state b_t.
- π: the agent's policy (π^* is the optimal policy).
- $\pi(b)$: the action prescribed by the policy π in the belief state b.
- Γ: the set of all possible policies.
- $V(b)$: the value function, which returns the expected reward if the agent is in belief state b.
- $\tau(b, a, o)$: the belief update function, which returns the new belief state if the agent performs action a in belief state b and perceives o.
- $Q(a, b)$: the expected reward of taking action a in the belief state b.
- $R_B(b, a)$: the reward for being in belief state b and doing action a.
- $\omega(b)$: the number of possible states for a specified belief state b.
- M: the number of variables describing a state.
- $\alpha(a, b, o)$: returns all reachable states considering: the current belief state b, the action performed a and the observation perceived o.
- D: the maximal depth of the online search.
- $\delta(b, d)$: returns an estimation of the value of being in belief state b by performing a search of depth d.
- $U(b)$: the estimated utility of belief state b.
- $\pi(b, D)$: the agent's online policy. Returns the action that has the best expected value evaluated with a search of depth D.

- $RTBSS(b, d)$: the estimated value of belief state b returned by the algorithm RTBSS with a search of depth d.
- $R_{\max}$ and $R_{\min}$: the maximum and the minimum possible rewards respectively for a given problem.
- $V_{\max}$ and $V_{\min}$: the maximum and the minimum expected rewards respectively for a given problem.

A.3 Notations for Chapter 4

- D: the set of all possible task descriptions.
- N: the number of available agents.
- R: the reward function that gives the reward if a task is accomplished.
- T: the transition function that gives the probability to go from one task description to another.
- i_t: an instance recorded at time t.
- d_t: the task description at time t ($d_t \in D$).
- n_t: the number of agents that tried the task d_t at time t.
- r_t: the reward obtained at time t.
- l: a leaf of the search tree (represents a task description).
- $Q(l)$: the expected reward if the agent tries to accomplish a task belonging to the leaf l.
- $\hat{R}(l)$: the estimated immediate reward if a task that belongs to the leaf l is chosen.
- $\hat{T}(l, l')$: the estimated probability that the next instance would be stored in leaf l' given that the current instance is stored in leaf l.
- I_l: the set of all instances stored in leaf l.
- nl: the number of leaf nodes in the tree.
- $sd(I_l)$: the standard deviation on the instances' expected rewards stored in leaf l.
- I_k: the subset of instances in I_l that have the k^{th} outcome for the potential test.

A.4 Notations for Chapter 5

- α: the machines' environment.
- P_m: multi-machines environment where m represents the number of machines.
- β: the constraints and the characteristics of the problem.
- p_j: the execution cost of the task j.
- d_j: the deadline of the task j.
- s_{jk}: the cost to change from task j to task k.
- γ: the optimization criterion.
- $\sum C_j$: the sum of completion times of all tasks.
- $\sum U_j$: the sum of unit penalty of all tasks.
- x: an instance for the KNN algorithm.
- $a_i(x)$: the value of the i^{th} attribute of the instance x.
- n: the number of attributes per instance.
- $d(x_i, x_j)$: the distance between two instances x_i and x_j.
- $\hat{f}(x_q)$: the estimated class of the instance x_q.
- V: the set of all possible classes for the classification function.
- C: the set of available attribute values in the instance to classify.