Parallelisation of hybrid metaheuristics for COP solving

Mohamed Khalil Labidi

▶ **To cite this version:**

Mohamed Khalil Labidi. Parallelisation of hybrid metaheuristics for COP solving. Other [cs.OH]. Université Paris sciences et lettres; Université de Tunis El-Manar. Faculté des Sciences de Tunis (Tunisie), 2018. English. NNT : 2018PSLED029 . tel-02008495

HAL Id: tel-02008495

https://theses.hal.science/tel-02008495

Submitted on 5 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée dans le cadre d'une cotutelle entre
L'Université de Tunis El-Manar – Faculté des Sciences de Tunis
et l'Université Paris-Dauphine

Parallelization of hybrid metaheuristics for COP solving

Ecole doctorale n°543

Spécialité Informatique

COMPOSITION DU JURY :

M. MAHJOUB Ali Ridha
Université Paris-Dauphine,
Co-Directeur de thèse

M. MAHJOUB Zaher
Faculté des Sciences de Tunis,
Co-directeur de thèse

M. YASSINE Adnan
Institut Supérieur d'Etudes Logistiques,
Rapporteur

M. LARDEUX Frédéric
Faculté des Sciences – Université d'Angers,
Rapporteur

M. DIARRASSOUBA Ibrahima
Université du Havre,
Examinateur

M. NAANAA Wady
Ecole Nationale d'Ingénieurs de Tunis,
Examinateur

M. TRYSTRAM Denis
Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées,
Président du Jury

Soutenue le 20 09 2018
par Mohamed Khalil LABIDI

Dirigée par **Ali Ridha MAHJOUB**
Zaher MAHJOUB

Je dédie ce modeste travail
À la mémoire de mon grand-père Mohamed,
À la mémoire de ma grand-mère Khadija,
À la mémoire de mon oncle Taoufik,
À la mémoire de mon ami et collègue Malek,
À ma chère soeur Myriam et son époux Khaled,
À mon cher frère Ahmed et son épouse Sarah,
À mes petits neuveux Mohamed Yassine et Amine,
À mes tantes Hedia, Olfa, Monia, Sondess et Wahida,
À mes oncles Yassine, Karim et Moncef,
À mes cousins et cousines Omar, Baligh, Salim, Houssem,
l'adorable Wissem, Merwan, Sammy, Ilyes, Sirine et Amir,
À mes frères et soeurs Omar, Nader, Bilel, Mohammed, Iaad & Meriem,
Alex, Sophiane, Yousuf, Imrane, Mika, Kadjou, Ilyes, Mehdi,
Steven, Mansour, Ayoub, Samir, Nicolas, Imen et Hela,
À mon cher ami Robert Jouanique et sa famille,
Et surtout à mon adorable grand-mère BiBiBi ainsi que mes
très chers parents Slah et Wassila, puissiez vous y trouver
le fruit d'un peu de vos efforts et sacrifices.
Je vous aime.

Remerciements

Je voudrais en premier lieu remercier tout particulièrement Monsieur A. Ridha Mahjoub, Professeur à l'Université Paris-Dauphine, de m'avoir apporté son soutien, scientifique et moral, tout au long de ce travail. Il a su me transmettre les connaissances, la rigueur, la motivation et la passion pour la recherche en général et l'optimisation combinatoire en particulier. Il m'a également permis de surmonter les moments difficiles grâce à son très grand intérêt pour mon travail et sa constante disponibilité.

Je tiens aussi à remercier très sincèrement Monsieur Zaher Mahjoub, Professeur à la Faculté des Sciences de Tunis, pour ses idées et sa détermination qui m'ont permis d'élargir mon domaine de connaissances et le travail réalisé durant cette thèse. Il m'a également appris, grâce à sa haute exigence scientifique et à sa très grande estime de mes capacités, à ne jamais renoncer et à donner le meilleur de moi-même.

Je remercie vivement Monsieur Ibrahima Diarrassouba, maître de conférences à l'université du Havre, pour toute l'aide qu'il m'a apportée durant cette thèse. Cette aide se situe bien sûr au niveau scientifique, grâce à nos collaborations et aux nombreuses discussions que nous avons eues sur mon travail et à ses conseils et idées.

Je remercie également Madame Anissa Omran, maître assistante à la Faculté des Sciences de Tunis, de m'avoir fait confiance en me proposant ce sujet de thèse. Aussi pour avoir toujours cru en moi et en mes capacités de réussir mes travaux de recherche.

J'adresse tous mes remerciements à Monsieur Adnan Yassine, Professeur des universités à l'Institut Supérieur d'Etudes Logistiques, ainsi qu'à Monsieur Frédéric Lardeux, maître de conférences à la Faculté des Sciences d'Angers, de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse.

J'exprime ma gratitude à Monsieur Denis Trystram, Professeur à l'Institut polytechnique de Grenoble, et à Monsieur Wady Naanaa, Professeur à l'Ecole Nationale d'Ingénieurs de Tunis, qui ont bien voulu examiner ma thèse.

Je tiens également à remercier très chaleureusement mes différents collègues et amis du LAMSADE qui ont partagé avec moi ces dernières années. Les discussions animées et les moments de détente partagés, leur bonne humeur, leurs encouragements et leur amitié, mais aussi les échanges scientifiques et les travaux effectués ensemble, ont grandement contribué à mon épanouissement scientifique et personnel lors de ces années de thèse. Je tiens à remercier Youcef, Mohamed Lamine, Fabien, Mohammad Amin, Mahdi, Thomas, Raja Haddad, Céline, Hossein, Boris, Ioannis, Meriem, Pedro, Yassine, Ons, Manel, Anaëlle, Diana, Hiba, Ian-Christopher, Justin, Marcel, Raja, Hmida, Oussama et surtout notre cher Olivier, avec qui j'ai partagé des moments inoubliables à Paris-Dauphine et ailleurs.

Un grand merci aussi à Juliette De Roquefeuil, Stéphanie Salon et Igor Bratusek pour leur patience et leur précieuse aide malgré les complications de mes démarches administratives.

J'exprime ma reconnaissance à Robert Jouanique et Mehrzia Ben Ismail ainsi que leurs familles respectives de m'avoir acceulli, aidé et soutenu pendant mes premières années de thèse. Je tiens également à remercier les familles EMFienne et AEMDienne pour tous les moments de joie et d'humanité qu'elles m'ont fait vivre en leurs compagnies.

Enfin, mes remerciements ne seraient pas complets si je ne remerciais pas ma famille et mes amis, qui m'ont supporté et appuyé toutes ces années et se sont réellement intéressés à mon travail et au domaine hautement incompréhensible de l'optimisation combinatoire en général.

Abstract

Combinatorial Optimization (CO) is an area of research that is in a constant progress. Solving a Combinatorial Optimization Problem (COP) consists essentially in finding the best solution (s) in a set of feasible solutions called a search space that is usually exponential in cardinality in the size of the problem.

To solve COPs, several methods have been proposed in the literature. A distinction is made mainly between exact methods and approximation methods.

Since it is not possible to aim for an exact resolution of NP-Complete problems when the size of the problem exceeds a certain threshold, researchers have increasingly used Hybrid (HA) or parallel computing algorithms in recent decades.

In this thesis we consider the COP class of Survivability Network Design Problems. We present an approximation parallel hybrid algorithm based on a greedy algorithm, a Lagrangian relaxation algorithm and a genetic algorithm which produces both lower and upper bounds for flow-based formulations.

In order to validate the proposed approach, a series of experiments is carried out on several applications: the k-Edge-Connected Hop-Constrained Network Design Problem (kHNDP) when $L = 2, 3$, The problem of the Steiner k-Edge-Connected Network Design Problem (SkESNDP) and then, two more general problems namely the kHNDP when $L \geq 2$ and the k-Edge-Connected Network Design Problem (kESNDP). The experimental study of the parallelisation is presented after that.

In the last part of this work, we present two parallel exact algorithms: a distributed Branch-and-Bound and a distributed Branch-and-Cut. A series of experiments has been made on a cluster of 128 processors and interesting speedups has been reached in kHNDP resolution when $k = 3$ and $L = 3$.

Key words : Branch-and-Bound, Branch-and-Cut, Combinatorial optimization, hybridization, metaheuristic, network design, parallel computing.

Résumé

L'Optimisation Combinatoire (OC) est un domaine de recherche qui est en perpétuel changement. Résoudre un problème d'optimisation combinatoire (POC) consiste essentiellement à trouver la ou les meilleures solutions dans un ensemble des solutions réalisables appelé espace de recherche qui est généralement de cardinalité exponentielle en la taille du problème.

Pour résoudre des POC, plusieurs méthodes ont été proposées dans la littérature. On distingue principalement les méthodes exactes et les méthodes d'approximation.

Ne pouvant pas viser une résolution exacte de problèmes NP-Complets lorsque la taille du problème dépasse une certain seuil, les chercheurs on eu de plus en plus recours, depuis quelques décennies, aux algorithmes dits hybrides (AH) ou encore au calcul parallèle.

Dans cette thèse, nous considérons la classe POC des problèmes de conception d'un réseau fiable. Nous présentons un algorithme hybride parallèle d'approximation basé sur un algorithme glouton, un algorithme de relaxation Lagrangienne et un algorithme génétique, qui produit des bornes inférieure et supérieure pour les formulations à base de flots.

Afin de valider l'approche proposée, une série d'expérimentations est menée sur plusieurs applications: le problème de conception d'un réseau k-arête-connexe avec contrainte de borne (kHNDP) avec $L = 2, 3$, le problème de conception d'un réseau fiable Steiner k-arête-connexe (SkESNDP) et ensuite deux problèmes plus généraux, à savoir le kHNDP avec $L = 2$ et le problème de conception d'un réseau fi able k-arête connexe kESNDP). L'étude expérimentale de la parallélisation est présentée par la suite.

Dans la dernière partie de ce travail, nous présentons deux algorithmes parallèles exacts: un Branch-and-Bound distribué et un Branch-and-Cut distribué. Une série d'expérimentations a été menée sur une grappe de 128 processeurs, et des accélérations

intéressantes ont été atteintes pour la résolution du problèmes kHNDP avec $k = 3$ et $L = 3$.

Mots clés : Algorithme de Branch-and-bound, algorithme de Branch-and-cut, calcul parallèle, conception de réseau, hybridation, métaheuristique, optimisation combinatoire.

Parallélisation de métaheuristiques hybrides pour la résolution de POC

Introduction

L'optimisation combinatoire (OC) est un domaine de recherche en mathématiques appliquées et en informatique qui a fait d'énormes progrès au cours des dernières décennies. Cette branche de l'optimisation est également étroitement liée à la programmation mathématique linéaire et non linéaire, à la recherche opérationnelle, aux algorithmes et à la théorie de la complexité.

Dans ce travail, nous étudions une classe de POC bien connue des problèmes de conception de réseaux fiables (SNDP).

Les SNDPs sont des problèmes de conception de réseaux qui visent à concevoir des réseaux qui continuent de fonctionner même en cas de défaillance/pannes. En effet, les réseaux sont aujourd'hui d'une importance cruciale vu la place que ces derniers occupent dans de nombreux (tous?) domaines (télécommunications, logistique, économie, informatique, etc.).

Dans un premier temps, nous présentons une étude dans laquelle nous visons la résolution approximative des SNDPs. Ainsi, nous présentons une approche hybride et parallèle basé sur un algorithme glouton, un algorithme de relaxation Lagrangien et un algorithme génétique qui produit à la fois des bornes inférieures et supérieures pour des formulations à base de flots. Afin de valider l'approche proposée, une série d'expériences est réalisée sur deux applications: le problème de conception de réseau k-arête-connexe avec contrainte de borne (kHNDP) et le problème de conception de réseau fiable k-arête-connexe (kESNDP).

Dans un second temps, nous étudions les SNDPs du point de vue de la résolution exacte. Nous utilisons le calcul parallèle pour proposer deux algorithmes exactes efficaces: un algorithme Branch-and-Bound et un algorithme Branch-and-Cut distribués. Une étude expérimentale est aussi présentée pour valider ces deux algorithmes. Le problème sélectionné pour cette phase est le problème de conception de réseau k-arête-connexe avec contrainte de borne (kHNDP) quand $L = 2, 3$.

Notions de base

kESNDP

Soit $G = (V, E)$ un graphe non orienté, $D \subseteq V \times V$ un ensemble de demandes, et une fonction coût $\omega : E \to \mathbb{R}$ qui associe un coût ω_{uv} à chaque arête $uv \in E$ de G. Le problème de conception d'un réseau fiable k-arête-connexe (kESNDP) consiste à trouver un sous-graphe de G de coût minimum tel qu'il existe k st-chemins arête-disjoints qui relient les terminaux de chaque demande $\{s, t\}$ de D.

Le kESNDP peut être formulé comme un programme linéaire en nombre entiers basé sur les flots (voir [193]).

Le kESNDP est équivalent à:

$$\min \sum_{uv \in E} \omega_{uv} x_{uv}$$

s.t.

$$\sum_{v \in V \setminus \{u\}} f^{st}_{uv} - \sum_{l \in V \setminus \{u\}} f^{st}_{lu} = \left\{ \begin{array}{rll} k, & if & u = s, \\ -k, & if & u = t, \\ 0, & if & u \in V \setminus \{s, t\}, \end{array} \right\}$$

$$\text{pour tout } u \in V \text{ and } \{s, t\} \in D, \tag{1}$$

$$\left. \begin{array}{l} f^{st}_{uv} \\ f^{st}_{vu} \end{array} \right\} \leq x_{uv}, \quad \text{pour tout } uv \in E \text{ and } \{s, t\} \in D, \tag{2}$$

$$f^{st}_{uv}, f^{st}_{vu} \geq 0, \quad \text{pour tout } uv \in E \text{ and } \{s, t\} \in D, \tag{3}$$

$$x_{uv} \leq 1, \quad \text{pour tout } uv \in E, \tag{4}$$

$$x_{uv} \in \{0, 1\}, \quad \text{pour tout } uv \in E, \tag{5}$$

$$f^{st}_{uv}, f^{st}_{vu} \in \{0, 1\}, \text{ pour tout } uv \in E, \ \{s, t\} \in D. \tag{6}$$

kHNDP

Soit $G = (V, E)$ un graphe non orienté, $D \subseteq V \times V$ un ensemble de demandes, et une fonction coût $c : E \to \mathbb{R}$ qui associe un coût $c(e)$ à chaque arête $e \in E$ de G. Le problème de conception d'un réseau fiable k-arête-connexe avec contrainte de borne (kHNDP) consiste à trouver un sous-graphe de G de coût minimum tel qu'il existe k L-st-chemins arête-disjoints qui relient les terminaux de chaque demande $\{s, t\}$ de D et de longueur (nombre de sauts) maximum L fixée.

Le problème de kHNDP peut être formulé dans le cas où $L = 3$ comme suit:

$$
\begin{aligned}
min \quad & \sum_{e \in E} c(e)x(e) \\
\sum_{a \in \delta^+(u)} f_a^{st} - \sum_{a \in \delta^-(u)} f_a^{st} &= \left\{ \begin{array}{lll} k & if & u = s \\ 0 & if & u \in \widetilde{V}_{st} \backslash \{s, t\} \\ -k & if & u = t \end{array} \right\}, \qquad \text{pour tout } u \in \widetilde{V}_{st} \\
f_a^{st} &\leq x(e), \quad \text{pour tout } a \in \widetilde{A}_{st}(e),\ \{s, t\} \in D,\ e \in E \\
f_a^{st} &\leq 1, \qquad \text{pour tout } a = (u, u'),\ u \in V \backslash \{s, t\},\ \{s, t\} \in D \\
f_a^{st} &\geq 0, \qquad \text{pour tout } a \in \widetilde{A}_{st}(e),\ \{s, t\} \in D \\
x(e) &\leq 1, \qquad \text{pour tout } e \in E \\
x \in \mathbb{Z}_+^E \qquad & f^{st} \in \mathbb{Z}_+^{\widetilde{A}_{st}}
\end{aligned}
\tag{7}
$$

La *formulation par flots séparés* (7) a été introduite par Diarrassouba et al. en 2013 [96] et met en œuvre un nombre polynomial de contraintes et de variables. Ceci est possible en utilisant les flots dans $|D|$ graphes en couche calculés à l'aide d'une transformation appliquée au graphe G pour chaque demande $\{s, t\}$ (Gouveia et Requejo [139]).

Résolution approchée

Dans un premier temps, nous avons essayé de tirer profit de la structure en blocs de la formulation (voir Figure 1). Nous présentons une approche parallèle basée sur la relaxation lagrangienne [154, 237], une heuristique dont l'idée de base est inspirée de la structure de la formulation du problème, ainsi qu'un algorithme génétique. Nous présentons aussi une comparaison des résultats de notre approche, avec ceux de $CPLEX$.

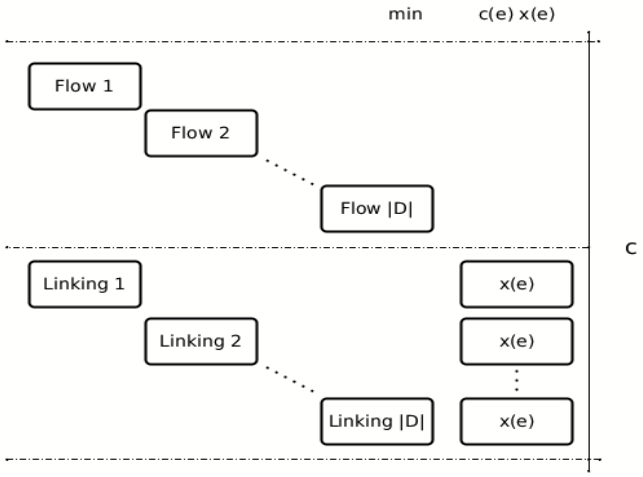

Figure 1: Dépendances des variables

Algorithme de la relaxation Lagrangienne

Dans notre approche, nous considérons les formulations à base de flots présentées au-dessus et relâchons les contraintes de couplage. Soit λ_{uv}^{st}, pour tout $\{s,t\} \in D$ et tout $(u,v) \in A$, soit le multiplicateur Lagrangien associé aux contraintes. Cela donne les problèmes suivant:

kESNDP

$$\min \sum_{uv \in E} \left[\omega_{uv} - \left(\sum_{\{s,t\} \in D} (\lambda_{uv}^{st} + \lambda_{vu}^{st}) \right) \right] x_{uv} + \sum_{\{s,t\} \in D} \sum_{uv \in E} (\lambda_{uv}^{st} f_{uv}^{st} + \lambda_{vu}^{st} f_{vu}^{st})$$

$$\sum_{v \in V \setminus \{u\}} f_{uv}^{st} - \sum_{l \in V \setminus \{u\}} f_{lu}^{st} = \left\{ \begin{array}{rll} k, & if & u = s, \\ -k, & if & u = t, \\ 0, & if & u \in V \setminus \{s,t\}, \end{array} \right\} \text{pour tout } u \in V \text{ and } \{s,t\} \in D,$$

$$\begin{array}{rl} 0 \leq f_{uv}^{st}, f_{vu}^{st} \leq 1, & \text{pour tout } uv \in E, \{s,t\} \in D, \\ x_{uv} \leq 1, & \text{pour tout } uv \in E, \\ x_{uv} \in \{0,1\}, & \text{pour tout } uv \in E, \\ f_{uv}^{st} \in \{0,1\}, & \text{pour tout } uv \in E, \{s,t\} \in D. \end{array}$$

kHNDP

$$min \quad \sum_{e\in E}\left[c(e) - \left(\sum_{\{s,t\}\in D}\sum_{a\in\widetilde{A}_{st}(e)}\lambda_a^{st}\right)\right]x(e) + \sum_{\{s,t\}\in D}\sum_{a\in\widetilde{A}_{st}(e)}\lambda_a^{st}f_a^{st}$$

$$\sum_{a\in\delta^+(u)} f_a^{st} - \sum_{a\in\delta^-(u)} f_a^{st} = \left\{\begin{array}{lll} k & if & u = s \\ 0 & if & u \in \widetilde{V}_{st}\backslash\{s,t\} \\ -k & if & u = t \end{array}\right\}, \text{pour tout } u \in \widetilde{V}_{st}$$

$$\begin{array}{rcll} f_a^{st} & \leq & 1, & \text{pour tout } a = (u,u'),\ u \in V\backslash\{s,t\},\ \{s,t\} \in D \\ f_a^{st} & \geq & 0, & \text{pour tout } a \in \widetilde{A}_{st}(e),\ \{s,t\} \in D \\ x(e) & \leq & 1, & \text{pour tout } e \in E \\ x & \in & \mathbb{Z}_+^E & \\ f^{st} & \in & \mathbb{Z}_+^{\widetilde{A}_{st}} & \end{array}$$

Sur la base des programmes obtenus, deux algorithmes (*RLA* et *RLASH*) ont été conçus pour le kESNDP et le kHNDP. RLA et RLASH commencent par résoudre les $|D|$ problèmes de flot d'une manière parallèle en utilisant l'algorithme "*Network Simplex*" [11, 77].

Pour fixer les multiplicateurs de Lagrange, nous utilisons l'algorithme du sous-gradient. A remarquer que durant cette phase, dans les deux algorithmes conçus, nous utilisons une heuristique dans le cas où le vecteur X_j ne constitue pas une solution réalisable pour les formulations initiales, pour le transformer en $\overline{X_j}$ réalisable. Dans RLA, nous utilisons une heuristique primale rapide pour arrondir à 1 chaque $x_j(e)$ qui viole une contrainte de couplage. Dans RLASH, au lieu d'une heuristique primale, nous utilisons l'heuristique SH que nous avons conçue et que nous allons expliquer par la section suivante.

Heuristique Gloutonne

L'idée principale de l'algorithme *Greedy Successive Heuristic* (*SH*) est inspirée par la structure des formulations à base de flots. En fait, si un arc dans $\widetilde{G}_{st}$ est sélectionné dans la solution du problème de flot $d_i = \{s,t\}$ (i.e. f_a^{st} mis à 1), l'arête correspondante dans G sera sélectionnée dans la solution (par conséquent $x(e) = 1$) en raison des contraintes de couplage. Ainsi, le meilleur choix à faire lors de la résolution du problème de flot suivant est de sélectionner des arcs dont les arêtes correspondantes

sont déjà sélectionnées et qui les rendent libres à être choisies pour la fonction objective principale. En d'autres termes, l'idée est plutôt de résoudre séparément les problèmes de flot maximum du réseau pour chaque $\{s,t\}$ et après avoir superposé les solutions pour construire une solution réalisable, nous résolvons chaque problème de flot i ayant pris en compte les solutions obtenues pour les $\{0...i-1\}$ problèmes de flot déjà résolus. Le but de ce mouvement est d'augmenter la corrélation entre les problèmes de flot $\{s,t\}$.

D'un point de vue pratique, avant de résoudre tout problème de flot $\{s,t\}$, nous commençons par mettre le coût de tous les arcs qui appartiennent à la solution partielle déjà calculée à 0 en superposant les $\{s,t\}$ solutions de flots déjà calculées.

Algorithme génétique

Population

La population d'entrée de l'Algorithme Génétique (GA) que nous choisissons pour notre algorithme hybride est composée des solutions réalisables (individus) des problèmes kHNDP et kESNDP calculées par les algorithmes SH et RLA présentés auparavant. En effet, dans RLA, et plus précisément dans la phase du sous-gradient, l'algorithme produit à chaque itération i, une solution réalisable pour le problème traité (les différents X_j).

Chaque individu de la population est codé en un vecteur repsrésentant des valeurs x_j, un ensemble de $|E|$ entiers indiquant si oui ou non (0 ou 1) une arête est sélectionnée dans la solution. Un chromosome est lui défini comme un ensemble de $|D|$ valeurs de flot (gènes) relatives au $\widetilde{G}_{st}$.

Evaluation & Selection

L'évaluation des individus est faite en fonction de leur valeur de fonction objective Z. Un "classement social" de la population est fait en se basant sur ce phénotype.

Pour chaque itération de GA, un tri fusion parallèle est appliqué sur le pool de population pour trier les individus en fonction de leur classement social.

La sélection, par la suite, est faite aléatoirement mais selon une distribution de probabiliste. Nous commençons par diviser la population en cinq catégories (A, B,

C, D et E) en fonction de leur classement social (valeur de la solution), puis nous sélectionnons $\frac{1}{10}$ de la population à reproduire sachant que la probabilité qu'un parent soit choisi est égale à 67%, et 19%, 10%, 3% et 1% respectivement à la catégorie sociale A, B, C, D et E.

Crossover & Reproduction

Le croisement est généralement un opérateur binaire qui combine deux individus sélectionnés (parents) afin de produire probablement une meilleure progéniture.

En tant que première version de notre GA, nous avons choisi de baser notre schéma de reproduction sur le croisement à deux points. Deux points choisis au hasard dans la permutation sont utilisés comme points de coupe. Chaque parent est utilisé une fois en tant que premier parent et une fois en tant que deuxième parent. Deux descendants sont formés en permutant les deux parents. Les composantes entre les deux points de coupure dans la permutation sont héritées du premier parent et les valeurs restantes sont remplies du second parent.

De plus, nous assignons à chaque paire de parents sélectionnés une mesure prédictive de la qualité possible des enfants qui seront produits. Il est évident que si les deux parents appartiennent à la classe sociale A, ces parents sont plus à même de produire des enfants de bonne qualité que d'autres. Ainsi, selon la valeur de l'alchimie, nous produisons plus de solutions (enfants) en fonction de différents croisements à deux points choisis aléatoirement.

Hybridation

Dans notre algorithme général PHA, les trois algorithmes RLA, GA et SH fonctionnent en parallèle. L'hybridation des trois algorithmes est faite de de la manière suivante: Tout d'abord, les solutions générées par les algorithmes SH et RLA sont introduites dans le pool de solutions (population) de GA. De plus, au début de PHA, la limite supérieure globale Z_{UB} est définie sur ∞. Ensuite, chaque fois que RLA, SH et GA génèrent une solution réalisable, la valeur Z de cette solution est comparée à Z_{UB}. Si $Z < Z_{UB}$, alors la meilleure limite supérieure Z_{UB} est mise à jour avec la valeur de Z. Notez que la valeur Z_{UB} est utilisée par RLA pour mettre à jour les multiplicateurs de Lagrange.

A noter également que notre algorithme veuille à ce que GA continue de fonctionner après la fin de la RLA et de SH afin de donner à GA suffisamment de temps pour traiter les solutions de RLA et SH.

Enfin, notez qu'avec la limite supérieure globale Z_{UB}, l'algorithme de relaxation Lagrangienne produit également une borne inférieure de la solution optimale du problème traité. Ceci est capital vu qu'il nous permet d'avoir une idée sur combien on pourrait améliorer notre borne primale.

La figure 2 ci-dessous décrit le schéma de communication de PHA.

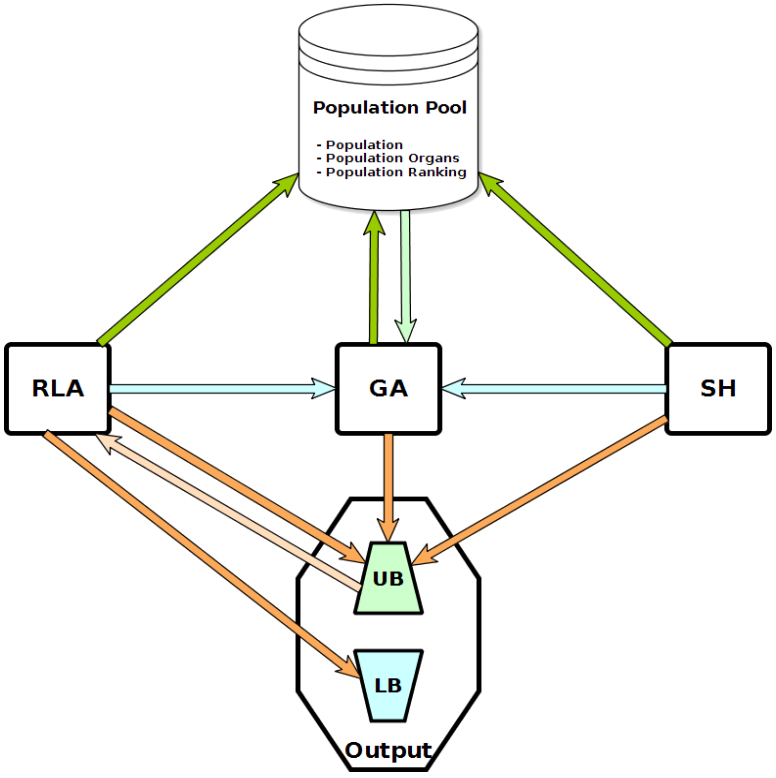

Figure 2: PHA communication scheme

Résolution exacte

Parallel B&B

Il est connu que la structure des arbres de dépendances des Branch-and-Bound *B&B* contient très peu de dépendances, par conséquent l'algorithme est intrinsèquement parallélisable. Néanmoins, les résultats de cette tâche peuvent être très mauvais si les communications ne sont pas traitées efficacement. Cela est d'autant plus vrai lorsque la mémoire est distribuée.

Ainsi, pour éviter d'avoir un surcoût important dû à l'architecture distribuée, le schéma de parallélisation que nous avons conçu suit le modèle maître/esclave. En fait, avoir une gestion centralisée des arbres réduit le nombre de communications à effectuer. Par conséquent, le processus maître est le processus responsable de la gestion de la recherche arborescente, de l'évaluation du nœud racine, de la gestion globale des limites supérieure et inférieure et de la gestion de l'arrêt de l'algorithme. Les processus esclaves sont responsables de l'évaluation des nœuds de l'arbre et du branchement. La figure 3 décrit le schéma de communication de notre algorithme parallèle B&B.

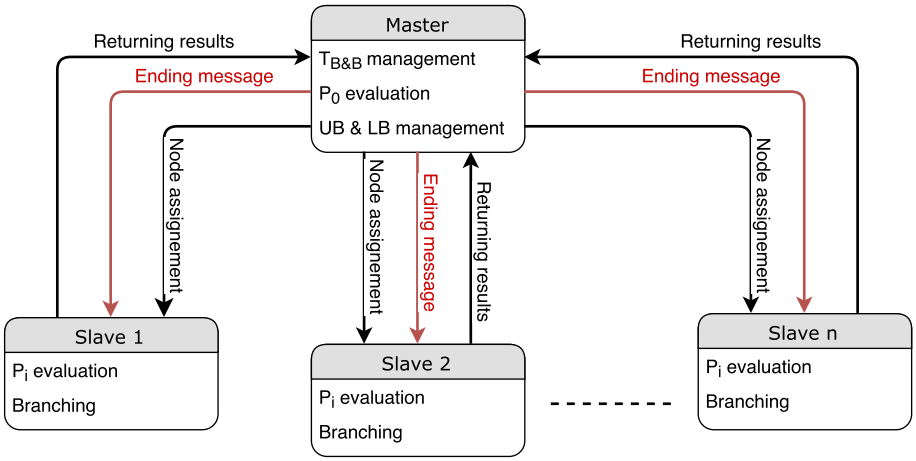

Figure 3: Distributed B&B communication scheme

Ci-dessous, nous présentons les deux algorithmes maitre et esclave qui forment notre

implémentation parallèle du B&B.

Algorithm 1: Processus maitre du B&B parallèle.

Data: Un graphe non orienté $G = (V, E)$, ensemble de demandes D, deux entiers positifs $k \geq 1$ et $L \geq 2$

Result: La solution optimale du problème traité, ou bien deux bornes supérieure et inférieure pour le problème.

```
begin
    Z_UB ← SH();
    data_root ← SOLVE();
    if IS_FRACTIONAL(data_root) then
        PRIMAL_HEURISTIC(data_root);
        P^1_LF, P^{+2}_LF ← BRANCHING(data_root) ;
        INSERT(T_B&B, P^1_LF);
        INSERT(T_B&B, P^2_LF);
    end
    while T_B&B is not empty do
        AFFECT_TASKS(T_B&B);
        RECIEVE_RESULTS(results_in);
        if IS_INTEGER(results_in) then
            if Z(results_in) < Z_UB then
                Z_UB ← Z(results_in);
            end
        else
            if Z(results_in) ≤ Z_UB then
                sol^I_k ← MAKE_INTEGER(results_in);
                if Z(sol^I_k) < Z_UB then
                    Z_UB ← Z(sol^I_k);
                end
                UPDATE(T_B&B, Z_LB, Z(sol_k));
                BRANCH_AND_INSERT(T_B&B, results_in);
            end
        end
        if Z_UB = Z_LB then
            stop;
        end
    end
    return (Z_LB, Z_UB);
end
```

Parallel B&C

Comme pour l'algorithme B&B, le véritable défi dans un algorithme de recherche arborescante parallèle est le choix de quoi, quand et combien de données échanger pendant

Algorithm 2: Processus esclave du B&B parallèle.

Data: Un noeud $T_{B\&B}$, un ensemble de branchements
Result: La solution optimale di noeud, une solution entière, l'indice de la variable de branchement.

```
begin
    while True do
        data_in ← RECIEVE_TASK();
        if IS_END_MSG(data_in) then
            stop;
        end
        DECODE_BRANCHINGS_SET();
        data_out ← SOLVE();
        if IS_FRACTIONAL(data_out) then
            PRIMAL_HEURISTIC(data_out);
            BRANCHING(data_out) ;
        end
        SEND_RESULTS(data_out) ;
    end
end
```

le processus d'optimisation. Cela est d'autant plus vrai dans un algorithme de Branch-and-Cut B&C qui évalue un nœud en utilisant une résolution de programme linéaire et des coupes de polyèdre.

Le B&C parallèle que nous avons conçu est basé sur le B&B parallèle que nous avons présenté dans la section précédente. La contribution majeure apportée par l'algorithme B&C est la gestion distribuées des coupes. En fait, une manière triviale de gérer les coupes séparées serait de séparer les différentes familles de contraintes valides dans les processus esclaves, puis de communiquer les contraintes séparées au maître. Cette communication se produirait quand un esclave renvoie les résultats de l'évaluation du nœud. Néanmoins, puisque nous ne sommes pas en mesure d'estimer le coût d'une telle communication a priori, cette méthode peut provoquer un goulot d'étranglement à l'application.

Pour répondre à cette limite, le B&C parallèle que nous avons conçu utilise, au-delà du processus maître, un processus gestionnaire de pool qui se chargera de la gestion des contraintes valides. La figure 4 montre le schéma de communication B&C distribué.

Ci-dessous, nous présentons les trois algorithmes maitre, gestionnaire de pool et esclave qui forment notre implémentation parallèle du B&C.

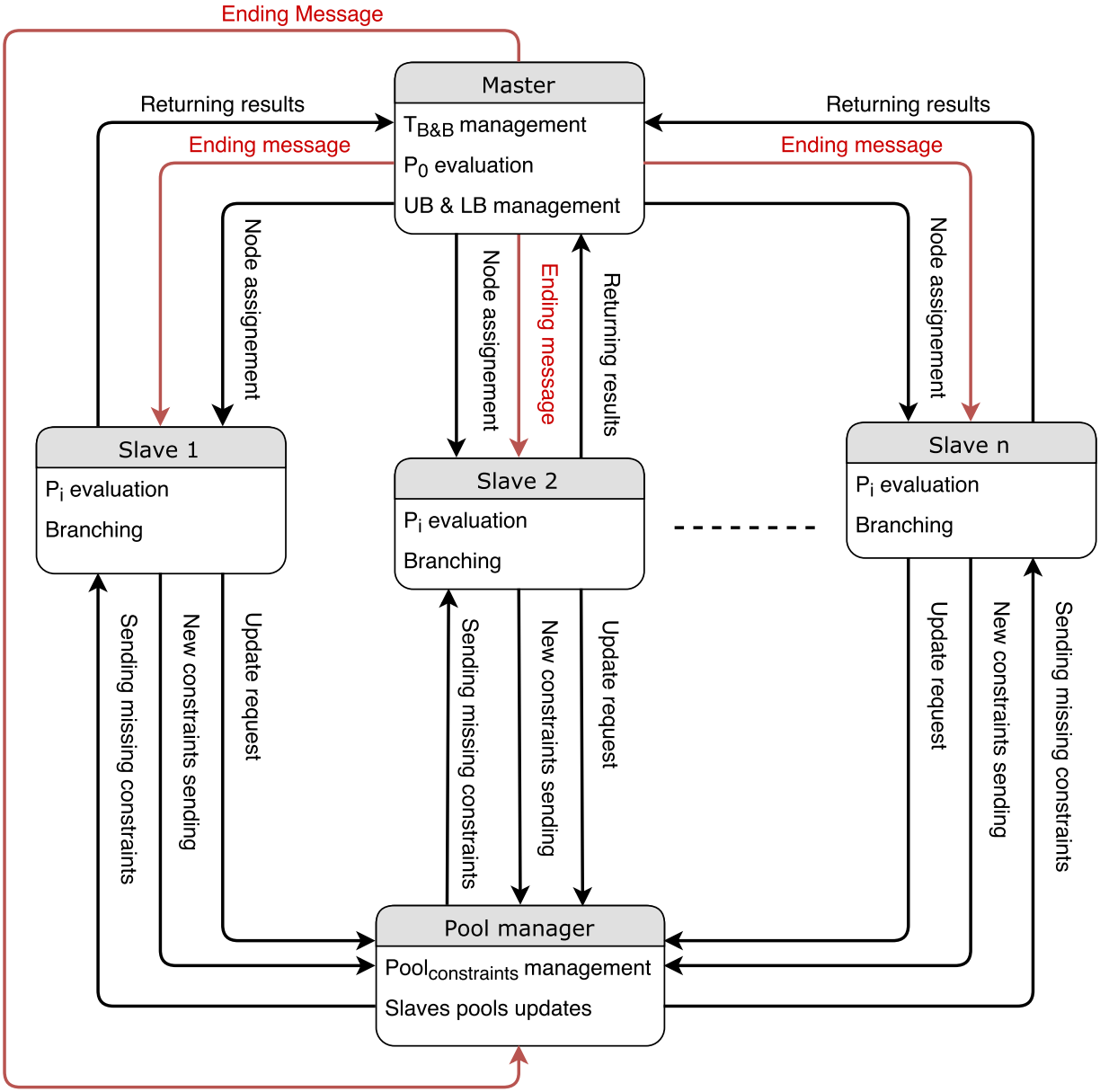

Figure 4: Distributed B&C communication scheme

Conclusion

Nous avons étudié, dans cette thèse, la parallélisation de métaheuristiques hybrides pour des problèmes de conception de réseaux. Dans un premier temps, nous avons proposé deux algorithmes hybrides et parallèles pour résoudre le problème de conception de réseaux Steiner k-arête-connexes (SkESNDP) et le problème de conception de

Algorithm 4: Processus esclave du B&C parallèle.

```
begin
    while True do
        data_in ← RECIEVE_TASK();
        if IS_END_MSG(data_in) then
            stop;
        end
        DECODE_BRANCHINGS_SET();
        NB_cons ← 0;
        repeat
            if NB_cons = 0 then
                UPDATE_POOL(pool^l_cons);
            else
                SEND_CONSTRAINTS(data_out);
                UPDATE_POOL(pool^l_cons);
            end
            data_out, NB_cons ← SOLVE();
        until NB_cons = 0;
        if IS_FRACTIONAL(data_out) then
            PRIMAL_HEURISTIC(data_out);
            BRANCHING(data_out) ;
        end
        SEND_RESULTS(data_out) ;
    end
end
```

réseaux fiables k-arête-connexes avec contraintes de bornes (kHNDP) quand $L = 2, 3$. Nous avons étendu ce travail après cela pour problème de conception de réseaux k-arête-connexes (kESNDP) et le problème de conception de réseaux fiables k-arête-connexes avec contraintes de bornes (kHNDP) lorsque $L \geq 2$. Les algorithmes sont basés sur un algorithme de relaxation lagrangienne, un algorithme génétique et un algorithme glouton, et visent à produire des bornes inférieures et supérieures pour le problème traité. Les expérimentations menées ont montré que notre algorithme hybride surpasse CPLEX en produisant de bonnes solutions réalisables, même pour des instances de grande taille, et ce dans un temps CPU relativement court. Ils ont également montré que l'hybridation des trois composants surpassait, dans la plupart des cas, chaque composant pris séparément. Enfin, nous avons montré que l'utilisation de plusieurs processeurs à l'intérieur de chaque composant de l'algorithme hybride permet de réduire le temps CPU de chaque composant.

Dans le dernier chapitre, nous avons présenté de nouvelles implémentations distribuées des algorithmes connus,Branch-and-Bound et Branch-and-Cut. Pour cela,

Algorithm 5: Processus gestionnaire de pool du B&C parallèle.

```
begin
    while True do
        data_in ← RECEIVE_REQUEST();
        if IS_END_MSG(data_in) then
            stop;
        end
        if IS_CLAIMANT_MSG(data_in) then
            SEND_CONSTRAINTS(data_in, pool_cons);
        end
        if IS_FEEDING_MSG(data_in) then
            SEND_CONSTRAINTS(data_in, pool_cons);
            RECEIVE_CONSTRAINTS(data_in, pool_cons);
        end
    end
end
```

nous avons commencé par présenter un état de l'art des études de parallélisation de ces méthodes. Nous avons également présenté deux algorithmes parallèles basés sur le Branch-and-Bound et le Branch-and-Cut. Enfin, nous avons validé nos algorithmes en présentant une étude expérimentale que nous avons menée sur un cluster de 128 processeurs pour résoudre le kHNDP lorsque $k = 3$ et $L = 3$. Ces expérimentations ont montré une accélération intéressante obtenue en exécutant nos algorithmes sur les 128 cœurs, et grâce à cette parallélisation, nous avons pu résoudre des instances à grande échelle pour le kHNDP et produire de meilleurs écarts pour des instances ayant $\simeq 500$ sommets et $\simeq 300$ demandes.

Cependant, ce que nous avons pu observer dans toutes nos expérimentations, c'est que les écarts entre les bornes inférieure et supérieure pour les instances à grande échelle étaient assez importants (plus de 40% pour la plupart des instances). Ainsi, il serait intéressant d'étudier plus en profondeur un moyen de produire de meilleures bornes inférieures afin de savoir à quel point les solutions produites par les algorithmes sont proches de la solution optimale.

Enfin, il serait également intéressant d'étudier la parallélisation des algorithmes et l'utilisation d'architectures plus sophistiquées, telles que les GPUs.

Algorithm 3: Processus maitre du B&C parallèle.

Data: Un graphe non orienté $G = (V, E)$, ensemble de demandes D, deux entiers positifs $k \geq 1$ et $L \geq 2$

Result: La solution optimale du problème traité, ou bien deux bornes supérieure et inférieure pour le problème.

```
begin
    Z_UB ← SH();
    NB_cons ← 0;
    repeat
        if NB_cons > 0 then
            SEND_CONSTRAINTS(data_out);
        end
        data_out, NB_cons ← SOLVE();
    until NB_cons = 0;
    if IS_FRACTIONAL(data_root) then
        PRIMAL_HEURISTIC(data_root);
        P^1_LF, P^{+2}_LF ← BRANCHING(data_root) ;
        INSERT(T_B&B, P^1_LF);
        INSERT(T_B&B, P^2_LF);
    end
    while T_B&B is not empty do
        AFFECT_TASKS(T_B&B);
        RECIEVE_RESULTS(results_in);
        if IS_INTEGER(results_in) then
            if Z(results_in) < Z_UB then
                Z_UB ← Z(results_in);
            end
        else
            if Z(results_in) ≤ Z_UB then
                sol^I_k ← MAKE_INTEGER(results_in);
                if Z(sol^I_k) < Z_UB then
                    Z_UB ← Z(sol^I_k);
                end
                UPDATE(T_B&B, Z_LB, Z(sol_k));
                BRANCH_AND_INSERT(T_B&B, results_in);
            end
        end
        if Z_UB = Z_LB then
            stop;
        end
    end
    return (Z_LB, Z_UB);
end
```

Contents

Introduction

Combinatorial Optimization (CO) is a research area in applied mathematics and computer science that has made huge progress over the past few decades. This branch of optimization is also closely linked to linear and nonlinear mathematical programming, operational research, algorithms and complexity theory.

Solving a combinatorial optimization (COP) problem consists essentially in finding the best solution(s) in a set of feasible solutions called the search space that is usually exponential in cardinality in the size of the problem. As it happens, the majority of COPs are NP-complete. Therefore, the determination of an optimal solution (by enumeration of the feasible solutions among others) proves to be an extremely expensive task, even impossible when the size of the problem exceeds a certain threshold.

In fact, to solve COPs, several methods have been proposed in the literature. A distinction is made mainly between exact methods and approximation methods.

In the exact methods we try to achieve an optimal solution by partially enumerating the elements (realizable solutions) of the search space via specific techniques such as Dynamic Programming, Branch-and-Bound, Cutting planes, Branch-and-Cut, etc. These methods are known for their effectiveness but they prove to be unusable in practice beyond a certain size of the treated problem and this for their prohibitive cost.

As an alternative, we are sometimes satisfied with an approximate solution constructed using heuristics, which are of two types: (i) constructive heuristics (often of the Greedy type) which construct feasible solutions based on characteristics of the problem; (ii) more elaborate methods known as heuristics of improvement generally based on metaheuristics whose principle is often borrowed from the modeling of natural phenomena of physics or biology (eg. simulated annealing methods, genetic algorithms, ant colony, etc.).

It should also be noted that hybrid algorithms (HA) are being used more and more

in which several types of methods are combined within the same resolution strategy.

On the other hand, at the hardware level, spectacular progress in recent years has challenged the way to design and especially to exploit the available computing resources. Thus, the parallelization of methods of solving COPs and their implementation on massively parallel platforms is of increasing interest to researchers. Several methods of resolution have been developed for different COPs since the 1980s and new methods are still been developed. It should be emphasized that parallelization thus makes it possible to solve, by polynomial approximation methods, instances of COP of high sizes and even to implement, in certain cases, exact methods of exponential complexity.

In this work we address a well known COP class of Survivable Network Design Problems (SNDP).

SNDPs are those network design problems which aim in designing networks that are still functionning even when failures occur. In fact, efficient networks are nowadays of crucial importance since networks take a large place in many fields (telecommunications, logistics, economics, IT, etc.). Addressing network design issues has raised a large class of problems.

As a first step, we approach the SNDPs from an approximation point of view. Thereby, we present a parallel hybrid algorithm based on a greedy algorithm, a Lagrangian relaxation algorithm and a genetic algorithm which produces both lower and upper bounds for flow-based formulations. In order to validate the approach proposed, a series of experiments is conducted on two applications: the k-Edge-Connected Hop-Constrained Network Design Problem (kHNDP) and the k-Edge-Connected Survivability Network Design Problem (kESNDP).

As a second step, we study the SNDPs from the exact solving point of view. To accomplish this task we use parallel computing to fasten a Branch-and-Bound and a Branch-and-Cut algorithms.

This dissertation is organized as follows:

In Chapter 1, we present basic notions of combinatorial optimization. This chapter also includes a state-of-the-art on metaheuristics and parallel computing.

In Chapter 2, we present the Survivable Network Design Problems in general (SNDP) and a state-of-the-art on the problems that we chose as applications. We present a state-of-the-art on parallel Branch-and-Bound algorithms, parallel Branch-and-Cut

algorithms, the taxonomies of hybridization and introduce after that the motivation of the thesis.

In Chapter 3, we consider the k-Edge-Connected Network Design Problem (kESNDP). We introduce a new approximation parallel and hybrid algorithm to solve flow-based formulations and proceed to an experimental study on two variants of the problem.

In Chapter 4, we study the k-Edge-Connected Hop-Constrained Network Design Problem (kHNDP). We present the application of the hybrid approach presented in the previous chapter on the kHNDP and proceed to its experimental study.

In Chapter 5, we present the parallelization strategies we have applied to the Branch-and-Bound and the Branch-and-Cut algorithms, and proceed to the experimental study related to the kHNDP.

Chapter 1

Preliminaries

Contents

In this Chapter we give some basic notions on combinatorial optimization, graph theory, the theory of complexity. We then perform a brief summary of exact methods, metaheuristics and the basics of parallel computing.

1.1 Combinatorial optimization

Combinatorial optimization is a branch of computer science and applied mathematics. It concerns the problems that can be formulated as follows: Let $E = \{e_1, \ldots, e_n\}$ a finite set called *basic set*, where each element e_i has a *weight* $c(e_i)$. Let $\mathcal{S}$ a family of subsets of E. If $S \in \mathcal{S}$, then $c(S) = \sum_{e_i \in S} c(e_i)$ is the weight of S. The problem is to determine an element of $\mathcal{S}$, with the smaller (or larger) weights. Such a problem is called a *combinatorial optimization problem*. The set $\mathcal{S}$ is called the *set of solutions of the problem*. In other words,

$$\min(\text{or } \max)\{c(S) : S \in \mathcal{S}\}.$$

The term *combinatorial* refers to the discrete structure of $\mathcal{S}$. In general, this structure is represented by a graph. The term *optimization* signifies that we are looking for the best element in the set of feasible solutions. This set generally contains an exponential number of solutions, therefore, one can not expect to solve a combinatorial optimization problem by exhaustively enumerate all its solutions. Such a problem is then considered as a hard problem.

Various effective approaches have been developed to tackle combinatorial optimization problems. Some of these approaches are based on graph theory, while others use linear and non-linear programming, integer programming and polyhedral approach. Besides, several practical problems arising in real life, can be formulated as combinatorial optimization problems. Their applications are in fields as diverse as telecommunications, transport, industrial production planing or staffing and scheduling in airline companies. Over the years, the discipline got thus, enriched by the structural results related to these problems. And, conversely, the progress made in computed science have made combinatorial optimization approaches even more efficient on real-world problems.

In fact, combinatorial optimization is closely related to algorithm theory and computational complexity theory as well. The next Section introduces computational issues of combinatorial optimization.

1.2 Computational complexity

Computational complexity theory is a branch of theoretical computer science and mathematics, whose study started with works of Cook [68], Edmonds [108] and Karp [171].

Its objective is to classify a given problem depending on its difficulty. A plentiful literature can be find on this topic, see for instance [122] for a detailed presentation of NP-completeness theory.

A *problem* is a question having some input parameters, and to which we aim to find an answer. A problem is defined by giving a general description of its parameters, and by listing the properties that must be satisfied by a solution. An *instance* of the problem is obtained by giving a specific value to all its input parameters. An *algorithm* is a sequence of elementary operations that allows to solve the problem for a given instance. The number of input parameters necessary to describe an instance of a problem is the *size* of that problem.

An algorithm is said to be polynomial if the number of elementary operations necessary to solve an instance of size n is bounded by a polynomial function in n. We define the class P as the class gathering all the problems for which there exists some polynomial algorithm for each problem instance. A problem that belongs to the class P is said to be "easy" or "tractable".

A *decision problem* is a problem with a *yes* or *no* answer. Let $\mathcal{P}$ be a decision problem and $\mathcal{I}$ the set of instances such that their answer is yes. $\mathcal{P}$ belongs to the *class NP* (Non-deterministic Polynomial) if there exists a polynomial algorithm allowing to check if the answer is yes for all the instances of $\mathcal{I}$. It is clear that a problem belonging to the class P is also in the class NP. Although the difference between P and NP has not been shown, it is a highly probable conjecture.

In the class NP, we distinguish some problems that may be harder to solve than others. This particular set of problems is called *NP-complete*. To determine whether a problem is NP-complete, we need the notion of *polynomial reducibility*. A decision problem P_1 can be polynomially reduced (or transformed) into another decision problem P_2, if there exists a polynomial function f such that for every instance I of P_1, the answer is "yes" if and only if the answer of $f(I)$ for P_2 is "yes". A problem $\mathcal{P}$ in NP is also NP-complete if every other problem in NP can be reduced into $\mathcal{P}$ in polynomial time. The Satisfiability Problem (SAT) is the first problem that was shown to be NP-complete (see [68]).

With every combinatorial optimization problem is associated a decision problem. Furthermore, each optimization problem whose decision problem is NP-complete is said to be *NP-hard*. Note that most of combinatorial optimization problems are NP-hard. One of the most efficient approaches developed to solve those problems is the so-called polyhedral approach.

1.3 Graph theory

In this Section we will introduce some basic definitions and notations of graph theory that will be used throughout the Chapters of this dissertation. For more details, we refer the reader to [243].

1.3.1 Undirected graphs

An undirected graph is denoted $G = (V, E)$ where V is the set of vertices or nodes and E is the set of edges. If e is an edge between two vertices u and v, then u and v are called the ends of E, and we write $e = uv$ or $e = \{u, v\}$. If u is an extremity of e, then u (resp. e) is said to be incident to e (resp. u). Similarly, two vertices u and v forming an edge are said to be adjacent. Since the graph G may have multiple edges, it may be that $e = uv$ and $f = uv$ but $e \neq f$.

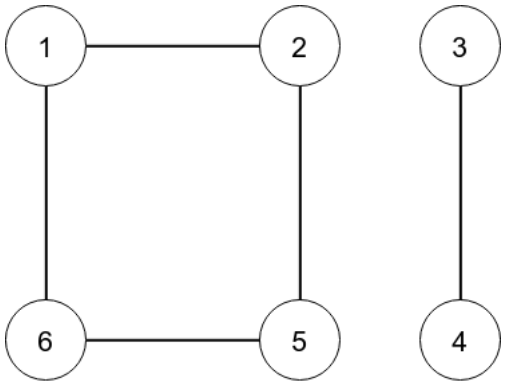

Figure 1.1: An undirected graph G

If $F \subseteq E$ is a subset of edges, then $V(F)$ represents the node set of edges of F. If $W \subseteq V$ is a subset of vertices, then $E(W)$ denotes the set of edges having their two ends in W. Let $V(H)$ and $E(H)$ be the sets U and F, respectively.

A subgraph $H = (U, F)$ of G is a graph such that $U \subseteq V$ and $F \subset E$. A subgraph $H = (U, F)$ of G is called *covering* or *spanning* if $U = V$. Let $W \subseteq V$, $H = (W, E(W))$ is said to be *subgraph* of G induced by W and will be denoted by $G[W]$.

If $F \subset E$ (resp. $W \subset V$), it is noted in $G \setminus F$ (resp. $G \setminus W$) the graph obtained from G by removing the edges of F (resp. nodes of W and the edges incident to W).

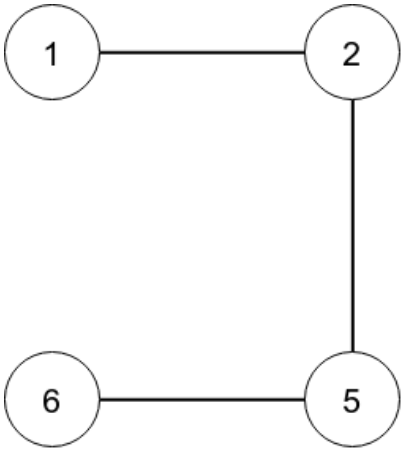

Figure 1.2: Subgraph H_1 of G

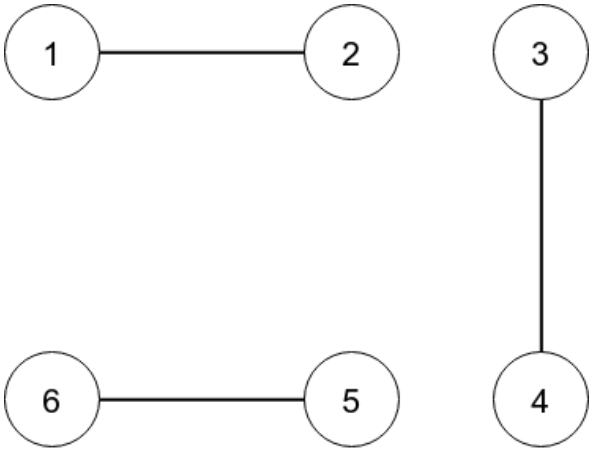

Figure 1.3: Spanning subgraph H_2 of G

If F (resp. W) is reduced to a single edge e (resp. a single vertex v), we write $G \setminus e$ (resp. $G \setminus v$). Let $W \subseteq V$, $\emptyset \neq W \neq V$, a subset of vertices of V. The set of edges having one end in W and the other in $V \setminus W$ is called *cut* and noted $\delta(W)$. By setting $\overline{W} = V \setminus W$, we have that $\delta(W) = \delta(\overline{W})$. If W is reduced to a single vertex v, we write $\delta(v)$. The cardinality of the cut $\delta(W)$ of a subset W is called the degree of W and noted $d(W)$. Given W and W' two disjoint subsets of V, then $[W, W']$ represents the set of edges of G which have one end in W and the other in W'.

An edge $e = v_1v_2 \in E$ is called a *cut edge* if G is connected and $G \setminus e$ is not connected, with $v_1, v_2 \in V$.

If $\{V_1, \ldots, V_p\}, p \geq 2$, is a partition of V, then $\delta(V_1, \ldots, V_p)$ is the set of edges having one end in V_i and the other one in V_j and $i \neq j$.

The *support graph* of an inequality is the graph induced by the vertices of variables having a non-zero coefficient in the inequality.

Let $G = (V \cup T, E)$ be a graph defined by a set of vertices $V \cup T$ where T is a set of distinguished nodes and E is a set of edges. We denote by $V(H)$, $T(H)$ and $E(H)$ its sets of nodes, terminals and edges, respectively. We denote by $t(G)$ the number of terminal in G, *i.e.*, $|T(G)| = t(G)$.

A *path* P is a set of p distinct vertices $v_1, v_2, \ldots, v_p$ such that for all $i \in \{1, \ldots, p-1\}$, v_iv_{i+1} is an edge. P is called *elementary* if it passes more than once by the same node (except for v_0 and v_k if they represent the same vertex in G). A basic chain is totally identified with its set of edges.

Two paths between two nodes u and v are called edge-disjoint (resp. node-disjoint)

if there is no edge (resp. no node different from of u and v) appearing in both chains.

Vertices $v_2, \ldots, v_{p-1}$ are called *the internal vertices* of P. Given a path P between two terminals $t, t' \in T$ such that $P \cap T = \{t, t'\}$, the set of internal vertices of P will be called *a terminal path* and denoted by $P_{tt'}$. A terminal path is *minimal* if it does not strictly contain a terminal path.

Given a graph $G = (V \cup T, E)$ and two subgraphs $G_1 = (V_1 \cup T_1, E_1)$, $G_2 = (V_2 \cup T_2, E_2)$ of G. Graph G_1 is said to be *completely included* in G_2, if $V_1 \cup T_1 \subseteq V_2 \cup T_2$.

1.3.2 Directed graphs

A directed graph is denoted $D = (V, A)$ where V is the set of nodes and A the set of arcs.

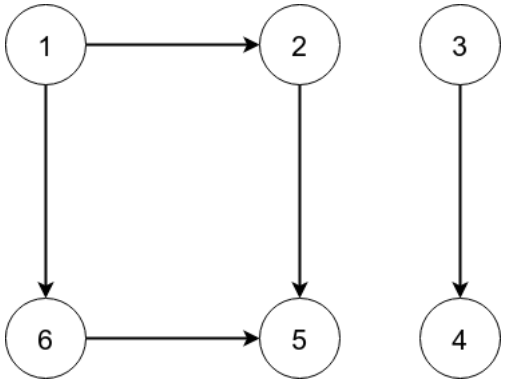

Figure 1.4: A directed graph D

If $a \in A$ is an arc connecting a vertex u to vertex v, then u will be called initial end and v final end and we write $a = (u, v)$. We say that a is an outgoing arc of u and v of an incoming arc. The vertices u and v are called ends of a. Vertex v (resp. a) is said to be incident to a (resp. v) if v is an end (initial or final) of a.

If $B \subseteq A$ is a subset of arcs, then $V(B)$ represents the node set of arcs of B. If $W \subseteq V$ is a subset of vertices, $A(W)$ is the set of arcs having their ends in W.

A subgraph $H = (U, F)$ of D is a graph such that $U \subseteq V$ and $F \subset A$. A subgraph $H = (U, F)$ of D is said covering if $U = V$.

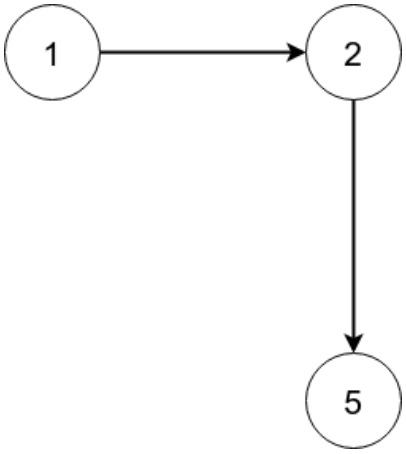

Figure 1.5: Directed Subgraph H_3 of D

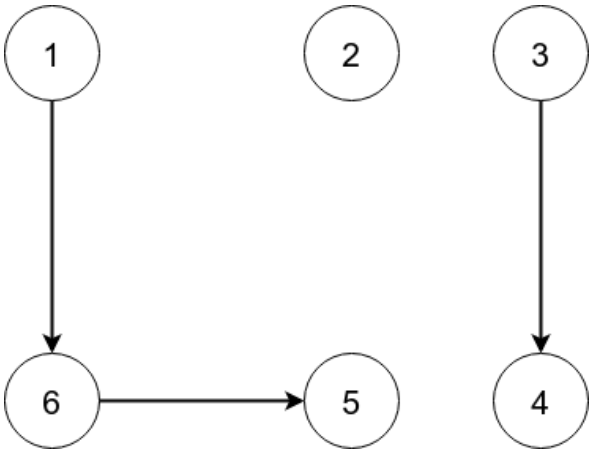

Figure 1.6: Covering directed subgraph H_4 of D

If $F \subset A$ (resp. $W \subset V$), we denote by $D \setminus F$ (resp. $D \setminus W$) the graph obtained from D by removing the F arcs (resp. node of W and edges incident to W). If F (resp. W) is reduced to a single arc a (resp. a single vertex v), we write $D \setminus a$ (resp. $D \setminus v$).

Let $W \subseteq V$, $\emptyset \neq W \neq V$, a subset of vertices V. The set of arcs having their initial end in W and their final nodes in $V \setminus W$ is called outgoing cut and denoted $\delta^+(W)$. The cardinality of the outgoing cut $\delta^+(W)$ of a subset W is called outgoing degree of W and denoted $d^+(W)$. If $u \in W$ and $v \in V \setminus W$, then the outgoing cut is also called uv-outgoing cut. If W is reduced to a single vertex v, we write respectively $\delta^+(v)$ and $d^+(v)$ instead of $\delta^+(\{v\})$ and $d^+(\{v\})$. The set of arcs having the final end in W and the initial end in $V \setminus W$ is called incoming cut and denoted $\delta^-(W)$. The cardinality of the incoming cut $\delta^-(W)$ of a subset W is called incoming degree of W and denoted $d^-(W)$. If $u \in W$ and $v \in V \setminus W$, then the incoming cut is also known as uv-incoming cut. If W is reduced to a single vertex v, we write respectively $\delta^-(v)$ and $d^-(v)$ instead of $\delta^-(\{v\})$ and $d^-(\{v\})$.

The cut of a set $W \subseteq V, \emptyset \neq W \neq V$, is denoted $\delta(W)$ and is the union of the arcs of the incoming cut and outgoing cut, i.e., $\delta(W) = \delta^+(W) \cup \delta^-(W)$. The cardinality of the cut is called the degree of W and denoted $d(W)$. If $u \in W$ and $v \in V \setminus W$, then the cut is also called uv-cut. If W is reduced to a single vertex v, we write respectively $\delta(v)$ and $d(v)$ instead of $\delta(\{v\})$ and $d(\{v\})$. If all W associated with the outgoing cut $\delta^+(W)$ contains the vertex u but not the vertex v, then we call it uv-outgoing cut.

Given disjoint subsets $W_1, W_2, \ldots, W_k$ of V, then $[W_1, W_2, \ldots, W_k]$ represents the set of arcs of D having one end in W_i and the other in W_j, $i \neq j$.

A directed graph $D = (V, A)$ is *weakly connected* if no cut of D is empty. The graph d is said to be *k-connected graph* if $d^-(W) \geq k$ for all $W \subseteq V, \emptyset \neq W \neq V$. A vertex $v \in V$ is called *cut vertex* of D if the number of connected components of the graph $D \setminus v$ is strictly greater than the number of related components of D.

If a graph $D = (V, A)$ does not contain circuit, then D is said acyclic.

1.4 Exact methods

1.4.1 Elements of polyhedral theory

The polyhedral method was initiated by Edmonds in 1965 [109] for a matching problem. It consists in describing the convex hull of problem solutions by a system of linear inequalities. The problem reduces then to the resolution of a linear program. In most of the cases, it is not straightforward to obtain a complete characterization of the convex hull of the solutions for a combinatorial optimization problem. However, having a system of linear inequalities that partially describes the solutions polyhedron may often lead to solve the problem in polynomial time. This approach has been successfully applied to several combinatorial optimization problems. In this Section, we present the basic notions of polyhedral theory. The reader is referred to works of Schrijver [243] and [196].

We shall first recall some definitions and properties related to polyhedral theory.

Let n be a positive integer and $x \in \mathbb{R}^n$. Let say that x is a *linear combination* of $x_1, x_2, \ldots, x_m \in \mathbb{R}^n$ if there exist m scalar $\lambda_1, \lambda_2, \ldots, \lambda_m$ such that $x = \sum_{i \in 1}^{m} \lambda_i x_i$. If $\sum_{i=1}^{m} \lambda_i = 1$, then x is said to be a *affine combination* of $x_1, x_2, \ldots, x_m$. Moreover, if $\lambda_i \geq 0$, for all $i \in \{1, \ldots, m\}$, we say that x is a *convex combination* of $x_1, x_2, \ldots, x_m$.

Given a set $S = \{x_1, \ldots, x_m\} \in \mathbb{R}^{n \times m}$, the *convex hull* of S is the set of points $x \in \mathbb{R}^n$ which are convex combination of $x_1, \ldots, x_m$ (see Figure 1.7), that is

$$conv(S) = \{x \in \mathbb{R}^n | x \text{ is a convex combination of } x_1, \ldots, x_m\}.$$

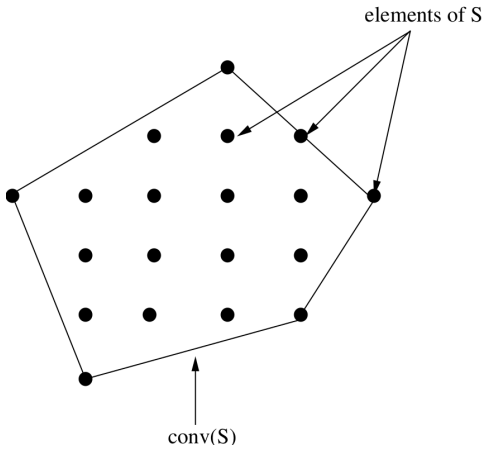

Figure 1.7: A convex hull

The points $x_1, \ldots, x_m \in \mathbb{R}^n$ are *linearly independents* if the unique solution of the system

$$\sum_{i=1}^{m} \lambda_i x_i = 0,$$

is $\lambda_i = 0$, for all $i \in \{1, \ldots, m\}$. They are *affinely independent* if the unique solution of the system

$$\sum_{i=1}^{m} \lambda_i x_i = 0,$$
$$\sum_{i=1}^{m} \lambda_i = 0,$$

is $\lambda_i = 0$, $i = 1, \ldots, m$.

A *polyhedron* P is the set of solutions of a linear system $Ax \leq b$, that is $P = \{x \in \mathbb{R}^n | Ax \leq b\}$, where A is a m-row n-columns matrix and $b \in \mathbb{R}^m$. A *polytope* is a bounded polyhedron. A point x of P will be also called a *solution* of P.

A polyhedron P is said to be of *dimension* p if it has at most $p+1$ affinely independent solutions. We denote it by $dim(P) = p$. We also have that $dim(P) = n - rank(A^=)$, where $A^=$ is the submatrix of A of *inequalities* that are satisfied with equality by all solutions of P (implicit equalities). The polyhedron P is full dimensional if $dim(P) = n$.

An inequality $ax \leq \alpha$ is *valid* for a polyhedron $P \subseteq \mathbb{R}^n$ if for every solution $\overline{x} \in P$, $a\overline{x} \leq \alpha$. This inequality is said to be *tight* for a solution $\overline{x} \in P$ if $a\overline{x} = \alpha$. The inequality $ax \leq \alpha$ is *violated* by $\overline{x} \in P$ if $a\overline{x} > \alpha$. Let $ax \leq \alpha$ be a valid inequality for the polyhedron P. $F = \{x \in P | ax = \alpha\}$ is called a *face* of P. We also say that F is a *face induced by* $ax \leq \alpha$. If $F \neq \emptyset$ and $F \neq P$, we say that F is a *proper* face of P. If F is a proper face and $dim(F) = dim(P) - 1$, then F is called a *facet* of P. We also say that $ax \leq \alpha$ induces a facet of P or is a *facet defining* inequality.

If P is full dimensional, then $ax \leq \alpha$ is a facet of P if and only if F is a proper face and there exists a facet of P induced by $bx \leq \beta$ and a scalar $\rho \neq 0$ such that $F \subseteq \{x \in P | bx = \beta\}$ and $b = \rho a$.

If P is not full dimensional, then $ax \leq \alpha$ is a facet of P if and only if F is a proper face and there exists a facet of P induced by $bx \leq \beta$, a scalar $\rho \neq 0$ and $\lambda \in \mathbb{R}^{q \times n}$ (where q is the number of lines of matrix $A^{=}$) such that $F \subseteq \{x \in P | bx = \beta\}$ and $b = \rho a + \lambda A^{=}$.

An inequality $ax \leq \alpha$ is *essential* if it defines a facet of P. It is *redundant* if the system $A'x \leq b'\}$ obtained by removing this inequality from $Ax \leq b$ defines the same polyhedron P. This is the case when $ax \leq \alpha$ can be written as a linear combination of inequalities of the system $A'x \leq b'$. A *complete minimal linear description* of a polyhedron consists of the system given by its facet defining inequalities and its implicit equalities.

A solution is an *extreme point* of a polyhedron P if and only if it cannot be written as the convex combination of two different solutions of P. It is equivalent to say that x induces a face of dimension 0. The polyhedron P can also be described by its extreme points. In fact, every solution of P can be written as a convex combination of some extreme points of P.

Figure 1.8 illustrates the main definitions given is this Section.

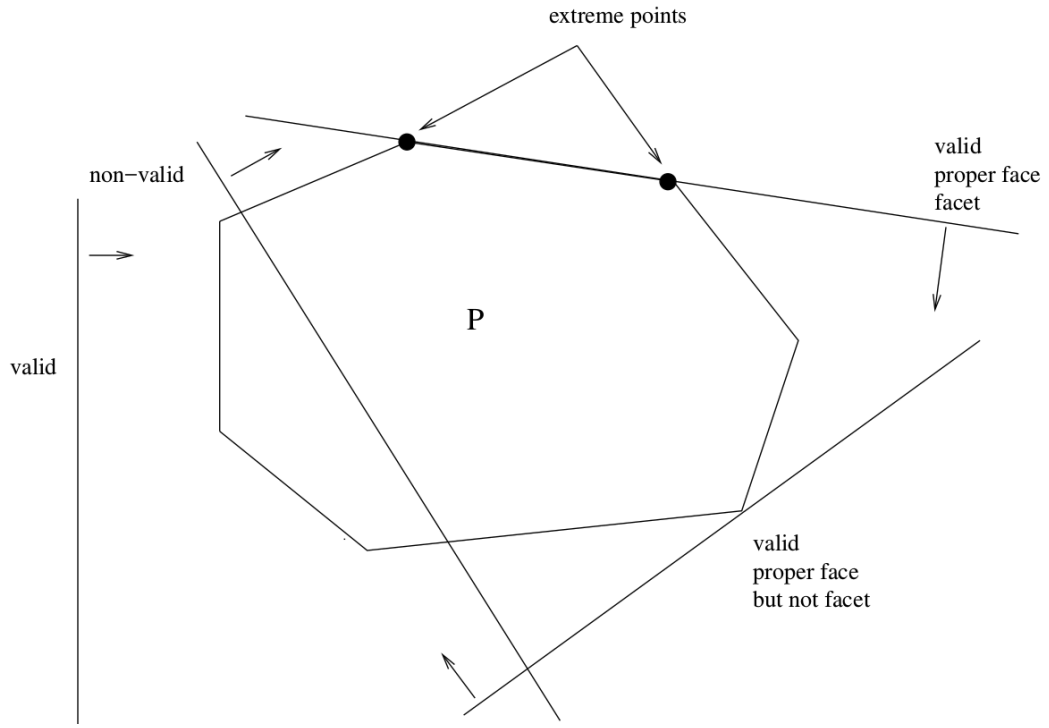

Figure 1.8: Valid inequality, facet and extreme points

Let $Ax \leq b$ be a linear system. The system $Ax \leq b$ is said to be *Totally Dual Integral* (TDI) if for all integer $c \in \mathbb{Z}^n$, the linear program $\min\{b^\top y : A^\top y \geq c; y \geq 0\}$ has an integer optimal solution, if such solution exists.

If $Ax \leq b$ is TDI and b is integral, then the polytope given by $Ax \leq b$ is integral.

1.4.2 Cutting plane method

Now let $\mathcal{P}$ be a combinatorial optimization problem, E its basic set, $c(.)$ the weight function associated with the variables of $\mathcal{P}$ and $\mathcal{S}$ the set of feasible solutions. Suppose that $\mathcal{P}$ consists in finding an element of $\mathcal{S}$ whose weight is maximum. If $F \subseteq E$, then the 0-1 vector $x^F \in \mathbb{R}^E$ such that $x^F(e) = 1$ if $e \in F$ and $x^F(e) = 0$ otherwise, is called the *incidence vector* of F. The polyhedron $P(\mathcal{S}) = conv\{x^S | S \in \mathcal{S}\}$ is the *polyhedron of the solutions* of $\mathcal{P}$ or *polyhedron associated with* $\mathcal{P}$. $\mathcal{P}$ is thus, equivalent to the linear program $max\{cx | x \in P(\mathcal{S})\}$. Notice that the polyhedron $P(\mathcal{S})$ can be described by a

set of a facet defining inequalities. And when all the inequalities of this set are known, then solving $\mathcal{P}$ is equivalent to solve a linear program.

Recall that the objective of the polyhedral approach for combinatorial optimization problems is to reduce the resolution of $\mathcal{P}$ to that of a linear program. This reduction induces a deep investigation of the polyhedron associated with $\mathcal{P}$. It is generally not easy to characterize the polyhedron of a combinatorial optimization problem by a system of linear inequalities. In particular, when the problem is NP-hard there is a very little hope to find such a characterization. Moreover, the number of inequalities describing this polyhedron is, most of the time, exponential. Therefore, even if we know the complete description of that polyhedron, its resolution remains in practice a hard task because of the large number of inequalities.

Fortunately, a technique called the *cutting plane method* can be used to overcome this difficulty. This method is described in what follows.

The cutting plane method is based on the so-called *separation problem*. This consists, given a polyhedron P of $\mathbb{R}^n$ and a point $x^* \in \mathbb{R}^n$, in verifying whether if x^* belongs to P, and if this is not the case, to identify an inequality $a^T x \leq b$, valid for P and violated by x^*. In the later case, we say that the hyperplane $a^T x = b$ separates P and x^* (see Figure 1.9).

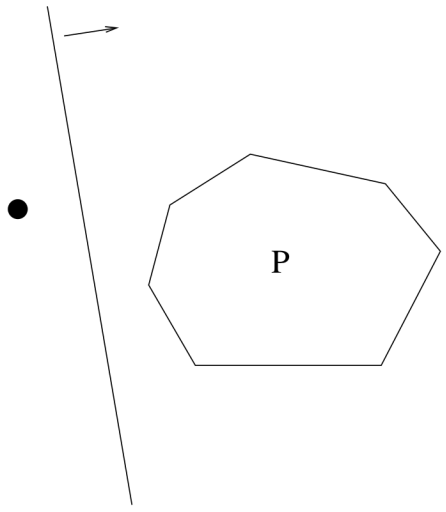

Figure 1.9: A hyperplane separating x^* and P

Grötschel, Lovász and Schrijver [143] have established the close relationship between separation and optimization. In fact, they prove that optimizing a problem over a polyhedron P can be performed in polynomial time if and only if the separation problem associated with P can be solved in polynomial time. This equivalence has permitted an important development of the polyhedral methods in general and the cutting plane method in particular. More precisely, the *cutting plane* method consists in solving successive linear programs, with possibly a large number of inequalities, by using the following steps. Let $LP = \max\{cx, Ax \leq b\}$ be a linear program and LP' a linear program obtained by considering a small number of inequalities among $Ax \leq b$. Let x^* be the optimal solution of the latter system. We solve the separation problem associated with $Ax \leq b$ and x^*. This phase is called the *separation phase*. If every inequality of $Ax \leq b$ is satisfied by x^*, then x^* is also optimal for LP. If not, let $ax \leq \alpha$ be an inequality violated by x^*. Then, we add $ax \leq \alpha$ to LP' and repeat this process until an optimal solution is found. Algorithm 6 summarizes the different cutting plane steps.

Algorithm 6: A cutting plane algorithm

Data: A linear program LP and its system of inequalities $Ax \leq b$
Result: Optimal solution x^* of LP
Consider a linear program LP' with a small number of inequalities of LP;
Solve LP' and let x^* be an optimal solution;
Solve the separation problem associated with $Ax \leq b$ and x^*;
if *an inequality* $ax \leq \alpha$ *of* LP *is violated by* x^* **then**
    Add $ax \leq \alpha$ to LP';
    Repeat step 2 ;
end
else
    x^* is optimal for LP;
    return x^*;
end

Note that at the end, a cutting-plane algorithm may not succeed in providing an optimal solution for the underlying combinatorial optimization problem. In this case a *Branch-and-Bound algorithm* can be used to achieve the resolution of the problem, yielding to the so-called *Branch-and-Cut algorithm*.

1.4.3 Branch-and-Cut algorithm

Consider again a combinatorial optimization problem $\mathcal{P}$ and suppose that $\mathcal{P}$ is equivalent to $\max\{cx|Ax \leq b, x \in \{0,1\}^n\}$, where $Ax \leq b$ has a large number of inequalities. A Branch-and-Cut algorithm starts by creating a Branch-and-Bound tree whose root node corresponds to a linear program $LP_0 = max\{cx|A_0x \leq b_0, x \in \mathbb{R}^n\}$, where $A_0x \leq b_0$ is a subsystem of $Ax \leq b$ having a small number of inequalities. Then, we solve the linear relaxation of $\mathcal{P}$ that is $LP = \{cx|Ax \leq b, x \in \mathbb{R}^n\}$ using a cutting plane algorithm whose starting from LP_0. Let x_0^* denote its optimal solution and $A_0'x \leq b_0'$ the set of inequalities added to LP_0 at the end of the cutting plane phase. If x_0^* is integral, then it is optimal. If x_0^* is fractional, then we perform a *branching phase*. This step consists in choosing a variable, say x^1, with a fractional value and adding two nodes P_1 and P_2 in the Branch-and-Cut tree. The node P_1 corresponds to the linear program $LP_1 = \max\{cx|A_0x \leq b_0, A_0'x \leq b_0', x^1 = 0, x \in \mathbb{R}^n\}$ and $LP_2 = \max\{cx|A_0x \leq b_0, A_0'x \leq b_0', x^1 = 1, x \in \mathbb{R}^n\}$. We then solve the linear program $\overline{LP}_1 = \max\{cx|Ax \leq b, x^1 = 0, x \in \mathbb{R}^n\}$ (resp. $\overline{LP}_2 = \max\{cx|Ax \leq b, x^1 = 1, x \in \mathbb{R}^n\}$) by a cutting plane method, starting from LP_1 (resp. LP_2). If the optimal solution of $\overline{LP}_1$ (resp. $\overline{LP}_2$) is integral then, it is feasible for $\mathcal{P}$. Its value is then a lower bound of the optimal solution of $\mathcal{P}$, and the node P_1 (resp. P_2) becomes a leaf of the Branch-and-Cut tree. If the solution is fractional, then we select a variable with a fractional value and add two children to the node P_1 (resp. P_2), and so on.

Note that sequentially adding constraints of type $x^i = 0$ and $x^i = 1$, where x^i is a fractional variable, may lead to an infeasible linear program at a given node of the Branch-and-Cut tree. Or, if it is feasible, its optimal solution may be worse than the best known lower bound of the problem. In both cases, that node is pruned from the Branch-and-Cut tree. The algorithm ends when all nodes have been explored and the optimal solution of $\mathcal{P}$ is the best feasible solution given by the Branch-and-Bound tree.

This algorithm can be improved by computing a good lower bound of the optimal solution of the problem before it starts. The lower bound can be used by the algorithm to prune the node which will not allow an improvement of this lower bound. This would permit to reduce the number of nodes generated in the Branch-and-Cut tree, and hence, reduce the time used by the algorithm. Furthermore, this lower bound may be improved by comparing at each node of the Branch-and-Cut tree a feasible solution when the solution obtained at the root node is fractional. Such a procedure is referred to as a *primal heuristic*. It aims to produce a feasible solution for $\mathcal{P}$ from the solution obtained at a given node of the Branch-and-Cut tree, when this later solution is fractional (and hence infeasible for $\mathcal{P}$). Moreover, the weight of this solution must be as good as possible. When the solution computed is better than the best known lower

bound, it may significantly reduce the number of generated nodes, as well as the CPU time. Moreover, this guarantees to have an approximation of the optimal solution of $\mathcal{P}$ before visiting all the nodes of Branch-and-Cut tree, for instance when a CPU time limit has been reached.

The Branch-and-Cut approach has shown a great efficiency to solve various problems of combinatorial optimization that are considered difficult to solve, such as the Travelling Salesman Problem [20]. Note a good knowledge of the polyhedron associated with the problem, together with efficient separation algorithms (exacts as well as heuristics), might help to improve the effectiveness of this approach. Besides, the cutting plane method is efficient when the number of variables is polynomial. However, when the number of variables is large (for instance exponential), further methods, as column generation are more likely to be used. In what follows, we briefly introduce the outline of this method.

1.5 Metaheuristics

Approximation methods are generally based on two major phases; A constructive heuristic and a phase of improvement.

A constructive heuristic constructs the solution by a series of partial and final choices. It is called a greedy method when it seeks to make the most judicious choice at each iteration. A greedy algorithm is thus called an algorithm which follows the principle of making, step by step, an optimal choice locally, in the hope of obtaining an optimal result globally. These algorithms have the advantage of being very fast however they do not provide any guarantees as to the quality of the solution.

The improvement phase, as the name suggests, tends to improve the quality of a fully constructed solution received as input by replacing at each iteration the solution by another one belonging to its neighborhood but of better quality. This local search generally leads to obtaining a local optimal solution.

Feignebaum and Feldman in 1963 [112], define a heuristic as an estimation rule, a strategy, a trick, a simplification, or any other kind of system that drastically limits the search for solutions in the space of possible configurations. In practice, heuristics are generally known and targeted to a particular problem.

In the 70ies, metaheuristics has emerged as a new kind of approximate algorithm that, in the other hand, takes place at a higher level and intervenes in all situations

where the engineer knows no effective heuristics to solve a given problem or when he considers that he does not have the necessary time to determine one.

In 1996, Osman and Laporte [214] defined metaheuristics as "an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions."

Metaheuristics are based on two axes, namely diversification, and intensification. Diversification generally refers to the exploration of the research space and thus the gathering of information on the resolution of the COP concerned. The term intensification refers to the exploitation of the accumulated search experience. Each metaheuristic application is characterized by a balance between diversification and intensification.

There are different ways to classify and describe metaheuristics. The most common and usefull for us in this work is to classify the metaheuristics into methods that perform single point vs. population based Seach. It refers to the number of solutions used by the metaheuristic at any iteration.

Algorithms that work on a single solution at any time are referred to as trajectory methods. They comprise all metaheuristics that are based on local search, such as tabu search, variable neighborhood search, etc. They all share the property that the search process describes a trajectory in the search space. Population-based metaheuristics, on the contrary, either perform search processes which can be described as the evolution of a set of points in the search space (as for example in evolutionary computation), or they perform search processes which can be described as the evolution of a probability distribution over the search space (as for example in ant colony optimization).

In what follows, we give a brief outline of some of the most important methods.

1.5.1 Trajectory metaheuristics

1.5.1.1 Simulated Annealing

Simulated Annealing (SA) is one of the oldest (if not the oldest) metaheuristic. For sure, SA is one of the first algorithms with an explicit strategy for escaping local minima. The algorithm has its origins in statistical mechanics and the Metropolis algorithm [203]. The idea of SA is inspired by the annealing process of metal and glass, which assume a low energy configuration when first heated up and then cooled

down sufficiently slowly. SA was first presented as a search algorithm for CO problems in [183] and [57].

The fundamental idea is to allow moves to solutions with objective function values that are worse than the objective function value of the current solution. This kind of move is often called uphill move. At each iteration, an elementary modification of the solution is carried out according to a well-defined model:

- If it lowers the temperature of the system ⇒ it is applied
- Otherwise ⇒ it is accepted at a probability p (calculated as a function of the temperature T_k of the system)

Moreover, during a run of SA, the value of T_k generally decreases. In this way, the probability of accepting a solution that is worse than the current one decreases during a run.

1.5.1.2 Tabu Search

Like SA, Tabu Search (TS) [131] is one of the older metaheuristics. A description of the method and its concepts can be found in [132].

The basic idea of TS is the explicit use of search history, noth to escape from local optima and to have a mechanism for the exploration of the search space. It uses a short-term memory, and on each iteration it makes the best change of the solution (even if negative) by saving the positions already explored is an adjustable-size FIFO queue (can be dynamic).

1.5.1.3 Iterated Local Search

Iterated Local Search (ILS) [247, 191] is a metaheuristic based on a simple but effective concept. Instead of repetitively applying a local search method on randomly constructed solutions (the most intuitive idea of finding a good local minimum), an ILS algorithm produces the starting point for the next iteration by a perturbation of the local optimal solution obtained by the previous application of a local search.

1.5.1.4 Variable neighborhood search

This metaheuristic was first proposed in [150, 151]. The main idea of this method is based on the fact that a local optima solution of a problem corresponds to a well-determined neighborhood function, is not necessarily optimal locally with respect to different neighborhood function. The metaheuristic then performs at each iteration the best change in the solution, and if the solution is no longer possible, the search neighborhood is enlarged until a limit K is reached.

1.5.2 Population-based metaheuristics

Population-based methods deal at each iteration of the algorithm with a set of solutions rather than with a single solution. They provide a natural way for the exploration of the search space. Yet, the final performance strongly depends on the way the population is manipulated. The most studied population-based metaheuristics are Evolutionary Algorithms (EA) and Ant Colony Optimization (ACO).

1.5.2.1 Ant Colony

The Ant Colony Optimization metaheuristic [103] was inspired by the observation of the shortest-path-finding behavior of natural ant colonies.

Initially, ants explore the area surrounding their nest in a random manner. As soon as an ant finds a source of food, it evaluates quantity and quality of the food and carries some of this food to the nest. During the return trip, the ant deposits a chemical pheromone trail on the ground. The quantity of pheromone deposited, which may depend on the quantity and quality of the food, will guide other ants to the food source. The indirect communication between the ants via the pheromone trails allows them to find shortest paths between their nest and food sources. This functionality of real ant colonies is exploited in artificial ant colonies in order to solve hard optimization problems.

In ACO algorithms the chemical pheromone trails are simulated via a parametrized probabilistic model that is called the pheromone model. It consists of a set of model parameters whose values are called the pheromone values. These values act as the memory that keeps track of the search process. The basic ingredient of ant colony optimization

algorithms is a constructive heuristic that is used for probabilistically constructing solutions using the pheromone values. In general, the ant colony optimization approach attempts to solve a CO problem by iterating the following two steps:

- Solutions are constructed using a pheromone model, that is, a parametrized probability distribution over the solution space.
- The constructed solutions and possibly solutions that were constructed in earlier iterations are used to modify the pheromone values in a way that is deemed to bias future sampling toward high quality solutions.

1.5.2.2 Genetic Algorithms

Evolutionary computation (EC) can be regarded as a metaphor for building, applying, and studying algorithms based on Darwinian principles of natural selection. The instances of algorithms that are based on evolutionary principles are called evolutionary algorithms (EAs) [25]. EAs can be characterized as computational models of evolutionary processes. They are inspired by nature's capability to evolve living beings well adapted to their environment. At the core of each EA is a population of individuals. At each algorithm iteration a number of reproduction operators is applied to the individuals of the current population to generate the individuals of the population of the next generation. EAs might use operators called recombination or crossover to recombine two or more individuals to produce new individuals. They also can use mutation or modification operators which cause a self-adaptation of individuals.

The driving force in EAs is the selection of individuals based on their fitness (which might be based on the objective function, the result of a simulation experiment, or some other kind of quality measure). Individuals with a higher fitness have a higher probability to be chosen as members of the population of the next generation (or as parents for the generation of new individuals). This corresponds to the principle of survival of the fittest in natural evolution. It is the capability of nature to adapt itself to a changing environment, which gave the inspiration for EAs.

There has been a variety of different EAs proposed over the decades. Three different stands of EAs developed independently in the early years. These are Evolutionary Programming (EP) as introduced by Fogel in [117] and Fogel et al. in [118], Evolutionary Strategies (ESs) proposed by Rechenberg in [232] and Genetic Algorithms (GAs) initiated by Holland in [157, 158].

1.6 Parallel computing

1.6.1 Hardware architectures

The evolution of parallel architectures over time has made possible the fact of producing machines whose computing power grows faster and faster. A simple way to understand the different types of architecture that are or have been the basis of these powerful machines is to compare their main features. To this end, we will present two complementary classifications known in the literature. The first is that of Flynn [116, 107] based on the two dimensions given instructions. As for the second classification, it identifies the different parallel architectures according to the interconnection of their components.

1.6.1.1 Flynn taxonomy

In 1972, Flynn [116, 107] proposed a classification of parallel machines based on the notion of stream(s) of data and stream(s) of instruction. It distinguishes four main types basically:

- the Single Instruction Single Data architecture (SISD),
- the Single Instruction Multiple Data architecture (SIMD),
- the Multiple Instruction Single Data architecture (MISD),
- and the Multiple Instruction Multiple Data architecture (MIMD).

1.6.1.1.1 SISD architecture

In these machines, only one instruction is executed and only one data is processed at any time. The treatments in this type of machines are therefore sequential (without parallelism). This model corresponds to a conventional single-core Von Neumann machine.

1.6.1.1.2 SIMD architecture

In these machines, all processors are identical and are controlled by a single centralized control unit. At each step, all processors execute the same instruction synchronously but on different data. This type of machine is used for specialized calculations (e.g. vector calculations). These machines correspond, for example, to graphic processors (GPUs) or floating point units (FPUs).

1.6.1.1.3 MISD architecture

These machines can execute multiple instructions on the same data. This model includes some pipeline architectures and fault-tolerant architectures by computational replication.

1.6.1.1.4 MIMD architecture

In this model, each processor is autonomous, has its own control unit, and runs its own stream of instructions on its own data stream. These machines are the most common today: one finds the machines with shared memory (SMP, MPP), the machines with distributed memory and the machines with hybrid memory.

1.6.1.2 Other taxonomy

The difficulty of transposing the Flynn classification into real physical architectures justifies the use of a second taxonomy [74, 30]. This second approach will give a complementary view to that of Flynn and classifies the architectures according to the interconnections of the processors and the memories. Five classes are often identified:

- the Symmetric MultiProcessors (SMP),
- the Massively Parallel Processors (MPP),
- the Multiple-cores,
- the Cluster,
- and the Grid.

1.6.1.2.1 SMP

This architecture is composed of several identical processors sharing the same memory (the processors read and write in the same memory, but generally have their own cache). The memory access time is identical between processors. These machines that access at the same speed all memory areas are also called UMA (Uniform Memory Access) machines. However, access to memory constitutes a bottleneck on this type of architecture as soon as the number of processors becomes important.

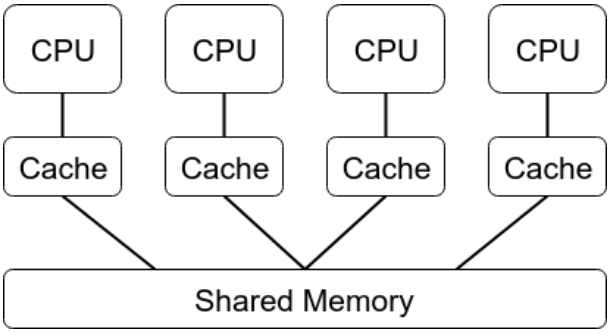

Figure 1.10: Example of an SMP architecture

1.6.1.2.2 MPP

In a multiprocessor architecture of MPP type, unlike SMP machines, processors do not share either a single memory or inputs and outputs. Each processor has its own memory and has a fast interconnection with other processors. It is therefore a MIMD type architecture with distributed memory. Any processor in an MPP may have its own operating system (OS), which makes it more difficult to implement a single-image system. The use of standard components results in a good cost / performance relationship. MPP machines have no limit on the number of processors and are easily expandable. However, due to the lack of a single-image OS, their programming is more difficult than that of SMP machines. Thus, communication and inter-processor coordination are explicit.

To overcome the deficiencies in the distributed memory of MPP architectures, an evolution concerns the creation of parallel machines with Distributed Shared Memory (DSM). Unlike SMP, access to the virtual memory commonly depends on the physical location of the processor and the memory in the parallel machine. This is called a NUMA (Non Uniform Memory Access) machine. However, in order not to complicate the work of the programmer, it is also necessary that the different cache levels of the

processors are synchronized with the memory addresses they represent. This is called ccNUMA (Cache Coherency NUMA).

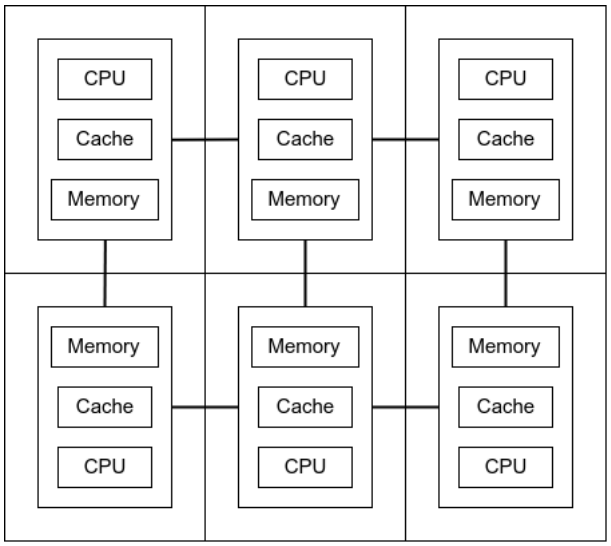

Figure 1.11: Example of an MPP architecture

1.6.1.2.3 Multiple-cores

A multi-core processor is composed of at least two computing units (cores) engraved within the same chip. Processor cores in most cases are homogeneous (identical). But IBM, Sony and Toshiba have exploited the case of heterogeneous (different) cores and specialized in specific fields (audio, display, pure calculation). Since 2005, multi-core processors have become predominent on the microprocessors market [160].

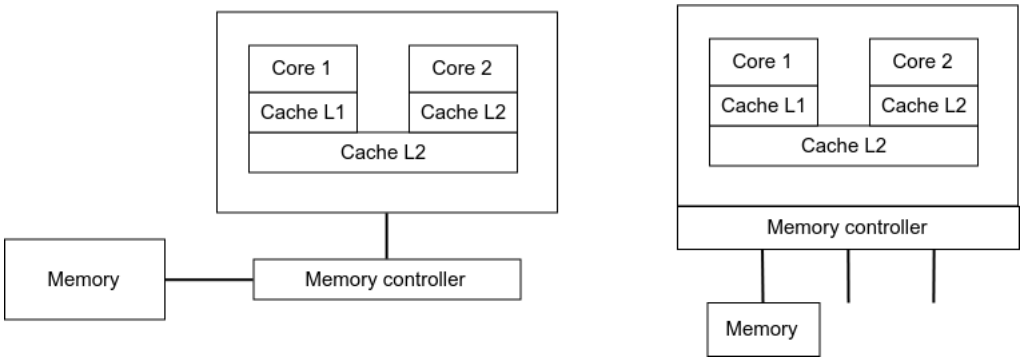

Figure 1.12: Two examples of multiple-cores architectures

1.6.1.2.4 Cluster

A cluster is not a single parallel machine in the traditional sense but consists of a set of nodes (single-processor machines, multi-core processors or SMPs) interconnected by a local (and often fast) network. It is therefore a network of computers in general of homogeneous character. One of the primary objectives of a cluster is to provide a single-image environment for a single system for cluster-based applications. For this, it is often associated with a shared virtual memory system. By its nature, a cluster consists of only standard components and thus has a very good cost/performance ratio. In addition, it is easily expandable and has become one of the most common parallel architectures today.

The difference between a cluster and an MPP decreases more and more. The existence of suitable environments now makes it possible to deploy a cluster with the same computational power as an MPP or an SMP parallel server which are much more expensive [73].

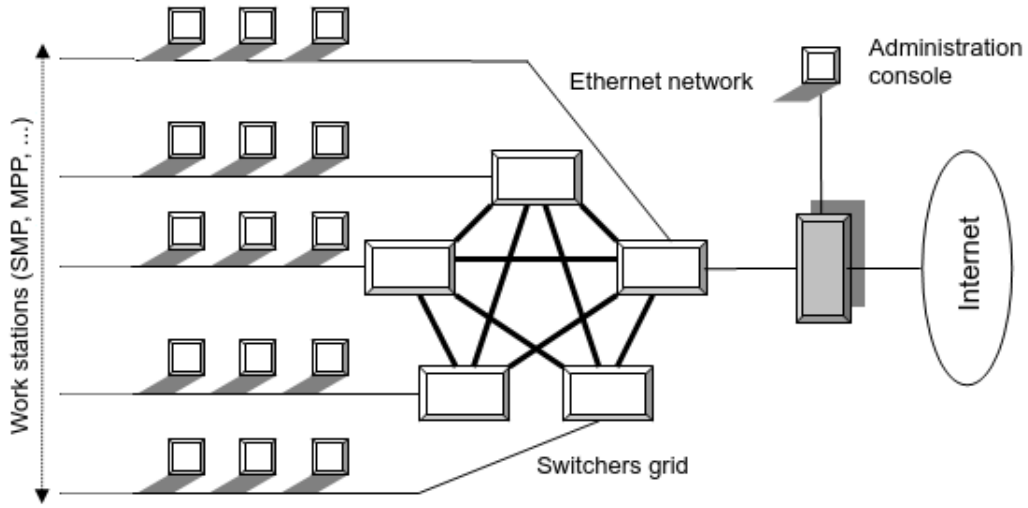

Figure 1.13: Example of a cluster architecture

1.6.1.2.5 Grid architecture

A grid is a set of clusters interconnected by a very high-speed network. It is a very heterogeneous architecture since the materials and the systems which constitute it can be very composite. Notice, for example, the unifying project of a grid for research in France, called Grid 5000 [66].

1.6.1.3 Summary

Among the classes cited, the most general and most used are those of the MIMD and SIMD machines. In the MIMD class, Distributed Memory (DM) machines, Shared Memory (SM) machines and Hybrid Memory (HM) machines are distinguished.

In the SM model, all the processors in the parallel machine share the same memory space in which they can read and write independently and asynchronously. The memory access costs can be identical for each of the processors (UMA: Uniform Memory Access) or processor dependent and the addresses accessed (NUMA machines). Simultaneous executions are performed by threads. The execution threads share the same memory space and the data exchanges between them are simply made by reading / writing in memory. It is therefore necessary to ensure that two execution threads do not modify at the same time the value of a data item in the same area of the shared memory. This can be done by using synchronization tools such as locks, semaphores, or synchronization barriers.

In the DM model, each processor of the parallel machine has its own local memory, and the processor memories are physically distant [188]. The data communication between the processors is done by exchanging messages and also the synchronizations between the processors. Communications can be synchronous. When a message is sent by a transmitter, the receiver receives them and its response must be explicit. Communications can be bipoint or collective, that is, by involving all processors in the exchanges. The main disadvantage of this model is the cumbersome programming, since a processor can only access data in its own memory. This architecture, which does not allow direct access to a remote memory, is also known as NORMA (Non Remote Memory Access) or distributed memory machines. Shared virtual memories (MVPs) provide a vision of a global space (NUMA) from a paging memory [146].

To combine the advantage of interprocessor communication speed in a shared memory machine and the high number of processors in the distributed memory architecture, manufacturers have been thinking about a new shared-distributed hybrid architecture. The machines of this type consist of nodes, called clusters, each of which is composed of several processors sharing a common memory, all of these nodes being connected by an interconnection network. Here, the intra-node shared memory architecture and the inter-node distributed memory architecture are assembled [146].

1.6.2 Modeling the program execution

In this section, we present the different representations of the execution of a parallel algorithm. Generally, three graph families are used to represent the execution of parallel programs: dependency graph, precedence graph and dataflow graph [128].

1.6.2.1 Dependency graph

A Dependency Graph (DG) is an undirected graph. The nodes of the graph represent the tasks, or even the data of the program, and the edges represent the dependencies between the tasks (bipoint communications), or between a task and a data item.

1.6.2.2 Precedence graph

A Precedence Graph (PG) is a circuit-free oriented graph (GOSC). The nodes of the graph represent the tasks of the program and the edges represent the dependencies between the tasks. Dependencies are introduced to set up data access conflicts. They can be interpreted as communication between tasks. In this type of graph, a task is considered ready as soon as all the tasks that precede it in the graph are completed. This type of graph is used for dynamic parallel programs such as, for example, recursive programs or programs containing parallel loops whose boundaries are not known before execution.

1.6.2.3 Dataflow graph

Since the dependencies between tasks are considered to be data exchange communications, it is possible to add information on data communicated in the graph. A data accessed by a task can be characterized by the type of operation that the task will perform on it, as detailed below.

- Read-only: the task will not modify the data but will simply read its value
- Write Only: The task can only use the data to assign it
- Write in accumulation: several tasks will carry out an associative and commutative operation on a data in writing (a calculation of scalar product for example)

- Change: The task will change the value of the data.

This graph is called a Dataflow graph (DFG). The difference between the previous (PG) and dataflow (DFG) graphs is at the synchronization level. In a DFG, a task is considered ready as soon as the data it has input is available. Note, however, that a DFG contains information from a precedence graph. For load control, this type of graph provides more information. Indeed, since the data communicated between the tasks are identified, the sizes of the data can also be provided. This information can be used to exploit the locality of the data during the placement of tasks or for the routing of the data accessed to the site of the task execution.

1.6.3 Parallelization methodology

The main reason for designing a parallel algorithm is the expected gain in execution time. There are essentially two methods for designing a parallel algorithm, one of which consists in detecting and exploiting the inherent parallelism in an already existing sequential algorithm, the other consisting in constructing a new algorithm dedicated to the given problem. The design of a parallel algorithm for a given problem is much more complex than that of a sequential algorithm. Several factors will have to be taken into account, inter alia, that part of the corresponding program which can be processed in parallel, how to distribute the data, data dependencies, charge distribution between the processors, synchronizations between the processors, etc.

In fact, there is no simple and general recipe for obtaining parallel algorithms. However, a methodology can be proposed that organizes the process in four distinct stages: (i) partitioning (segmentation), (ii) communication, (iii) agglomeration, and (iv) placement [87]. The first two steps seek to develop competing and scalable algorithms. It should be noted that the scalability of a parallel algorithm on a parallel architecture refers to the ability of the algorithm-architecture combination to deliver increasing speedups in proportion to the increase in the number of processors. As for the last two stages, they focus on the locality and the performance problems [208].

1.6.3.1 Partitionning

Partitioning (or segmentation) refers to the decomposition of computing activities and associated data on which it operates in several small tasks. The decomposition of data associated with a problem is called domain/data decomposition, and the computational

decomposition into disjoint tasks is called functional decomposition [208]. Various paradigms that can be used in this stage have emerged in the literature, including the famous Divide and Conquer Paradigm (DPR).

1.6.3.2 Communication

It focuses on the flow of information and coordination between the tasks that are created during the partitioning step. The nature of the problem and the method of decomposition determine the mode of communication between the cooperative tasks of a parallel program. The four modes of communication commonly used in parallel programs are (i) local/global, (ii) structured/unstructured, (iii) static/dynamic, and (iv) synchronous/asynchronous.

1.6.3.3 Agglomeration

In this step, the tasks and communication structure defined in the first two steps are evaluated based on performance requirements and implementation costs. If necessary, tasks are grouped into larger tasks (large grains) to improve performance or reduce development costs. Individual communications can be grouped into a super communication. This will reduce communications costs by increasing computational and communication granularities, gain flexibility based on scalability and investment decisions, and reduce engineering software costs.

1.6.3.4 Placement

It involves assigning each task to a processor to maximize the use of system resources (such as CPUs) while minimizing communication costs. Placement decisions can be taken statically (at compile time and prior to execution of the program) or dynamically during execution by load balancing methods. It should be noted that several major challenge applications have been designed using the above methodology [87].

1.6.4 Performance metrics

Consider a parallel algorithm P_A to implement on a multiprocessor machine M made up of p identical processors. Let T_p be the execution time of P_A on the p processors and T_1 the execution time on only 1 processor. We then define [128, 164, 208]:

- the Cost $C_p = pT_p$
- the Speedup $S_p = \frac{T_1}{T_p}$
- the Efficiency $E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} = \frac{T_1}{C_p}$

The notion of the cost of an algorithm is intended to take into account both the execution time and the number of processors used to obtain it. Adding processors is always expensive (purchase, installation, maintenance, etc.), for the same run time. With a fixed duration, an algorithm that requires the use of fewer processors than another is therefore always preferable.

The ideal speedup is equal to p. If we consider this function, the parallel algorithm is all the more efficient as the curve of variations is close to the line of equation $y = p$. In this case we will speak of linear speedup. To have good speedups, the extra cost of parallelization in computation time, and especially in communication volume, must be minimum. It is also necessary to apply to minimize the inactivity times of the processors due to an imbalance in the distribution of the charges between the processors. In some cases, the latter may be inevitable, in particular because of the inter-task precedences or the heterogeneity of the processors.

Note that we can introduce the notion of relative speedup, as opposed to absolute speedup, which does not take into account the intrinsically sequential (and therefore nonparallelizable) parts of the algorithm. Sometimes, the speedup is greater than p. In this case, we will say that it is super-linear [164]. This happens, for example, when the parallel algorithm allows a better use of the memory hierarchies, and thus reduces the total time. This is also possible if the sequential reference algorithm is not the same as the algorithm used for parallelization and is less efficient.

The efficiency of a parallel algorithm is generally less than 1 (or 100% if defined as a percentage). The closer the efficiency is to 1, the better the algorithm has good parallel qualities. The most important factor in decreasing efficiency is the cost of communication.

A parallel algorithm will be said of optimal cost if one has $C_p = \Theta(T_1)$, which is equivalent to $E_p = \Theta(1)$. We extend this definition for the purposes of our study by considering that a parallel algorithm is said to be of optimal cost if we have: $\frac{C_p}{T_1} \longrightarrow 1$ when $n \longrightarrow \infty$, n being the size of the problem solved by P_A.

On the other hand, in order to avoid confusion between the notion of cost of a parallel algorithm that has just been presented ($p \times duration$) and that of classical cost meaning

duration or complexity, used fairly in the literature, we will designate, in the following, the first (total work of the parallel algorithm) by the term cost C_p and the second (execution time) by the term cost only.

The communication-computation ratio associated with P_A is also defined by:

$$P_{CC} = \frac{Communication\ time}{Calculation\ time}$$

This ratio quantifies the overhead due to communications relative to the volume of computation.

Conclusion

In this Chapter we presented some basic notions on combinatorial optimization, graph theory and the theory of complexity. We then perform a brief summary of the elements of polyhedral theory in order to introduce some combinatorial exact methods, some metaheuristics and basics of parallel computing.

Chapter 2

State-of-the-art and Motivation

Contents

In this Chapter we introduce the reader to a class of COPs known as Survivable Network Design Problems (SNDPs). We present a state-of-the-art on two known problems from this class that we chose as applications and talk about the main motivation of this thesis. We introduce after that a state-of-the-art on hybridization taxonomies and a brief state-of-the-art on the history of the parallel computing works on combinatorial optimization.

2.1 Survivable Network Design Problems

Designing efficient networks are nowadays of crucial importance since networks take a large place in many fields (telecommunications, logistics, economics, IT, etc.). Addressing network design issues has raised a large class of problems.

2.1.1 Definition

Survivable Network Design Problems (SNDP) are those network design problems which aim in designing networks that are still functionning even when failures occur. The importance of survivabililty in networks has lead to a wide litterature on these problems, and several algorithms, both exact and heuristic methods, have been proposed to address them.

We consider in this chapter two SNDP, namely, the k-Edge-Connected Survivable Network Design Problem and the k-Edge-Connected Hop-Constrained Survivable Network Design Problem (respectively kESNDP and kHNDP for short). The two problems are defined as follows. Given a weighted undirected graph $G = (V, E)$ where each edge e has a weight ω_e, a set of demands $D \subseteq V \times V$, a positive integer k, the kESNDP is to find a minimum weight subgraph of G such that for each demand $\{s, t\} \in D$, there exist k edge-disjoint paths between s and t. If, in addition, we require that, for all demand $\{s, t\} \in D$, there exist k edge-disjoint st-paths of length at most L, for some integer $L \geq 2$, the problem obtained is the kHNDP.

2.1.2 State of the art

Both kESNDP and kHNDP are NP-hard and have been investigated in the litterature. The kESNDP is a particular case of a more general problem, called *General Survivable Network Design Problem* (GSNDP for short) in which it is required that there exist r_{st} edge-disjoint paths between s and t, for all $\{s, t\} \in D$. This latter, and its variants, have been studied by several authors (see for instance [246, 193, 256, 140, 142, 133]). In [133], Goemans and Bertsimas studied a variant of the problem in which each node u of the graph is given an integer $r_u \geq 0$ which corresponds to the minimum number of paths connecting u to the rest of the network, and proposed a heuristic to

solve this problem. Grötschel and Monma [140], and Grötschel et al. [142] considered the polytope associated with the GSNDP and gave some valid inequalities as well as conditions for these inequalities to define facets. In [193] presented several integer programming formulations for the GSNDP for both undirected and directed graphs. They also discussed the efficiency of each formulation in terms of LP-relaxation. Kerivin and Mahjoub in [176] presented a review of the main models associated with the GSNDP as weel as the main associated polyhedral results. Another well known SNDP is the so-called *k-edge-connected subgraph problem* (kECSP for short) in which it is required that there exist k edge-disjoint paths between each pair of nodes of the graph. This problem corresponds to the kESNDP where $D = \{\{s,t\},\ \text{for all } s,t \in V \text{ with } s \neq t\}$, and $r_{st} = k$, for all $\{s,t\} \in D$. Several papers deal with the structural properties of the solution of the kECSP. Among these, we mention the paper of Kerivin et al. [177] and Bendali et al. [39] which proposed a polyhedral approach and Branch-and-Cut algorithms for the kECSP, respectively when $k = 2$ and $k \geq 3$.

The kHNDP has also been widely studied. Dahl [80] considers the k-edge-connected hop-constrained path problem, that is the problem of finding between two distinguished nodes s and t a minimum cost path with no more than L edges ($L \leq 3$). In other words he considered the kHNDP with $k = 1$ and $|D| = 1$, and a complete description of the dominant of the polytope when $L \geq 3$. Dahl and Gouveia [81] considered the directed version of the problem. They described some valid inequalities and give a complete description of the polytope of the problem when $k = 1$, $|D| = 1$ and $L \leq 3$.

For the case where several pairs $\{s,t\}$ of terminals have to be linked by k L-hop-constrained paths, Dahl and Johannessen [83] study the 2-path network design problem which consists in finding a minimum cost subgraph connecting each pair of terminal nodes by at least one path of length at most 2. They proved also that the problem is NP-Hard even when $k = 1$ and $L = 2$. Ribeiro and Rosseti in 2002 proposed a parallel GRASP heuristic for the 2-path network design problem [233]. In [161], Huygens et al. proposed a polyhedral approach for the kHNDP when $k = 2$, $|D| \geq 2$ and $L = 2, 3$. They introduced several valid inequalities and devised a Branch-and-Cut algorithm for the problem. Diarrassouba et al. [97] investigated the kHNDP when $L = 2, 3$. They presented several integer programming formulations based on graph transformations. They also compared these formulations both in terms of LP-relaxation and in terms of efficiency. In [49], Botton et al. present the first formulation of the k-HNDP for any k, $L \geq 1$ and use a Benders decomposition method to handle the big number of variables and constraints. They present as well a computational study of various cutting plane and branch-and-cuts algorithms.

Diarrassouba et al. [100] also investigated a version of the kHNDP in which it is required that the paths are node-disjoint. They presented some valid inequalities and devised a Branch-and-Cut algorithm for this problem when $k = 2$.

2.2 Parallel computing and combinatorial optimization

Traditionally, the emphasis in parallel computing has been put on numerical (linear) problems due to the huge number of applications demand in computational aerodynamics, weather forecasting, satellite image processing, military uses, etc.

We had to wait till the last of the 80s to see the first works on parallel combinatorial optimization where in [179, 180, 181], Kindervater and Lenstra presented a review of the literature on parallel computers and algorithms as far and discussed how its relevantness for the area of combinatorial optimization. Roucairol, after that, in 1989 [236], showed the interest and the impact of parallel computers on the field of non-numerical (combinatorial) algorithms. Since then, due to the progress made in the parallelism field (architectures, languages and algorithms) over the last decade has brought about real advances in combinatorial optimization at the start of the 21st century. The recent media "storm" following the solving of several hard instances of famous problems like the traveling salesman or quadratic assignment is proof of this [75]. In fact, in a continuation of his work since 1998, Peter Hahn from University of Pennsylvania used parallel computing to solve the example called Krarup30a (size 30 elements to be assigned) [147, 148, 149], and the group Anstreicher, Brixiut, Goux and Linderoth from the University of Iowa and the Argonne National Laboratory the examples Nugent27 and Nugent30 in [18, 19, 17]. Solving the instance of Nugent30 was reported widely in the American press (*Chicago Tribune, Chicago Sun Times, HPCWire, WNCSA Access Magazine,* etc.) and in a number of French papers and journals (*InfoScience, Le Monde, Transfert*, etc.). The extent of the impact of this solution was as important as that provoked by the victory of IMB's chess-playing parallel machine DeepBlue against the "human" world champion Gary Kasparov. Still in the years 1998-2000, the team, Bixby, Chvátal and Cook [21] successively solved the instances usa13509 and d15112 of the traveling salesman problem, and therefore instances of more than $10,0000$ cities, on a platform composed of about 100 machines. Notice that in 2002, the team Brixius, Goux and Linderoth [17] solved for the first time the instance Nugent30 on a platform of 2510 machines with an average of around 700 active machines [75]. In Section 2.5

we give a more detailed view about the different parallelization strategies that can be used with tree search algorithms and more precisely the Branch-and-Bound and the Branch-and-Cut algorithms.

While the optimal solution of these problems was obtained for the first time thanks to parallelism, proving its direct interest here, it is not always the same with other applications.

In this context, metaheuristics have proved their worth in combinatorial optimization when the problem to be treated is too large or too difficult to think about an exact method. Many works have presented since their high time consumption is directly linked to the quality of the solution found, metaheuristics are very often parallelized. The great number of metaheuristics multiplied by their hybridization possibilities makes it an extremely prolific domain with an extensive literature. This is the reason why we will give one of the most common classification of parallel metaheuristics in Section 2.4 a bit further.

2.3 Hybridization

The concept of hybrid metaheuristics has been commonly accepted only in the middle of the last two decades, even if the idea of combining different metaheuristic strategies and algorithms dates back to the 1980s. Today, we can observe a generalized common agreement on the advantage of combining components from different search techniques and the tendency of designing hybrid techniques is widespread in the fields of operations research and artificial intelligence. The consolidated interest around hybrid metaheuristics is also demonstrated by publications on classifications, taxonomies and overviews on the subject [72, 47, 71, 110, 225, 224, 248, 167].

We will start in what follows to introduce a brief presentation of the most commonly used taxonomies of hybrid metaheuristics that resumes well, in our vision, the hybridization literature.

Notice that we adopt the definition of hybrid metaheuristic in the broad sense of integration of a metaheuristic related concept with some other techniques (possibly another metaheuristic).

2.3.1 Talbi's taxonomy

In this classification, hybridization of metaheuristics is studied from two points of view: design and implementation. From the design point of view, a hierarchical classification in which four main classes are identified, is depicted in Figure 2.1. In the first level of the hierarchical classification, the two classes low level and high level hybrids refer to the functional composition of the hybridized metaheuristics. In low level hybrids, one of the search algorithms is encapsulated by the second method, while in high level hybrids each method is self contained and is viewed as a black box by the other method. The second level of the hierarchical classification dresses the way the hybridized metaheuristics interact with each other, or the way they are executed. The Relay class means the metaheuristics are executed sequentially or in a pipeline fashion, which means the output of one method is the input for the next method in the queue. The four basic classes are then combined to form the following generic classes: Low level Relay Hybrid (LRH), Low level Team-work Hybrid (LTH), High level Relay Hybrid (HRH) and High level Team-work Hybrid (HTH).

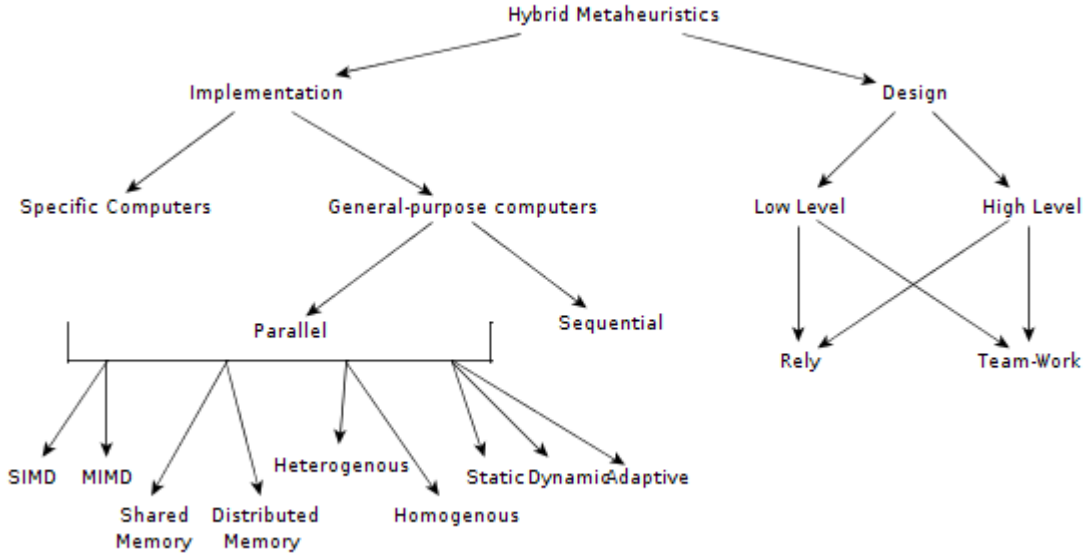

Figure 2.1: Talbi's hybrid metaheuristics classification scheme [248]

- **LRH** (low-level relay hybrid): In this class, a functional component in one method is replaced by the other method and the two methods are executed in sequence. For example, this kind of hybridization can be found in a single-solution metaheuristic embedding another single-solution metaheuristic algorithm in order to intensify the search in some regions at the end of the search. The two

methods are executed sequentially. Nevertheless, due to the similarity between the two hybridized metaheuristics, this class of hybrids is not very used.

- **LTH** (low-level teamwork hybrid): The LTH scheme regroups hybrid methods combining metaheuristics having divergent and complementary objectives. One of the metaheuristics is embedded in the other one and each metaheuristic is powerful either in exploration of new subspaces or in exploitation of good quality solutions. For example, population-based metaheuristics such as GAs are known to be powerful in exploration because of the use of multiple starting points leading to a better sampling of the solution space. On the other hand, single-solution metaheuristics are known for their exploitation capabilities. The combination of these two classes of metaheuristics justifies by definition the creation of the new class of hybrid metaheuristics whose the objective is to take advantage from the points of strength of each method. For example, it has been shown in many works that hybrid GAs combined with local search (or any s-metaheuristic such as tabu search and simulated annealing) in place of mutation or crossover operators yields much better results compared to standard GAs for combinatorial optimization and real world problems. Yet, this kind of hybridizations is very common in the literature [115, 105].

- **HRH** (high-level relay hybrid): In the HRH schemes, the hybridized metaheuristics are self-contained and executed sequentially. This class contains hybrids in which one of the methods is used either to generate initial solutions for the second method or to improve its final solution. For example, for some problems, greedy heuristics are used to construct a good quality feasible solution and this solution is used as starting solution in the metaheuristic. The other possibility is to use one metaheuristic to globally optimize the problem and use a different metaheuristic to optimize locally around the final best found solution in the global optimization step. This last scheme is used for example to improve the best solutions in the final population of a GA using a powerful intensifying metaheuristic such as simulated annealing (SA) and tabu search (TS).

- **HTH** (high-level teamwork hybrid): This class of hybrids is also considered as a parallel metaheuristic. It involves several self-contained metaheuristics which cooperate together to solve the same problem. Each metaheuristic evolves independently from the others and exchanges information with neighboring metaheuristics following a given interconnection structure or topology. For example, a set of cooperating GAs could be considered as a HTH scheme. Each GA is evolving a different subpopulation and communicates through a given topology with neighboring GAs in order to exchange some individuals to diversify the search.

The search operators and parameters used in each GA could be either the same or different. This example is also known as GAs island parallel model in parallel metaheuristics classifications [14]. In the island model even if the cooperating metaheuristics are homogeneous (multiple processes of the same algorithm), it is considered as a hybrid scheme because each island is performing an independent search in a different region of the solution space. In the taxonomy proposed in [248] there is a flat portion which refers to this last consideration (homogeneous and heterogeneous hybridizations).

From the implementation point of view, the taxonomy starts by distinguishing whether the hybrid metaheuristic is implemented on a specific or a general purpose computer. Specific purpose computers are machines built specifically for metaheuristics using programmable logic devices, i.e. simulated annealing[9]; genetic algorithms [238], in [10] a general architecture acting as a template for designing a number of specific machines for different metaheursitics (SA, TS, etc.) has been proposed.

In the second place, the nature of the hybrid metaheuristic algorithm is distinguished where most of the proposed hybrid metaheuristics are sequential programs, and according to the size of problems, the parallel implementations of hybrid algorithms have been considered instead. Using a hierarchical classification, Talbi continues by classifying parallel algorithms using the different characteristics of the target parallel architecture:

- SIMD versus MIMD: In SIMD (Single Instruction stream, Multiple Data stream) parallel machines, the processors are restricted to execute the same program. They are very efficient in executing synchronized parallel algorithms that contain regular computations and regular data transfers. When the computations or the data transfers become irregular or asynchronous, the SIMD machines become much less efficient. In parallel MIMD (Multiple Instruction stream, Multiple data stream), the processors are allowed to perform different types of instructions on different data.
- Shared-memory versus Distributed-memory: The advantages of parallel hybrids implemented on shared memory parallel architectures are their simplicity. However, parallel distributed-memory architectures offer a more flexible and fault-tolerant programming platform.
- Homogeneous versus Heterogeneous: Most massively parallel machines (MPP) and cluster of workstations (COW) are composed of homogeneous processors. The proliferation of powerful workstations and fast communication networks have

shown the emergence of heterogeneous network of workstations as platforms for high-performance computing.

Finally, Talbi underlines the fact that parallel heuristics fall into three categories depending on whether the number and/or the location of work (tasks, data) depend or not on the load state of the target parallel machine:

- Static: This category represents parallel heuristics in which both the number of tasks of the application and the location of work (tasks or data) are generated at compilation time (static scheduling). The allocation of processors to tasks (or data) remains unchanged during the execution of the application regardless of the current state of the parallel machine. Most of the proposed parallel heuristics belong to this class.
- Dynamic: This class represents heuristics for which the number of tasks is fixed at compilation time, but the location of work (tasks, data) is determined and/or changed at run-time.
- Adaptive: Parallel adaptive programs are parallel computations with a dynamically changing set of tasks. Tasks may be created or killed as a function of the load state of the parallel machine. A task is created automatically when a node becomes idle. When a node becomes busy, the task is killed.

2.3.2 Raidl's taxonomy

In order to present a global hybrid metaheuristics classification scheme, Raidl presented a taxonomy that combines different points of view dealt in the ones that are already given in the litterature. In a large part of it we can see aspects from Talbi's taxonomy [248] with the points-of-view from Cotta [72] and Blum et al. [47]. Classification with particular respect to parallel metaheuristics are partly adopted from El-Abd and Kamel [110] and Cotta et al. [71] and with respect to the hybridization of metaheuristics with exact optimization techniques from Puchinger and Raidl [224].

In this taxonomy we start by distinguishing what is hybridized, i.e. which kind of algorithms; (a) different metaheuristic strategies, (b) metaheuristics with certain algorithms specific for the problem we are considering, such as special simulations, or (c) metaheuristics with other more general techniques coming from fields like operations research (OR) and artificial intelligence (AI). Prominent examples for optimization

methods from other fields that have been successfully combined with metaheuristics are exact approaches like branch-and-bound, dynamic programming, and various specific integer linear programming techniques on one side and soft computation techniques like neural networks and fuzzy logic on the other side.

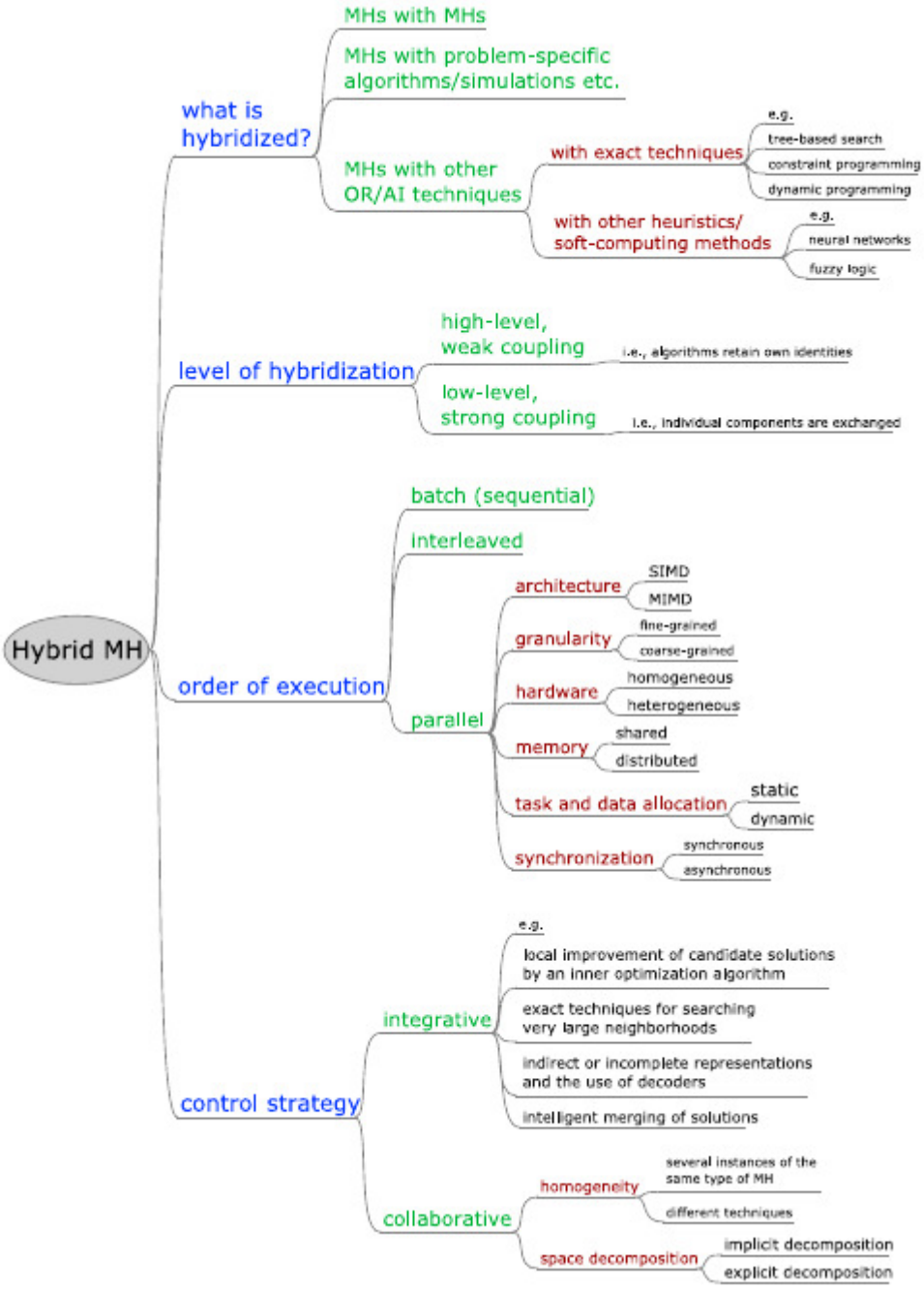

Figure 2.2: Raidl's hybrid metheuristics classification scheme [225]

Beside this differentiation, previous taxonomies of hybrid metaheuristics (i.e. Talbi's taxonomy presented above) primarily distinguish the level (or strength) at which the different algorithms are combined: High-level combinations in principle retain the individual identities of the original algorithms and cooperate over a relatively well defined interface; there is no direct, strong relationship of the internal workings of the algorithms. On the contrary, algorithms in low-level combinations strongly depend on each other – individual components or functions of the algorithms are exchanged.

Another property by which we distinguish hybrid systems is the order of execution. In the batch model, one algorithm is strictly performed after the other, and information is passed only in one direction. An intelligent preprocessing of input data or a postprocessing of the results from another algorithm would fall into this category. Another example are multi-level problems which are solved by considering one level after the other by dedicated optimization algorithms. On the contrary, we have the interleaved and parallel models, in which the algorithms might interact in more sophisticated ways. Parallel metaheuristics are nowadays a large and important research field for their own, see [13]. Detailed classifications of hybrid parallel metaheuristics can be found in [110, 71]. Following general characterizations of parallel algorithms, Raidl distinguishes the architecture (SIMD: single instruction, multiple data streams versus MIMD: multiple instruction, multiple data streams), the granularity of parallelism (fine- versus coarse-grained), the hardware (homogeneous versus heterogeneous), the memory strategy (shared versus distributed memory), the task and data allocation strategy (static versus dynamic), and whether the different tasks are synchronized or run in an asynchronous way.

The taxonomy distinguishes hybrid metaheuristics according to their control strategy aswell. Following [72, 224], there exist integrative (coercive) and collaborative (cooperative) combinations.

In integrative approaches, one algorithm is considered a subordinate, embedded component of another algorithm. This approach is extremely popular.

In collaborative combinations, algorithms exchange information, but are not part of each other. For example, the popular island model [137] for parallelizing evolutionary algorithms falls into this category. We can further classify the traditional island model as a homogeneous approach since several instances of the same metaheuristic are performed. In contrast, Talukdar et al. [249, 250] suggested a heterogeneous framework called asynchronous teams (A-Teams). An A-Team is a problem solving architecture consisting of a collection of agents and memories connected into a strongly cyclic directed network. Each of these agents is an optimization algorithm and can work on

the target problem, on a relaxation of it, i.e. a superclass, or on a subclass. The basic idea of A-Teams is having these agents work asynchronously and autonomously on a set of shared memories. Denzinger and Offermann [92] presented a similar multi-agent based approach for achieving cooperation between search-systems with different search paradigms, such as evolutionary algorithms and branch-and-bound.

In particular in collaborative combinations, a further question is which search spaces are actually explored by the individual algorithms. According to [110] we can distinguish between an implicit decomposition resulting from different initial solutions, different parameter values etc., and an explicit decomposition in which each algorithm works on an explicitly defined subspace. Effectively decomposing large problems is in practice often an issue of crucial importance. Occasionally, problems can be decomposed in very natural ways, but in most cases finding an ideal decomposition into relatively independent parts is difficult. Therefore, (self-)adaptive schemes are sometimes also used.

2.4 Parallel metaheuristics

The main aim of parallelizing a metaheuristic is to solve a large problem more quickly. But, in the context of metaheuristics, parallelism opens other equally interesting possibilities:

- directly testing different values of the parameters at the same time, which enables us to obtain more robust algorithms; and
- exploring the solution in parallel, and therefore more exhaustively, enabling us to obtain solutions that are, if not of better quality, then at least of "better proven" quality.

Classical criteria such as acceleration/speedup or effectiveness/efficiency are not well suited as performance measures in the parallel metaheuristics domain. Explorations carried out sequentially and in parallel can indeed be different. Furthermore, solutions of different quality or even structure can be found. This is the reason why authors in general proposed comparing the quality of solutions with fixed execution times, or comparing execution times with fixed solution quality.

In order to classify parallel metaheuristics, Cung, Le Cun and Roucairol in [75] uses the notion of "walks" introduced by Verhoven and Aarts in [253]:

Theorem 2.1 *(Independent walks theory) If $Q_p(t)$ is the probability of not finding a solution that exceeds the optimum of ϵ in t units of time with p independent walks, and if:*

$$Q_1(t) = e^{-\frac{t}{\lambda}} \ \ with \ \ \lambda \in R^+$$

where Q_1 is the complementary distribution function of a negative exponential distribution then:

$$Q_p(t) = Q_1(pt)$$

In other words, the acceleration can be linear if the probability of finding a suboptimal solution in t units of time is in the form of $Q_1(t)$. The term walk here refers to the passage from solution to solution of a metaheuristic and in this way characterizes its path in the solutions space. This space becomes a graph from the definition of the neighborhoods of each method: if x is a solution and y a solution that belongs to the neighborhood of x, (x, y) is an arc of this graph.

In the parallelization of metaheuristics, the authors mainly distinguish the parallelization of one single walk and that of multiple walks. They only consider, in this context, generic parallelizations, more exactly those that do not depend on a specific application like that of evaluating a solution (see Figure 2.3).

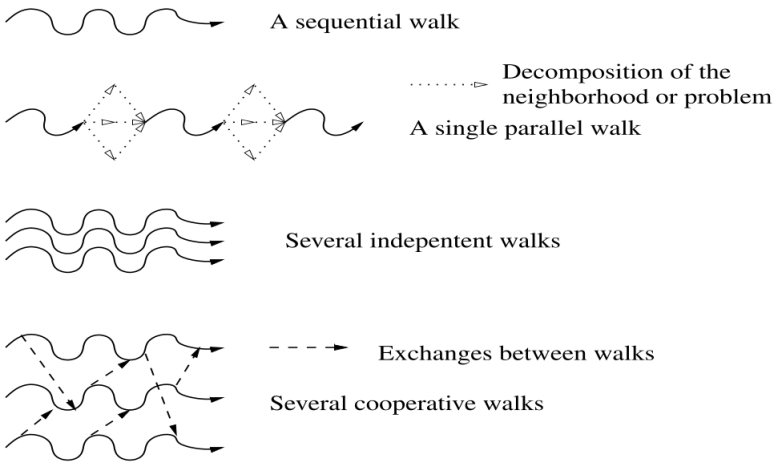

Figure 2.3: Parallelization of metaheuristics [75]

With a single walk, it is generally interesting to parallelize the management of the neighborhood in the case of a local search, or of the population in the case of an evolutionary method (that is the neighborhood model). If the same neighborhood is

conserved (or the same population respectively) with regard to the sequential method, the aim of parallelization becomes the acceleration of the sequential walk using a quicker exploration of the neighborhood (or the population respectively). On the other hand, if a larger neighborhood (or a population respectively) or even several neighborhoods are used in parallel, the walk can become more efficient because it is better guided. Nevertheless, this is about exploiting a fine granularity parallelism (duration of the task before synchronization), which requires a lot of communications between the parallel tasks for their synchronization. It is therefore appropriate to choose only this type of parallelization in the following cases:

- we have a material platform which allows rapid communications;
- we have to deal with extremely complex neighborhoods or cost variation calculations.

Note that for complex combinatorial optimization problems that have hard constraints, certain authors choose to decompose the problem in order to process parts of solutions in parallel. The complete solution is then recomposed at each step. Cung, Le Cun and Roucairol class also this type of parallelization in this single walk category.

With multiple walks, it is appropriate to distinguish the case where walks are independent from the case where they are cooperative. It is this category that has seen the most implementations in the literature in the last decades.

Independent walks are simple to implement; they have good potential accelerations, are robust if different parameters are used, and do not generate any redundant work if the solutions space is partitioned. On the other hand, this parallelization is certainly "slightly simplistic" inasmuch as it is reproducible with a sequential algorithm of the "multi-start" type (we restart the algorithm successively from different initial solutions). No walk may benefit from the experience of the others. Nevertheless, this type of parallelization can serve as a basic example for comparison with the other types.

Cooperative walks are harder to implement following the definition of data that are exchangeable between the walks, but they are potentially much more interesting in terms of "convergence speed" (number of iterations to obtain a good solution) and in terms of solutions quality. The data exchanged can be, for example:

- the values of the parameters for robustness;

- the value of the solutions to induce possible (local) diversifications for walks that are situated in mediocre regions of the solutions space (no solution that gives good values to the optimization criterion);
- the structure of the good solutions to intensify the search around these solutions;
- the frequencies of the (long-term memory) movements and/or the Tabu list that allows global diversifications if these data are shared.

This type of parallelization includes the island model of evolutionary algorithms where the evolution of a subpopulation is seen as a walk.

The major difficulty for cooperative walks lies in the choice of the information to exchange, how it circulates, and the frequency of the exchanges.

Although for a given machine and a given problem a given strategy seems to give good results, it will not necessarily be effective if we change machines or problems. All these parameters therefore make the implementation of parallel metaheuristics difficult. This is why, with regard to exact methods, programming environments have been and are still being developed, which, for an implementation of problem solving, allow the easy changing of types of parallelization.

2.5 Parallelization of exact methods

From now on, we denote for simplicity the Branch-and-Bound by B&B, and the Branch-and-Cut by B&C.

2.5.1 Parallel Branch-and-Bounds literature

A large number of parallel Branch-and-Bound algorithms have been proposed in the literature. Melab in 2005 [201] presented a taxonomy to classify them. This taxonomy is based on the classifications proposed in [76] and [127]. Melab identified four models:

- the multi-parametric parallel model;
- the parallel evaluation of bounds model;
- the parallel evaluation of a bound model;
- the parallel tree exploration model.

2.5.1.1 Multi-parametric parallel model

The multi-parametric parallel model is based on the use of several B&B algorithms, with different parameterization, running in parallel. Several variants of the multi-parametric parallel model may be considered according to the choice of one or more parameter(s) of the B&B algorithm. In [205] for example, Miller and Pekny run multiple B&Bs that differ only by the branching operator. Janakiram and al. in [165] run algorithms that use different selection operators. In [186], each algorithm uses a different upper bound. The idea is that one algorithm uses the best upper bound found while the others use this bound reduced by an ϵ value(ϵ-optimal, where $\epsilon > 0$).

2.5.1.2 Parallel evaluation of bounds model

The parallel evaluation of bounds model allows the execution in parallel of the subproblems bounding (generated by the branching operator). The model does not change the order nor the number of explored subproblems in the parallel B&B algorithm compared to the serial one. [223]

2.5.1.3 Parallel evaluation of a bound model

In this model, the bounding of each subproblem is parallelized. It does not change the semantics of the algorithm because it is similar to the serial version except that the bounding operator is faster. [219]

2.5.1.4 Parallel tree exploration model

The parallel tree exploration model consists of simultaneously exploring several subproblems that define different research subspaces of the initial problem [182, 89, 90, 189]. This means that selection, branching, bounding and pruning operators are executed in parallel synchronously or asynchronously by different processes exploring these subspaces.

Compared to other models, the parallel tree exploration model is more frequently used and is the subject of abundant research for two main reasons. On the one hand, the degree of parallelism of this model may be important when solving large problem

instances justifying the use of multi-core computing or a high performance computing system. On the other hand, the implementation of the model raises several issues that constitute interesting research challenges in parallel computing. Among these issues, we can include the placement and management of the set of subproblems to be solved, the distribution and sharing of the load (generated subproblems), the communication of the best solution found so far, detecting the termination of the algorithm, and fault tolerance.

2.5.2 Parallel Branch-and-Cut's literature

As we know, there was no parallel B&C algorithms for network design problems. In general, there have been few works on the parallelization of this method, and this is due to its irregularity as said before. The first study, we could find, dates back to 1995, in which Ralphs in his thesis [228] presented a black box framework for implementing a parallel B&C for the Vehicle Routing Problem (VRP). More details about this parallel B&C can be found in [229] and an experimental study have been presented by running the algorithm on a *Beawulf Cluster* (a high-performance parallel computing cluster formed by interconnecting inexpensive personal computers).

In 1996, Christof et Reinelt [64] were interested in the parallelization of the search for the violated constraints by assigning to each processor a separation algorithm of the Symmetric Traveling Salesman Problem (STSP) facet family.

In 1998, Bouzgarrou [51] presented in his thesis three strategies to parallelize a B&C for the Traveling Salesman Problem (TSP). Each strategy corresponds to a level of granularity of parallelism. We will present three levels of parallelization: one parallelization with a coarse grain, another with a medium grain and one with a fine grain.

The first strategy of parallelization corresponds to a coarse grain granularity and consists in executing several B&C in parallel, i.e. performing competing explorations. In each execution, one takes different choices of route, initialization or method of resolution. By exchanging information between executions to accelerate one or more explorations. The second strategy takes advantage from the fact that the B&C algorithm is presented as a set of independent tasks. Thus in a parallelization of medium grain, each process explores a part of the search space. In the third strategy, the evaluation of each node of the B&C tree is done in parallel by several processes.

In his implementation, Bouzgarrou used the second strategy putting in practice a master-slave centralized architecture that manages the construction and path of the

B&C tree. The master process initializes the calculation, manages the branch and cut tree path and the division of labor between solver processes. It also maintains the value of the upper bound. And manages the termination of the algorithm. The solver process takes care of the evaluation of the node of the tree assigned to it. It selects the constraint or branch variable. It generates new child nodes, evaluates them and then communicates them to the master process. When the solver no longer has a node to process, it sends a request to the master to ask for work.

After that, Ralphs integrated his framework to the COIN-OR [67] repository under the name *SYMPHONY* [227, 230] and in [226], he discussed the main issues that arise in parallelizing the B&C algorithm for solving mixed-integer linear programs. He explained how designing an efficient parallelization scheme of this method requires careful analysis of various tradeoffs involving the degree of synchronization, the degree of centralized storage of information, and the degree to which information discovered during the algorithm is shared between processors. He presented also computational results obtained solving, using a parallel B&C implemented with SYMPHONY, the Vehicle Routing Problem (VRP) and the Set Partitionning Problem (SPP). These results illustrated the degree to which various sources of parallel overhead affect scalability and demonstrated that properties of the problem class itself dictate the effectiveness of the methodology.

Ralphs et al. in [231, 260] presented the *Abstract Library for Parallel Search* (ALPS) which is a generic framework for implementing parallel tree search algorithms and included it to COIN-OR repository. This library is based on the previous work of Ralphs (SYMPHONY) and provide new features for handling the data, specially the ones needed for implementing data-intensive algorithms such as Branch-Cut-and-Price.

2.6 Motivation of the Thesis

Despite the continuous and significant development of computer's calculation performance, it remains difficult to solve to optimality, within a reasonable amount of time, many combinatorial optimization problems (COPs) for a large scale input data as said before. This remark is all true for SNDPs and more precisely kESNDP and kHNDP. One of the main issues when solving to optimality with Branch-and-Cut or Branch-and-Price algorithms is that solving even the linear relaxation of the considered integer programming formulation can be time consuming for large scale instances. This may prevent the Branch-and-Cut or Branch-and-Price algorithm from a good exploration of the enumeration tree and finding good quality solutions (see for instance [100]).

In this thesis, we address these two issues (excessive CPU time and good exploration of the solutions space) by using both parallel computing and hybridization.

As seen in the literature of both class of problems, the main orientation of the researchers was around studying the problem from a polyhedral aspect. Studies that gave many important results that was able to improve along with hardware development these COPs resolution. What intrigues us on the other hand is whether the use of parallel computing and/or hybridization of methods can make us exceed the limits of instances size that we see in literature.

To adress this problematic, in the first hand, we start by looking to our problems from an approximation point of view. We devise for that a Lagrangian relaxation algorithm for both the kESNDP and kHNDP and use parallel computing in order to reduce the CPU time needed to obtain good lower bound for the problem, a parallel population-based metaheuristic, namely a genetic algorithm, in order to ensure a good exploration of the solutions space and a greedy heuristic for having simply and quickly heuristic solutions of the problems. We then hybridize the three algorithms in order to improve the quality of both the lower and upper bounds. As we will see below, the algorithm takes advantage of the structure of the integer programming formulations we consider for the two class of problems.

In a second hand, we look into the problems from an exact solving point of view. We present and implement two distributed generic algorithms based on the Branch-and-Bound and the Branch-and-Cut methods.

Chapter 3

The k-Edge-Connected Network Design Problem

Contents

In this chapter, we consider the k-Edge-Connected Survivable Network Design Problem (kESNDP) and its variant the Steiner k-Edge-Connected Survivable Network Design Problem (SkESNDP). We propose a parallel hybrid algorithm which aims to produce good solutions for large scale instances of the two problems. Our approach is based on a Lagrangian relaxation of a flow-based integer programming formulation of the problem, a greedy and a genetic algorithms.

In this chapter, we introduce a new parallel resolution approach for the *k-Edge-Connected Survivable Network Design Problem* (kESNDP for short) and its variant problem called *Steiner k-Edge-Connected Survivable Network Design Problem* (SkESNDP for short) based on the hybridization of a Lagrangian relaxation algorithm, a greedy algorithm and a genetic algorithm, used in a parallel computing framework. We test the algorithms within an extensive computational study and compare their efficiency for solving both kESNDP and SkESNDP against CPLEX.

The chapter is organized as follows. In Section 3.1, we present the two integer programming formulations related to kESNDP and SkESNDP. Then, in Section 3.2, we present the parallel hybrid algorithm, its main components. In Section 3.3, we present the computational results for both problem variants.

3.1 Integer programming formulations

3.1.1 kESNDP

Let $G = (V, E)$ be an undirected graph, $D \subseteq V \times V$ a set of demands given by origin-destination pairs $\{s, t\} \in D$ with $s \neq t$, and k a positive integer. The *k-Edge-Connected Survivability Network Design Problem* (kESNDP for short) consists in finding a minimum cost sub-graph of G such that there exist k edge-disjoint st-paths between the terminals of each demand $\{s, t\}$ of D.

The kESNDP can be formulated as a flow-based integer program (see [193]). For each edge $uv \in E$, let x_{uv}, be the $0 - 1$ variable which takes value 1 if the edge uv is in a solution of the problem and 0 otherwise. Also let $\widetilde{G} = (V, A)$ be the directed graph obtained from G by replacing each edge $uv \in E$ by two arcs of the form (u, v) and (v, u) (see Figure 3.1 for an illustration).

Finally, let f_{uv}^{st} be the flow variable associated with every arc (u, v) of $\widetilde{G}$ and a pair $\{s, t\} \in D$. The kESNDP is equivalent to

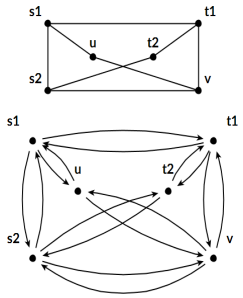

Figure 3.1: Graph transformation for the kESNDP

$$\min \sum_{uv \in E} \omega_{uv} x_{uv}$$

s.t.

$$\sum_{v \in V \setminus \{u\}} f^{st}_{uv} - \sum_{l \in V \setminus \{u\}} f^{st}_{lu} = \left\{ \begin{array}{lll} k, & if & u = s, \\ -k, & if & u = t, \\ 0, & if & u \in V \setminus \{s,t\}, \end{array} \right\}$$

$$\text{for all } u \in V \text{ and } \{s,t\} \in D, \tag{3.1}$$

$$\left. \begin{array}{l} f^{st}_{uv} \\ f^{st}_{vu} \end{array} \right\} \leq x_{uv}, \quad \text{for all } uv \in E \text{ and } \{s,t\} \in D, \tag{3.2}$$

$$f^{st}_{uv}, f^{st}_{vu} \geq 0, \quad \text{for all } uv \in E \text{ and } \{s,t\} \in D, \tag{3.3}$$

$$x_{uv} \leq 1, \quad \text{for all } uv \in E, \tag{3.4}$$

$$x_{uv} \in \{0,1\}, \quad \text{for all } uv \in E, \tag{3.5}$$

$$f^{st}_{uv}, f^{st}_{vu} \in \{0,1\}, \text{ for all } uv \in E, \ \{s,t\} \in D. \tag{3.6}$$

The formulation is called "Undirected Flow Formulation" (UFP for short). Inequalities (3.1) are the *flow conservation constraints*. Inequalities (3.2) are the *linking constraints*. They ensure that if an edge uv is not taken in the solution, then no st-flow can use this edge. Inequalities (3.3) and (3.4) are the trivial inequalities.

One can easily see that the above formulation has a block structure.

Each block "Flow i" corresponds to the flow conservation constraints associated with a demand $\{s,t\} \in D$.

3.1.2 SkESNDP

Let $G = (V, E)$ be an undirected graph, a subset of nodes $S \subseteq V$ called *terminals*, a weight function $\omega : E \to \mathbb{R}$ which associates the weight ω_e with each edge $e \in E$. The *Steiner k-Edge-Connected Survivability Network Design Problem* (SkESNDP for short) is the problem of finding a minimum weight subgraph of G spanning S such that between every two nodes $u, v \in S$, there are at least k edge-disjoint paths.

The Steiner k-Edge-Connected Survivable Network Design Problem (SkESNDP) is a special case of a more general model, introduced by [246] and later called *generalized Steiner problem* by [256].

The SkESNDP is well known to be NP-hard and is a generalization of the Steiner tree problem in which it is required that the nodes of S are spanned by a Steiner tree of minimum weight.

Several works have been done on the Steiner network problem and its variants. Mahjoub and Kerivin [176] have presented a survey of the main variants of survivable network design problems and integer programming formulations as well as some polyhedral descriptions. Also, Magnanti and Raghavan [193] have presented differents types of formulations for a quite more general problem, which includes the SkESNDP. In particular, they have presented several formulations based on flow variables and compare them in terms of LP-bounds.

To introduce the formulation proposed by Magnanti and Raghavan in [193], first, we denote by $D = \{\{s, t\} \mid \text{for all } s, t \in S, \text{with } s \neq t\}$. We also let $d = |D| = |S|(|S| - 1)/2$. For each edge $uv \in E$, let x_{uv} be the $0 - 1$ variable which takes value 1 if the edge uv is in the solution and 0 otherwise. Also let $\widetilde{G} = \{V, A\}$ the directed graph obtained from G by replacing each $uv \in E$ by two arcs (u, v) and (v, u), and for every pair $\{s, t\} \in D$, let f_{uv}^{st} be the flow variable associated with the arc (u, v) of $\widetilde{G}$. The SkESNDP is equivalent to

$$\min \sum_{uv \in E} \omega_{uv} x_{uv}$$

$$\sum_{v \in V \setminus \{u\}} f_{uv}^{st} - \sum_{l \in V \setminus \{u\}} f_{lu}^{st} = \left\{ \begin{array}{rll} k, & if & u = s, \\ -k, & if & u = t, \\ 0, & if & u \in V \setminus \{s,t\}, \end{array} \right\} \text{for all } u \in V \text{ and } \{s,t\} \in D, \tag{3.7}$$

$$\left. \begin{array}{r} f_{uv}^{st} \\ f_{vu}^{st} \end{array} \right\} \leq x_{uv}, \qquad \text{for all } uv \in E \text{ and } \{s,t\} \in D, \tag{3.8}$$

$$f_{uv}^{st}, f_{vu}^{st} \geq 0, \qquad \text{for all } uv \in E \text{ and } \{s,t\} \in D, \tag{3.9}$$

$$x_{uv} \leq 1, \qquad \text{for all } uv \in E, \tag{3.10}$$

$$x_{uv} \in \{0,1\}, \qquad \text{for all } uv \in E, \tag{3.11}$$

$$f_{uv}^{st} \in \{0,1\}, \qquad \text{for all } uv \in E, \{s,t\} \in D. \tag{3.12}$$

This formulation is called *Undirected Flow Formulation*. Inequalities (3.7) are the *flow conservation constraints*. Inequalities (3.8) are the *linking constraints*. They ensure that if an edge uv is not taken in the solution, then no st-flow can use this edge. Inequalities (3.9) and (3.10) are the trivial inequalities.

3.2 Parallel hybrid optimization algorithm

Our algorithm relies on the usage of parallel computing for solving the kESNDP. Namely, we devise a heuristic for the problem based on three algorithms

- *a greedy heuristic (SH)*;
- *a Lagrangian relaxation algorithm (RLA)*;
- *a genetic algorithm (GA)*.

For our purpose, we run these three algorithms in a parallel computing framework. Also, as we will see, each iteration of each algorithm is used to improve the other algorithms, and hence, improve the whole algorithm. Moreover, we solve the Lagrangian relaxation algorithm using parallel computing.

We describe, in the remain of the section, each algorithm in the following and the communication between them.

3.2.1 The greedy successive heuristic

Constructive heuristics are typically the fastest approximate methods. They generate solutions from scratch by adding opportunely defined solution components to an initially empty partial solution. This is done until a solution is complete or other stopping criteria are satisfied. A well-known class of constructive heuristics are greedy heuristics. They make use of a weighting function that assigns at each construction step a positive weight to each feasible solution component. The solution component with the highest weight is chosen at each construction step to extend the current partial solution.

Our greedy algorithm (SH) for solving the kESNDP and the SkESNDP consists in computing a series of minimum cost st-flows in the graph $\widetilde{G}$, for all $\{s,t\} \in D$.

First, we sort the pairs of D in an arbitrary order, say $\{s_1,t_1\},...,\{s_d,t_d\}$. We start with the pair $\{s_1,t_1\}$, and compute a minimum cost s_1t_1-flow in $\widetilde{G}$, where all the arcs have capacity 1 and both arcs (u,v) and (v,u) are cost ω_{uv}, for all $uv \in E$. Let A_1 be the set of arcs that have a s_1t_1-flow value of 1, and E_1 be the set of edges of G corresponding to the arcs of A_1. Then we choose the pair $\{s_2,t_2\}$, fix to ω_{uv} the cost of the arcs (u,v) and (v,u) for all $uv \in E \setminus E_1$ and fix to 0 the cost of the arcs (u,v) and (v,u), for all $uv \in E_1$, and compute a minimum cost s_2t_2-flow. We build the arc set A_2 and the edge set E_2, as before A_1 and E_1, and so on until all the pairs $\{s_i,t_i\}$ have been explored. Finally, we build a solution of the kESNDP by considering the edges of $E_1 \cup E_2 \cup ... \cup E_d$.

3.2.2 The Lagrangian relaxation algorithm

3.2.2.1 Basics

Consider the following general zero-one problem (written in matrix notation):

$$Problem\ (P) \begin{cases} \quad min & cx \\ Ax & \geq & b \\ Bx & \geq & d \\ \quad x & \in & (0,1) \end{cases}$$

Note that although we deal with zero-one integer programs the approach presented is equally applicable both to pure (general) integer programs and to mixed-integer programs.

A well known way to generate a lower bound on the optimal solution to problem (P) is via the linear programming relaxation. This entails replacing the integrality constraint $x \in (0,1)^E$ to give the following linear program:

$$\begin{array}{rcl} min & cx & \\ Ax & \geq & b \\ Bx & \geq & d \\ 0 \leq & x & \leq 1 \end{array}$$

This linear program can be solved exactly using standard algorithm and the solution value obtained gives a lower bound on the optimal solution to the original problem (problem P).

In many cases however solving the linear programming relaxation of P is impracticable, typically because P involves a large (often extremely large) number of variables and/or constraints. We therefore need alternative techniques for generating lower bounds.

Lagrangian relaxation was developed by Held and Karp in the early 1970's [154] and is today an indispensable technique for generating lower bounds for use in algorithms to solve combinatorial optimisation problems. Note that a more detailed presentation of the Lagrangian technique can be found in [34, 237].

We define the lagrangian relaxation of problem P with respect to the constraint set $Ax \geq b$ by introducing a lagrange multiplier vector $\lambda \geq 0$ which is attached to this constraint set and brought into objective function to give:

$$\begin{array}{rcccl} min & cx & + & \lambda(b - Ax) & \\ & Bx & \geq & d & \\ & x & \in & (0,1) & \end{array}$$

i.e. what we have done here is:

a) to have chosen some set of constraints in the problem for relaxation; and

b) attached lagrange multipliers to these constraints in order to bring them into the objective function.

The key point is that the program we are left with after lagrangian relaxation, for any $\lambda \geq 0$, gives a lower bound on the optimal solution to the original problem P. In

fact, since as $\lambda \geq 0$ and $(b - Ax) \leq 0$ we are merely adding a term which is ≤ 0 to the objective function, and since removing a set of constraints from a minimisation problem can only reduce the objective function value.

The program after lagrangian relaxation, namely:

$$P(\lambda) \begin{cases} L(\lambda) & = & min & (c - \lambda A)x + \lambda b \\ Bx & \geq & d & \\ x & \in & (0,1) & \end{cases}$$

can be called the *lagrangian lower bound program* ($LLBP$) since, as shown above, it provides a lower bound on the optimal solution to the original problem P for any $\lambda \geq 0$.

Besides the strategic issue of why we choose to relax the set of constraints $Ax \geq b$ instead of the set $Bx \geq d$, a key issue highlighted by the above lagrangian relaxation is a tactical issue, namely how can we find numerical values for the multipliers.

In particular note here that we are interested in finding the values for the multipliers that give the maximum lower bound, i.e. the lower bound that is as close as possible to value of the optimal integer solution. This involves finding multipliers which correspond to:

$$L^* = max \{L(\lambda), \lambda \geq 0\}$$

This program is called the *lagrangian dual program.*

There are two basic approaches to deciding values for the lagrange multipliers (λ_i):

a) subgradient optimisation [155]; and

b) multiplier adjustment [114].

In what follows, we introduce a brief presentation of the subgradient optimisation. However, more details about the two techniques can be found in [34].

The subgradient optimisation is an iterative procedure which, from initial set of multipliers, involves generating further lagrange multipliers in a systematic fashion. It can be viewed as a procedure which attempts to maximise the lower bound value obtained from LLBP by suitable choice of multipliers.

Switching from matrix notation to summation notation, so that the relaxed constraints are $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$ $(i = 1, ..., m)$, the basic subgradient optimisation iterative procedure is as follows:

1) Let π_i be a user decided parameter satisfying $0 < \pi_i \leq 2$. Initialise Z_{UB} (e.g. from some heuristic for the problem). Decide upon an initial set (λ_i) of multipliers.

2) Solve LLBP with the current set (λ_i) of multipliers, to get a solution (X_j) of value Z_{LB} (not necessarily feasible for our initial program).

3) Define subgradients γ_i for the relaxed constraints, evaluated at the current solution, by:

$$\gamma_i = \frac{\partial L(x,\lambda)}{\partial \lambda} = b_i - \sum_{j=1}^{n} a_{ij} X_j \quad i = 1, ..., m$$

4) Define a (scalar) step size ρ_i by $\rho_i = \pi_i \frac{(Z_{UB} - Z_{LB})}{\sum_{j=1}^{n} (\gamma_i)^2}$
This step size depends upon the gap between the current lower bound (Z_{LB}) and the upper bound (Z_{UB}) and the user defined parameter π_i (more of which below) with the $\sum_{j=1}^{n} (\gamma_i)^2$ factor being a scaling factor.

5) Update λ_i using $\lambda_i = max(0, \lambda_i + \rho_i \gamma_i) \quad i = 1, ..., m$ and go to 2) to resolve LLBP with this new set of multipliers.

As currently set out the above iterative procedure would never terminate. In fact we introduce a termination rule based upon either:

(a) limiting the number of iterations that can be done; or

(b) the value of π (reducing π during the course of the procedure and terminating when π is small)

Ideally the optimal value of the lagrangian dual program (a maximisation program) is equal to the optimal value of the original zero-one integer program (a minimisation problem). If the two programs do not have optimal values which are equal then a duality gap is said to exist, the size of which is measured by the (relative) difference between the two optimal values.

3.2.2.2 A Lagrangian relaxation algorithm for the kESNDP

In our framework, we consider the Undirected Flow Formulation and relax the linking constraints (3.2). Let λ_{uv}^{st}, for all $\{s,t\} \in D$ and all $(u,v) \in A$, be the Lagrangian multiplier associated with constraints (3.2). This yields the following problem

$$\min \sum_{uv\in E}\left[\omega_{uv}-\left(\sum_{\{s,t\}\in D}(\lambda_{uv}^{st}+\lambda_{vu}^{st})\right)\right]x_{uv}+\sum_{\{s,t\}\in D}\sum_{uv\in E}(\lambda_{uv}^{st}f_{uv}^{st}+\lambda_{vu}^{st}f_{vu}^{st})$$

$$\sum_{v\in V\setminus\{u\}}f_{uv}^{st}-\sum_{l\in V\setminus\{u\}}f_{lu}^{st}=\left\{\begin{array}{rll} k, & if & u=s,\\ -k, & if & u=t,\\ 0, & if & u\in V\setminus\{s,t\},\end{array}\right\}\text{for all } u\in V \text{ and } \{s,t\}\in D,$$

$$\begin{array}{rl} 0\le f_{uv}^{st},f_{vu}^{st}\le 1, & \text{for all } uv\in E,\ \{s,t\}\in D,\\ x_{uv}\le 1, & \text{for all } uv\in E,\\ x_{uv}\in\{0,1\}, & \text{for all } uv\in E,\\ f_{uv}^{st}\in\{0,1\}, & \text{for all } uv\in E, \{s,t\}\in D.\end{array}$$

One can easily see that if $(\overline{f}^{st},\ \{s,t\}\in D)$ is an optimal solution for problem (LR'), then solution $(\overline{x},\overline{f}^{st},\ \{s,t\}\in D)$ where

$$\overline{x}(uv)=\left\{\begin{array}{ll} 1 & \text{if } \omega_{uv}-\displaystyle\sum_{\{s,t\}\in D}(\lambda_{uv}^{st}+\lambda_{vu}^{st})<0,\\ 0 & \text{otherwise,}\end{array}\right. ,\ \text{for all } uv\in E,$$

is optimal for problem (LR).

Also, it is not hard to see that problem (LR') consists in $|D|$ independent minimum cost st-flow problems in graph $\widetilde{G}$. Thus problem (LR') can be solved by using any combinatorial algorithm solving the minimum cost flow problem. The reader can refer to [12] for more details on minimum cost flow problems and the associated algorithms. Moreover, since the minimum cost flow problems of (LR') are independent, we solve (LR') in a parallel multithreaded fashion. Thus, we can solve (LR) in time $O(MF)$ when using $|D|$ processors, where $O(MF)$ is the runtime for solving a minimum cost st-flow problem.

An issue to address in the Lagrangian relaxation algorithm is how the Lagrange multipliers, λ_{uv}^{st}, $(u,v)\in A$ and $\{s,t\}\in D$, are updated. For this, we use the so-called subgradient method explained before. Let, at iteration k of the Lagrangian relaxation algorithm, $\lambda_{uv,k}^{st}$ be the vector of the current Lagrange multipliers, $(\overline{x},\overline{f}^{st},\ \{s,t\}\in D)$ an optimal solution of problem (LR), z_k the optimal value of (LR) and Z_{UB} an upper bound of the optimal solution of the original problem (UFP). Then, the Lagrange multipliers at iteration $k+1$ are given by

$$\lambda_{uv,k+1}^{st}=\max\{0,\lambda_{uv,k}^{st}-\rho_k\gamma_{uv,k}^{st}\}$$

where

$$\left.\begin{array}{r} \gamma^{st}_{uv,k} = \overline{x}^{*}_{uv} - \overline{f}^{st}_{uv} \\ \gamma^{st}_{uv,k} = \overline{x}^{*}_{uv} - \overline{f}^{st}_{vu} \end{array}\right\}, \text{for all } uv \in E,$$
$$\rho_k = \frac{1}{2^k} \frac{Z_{UB} - z_k}{\| \gamma_k \|^2}.$$

The upper bound Z_{UB} can be obtained by a heuristic running independently from the Lagrangian relaxation algorithm. In our case, we produce a feasible solution to the kESNDP at each iteration of the Lagrangian relaxation algorithm and choose Z_{UB} as the best value among all the solutions thus obtained. A feasible solution, say $\overline{y} \in \mathbb{R}^E$, to kESNDP can be obtained from the optimal solution of problem (LR) by chosing $\overline{y}_{uv} = \max\{\overline{f}^{st}_{uv}, \overline{f}^{st}_{vu}, \text{ for all } \{s,t\} \in D\}$. Clearly, the solution $(\overline{y}, \overline{f}^{st}, \ \{s,t\} \in D)$ satisfies constraints (3.1)-(3.6).

3.2.2.3 A Lagrangian relaxation algorithm for the SkESNDP [99]

For the SkESNDP, the Lagrangian relaxation is obtained by relaxing the linking constraints (3.8). We denote by λ^{st}_{uv}, the Lagrange multipliers associated with the linking constraints (3.8), for all $\{s,t\} \in D$ and $(u,v) \in A$. The Lagrangian relaxation thus obtained, denoted by (LR), is given by

$$\min \quad \sum_{uv \in E} \left[\omega_{uv} - \sum_{\{s,t\} \in D} \left(\lambda^{st}_{uv} + \lambda^{st}_{vu} \right) \right] x_{uv} + \sum_{\{s,t\} \in S} \sum_{uv \in E} \left(\lambda^{st}_{uv} f^{st}_{uv} + \lambda^{st}_{vu} f^{st}_{vu} \right)$$

s.t.

$$\sum_{v \in V} f^{st}_{uv} - \sum_{l \in V} f^{st}_{ul} = \left\{ \begin{array}{ll} k, & if\ u = s, \\ -k, & if\ u = t, \\ 0, & if\ u \in V \setminus \{s,t\}, \end{array} \right\}, \text{for all } u \in V \text{and } \{s,t\} \in D,$$

$$0 \leq f^{st}_{uv}, f^{st}_{vu} \leq 1, \text{ for all } uv \in E, \{s,t\} \in D,$$

$$0 \leq x_{uv} \leq 1, \qquad \text{for all } (u,v) \in E,$$

$$x_{uv} \in \mathbb{Z}, \qquad \text{for all } (u,v) \in E,$$

$$f^{st}_{uv} \in \mathbb{Z}, \qquad \text{for all } (u,v) \in A, \ \{s,t\} \in D.$$

Observe that, apart the objective function, the variable x appears in problem (LR) only in the trivial constraints $0 \leq x_{uv} \leq 1$. Thus, solving problem (LR) reduces to solving the problem (LR') below

$$\min \sum_{\{s,t\} \in S} \sum_{uv \in E} \left(\lambda_{uv}^{st} f_{uv}^{st} + \lambda_{vu}^{st} f_{vu}^{st} \right)$$

s.t.

$$\sum_{v \in V} f_{uv}^{st} - \sum_{l \in V} f_{ul}^{st} = \left\{ \begin{array}{rl} k, & if\ u = s, \\ -k, & if\ u = t, \\ 0, & if\ u \in V \setminus \{s,t\}, \end{array} \right\}, \text{for all } u \in V \text{and } \{s,t\} \in D,$$

$$0 \leq f_{uv}^{st}, f_{vu}^{st} \leq 1, \text{ for all } uv \in E,\ \{s,t\} \in D,$$

$$f_{uv}^{st} \in \mathbb{Z}, \qquad \text{for all } (u,v) \in A,\ \{s,t\} \in D.$$

One can see that solving this relaxation consists in solving d independant minimum cost st-flow problems in the graph $\widetilde{G}$, for all $\{s,t\} \in D$. In fact, we can see that solving the minimum cost st-flow problems gives the optimal value of the flow variables f_{uuv}^{st}, and for the variables x_{uv}, the optimal values are $x_{uv} = 1$ if $\omega_{uv} - \sum_{\{s,t\} \in D} (\lambda_{uv}^{st} + \lambda_{vu}^{st}) > 0$ and 0 otherwise.

Since the minimum cost st-flow problems are independant, they can be solved in parallel using d processors.

For the Lagrangian relaxation algorithm, the Lagrangian multipliers are updated using the subgradient method.

Also, notice that each iteration of the Lagrangian relaxation algorithm produces a feasible solution. The solution is obtained by considering all the edges of G corresponding to the arcs of $\widetilde{G}$ that get at least once a flow value of 1 during the resolution of the minimum cost st-flow problems. As we will see, the solution thus obtained are used as input of the genetic algorithm, which is described below.

Finally, the Lagrangian relaxation algorithm stops either when a CPU time of 2 hours is reached or the algorithm processes 2000 iterations or after 100 iterations without improving the best known upper bound. The algorithm returns the best lower and upper bounds obtained.

3.2.3 The genetic algorithm

Now we turn our attention to the genetic algorithm. We recall that the genetic algorithm consists in considering a set of solutions, feasible or not for the kESNDP, in order to produce one or more new feasible solutions. The set of solutions is called the *population*. The algorithm randomly chooses two solutions, called *parents*, in the population and combine them to produce one or more new solutions, called *children*. For a general combinatorial optimization problem, the children solutions may not be feasible for the problem. In this case, the algorithm tries to transform them into feasible solutions and put them into the population. In our case, we will see that the children solutions we build are feasible for the kESNDP. The designing of a genetic algorithm takes into account the following issues

- solution encoding and evaluation,
- the parent selection,
- the crossover (how the parent are combined),
- the population management,
- the stopping criterion.

For more details on genetic algorithms, the reader can refer to [248]. In the remainder of this section, we discuss these issues for our genetic algorithm for the kESNDP.

3.2.3.1 Encoding and evaluation of a solution

A solution of the kESNDP can be represented in different ways, but in our algorithm, we represent a solution by a set of $0-1$ vectors $(\overline{f}^{s_1t_1}, ..., \overline{f}^{s_dt_d})$, where $\overline{f}^{s_it_i}$ is a flow vector associated with demand s_it_i.

The evaluation of a solution encoded by $(\overline{f}^{s_1t_1}, ..., \overline{f}^{s_dt_d})$ consists in giving the weight of the subgraph of G corresponding to that solution. The weight of such a solution is given by $Z = \sum_{uv \in E} \omega_{uv}\overline{x}_{uv}$, where

$$\overline{x}_{uv} = \max\{\overline{f}^{st}_{uv}, \overline{f}^{st}_{vu}, \text{ for all } \{s,t\} \in D\}.$$

3.2.3.2 Crossover and reproduction

The selection is made randomly according to a specific probability distribution scale. We start by dividing the population into five categories (A, B, C, D and E) based on the social ranking (fitness value), and then we select $\frac{1}{10}$ of the population to be reproduced knowing that the probability that a parent is chosen from the A social category for example is equal to 67%, and respectively to the B, C, D and E are equal to 19%, 10%, 3% and 1%.

The generation of children solutions is done in the following way. For two solutions, say $P_1 = (\overline{f}^{s_1t_1}, ..., \overline{f}^{s_dt_d})$ and $P_2 = (\overline{g}^{s_1t_1}, ..., \overline{g}^{s_dt_d})$, we randomly choose two integers a and b with $2 \leq a < b \leq d$. Then, we cross P_1 and P_2 according to integers a and b, and produce two solutions C_1 and C_2 such that

$$C_1 = (\overline{f}^{s_1t_1}, ..., \overline{f}^{s_{a-1}t_{a-1}}, \overline{g}^{s_at_a}, ..., \overline{g}^{s_{b-1}t_{b-1}}, \overline{f}^{s_bt_b}, ..., \overline{f}^{s_dt_d})$$

and

$$C_2 = (\overline{g}^{s_1t_1}, ..., \overline{g}^{s_{a-1}t_{a-1}}, \overline{f}^{s_at_a}, ..., \overline{f}^{s_{b-1}t_{b-1}}, \overline{g}^{s_bt_b}, ..., \overline{g}^{s_dt_d}).$$

Clearly, C_1 and C_2 are feasible for the kESNDP if P_1 and P_2 are feasible for the kESNDP.

The children generation phase is done by applying the above procedure to $0.1N_{pool}$ pairs of solutions (P_1, P_2), where N_{pool} is the number of elements into the pool before the generation of the new solutions.

3.2.3.3 Population management

The pool of solutions is intialized with all the feasible solutions produced by the greedy algorithm which is described in Section 3.2.1. During the algorithm, we ensure that the pool of solutions contains no more than 100 solutions. The solutions are sorted in increasing order w.r.t. their evaluation. For simplicity, we denote by N^i_{pool} the number of solutions in the pool after the generation of the new solutions at iteration i. In the following we discuss the cases where the genetic algorithm runs simultaneously with the Lagrangian relaxation and greedy algorithms and when it is not the case.

We discuss first the case where the genetic algorithm is running simultaneously with the Lagrangian relaxation and greedy algorithms. In this case, for every iteration $i \geq 1$,

we remove from the pool the $N^i_{pool} - 100$ worst solutions (starting with the duplicates), if $N^i_{pool} \geq 100$. Otherwise, we do not remove any solution.

Now we consider the case where the genetic algorithm is not running simultaneously with the Lagrangian relaxation and greedy algorithms. This is the case when both, the Lagrangian relaxation and greedy algorithms, are terminated or when the genetic algorithm is running separately from the hybridization. In this case, for every iteration i such that $1 \leq i \leq 200$, if $N^i_{pool} \geq 100$, then we remove the $N^i_{pool} - 100$ worst solutions. When $i > 200$, if $N^i_{pool} \geq 20$, we remove from the pool the $(N^i_{pool} - N^{i-1}_{pool}) + 5\%N^{i-1}_{pool}$ worst solutions. If $N^i_{pool} < 20$, then we remove the $(N^i_{pool} - N^{i-1}_{pool}) + 1$ worst solutions.

Note that this management strategy of the pool guarantees that the number of solutions in the pool slowly decreases until it remains one solution in the pool, and this, even in the hybridization or not.

3.2.3.4 Stopping criterion

The genetic algorithm stops either when the CPU time reaches 2 hours or the pool of solutions contains only one solution. Note that in both cases, the first solution of the pool is the best solution obtained by the algorithm.

3.2.4 The hybridization and parallelization scheme

Now we present the hybridization and parallelization scheme of the overall algorithm. For simplicity, we denote the Lagrangian relaxation algorithm by LRA, the genetic algorithm by GA and the greedy algorithm by SH. The parallel hybrid algorithm will be denoted by PHA.

We remark that in PHA, the three algorithms LRA, GA and SH run in parallel. We also hybridize the three algorithms in the following way. First, the solutions generated by algorithms SH and LRA are introduced in the pool of solutions of GA. Also, at the begining of PHA, the global upper bound Z_{UB} is set to ∞. Then, each time LRA, SH and GA generate a feasible solution, the value Z of this solution is compared to Z_{UB}. If $Z < Z_{UB}$, then the best upper bound Z_{UB} is updated with the value of Z. Note that

the value Z_{UB} is used by LRA to update the Lagrange multipliers.

We also ensure that GA continues running after LRA and SH are terminated. This is done in order to give to GA enough time to process the solutions of LRA and SH.

Finally, notice that toghether with the global upper bound Z_{UB}, the Lagrangian relaxation algorithm also produces a lower bound of the optimal solution of the SkESNDP.

The main operations in PHA are summarized below, in Algorithm 7, while Figure 3.2 describes the communication scheme of PHA.

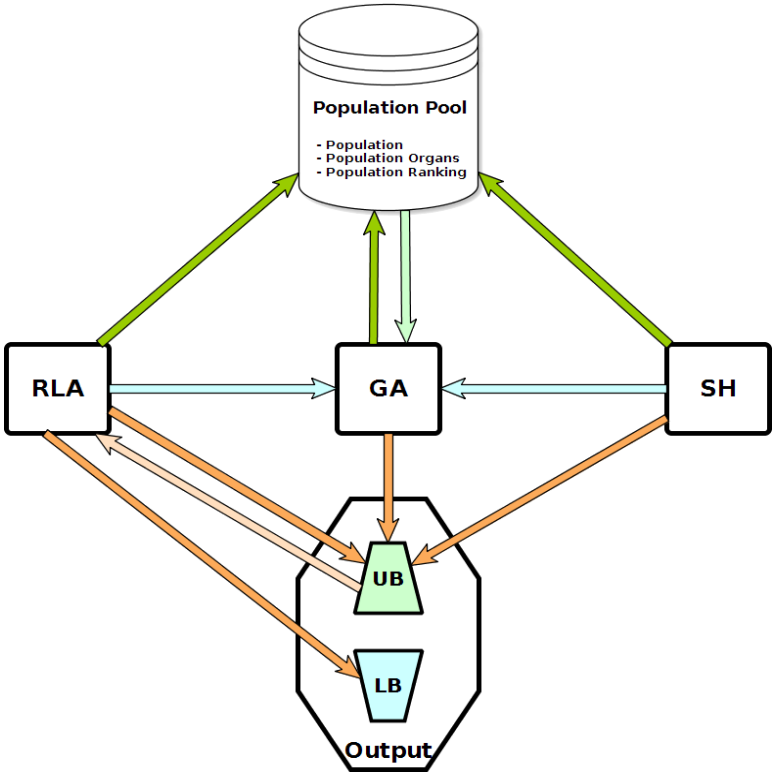

Figure 3.2: PHA communication scheme

3.3 Experimental Study

In this section, we present the computational experiments we have conducted for the kESNDP and the SkESNDP. The aim is to show the efficiency of our algorithm in

Algorithm 7: Algorithm PHA for the kESNDP.

Data: An undirected graph $G = (V, E)$, the demand set D, a positive integer $k \geq 1$
Result: A lower and upper bounds of the optimal solution of the kESNDP
begin
  $Z_{UB} \leftarrow \infty$;
  Execute SH, LRA et GA in a parallel;

  SH:
  Computes $|D|$ arbitrary ordering of the demands;
  for *each ordering on the demands* **do**
    Computing, using a series of minimum cost flow computation, a feasible solution according to that ordering;
    Add the new solution obtained into the pool of population. Let Z_{SH} be its value;
    if $Z_{SH} < Z_{UB}$ **then**
      $Z_{UB} \leftarrow Z_{SH}$;
    end
  end
  Inform GA that is has ended;

  LRA:
  for *each iteration* **do**
    Compute the Lagrange multipliers using Z_{UB} and the current solution of problem (LR). Let z be the optimal value of (LR);
    Compute a feasible solution from the solution of problem (LR). Let Z_{LRA} be its value;
    Add this solution into the pool;
    if $Z_{LRA} < Z_{UB}$ **then**
      $Z_{UB} \leftarrow Z_{RLA}$;
    end
    if $z > Z_{LB}$ **then**
      $Z_{LB} \leftarrow z$;
    end
  end
  Inform GA that is has ended;

  GA:
  for *each iteration* **do**
    Sort the solutions of the pool by increasing order w.r.t. to their weight;
    Randomly choose several pairs of solutions from the pool and combine the solutions of each pair in order to generate new solutions;
    for *every new solution* **do**
      Let Z_{GA} be the value of the solution;
      if $Z_{GA} < Z_{UB}$ **then**
        $Z_{UB} \leftarrow Z_{GA}$;
      end
      Add the solution to the pool of solutions;
    end
    Delete the worst solutions from the pool (according to the pool management strategy);
  end

  return (Z_{LB}, Z_{UB});
end

producing solutions of good quality for the problem, and this, in a relatively short computation time. To do this, we first compare the results obtained, for several instances, by algorithm PHA against those obtained by solving the flow formulation with CPLEX. Then, we compare each algorithm SH, LRA and GA against the overall PHA algorithm for solving the kESNDP and SkESNDP. Notice that since one of the components of parallel hybrid algorithm have stochastic components (e.g. the GA), each PHA entry represents the mean of 5 runs on the same instance.

All the algorithms have been implemented in C++ using LEMON graph structures [8] and we have used CPLEX [1] for solving the undirected flow formulation of the kESNDP. The experiments have been conducted on a computer equiped with an Intel i7 processor at 2.5 Ghz with 8 Gb of RAM, running under Linux. We have set the maximum CPU time of all the algorithms, including CPLEX, to 2 hours.

We have used OpenMP library and the C++ inner parallel library `std::thread` in order to manage the multithreading parallel computations. Also, in order to avoid inconsistencies in the solutions pool, we use the C++ library `std::mutex` for the shared memory management.

3.3.1 kESNDP

The test problems have composed of graphs from the [2] library, that are complete Euclidean graphs. For the demands, we have considered single-source multi-destination demand set (called rooted demands) and multi-source multi-destination demand set (arbitrary demands). The number of nodes of the graphs are from 30 to 318 while the number of demands varies from 3 to 15, for the rooted case, and from 10 to 159 for arbitrary demands. We have also solved each instance with connectivity requirement $k = 3$. Each instance is described by its name followed by the number of nodes of the graph and the number of demands. When the demands are rooted, the number of demands is preceded by “r" while arbitrary demands are indicated by “a" before the number of demands. For example, berlin30-r10 denotes an instance composed of a graph from TSPLIB with 30 nodes and 10 rooted demands, and st70-a35 denotes an instances composed of a graph from TSPLIB with 70 nodes and 35 arbitrary demands.

We have also implemented the Undirected Flow Formulation using CPLEX 12.6 concert technology library and have solved the problem for the same instances. Tables 3.1 and 3.2 show the results for the kESNDP with $k = 3$ by both PHA and CPLEX for rooted demands and arbitrary demands respectively. The entries of each tables are

$\|V\|$:	number of nodes of the graph,
$\|D\|$:	number of demands,
UB:	best upper bound achieved by PHA (resp. CPLEX),
LB:	best lower bound achieved by PHA (resp. CPLEX),
Gap:	relative error between the best upper and lower bounds achieved by PHA (resp. CPLEX),
CPU:	total CPU time in hours:min:sec achieved by PHA (resp. CPLEX).

Remark that for some instances, CPLEX has not been able to solve even the linear relaxation after the maximum CPU time (2 hours). The results for these instances are indicated with "–".

From Table 3.1, we can observe that for the rooted instances, 2 instances over 22 have been solved to optimality by CPLEX while PHA produces an upper bound of the optimal solution for all the instances. Also, for 9 instances, CPLEX produces a better feasible solution than that obtained by PHA. However, for all the other instances, the upper bound produced by PHA is better than that obtained by CPLEX after the maximum CPU time. For example, for instances st70-r35 and kroA100-r50, PHA produces an upper bound of 1265 and 47454, respectively, while CPLEX produces an upper bound of 1278 and 49323. We have also compared the lower bound achieved by PHA with that obtained by CPLEX. For several instances (12 over 22), the lower bound obtained by CPLEX after the maximum CPU time is better than that obtained by PHA. This can be explained by the fact that CPLEX, during the resolution process, adds several valid inequalities, such as Gomory cuts and General Upper Bound inequalities, which allows strengthning the linear relaxation of the problem and improve the quality of the lower bound.

For the arbitrary instances (see Table 3.2), the observations are almost the same. PHA produces better upper bounds for 11 instances over 20. PHA is even able to produce both lower and upper bounds for 5 instances where CPLEX has not been able to solve the linear relaxation of the problem. Also, for 9 instances, PHA produces a better lower bound than that obtained by CPLEX. For the other instances, the lower bound achieved by CPLEX is better than that obtained PHA.

Finally, we compare CPLEX and PHA in terms of CPU time. We clearly see, from Tables 3.1 and 3.2, that except 2 instances with rooted demands instances and with

Table 3.1: Results for PHA and CPLEX for the kESNDP and rooted demands with $k = 3$.

Instances			PHA				CPLEX			
name	$\|V\|$	$\|D\|$	*UB*	*LB*	*Gap*	*CPU*	*UB*	*LB*	*Gap*	*CPU*
berlin	30	10	7168	6072.29	15.29	00:00:11	6241	6241	0	00:43:55
berlin	30	15	10970	7856.13	28.39	00:00:17	7982	7828	1.93	02:00:00
berlin	30	20	12173	8391.95	31.06	00:00:27	8510	8510	0	00:56:08
berlin	30	25	12805	8832.3	31.02	00:00:36	9040	9014.5	0.28	02:00:00
berlin	52	10	6815	4740.6	30.44	00:00:45	5399	4959.58	8.14	02:00:00
berlin	52	15	9871	6821.52	30.89	00:01:12	7505	6680.3	10.99	02:00:00
berlin	52	26	13033	7639.51	41.38	00:01:58	9270	8223.08	11.29	02:00:00
st	70	15	761	523.64	31.19	00:01:21	607	438.95	27.69	02:00:00
st	70	26	1077	707.24	34.33	00:02:24	928	612.011	34.05	02:00:00
st	70	35	1265	805.39	36.33	00:03:22	1278	707.5	44.64	02:00:00
kroA	100	20	26677	11864.8	55.52	00:05:52	22637	12354.3	45.42	02:00:00
kroA	100	35	39378	16182.3	58.91	00:10:12	32738	19189	41.39	02:00:00
kroA	100	50	47454	18875.4	60.22	00:15:03	49323	23501.4	52.35	02:00:00
kroA	150	30	36340	11596.3	68.09	00:19:09	43987	16106.5	63.38	02:00:00
kroA	150	50	47764	15774.2	66.97	00:33:04	64280	22374.3	65.19	02:00:00
kroA	150	75	57909	17742.1	69.36	00:50:48	418670	28031.4	93.3	02:00:00
kroA	200	40	41724	12080.4	71.05	00:45:44	65762	17020	74.12	02:00:00
kroA	200	75	59482	16096.8	72.94	01:28:01	–	–	–	–
kroA	200	100	69462	18792.2	72.95	02:00:00	–	–	–	–
lin	318	61	26723	6068.56	77.29	02:00:00	–	–	–	–
lin	318	111	55000	8702.16	84.18	02:00:00	–	–	–	–
lin	318	159	71464	8719.74	87.8	02:00:00	–	–	–	–

arbitrary demands, CPLEX reaches the maximum CPU time for all the instances. On the contrary, the CPU time achieved by PHA is relatively small for most of the instances. Indeed, the CPU time is less than 6 minutes for 50% of the rooted demand instances and less than 7 minutes for 45% of the arbitrary demand instances. Only 4 instances for both the rooted demands and arbitrary demands, have reached the maximum CPU time. This, together with the above observations, shows that PHA is able to obtain better solutions than CPLEX, and this, in quite short CPU time.

Now we turn our attention to the efficiency of PHA w.r.t its components, that are LRA, GA and SH. The aim is to see if each algorithm taken separately is more efficient than the hybridization or not. For this, we compare the results obtained by LRA, GA

Table 3.2: Results for PHA and CPLEX for the kESNDP and arbitrary demands with $k = 3$.

Instances			PHA				CPLEX			
name	$\|V\|$	$\|D\|$	*UB*	*LB*	*Gap*	*CPU*	*UB*	*LB*	*Gap*	*CPU*
berlin	30	10	10422	8216.0	21.17	00:00:15	9276	9276.0	0	00:00:08
berlin	30	15	12385	9716.2	21.55	00:00:24	10749	10655.2	0.87	02:00:00
berlin	52	10	10553	7772.5	26.35	00:00:42	8508	8256.9	2.95	02:00:00
berlin	52	15	13588	9141.0	32.73	00:01:23	10270	9870.4	3.89	02:00:00
berlin	52	20	15603	10347.5	33.68	00:01:37	11499	11499.0	0	00:30:04
st	70	15	1176	678.3	42.32	00:01:31	773	631.3	18.33	02:00:00
st	70	26	1531	979.5	36.02	00:02:56	1067	866.5	18.79	02:00:00
st	70	35	1888	1202.6	36.31	00:04:20	1534	1082.5	29.43	02:00:00
kroA	100	20	42845	19839.5	53.69	00:06:58	49089	19602.5	60.07	02:00:00
kroA	100	35	61802	26609.3	56.94	00:12:54	86377	28331.3	67.2	02:00:00
kroA	100	50	68537	31378.5	54.22	00:18:27	155967	33839.0	78.3	02:00:00
kroA	150	30	56725	22773.6	59.85	00:23:24	49126	24173.6	50.79	02:00:00
kroA	150	50	71392	29684.8	58.42	00:40:14	154870	32461.8	79.04	02:00:00
kroA	150	75	88672	36283.2	59.08	01:02:03	553614	42973.0	92.24	02:00:00
kroA	200	40	64500	23745.5	63.19	00:54:46	353805	26323.2	92.56	02:00:00
kroA	200	75	91687	32872.0	64.15	01:50:27	–	–	–	–
kroA	200	100	102137	37932.2	62.86	02:00:00	–	–	–	–
lin	318	61	49769	17303.4	65.23	02:00:00	–	–	–	–
lin	318	111	86928	21634.2	75.11	02:00:00	–	–	–	–
lin	318	159	121811	24089.2	80.22	02:00:00	–	–	–	–

and SH separately with those obtained by PHA. The results are given by Tables 3.3 and 3.4 for rooted and arbitrary demands respectively.

We can see from Table 3.3 that the upper bounds produced by PHA for all the instances with rooted demands are better than those obtained by LRA, SH and GA, taken separately. Also, for the arbitrary demands (see Table 3.4), PHA outperforms algorithms LRA, SH and GA taken separately. This clearly shows that, with both rooted and arbitrary demands, the hybridization of the three components produce better results than each component taken separately. When comparing PHA and LRA in terms of lower bounds, we can see that for almost all the instances, with both rooted and arbitrary demands, the lower bound obtained by LRA is better than that obtained by PHA. Thus, clearly, the hybridization helps in producing better upper bounds for most of the instances, but does not help in improving the lower bounds.

Table 3.3: Results for PHA versus LRA, SH and GA for the kESNDP and rooted demands with $k = 3$.

Instances			PHA		LRA		SH	GA
name	$\|V\|$	$\|D\|$	UB	LB	UB	LB	UB	UB
berlin	30	10	7168	6072.3	10543	6298.8	9128	7459
berlin	30	15	10970	7856.1	16661	8163.8	12930	10970
berlin	30	20	12173	8392.0	23165	8356.8	14271	12311
berlin	30	25	12805	8832.3	22521	8905.6	15355	13271
berlin	52	10	6815	4740.6	13012	5272.4	8668	7034
berlin	52	15	9871	6821.5	22157	6894.8	12212	10373
berlin	52	26	13033	7639.5	30418	8000.7	15257	13559
st	70	15	761	523.6	2068	539.0	933	789
st	70	26	1077	707.2	3445	809.6	1271	1095
st	70	35	1265	805.4	4304	899.0	1442	1368
kroA	100	20	26677	11864.8	71777	12605.4	34141	28204
kroA	100	35	39378	16182.3	114336	16922.2	47090	39857
kroA	100	50	47454	18875.4	160719	20416.9	53756	47531
kroA	150	30	36340	11596.3	101835	13239.7	44497	37824
kroA	150	50	47764	15774.2	170169	19149.2	53690	48303
kroA	150	75	57909	17742.1	224847	23497.4	66579	60797
kroA	200	40	41724	12080.4	154211	14451.4	51335	43110
kroA	200	75	59482	16096.8	277544	19038.9	69033	60606
kroA	200	100	69462	18792.2	361814	23728.7	80717	69929
lin	318	61	26723	6068.6	139868	7713.9	31788	27143
lin	318	111	55000	8702.2	375430	13495.8	56199	55559
lin	318	159	71464	8719.7	559326	14637.2	77340	75389

3.3.2 SkESNDP

For the SkESNDP, we have used the same environment as used for kESNDP. The test problems have been obtained from the TSPLIB library [2]. Each test set consists in complete graphs whose edge weights are the rounded Euclidean distance between the edge vertices. In addition, for each graph, a set S of terminals is defined including the first $|S|$ nodes of G.

The results obtained for SkESNDP by all the algorithms are summarized in Table

Table 3.4: Results for PHA versus LRA, SH and GA for the kESNDP and arbitrary demands with $k = 3$.

Instances			PHA		LRA		SH	GA
name	$\|V\|$	$\|D\|$	UB	LB	UB	LB	UB	UB
berlin	30	10	10422	8216.0	11884	8384.9	12808	10597
berlin	30	15	12385	9716.2	18118	9683.7	15283	12584
berlin	52	10	10553	7772.5	14052	7691.9	12369	10725
berlin	52	15	13588	9141.0	19042	9210.8	15496	13771
berlin	52	20	15603	10347.5	22341	10364.2	17908	16152
st	70	15	1176	678.3	2236	692.1	1267	1208
st	70	26	1531	979.5	3277	1030.3	1743	1567
st	70	35	1888	1202.6	4673	1289.0	2137	1908
kroA	100	20	42845	19839.5	110812	20089.4	48435	44661
kroA	100	35	61802	26609.3	183685	27237.4	65024	61802
kroA	100	50	68537	31378.5	269704	31777.2	74250	69406
kroA	150	30	56725	22773.6	164720	23388.5	59947	56725
kroA	150	50	71392	29684.8	269616	30290.8	74614	71392
kroA	150	75	88672	36283.2	418821	38724.2	92894	89342
kroA	200	40	64500	23745.5	223326	24728.0	71033	66202
kroA	200	75	91687	32872.0	439205	35395.6	99364	92285
kroA	200	100	102137	37932.2	566823	42194.5	113464	102577
lin	318	61	49769	17303.4	74080	21788.7	53054	50168
lin	318	111	86928	21634.2	137343	31754.5	92973	89158
lin	318	159	121811	24089.2	206426	39037.3	129651	122911

3.5 whose entries are:

$\|V\|$	:	number of nodes of the graph,
$\|S\|$	:	number of terminals,
UB	:	best upper bound,
LB	:	best lower bound,
Gap	:	relative error between the best upper and lower bounds,
CPU	:	total CPU time in hours:min:sec.

The results of PHA algorithm are interesting (see Table 3.5).

First we can easily notice that the approach is able to improve, for all the instances, the upper bounds given by SH and RLA. However we notice that, the upper bound given by RLA is very weak as in 17 instances over 20.

Second, comparing PHA to CPLEX we can see that our algorithm produces better

Table 3.5: Numerical results for the algorithms RLA, SH, PHA and CPLEX for $k = 3$

Instances			RLA				SH		PHA				CPLEX			
name	$\|V\|$	$\|S\|$	*UB*	*LB*	*Gap*	*CPU*	*UB*	*CPU*	*UB*	*LB*	*Gap*	*CPU*	*UB*	*LB*	*Gap*	*CPU*
	30	3	2539	2409.13	5.12	00:00:03	2800	00:00:00	2503	2285.98	8.67	00:00:00	2489	2489	0	00:00:00
	30	5	5507	2993.18	45.65	00:00:06	4022	00:00:00	3638	2949.69	18.92	00:00:45	3191	3191	0	00:02:08
berlin	52	3	2536	1501.06	40.81	00:00:04	2871	00:00:00	1853	1653.38	10.77	00:00:03	1776	1776	0	00:00:02
	52	5	11793	2201.2	81.33	00:00:20	3408	00:00:00	2971	2135.77	28.11	00:01:33	2462	2298.33	6.65	02:00:00
	52	7	20400	3127.03	84.67	00:00:37	4761	00:00:00	4187	3006.06	28.2	00:03:19	3802	3389.61	10.85	02:00:00
	70	5	1040	152.446	85.34	00:00:29	360	00:00:00	236	164.30	30.38	00:00:07	223	179	19.73	02:00:00
st	70	7	1548	172.465	88.86	00:00:43	408	00:00:00	364	204.73	43.76	00:03:02	252	225	10.71	02:00:00
	70	9	2920	201.558	93.1	00:01:13	552	00:00:00	507	222.73	56.07	00:07:10	688	297.111	56.82	02:00:00
	100	5	62059	5317.46	91.43	00:01:16	16366	00:00:00	13151	5085.76	61.33	00:05:10	15292	5812.28	61.99	02:00:00
	100	7	106829	5418.4	94.93	00:02:32	19902	00:00:00	19469	5871.49	69.84	00:12:48	24360	6270.08	74.26	02:00:00
	100	9	151739	5358.9	96.47	00:04:12	21614	00:00:00	19846	5270.00	73.45	00:24:35	49051	7821.06	84.06	02:00:00
	150	7	106958	4248.35	96.03	00:05:38	19870	00:00:00	17882	4280.29	76.06	00:26:35	31512	5556.12	82.37	02:00:00
kroA	150	9	155197	5034.3	96.76	00:09:34	21345	00:00:00	20038	4826.98	75.91	00:52:44	51384	6582	87.19	02:00:00
	150	11	240264	7681.11	96.8	00:14:15	24937	00:00:00	24016	7517.20	68.7	01:27:04	50323	8060.58	83.98	02:00:00
	200	9	194591	6294.12	96.77	00:17:02	22947	00:00:00	20403	6088.26	70.16	01:33:39	117741	5458.25	95.36	02:00:00
	200	11	296236	7472.18	97.48	00:22:48	27178	00:00:01	25195	7173.50	71.53	02:00:00	222560	6361.25	97.14	02:00:00
	200	13	375741	9256.4	97.54	00:35:33	29989	00:00:01	26397	8525.46	67.7	02:00:00	294825	7890.25	97.32	02:00:00
	318	11	34628	2848.95	91.77	00:54:49	6252	00:00:04	5671	1978.31	65.12	02:00:00	251507	0	100	02:00:00
lin	318	13	56637	3553.51	93.73	01:00:24	8051	00:00:05	7660	2656.36	65.32	02:00:00	93116900	0	100	02:00:00
	318	15	75722	3369.12	95.55	01:37:53	9331	00:00:07	9017	2749.56	69.51	02:00:00	–	–	–	–

upper bounds in 13 cases over 20 while CPLEX is able to solve to optimality three instances. We can also see that for 14 instances, CPLEX produces a better lower bound than PHA. Also, the gap produced by PHA is better than that produced by CPLEX for 13 instances.

Now, we compare the algorithms SH, RLA and GA, taken separately, with algorithm PHA. We can see that for all the instances, algorithm PHA produces a better upper bound than SH and RLA. We can also observe that, for several instances, the lower bound achieved by RLA taken separately is better than that obtained by PHA. These two observations show that the combination of the three algorithms does not always help in improving the lower bound, but clearly helps in improving the upper bound.

Finally, we can see that the CPU time of PHA is relatively small (less than 1h) for 13 instances over 20, while CPLEX reaches the maximum CPU time for almost all the instances (18 instances over 20). Moreover, PHA has been able to produce an upper bound for instance lin318-15 while CPLEX was not able to produce even a feasible solution due to lack of memory.

3.4 Conclusion

In this Chapter we considered the k-Edge-Connected Network Design Problem (kESNDP) and its variant the Steiner k-Edge-Connected Network Design Problem (kESNDP). We propose a parallel hybrid algorithm which aims to produce good solutions for large scale instances of the two problems. Our approach is based on a Lagrangian relaxation of a flow-based integer programming formulation of the problem, a greedy and a genetic algorithms.

Chapter 4

The k-Edge-Connected Hop-Constrained Network Design Problem

Contents

In this chapter, we address the k-edge-connected hop-constrained network design problem (kHNDP) which is known to be NP-hard. We apply and adapt the approach

presented in the previous chapter for solving the kESNDP to the kHNDP. We test the algorithms within an extensive computational study and compare their efficiency for solving the kHNDP against CPLEX.

Let $G = (V, E)$ be an undirected graph, a set of demands $D \subseteq V \times V$, a cost function $c : E \to \mathbb{R}$ which associates the cost $c(e)$ with each edge $e \in E$. We recall that the *k-Edge-Connected Hop-Constrained Network Design Problem* (kHNDP for short) consists in finding a minimum cost sub-graph of G such that there exist k edge-disjoint L-st-paths (st paths whose lengths are $\leq L$) between the terminals of each demand $\{s, t\}$ of D.

In this chapter, we introduce a study on the kHNDP when $k \geq 2$, $|D| \geq 1$ and $L \geq 2$. In fact, while in the literature the focus was on the polyhedral point of view of the kHNDP trying to solve to optimality the problem, we noticed that most algorithms proposed could not solve the instances when $|V| \geq 50$ and $|D| \geq 20$. Thereby, we introduce a new parallel hybrid approach based on a Lagrangian relaxation method, a fast greedy algorithm and a genetic algorithm to solve in an approximate way the kHNDP.

The chapter is organized as follows. In Section 4.1, we present the graph transformations, and the flow based formulations for the problem in both cases when $L = 2, 3$ and $L \geq 4$. In Section 4.2, we introduce the application of PHA on the different problem formulations. Then, in the last section, we present some computational results and proceed to a comparison between our results and those of CPLEX.

4.1 Integer programming formulations

4.1.1 When $L = 2, 3$

In this section, we present the integer programming formulation of the kHNDP, proposed by Diarrassouba et al. [96], on which we based our study. This formulation use a directed layered graph to model each hop-constrained subproblem.

4.1.1.1 Graph transformation

Let $\{s, t\} \in D$ and $\widetilde{G}_{st} = (\widetilde{V}_{st}, \widetilde{A}_{st})$ be the directed graph obtained from G using the following procedure.

Let $N_{st} = V \setminus \{s,t\}$, N'_{st} be a copy of N_{st} and $\widetilde{V}_{st} = N_{st} \cup N'_{st} \cup \{s,t\}$. The copy in N'_{st} of a node $u \in N_{st}$ will be denoted by u'. To each edge $e = st \in E$, we associate an arc (s,t) in $\widetilde{G}_{st}$ with capacity 1. With each edge $su \in E$ (resp. $vt \in E$), we associate in $\widetilde{G}_{st}$ the arc (s,u), $u \in N_{st}$ (resp. (v',t), $v' \in N'_{st}$) with capacity 1. With each node $u \in V \setminus \{s,t\}$, we associate in $\widetilde{G}_{st}$ arc (u,u') with an infinite capacity. Finally, if $L = 3$ we associate with each edge $uv \in E \setminus \{s,t\}$, two arcs (u,v') and (v,u'), with $u,v \in N_{st}$ and $u',v' \in N'_{st}$ with capacity 1 (see Figure 4.1 for an illustration with $D = \{s_1,t_1\},\{s_1,t_2\},\{s_3,t_3\}$ and $L = 3$).

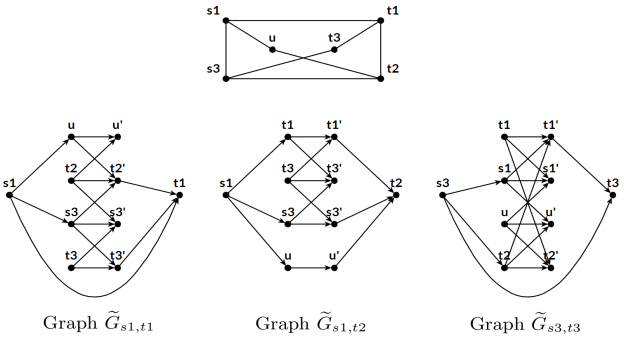

Figure 4.1: Construction of $\widetilde{G}_{st}$ graphs

This transformation was first introduced by Gouveia and Requejo in [139].

Given a demand $\{s,t\}$, the associated graph $\widetilde{G}_{st} = (\widetilde{V}_{st}, \widetilde{A}_{st})$, and for an edge $e \in E$, we denote by $\widetilde{A}_{st}(e)$ the set of arcs of $\widetilde{G}_{st}$ corresponding to the edge e. It is not hard to see that $\widetilde{G}_{st}$ does not contain any circuit. Also, observe that any st-dipath in $\widetilde{G}_{st}$ is of length no more than 3. Moreover each L-st-path in G corresponds to an st-dipath in $\widetilde{G}_{st}$ and conversely. In fact, if $L \in \{2,3\}$, every 3-st-path (s,u,v,t), with $u \neq v$,$u,v \in V \setminus \{s,t\}$, corresponds to an st-dipath in $\widetilde{G}_{st}$ of the form (s,u,v',t) with $u \in N_{st}$ and $v' \in N'_{st}$. Every 2-st-path (s,u,t), $u \in V \setminus \{s,t\}$, corresponds to an st-dipath in $\widetilde{G}_{st}$ of the form (s,u,u',t).
We have the following lemma (see [96] for the proof).

Lemma 1 *Let $L \in \{2,3\}$ and $\{s,t\} \in D$.*

i) If two L-st-paths of G are edge-disjoint, then the corresponding st-dipaths in $\widetilde{G}_{st}$ are arc-disjoint.

ii) If two st-dipaths of $\widetilde{G}_{st}$ are arc-disjoint, then the corresponding st-paths in G contain two edge-disjoint L-st-paths.

As a consequence of Lemma 1, for $L \in \{2,3\}$ and every demand $\{s,t\} \in D$, a set of k edge-disjoint L-st-paths of G corresponds to a set of k arc-disjoint st-dipaths of $\widetilde{G}_{st}$, and k arc-disjoint st-dipaths of $\widetilde{G}_{st}$ correspond to k edge-disjoint L-st-paths of G. Using these results, Diarrassouba et al. [96] introduced a flow-based formulation for the kHNDP which rely on the transformed graphs $\widetilde{G}_{st}$. This formulation, called separated flow formulation, is described in the next section.

Therefore we have the following corollary:

Corollary 1 *Let H be a subgraph of G and $\widetilde{H}_{st}$, $\{s,t\} \in D$, the subgraph of $\widetilde{G}_{st}$ obtained by considering all the arcs of $\widetilde{G}_{st}$ corresponding to an edge of H, plus the arcs of the form (u,u'), $u \in V \setminus \{s,t\}$. Then H induces a solution of the kHNDP if $\widetilde{H}_{st}$ contains k arc-disjoint st-dipaths, for every $\{s,t\} \in D$. Conversely, given a set of subgraphs $\widetilde{H}_{st}$ of $\widetilde{G}_{st}$, $\{s,t\} \in D$, if H is the subgraph of G obtained by considering all the edges of G associated with at least one arc in a subgraph $\widetilde{H}_{st}$, then H induces a solution of the kHNDP only if $\widetilde{H}_{st}$ contains k arc-disjoint st-dipaths, for every $\{s,t\} \in D$.*

4.1.1.2 The separated flow formulation

The Corollary 1 suggests at once the following formulation for the kHNDP when $L = 2,3$.

Let $F \subseteq E$ be a subgraph of G. Given a demand $\{s,t\}$, we let $f^{st} \in \mathbb{R}^{\widetilde{A}_{st}}$ be a flow vector on $\widetilde{G}_{st}$ of value k between s and t. Then f^{st} satisfies the *flow conservation constraints*.

$$\min \sum_{e \in E} c(e)x(e)$$

$$\sum_{a \in \delta^+(u)} f_a^{st} - \sum_{a \in \delta^-(u)} f_a^{st} = \left\{ \begin{array}{lll} k & if & u = s \\ 0 & if & u \in \widetilde{V}_{st} \setminus \{s,t\} \\ -k & if & u = t \end{array} \right\} \tag{4.1}$$

$$f_a^{st} \leq x(e) \qquad x \in \mathbb{Z}_+^E; f^{st} \in \mathbb{Z}_+^{\widetilde{A}_{st}} \tag{4.2}$$

$$f_a^{st} \leq 1 \qquad f^{st} \in \mathbb{Z}_+^{\widetilde{A}_{st}} \tag{4.3}$$

$$f_a^{st} \geq 0 \qquad f^{st} \in \mathbb{Z}_+^{\widetilde{A}_{st}} \tag{4.4}$$

$$x(e) \leq 1 \qquad x \in \mathbb{Z}_+^E \tag{4.5}$$

(4.1) for all $u \in \widetilde{V}_{st}$
(4.2) for all $a \in \widetilde{A}_{st}(e)$, $\{s,t\} \in D$, $e \in E$
(4.3) for all $a = (u,u')$, $u \in V \setminus \{s,t\}$, $\{s,t\} \in D$
(4.4) for all $a \in \widetilde{A}_{st}(e)$, $\{s,t\} \in D$
(4.5) for all $e \in E$

The *separated flow formulation* has been introduced firstly by Diarrassouba et al. [96] and uses a polynomial number of variables and constraints. In fact, the formulation uses flows in $|D|$ graphs layered graphs obtained by applying a transformation on the original graph G for each demand $\{s,t\}$ (Gouveia and Requejo [139]).

4.1.2 When $L \geq 4$

4.1.2.1 Graph transformation

For our purpose, we use the graph transformation proposed by Diarrassouba et al. [101], which transforms the original graph G into a set of directed layered graphs. This transformation is presented below. For each demand $\{s,t\} \in D$, first let $V_{st}^l = \{(u,l), \text{ for all } u \in V \setminus \{s,t\}\}$, $l = 1, ..., L-1$. Then, let $\widetilde{G}_{st} = (\widetilde{V}_{st}, \widetilde{A}_{st})$ be the directed graph where $\widetilde{V}_{st} = \bigcup_{l=1}^{L-1} V_{st}^l \cup \{s,t\}$. The arc set $\widetilde{A}_{st}$ is obtained by adding in $\widetilde{G}_{st}$

- two arcs of the form $((u,l),(v,l+1))$ and $((v,l),(u,l+1))$, for all $l \in \{1, ..., L-2\}$ and every edge $uv \in E$ such that $u, v \in V \setminus \{s,t\}$,
- an arc of the form $(s,(u,1))$, for every edge $su \in E$ with $u \in V \setminus \{s,t\}$,
- an arc of the form $((u,L-1),t)$, for every edge $ut \in E$ with $u \in V \setminus \{s,t\}$,
- an arc of the form (s,t), for every edge $st \in E$,
- an arc of the form $((u,l),(u,l+1))$, for every $u \in V \setminus \{s,t\}$ and $l \in \{1, ..., L-2\}$.

Figure 4.2 gives an illustration with $D = \{\{s_1,t_1\},\{s_2,t_2\}\}$ and $L = 4$.

Diarrassouba et al. [101] showed that every L-st-path of G corresponds to an st-path in $\widetilde{G}_{st}$ and vice-versa. Moreover, They showed that there exists k edge-disjoint L-st-paths in G if and only if there exists k arc-disjoint st-paths in $\widetilde{G}_{st}$ such that any pair of arcs used in these paths corresponds to the same edge of G.

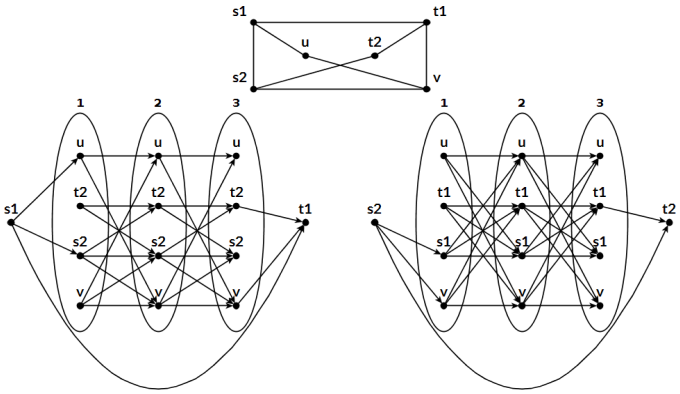

Figure 4.2: Graph transformation for $L = 4$

4.1.2.2 The separated flow formulation

The transformation above yields to the following integer programming formulation [101], called *flow formulation*, for the kHNDP when $L \geq 4$. Let x_e, for each edge $e \in E$, be the $0-1$ variable which takes value 1 if the edge e is in the solution and 0 otherwise. Let f_a^{st} be a flow variable associated with arc a of $\widetilde{G}_{st}$. The kHNDP is equivalent to the following integer program

$$\min \quad \sum_{uv \in E} \omega_{uv} x_{uv}$$

s.t.

$$\sum_{a \in \delta^+(u)} f_a^d - \sum_{a \in \delta^-(u)} f_a^d = \left\{ \begin{array}{rcl} k & if & u = s \\ -k & if & u = t \\ 0 & if & u \in \widetilde{V}_{st} \setminus \{s,t\}, \end{array} \right\},$$

$$\text{for all } u \in \widetilde{V}_{st} \text{ and } \{s,t\} \in D, \tag{4.6}$$

$$\sum_{a \in A_{st}(e)} f_a^{st} \leq x_e, \text{for all } e \in E \text{ and } \{s,t\} \in D, \tag{4.7}$$

$$f_a^{st} \geq 0, \text{ for all } a \in \widetilde{A}_{st} \text{ and } \{s,t\} \in D, \tag{4.8}$$

$$x_e \leq 1, \text{ for all } e \in E, \tag{4.9}$$

$$x_e \in \{0,1\}, \text{ for all } e \in E, \tag{4.10}$$

$$f_a^{st} \in \{0,1\}, \text{ for all } a \in \widetilde{A}_{st} \text{ and } \{s,t\} \in D. \tag{4.11}$$

4.2 The Parallel Hybrid Algorithm

In this section we introduce the application of the parallel, hybrid algorithm (PHA) for solving the kHNDP. According to Talbi's taxonomy, the approach consists in a parallel, HTH (High-level, Teamwork) hybrid, approximation algorithm based on three components; a lagrangian relaxation algorithm (RLA), a greedy successive heuristic (SH) and a genetic algorithm (GA).

4.2.1 Greedy Successive Heuristic

Our greedy heursitic for the kHNDP works similarly to that we have devised for the kESNDP. We start by randomly generating a set of $|D|$ ordering of the demands. For a given ordering, we let $D = \{\{s_1, t_1\}, \{s_2, t_2\}, ...\{s_d, t_d\}\}$ be the demands w.r.t that ordering. For $i \in \{1, ..., |D|\}$, we denote by E_i the set of edges of G having a flow value of 1 in a minimum cost s_it_i-flow of value k in $\widetilde{G}_{s_it_i}$. Moreover, in this flow at most one arc corresponding to the same edge have a flow value of 1. The edge set E_i is computed as follows. First, compute a minimum cost s_it_i-flow of value k in $\widetilde{G}_{s_it_i}$, where all the arcs have capacity 1, all the arcs corresponding to an edge of $\bigcup_{j=1}^{i-1} E_j$ have a cost of 0, and all the arcs corresponding to the edges $e \in E \setminus \bigcup_{j=1}^{i-1} E_j$ have a cost ω_e. Let $\overline{f}^{s_it_i}$ be the obtained flow vector and let $F_{s_it_i}$ be the set of edges $e \in E$ such that $\sum_{a \in \widetilde{A}_{s_it_i}(e)} \overline{f}_a^{s_it_i} > 1$. If $F_{s_it_i} = \emptyset$, then we let E_i be the set of edges $e \in E$ such that $\sum_{a \in \widetilde{A}_{s_it_i}(e)} \overline{f}_a^{s_it_i} = 1$. If $F_{s_it_i} \neq \emptyset$, then, for each edge $e \in F_{s_it_i}$, we arbitrarily choose an arc $a_0 \in \widetilde{A}_{s_it_i}(e)$, give him a capacity 1, give a capacity 0 to all the arcs of $\widetilde{A}_{s_it_i}(e) \setminus \{a_0\}$, and compute minimum cost flow of value k in $\widetilde{G}_{s_it_i}$ with the same arc costs as before. Finally, if $\overline{f}'^{st}$ is the new flow vector, then we let E_i be the set of edges $e \in E$ such that $\sum_{a \in \widetilde{A}_{s_it_i}(e)} \overline{f}_a'^{st} = 1$.

Finally, it is not hard to see that the edge set $\bigcup_{j=1}^{|D|} E_j$ is a feasible solution for the kHNDP. This procedure is repeated for all the ordering of the demands, yielding $|D|$ feasible solutions. The algorithm ends by finding the best one among these solutions.

4.2.2 Lagrangian Relaxation

4.2.2.1 When $L = 2, 3$

Analysing the variable dependencies's of the *Separated Flow Formulation* presented in the previous Section 4.1.1, we notice that if we apply the lagrangian relaxation on the linking constraints set, solving the integer program will consist in solving $|D|$ separated flow problem.

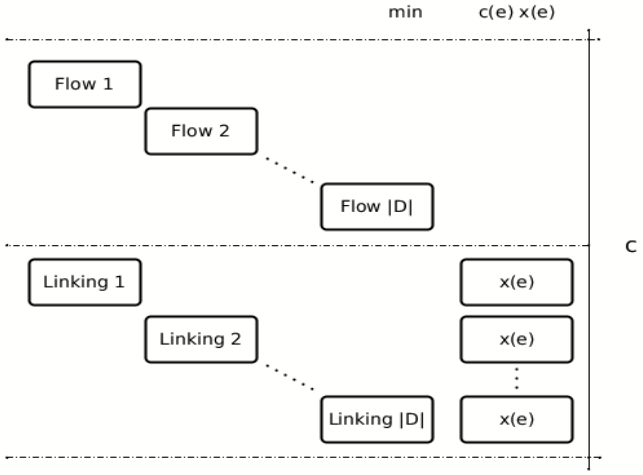

Figure 4.3: Formulation variable dependencies

As we can see, in the first part of the LP we have $|D|$ flow problems completely independent. In the second part, we find the $\sum_{\{s,t\}\in D} |\widetilde{A}_{st}(e)|$ linking equations that harden the problem solving. In other words, if we write the kHNDP in matrix notation:

$$\begin{array}{rcll} & min & cx & \\ Af & \geq & b & f \in \mathbb{Z} \\ Df - Ex & \leq & 0 & f, x \in \mathbb{Z} \\ 0 \leq & f & \leq 1 & f \in \mathbb{Z} \\ x & \leq & 1 & x \in \mathbb{Z} \end{array}$$

We define the lagrangian relaxation of our problem with respect to the constraint set $Df - Ex \leq 0$ by introducing a lagrange multiplier vector $\lambda \geq 0$ which is attached to this constraint set and brought into objective function to get:

$$\begin{array}{rrl} min & (c-\lambda E)x+\lambda Df & \\ & Af & \geq b \\ 0 \leq & f & \leq 1 \\ & x & \leq 1 \\ & f,x & \in \mathbb{Z} \end{array}$$

Hence, switching back from matrix notation to summation notation, we obtain the following $P(\lambda)$:

$$min \quad \sum_{e\in E}\left[c(e)-\left(\sum_{\{s,t\}\in D}\sum_{a\in\widetilde{A}_{st}(e)}\lambda_a^{st}\right)\right]x(e)+$$

$$\sum_{\{s,t\}\in D}\sum_{a\in\widetilde{A}_{st}(e)}\lambda_a^{st}f_a^{st}$$

$$\sum_{a\in\delta^+(u)}f_a^{st}-\sum_{a\in\delta^-(u)}f_a^{st}=\left\{\begin{array}{lll} k & if & u=s \\ 0 & if & u\in\widetilde{V}_{st}\backslash\{s,t\} \\ -k & if & u=t \end{array}\right\}$$

for all $u \in \widetilde{V}_{st}$

$$\begin{array}{rcll} f_a^{st} & \leq & 1, & \text{for all } a=(u,u'),\ u\in V\backslash\{s,t\},\ \{s,t\}\in D \\ f_a^{st} & \geq & 0, & \text{for all } a\in\widetilde{A}_{st}(e),\ \{s,t\}\in D \\ x(e) & \leq & 1, & \text{for all } e\in E \\ x & \in & \mathbb{Z}_+^E & \\ f^{st} & \in & \mathbb{Z}_+^{\widetilde{A}_{st}} & \end{array}$$

Based on the program obtained, two algorithms (RLA and $RLASH$) have been designed for the $kHNDP$. RLA and RLASH start by solving the $|D|$ flow problems in a parallel multithreaded fashion using the $Network\ Simplex$ algorithm [11, 77].

To fix the lagrange multipliers, the subgradient algorithm is used. In the subgradient phase of both algorithms, we use a heuristic in case of having an X_j non feasable for the initial kHNDP formulation to transform it to a feasible $\overline{X_j}$. In RLA, we use a fast primal heuristic to round to 1 each $x_j(e)$ that violates a $f_a^{st} \leq x(e)$ constraint. In RLASH, instead of a primal heuristic, we use the heuristic SH that we designed. For

each graph $\widetilde{G}_{st}$, the algorithm starts by setting up the cost of each arc related to an edge already selected in the solution X_j ($x_j(e) = 1$) to 0 and then solve the current flow problem.

4.2.2.2 When $L \geq 4$ [98]

As for the case when $L = 2, 3$, our Lagrangian algorithm is based on the relaxation of the linking constraints (4.7). This yields the following linear program, called problem (LR), where λ_{uv}^{st}, for all $uv \in E$, are the Lagrange multipliers associated with the linking constraints,

$$\min \quad \sum_{uv \in E} \left(\omega_{uv} - \sum_{\{s,t\} \in D} \lambda_{uv}^{st} \right) x_{uv} + \sum_{\{s,t\} \in D} \sum_{uv \in E} \lambda_{uv}^{st} \sum_{a \in \widetilde{A}_{st}(uv)} f_a^{st}$$

s.t.

$$\sum_{a \in \delta^+(u)} f_a^d - \sum_{a \in \delta^-(u)} f_a^d = \left\{ \begin{array}{rcl} k & if & u = s \\ -k & if & u = t \\ 0 & if & u \in \widetilde{V}_{st} \setminus \{s,t\}, \end{array} \right\},$$

$$\text{for all } u \in \widetilde{V}_{st} \text{ and } \{s,t\} \in D,$$

$$0 \leq f_a^{st} \leq 1, \text{ for all } a \in \widetilde{A}_{st} \text{ and } \{s,t\} \in D,$$

$$0 \leq x_e \leq 1, \text{ for all } e \in E,$$

$$x_e \in \{0,1\}, \text{ for all } e \in E,$$

$$f_a^{st} \in \{0,1\}, \text{ for all } a \in \widetilde{A}_{st} \text{ and } \{s,t\} \in D.$$

As before, one can easily see that solving problem (LR) reduces to solving $|D|$ minimum cost st-flow of value k in graphs $\widetilde{G}_{st}$, which can be done in polynomial time. Also, as for the kESNDP, the Lagrange multipliers are updated using the sub-gradient method.

It should be noticed that, contrarily to the kESNDP, the solution $\overline{y} \in \mathbb{R}^E$ such that

$$\overline{y}_e = \max\{\overline{f}_a^{st}, \text{ for all } a \in \widetilde{A}_{st}(e) \text{ and } \{s,t\} \in D\}, \text{for all } e \in E,$$

may not be feasible for the kHNDP. Thus, at each iteration of the algorithm, we check if $\overline{y}$ is feasible for the kHNDP, and if not, we transform $\overline{y}$ into a feasible solution. To

see if $\overline{y}$ is feasible or not, it suffices to check if each flow vector $\overline{f}^{st}$, for every $\{s,t\} \in D$, is such that

$$\sum_{a \in \widetilde{A}_{st}(e)} \overline{f}_a^{st} \leq \overline{y}_e. \tag{4.12}$$

If $\overline{y}$ is not feasible for the kHNDP, then for each $\{s,t\} \in D$, we let F_{st} be the set of edges of G for which an inequality (4.12) is violated. We build a new solution $\overline{y}' \in \mathbb{R}^E$ as follows. If for a demand $\{s,t\} \in D$, $F_{st} = \emptyset$, then we let $\overline{g}^{st}$ be a flow vector such that $\overline{g}^{st} = \overline{f}^{st}$. If $F_{st} \neq \emptyset$, then for all $e \in F_{st}$, we arbitrarily choose an arc $a_0 \in \widetilde{A}_{st}(e)$, give him a capacity 1, give a capacity 0 to all the arcs of $\widetilde{A}_{st}(e) \setminus a_0$, and compute a minimum cost st-flow of value k, each arc $a \in \widetilde{A}_{st}(uv)$ having a cost ω_{uv}, for all $uv \in E$. We let $\overline{g}^{st}$ be the value of that minimum cost flow. Finally, the solution $\overline{y}'$ is such that

$$\overline{y}'_e = \max\{\overline{g}_a^{st}, \text{ for all } a \in \widetilde{A}_{st}(e) \text{ and } \{s,t\} \in D\}, \text{all } e \in E,$$

and is clearly feasible for the kHNDP when $L \geq 4$.

4.2.3 Genetic Algorithm

Our genetic algorithm for the kHNDP follows the same lines that we have devised for the kESNDP. The reader can refer to Section 3.2.3 for the details.

4.2.4 Hybridization

The hybridization and parallelization scheme for the parallel hybrid algorithm for the kHNDP when are the same as that devised for the SkESNDP. We refer the reader to Section 3.2.4 for the details.

4.3 Experimental Study

In this section we present a computational study of the different algorithms introduced in this chapter. We aim to check their efficiency for solving the problem for large scale

instances, and compare them to each other from a computational time, and solution quality point of view.

The algorithms described in the previous section have been implemented in C++, using LEMON graph structures [8]. They were tested on an Intel i7 2.5 Ghz with 8 Gb of RAM, running under Linux. We fixed the maximum CPU time to 2 hours. The test problems were obtained by taking TSP test problems from the TSPLIB library [2]. The test set consists in complete graphs whose edge weights are the rounded Euclidean distance between the edge's vertices. In addition, for each instance, a set of $|D|$ demand in the form $\{i, i+1\}$ is defined including the first $2|D|$ nodes.

4.3.1 When $L = 2, 3$

Table 4.1 exhibits computational results obtained for a set of 28 TSP test instances in form of a comparison between our algorithms and CPLEX 12.6 [1] executed in parallel. The results given are obtained for $k = 3$ and $L = 3$. Each instance is given by the graph's number of nodes and preceded by its name. Since one of the components of parallel hybrid algorithm have stochastic components (e.g. the GA), each PHA entry represents the mean of 5 runs on the same instance. Finally, note that empty entries does not mean that the time needed to compute a feasible solution exceeded 2 hours, but that the system was not able to charge the problem itself to memory.

$\|V\|$	:	number of nodes of the graph;
$\|D\|$	:	number of demands
UB	:	objective function value (best upper bound)
LB	:	best lower bound
Gap	:	relative error between the best upper and lower bounds
CPU	:	total CPU time in hours:min:sec

First, by observing Table 4.1, we notice that for $L = 3$, the problem is solved to optimality by parallel CPLEX in 4 instances (having $|D| \leq 10$) over 28 (14.2%) and just once by our approximating algorithms (RLASH algorithm).

We remark that our lagrangian relaxation algorithms outperforms parallel CPLEX relatively to the solution cost (UB) in 13 instances over 28 (46.4%). The SH algorithm outperforms CPLEX as well but in 19 cases out of 28 (67.8%). (See Figure 4.4 for the solutions cost distribution).

For PHA algorithm, the results have been encouraging (see Table 4.1), where the approach has been able to improve the best upper bound obtained by the SH and RLA algorithms in 16 instances over 28 (57.1%). PHA could also benefit from the efficiency of the SH algorithm in cases where the graph is very large, and at least returns an upper bound for our problem. However we notice that, due to the weak lower bound obtained by the RLA algorithm, the gap is high relatively to the quality of the solution as in 5 instances, PHA obtains better upper bounds than CPLEX but giving, on the other hand, a weaker gap.

Figure 4.4: Solutions cost variation

Now, we turn our attention to the comparison of the algorithms in terms of CPU time, we notice that for all the instances, the greedy successive heuristic (SH) could give an upper bound to the kHNDP. In fact, it takes a bit more than 4 minutes to solve a 575 nodes graph with 287 demands while CPLEX cannot charge the mathematical programming model, for a graph of 318 nodes and 111 demands, into memory.

In Figure 4.5, we can see that for both RLA and RLASH, a range of 15 graphs (number of nodes from 52 to 318, and number of demands from 15 to 61) exists, in which they outperform CPLEX in computational time.

In the same way, we can see that PHA could in 11 instances over 28 (39.2%) produce solutions faster than CPLEX, reach the limit of 2 hours as same as CPLEX in 5 instances (17.8%), and could solve 4 large instances that CPLEX could not charge in memory.

Instances			RLA				SH		RLASH				PHA				CPLEX			
name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	CPU	UB	LB	Gap	CPU	UB	LB	Gap	CPU	UB	LB	Gap	CPU
	10	7	6100	4691.4	23.09	0:00:01	5315	0:00:00	4698	4690	0.17	0:00:01	5315	4691.6	11.73	0:00:00	4698	4698	0	0:00:00
	10	9	6966	5437.1	21.95	0:00:01	6452	0:00:00	6262	5438	13.16	0:00:01	6205	5440.1	12.33	0:00:00	5915	5915	0	0:00:00
	30	10	11914	9566.2	19.71	0:00:06	11756	0:00:00	11513	9553.4	17.02	0:00:14	10695	9467.6	11.48	0:00:37	10254	10170.49	0.81	0:00:29
	30	15	17198	12091.8	29.69	0:00:10	14854	0:00:00	16737	12079.3	27.83	0:00:28	13743	11847.4	13.79	0:01:12	13429	12829.37	4.47	2:00:00
berlin	52	10	13715	9361.3	31.74	0:00:21	11359	0:00:00	10625	9323.7	12.25	0:00:48	10298	9136	11.28	0:01:46	9919	9851.46	0.68	0:00:33
	52	15	19042	11828.7	37.88	0:00:32	14508	0:00:00	17458	11813.4	32.33	0:01:29	13416	11596.6	13.56	0:02:54	13278	12575.49	5.29	2:00:00
	52	20	22341	13715.4	38.61	0:00:47	17633	0:00:00	20659	13878.8	32.82	0:02:06	16534	13448.6	18.66	0:04:06	15891	14612.79	8.04	2:00:00
	70	15	2213	1038.9	53.05	0:00:45	1536	0:00:00	1884	1054.1	44.05	0:01:47	1389	971.2	30.07	0:03:21	1336	1119.59	16.20	2:00:00
st	70	26	3275	1421.1	56.61	0:01:30	2269	0:00:00	3103	1385.2	55.36	0:02:52	2107	1312.4	37.72	0:07:09	2176	1610.73	25.98	2:00:00
	70	35	4665	1822.7	60.93	0:01:55	2975	0:00:00	4494	1810.3	59.72	0:03:52	2834	1714.4	39.50	0:11:37	5299	2142.75	59.56	2:00:00
	100	20	110812	37839.6	65.85	0:02:53	66190	0:00:00	83141	39564.9	52.41	0:05:49	62128	37704.6	39.31	0:12:47	66738	44958.93	32.63	2:00:00
	100	35	183685	56581.2	69.20	0:05:00	111403	0:00:00	150595	55463.8	63.17	0:10:08	103164	55049.4	46.64	0:26:12	348018	68944.14	80.19	2:00:00
	100	50	269705	74624.1	72.33	0:06:57	151175	0:00:00	224709	72617.5	67.68	0:14:15	144842	72889.6	49.68	0:40:42	558391	92782.7	83.38	2:00:00
	150	30	165239	46682.3	71.75	0:10:03	101709	0:00:01	139662	47695.6	65.85	0:19:55	93542	45123	51.76	0:48:19	204665	61908.4	69.75	2:00:00
kroA	150	50	270609	68616.1	74.64	0:15:47	149949	0:00:01	234041	66428.5	71.62	0:32:39	144861	60073.2	58.53	1:39:41	441351	0	100.00	2:00:00
	150	75	420836	75891.5	81.97	0:23:28	214646	0:00:02	349514	74319.9	78.74	0:48:24	209575	75856.6	63.80	2:00:00	657939	0	100.00	2:00:00
	200	40	223098	48169.1	78.41	0:23:50	117931	0:00:02	186232	51255.1	72.48	0:46:42	114116	46091.6	59.61	2:00:00	442010	75495.36	82.92	2:00:00
	200	75	437497	76827.5	82.44	0:44:01	218219	0:00:04	352579	80010	77.31	1:26:56	212504	66209.2	68.84	2:00:00	848688	0	100.00	2:00:00
	200	100	565116	59827.8	89.41	0:55:50	278363	0:00:06	499671	58897.1	88.21	1:54:08	274917	67317	75.51	2:00:00	1125839	0	100.00	2:00:00
	318	61	74755	21638.4	71.05	1:27:34	56285	0:00:10	74755	26397.7	64.69	2:00:00	54721	20638.8	62.28	2:00:00	702945	0	100.00	2:00:00
lin	318	111	137906	26664.8	80.66	2:00:00	103297	0:00:17	137982	34667.4	74.88	2:00:00	100666	26664.8	73.51	2:00:00	–	–	–	–
	318	159	206681	30224.6	85.38	2:00:00	152685	0:00:25	206681	54128.7	73.81	2:00:00	149070	32224.6	78.38	2:00:00	–	–	–	–
	439	99	172774	41792.1	75.81	2:00:00	137614	0:00:40	172774	57137.6	66.93	2:00:00	135072	41059	69.60	2:00:00	–	–	–	–
pr	439	151	243527	47816.3	80.37	2:00:00	198826	0:01:01	243216	49612.3	79.60	2:00:00	196537	44071	77.58	2:00:00	–	–	–	–
	439	219	510121	88791.4	82.59	2:00:00	361150	0:01:30	509513	90316.7	82.27	2:00:00	355518	8751.5	97.54	2:00:00	–	–	–	–
	575	121	10974	862.409	92.14	1:21:49	8833	0:01:32	10319	851.7	91.75	2:00:00	8671	713.5	91.77	2:00:00	–	–	–	–
rat	575	201	18503	1072.2	94.21	2:00:00	14516	0:02:36	17428	1144.8	93.43	2:00:00	14363	1012.2	92.95	2:00:00	–	–	–	–
	575	287	25767	1487.69	94.23	2:00:00	20351	0:04:19	23591	1472.6	93.76	2:00:00	20068	1275.72	93.64	2:00:00	–	–	–	–

Table 4.1: Numerical results for the algorithms RLA, SH, RLASH, PHA and CPLEX

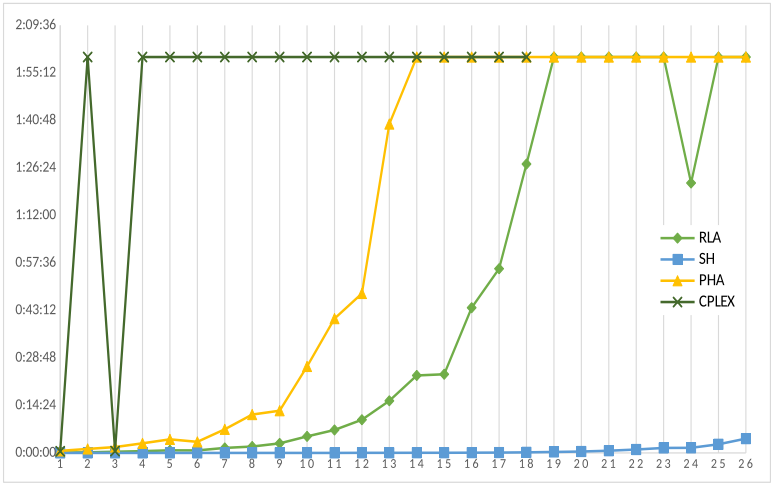

Figure 4.5: Computation time variation

4.3.2 When $L \geq 4$

Tables 4.2, 4.3 and 4.4 give the results obtained by PHA and CPLEX for the kHNDP with $k = 3$ and with $L = 3$, $L = 4$ and $L = 5$, respectively. The results presented here are those obtained for the instance with arbitrary demands. The entries of the table are

$\|V\|$:	number of nodes of the graph,
$\|D\|$:	number of demands,
UB:	best upper bound achieved by PHA (resp. CPLEX),
LB:	best lower bound achieved by PHA (resp. CPLEX),
Gap:	relative error between the best upper and lower bounds achieved by PHA (resp. CPLEX),
CPU:	total CPU time in hours:min:sec achieved by PHA (resp. CPLEX).

As before, when CPLEX does not solve the linear relaxation of the problem after the maximum CPU time (2 hours), the results are indicated with "–".

From Table 4.2, we can see that when $k = 3$ and $L = 3$, CPLEX outperforms PHA in producing upper bounds for the first 6 instances. For the others (14 over 20), PHA produces better upper bounds than CPLEX. Also, for almost all of the instances (18 instances over 20), CPLEX reaches the maximum CPU time while PHA requires less than 12 minutes for 9 instances.

Table 4.2: Results for PHA and CPLEX for the kHNDP and arbitrary demands with $k = 3$ and $L = 3$.

Instances			PHA				CPLEX			
name	$\|V\|$	$\|D\|$	*UB*	*LB*	*Gap*	*CPU*	*UB*	*LB*	*Gap*	*CPU*
berlin	30	10	10695	9467,6	11.48	00:00:37	10254	10170.49	0.81	00:00:29
berlin	30	15	13743	11847.4	13.79	00:01:12	13429	12829.37	4.47	02:00:00
berlin	52	10	10298	9136	11.28	00:01:46	9919	9851.46	0.68	00:00:33
berlin	52	15	13416	11596.6	13.56	00:02:54	13278	12575.49	5.29	02:00:00
berlin	52	20	16534	13448.6	18.66	00:04:06	15891	14612.79	8.04	02:00:00
st	70	15	1389	971.2	30.07	00:03:21	1336	1119.59	16.20	02:00:00
st	70	26	2107	1312.4	37.72	00:07:09	2176	1610.73	25.98	02:00:00
st	70	35	2834	1714.4	39.50	00:11:37	5299	2142.75	59.56	02:00:00
kroA	100	20	62128	37704.6	39.31	00:12:47	66738	44958.93	32.63	02:00:00
kroA	100	35	103164	55049.4	46.64	00:26:12	348018	68944.14	80.19	02:00:00
kroA	100	50	144842	72889.6	49.68	00:40:42	558391	92782.7	83.38	02:00:00
kroA	150	30	93542	45123	51.76	00:48:19	204665	61908.4	69.75	02:00:00
kroA	150	50	144861	60073.2	58.53	01:39:41	–	–	–	02:00:00
kroA	150	75	209575	75856.6	63.80	02:00:00	–	–	–	02:00:00
kroA	200	40	114116	46091.6	59.61	02:00:00	442010	75495.36	82.92	02:00:00
kroA	200	75	212504	66209.2	68.84	02:00:00	–	–	–	02:00:00
kroA	200	100	274917	67317	75.51	02:00:00	–	–	–	02:00:00
lin	318	61	54721	20638.8	62.28	02:00:00	–	–	–	02:00:00
lin	318	111	100666	26664.8	73.51	02:00:00	–	–	–	02:00:00
lin	318	159	149070	32224.6	78.38	02:00:00	–	–	–	02:00:00

For the case where $k = 3$ and $L = 4$ (Table 4.3), we can see that CPLEX has reached the maximum CPU time while the CPU time required by PHA does not exceed 15 minutes for 45% of the instances. Also, except 5 instances, PHA outperforms CPLEX in producing good upper bounds. Moreover, CPLEX has been able to find even a feasible solution after 2 hours of CPU time while PHA does.

For the case where $k = 3$ and $L = 5$ (see Table 4.4), the observations are quite similar to the cases where $L = 3$ and $L = 4$. Indeed, it appears that PHA outperforms CPLEX in producing good upper bound, and this, within a quite short CPU time.

Now we compare PHA with each of its components. We do this comparison only in the case where $k = 3$ and $L = 3$, as our experiments lead to the same observations when $L = 4$ and $L = 5$. Table 4.5 below gives the lower and upper bounds obtained by PHA and LRA and the upper bounds obtained by GA and SH, for the kHNDP with

Table 4.3: Results for PHA and CPLEX for the kHNDP and arbitrary demands with $k = 3$ and $L = 4$.

Instances			PHA				CPLEX			
name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	30	10	10552	8836.4	16.26	00:00:22	9433	9433.0	0	01:46:10
berlin	30	15	13774	10638.4	22.76	00:00:39	11898	11194.7	5.91	02:00:00
berlin	52	10	10457	8396.0	19.71	00:01:24	9252	8903.7	3.76	02:00:00
berlin	52	15	13672	10283.6	24.78	00:02:20	12021	11011.2	8.4	02:00:00
berlin	52	20	16541	11844.6	28.39	00:03:12	15541	12787.4	17.72	02:00:00
st	70	15	1326	825.5	37.75	00:03:57	1346	910.4	32.37	02:00:00
st	70	26	2023	1043.2	48.43	00:07:13	2572	1263.9	50.86	02:00:00
st	70	35	2818	1209.6	57.08	00:08:44	5588	1613.4	71.13	02:00:00
kroA	100	20	60786	28152.7	53.69	00:14:11	71728	34511.5	51.89	02:00:00
kroA	100	35	100081	39413.7	60.62	00:25:33	–	–	–	02:00:00
kroA	100	50	141228	50301.7	64.38	00:37:25	386054	62125.6	83.91	02:00:00
kroA	150	30	89109	35455.9	60.21	00:50:58	204376	44932.4	78.01	02:00:00
kroA	150	50	141829	46229.2	67.40	01:27:57	–	–	–	02:00:00
kroA	150	75	204943	58525.0	71.44	02:00:00	–	–	–	02:00:00
kroA	200	40	107850	35872.6	66.74	02:00:00	–	–	–	02:00:00
kroA	200	75	209367	47121.1	77.49	02:00:00	–	–	–	02:00:00
kroA	200	100	269006	48113.7	82.11	02:00:00	–	–	–	02:00:00
lin	318	61	53359	17987.0	66.29	02:00:00	–	–	–	02:00:00
lin	318	111	98525	17786.1	81.95	02:00:00	–	–	–	02:00:00
lin	318	159	141712	15760.9	88.88	02:00:00	–	–	–	02:00:00

$k = 3$ and $L = 3$.

Table 4.5 clearly shows that PHA outperforms its LRA, GA and SH taken separately, for all the instances. Indeed, the upper bound produced by PHA is better for all the instances than that obtained by GA, LRA and SH. However, when comparing the lower bounds obtained by PHA and LRA, except 2 instances (kroA200-a100 and lin318-a159), the lower bound produced by LRA is better than that obtained by PHA. Consequently, as before, the hybridization clearly helps in producing better feasible solutions for the kHNDP but does not produce better lower bounds.

Table 4.4: Results for PHA and CPLEX for the kHNDP and arbitrary demands with $k = 3$ and $L = 5$.

Instances			PHA				CPLEX			
name	$\|V\|$	$\|D\|$	*UB*	*LB*	*Gap*	*CPU*	*UB*	*LB*	*Gap*	*CPU*
berlin	30	10	9989	8527.6	14.63	00:00:47	8897	8897.0	0.00	00:06:24
berlin	30	15	12651	10156.4	19.72	00:01:11	11397	10647.4	6.58	02:00:00
berlin	52	10	9967	8073.6	19.00	00:02:39	8855	8511.1	3.88	02:00:00
berlin	52	15	12848	9737.3	24.21	00:04:36	11197	10466.3	6.53	02:00:00
berlin	52	20	15870	11211.8	29.35	00:05:59	14981	12051.5	19.55	02:00:00
st	70	15	1242	733.4	40.95	00:06:38	1187	827.2	30.31	02:00:00
st	70	26	1808	954.7	47.20	00:10:18	4257	1120.7	73.68	02:00:00
st	70	35	2525	1172.1	53.58	00:15:33	–	–	–	02:00:00
kroA	100	20	56668	25532.4	54.94	00:23:19	71110	30451.0	57.18	02:00:00
kroA	100	35	90788	34670.5	61.81	00:40:52	–	–	–	02:00:00
kroA	100	50	125022	43359.5	65.32	00:59:12	–	–	–	02:00:00
kroA	150	30	80854	29698.5	63.27	01:19:47	–	–	–	02:00:00
kroA	150	50	126533	39383.0	68.88	02:00:12	–	–	–	02:00:00
kroA	150	75	188290	47148.4	74.96	02:00:28	–	–	–	02:00:00
kroA	200	40	102520	29630.6	71.10	02:00:13	–	–	–	02:00:00
kroA	200	75	193169	38005.0	80.33	02:01:52	–	–	–	02:00:00
kroA	200	100	246339	39859.9	83.82	02:02:30	–	–	–	02:00:00
lin	318	61	50570	14214.8	71.89	02:03:02	–	–	–	02:00:00
lin	318	111	94004	13372.0	85.78	02:07:55	–	–	–	02:00:00
lin	318	159	137215	12026.7	91.24	02:13:49	–	–	–	02:00:00

4.4 The Impact of the Parallelization

In the previous sections, we have shown that using the Lagrangian relaxation, genetic and greedy algorithms in parallel allows an improvment of the quality of the feasible solutions known for the kESNDP and kHNDP. However, an important question is wether it is relevant to use parallel computing inside each component of PHA. In this section, we investigate this question. Indeed, as mentioned above, solving each sub-problem of the Lagrangian relaxation (*i.e.* LRA) and the greedy (*i.e.* SH) algorithms reduces to $|D|$ independent minimum cost flow problems, which can be done in $O(|E|^2 \log |V| + |E||V| \log^2 |V|)$ using $|D|$ processors in parallel. Also, in the genetic algorithm (*i.e.* GA), parallel computing is used in the reproduction phase at each iteration, where, in parallel, we cross several pairs of parent solutions.

Table 4.5: Results for PHA versus LRA, SH and GA for the kHNDP and arbitrary demands with $k = 3$ and $L = 3$.

Instances			PHA		LRA		SH	GA
name	$\|V\|$	$\|D\|$	UB	LB	UB	LB	UB	UB
berlin	30	10	10695	9467,6	11914	9566	11756	11376
berlin	30	15	13743	11847,4	17198	12091	14854	14854
berlin	52	10	10298	9136	13715	9361	11359	10980
berlin	52	15	13416	11596,6	19042	11828	14508	14491
berlin	52	20	16534	13448,6	22341	13715	17633	17633
st	70	15	1389	971,2	2213	1038	1536	1424
st	70	26	2107	1312,4	3275	1421	2269	2192
st	70	35	2834	1714,4	4665	1822	2975	2957
kroA	100	20	62128	37704,6	110812	37839	66190	64081
kroA	100	35	103164	55049,4	183685	56581	111403	105224
kroA	100	50	144842	72889,6	269705	74624	151175	147820
kroA	150	30	93542	45123	165239	46682	101709	95986
kroA	150	50	144861	60073,2	270609	68616	149949	146986
kroA	150	75	209575	75856,6	420836	75891	214646	212735
kroA	200	40	114116	46091,6	223098	48169	117931	114989
kroA	200	75	212504	66209,2	437497	76827	218219	214878
kroA	200	100	274917	67317	565116	59827	278363	273934
lin	318	61	54721	20638,8	74755	21638,8	56285	55544
lin	318	111	100666	26664,8	137906	26664,8	103297	101115
lin	318	159	149070	32224,6	206681	30224,6	152685	150965

The question raised here is important since the CPU time needed to run an algorithm using several processors (for instance for accessing a shared memory, for managing the interactions with the operating system or between the processors, etc.) may exceed the CPU time when using a single processor.

To answer this question for our parallel hybrid algorithm, we measure the CPU time for each algorithm, LRA, SH and GA, regarding an increasing of the number of processors used to run the algorithms. We use the algorithms devised for the kHNDP with $k = 3$ and $L = 3$. Tables 4.6, 4.7 and 4.8 show the CPU time for LRA, SH and GA, respectively, obtained for a subset of 10 test instances. The CPU computation time presented corresponds to the mean of 5 runs CPU times. The different entries of the tables are:

p : number of processors(cores)
CPU : total CPU time in sec
S : speedup
E : efficiency

Figures 4.6, 4.7 and 4.8 below present the CPU time evolution curve for LRA, SH and GA, respectively, as a function of the number of processors used to run the algorithms.

We have limited the number of processors to 8 since the computer we have used for our experiments is equiped with a 8-cores processor.

Table 4.6: CPU computation time for LRA with p variation

name	$\|V\|$	$\|D\|$	LRA				
			$p=1$	$p=2$	$p=4$	$p=6$	$p=8$
berlin	30	10	10.225	7.851	7.739	7.765	7.652
	30	15	16.071	12.251	11.054	11.050	11.044
	52	15	62.390	48.752	45.806	44.113	42.912
	52	20	83.443	63.194	56.832	57.102	56.549
st	70	26	161.168	120.659	109.646	111.536	108.144
	70	35	221.793	165.477	153.701	152.331	146.576
kroA	100	35	573.306	439.237	401.064	393.803	377.244
	100	50	804.708	596.874	554.943	538.349	517.985
	150	50	1842.480	1421.370	1284.303	1234.657	1188.387
	150	75	2752.303	2023.787	1875.793	1835.800	1752.697

The three tables clearly show that, for the three algorithms, the CPU time decreases when the number of processors increases. Moreover, using two processors significantly decreases the CPU compared to the usage of a single processor. We can also notice

Table 4.7: CPU computation time for SH with p variation

name	$\|V\|$	$\|D\|$	SH				
			$p=1$	$p=2$	$p=4$	$p=6$	$p=8$
berlin	30	10	0.055	0.028	0.022	0.019	0.017
	30	15	0.116	0.063	0.045	0.042	0.031
	52	15	0.348	0.194	0.150	0.097	0.098
	52	20	0.604	0.320	0.227	0.186	0.151
st	70	26	1.815	0.959	0.645	0.559	0.474
	70	35	3.312	1.711	1.143	0.967	0.814
kroA	100	35	7.060	3.687	2.308	1.978	1.747
	100	50	14.185	7.332	4.889	3.858	3.383
	150	50	32.607	16.875	11.720	9.740	9.184
	150	75	72.581	37.480	24.576	19.415	17.141

Table 4.8: CPU computation time for GA with p variation

name	$\|V\|$	$\|D\|$	GA				
			$p=1$	$p=2$	$p=4$	$p=6$	$p=8$
berlin	30	10	12.367	8.122	7.593	7.534	7.618
	30	15	32.067	20.201	18.682	18.185	18.445
	52	15	114.795	70.800	66.572	62.597	61.592
	52	20	194.075	119.378	108.149	103.086	102.124
st	70	26	374.256	220.177	192.523	188.955	176.743
	70	35	711.533	416.747	362.847	356.526	335.136
kroA	100	35	1474.193	812.080	739.961	730.558	695.074
	100	50	2532.300	1393.757	1268.620	1258.657	1191.080
	150	50	5748.553	3161.837	2874.013	2804.983	2697.567
	150	75	7325.273	4007.933	3606.080	3501.723	3359.725

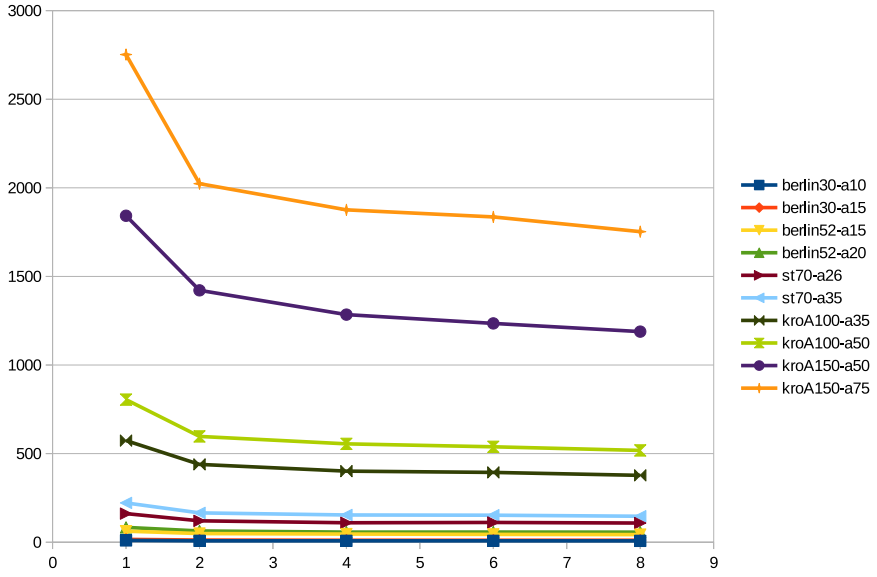

Figure 4.6: The CPU time of LRA w.r.t. to the number of processors

that, even if the CPU time decreases as the number of the processors increases, the gain of using height processors is not significant compared to using two processors. In conclusion of these experiments, using several processors improve the CPU time of each component of PHA, and using two processors seems to be the best choice.

It should be noticed that the above conclusion holds if we do not use more than height processors, and depends on architecture of the computer used for the experiments. Using a different architecture and more than height processors may lead to different conclusions.

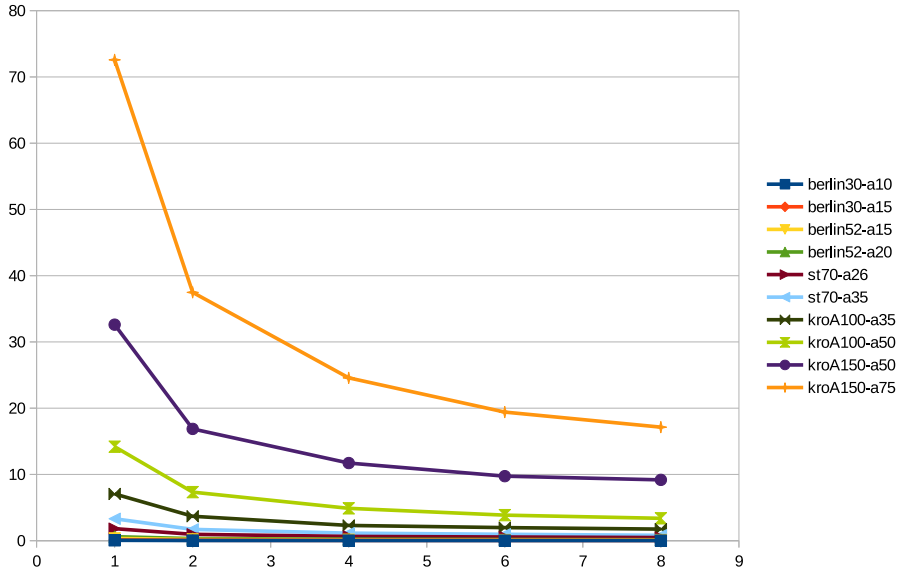

Figure 4.7: The CPU time of SH w.r.t. to the number of processors

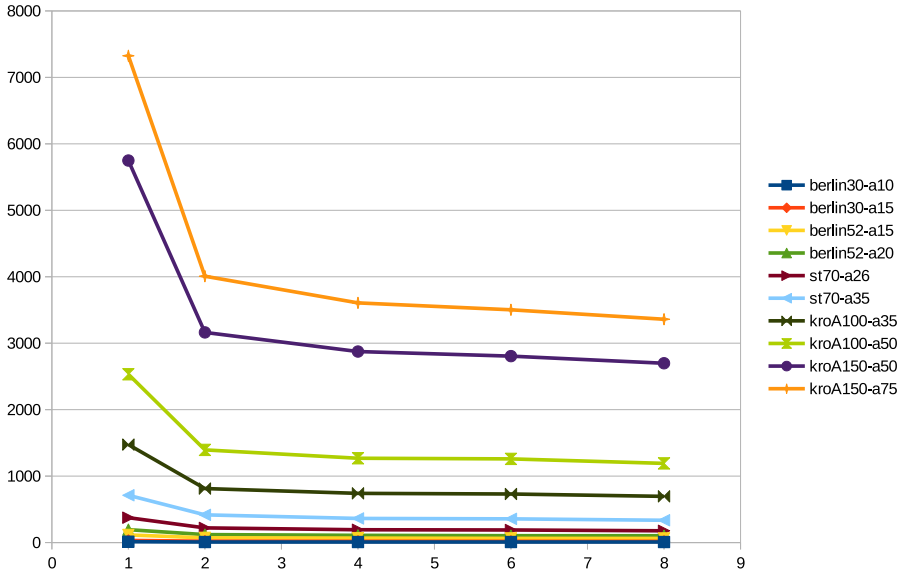

Figure 4.8: The CPU time of GA w.r.t. to the number of processors

4.4.1 Speedup & Efficiency

In what follows, we present the speedup and the efficiency results. Tables 4.9, 4.10 and 4.11 and Figures 4.9, 4.10 and 4.11 show the results and the evolution curve of the speedup and the efficiency in function of p for respectively LRA, SH and GA.

Table 4.9: LRA speedup and efficiency results

name	$\|V\|$	$\|D\|$	Speedup				Efficiency			
			$p=2$	$p=4$	$p=6$	$p=8$	$p=2$	$p=4$	$p=6$	$p=8$
berlin	30	10	1.302	1.321	1.317	1.336	0.651	0.330	0.219	0.167
	30	15	1.312	1.454	1.454	1.455	0.656	0.363	0.242	0.182
	52	15	1.280	1.362	1.414	1.454	0.640	0.341	0.236	0.182
	52	20	1.320	1.468	1.461	1.476	0.660	0.367	0.244	0.184
st	70	26	1.336	1.470	1.445	1.490	0.668	0.367	0.241	0.186
	70	35	1.340	1.443	1.456	1.513	0.670	0.361	0.243	0.189
kroA	100	35	1.305	1.429	1.456	1.520	0.653	0.357	0.243	0.190
	100	50	1.348	1.450	1.495	1.554	0.674	0.363	0.249	0.194
	150	50	1.296	1.435	1.492	1.550	0.648	0.359	0.249	0.194
	150	75	1.360	1.467	1.499	1.570	0.680	0.367	0.250	0.196

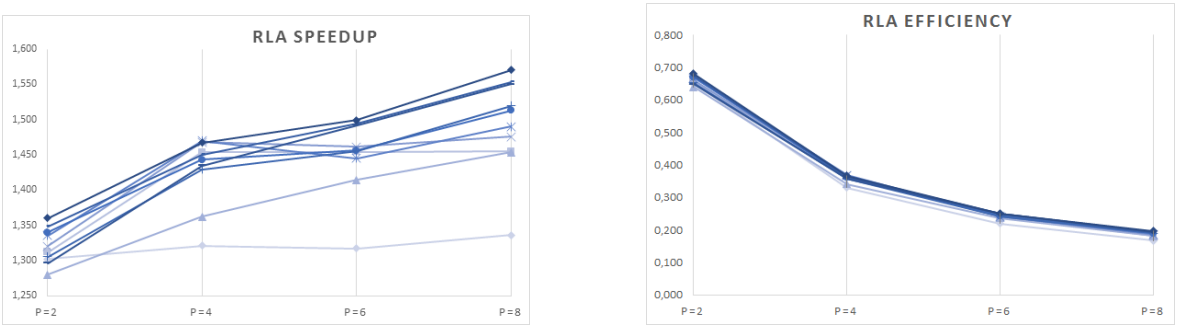

Figure 4.9: LRA speedup and efficiency

For a fixed graph and an increasing p, we remark that LRA speedup increases in the most of the cases, exept for 3 instances over 10 (30%) while we pass from using 4 to 6 cores. On the other side, efficiency is strictly decreasing.

When we fix a number of cores and variate the input graphs, both metrics are increasing when the input graph has greater numbers of nodes and demands, but decreases for a bigger graph and an equal number of demands $|D|$, and this behavior is logical because the number of parallelized tasks in every parallelized region of LRA depends of $|D|$, so if the number of nodes increases and not $|D|$, the algorithm will have to compute more, manage a bigger amount of data, without having more parallelism.

For SH, we remark that the speedup is always increasing for a fixed graph and an increasing p, the efficiency on the other hand, is strictly decreasing.

When we fix a number of cores and variate the input graphs, same as said for LRA,

Table 4.10: SH speedup and efficiency results

name	$\|V\|$	$\|D\|$	Speedup				Efficiency			
			$p=2$	$p=4$	$p=6$	$p=8$	$p=2$	$p=4$	$p=6$	$p=8$
berlin	30	10	1.987	2.566	2.969	3.173	0.994	0.641	0.495	0.397
	30	15	1.839	2.604	2.786	3.697	0.919	0.651	0.464	0.462
	52	15	1.795	2.323	3.578	3.568	0.898	0.581	0.596	0.446
	52	20	1.887	2.661	3.241	3.991	0.944	0.665	0.540	0.499
st	70	26	1.893	2.815	3.247	3.829	0.947	0.704	0.541	0.479
	70	35	1.935	2.898	3.426	4.070	0.968	0.724	0.571	0.509
kroA	100	35	1.915	3.058	3.569	4.041	0.957	0.765	0.595	0.505
	100	50	1.935	2.901	3.677	4.193	0.967	0.725	0.613	0.524
	150	50	1.932	2.782	3.348	3.551	0.966	0.696	0.558	0.444
	150	75	1.937	2.953	3.738	4.234	0.968	0.738	0.623	0.529

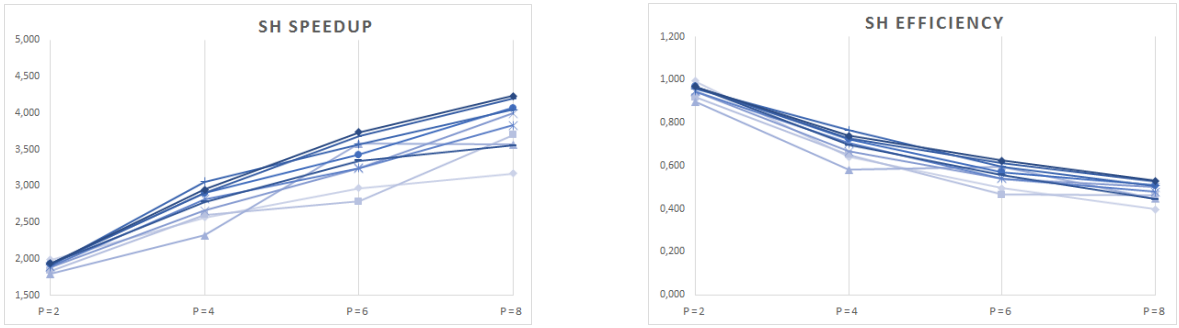

Figure 4.10: SH speedup and efficiency

for SH both metrics are increasing when the input graph has greater numbers of nodes and demands, but decreases for a bigger graph and an equal number of demands $|D|$. We notice also that SH in comparaison with LRA has a higher speedup, that is close from reaching for the perfect case $p = 2$ having $S \approx p$, and this is due to the higher degree of parallelism of SH.

If we wanted to sort LRA, SH and GA's parallelisation, GA would be in the second position after SH. In fact, we can see that the results obtained in terms of speedup and efficiency are better than LRA's ones. Here again we can easily see that the speedup is increasing for a fixed graph and an increasing p (we have only one irregular case, *berlin* 30 10 with p from 4 to 6 cores) and that the efficiency is always decreasing.

However, for a fixed number of cores and a variant input graphs, GA's speedup and efficiency were always increasing. This behavior can be explained by the fact that the number of parallel tasks depends from the solution population pool size (depends itself

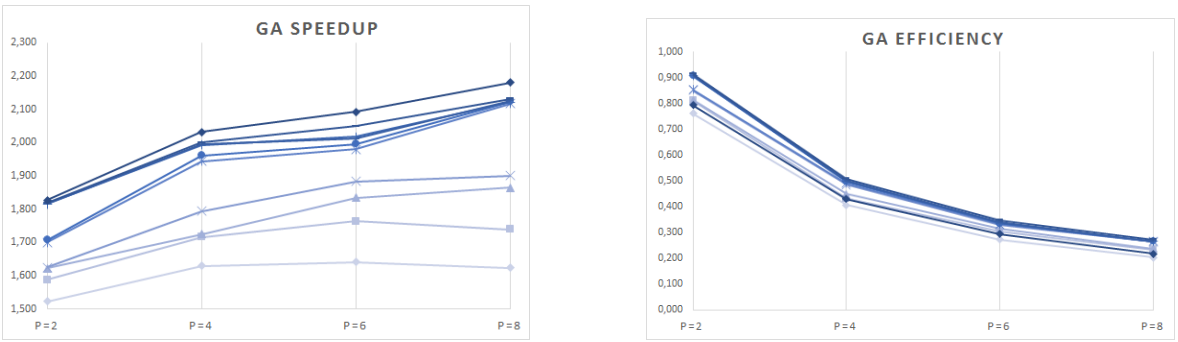

Figure 4.11: GA speedup and efficiency

from the number of nodes of a graph and the number of demands $|D|$) and not only from the number of demands $|D|$.

Table 4.11: GA speedup and efficiency results

name	$\|V\|$	$\|D\|$	Speedup				Efficiency			
			$p=2$	$p=4$	$p=6$	$p=8$	$p=2$	$p=4$	$p=6$	$p=8$
berlin	30	10	1.523	1.629	1.641	1.623	0.761	0.407	0.274	0.203
	30	15	1.587	1.716	1.763	1.739	0.794	0.429	0.294	0.217
	52	15	1.621	1.724	1.834	1.864	0.811	0.431	0.306	0.233
	52	20	1.626	1.795	1.883	1.900	0.813	0.449	0.314	0.238
st	70	26	1.700	1.944	1.981	2.118	0.850	0.486	0.330	0.265
	70	35	1.707	1.961	1.996	2.123	0.854	0.490	0.333	0.265
kroA	100	35	1.815	1.992	2.018	2.121	0.908	0.498	0.336	0.265
	100	50	1.817	1.996	2.012	2.126	0.908	0.499	0.335	0.266
	150	50	1.818	2.000	2.049	2.131	0.909	0.500	0.342	0.266
	150	75	1.828	2.031	2.092	2.180	0.914	0.508	0.349	0.273

4.5 Conclusion

In this Chapter we considered the k-Edge-Connected Hop-Constrained Network Design Problem (kHNDP). We presented the application of the approach PHA presented in the previous chapter for solving the kESNDP to the kHNDP and we tested the algorithms within an extensive computational study and compare their efficiency for solving the kHNDP against CPLEX.

Chapter 5

Parallelizing Branch-and-Cut algorithms for some survivability network design problems

Contents

We continue here our study of the k-edge connected Hop-constrained Network Design Problem (kHNDP). In this study we aim optimal solving of the kHNDP. We devise and present two distributed exact algorithms: a distributed Branch-and-Bound and a distributed Branch-and-Cut. We present after that, an experimental study in which we compare our algorithms to CPLEX.

The Branch-and-Bound and the Branch-and-Cut algorithms, are presented as a set of tasks with very few, if any, data dependencies. This implies that the processes that perform calculations on these data have few dependencies and can be scheduled in any order. The only relationship that can exist between two tasks is that of kinship. Thus the dependency graph is a tree that is not very usable. The computational times of the nodes of the tree can not be evaluated a priori.

It is therefore not easy to parallelize such applications efficiently since its irregularity prevents predicting the shape of the tree, its size and the amount of work that each node represents in the tree.

In this chapter, we present two distributed exact algorithms (a distributed Branch-and-Bound and a distributed Branch-and-Cut) to solve the *k-Edge-Connected Hop-Constrained Network Design Problem* (kHNDP for short). We also discuss the parallelization techniques and strategies that can be used to adapt the implementation we present to solve more efficiently combinatorial optimization problems that has other characteristics.

The chapter is organized as follows. In Section 5.1, we introduce the different parallelization strategies that can be applied to both, Branch-and-Bound and Branch-and-Cut algorithms. In Section 5.2, we present the kHNDP formulations and some valid inequalities for the problem. Then, in Section 5.3, we describe our parallel Branch-and-Bound and parallel Branch-and-Cut algorithms. Finally, in Section 5.4 we present the related experimental study.

5.1 Parallelization strategies

To parallelize a B&B or a B&C algorithm, three major granularity levels can be distinguished (Bouzgarrou in [51]):

- coarse grain granularity,
- medium grain,
- fine grain.

As more flexible but complicated schemes, one can think of a new parallelization approach that uses a nested parallelization hierarchy. The main idea behind that is to be able to hybridize two or more of the Bouzgarrou's granularities in order to implement a parallel algorithm in which the granularity can be adapted to the problem and/or the hardware specifications. In what follows we describe how these parallelization strategies can be applied on the B&B and the B&C algorithms.

5.1.1 Coarse grain

It consists in executing several B&C algorithms in parallel, one starting from an initial feasable solution computed by a primal heuristic, and the second an initial solution given by the approach PHA. During the execution the algorithms exchange informations aiming to accelerate one or both explorations.

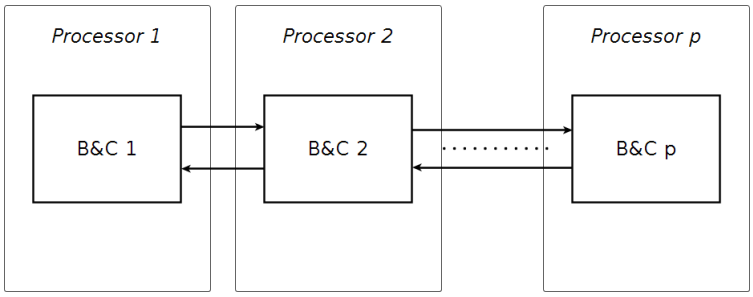

Figure 5.1: Coarse grain parallelization scheme

5.1.2 Medium grain

Each process explores a part of the search space. In this case we can use SYMPHONY to devise a parallel B&C in which we explore simultaneously multiple nodes of the search tree. Two types of threads exist, the first is the Master thread which handles the upper and lower bounds, the constraints pool and the sub-problems list. The second will be nodes threads that will be created by the master thread and deployed on a processor.

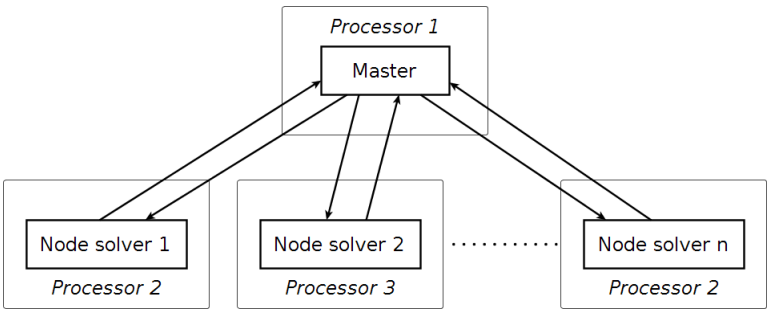

Figure 5.2: Medium grain parallelization scheme

5.1.3 Fine grain

In this strategy the evaluation of each node of the B&C tree is done in parallel by several processes. For this scheme we can handle simultaneously the separation of our valid inequalities. One can also think about improving such algorithms by designing new ones that can be massively parallel and implement them on GPUs.

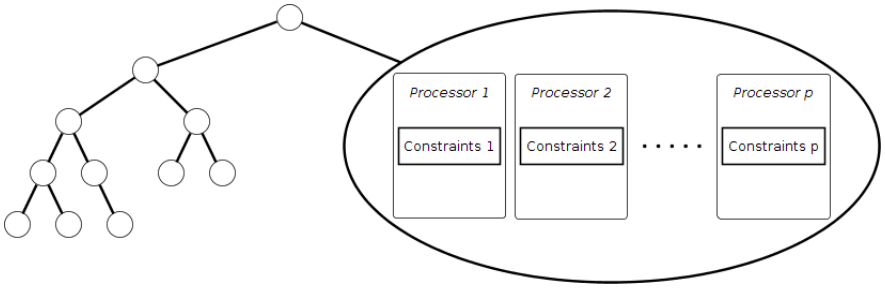

Figure 5.3: Fine grain parallelization scheme

5.1.4 Nested parallelization strategies

What is remarkable in the previous strategies is that they are quite rigid. In other words, these strategies are not and can not be efficient and adjusted according to the structure of the parallel computer available. For example, a coarse grain parallelization scheme of a B&B or a B&C for the kHNDP using multithreading techniques would not be efficient for all input graph sizes in practice. In fact, in our experimental studies presented in previous chapters, we have shown how exact approaches for the kHNDP are greedy in terms of memory. For this reason, running multiple exact algorithms in the same environment will probably slower down the algorithm and decrease probably the instances size limit that could be solved.

A different idea, that was not presented in the few works that studied the parallelization of the B&C, could be to combine these classic strategies to obtain new ones. Fine grain, for example, can be used with coarse grain. Another possible combination would be the one that combines fine grain with medium grain parallelization schemes. This later can be interesting for the kHNDP because of the amount of computation time and memory requirements of the exact algorithms components. As a direct consequence of this combination, the user would have the ability to adjust the deployment in function of the instance size/its solving hardness and/or the architecture of the parallel machine available for the computation.

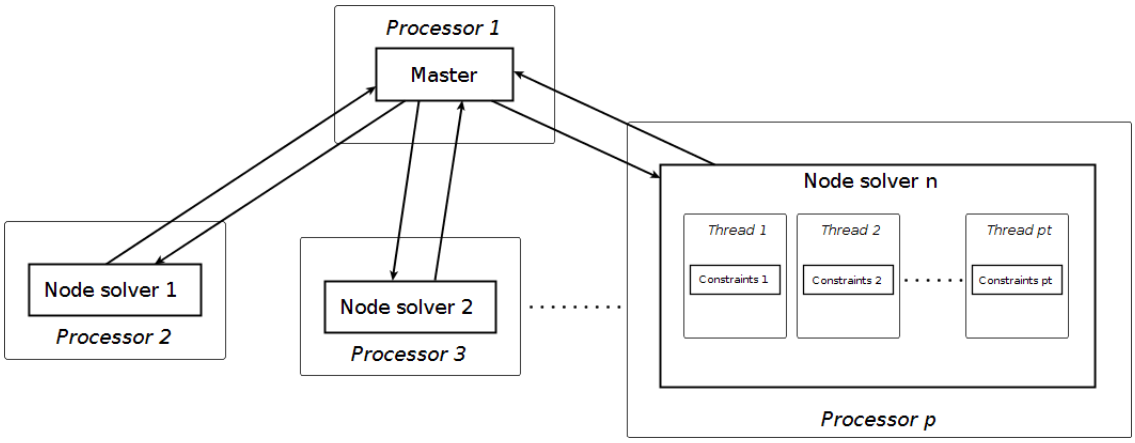

Figure 5.4: Medium/Fine grain parallelization scheme

5.2 Formulations & Valid inequalities

We recall that as application of our parallel exact algorithms, we have chosen to consider in this chapter the k-Edge-Connected Hop-Constrained Survivable Network Design Problem (kHNDP) when $L = 2, 3$. We recall that the problem is defined as follows. Given a weighted undirected graph $G = (V, E)$ where each edge e has a weight ω_e, a set of demands $D \subseteq V \times V$, a positive integer k, the kHNDP is to find a minimum weight subgraph of G such that for each demand $\{s, t\} \in D$, there exist k edge-disjoint paths of length at most L, for some integer $L \geq 2$, between s and t.

5.2.1 Formulations

Let $G = (V, E)$ be an undirected graph, a set of demands $D \subseteq V \times V$, a cost function $c : E \longrightarrow \mathbb{R}$, which associates the cost $c(e)$ with each edge $e \in E$. We recall that the *k-Edge-Connected Hop-Constrained Network Design Problem* (kHNDP for short) consists in finding a minimum cost subgraph of G such that there exist k-edge-disjoint L-st-paths between the terminals of each demand $\{s, t\}$ of D.

We have shown in Section 4.1.1 that the problem kHNDP can be formulated using some network flow sub-problems on transformed graphs when $L = 2, 3$. We have also presented the generalized flow formulation in Section 4.1.2 for the same problem when $L \geq 2$. In this section we will present two other formulations from the literature.

Notice that in what follows, we will denote the flow formulation for kHNDP when $L = 2, 3$ by $kHNDP^{2,3}_{flow}$, and the one for the problem when $L =\geq 2$ by $kHNDP_{flow}$.

5.2.1.1 Natural formulation

Let $L \geq 2$ and $D = \{\{s_1, t_1\}, ..., \{s_d, t_d\}\}, d \geq 2$, be the sets of demands. We will denote by R_D the set of terminal nodes of G, that is those nodes of G which are involved in at least one demand. It is clear that the incidence vector x^F of any solution (V, F) of the kHNDP satisfies the following inequalities:

$$x(\delta(W)) \geq k, \qquad \text{for all } st\text{-cut}, (s,t) \in D, \tag{5.1}$$

$$x(T) \geq k, \qquad \text{for all } L\text{-}st\text{-path-cut}, (s,t) \in D, \tag{5.2}$$

$$x(e) \geq 0, \qquad \text{for all } e \in E, \tag{5.3}$$

$$x(e) \leq 1, \qquad \text{for all } e \in E. \tag{5.4}$$

Conversely, any integer solution of the system defined by inequalities (5.1)-(5.4) is the incidence vector of a solution of the kHNDP when $L = 2, 3$.

Recall that inequalities (5.1)-(5.2) and (5.3)-(5.4) are called respectively *st-cut inequalities*, *L-st-path-cut inequalities* and *trivial inequalities*.

It is not hard to see that the kHNDP can be formulated as a linear integer program. The following lemma and theorem give this result.

Lemma 5.1 *Let $G = (V, E)$ be an undirected graph and s and t two nodes of V. Suppose that there do not exist k edge-disjoint L-st-paths in G, with $k \geq 2$. Then there exists a set of at most $k - 1$ edges that intersects every L-st-path.*

Theorem 5.2 *Let $G = (V, E)$ be a graph, $k \geq 2$ and $L \in \{2, 3\}$. Then the kHNDP is equivalent to the following inter program*

$$\min\{cx; \textit{subject to } (5.1) - (5.4), x \in \mathbb{Z}^E\}. \tag{5.5}$$

Formulation (5.10) is called *Natural formulation.* We will denote by $k\text{HNDP}_{Nat}$. It only uses the design variables.

5.2.1.2 Cut formulation

The Cut formulation is based on cuts in the graphs $\widetilde{G}_{st}$ (see 4.1.1.1 for the transformation details), $\{s, t\} \in D$. Given a subgraph $\widetilde{H}_{st}$ of $\widetilde{G}_{st}$, we let $y_{st}^{\widetilde{H}_{st}} \in \mathbb{R}^{\widetilde{A}_{st}}$ be the incidence vector of $\widetilde{H}_{st}$, that is to say $y_{st}^{\widetilde{H}_{st}}(a) = 1$ if $a \in \widetilde{H}_{st}$ and $y_{st}^{\widetilde{H}_{st}}(a) = 0$ if not. If a subgraph H of G induces a solution of the kHNDP, then the subgraph $\widetilde{H}_{st}$ contains at least k arc-disjoint st-dipaths, for all $\{s, t\} \in D$, and conversely. Thus, for any solution H of the kHNDP, the following inequalities are satisfied by $y_{st}^{\widetilde{H}_{st}}$, for all $\{s, t\} \in D$,

$$y_{st}(\delta^+(\widetilde{W})) \geq k, \qquad \text{for all } st\text{-dicut } \delta^+(\widetilde{W}) \text{ of } \widetilde{G}_{st}, \tag{5.6}$$
$$y_{st}(a) \leq x(e), \qquad \text{for all } a \in \widetilde{A}_{st}, e \in E, \tag{5.7}$$
$$y_{st}(a) \geq 0, \qquad \text{for all } a \in \widetilde{A}_{st}, \tag{5.8}$$
$$x(e) \leq 1, \qquad \text{for all } e \in E. \tag{5.9}$$

where $y_{st} \in \mathbb{R}^{\widetilde{A}_{st}}$ for all $\{s, t\} \in D$ and $x \in \mathbb{R}^E$.

Inequalities (5.6) will be called *directed st-cut inequalities* or *st-dicut inequalities* and inequalities (5.7) *linking inequalities.* Inequalities (5.7) indicate that an arc $a \in \widetilde{A}_{st}$ corresponding to an edge e is not in $\widetilde{H}_{st}$ if e is not taken in H. Inequalities (5.8) and (5.9) are called *trivial inequalities.*

We have the following result which is given without proof since it easily follows from the above results.

Theorem 5.3 *The kHNDP for $L = 2, 3$ is equivalent to the following integer program*

$$\min\{cx; \textit{subject to } (5.6) - (5.9), x \in \mathbb{Z}_+^E, y_{st} \in \mathbb{Z}_+^{\widetilde{A}_{st}}\}. \tag{5.10}$$

This formulation is called *Cut formulation* and denoted by kHNDP$_{Cu}$. It contains

$$|E| + \sum_{\{s,t\}\in D} |\widetilde{A}_{st}| = |E| + d(n-2) + \sum_{\{s,t\}\in D} |\delta(s)| + \sum_{\{s,t\}\in D} |\delta(t)| - \sum_{\{s,t\}\in D} |[s,t]|$$

variables if $L = 2$ and

$$|E| + \sum_{\{s,t\}\in D} |\widetilde{A}_{st}| = |E| + 2d|E| + d(n-2) - \sum_{\{s,t\}\in D} |\delta(s)| - \sum_{\{s,t\}\in D} |\delta(t)| + \sum_{\{s,t\}\in D} |[s,t]|$$

variables if $L = 3$ (remind that $d = |D|$).

However, the number of constraints is exponential since the directed st-cuts are in exponential number in $\widetilde{G}_{st}$, for all $\{s,t\} \in D$. As its is know that its linear relaxation can be solved in polynomial time using cutting plane algorithm.

5.2.2 Valid inequalities [95]

5.2.2.1 Double cut inequalities

In the following we introduce a class of inequalities that are valid for the kHNDP polytopes for $L \geq 2$ and $k \geq 2$. They are given by the following theorem.

Theorem 5.4 *Let $\{s,t\}$ be a demand, $i_0 \in \{0,...,L\}$ and $\prod = \{V_0,...,V_{i_0-1},V^1_{i_0},V^2_{i_0},V_{i_0+1},...,V_{L+1}\}$ a family of node sets of V such that $\pi = (V_0,...,V_{i_0-1},V^1_{i_0},V^2_{i_0} \cup V_{i_0+1},V_{i_0+2},...,V_{L+1})$ induces a partition of V. Suppose that*

1) *$V^1_{i_0} \cup V^2_{i_0}$ induces an $s_{j_1}t_{j_1}$-cut of G with $\{s_{j_1}t_{j_1}\} \in D$ and $s_{j_1} \in V^1_{i_0}$ or $t_{j_1} \in V^1_{i_0}$ (note that s_{j_1} and t_{j_1} cannot be simultaneously in $V^1_{i_0}$ and are not in $V^2_{i_0}$. Also note that $V^2_{i_0}$ may be empty);*

2) *V_{i_0+1} induces an $s_{j_2}t_{j_2}$-cut of G with $\{s_{j_2}t_{j_2}\} \in D$ (note that j_1 and j_2 may be equal);*

3) *π induces an L-st-path-cut of G with $s \in V_0$ (resp. $t \in V_0$) and $t \in V_{L+1}$ (resp. $s \in V_{L+1}$).*

Let $\overline{E} = [V_{i_0-1},V^1_{i_0}] \cup [V_{i_0+2},V^2_{i_0} \cup V_{i_0+1}] \cup \left(\bigcup_{k,l\notin\{i_0,i_0+1\},|k-l|>1}[V_k,V_l]\right)$ and $F \subseteq \overline{E}$ such that $|F|$ and k have different parities.

Let also $\widehat{E} = \left(\bigcup_{i=0}^{i_0-2}[V_i, V_{i+1}]\right) \cup \left(\bigcup_{i=i_0+2}^{L}[V_i, V_{i+1}]\right) \cup F$. *Then, the inequality*

$$x\left(\delta(\pi) \setminus \widehat{E}\right) \geq \left\lceil \frac{3k - |F|}{2} \right\rceil, \tag{5.11}$$

is valid for $kHNDP_{Nat}$, $kHNDP_{Cu}(G, D)$, $kHNDP^{2,3}_{flow}(G, D)$ *(recall that* $\delta(\pi)$ *is the set of edges of the* E *having their end nodes in different elements of* π*).*

5.2.2.2 Triple path-cut inequalities

Here is a further class of valid inequalities. We distinguish the cases where $L = 2$ and $L = 3$. We have the following theorem.

Theorem 5.5 *i) Let* $L = 2$ *and* $\{V_0, V_1, V_2, V_3^1, V_3^2, V_4^1, V_4^2\}$ *be a family of node sets of* V *such that* $(V_0, V_1, V_2, V_3^1 \cup V_3^2, V_4^1 \cup V_4^2$ *induces a partition of* V *and there exist two demands* $\{s_1, t_1\}$ *and* $\{s_2, t_2\}$ *with* $s_1, s_2 \in V_0$, $t_1 \in V_3^2$ *and* $t_2 \in V_4^2$. *The sets* V_3^1 *and* V_4^1 *may be empty and* s_1 *and* s_2 *may be the same. Let also* $V_3 = V_3^1 \cup V_3^2$, $V_4 = V_4^1 \cup V_4^2$ *and* $F \subseteq [V_3^2, V_1 \cup V_4^1] \cup [V_3^1, V_4^2]$ *such that* $|F|$ *and* k *have different parities. Then, the inequality*

$$\begin{aligned} &2x([V_0, V_2]) + x([V_0, V_3 \cup V_4]) + x([V_4^2, V_1 \cup V_3^2]) \\ &\quad + x(([V_3^2, V_1 \cup V_4^1] \cup [V_3^1, V_4^2]) \setminus F) \geq \left\lceil \frac{3k - |F|}{2} \right\rceil, \end{aligned} \tag{5.12}$$

ii) Let $L = 3$ *and* $\{V_0, ..., V_3, V_4^1, V_4^2, V_5^1, V_5^2\}$ *be a family of node sets of* V *such that* $(V_0, ..., V_3, V_4^1 \cup V_4^2, V_5^1 \cup V_5^2$ *induces a partition of* V *and there exist two demands* $\{s_1, t_1\}$ *and* $\{s_2, t_2\}$ *with* $s_1, s_2 \in V_0$, $t_1 \in V_4^2$ *and* $t_2 \in V_5^2$. *The sets* V_4^1 *and* V_5^1 *may be empty and* s_1 *and* s_2 *may be the same. Let also* $V_4 = V_4^1 \cup V_4^2$, $V_5 = V_5^1 \cup V_5^2$ *and* $F \subseteq [V_2, V_4^2] \cup [V_3, V_4 \cup V_5]$ *such that* $|F|$ *and* k *have different parities. Then, the inequality*

$$\begin{aligned} &2x([V_0, V_2]) + 2x([V_0, V_3]) + 2x([V_1, V_3]) + x([V_0 \cup V_1, V_4 \cup V_5]) + x([V_4, V_5]) \\ &\quad + x([V_2, V_5^2]) + x(([V_2, V_4^2] \cup [V_3, V_4 \cup V_5]) \setminus F) \geq \left\lceil \frac{3k - |F|}{2} \right\rceil, \end{aligned} \tag{5.13}$$

is valid for $kHNDP_{Nat}$, $kHNDP_{Cu}(G, D)$, $kHNDP^{2,3}_{flow}(G, D)$.

5.2.2.3 Steiner-partition inequalities

Let $(V_0, V_1, ..., V_p)$, $p \geq 2$, be a partition of V such that $V_0 \subseteq V \setminus R_D$, where R_D is the set of terminal nodes of G, and for all $i \in \{1, ..., p\}$ there is a demand $\{s, t\} \in D$ such that V_i induces an st-cut of G. Note that V_0 may be empty. Such a partition is called a $Steiner - partition$. With a Steiner-partition, we associate the inequality

$$x(\delta(V_0, V_1, ..., V_p)) \geq \left\lceil \frac{kp}{2} \right\rceil, \tag{5.14}$$

Inequalities of type (5.14) will be called *Steiner-partition inequalities*. We have the following result.

Theorem 5.6 *Inequality* (5.14) *is valid for* $kHNDP_{Nat}$, $kHNDP_{Cu}(G, D)$, $kHNDP^{2,3}_{flow}(G, D)$.

Inequality (5.14) expreses the fact that, in a solution of the kHNDP, the multicut induced by a Steiner-partition $(V_0, V_1, ..., V_p)$, $p \geq 2$, must contain at least $\lceil \frac{kp}{2} \rceil$ edges, since there must exist k edge-disjoint paths between every pair of nodes $\{s, t\} \in D$.

5.2.2.4 Steiner-SP-partition inequalities

Let $\pi = (V_1, ..., V_p)$, $p \geq 3$, be a partition of V such that the graph $G_\pi = (V_\pi, E_\pi)$ is series-parallel (G_π is the subgraph of G induced by π). Suppose that $V_\pi = \{v_1, ..., v_p\}$ where v_i is the node of G_π corresponding to the set V_i, $i = 1, ..., p$. The partition π is said to be a $Steiner - SP - partition$ if and only if π is a Steiner-partition and either

1) $p = 3$ or

2) $p \geq 4$ and there exists a node $v_{i_0} \in V_\pi$ incident to exactly two nodes $v_{i_0} - 1$ and $v_{i_0} + 1$ such that the partitions π_1 and π_2 obtained from π by contracting respectively the sets V_{i_0}, V_{i_0-1} and V_{i_0}, V_{i_0+1} are themselves Steiner-SP-partitions.

The procedure to check if a partition is a Steiner-SP-partition is recursive. It stops when the partition obtained after the different contractions is either a Steiner-partition and of size three or it is not a Steiner-partition.

In the following theorem, we give necessary and sufficient condition for a Steiner-partition to be a Steiner-SP-partition. Remind that the demand graph is denoted by $G_D = (R_D, E_D)$, where R_D is the set of terminal nodes of G. The edge set E_D is obtained by adding an edge between two nodes of R_D if and only if $\{u, v\} \in D$.

Theorem 5.7 *Let $\pi = (V_1, ..., V_p)$, $P \geq 3$, be a partition of V such that G_π is series-parallel. The partition π is a Steiner-SP-partition of G if and only if the subgraph of G_D induced by π is connected.*

As a consequence of Theorem 5.7, if the demand graph is connected (this is the case when, for instance, all the demands are rooted in the same node), then every Steiner-partition of V inducing a series-parallel subgraph of G is a Steiner-SP-partition of V.

With a Steiner-SP-partition $(V_1, ..., V_p)$, $p \geq 3$, we associate the following inequality

$$x(\delta(V_1, ..., V_p)) \geq \left\lceil \frac{k}{2} \right\rceil p - 1, \tag{5.15}$$

Inequalities of type (5.15) will be called *Steiner-SP-partition* inequalities. We have the following.

Theorem 5.8 *Inequality* (5.15) *is valid for $kHNDP_{Nat}$, $kHNDP_{Cu}(G, D)$, $kHNDP^{2,3}_{flow}(G, D)$.*

Inequality (5.15) expresses the fact that in a solution of the kHNDP the multicut induced by a Steiner-SP-partition contains at least $\lceil \frac{k}{2} \rceil p - 1$ edges, since this solution contains k edge-disjoint paths between every pair of nodes $\{s, t\} \in D$.

5.3 Parallel algorithms for the kHNDP

5.3.1 Sequential algorithms

In this section, we describe our sequential B&B and sequential B&C implementations on which our parallel algorithms are based.

5.3.1.1 Implementation of the B&B algorithm

Let P_{LF} be the linear relaxation of the undirected flow formulation (see Section 4.1.2 for more details). P_{LF} is defined as:

$$P_{LF}: \quad \min \sum_{uv \in E} \omega_{uv} x_{uv}$$

s.t.

$$\sum_{a \in \delta^+(u)} f_a^d - \sum_{a \in \delta^-(u)} f_a^d = \left\{ \begin{array}{rcl} k & if & u = s \\ -k & if & u = t \\ 0 & if & u \in \widetilde{V}_{st} \backslash \{s,t\}, \end{array} \right\},$$

$$\text{for all } u \in \widetilde{V}_{st} \text{ and } \{s,t\} \in D, \tag{5.16}$$

$$\sum_{a \in A_{st}(e)} f_a^{st} \leq x_e, \text{for all } e \in E \text{ and } \{s,t\} \in D, \tag{5.17}$$

$$f_a^{st} \geq 0, \text{ for all } a \in \widetilde{A}_{st} \text{ and } \{s,t\} \in D, \tag{5.18}$$

$$x_e \leq 1, \text{ for all } e \in E, \tag{5.19}$$

$$x_e \in \mathbb{R}, \text{ for all } e \in E, \tag{5.20}$$

$$f_a^{st} \in \mathbb{R}, \text{ for all } a \in \widetilde{A}_{st} \text{ and } \{s,t\} \in D. \tag{5.21}$$

It is obvious that the optimal solution of P_{LF} constitutes a lower bound of the integer kHNDP. Thus, to solve the kHNDP, our B&B algorithm starts by initializing the best upper bound Z_{UB} using the fast greedy algorithm SH presented in Section 4.2.1. Then, it creates a root node problem $P_{LF}^{n_{T_{B\&B}}^0=0}$, and add it to the nodes queue $T_{B\&B}$ that represents the search tree ($n_{T_{B\&B}}^k$ represents the number of nodes in the tree search $T_{B\&B}$ at iteration k). A search tree which is handled by a BEST-FIRST algorithm. Thus, this node is selected and solved using CPLEX as a black box LP solver. The algorithm tests, as after each node evaluation, if the solution is integer or fractional. If it is integer, the algorithm concludes updates the integer upper bound and continues with the node selection if the queue is not empty. If the solution is fractional, it selects a variable using a specific criteria, fixes it to 0 to obtain a new problem $P_{LF}^{n_{T_{B\&B}}^k+1}$ and to 1 to get $P_{LF}^{n_{T_{B\&B}}^k+2}$ with i representing the number of nodes in the tree at iteration k. Both $P_{LF}^{n_{T_{B\&B}}^k+1}$ and $P_{LF}^{n_{T_{B\&B}}^k+2}$ are added to the nodes queue. In addition, at each iteration k, if the solution found is fractional, we apply a greedy heuristic to make the solution integer and try to update/strengthen the global upper bound. Finally, we precise that the algorithm runs as long as $T_{B\&B} \neq \emptyset$.

Algorithm 8 summarizes our sequential Branch-and-Bound.

Algorithm 8: Sequential Branch-and-Bound for the kHNDP.

Data: An undirected graph $G = (V, E)$, the demand set D, a positive integers $k \geq 1$ and $L \geq 2$

Result: The optimal solution of the kHNDP, or a lower and upper bounds of it.

begin
- $Z_{UB} \leftarrow SH()$;
- $INSERT(T_{B\&B}, P^0_{LF})$;
- **while** $T_{B\&B}$ *is not empty* **do**
  - subproblem $p \leftarrow SELECT\_BEST(T_{B\&B})$;
  - $sol_k \leftarrow SOLVE(p)$;
  - **if** $IS\_INTEGER(sol_k) = True$ **then**
    - **if** $Z(sol_k) < Z_{UB}$ **then**
      - $Z_{UB} \leftarrow Z(sol_k)$;
    - **end**
  - **else**
    - **if** $Z(sol_k) \leq Z_{UB}$ **then**
      - $sol^I_k \leftarrow MAKE\_INTEGER(sol_k)$;
      - **if** $Z(sol^I_k) < Z_{UB}$ **then**
        - $Z_{UB} \leftarrow Z(sol^I_k)$;
      - **end**
      - $UPDATE(T_{B\&B}, Z_{LB}, Z(sol_k))$;
      - $P_{LF}^{n^k_{T_{B\&B}}+1}, P_{LF}^{n^k_{T_{B\&B}}+2} \leftarrow BRANCHING(sol_k)$;
      - $INSERT(T_{B\&B}, P_{LF}^{n^k_{T_{B\&B}}+1})$;
      - $INSERT(T_{B\&B}, P_{LF}^{n^k_{T_{B\&B}}+2})$;
    - **end**
  - **end**
  - **if** $Z_{UB} = Z_{LB}$ **then**
    - *stop*;
  - **end**
- **end**
- return (Z_{LB}, Z_{UB});

end

Notice that for the branching rule, we have tested three strategies on different instances: the first fractional x_e variable; the fractional variable x_e with the lowest reduced cost; the fractional variable x_e which is the closest to 0.5, and we have selected the one third strategy because of it has given the best results.

5.3.1.2 Implementation of the B&C algorithm

For the flow formulation, the optimization starts by considering the linear program P_{LF} presented in the previous section.

The optimal solution $(\overline{x}, \overline{f}^{s_1t_1}, ..., \overline{f}^{s_dt_d})$ is feasible for $k\text{HNDP}^{2,3}_{flow}$ if $(\overline{x}, \overline{f}^{s_1t_1}, ..., \overline{f}^{s_dt_d})$ is integral. If $(\overline{x}, \overline{f}^{s_1t_1}, ..., \overline{f}^{s_dt_d})$ is not feasible for the problem, then we generate, further valid inequalities for $k\text{HNDP}^{2,3}_{flow}(G, D)$ that are violated by $(\overline{x}, \overline{f}^{s_1t_1}, ..., \overline{f}^{s_dt_d})$. For this, we look for the following inequalities, in this order,

1) st-dicut inequalities,

2) double cut inequalities,

3) triple path-cut inequalities,

4) Steiner-partition inequalities,

5) Steiner-SP-partition inequalities.

Notice that the whole search tree management is done by our implementation of the the B&B algorithm presented in the previous section.

5.3.2 Parallel B&B

As mentioned above, the structure of the $B\&B$ contains very few dependencies, thus, the algorithm is inherently parallelizable. Nevertheless, the results of this task may be very bad if communications are not handled efficiently. That is all the more true when the memory is distributed.

Thereby, to avoid having an important overhead due to the distributed architecture, the parallelization scheme we have designed follows the master/slave model. In fact, having a centralized tree management reduces the number of communications that should be made. Therefore, the master process is the process responsible of the tree search management, the root node evaluation, the global upper and lower bounds management and the algorithm stopping management. The slave processes are the ones responsible of evaluating the tree nodes and the branching. Figure 5.5 describes the communication scheme of our parallel B&B algorithm.

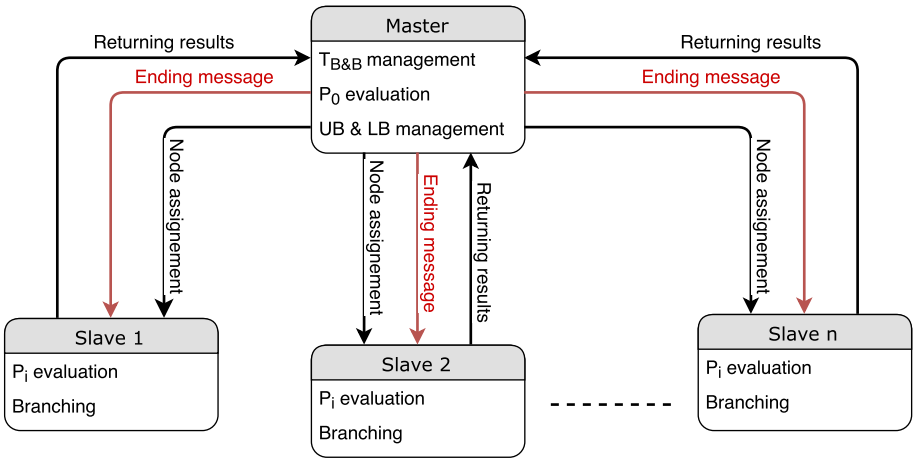

Figure 5.5: Distributed B&B communication scheme

In addition, to minimize the amount of data that is sent in each node assignment, we code a problem $P_{LF}^{n_{T_{B\&B}}^k}$ into $1+2\,level_{T_{B\&B}}\left(P_{LF}^{n_{T_{B\&B}}^k}\right)$ scalars ($level_{T_{B\&B}}(P)$ being the level in $T_{B\&B}$ of P) that represent the level of the node in the tree and the branchings set related to the $T_{B\&B}$ path seperating P_i and the root node P_0. Figure 5.6 describes an example of node P_5 affectation. From left to right, 3 represent the level of P_5 and the pairs $(7,1),(3,1),(1,0)$ the branchings.

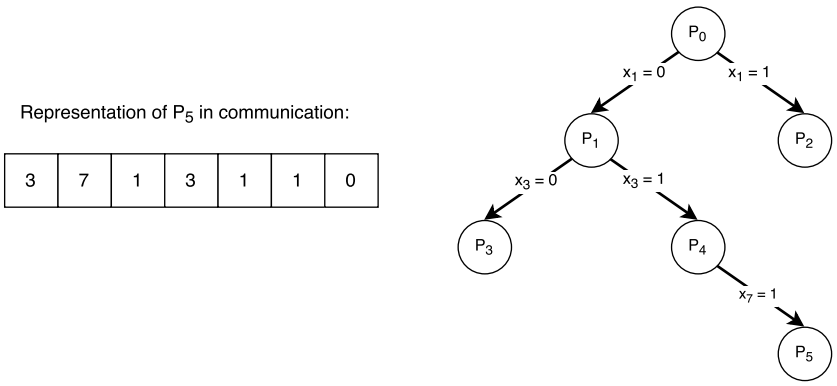

Figure 5.6: B&B node coding in communication

In the other way back, when a slave communicates the node evaluation results to a master, the message is essentially composed of 3 values: the P_i LP relaxation optimal value (lower bound), the primal bound calculated with the heuristic if the solution is fractional, and the variable on which we should branch if the node is not pruned by the master.

In a master/slave paradigm, load balancing is fair if one considers that the sub-problems (tasks) that are in the master's queue are non preemptive and if the processors are homogeneous. In fact, the master process gives work to the slaves according to a FIFO (First In First Out) order: The first free process is first served, and each time a process sends back results, the master responds with a new computation task.

Algorithm 9: Master process in parallel B&B for the kHNDP.

Data: An undirected graph $G = (V, E)$, the demand set D, a positive integers $k \geq 1$ and $L \geq 2$
Result: The optimal solution of the kHNDP, or a lower and upper bounds of it.

```
begin
    Z_UB ← SH();
    data_root ← SOLVE();
    if IS_FRACTIONAL(data_root) then
        PRIMAL_HEURISTIC(data_root);
        P^1_LF, P^{+2}_LF ← BRANCHING(data_root) ;
        INSERT(T_B&B, P^1_LF);
        INSERT(T_B&B, P^2_LF);
    end
    while T_B&B is not empty do
        AFFECT_TASKS(T_B&B);
        RECIEVE_RESULTS(results_in);
        if IS_INTEGER(results_in) then
            if Z(results_in) < Z_UB then
                Z_UB ← Z(results_in);
            end
        else
            if Z(results_in) ≤ Z_UB then
                sol^I_k ← MAKE_INTEGER(results_in);
                if Z(sol^I_k) < Z_UB then
                    Z_UB ← Z(sol^I_k);
                end
                UPDATE(T_B&B, Z_LB, Z(sol_k));
                BRANCH_AND_INSERT(T_B&B, results_in);
            end
        end
        if Z_UB = Z_LB then
            stop;
        end
    end
    return (Z_LB, Z_UB);
end
```

However, during the execution of a parallel B&B, the slaves are inactive during the start of the algorithm and during its final phase. This inactivity is due to the limited

Algorithm 10: Slave process in parallel B&B for the kHNDP.

```
Data: A T_{B&B} node, a branchings set
Result: The optimal solution of the node, an integer solution, a branching variable index.
begin
    while True do
        data_in ← RECIEVE_TASK();
        if IS_END_MSG(data_in) then
            stop;
        end
        DECODE_BRANCHINGS_SET();
        data_out ← SOLVE();
        if IS_FRACTIONAL(data_out) then
            PRIMAL_HEURISTIC(data_out);
            BRANCHING(data_out) ;
        end
        SEND_RESULTS(data_out) ;
    end
end
```

number of sub-problems that can be executed in a parallel way for a more or less important period of time. If such a phenomenon happens in practice, the main cause would be the slowness of the evaluation of the LP relaxation for the input graph.

5.3.3 Parallel B&C

As for the B&B algorithm, the real challenge in a parallel tree search algorithm is the choice of what, when and how many data we exchange during the optimization process. This is all the more true in a B&C algorithm that evaluates a node using linear program resolution(s) and polyhedron cutting.

The parallel B&C we have designed is based on the parallel B&B we introduced in the previous section. The major contribution that the B&C algorithm brings is the distributed cuts management. In fact, a trivial way to handle the separated cuts would be to separate the different valid constraints families in the slaves processes and then communicate the separated constraints to the master. This communication would happen while a slave sends back the node evaluation results. Nevertheless, since we are not able to estimate the cost of such a communication a priori, this method can cause a bottleneck to the application.

To respond to this limit, the parallel B&C we have designed uses, beyond the master process, a pool manager process that will be in charge of the cuts pool management.

Figure 5.7 shows the distributed B&C communication scheme.

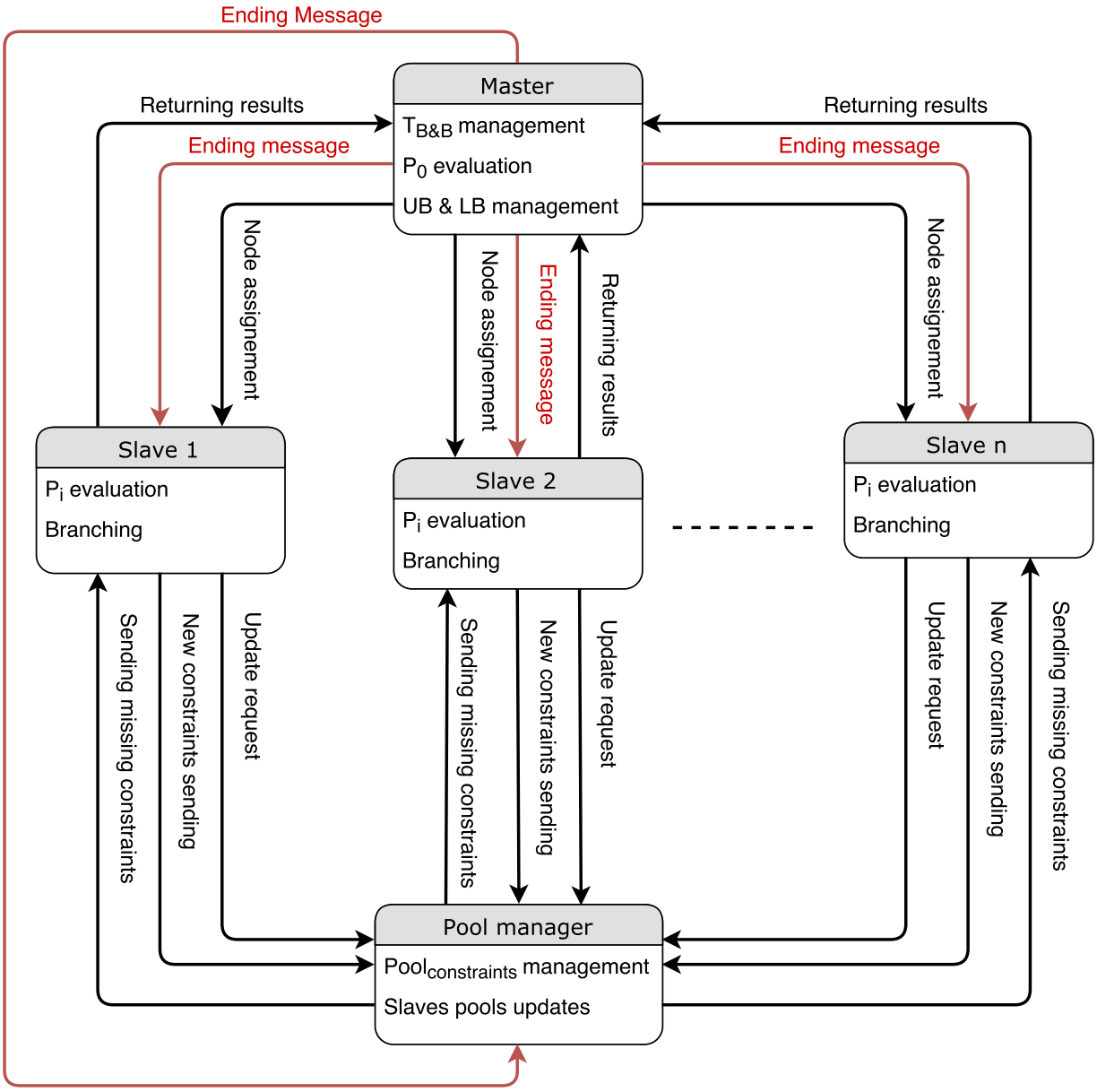

Figure 5.7: Distributed B&C communication scheme

In our parallel B&C, the master process is equivalent to the one presented for the parallel B&B. The only change we have made consists in adding the cuts separation for the root node evaluation. If this added step ends positively, the master sends the new separated constraints to the pool manager.

Algorithm 11: Master process in parallel B&C for the kHNDP.

Data: An undirected graph $G = (V, E)$, the demand set D, a positive integers $k \geq 1$ and $L \geq 2$
Result: The optimal solution of the kHNDP, or a lower and upper bounds of it.

```
begin
    Z_UB ← SH();
    NB_cons ← 0;
    repeat
        if NB_cons > 0 then
            SEND_CONSTRAINTS(data_out);
        end
        data_out, NB_cons ← SOLVE();
    until NB_cons = 0;
    if IS_FRACTIONAL(data_root) then
        PRIMAL_HEURISTIC(data_root);
        P^1_LF, P^{+2}_LF ← BRANCHING(data_root) ;
        INSERT(T_B&B, P^1_LF);
        INSERT(T_B&B, P^2_LF);
    end
    while T_B&B is not empty do
        AFFECT_TASKS(T_B&B);
        RECIEVE_RESULTS(results_in);
        if IS_INTEGER(results_in) then
            if Z(results_in) < Z_UB then
                Z_UB ← Z(results_in);
            end
        else
            if Z(results_in) ≤ Z_UB then
                sol^I_k ← MAKE_INTEGER(results_in);
                if Z(sol^I_k) < Z_UB then
                    Z_UB ← Z(sol^I_k);
                end
                UPDATE(T_B&B, Z_LB, Z(sol_k));
                BRANCH_AND_INSERT(T_B&B, results_in);
            end
        end
        if Z_UB = Z_LB then
            stop;
        end
    end
    return (Z_LB, Z_UB);
end
```

In the slaves processes, each time the process has to solve the node LP, he sends a request to the pool manager to check if his local constraints pool is up to date. If not, the pool manager sends back the missing constraints (that has been separated by the other processes).

Algorithm 12: Slave process in parallel B&C for the kHNDP.

```
begin
    while True do
        data_in ← RECIEVE_TASK();
        if IS_END_MSG(data_in) then
            stop;
        end
        DECODE_BRANCHINGS_SET();
        NB_cons ← 0;
        repeat
            if NB_cons = 0 then
                UPDATE_POOL(pool^l_cons);
            else
                SEND_CONSTRAINTS(data_out);
                UPDATE_POOL(pool^l_cons);
            end
            data_out, NB_cons ← SOLVE();
        until NB_cons = 0;
        if IS_FRACTIONAL(data_out) then
            PRIMAL_HEURISTIC(data_out);
            BRANCHING(data_out) ;
        end
        SEND_RESULTS(data_out) ;
    end
end
```

In order to decrease the amount of data to be sent, we assumed that the global constraints pool (stored in the pool manager memory) and the local constraints pools (stored in the slaves memories) are stored in the same order. This information allows the pool manager process to know, at any time, the size of the pool of each slave process, and therefore what are the constraints that are missing in his local pool. For such an assumption to be true, each slave has to update his local constraints pool before adding to it any new separated constraint. The algorithm is presented in what follows.

At a manager/slave communication point of view, the constraints are coded into $2\,nz(C_i)+3$ scalars ($nz(i)$ gives the number of the non-zero coefficients in the constraint i), representing in order:

- the number of integers that has been sent;
- $nz(C_i)$ pairs of scalars of the form $\{c_i, i\}$ indicating the non-zero coefficient of the variable of index i;

Algorithm 13: Pool manager process in parallel B&C for the kHNDP.

```
begin
    while True do
        data_in ← RECEIVE_REQUEST();
        if IS_END_MSG(data_in) then
            stop;
        end
        if IS_CLAIMANT_MSG(data_in) then
            SEND_CONSTRAINTS(data_in, pool_cons);
        end
        if IS_FEEDING_MSG(data_in) then
            SEND_CONSTRAINTS(data_in, pool_cons);
            RECEIVE_CONSTRAINTS(data_in, pool_cons);
        end
    end
end
```

- a scalar that represent the comparing sign;
- a scalar related to the right hand side of the constraint.

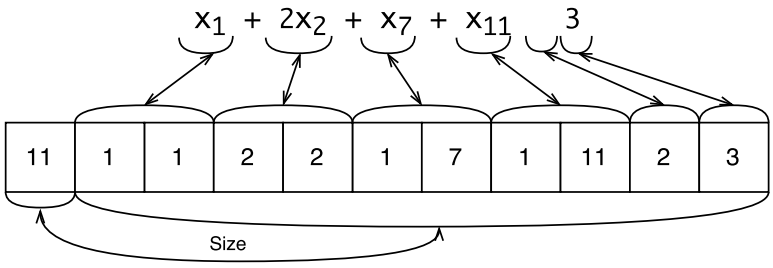

Figure 5.8: B&C constraint coding in communication

5.4 Experimental study

All the algorithms have been implemented in C++ and we have used CPLEX (12.6) for solving the undirected flow formulation for the kHNDP. The experiments have been conducted on a cluster composed of: 1 master machine equipped with an Intel Xeon E5-2603 v3 processor at 6×1.60 Ghz with 126 Gb of RAM, and 9 homogenous slaves. Each slave is equipped with Inter Xeon E5-2630 v3 processor at 16×2.40 Ghz with 48 Gb of RAM. All the machines are interconnected with a Gigabit Ethernet (GbE), and are running under Linux. We have set the maximum CPU time to 5 hours.

We recall that the test problems were obtained by taking TSP test problems from the TSPLIB library [2]. The test set consists in complete graphs whose edge weights are the rounded Euclidean distance between the edge's vertices. In addition, for each instance, a set of $|D|$ demand in the form $\{i, i+1\}$ is defined including the first $2|D|$ nodes.

5.4.1 kHNDP solving study

In this section we present the computational experiments we have conducted for the kHNDP with $k = 3$ and $L = 3$. The aim is to show the efficiency of our algorithms in solving mid/large size instances of the kHNDP. We recall that our comparaison criteria sorted in increasing order priorities are: the UB, the Gap, the CPU time.

To do this, we compare our distributed B&B to the best known method in the literature for these instances, which is CPLEX as a parallel black-box solver (with its inner parallelism) applied on the compact kHNDP flow formulation.

The entries of the tables are

$\|V\|$:	number of nodes of the graph,
$\|D\|$:	number of demands,
UB:	best upper bound achieved by parallel B&B (resp. CPLEX),
LB:	best lower bound achieved by parallel B&B (resp. CPLEX),
Gap:	relative error between the best upper and lower bounds achieved by parallel B&B (resp. CPLEX),
CPU:	total CPU time in hours:min:sec achieved by parallel B&B (resp. CPLEX).

Notice that when CPLEX or our algorithm do not solve the linear relaxation of the problem after the maximum CPU time (5 hours), the results are indicated with "–".

5.4.1.1 Parallel B&B results

Tables 5.1 give the results obtained by parallel B&B and CPLEX for the kHNDP with $k = 3$ and $L = 3$ and instances with arbitrary demands.

From Table 5.1, we can see that when $k = 3$ and $L = 3$, CPLEX was able to solve to optimality 5 instances over 28, while our parallel B&B was able to solve just 4 (instance $berlin\_30\_d15$ reached 8.1% after 5 hours). Also, our parallel B&B outperforms CPLEX in producing upper bounds for 16 instances over 28. For the others 5 instances where CPLEX produced better upper bounds, the solutions produced by our algorithm were close (at most 11.21% higher) to those obtained by CPLEX relatively to the other way around (CPLEX upper bounds which are 88.46% higher than the ones produced by our algorithm). From the lower bounds point of view, both our parallel B&B and CPLEX outperformed equally each other in producing better lower bounds in 5 instances over 28. We also notice that our parallel B&B was able to produce an upper bound for 4 instances while CPLEX fails to do so, even after 5 hours of CPU time.

5.4.1.2 Parallel B&C results

Tables 5.2 and 5.3 give the results obtained by parallel B&C and CPLEX for the kHNDP with $k = 3$ and $L = 3$ and instances with arbitrary demands.

5.4.1.2.1 Flow formulation

From Table 5.2, we can see that for the kHNDP when $k = 3$ and $L = 3$, CPLEX was able to solve to optimality 5 instances over 28, while our parallel B&C applied to the flow formulation was able to solve just 4 (instance $berlin\_30\_d15$ reached 6.21% after 5 hours). We notice that our parallel B&C outperforms CPLEX in producing upper bounds for 15 instances over 28. For the others 6 instances where CPLEX produced better upper bounds, the solutions produced by our algorithm were close

Table 5.1: Results for parallel B&B and CPLEX for the kHNDP with $k = 3$ and $L = 3$.

Instances			Parallel B&B				CPLEX			
name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	10	3	3678	3678.0	0	0.1	3678	3678.0	0	0.2
berlin	10	5	6019	6019.0	0	0.1	6019	6019.0	0	0.4
berlin	30	10	10254	10254.0	0	128.9	10254	10254.0	0	16.9
berlin	30	15	13584	12484.0	8.1	18004.7	13246	13244.7	0.01	6474.7
berlin	52	10	9919	9919.0	0	142.7	9919	9919.0	0	22.9
berlin	52	20	17000	14615.0	14.03	18122.4	15658	14750.8	5.79	18036.2
berlin	52	26	21991	18059.0	17.88	18244.3	20115	18136.6	9.84	18000.1
st	70	15	1442	1126.7	21.87	18070.6	1309	1131.2	13.58	18000.1
st	70	26	2230	1616.0	27.53	18735.8	1980	1606.6	18.86	18000.2
st	70	35	3013	2150.1	28.64	18697.1	3242	2142.9	33.9	18000.3
kroA	100	20	60216	45021.5	25.23	18842.5	62296	44859.8	27.99	18000.2
kroA	100	35	111458	69284.6	37.84	18244.3	115973	69083.6	40.43	18001.2
kroA	100	50	150382	92923.4	38.21	18646.7	318353	92958.0	70.8	18000.7
kroA	150	30	101756	62173.6	38.9	18450.3	116442	62086.9	46.68	18001.1
kroA	150	50	150210	92781.1	38.23	18075.6	382754	92781.1	75.76	18000.8
kroA	150	75	214773	130849.0	39.08	18429.7	552137	130849.0	76.3	18001.0
kroA	200	40	118001	75495.4	36.02	18105.6	292715	75495.4	74.21	18001.1
kroA	200	75	218351	0.0	100	18965.3	582242	0.0	100	18002.8
kroA	200	100	278531	0.0	100	18916.0	742827	0.0	100	18007.5
lin	318	61	56355	0.0	100	18280.8	488647	0.0	100	18006.4
lin	318	111	103453	0.0	100	18378.9	750205	0.0	100	18113.2
lin	318	159	152861	0.0	100	18580.1	–	–	–	–
pr	439	99	137614	0.0	100	18716.5	–	–	–	–
pr	439	151	198714	0.0	100	18776.7	–	–	–	–
pr	439	219	361513	0.0	100	18774.3	–	–	–	–
rat	575	121	–	–	–	–	–	–	–	–
rat	575	201	–	–	–	–	–	–	–	–
rat	575	287	–	–	–	–	–	–	–	–

(at most 10.37% higher) to those obtained by CPLEX relatively to the other way around (CPLEX upper bounds which are 88.47% higher than the ones produced by our algorithm). From the lower bounds point of view, both our parallel B&C and CPLEX outperformed almost equally each other in producing better lower bounds in 6 (our algorithm) and 5 (CPLEX) instances over 28. We also notice that our parallel B&C was able to produce an upper bound for 4 instances while CPLEX fails to do so, even after 5 hours of CPU time.

Table 5.2: Results for parallel B&C (flow formulation) and CPLEX for the kHNDP with $k = 3$ and $L = 3$.

Instances			Parallel B&C Flow				CPLEX			
name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	10	3	3678	3678.0	0.00	0.1	3678	3678.0	0.00	0.2
berlin	10	5	6019	6019.0	0.00	0.2	6019	6019.0	0.00	0.4
berlin	30	10	10254	10254.0	0.00	217.6	10254	10254.0	0.00	16.9
berlin	30	15	13593	12748.4	6.21	18000.0	13246	13244.7	0.01	6474.7
berlin	52	10	9919	9919.0	0.00	1697.6	9919	9919.0	0.00	22.9
berlin	52	20	16659	14741.7	11.51	18000.0	15658	14750.8	5.79	18036.2
berlin	52	26	21563	18208.4	15.56	18000.0	20115	18136.6	9.84	18000.1
st	70	15	1433	1130.3	21.12	18000.0	1309	1131.2	13.58	18000.1
st	70	26	2209	1643.7	25.59	18000.0	1980	1606.6	18.86	18000.2
st	70	35	3013	2147.3	28.73	18000.0	3242	2142.9	33.90	18000.3
kroA	100	20	66216	44949.6	32.12	18000.0	62296	44859.8	27.99	18000.2
kroA	100	35	111458	69124.5	37.98	18000.0	115973	69083.6	40.43	18001.2
kroA	100	50	150382	92826.0	38.27	18000.0	318353	92958.0	70.80	18000.7
kroA	150	30	98775	61976.1	37.26	18000.0	116442	62086.9	46.68	18001.1
kroA	150	50	150210	92781.8	38.23	18000.0	382754	92781.1	75.76	18000.8
kroA	150	75	214773	130849.0	39.08	18000.0	552137	130849.0	76.30	18001.0
kroA	200	40	118001	75497.5	36.02	18000.0	292715	75495.4	74.21	18001.1
kroA	200	75	218351	0.0	100.00	18000.0	582242	0.0	100.00	18002.8
kroA	200	100	278531	0.0	100.00	18000.0	742827	0.0	100.00	18007.5
lin	318	61	56355	0.0	100.00	18000.0	488647	0.0	100.00	18006.4
lin	318	111	103453	0.0	100.00	18000.0	750205	0.0	100.00	18113.2
lin	318	159	152861	0.0	100.00	18000.0	–	–	–	–
pr	439	99	137614	0.0	100.00	18000.0	–	–	–	–
pr	439	151	198714	0.0	100.00	18000.0	–	–	–	–
pr	439	219	361513	0.0	100.00	18000.0	–	–	–	–
rat	575	121	–	–	–	–	–	–	–	–
rat	575	121	–	–	–	–	–	–	–	–
rat	575	121	–	–	–	–	–	–	–	–

From Table 5.3 can see that our parallel B&C outperforms our parallel B&B in producing upper bounds in 5 instances over 28 while parallel B&B outperforms parallel B&C in just 2 cases. In terms of lower bounds we notice that parallel B&C produces better lower bounds in 8 instances over the 28, and parallel B&B in 5. What is interesting also to see is that the 8 instances in which parallel B&C produced better lower bounds where in majority the small-middle range instances. This is quite logic because our algorithm tends to explore more rapidly the search tree and thus is able to produce more violated constraints that strengthen the linear relaxation of the problem.

Table 5.3: Comparaison of parallel B&C (flow formulation), parallel B&B and CPLEX for the kHNDP with $k = 3$ and $L = 3$.

Instances			Parallel B&B			Parallel B&C Flow			CPLEX		
name	$\|V\|$	$\|D\|$	UB	LB	Gap	UB	LB	Gap	UB	LB	Gap
berlin	10	3	3678	3678.0	0.00	3678	3678.0	0.00	3678	3678.0	0.00
berlin	10	5	6019	6019.0	0.00	6019	6019.0	0.00	6019	6019.0	0.00
berlin	30	10	10254	10254.0	0.00	10254	10254.0	0.00	10254	10254.0	0.00
berlin	30	15	13584	12484.0	8.10	13593	12748.4	6.21	13246	13244.7	0.01
berlin	52	10	9918	9918.0	0.00	9919	9919.0	0.00	9919	9919.0	0.00
berlin	52	20	17000	14615.0	14.03	16659	14741.7	11.51	15658	14750.8	5.79
berlin	52	26	21991	18059.0	17.88	21563	18208.4	15.56	20115	18136.6	9.84
st	70	15	1442	1126.7	21.87	1433	1130.3	21.12	1309	1131.2	13.58
st	70	26	2230	1616.0	27.53	2209	1643.7	25.59	1980	1606.6	18.86
st	70	35	3013	2150.1	28.64	3013	2147.3	28.73	3242	2142.9	33.90
kroA	100	20	66216	45021.5	32.01	66216	44949.6	32.12	62296	44859.8	27.99
kroA	100	35	111458	69284.6	37.84	111458	69124.5	37.98	115973	69083.6	40.43
kroA	100	50	150382	92923.4	38.21	150382	92826.0	38.27	318353	92958.0	70.80
kroA	150	30	101756	62173.6	38.90	98775	61976.1	37.26	116442	62086.9	46.68
kroA	150	50	150210	92781.1	38.23	150210	92781.8	38.23	382754	92781.1	75.76
kroA	150	75	214773	130849.0	39.08	214773	130849.0	39.08	552137	130849.0	76.30
kroA	200	40	118001	75495.4	36.02	118001	75497.5	36.02	292715	75495.4	74.21
kroA	200	75	218351	0.0	100.00	218351	0.0	100.00	582242	0.0	100.00
kroA	200	100	278531	0.0	100.00	278531	0.0	100.00	742827	0.0	100.00
lin	318	61	56355	0.0	100.00	56355	0.0	100.00	488647	0.0	100.00
lin	318	111	103453	0.0	100.00	103453	0.0	100.00	750205	0.0	100.00
lin	318	159	152861	0.0	100.00	152861	0.0	100.00	–	–	–
pr	439	99	137614	0.0	100.00	137614	0.0	100.00	–	–	–
pr	439	151	198714	0.0	100.00	198714	0.0	100.00	–	–	–
pr	439	219	361513	0.0	100.00	361513	0.0	100.00	–	–	–
rat	575	121	–	–	–	–	–	–	–	–	–
rat	575	201	–	–	–	–	–	–	–	–	–
rat	575	287	–	–	–	–	–	–	–	–	–

Finally we can see easily that the limit of producing lower bounds of our parallel B&C is the same as for CPLEX and parallel B&B. This is basically due to the fact that our algorithm is based on CPLEX as linear solver, and the lack of memory limit using this formulation is the same because we use the same initial number of variables and constraints with CPLEX.

5.4.1.2.2 Natural formulation

Table 5.4 shows that for the kHNDP when $k = 3$ and $L = 3$, while CPLEX solved to optimality 5 instances over 28, our parallel B&C applied to the natural formulation

Table 5.4: Results for parallel B&C (natural formulation) and CPLEX for the kHNDP with $k = 3$ and $L = 3$.

Instances			Parallel B&C Natural				CPLEX			
name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	10	3	3678	3678.0	0.00	0.0	3678	3678.0	0.00	0.2
berlin	10	5	6019	6019.0	0.00	0.3	6019	6019.0	0.00	0.4
berlin	30	10	10891	7141.0	34.43	18000.0	10254	10254.0	0.00	16.9
berlin	30	15	14264	9318.3	34.67	18000.0	13246	13244.7	0.01	6474.7
berlin	52	10	11376	6824.0	40.01	18000.0	9919	9919.0	0.00	22.9
berlin	52	20	17657	7835.7	55.62	18000.0	15658	14750.8	5.79	18036.2
berlin	52	26	23242	8076.5	65.25	18000.0	20115	18136.6	9.84	18000.1
st	70	15	1548	588.3	62.00	18000.0	1309	1131.2	13.58	18000.1
st	70	26	2298	844.3	63.26	18000.0	1980	1606.6	18.86	18000.2
st	70	35	3013	796.6	73.56	18000.0	3242	2142.9	33.90	18000.3
kroA	100	20	66216	18668.2	71.81	18000.0	62296	44859.8	27.99	18000.2
kroA	100	35	111458	28380.7	74.54	18000.0	115973	69083.6	40.43	18001.2
kroA	100	50	150382	35170.2	76.61	18000.0	318353	92958.0	70.80	18000.7
kroA	150	30	101756	23347.5	77.06	18000.0	116442	62086.9	46.68	18001.1
kroA	150	50	150210	31409.7	79.09	18000.0	382754	92781.1	75.76	18000.8
kroA	150	75	214773	55676.4	74.08	18000.0	552137	130849.0	76.30	18001.0
kroA	200	40	118001	31118.8	73.63	18000.0	292715	75495.4	74.21	18001.1
kroA	200	75	218351	44778.9	79.49	18000.0	582242	0.0	100.00	18002.8
kroA	200	100	278531	51515.9	81.50	18000.0	742827	0.0	100.00	18007.5
lin	318	61	56355	19341.2	65.68	18000.0	488647	0.0	100.00	18006.4
lin	318	111	103453	34216.4	66.93	18000.0	750205	0.0	100.00	18113.2
lin	318	159	152861	47923.4	68.65	18000.0	–	–	–	–
pr	439	99	137704	49855.8	63.79	18000.0	–	–	–	–
pr	439	151	198714	72979.2	63.27	18000.0	–	–	–	–
pr	439	219	361513	110875.0	69.33	18000.0	–	–	–	–
rat	575	121	9033	3421.8	62.12	18000.0	–	–	–	–
rat	575	201	14750	5584.3	62.14	18000.0	–	–	–	–
rat	575	287	20642	7943.7	61.52	18000.0	–	–	–	–

was able to solve just 2. On the other hand, our parallel B&C outperforms CPLEX in producing upper bounds in 18 instances over 28. For the others 10 instances where CPLEX produced better upper bounds, the solutions produced by our algorithm were close (at most 15.44% higher) to those obtained by CPLEX relatively to the other way around (CPLEX upper bounds which are 88.47% higher than the ones produced by our algorithm). From the gap point of view, both our parallel B&C and CPLEX outperformed equally each other in producing lower gaps. In 13 cases CPLEX (resp. parallel B&C applied to the natural formulation) over 28 produced better gaps than

the ones produced by parallel B&C (resp. CPLEX). Moroever, we can easily verify that the distrubution of the best gaps was uniform, for the smallest graphs CPLEX was better and from the instance $kroA\_150\_50$, parallel B&C performed better. Finally, we notice that parallel B&C was able to produced both a lower and upper bound for the 11 largest instances that CPLEX with the flow formulation was not able to charge to memory.

5.4.2 Parallel study

As said by Cung, Le Cun and Roucairol in [75] when talking about parallel computing in combinatorial optimization: *"Classical criteria such as acceleration (ratio of the time of the best sequential algorithm to the time of the parallel algorithm) or effectiveness (the acceleration on the number of processors used) are not well suited as performance measures in the metaheuristics domain. Explorations carried out sequentially and in parallel can indeed be different. Furthermore, solutions of different quality or even structure can be found. This is the reason why authors have proposed comparing the quality of solutions with fixed execution times, or comparing execution times with fixed solution quality."*, we have noticed the same problems during this experimental study of our parallel combinatorial optimization algorithms. In fact, we were not able to do a common parallelization study due to the facts that we could not compare our intrinsic parallel implementation of the algorithms to the sequential versions, that for the hardest input graphs we could not let our algorithms compute beyond our 5 hours limit and that our main validation criteria is the kHNDP solving.

For these reasons, to show the impact of the parallelization on our algorithms computation time, we first start by introducing new metrics that will be used to simplify/clarify the study that follows:

5.4.2.1 Pseudo metrics

Consider a parallel algorithm P_A to implement on a multiprocessor machine M made up of p identical processors. Let T_p be the execution time of P_A on the p processors and T_{min} the execution time on the minimum number of processors. We then define:

- the R_Speedup $RS_p = \frac{T_{min}}{T_p}$
- the R_Efficiency $RE_p = \frac{RS_p}{p} = \frac{T_{min}}{pT_p}$

The main difference between these metrics and the common ones (speedup and efficiency) is that the common ones underlign the difference of using the parallelism with p processors and the sequentiel version, while the new ones underlign the use of $p2$ processors instead of $p1$.

It is easy to see that the ideal R_acceleration is equal to $p2/p1$. If we consider this function, the parallel algorithm is all the more efficient as the curve of variations is close to the line of equation $y = p2/p1$. In this case we will speak of linear acceleration. To have good accelerations, the extra cost of parallelization in computation time, and especially in communication volume, must be minimum. It is also necessary to apply to minimize the inactivity times of the processors due to an imbalance in the distribution of the charges between the processors. In some cases, the latter may be inevitable, in particular because of the inter-task precedences or the heterogeneity of the processors.

The R_efficiency of a parallel algorithm is generally less than 1 (or 100% if defined as a percentage). The closer the efficiency is to 1, the better the algorithm has good parallel qualities. The most important factor in decreasing efficiency is the cost of communication.

5.4.2.2 Impact of parallelism on the B&B

In order to measure the impact of the parallelization in the parallel B&B, we have considered the CPU time for our algorithm used for the kHNDP with $k = 3$ and $L = 3$. We have used $p = 2, 8, 16, 32, 64, 128$ processors. The choice of these numbers is due to the fact that each node of the cluster contains 16 cores and running the algorithms with these numbers can help algorithm to take full advantage from the architecture.

Notice that our minimum number of processors here is 2. In fact for $p = 2$ the behavior of our algorithm is very similar to a sequential version due to the fact that the two processes are necessary to the main algorithm to end, and in the same time these two processes does not do any simultaneous computation. Figure 5.9 shows an example of a run.

Tables 5.5, 5.6 and 5.7 below exhibit respectively the CPU time, the speedup and the efficiency obtained for a subset of 7 test instances. The CPU computation time presented corresponds to the mean of 5 runs CPU times.

Figures 5.10, 5.11 and 5.12 shows respectively the CPU time, the R_Speedup and the R_Efficiency variation in function of p and the input graphs.

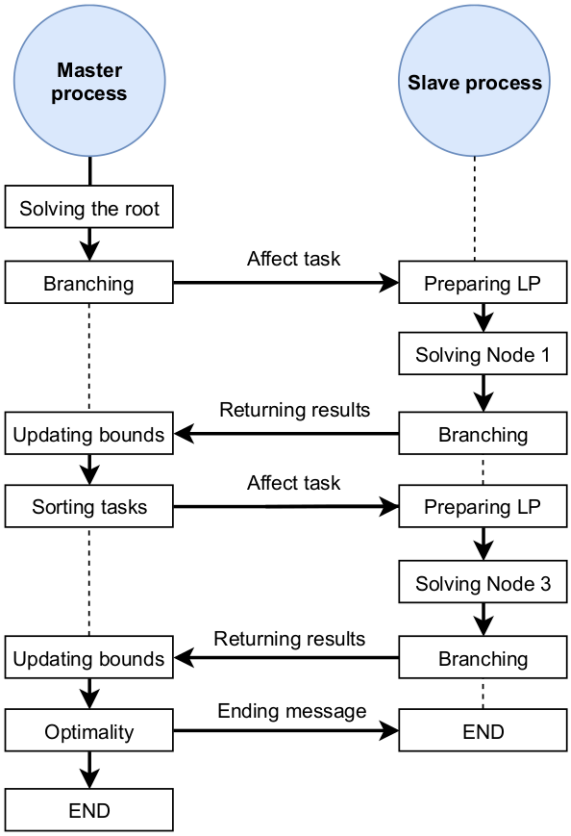

Figure 5.9: Parallel B&B run example when $p = 2$

Table 5.5: CPU computation time for Parallel B&B with p variation

name	$\|V\|$	$\|D\|$	CPU time					
			$p = 2$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
st	70	5	9.06	2.43	5.38	5.07	5.78	5.15
berlin	52	5	28.72	6.40	4.93	4.13	4.04	3.79
berlin	52	7	34.42	4.82	4.79	3.92	4.53	4.47
berlin	30	7	45.32	5.40	3.38	2.12	1.43	1.61
st	70	7	473.75	61.01	53.47	44.09	28.63	19.85
berlin	52	10	4123.36	508.29	438.68	334.32	226.72	142.71
berlin	30	10	9530.88	1021.92	897.29	631.11	265.46	128.94

Table 5.5 and Figure 5.10 clearly show that the CPU time is decreasing when $p = 2, 8, 16, 32, 64$ and 128 for the graphs $berlin52\_5$, $st70\_7$, $berlin52\_10$ and $berlin30\_10$. But for instances $st70\_5$, $berlin52\_7$ and $berlin30\_7$ the CPU time decreases until reaching a limit (resp. $p = 8$, $p = 32$ and $p = 64$) and increases after that.

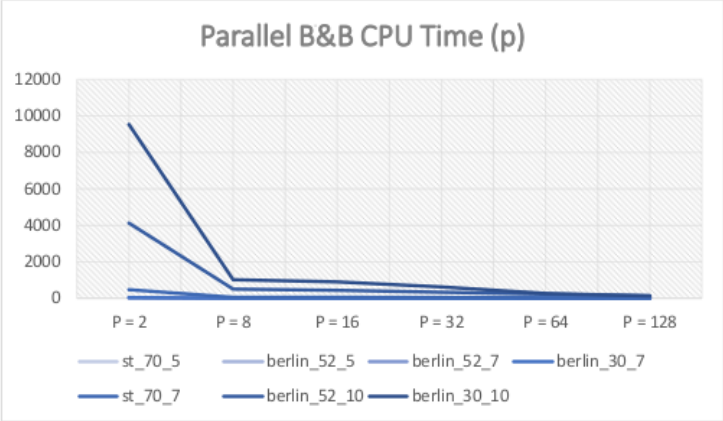

Figure 5.10: Parallel B&B CPU Time

Table 5.6: Parallel B&B R_Speedup results

name	$\|V\|$	$\|D\|$	R_Speedup				
			$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
st	70	5	3.74	1.68	1.79	1.57	1.76
berlin	52	5	4.49	5.83	6.96	7.12	7.58
berlin	52	7	7.14	7.19	8.79	7.60	7.70
berlin	30	7	8.39	13.40	21.35	31.62	28.19
st	70	7	7.76	8.86	10.75	16.55	23.86
berlin	52	10	8.11	9.40	12.33	18.19	28.89
berlin	30	10	9.33	10.62	15.10	35.90	73.92

Same remarks for the R_Speedup in Table 5.6 and Figure 5.11. We can easily see that the R_Speedup is strictly increasing for the graphs $berlin52\_5, st70\_7, berlin52\_10$ and $berlin30\_10$. But increases for instances $st70\_5, berlin52\_7$ and $berlin30\_7$ after reaching the limits (resp. $p = 8$, $p = 32$ and $p = 64$). This can be due to the fact that reaching a certain point of improvement for some graphs, the communication cost (in a time point of view) becomes more important than the parallel computation improvement which increases the total CPU time. What is good overall, is that this time deterioration is decreasing for an increasing input graph size ($\simeq 3.5s$ for $st70\_5$, $\simeq 1s$ for $berlin52\_7$ and $\simeq 0.2s$ for $berlin30\_7$).

What is interesting to see also is that the R_Speedup increases for the same value of p but for instances that are more time consuming. Except for the instance $berlin30\_7$ which produces higher unexpected superlinear R_Speedups. This behavior is common in the use of parallel computing in combinatorial optimization. It can be due to the fact that, for this instance, the structure of the B&B tree is scattered in a way that

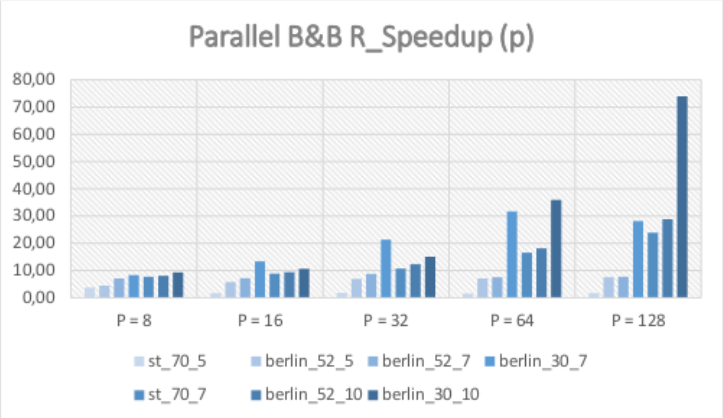

Figure 5.11: Parallel B&B R_Speedup

is explored more efficiently when the parallelism is used. In other words, evaluating the tree nodes in parallel increases much more our chance to explore the right tree branches and thus produce better bounds earlier.

Table 5.7: Parallel B&B R_Efficiency results

name	$\|V\|$	$\|D\|$	R_Efficiency				
			$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
st	70	5	0.47	0.11	0.06	0.02	0.01
berlin	52	5	0.56	0.36	0.22	0.11	0.06
berlin	52	7	0.89	0.45	0.27	0.12	0.06
berlin	30	7	1.05	0.84	0.67	0.49	0.22
st	70	7	0.97	0.55	0.34	0.26	0.19
berlin	52	10	1.01	0.59	0.39	0.28	0.23
berlin	30	10	1.17	0.66	0.47	0.56	0.58

In Table 5.7 and Figure 5.12, the R_Efficiency is strictly decreasing with p except for *berlin*30_10 where it increases for $p = 64, 128$. We can notice that in three situations we had $R\_E > 1$ which can be due to the B&B tree structure. What is interesting to see also is the increasing behavior of the R_Efficiency for the same number of processors but increasing problem size (in terms of needed computation time). This later remark shows the importance of using the parallel computing techniques to solve large size problems.

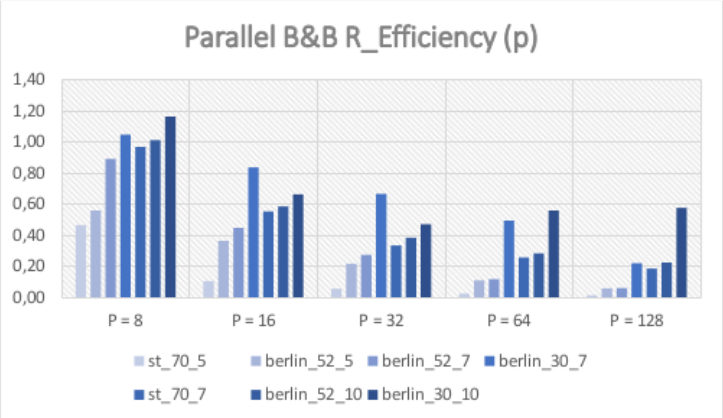

Figure 5.12: Parallel B&B R_Efficiency

5.4.2.3 Impact of parallelism on the B&C

As for the parallel B&B, to study the impact of parallelism on the B&C we consider the CPU time for our algorithm applied to the flow formulation used for the kHNDP with $k = 3$ and $L = 3$. We have used $p = 3, 8, 16, 32, 64, 128$ processors.

For this case, when $p = 3$ the behavior of our algorithm is very similar to a sequential version due to the fact that the three processes (master, slave and pool manager process) does not do any simultaneous computation. Figure 5.13 shows an example of a run.

Tables 5.8, 5.9 and 5.10 below exhibit respectively the CPU time, the speedup and the efficiency obtained for a subset of 7 test instances. The CPU computation time presented corresponds to the mean of 5 runs CPU times.

Figures 5.14, 5.15 and 5.16 shows respectively the CPU time, the R_Speedup and the R_Efficiency variation in function of p and the input graphs.

Table 5.8 and Figure 5.14 show that the CPU time is decreasing when $p = 3, 8, 16, 32, 64$ and 128 for all graphs except for $st70\_5$. In fact for this instance, which is the easiest to solve for the kHNDP when $k = 3$ and $L = 3$, the CPU time decreases until reaching a limit (when $p = 8$) and increases after that.

The major point to raise, from these results, is that for instances $berlin30\_10$, $berlin52\_10$ and $st70\_7$, which are the hardest to solve, the algorithm was not able to solve them to optimality unless we increase our parallelism. In fact, for $berlin30\_10$ and $p = 3, 8$ (resp. $berlin52\_10$, $p = 3, 8$ and $st70\_7$, $p = 3, 8, 16, 32$) the CPU time

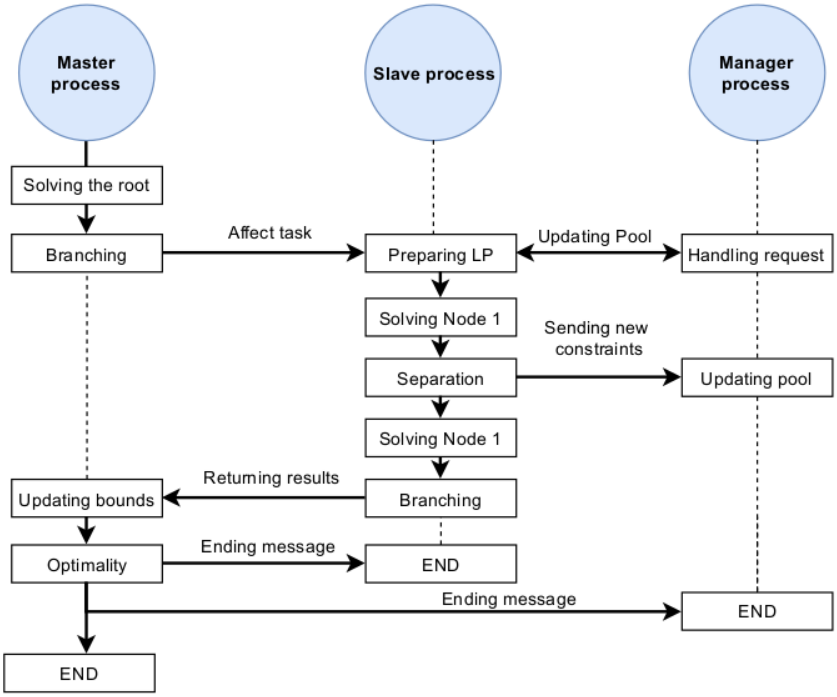

Figure 5.13: Parallel B&C run example when $p = 3$

Table 5.8: CPU computation time for Parallel B&C with p variation

name	$\|V\|$	$\|D\|$	CPU time					
			$p = 3$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
st	70	5	22.34	5.09	6.04	7.08	6.85	7.10
berlin	52	7	147.89	11.82	10.74	8.18	5.84	5.13
berlin	30	7	335.52	52.99	35.71	23.18	12.05	5.94
berlin	52	5	695.20	107.05	73.60	41.34	21.87	9.21
berlin	30	10	18000.00	18000.00	2416.19	737.75	376.19	203.82
berlin	52	10	18000.00	18000.00	16219.19	7613.17	3929.98	2074.82
st	70	7	18000.00	18000.00	18000.00	18000.00	13405.19	8055.64

reached the limit of 5 hours of calculation, and the algorithm was stopped, while for a greater number of processors the algorithms solved the instance to optimality. This result is very satisfying since our first validation criteria is the kHNDP solving.

In Figure 5.14, we splitted the results into two graphics to show that for the first set of the tables the CPU Time was strictly decreasing, while in the second the time starts to increase just when we use enough processors to allow the algorithm to solve the instance to optimality.

Figure 5.14: Parallel B&C CPU Time

Notice that since the algorithms were stopped after 5 hours of calculation for some values of p and specially for $p = 3$ on which the values of R_Speedup and R_Efficiency are based, we will not base our study on them for these instances. We just keep them in the tables in case the reader would be interested to see them but we think that they are not precise enough to let us study the algorithm's behavior. In consequence, our R_Speedup and R_Efficiency study will be based on the results for the 4 firsts instances that are *st*70_5, *berlin*52_7, *berlin*30_7, *berlin*52_5.

Table 5.9: Parallel B&C R_speedup results

name	$\|V\|$	$\|D\|$	R_Speedup				
			$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
st	70	5	4.38	3.70	3.15	3.26	3.15
berlin	52	7	12.51	13.77	18.08	25.34	28.81
berlin	30	7	6.33	9.39	14.47	27.84	56.50
berlin	52	5	6.49	9.45	16.82	31.79	75.49
berlin	30	10	1.00	7.45	24.40	47.85	88.31
berlin	52	10	1.00	1.11	2.36	4.58	8.68
st	70	7	1.00	1.00	1.00	1.34	2.23

For the R_Speedup in Table 5.9 and Figure 5.15 we notice the same remarks that was seen for the CPU time. We can easily see that the R_Speedup is strictly increasing for all the graphs of the first set with p. But deccreases for instances *st*70_5 for $p > 8$. What is interesting to see also is that R_Speedup increases for the same value of p but for instances that are more time consuming. Except for the instance *berlin*52_7 which produces unexpected (and even superlinear for $p = 8$) higher R_Speedups. This behavior could be due to the same reason that we presented in Section 5.4.2.2. In fact, for this instance, the structure of the B&C tree is probably scattered in a way that the algorithm explores it more efficiently when the parallelism is used. In other words,

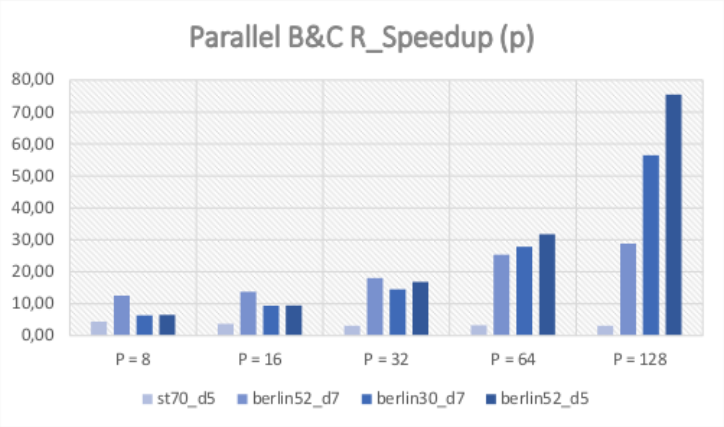

Figure 5.15: Parallel B&C R_Speedup

evaluating the tree nodes in parallel increases much more our chance to explore the right tree branches and then produce better bounds earlier.

Table 5.10: Parallel B&C R_Efficiency results

name	$\|V\|$	$\|D\|$	R_Efficiency				
			$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
st	70	5	0.55	0.23	0.10	0.05	0.02
berlin	52	7	1.56	0.86	0.57	0.40	0.23
berlin	30	7	0.79	0.59	0.45	0.44	0.44
berlin	52	5	0.81	0.59	0.53	0.50	0.59
berlin	30	10	0.13	0.47	0.76	0.75	0.69
berlin	52	10	0.13	0.07	0.07	0.07	0.07
st	70	7	0.13	0.06	0.03	0.02	0.02

In Table 5.7 and Figure 5.16, the R_Efficiency is strictly decreasing except for *berlin*52_5 where it increases for $p = 128$. We can notice that in one situation we had $R\_E > 1$ which is related to the superlinear R_Speedup. It is interesting to see also that R_E is strictly increasing when we fix the number of processors but increase the problem size (in terms of needed computation time). This later remark shows the importance of using the parallel computing techniques to solve large size problems.

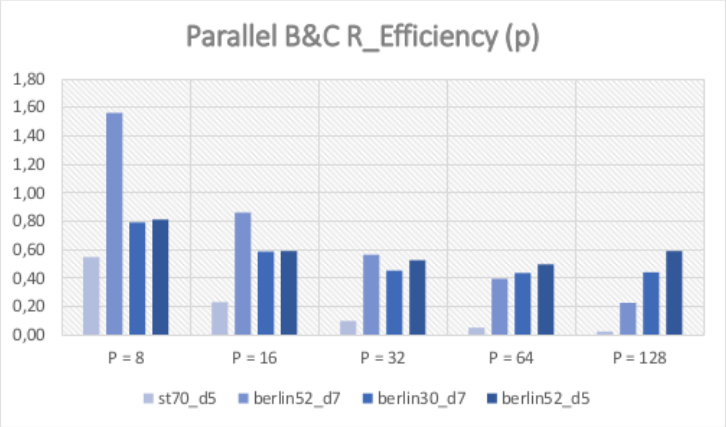

Figure 5.16: Parallel B&C R_Efficiency

5.5 Conclusion

We studied in this chapter the k-edge connected Hop-constrained Network Design Problem (kHNDP) aiming optimal resolution. We presented two distributed exact algorithms: a distributed Branch-and-Bound and a distributed Branch-and-Cut. We presented after that, an experimental study in which we compare our algorithms to CPLEX that shows that even if CPLEX is still the best solving method for small graphs, our parallel Branch-and-Cut applied to the flow formulation is the best for mid-range size graphs and our parallel Branch-and-Cut applied to the natural formulation is the method to use for large ones.

Conclusion

In this dissertation, we have studied the parallelization of hybrid metaheuristics for network design problems. In a first step, we have proposed two hybrid parallel algorithms for solving the Steiner k-Connected Network Design Problem (SkESNDP), and the k-Edge-Connected Hop-Constrained Survivable Network Design Problem (kHNDP) when $L = 2, 3$. We have extended this work after that for the k-Edge-Connected Survivable Network Design (kESNDP) and the k-Edge-Connected Hop-Constrained Survivable Network Design Problem (kHNDP) when $L \geq 2$. The algorithms are based on a Lagrangian relaxation algorithm, a genetic algorithm and a greedy algorithm, and aims in producing both lower and upper bounds. The experiments conducted have shown that our hybrid algorithm outperforms CPLEX in producing good feasible solutions, even for large size instances, and this within a relatively short CPU time. They have also shown that the hybridization of the three components outperforms, in most cases, each component taken separately. Finally, we have shown that using several processors inside each component of the hybrid algorithm helps in decreasing the CPU time for each component.

The parallel computing framework we have proposed is generic and can be applied to all the network design problems, and not only, which has the same block structure as the problems studied. Also, the implementation we have proposed is quite simple and can be easily adapted to other problems. For example, in the parallel hybrid algorithm, one can replace the genetic algorithm by any population-based metaheuritic, and obtain an identical framework.

It should be noticed that our algorithm PHA is heuristic, and contrarily to many heuristics, is able to produce both upper and lower bounds of the optimal solution. This can give an indication on the quality of the feasible solution obtained, in particular when the gap between the lower and upper bounds is small. Indeed, a very small gap (0% in the better case) implies that the solution is close to the optimal solution. We

can observe that in all our experiments, that the gaps between the lower and upper bounds are quite large (more than 40% for most of the instances). Thus, it would be interesting to further investigate a way of producing better lower bounds in order to know how close the solutions produced by algorithm PHA are from the optimal solution.

In the last Chapter we have presented new distributed implementations of the well known branch-and-bound and branch-and-cut algorithms. For that we started by introducing a state of the art about the parallelization studies of the Branch-and-Bound and the Branch-and-Cut methods. We have also presented two parallel algorithms based on the Branch-and-Bound and the Branch-and-Cut. Finally we have validate our algorithms by showing the experimental study we have conducted on a cluster of 128 processors for solving the kHNDP when $k = 3$ and $L = 3$. These experimentations showed an interesting speedup obtained by running our algorithms on the 128 cores, and through this parallelisation we was able to solve large scale instances for the kHNDP and produce better gaps for instances with $\simeq 500$ nodes and $\simeq 300$ demands.

What we could observe that in all our experiments, that the gaps between the lower and upper bounds for large scale instances were quite large (more than 40% for most of the instances). Thus, it would be interesting to further investigate a way of producing better lower bounds in order to know how close the solutions produced by algorithms are to the optimal solution.

Finally it would also be interesting to further investigate the parallelization of algorithms and the use of more sophisticated architectures, like GPUs.

Bibliography

[1] http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[2] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[3] http://www.boost.org/doc/libs/1_60_0/libs/graph/.

[4] http://www.info.univ-angers.fr/pub/porumbel/graphs/.

[5] https://gephi.org/.

[6] https://lemon.cs.elte.hu/trac/lemon.

[7] https://www.yworks.com/products/yed.

[8] COIN-OR library for efficient modeling and optimization in networks lemon. http://lemon.cs.elte.hu/trac/lemon. Accessed: 2015-12-13.

[9] D. Abramson. A very high speed architecture for simulated annealing. *Computer*, 25(5):27–36, 1992.

[10] D. Abramson, P. Logothetis, A. Postula, and M. Randall. Application specific computers for combinatorial optimisation. In *Australian Computer Architecture Workshop*. Citeseer, 1997.

[11] R. K. Ahuja. *Network flows*. PhD thesis, TECHNISCHE HOCHSCHULE DARMSTADT, 1993.

[12] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[13] E. Alba. *Parallel metaheuristics: a new class of algorithms*, volume 47. John Wiley & Sons, 2005.

[14] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5):443–462, 2002.

[15] C. Alves and J. V. de Carvalho. A stabilized branch-and-price-and-cut algorithm for the multiple length cutting stock problem. *Computers & Operations Research*, 35(4):1315 – 1328, 2008.

[16] K. Andreev and H. Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 120–124. ACM, 2004.

[17] K. Anstreicher, N. Brixius, J.-P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3):563–588, 2002.

[18] K. M. Anstreicher and N. W. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. *Mathematical Programming*, 89(3):341–357, 2001.

[19] K. M. Anstreicher and N. W. Brixius. Solving quadratic assignment problems using convex quadratic programming relaxations. *Optimization Methods and Software*, 16(1-4):49–68, 2001.

[20] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Implementing the dantzig-fulkerson-johnson algorithm for large traveling salesman problems, 2003.

[21] D. Applegate, R. Bixby, W. Cook, and V. Chvátal. On the solution of traveling salesman problems. 1998.

[22] S. Arora, D. Karger, and M. Karpinski. Polynomial time approximation schemes for dense instances of np-hard problems. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 284–293. ACM, 1995.

[23] A. Arulselvan, C. W. Commander, L. Elefteriadou, and P. M. Pardalos. Detecting critical nodes in sparse graphs. *Computers & Operations Research*, 36(7):2193–2200, 2009.

[24] T. Bäck. Evolutionary algorithms in theory and practice. 1996.

[25] T. Bäck, D. B. Fogel, and Z. Michalewicz. Handbook of evolutionary computation. *New York: Oxford*, 1997.

[26] M. Baïou, F. Barahona, and A. R. Mahjoub. Separation of partition inequalities. *Mathematics of Operations Research*, 25(2):243–254, 2000.

[27] E. Balas and C. C. de Souza. The vertex separator problem: a polyhedral investigation. *Mathematical Programming*, 103(3):583–608, 2005.

[28] F. Barahona and A. R. Mahjoub. Facets of the balanced (acyclic) induced subgraph polytope. *Mathematical Programming*, 45(1-3):21–33, 1989.

[29] F. Barahona and A. R. Mahjoub. On two-connected subgraph polytopes. *Discrete Mathematics*, 147(1):19–34, 1995.

[30] B. Barney. What is parallel computing. *Introduction to Parallel Computing*, 2012.

[31] C. Barnhart, A. M. Cohn, E. L. Johnson, D. Klabjan, G. L. Nemhauser, and P. H. Vance. Airline crew scheduling. In *Handbook of Transportation Science*, volume 56 of *International Series in Operations Research & Management Science*, pages 517–560. 2003.

[32] C. Barnhart, C. A. Hane, and P. H. Vance. Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems. *Oper. Res.*, 48(2):318–326, 2000.

[33] M. Bateni, M. Hajiaghayi, P. N. Klein, and C. Mathieu. A polynomial-time approximation scheme for planar multiway cut. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 639–655. Society for Industrial and Applied Mathematics, 2012.

[34] J. E. Beasley. Modern heuristic techniques for combinatorial problems. chapter Lagrangian Relaxation, pages 243–303. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[35] W. Ben-Ameur and M. D. Biha. Algorithms and formulations for the minimum cut separator problem. *Electronic Notes in Discrete Mathematics*, 36:977–983, 2010.

[36] W. Ben-Ameur and M. Didi Biha. On the minimum cut separator problem. *Networks*, 59(1):30–36, 2012.

[37] W. Ben-Ameur, M.-A. Mohamed-Sidi, and J. Neto. The k-separator problem. In *Computing and Combinatorics*, pages 337–348. Springer, 2013.

[38] W. Ben-Ameur, M.-A. Mohamed-Sidi, and J. Neto. The k-separator problem: polyhedra, complexity and approximation results. *Journal of Combinatorial Optimization*, 29(1):276–307, 2015.

[39] F. Bendali, I. Diarrassouba, M. Didi Biha, A. R. Mahjoub, and J. Mailfert. A branch-and-cut algorithm for the k-edge connected subgraph problem. *Networks*, 55(1):13–32, 2010.

[40] A. Berger, A. Grigoriev, and R. Zwaan. Complexity and approximability of the k-way vertex cut. *Networks*, 63(2):170–178, 2014.

[41] M. D. Biha. Personal communication. page 2014.

[42] M. D. Biha. On the 3-terminal cut polyhedron. *SIAM Journal on Discrete Mathematics*, 19(3):575–587, 2005.

[43] M. D. Biha and A. R. Mahjoub. k-edge connected polyhedra on series-parallel graphs. *Operations research letters*, 19(2):71–78, 1996.

[44] M. D. Biha and M.-J. Meurs. An exact algorithm for solving the vertex separator problem. *Journal of Global Optimization*, 49(3):425–434, 2011.

[45] H. T. T. Binh, S. H. Ngo, and D. N. Nguyen. *Genetic Algorithm for Solving Survivable Network Design with Simultaneous Unicast and Anycast Flows*, pages 1237–1247. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[46] C. Blum and A. Roli. Hybrid metaheuristics: an introduction. In *Hybrid Metaheuristics*, pages 1–30. Springer, 2008.

[47] C. Blumi, A. Roli, and E. Alba. 1 an introduction to metaheuristic techniques. *Parallel Metaheuristics: A New Class of Algorithms*, 47:1, 2005.

[48] V. Boginski and C. W. Commander. Identifying critical nodes in protein-protein interaction networks. *Clustering challenges in biological networks*, pages 153–167, 2009.

[49] Q. Botton, B. Fortz, L. Gouveia, and M. Poss. Benders decomposition for the hop-constrained survivable network design problem. *INFORMS journal on computing*, 25(1):13–26, 2013.

[50] M. Bouchakour. *Composition de graphes et le polytope des absorbants. Un algorithme de coupes pour le problème du flot à coûts fixes*. PhD thesis, Université de Rennes 1, 1996.

[51] M. E. Bouzgarrou. *Parallélisation de la méthode du" Branch and Cut" pour résoudre le problème du voyageur de commerce*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 1998.

[52] L. T. Bui and H. Thi Thanh Binh. A survivable design of last mile communication networks using multi-objective genetic algorithms. *Memetic Computing*, 8(2):97–108, 2016.

[53] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[54] G. Călinescu, H. Karloff, and Y. Rabani. An improved approximation algorithm for multiway cut. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 48–52. ACM, 1998.

[55] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[56] S. A. Canuto, M. G. Resende, and C. C. Ribeiro. Local search with perturbations for the prize-collecting steiner tree problem in graphs. *Networks*, 38(1):50–58, 2001.

[57] V. Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[58] J. Chen, Y. Liu, and S. Lu. An improved parameterized algorithm for the minimum node multiway cut problem. *Algorithmica*, 55(1):1–13, 2009.

[59] X. Cheng and D.-Z. Du. *Steiner trees in industry*, volume 11. Springer Science & Business Media, 2013.

[60] R. Chitnis, M. Cygan, M. Hajiaghayi, M. Pilipczuk, and M. Pilipczuk. Designing fpt algorithms for cut problems using randomized contractions. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 460–469. IEEE, 2012.

[61] S. Chopra and J. H. Owen. Extended formulations for the a-cut problem. *Mathematical programming*, 73(1):7–30, 1996.

[62] S. Chopra and M. R. Rao. On the multiway cut polyhedron. *Networks*, 21(1):51–89, 1991.

[63] W. Chou and H. Frank. Survivable communication networks and the terminal capacity matrix. *IEEE Transactions on Circuit Theory*, 17(2):192–197, 1970.

[64] T. Christof and G. Reinelt. Combinatorial optimization and small polytopes. *Top*, 4(1):1–53, 1996.

[65] N. Christofides and C. Whitlock. Network synthesis with connectivity constraints: a survey. *Operational Research*, 81:705–723, 1981.

[66] I. CNRS. GRID5000. https://www.grid5000.fr/, 2008.

[67] COIN-OR. Computational Optimization Infrastructure for Operations Research. https://github.com/coin-or, 2000.

[68] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[69] D. Cornaz, F. Furini, M. Lacroix, E. Malaguti, A. R. Mahjoub, and S. Martin. Mathematical formulations for the balanced vertex k-separator problem. In *Control, Decision and Information Technologies (CoDIT), 2014 International Conference on*, pages 176–181. IEEE, 2014.

[70] D. Cornaz, Y. Magnouche, A. R. Mahjoub, and S. Martin. The multi-terminal vertex separator problem: Polyhedral analysis and branch-and-cut. In *International Conference on Computers & Industrial Engineering (CIE45)*, 2015.

[71] C. Cotta, E. Talbi, and E. Alba. Parallel hybrid metaheuristics. In *PARALLEL METAHEURISTICS, A NEW CLASS OF ALGORITHMS*. Citeseer, 2005.

[72] C. Cotta-Porras. A study of hybridisation techniques and their application to the design of evolutionary algorithms. *AI Communications*, 11(3, 4):223–224, 1998.

[73] M. Creel and W. L. Goffe. Multi-core cpus, clusters, and grid computing: A tutorial. *Computational economics*, 32(4):353–382, 2008.

[74] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[75] V. Cung, B. L. Cun, and C. Roucairol. *Parallel Combinatorial Optimization*, chapter 8, pages 225–251. Wiley-Blackwell, 2014.

[76] V. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Parallel and distributed branch-and-bound/a* algorithms. *Techncial report, Laboratoire PRISM, Universite de Versailles*, page 94, 1994.

[77] W. H. Cunningham. A network simplex method. *Mathematical Programming*, 11(1):105–116, 1976.

[78] W. H. Cunningham and L. Tang. Optimal 3-terminal cuts and linear programming. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 114–125. Springer, 1999.

[79] M. Cygan, M. Pilipczuk, M. Pilipczuk, and J. O. Wojtaszczyk. On multiway cut parameterized above lower bounds. In *Parameterized and Exact Computation*, pages 1–12. Springer, 2011.

[80] G. Dahl. Notes on polyhedra associated with hop-constrained paths. *Operations research letters*, 25(2):97–100, 1999.

[81] G. Dahl and L. Gouveia. On the directed hop-constrained shortest path problem. *Operations Research Letters*, 32(1):15–22, 2004.

[82] G. Dahl, D. Huygens, A. R. Mahjoub, and P. Pesneau. On the k edge-disjoint 2-hop-constrained paths polytope. *Operations research letters*, 34(5):577–582, 2006.

[83] G. Dahl and B. Johannessen. The 2-path network problem. *Networks*, 43(3):190–199, 2004.

[84] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994.

[85] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):pp. 101–111, 1960.

[86] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations research*, 8(1):101–111, 1960.

[87] M. S. Darwish, T. Saad, and Z. Hamdan. A high scalability parallel algebraic multigrid solver. In *ECCOMAS CFD 2006: Proceedings of the European Conference on Computational Fluid Dynamics, Egmond aan Zee, The Netherlands, September 5-8, 2006*. Delft University of Technology; European Community on Computational Methods in Applied Sciences (ECCOMAS), 2006.

[88] C. de Souza and E. Balas. The vertex separator problem: algorithms and computations. *Mathematical Programming*, 103(3):609–631, 2005.

[89] F. Dehne, A. G. Ferreira, and A. Rau-Chaplin. Parallel branch and bound on fine-grained hypercube multiprocessors. In *Tools for Artificial Intelligence, 1989. Architectures, Languages and Algorithms, IEEE International Workshop on*, pages 616–622. IEEE, 1989.

[90] F. Dehne, A. G. Ferreira, and A. Rau-Chaplin. Parallel branch and bound on fine-grained hypercube multiprocessors. *Parallel computing*, 15(1-3):201–209, 1990.

[91] E. D. Demaine, M. Hajiaghayi, and P. N. Klein. Node-weighted steiner tree and group steiner tree in planar graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 328–340. Springer, 2009.

[92] J. Denzinger and T. Offermann. On cooperation between evolutionary algorithms and other search paradigms. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3. IEEE, 1999.

[93] G. Desaulniers, J. Desrosiers, and M. M. Solomon. Column generation. volume 5. Springer-Verlag New York Incorporated, 2005.

[94] M. Di Summa, A. Grosso, and M. Locatelli. Branch and cut algorithms for detecting critical nodes in undirected graphs. *Computational Optimization and Applications*, 53(3):649–680, 2012.

[95] I. Diarrassouba. *Survivale Network Design Problems with High Connectivity Requirement*. PhD thesis, Université Blaise Pascal-Clermont-Ferrand II, 2009.

[96] I. Diarrassouba, V. Gabrel, L. Gouveia, A. Mahjoub, and P. Pesneau. Integer programming formulations for the k-edge-connected 3-hop-constrained network design problem. In *International Network Optimization Conference*, 2013.

[97] I. Diarrassouba, V. Gabrel, L. Gouveia, A. R. Mahjoub, and P. Pesneau. Integer programming formulations for the k-edge-connected 3-hop-constrained network design problem. *Networks*, 67(2):148–169, 2016.

[98] I. Diarrassouba, M. K. Labidi, and A. R. Mahjoub. A parallel hybrid optimization algorithm for some network design problems. *Soft Computing*, pages 1–18, 2017.

[99] I. Diarrassouba, M. K. Labidi, and A. R. Mahjoub. A hybrid optimization approach for the steiner k-connected network design problem. *Electronic Notes in Discrete Mathematics*, 64:305–314, 2018.

[100] I. Diarrassouba, A. R. Mahjoub, and H. Kutucu. Two node-disjoint hop-constrained survivable network design and polyhedra. *Networks*, 67(4):316–337, 2016.

[101] I. Diarrassouba, A. R. Mahjoub, and H. Yaman. Integer programming formulations for k-node-connected hop-constrained survivable network design problem. *Cahier du LAMSADE*, 382, 2017.

[102] T. N. Dinh, Y. Xuan, M. T. Thai, E. Park, and T. Znati. On approximation of new optimization methods for assessing network vulnerability. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[103] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[104] R. G. Downey, V. Estivill-Castro, M. R. Fellows, E. Prieto, F. A. Rosamond, et al. Cutting up is hard to do: the parameterized complexity of k-cut and related probelms| nova. the university of newcastle's digital repository. 2003.

[105] Z. Drezner. Extensive experiments with hybrid genetic algorithms for the solution of the quadratic assignment problem. *Computers & Operations Research*, 35(3):717–736, 2008.

[106] D.-Z. Du, J. Smith, and J. H. Rubinstein. *Advances in Steiner trees*, volume 6. Springer Science & Business Media, 2013.

[107] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[108] J. Edmonds. Covers and packings in a family of sets. *Bulletin of the American Mathematical Society*, 68(5):494–499, 1962.

[109] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards (B) 69*, 69:9–14, 1965.

[110] M. El-Abd and M. Kamel. A taxonomy of cooperative search algorithms. In *Hybrid Metaheuristics*, pages 32–41. Springer, 2005.

[111] N. Fan and P. M. Pardalos. Robust optimization of graph partitioning and critical node detection in analyzing networks. In *International Conference on Combinatorial Optimization and Applications*, pages 170–183. Springer, 2010.

[112] E. A. Feigenbaum, J. Feldman, et al. *Computers and thought*. New York, 1963.

[113] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, pages 241–247. ACM, 1988.

[114] M. L. Fisher, R. Jaikumar, and L. N. Van Wassenhove. A multiplier adjustment method for the generalized assignment problem. *Management Science*, 32(9):1095–1103, 1986.

[115] C. Fleurent and J. A. Ferland. Genetic hybrids for the quadratic assignment problem. *Quadratic assignment and related problems*, 16:173–187, 1994.

[116] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[117] L. Fogel. Toward inductive inference automata. In *Communications of the ACM*, volume 5, pages 319–319. ASSOC COMPUTING MACHINERY 1515 BROADWAY, NEW YORK, NY 10036, 1962.

[118] L. J. Fogel, A. J. Owens, and M. J. Walsh. Artificial intelligence through simulated evolution. 1966.

[119] J. Fonlupt and A. R. Mahjoub. Critical extreme points of the 2-edge connected spannning subgraph polytope. In *Integer Programming and Combinatorial Optimization*, pages 166–182. Springer, 1999.

[120] A. Frank. Augmenting graphs to meet edge-connectivity requirements. *SIAM Journal on Discrete Mathematics*, 5(1):25–53, 1992.

[121] J. Fukuyama. Np-completeness of the planar separator problems. *J. Graph Algorithms Appl.*, 10(2):317–328, 2006.

[122] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, 1979.

[123] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[124] N. Garg, V. V. Vazirani, and M. Yannakakis. Multiway cuts in directed and node weighted graphs. In *Automata, Languages and Programming*, pages 487–498. Springer, 1994.

[125] N. Garg, V. V. Vazirani, and M. Yannakakis. Multiway cuts in node weighted graphs. *Journal of Algorithms*, 50(1):49–61, 2004.

[126] M. Gen and R. Cheng. *Genetic algorithms and engineering optimization*, volume 7. John Wiley & Sons, 2000.

[127] B. Gendron and T. G. Crainic. Parallel branch-and-branch algorithms: Survey and synthesis. *Operations research*, 42(6):1042–1066, 1994.

[128] M. Gengler, S. Ubéda, and F. Desprez. *Initiation au parallélisme: concepts, architectures et algorithmes*. Masson, 1996.

[129] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[130] P. C. Gilmore and R. E. Gomory. A Linear Programming Approach to the Cutting Stock Problem-Part II. *Operations Research*, 11(6):863–888, 1963.

[131] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[132] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[133] M. X. Goemans and D. J. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. *Mathematical Programming*, 60(1-3):145–166, 1993.

[134] M. X. Goemans and Y.-S. Myung. A catalog of steiner tree formulations. *Networks*, 23(1):19–28, 1993.

[135] D. E. Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989, 1989.

[136] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[137] D. E. Goldberg et al. *Genetic algorithms in search optimization and machine learning*, volume 412. Addison-wesley Reading Menlo Park, 1989.

[138] O. Goldschmidt and D. S. Hochbaum. A polynomial algorithm for the k-cut problem for fixed k. *Mathematics of operations research*, 19(1):24–37, 1994.

[139] L. Gouveia and C. Requejo. A new lagrangean relaxation approach for the hop-constrained minimum spanning tree problem. *European Journal of Operational Research*, 132(3):539–552, 2001.

[140] M. Grötschel and C. L. Monma. Integer polyhedra arising from certain network design problems with connectivity constraints. *SIAM Journal on Discrete Mathematics*, 3(4):502–523, 1990.

[141] M. Grötschel and C. L. Monma. Integer polyhedra arising from certain network design problems with connectivity constraints. *SIAM Journal on Discrete Mathematics*, 3(4):502–523, 1990.

[142] M. Grötschel, C. L. Monma, and M. Stoer. Facets for polyhedra arising in the design of communication networks with low-connectivity constraints. *SIAM Journal on Optimization*, 2(3):474–504, 1992.

[143] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[144] S. Guillemot. Fpt algorithms for path-transversals and cycle-transversals problems in graphs. In *International Workshop on Parameterized and Exact Computation*, pages 129–140. Springer, 2008.

[145] J. Guo, F. Hüffner, E. Kenar, R. Niedermeier, and J. Uhlmann. Complexity and exact algorithms for vertex multicut in interval and bounded treewidth graphs. *European Journal of Operational Research*, 186(2):542–553, 2008.

[146] G. Hager, G. Jost, and R. Rabenseifner. Communication characteristics and hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Proceedings of Cray User Group Conference*, volume 4, page 5455, 2009.

[147] P. Hahn and T. Grant. Lower bounds for the quadratic assignment problem based upon a dual formulation. *Operations Research*, 46(6):912–922, 1998.

[148] P. Hahn, T. Grant, and N. Hall. A branch-and-bound algorithm for the quadratic assignment problem based on the hungarian method. *European Journal of Operational Research*, 108(3):629–640, 1998.

[149] P. Hahn, W. Hightower, T. Johnson, M. Guignard-Spielberg, and C. Roucairol. Tree elaboration strategies in branch-and-bound algorithms for solving the quadratic assignment problem. *Yugoslav Journal of Operations Research*, 11(1):41–60, 2001.

[150] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In *Meta-heuristics*, pages 433–458. Springer, 1999.

[151] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European journal of operational research*, 130(3):449–467, 2001.

[152] M. Hauptmann and M. Karpiński. *A compendium on steiner tree problems*. Inst. für Informatik, 2013.

[153] X. He. An improved algorithm for the planar 3-cut problem. *Journal of Algorithms*, 12(1):23–37, 1991.

[154] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical programming*, 1(1):6–25, 1971.

[155] M. Held, P. Wolfe, and H. P. Crowder. Validation of subgradient optimization. *Mathematical programming*, 6(1):62–88, 1974.

[156] D. S. Hochbaum and D. B. Shmoys. An o(|v|ˆ2) algorithm for the planar 3-cut problem. *SIAM Journal on Algebraic Discrete Methods*, 6(4):707–712, 1985.

[157] J. H. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM (JACM)*, 9(3):297–314, 1962.

[158] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. 1992.

[159] T. C. Hu. *Integer programming and network flows*. Addison-Wesley Publ., 1970.

[160] J. Hughes, G. Morton, J. Pechanec, C. Schuba, L. Spracklen, and B. Yenduri. Transparent multi-core cryptographic support on niagara cmt processors. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 81–88. IEEE Computer Society, 2009.

[161] D. Huygens, M. Labbé, A. R. Mahjoub, and P. Pesneau. The two-edge connected hop-constrained network design problem: Valid inequalities and branch-and-cut. *Networks*, 49(1):116–133, 2007.

[162] D. Huygens and A. Ridha Mahjoub. Integer programming formulations for the two 4-hop-constrained paths problem. *Networks*, 49(2):135–144, 2007.

[163] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner tree problem*, volume 53. Elsevier, 1992.

[164] J. JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[165] V. K. Janakiram, E. F. Gehringer, D. P. Agrawal, and R. Mehrotra. A randomized parallel branch-and-bound algorithm. *International Journal of Parallel Programming*, 17(3):277–301, 1988.

[166] X. Ji and J. E. Mitchell. Branch-and-price-and-cut on the clique partitioning problem with minimum clique size requirement. *Discrete Optimization*, 4(1):87–102, 2007.

[167] L. Jourdan, M. Basseur, and E.-G. Talbi. Hybridizing exact methods and metaheuristics: A taxonomy. *European Journal of Operational Research*, 199(3):620–629, 2009.

[168] A. B. Kahng and G. Robins. A new class of iterative steiner tree heuristics with good performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(7):893–902, 1992.

[169] Y. Kamidoi, N. Yoshida, and H. Nagamochi. A deterministic algorithm for finding all minimum k-way cuts. *SIAM Journal on Computing*, 36(5):1329–1341, 2006.

[170] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.

[171] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. 1972.

[172] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[173] M. Karpinski and A. Zelikovsky. New approximation algorithms for the steiner tree problems. *Journal of Combinatorial Optimization*, 1(1):47–65, 1997.

[174] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[175] K.-i. Kawarabayashi and M. Thorup. The minimum k-way cut of bounded size is fixed-parameter tractable. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 160–169. IEEE, 2011.

[176] H. Kerivin and A. R. Mahjoub. Design of survivable networks: A survey. *Networks*, 46(1):1–21, 2005.

[177] H. Kerivin, A. R. Mahjoub, and C. Nocq. (1,2)-survivable networks: Facets and branch&cut. *The Sharpest-Cut, M. Grötschel (Editor), MPS/SIAM Optimization*, pages 121–152, 2004.

[178] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[179] G. Kindervater. A parallel branch and bound algorithm for the job shop problem, presentation. In *8th European Conference on Operational Research, Lisbon*, 1986.

[180] G. A. Kindervater and J. K. Lenstra. An introduction to parallelism in combinatorial optimization. *Discrete Applied Mathematics*, 14(2):135–156, 1986.

[181] G. A. Kindervater and J. K. Lenstra. Parallel computing in combinatorial optimization. *Annals of Operations Research*, 14(1):245–289, 1988.

[182] G. A. P. Kindervater and H. W. J. M. Trienekens. Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research*, 33(1):65–81, 1988.

[183] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[184] D. König. Graphok és alkalmazásuk a determinánsok és a halmazok elméletére. *Mathematikai és Természettudományi Ertesito*, 34:104–119, 1916.

[185] R. Krauthgamer and U. Feige. A polylogarithmic approximation of the minimum bisection. *SIAM review*, 48(1):99–130, 2006.

[186] V. Kumar and L. N. Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE transactions on pattern analysis and machine intelligence*, (6):768–778, 1984.

[187] M. K. Labidi, I. Diarrassouba, A. R. Mahjoub, and A. Omrane. A parallel hybrid genetic algorithm for the k-edge-connected hop-constrained network design problem. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, GECCO '16, pages 685–692, New York, NY, USA, 2016. ACM.

[188] A. L. Lastovetsky. *Parallel computing on heterogeneous networks*, volume 24. John Wiley & Sons, 2008.

[189] R. Leroy. *Parallel Branch-and-Bound revisited for solving permutation combinatorial optimization problems on multi-core processors and coprocessors*. PhD thesis, Université Lille 1, 2015.

[190] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[191] H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[192] M. E. Lübbecke and J. Desrosiers. Selected Topics in Column Generation. *Operations Research*, 53(6):1007–1023, 2005.

[193] T. L. Magnanti and S. Raghavan. Strong formulations for network design problems with connectivity requirements. *Networks*, 45(2):61–79, 2005.

[194] A. R. Mahjoub. Two-edge connected spanning subgraphs and polyhedra. *Mathematical Programming*, 64(1-3):199–208, 1994.

[195] A. R. Mahjoub. On perfectly two-edge connected graphs. *Discrete Mathematics*, 170(1):153–172, 1997.

[196] A. R. Mahjoub. *Polyhedral Approaches*, pages 261–324. Wiley Online Library, 2013.

[197] S. Martin. *Analyse structurelle des systèmes algébro-différentiels conditionnels : compléxité, modèles et polyèdres*. PhD thesis, Université Paris-Dauphine, 2011.

[198] S. Martins, M. Resende, C. Ribeiro, and P. Pardalos. A parallel grasp for the steiner tree problem in graphs using a hybrid local search strategy. *Journal of Global Optimization*, 17(1-4):267–283, 2000.

[199] D. Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351(3):394–406, 2006.

[200] T. C. Matisziw and A. T. Murray. Modeling s–t path availability to support disaster vulnerability assessment of network infrastructure. *Computers & Operations Research*, 36(1):16–26, 2009.

[201] N. Melab. *Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul.* PhD thesis, 2005.

[202] K. Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[203] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[204] R. G. Michael and S. J. David. Computers and intractability: a guide to the theory of np-completeness. *WH Free. Co., San Fr*, 1979.

[205] D. L. Miller and J. F. Pekny. The role of performance metrics for parallel mathematical programming algorithms. *ORSA Journal on Computing*, 5:26–28, 1993.

[206] D. Mölle, S. Richter, and P. Rossmanith. A faster algorithm for the steiner tree problem. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 561–570. Springer, 2006.

[207] Y.-S. Myung and H.-j. Kim. A cutting plane algorithm for computing k-edge survivability of a network. *European Journal of Operational Research*, 156(3):579–589, 2004.

[208] W. Nasri. *Parallélisation d'algorithmes matriciels récursifs par le paradigme 'Diviser pour Paralléliser' dans des environnements homogène et hétérogène.* PhD thesis, Faculty of Sciences of Tunis, 202.

[209] G. Naves and V. Jost. The graphs with the max-mader-flow-min-multiway-cut property. *arXiv preprint arXiv:1101.2061*, 2011.

[210] G. L. Nemhauser and G. Sigismondi. A Strong Cutting Plane/Branch-and-Bound Algorithm for Node Packing. *The Journal of the Operational Research Society*, 43(5):443–457, 1992.

[211] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, 1988.

[212] M. Oosten, J. H. Rutten, and F. C. Spieksma. Disconnecting graphs by removing vertices: a polyhedral approach. *Statistica Neerlandica*, 61(1):35–60, 2007.

[213] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. SNDlib 1.0–Survivable Network Design Library. In *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*, April 2007. http://sndlib.zib.de, extended version accepted in Networks, 2009.

[214] I. H. Osman and G. Laporte. Metaheuristics: A bibliography, 1996.

[215] M. W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5(1):199–215, 1973.

[216] M. W. Padberg. A note on 0-1 programming. *Operations Research*, 23:833–837, 1979.

[217] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.

[218] C. Papadopoulos. Restricted vertex multicut on permutation graphs. *Discrete Applied Mathematics*, 160(12):1791–1797, 2012.

[219] J. F. Pekny and D. L. Miller. A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. *Mathematical programming*, 55(1-3):17–33, 1992.

[220] T. Polzin and S. V. Daneshmand. A comparison of steiner tree relaxations. *Discrete Applied Mathematics*, 112(1):241–261, 2001.

[221] T. Polzin and S. V. Daneshmand. Improved algorithms for the steiner problem in networks. *Discrete Applied Mathematics*, 112(1):263–300, 2001.

[222] H. J. Prömel and A. Steger. *The Steiner tree problem: a tour through graphs, algorithms, and complexity*. Springer Science & Business Media, 2012.

[223] E. Pruul, G. Nemhauser, and R. Rushmeier. Branch-and-bound and parallel computation: A historical note. *Operations Research Letters*, 7(2):65–69, 1988.

[224] J. Puchinger and G. R. Raidl. *Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification*. Springer, 2005.

[225] G. R. Raidl. A unified view on hybrid metaheuristics. In *Hybrid Metaheuristics*, pages 1–12. Springer, 2006.

[226] T. Ralphs. Parallel branch and cut. *Parallel Combinatorial Optimization*, 58:53, 2006.

[227] T. Ralphs and L. Ladányi. Symphony: A parallel framework for branch and cut. 1999.

[228] T. K. Ralphs. *Parallel Branch and Cut for Vehicle Routing.* PhD thesis, Ithaca, NY, USA, 1995. UMI Order No. GAX95-28230.

[229] T. K. Ralphs. Parallel branch and cut for capacitated vehicle routing. *Parallel Comput.*, 29(5):607–629, May 2003.

[230] T. K. Ralphs and M. Güzelsoy. *The Symphony Callable Library for Mixed Integer Programming*, pages 61–76. Springer US, Boston, MA, 2005.

[231] T. K. Ralphs, L. Ladányi, and M. J. Saltzman. A library hierarchy for implementing scalable parallel search algorithms. *The Journal of Supercomputing*, 28(2):215–234, 2004.

[232] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Problemata, 15. Frommann-Holzboog, 1973.

[233] C. C. Ribeiro and I. Rosseti. A parallel grasp heuristic for the 2-path network design problem. In *Euro-Par 2002 Parallel Processing*, pages 922–926. Springer, 2002.

[234] S. Ropke and J. F. Cordeau. Branch and Cut and Price for the Pickup and Delivery Problem with Time Windows. *Transportation Science*, 43(3):267–286, 2009.

[235] A. L. Rosenberg and L. S. Heath. *Graph separators, with applications.* Springer Science & Business Media, 2001.

[236] C. Roucairol. *Parallel computing in combinatorial optimization.* PhD thesis, INRIA, 1989.

[237] A. M. Rush and M. Collins. A tutorial on dual decomposition and lagrangian relaxation for inference in natural language processing. *Journal of Artificial Intelligence Research*, 2012.

[238] M. Salami and G. Cain. Genetic algorithm processor on reprogrammable architectures. In *Evolutionary Programming*, pages 355–361, 1996.

[239] H. Saran and V. V. Vazirani. Finding k-cuts within twice the optimal. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 743–751. IEEE, 1991.

[240] H. Saran and V. V. Vazirani. Finding k cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108, 1995.

[241] M. Savelsbergh. A branch-and-price algorithm for the generalized assignment problem. *Operations Research*, 45(6):pp. 831–841, 1997.

[242] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[243] A. Schrijver. *Combinatorial Optimization : Polyhedra and Efficiency. Algorithms and Combinatorics*, volume 24. Springer, 2003.

[244] J. Schrum and R. Miikkulainen. Solving interleaved and blended sequential decision-making problems through modular neuroevolution. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 345–352, New York, NY, USA, 2015. ACM.

[245] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[246] K. Steiglitz, P. Weiner, and D. Kleitman. The design of minimum-cost survivable networks. *IEEE Transactions on Circuit Theory*, 16(4):455–460, 1969.

[247] T. G. Stützle. *Local search algorithms for combinatorial problems: analysis, improvements, and new applications*, volume 220. Infix Sankt Augustin, 1999.

[248] E.-G. Talbi. A taxonomy of hybrid metaheuristics. *Journal of heuristics*, 8(5):541–564, 2002.

[249] S. Talukdar, L. Baerentzen, A. Gove, and P. De Souza. Asynchronous teams: Cooperation schemes for autonomous agents. *Journal of Heuristics*, 4(4):295–321, 1998.

[250] S. Talukdar, S. Murthy, and R. Akkiraju. Asynchronous teams. *Handbook of Metaheuristics*, pages 537–556, 2003.

[251] M. Thorup. Minimum k-way cuts via deterministic greedy tree packing. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 159–166. ACM, 2008.

[252] F. Vanderbeck. *Decomposition and column generation for integer programming.* PhD thesis, Université Catholique de Louvain, Belgium, 1994.

[253] M. G. A. Verhoeven and E. H. Aarts. Parallel local search. *Journal of Heuristics*, 1(1):43–65, 1995.

[254] S. Voss. *Steiner-probleme in graphen.* Anton Hain, 1990.

[255] S. Voß. Steiner's problem in graphs: heuristic methods. *Discrete Applied Mathematics*, 40(1):45–72, 1992.

[256] P. Winter. Steiner problem in networks: a survey. *Networks*, 17(2):129–167, 1987.

[257] P. Winter and J. M. Smith. Path-distance heuristics for the steiner problem in undirected networks. *Algorithmica*, 7(1-6):309–327, 1992.

[258] L. A. Wolsey and G. L. Nemhauser. *Integer and combinatorial optimization.* John Wiley & Sons, 2014.

[259] M. Xiao. Simple and improved parameterized algorithms for multiterminal cuts. *Theory of Computing Systems*, 46(4):723–736, 2010.

[260] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Alps: A framework for implementing parallel tree search algorithms. In *The next wave in computing, optimization, and decision technologies*, pages 319–334. Springer, 2005.

[261] M. Yannakakis. Node-deletion problems on bipartite graphs. *SIAM Journal on Computing*, 10(2):310–327, 1981.

Résumé

Dans cette thèse, nous considérons la classe POC des problèmes de conception d'un réseau fiable. Nous présentons un algorithme hybride parallèle d'approximation basé sur un algorithme glouton, un algorithme de relaxation Lagrangienne et un algorithme génétique, qui produit des bornes inférieure et supérieure pour les formulations à base de flows.

Afin de valider l'approche proposée, une série d'expérimentations est menée sur deux applications: le Problème de conception d'un réseau k-arête-connexe avec contrainte de borne (kHNDP) et le problème de conception d'un réseau fiable k-arête-connexe (kESNDP). L'étude expérimentale de la parallélisation est présentée après cela.

Dans la dernière partie de ce travail, nous présentons deux algorithmes parallèles exacts: un Branch-and-Bound distribué et un Branch-and-Cut distribué. Une série d'expérimentations a été menée sur une grappe de 128 processeurs, et des accélérations intéressantes ont été atteintes pour la résolution du problèmes kHNDP avec k = 3 et L = 3.

Abstract

In this thesis we consider the COP class of Survivability Network Design Problems. We present an approximation parallel hybrid algorithm based on a greedy algorithm, a Lagrangian relaxation algorithm and a genetic algorithm which produces both lower and upper bounds for flow-based formulations.

In order to validate the approach proposed, a series of experiments is condusted on two applications: the k-Edge-Connected Hop-Constrained Network Design Problem (kHNDP) and the k-Edge-Connected Survivability Network Design Problem (kESNDP). The parallelization experimental study is presented after that.

In the last part of this work, we present two parallel exact algorithms: a distributed Branch-and-Bound and a distributed Branch-and-Cut. A series of experiments has been made on a cluster of 128 processors and interesting speedups has been reached in kHNDP resolution when k=3 and L=3.

Mots Clés

Algorithme de Branch-and-bound, algorithme de Branch-and-cut, calcul parallèle, conception de réseau, hybridation, métaheuristique, optimisation combinatoire.

Keywords

Branch-and-Bound, Branch-and-Cut, Combinatorial optimization, hybridization, metaheuristic, network design, parallel computing.