UNIVERSITÉ DE NANTES

Résolution du problème du voyageur de commerce asymétrique par séparation et évaluation de sa relaxation combinatoire en problème d'affectation

Thibaut Barthélemy Nazim Coindet Cheikh Fall

Sous la direction du Docteur Anthony Przybylski

13 mai 2011

Table des matières

Chapitre 1

Introduction

Au 19ème siècle, William Rowan Hamilton, physicien Irlandais à l'origine notamment des quaternions et des progrès dans la formalisation de la mécanique quantique, s'intéresse à la détermination du plus petit cycle passant une seule fois par tous les sommets d'un graphe. Concrètement, cette question s'apparente à la recherche du plus court circuit passant par toutes les villes d'une carte, chacune ne pouvant être visitée qu'une fois. Originellement, on considérait des graphes non orientés, si bien que la distance entre deux villes ne dépendait pas du sens du trajet. Ce problème, symétrique, fut appelé « problème du voyageur de commerce », que l'on abrège par « TSP ». Ici, nous traiterons la variante asymétrique du problème, dite « ATSP », où les distances dépendent du sens de parcours.

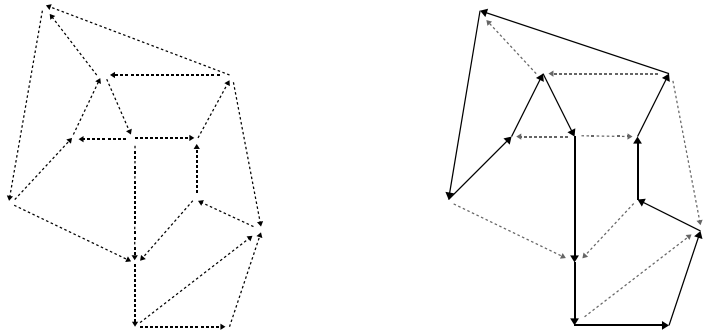

Figure 1.1 – À gauche, un graphe orienté. À droite, une solution.

L'intérêt du TSP et de ses variantes est fort. Si l'on y fait clairement face pour la planification de transports, le problème se pose aussi lors du routage de cartes électroniques et pour l'édification des liens entre les transistors d'un microprocesseur. En outre, et de manière tout aussi évidente, dans l'industrie, le perçage de trous sur une plaque, ou encore le dépôt de composants discrets sur un circuit imprimé avant soudure, sont des applications du TSP. Enfin, il en existe des applications moins intuitives. On y fait face lors de l'ordonnancement de tâches ou d'éléments dont l'ordre relatif importe mais pas leur positionnement absolu, ainsi qu'en cristallographie pour séquencer la mesure de rayons X selon de nombreux angles propres au cristal étudié [1].

Le nombre de solutions admissibles dans un graphe complet, même de faible cardinalité, est considérable. Soit V l'ensemble des sommets d'un graphe asymétrique complet. Le nombre de circuits hamiltoniens réalisables se calcule trivialement et vaut

$$(|V| - 1)\,!$$

Les meilleures méthodes connues pour résoudre l'ATSP peinent à traiter des instances aussi grandes que celles résolues par les méthodes dédiées au TSP symétrique. Quoi qu'il en soit, ces deux problèmes de voyageur de commerce sont classés *NP Complets* [2], si bien qu'il existe des instances insolubles en un temps acceptable. Beaucoup d'autres instances sont solubles malgré leur taille importante, grâce à des méthodes qui exploitent leurs particularités. En l'occurrence, nous allons nous attacher à résoudre le problème du voyageur de commerce asymétrique par séparation et évaluation d'un sous problème classé P. L'ATSP sera en effet relaxé en un sous problème d'affectation, évaluable en temps polynomial, et la séparation consistera en l'élimination des sous tours issus de la relaxation.

Chapitre 2

Formalisation

2.1 Le problème du voyageur de commerce

2.1.1 Description

Formellement, le problème du voyageur de commerce peut s'écrire comme un programme linéaire. Ci dessous, V est l'ensemble des n sommets du graphe. x_{ij} désigne l'arc (i, j) et vaut 1 s'il fait partie de la solution, 0 sinon. c_{ij} représente le poids de l'arc (i, j).

$$
\begin{cases}
\min \quad z = \displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} & & & \\
\text{s. t.} & & & \\
\qquad \displaystyle\sum_{j=1}^{n} x_{ij} = 1 & & \forall i \in V & (1) \\
\qquad \displaystyle\sum_{i=1}^{n} x_{ij} = 1 & & \forall j \in V & (2) \\
\qquad \displaystyle\sum_{(i,j)\in P^2} x_{ij} \leq |P| - 1 & & \forall P \subset V\,,\, P \neq \varnothing & (3) \\
\qquad x_{ij} \in \{0\,;1\} & & \forall (i,j) \in V^2 & (4)
\end{cases}
$$

(1) Chaque sommet i ne peut accepter qu'un arc sortant vers un autre sommet j.

(2) Chaque sommet j ne peut accepter qu'un arc entrant d'un autre sommet i.

(3) Dans tout sous ensemble P de sommets, sélectionner $|P|$ arcs permettrait de former des circuits hamiltoniens. On interdit cela.

2.1.2 Relaxation

En oubliant l'ensemble de n^2 contraintes (4), qui impose l'intégrité des variables x_{ij}, minimiser z tout en respectant l'ensemble d'inéquations (1), (2) et (3) serait réalisable par l'utilisation d'un algorithme tel que celui du simplexe. Cela constituerait une *relaxation continue* et, sauf cas d'une rareté extrême, les valeurs des x_{ij} de la solution ne seraient ni 1 ni 0. En effet, les algorithmes d'optimisation linéaire tel que celui cité ne fonctionnent que

dans l'ensemble des réels. Il y aurait donc nécessité de procéder, pour obtenir des solutions entières, à la séparation du programme linéaire relaxé en créant des sous espaces dans le polytope défini par les contraintes, jusqu'à forcer l'obtention de solutions admissibles (entières). L'arbre de recherche ainsi défini serait élagué de manière à n'effectuer que les séparations susceptibles de mener à la solution optimale. Ce procédé est couramment nommé *séparation et évaluation* et fut inventé en 1960 par les australiennes Ailsa H. Land et Alison G. Doig [3]. S'il permet d'obtenir les solutions optimales de certains programmes linéaires en un temps satisfaisant, il n'est pas le plus performant pour résoudre le TSP et l'ASTP. Nous le montrerons plus loin, à la section 2.3.

Pour ces problèmes, l'utilisation d'autres types de relaxation accélère la résolution de la quasi totalité des instances. Nous allons nous intéresser à une relaxation combinatoire. Cela consiste au retrait d'une ou plusieurs contraintes autres que celles d'intégrité. En l'occurrence, nous nous affranchissons de l'ensemble de contraintes (3). Ces dernières sont aussi nombreuses qu'il y a de parties dans V, sauf $\varnothing$ et V au complet, soit $2^n - 2$. La complexité du problème est diminuée car il n'y reste plus que $n^2 + 2n$ contraintes. Là est l'intérêt d'une relaxation combinatoire.

À cette étape, le programme linéaire à résoudre n'est plus constitué que d'un ensemble de $2n$ contraintes d'égalité, auquel s'ajoutent les n^2 contraintes d'intégrité. Si l'on exprime l'ensemble des contraintes comme le produit d'une matrice T par un vecteur x, le fait qu'aucune variable ne soit pondérée rend cette matrice totalement unimodulaire : le déterminant de toute sous matrice de T est dans $\{-1, 0, 1\}$. De plus, les membres de droite des contraintes sont tous entiers. Par conséquent, la résolution du programme linéaire sans tenir compte des contraintes d'intégrité produira tout de même une solution où les variables x_{ij} sont entières [4]. Mieux, les ensembles (1) et (2) restreignent implicitement les variables dans l'intervalle $[0\,;1]$. Notre relaxation combinatoire a autorisé une relaxation continue sans incidence sur l'intégrité des variables, ce qui retire du problème les n^2 contraintes les plus difficiles à satisfaire. Il n'en reste plus que $2n$.

2.1.3 Séparation en sous problèmes

Nous venons de retirer de nombreuses contraintes afin de diminuer la complexité de la tâche. En première conséquence, des sous ensembles de sommets parcourus par leur propre circuit hamiltonien apparaitront dans les solutions du problème relaxé. Nous appellerons cela des « sous tours ». Dans l'optique d'annihiler progressivement ces circuits, une séparation s'impose. Nous nous conformerons à la méthode que les italiens Giorgio Carpaneto et Paolo Toth ont publiée en 1980 [5].

Considérons une solution du problème relaxé telle que celle présentée à gauche de la figure 2.1. Les arcs en pointillés sont des arcs présents dans le graphe mais non sélectionnés dans la solution. Carpaneto et Toth recommandent d'utiliser le sous tour de plus faible cardinalité pour la séparation [1]. Ici, il s'agit de (1,2,3,4,5), de taille 5. Sur cette base, nous générons les 5 nouveaux sous problèmes suivants :

1. plus exactement, le sous tour dont le nombre d'arcs non imposés par la succession de séparations menant au sous problème est minimum

(a) L'arc (1,2) est retiré de l'instance.

(b) L'arc (2,3) est retiré de l'instance et l'arc (1,2) est imposé.

(c) L'arc (3,4) est retiré de l'instance et les arcs (2,3) et (1,2) sont imposés.

(d) L'arc (4,5) est retiré de l'instance et les arcs (3,4), (2,3) et (1,2) sont imposés.

(e) L'arc (5,1) est retiré de l'instance et les arcs (4,5), (3,4), (2,3) et (1,2) sont imposés.

À titre d'exemple, les contraintes de séparation qui génèrent le sous problème (c) sont représentées au centre de la figure. De nouveaux sous tours sont susceptibles d'apparaître au sein des solutions des différents sous problèmes (comme à droite sur la figure). L'arbre de recherche croît en conséquence, sur la base des nouveaux sous tours comportant le moins d'arcs non imposés. En règle générale, et c'est notamment là que repose la force du procédé, il n'est pas nécessaire d'imposer n arcs ou d'en retirer $n^2 - n$ pour qu'une solution au problème relaxé devienne admissible pour le problème originel.

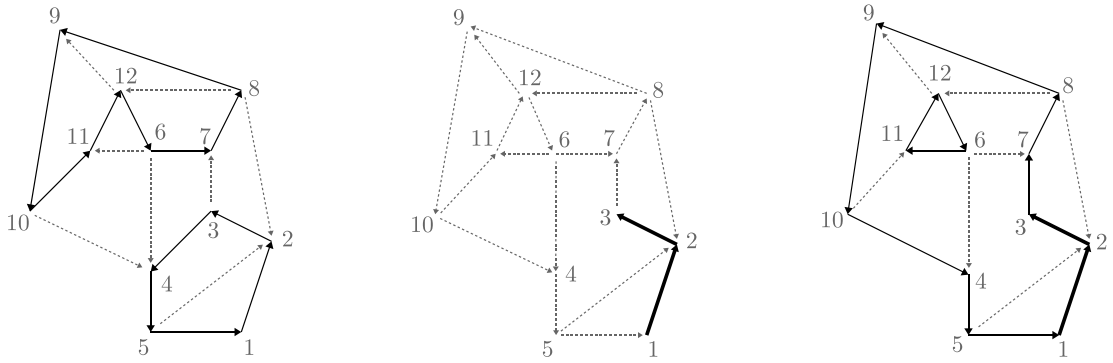

FIGURE 2.1 – Sélection, séparation, évaluation

La seconde conséquence de la relaxation est que la valeur optimale de la fonction objectif du problème relaxé à la racine de l'arbre ne peut dépasser la valeur optimale de la fonction objectif du problème originel. Considérant que les séparations successives consistent en l'ajout de contraintes, cette borne inférieure ne peut qu'augmenter lorsque l'on passe d'un nœud de l'arbre à son fils. De fait, un élagage de type *branch and bound* est possible. Au cours de l'exploration de l'arbre des sous problèmes, il est inutile de séparer ceux dont l'évaluation dépasse l'évaluation de la meilleure solution admissible connue pour l'ATSP. Par conséquent, de nombreuses branches de l'arbre ne seront pas explorées et un gain de temps substantiel en résultera.

2.2 Le sous problème d'affectation

2.2.1 Description

Suite à la relaxation décrite plus haut, $2n$ contraintes subsistent dans notre programme linéaire, contre $2^n + n^2 + 2n - 2$ à l'origine. Celles retirées étant les plus nombreuses (interdiction des $2^n - 2$ sous tours potentiels) et les plus difficiles à satisfaire (intégrité des variables), le problème résultant est sensiblement moins complexe que le problème originel. Il s'avère équivalent au *problème d'affectation de coût minimal*, classé P, là où le TSP et l'ATSP étaient classés *NP*. Voici sa formulation mathématique :

$$
\begin{cases}
\min \quad z = \displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} & & & \\
\text{s. t.} & & & \\
\qquad \displaystyle\sum_{j=1}^{n} x_{ij} & = 1 & \forall i \in I & (1) \\
\qquad \displaystyle\sum_{i=1}^{n} x_{ij} & = 1 & \forall j \in J & (2) \\
\qquad x_{ij} & \in \mathbf{R}_+ & \forall (i,j) \in I \times J &
\end{cases}
$$

(1) Dans le graphe bipartie de sommets (I, J), chaque sommet de I ne peut être affecté qu'à un sommet de J.

(2) Chaque sommet de J ne peut être affecté qu'à un sommet de I.

2.2.2 Relation avec le problème du voyageur de commerce

Avant de poursuivre, formalisons la relation entre le graphe bipartie de sommets (I, J) et le graphe de sommets V du problème initial. Posons $I = V$ et $J = V$. Nous considérons qu'on ne peut que sortir d'un sommet de I, et qu'on ne peut qu'entrer sur un sommet de J. Ainsi, l'affectation (i, j) sera interprétée comme un arc qui sort du sommet i de V pour entrer sur le sommet j de V. La figure 2.2 illustre cela.

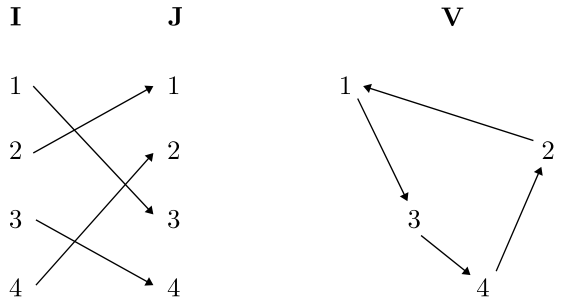

FIGURE 2.2 – Parallèle entre le problème d'affectation et l'ATSP

2.2.3 Premiers algorithmes

L'exactitude des premières méthodes de résolution du problème d'affectation fut prouvée grâce à un théorème découvert en 1931, indépendamment, par les mathématiciens hongrois Dénes KÖNIG et Jenö EGERVARY :

Théorème 1. Dans une matrice binaire [2], le nombre maximum de 0 qu'il est possible de considérer en s'interdisant d'en considérer plus d'un par ligne et par colonne, est égal au minimum de lignes et de colonnes qui permettent de couvrir tous les 0.

2. Matrice dont les éléments valent soit 0 soit 1.

Pour justifier les algorithmes sus-cités, le théorème est employé dans une forme équivalente, où les éléments de la matrice traitée par le problème sont catégorisés en deux groupes. Sont considérés d'une part les zéros, d'autre part les valeurs positives.

Parmi les nombreux algorithmes existants, le plus connu, peut-être par sa simplicité de mise en œuvre, est la *méthode hongroise*. Il fut conçu par l'américain Harold Kuhn en 1955 [6] et nommé en hommage à König et Egervary pour leur théorème essentiel. Pour autant, dès 1865, le mathématicien allemand Carl Gustav J. Jacobi avait publié une méthode similaire [7], ignorée à l'époque comme beaucoup de ses travaux. Aujourd'hui, Kuhn reconnait avoir été devancé d'un siècle [8].

Le chercheur américain James Munkres publia en 1957 une version améliorée de la méthode hongroise, dont il prouvera la complexité polynomiale [9]. Par ailleurs, des algorithmes travaillant sur des graphes biparties et dotés d'une complexité comparable furent proposés.

2.2.4 L'algorithme primal-dual

Fondements théoriques

Pour résoudre le problème d'affectation, plusieurs algorithmes de complexité équivalente existent et se basent souvent sur le théorème 1. Nous avons opté pour une variante issue de considérations propres à la programmation linéaire, basée sur un autre théorème. Elle fut notamment décrite par le belge Jacques Teghem en 2003 [4].

Théorème 2. *Théorème des écarts complémentaires.* Soient un programme linéaire et son dual. Lorsque les deux sont résolus à leur optimum, alors, quelle que soit la contrainte primale k considérée : soit sa variable d'écart e_k est nulle, soit la variable duale u_k associée à la contrainte est nulle, soit les deux sont nulles.

Ce théorème est lié à deux propriétés fondamentales des programmes linéaires, selon lesquelles, dans un problème résolu à l'optimum,

- les variables d'écart contribuent nullement à la fonction objectif : $\bar{c}_k e_k = 0$,
- au signe près, les coûts réduits $\bar{c}_k$ des variables d'écart sont les valeurs des variables duales u_k.

Soit $T_k\, x$ le membre de gauche de la $k^{\text{ème}}$ contrainte, et d_k sa borne. La première propriété s'écrit alors $\bar{c}_k\,(d_k - T_k x) = 0$. Enfin, à l'aide de la seconde propriété, nous pouvons poser

$$u_k\,(d_k - T_k x) = 0 \text{ ssi l'optimum est atteint.}$$

La réciproque de ce résultat s'obtient trivialement en considérant le problème dual comme étant primal, et vice versa. C'est elle que nous exploiterons dans la suite :

$$(c_k - uT_k)\, x_k = 0 \text{ ssi l'optimum est atteint.} \tag{2.1}$$

Pour pouvoir poursuivre, nous devons exprimer le programme linéaire dual du problème d'affectation. L'exercice étant difficile par la force de l'imagination, nous proposons d'observer un exemple, puis nous généraliserons. Soit le programme linéaire suivant, qui décrit le problème d'affectation dans une graphe bipartie de 2×3 sommets. Sa forme diffère de l'expression très formelle et générale du problème d'affectation que nous avons posée plus haut, mais s'y conforme tout à fait : les 3 premières contraintes expriment l'obligation d'affecter 1 et un seul sommet de la partie J à chaque sommet de la partie I ; les 3

dernières contraintes expriment l'obligation d'affecter 1 et un seul sommet de la partie I à chaque sommet de la partie J.

z	$c_{11}\,x_{11}$	$+$	$c_{12}\,x_{12}$	$+$	$c_{13}\,x_{13}$	$+$	$c_{21}\,x_{21}$	$+$	$c_{22}\,x_{22}$	$+$	$c_{23}\,x_{23}$	$+$	...	
u_1	x_{11}	$+$	x_{12}	$+$	x_{13}									$=1$
u_2							x_{21}	$+$	x_{22}	$+$	x_{23}			$=1$
u_3													...	$=1$
v_1	x_{11}					$+$	x_{21}					$+$	...	$=1$
v_2			x_{12}					$+$	x_{22}			$+$	...	$=1$
v_3					x_{13}					$+$	x_{23}	$+$	...	$=1$

Pour faciliter la lecture du programme dual, chacune de ses variables, associée à une contrainte du programme primal par nature, est inscrite à gauche. Le programme dual se lit dans les colonnes. Le voici :

$$\begin{cases} \max \quad z = u_1 + u_2 + u_3 + v_1 + v_2 + v_3 \\ \text{s. t.} \\ \qquad\quad u_1 + v_1 \;\; \leq c_{11} \\ \qquad\quad u_1 + v_2 \;\; \leq c_{12} \\ \qquad\quad u_1 + v_3 \;\; \leq c_{13} \\ \qquad\quad u_2 + v_1 \;\; \leq c_{21} \\ \qquad\quad u_2 + v_2 \;\; \leq c_{22} \\ \qquad\quad u_2 + v_3 \;\; \leq c_{23} \\ \qquad\quad \ldots \\ \qquad\quad u_i, v_j \;\; \in \mathbf{R} \end{cases}$$

Le programme linéaire dual d'un problème d'affectation se généralise alors intuitivement :

$$\begin{cases} \max \quad z = \displaystyle\sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \\ \text{s. t.} \\ \qquad\quad u_i + v_j \;\; \leq c_{ij} \quad \forall (i,j) \in \{1...n\}^2 \end{cases}$$

Nous avons vu que la relation fondamentale 2.1 s'appliquait à toutes les contraintes d'un programme linéaire. Pour les contraintes du dual du problème d'affectation, elle s'écrit

$$(c_{ij} - u_i - v_j)\, x_{ij} = 0 \text{ ssi l'optimum est atteint.} \tag{2.2}$$

À présent, construisons une matrice pour y ranger les x_{ij}. Dans la mesure où cette matrice va nous servir à ranger d'autres valeurs, nous gagnons en clarté en représentant les x_{ij} qui valent 1 par un carré dans la case (i, j), et rien sinon. À gauche de la figure 2.3, nous voyons par exemple que x_{31} vaut 1.

Dans la matrice, nous superposons les valeurs de $c_{ij} - u_i - v_j$ (à droite sur la figure). Nous remarquons immédiatement que les cases sélectionnées sont des cases où sont inscrits des zéros. Cela est dû à la relation 2.2. En effet, pour qu'elle soit respectée lorsque x_{ij} vaut 1, $c_{ij} - u_i - v_j$ doit être nul. Notons également que lorsqu'une variable duale change de valeur, c'est toute une ligne ou toute une colonne qui est modifiée.

La matrice que nous venons de construire est dite « matrice des coûts réduits ».

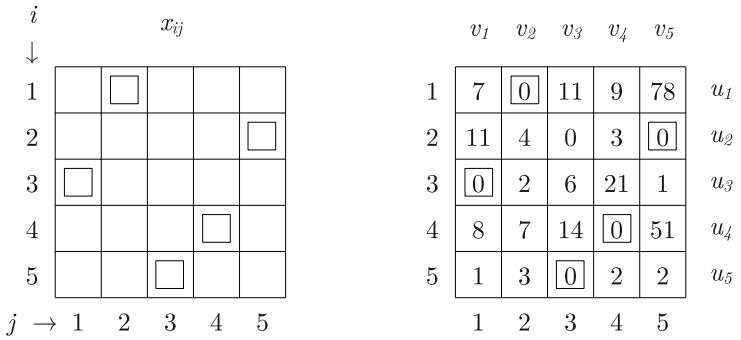

FIGURE 2.3 – À gauche, la matrice des x_{ij}. À droite, nous superposons les $c_{ij} - u_i - v_j$.

Algorithme de résolution

L'ensemble des considérations ci-avant étant prises, nous sommes en mesure de résoudre notre sous problème d'affectation. Nous savons à présent que les arcs dont la sélection engendre un coût optimal correspondent aux éléments de la matrice de valeur nulle. Deux phases sont nécessaires pour résoudre le problème ; l'une pour faire apparaître les premiers éléments de valeur zéro et en sélectionner quelques uns, l'autre pour faire croître le nombre de zéros sélectionnés et en calculer de nouveaux au besoin.

L'algorithme 1 (page suivante) présente la première phase, où l'on recherche l'élément minimum de chaque ligne pour le soustraire à l'ensemble de la ligne. On procède de même pour les colonnes. Ainsi, il existe au moins un zéro par ligne et par colonne. Si les valeurs minimales sont choisies, c'est pour éviter l'apparition d'éléments négatifs, qui violeraient les contraintes du problème dual. En effet, une valeur $c_{ij}-u_i-v_j$ négative impliquerait $u_i-v_j > c_{ij}$. Suite à cette première création de zéros, il est possible de fixer quelques variables x_{ij} à 1. Concrètement, on sélectionne des zéros de manière gloutonne, sous contrainte de n'en sélectionner qu'un seul par ligne et par colonne. Il s'agit d'une première affectation, souvent partielle. En effet, dans la plupart de cas, trop peu de zéros sont créés par les opérations de soustraction pour qu'il soit possible de tous les sélectionner. Et lorsque cela est possible, l'affectation gloutonne a tout de même de grandes chances d'échouer. On passe alors à la seconde phase, plus complexe, dont le point d'entrée est l'algorithme 2.

Tant qu'il existe au moins une ligne i_0 sans zéro sélectionné, nous la prenons comme point de départ pour la recherche d'une chaine augmentante, par un appel à la fonction de repérage *scanner_ligne* (algorithme 3). Cette dernière superpose l'ensemble de toutes les chaines alternées[3] réalisables à partir de la ligne i_0, c'est pourquoi elle est récursive.

Si elle rencontre une colonne ne comportant aucun zéro sélectionné, alors une chaine augmentante est trouvée. Le cas échéant, un *backtrack* nous fait revenir immédiatement dans l'algorithme 2, où nous lançons une procédure d'alternance. Cette dernière utilise les repères mis en place par la fonction *scanner_ligne* pour évoluer le long de la chaine augmentante. Elle y sélectionne tous les zéros qui n'étaient pas sélectionnés et inversement. Du fait qu'il y avait un zéro non sélectionné à chaque extrémité de la chaine, l'alternance provoque l'apparition d'une sélection supplémentaire.

3. Succession de colonnes accédées par leurs zéros non sélectionnés et de lignes accédées par leurs zéros sélectionnés. Dans un graphe, il s'agit d'arcs consécutifs et opposés dont la sélection engendre un coût optimal.

Il arrive aussi que la fonction *scanner_ligne* ne découvre aucune chaine augmentante, auquel cas elle ne retourne pas d'entier valide à l'algorithme 2. Dans un tel cas, ce dernier lance la procédure de calcul de nouveaux zéros décrite par l'algorithme 4. L'addition alors réalisée dans les colonnes peut provoquer la disparition de tous les zéros d'une ligne. C'est pourquoi nous corrigeons son travail juste après l'appel. Enfin, le ou les nouveaux zéros apparus via ces opérations autorisent le repérage d'une ou plusieurs nouvelles colonnes dans la matrice. Elles comporteront soit un zéro sélectionné qui permettra la poursuite du processus de repérage, soit nous serons face à une chaine augmentante et nous pourrons sélectionner un zéro de plus.

L'algorithme s'arrête lorsqu'il s'apprête à sélectionner un élément de la matrice correspondant à un arc non présent dans le graphe. Le sous problème est alors insoluble, ce qui peut arriver suite aux retraits d'arcs opérés par la séparation (section 2.1.3) et par notre optimisation dynamique (section 3.4).

algorithme *initialiser*

i, j : **entiers**
a_{ij} : éléments de la matrice d'adjacence du graphe d'ordre n
m : **entier**

pour chaque $i \in \{1...n\}$ **faire**
    $m \leftarrow \min_{j \in \{1...n\}} a_{ij}$
    $a_{ij} \leftarrow a_{ij} - m \quad \forall j \in \{1...n\}$
fin
pour chaque $j \in \{1...n\}$ **faire**
    $m \leftarrow \min_{i \in \{1...n\}} a_{ij}$
    $a_{ij} \leftarrow a_{ij} - m \quad \forall i \in \{1...n\}$
fin
si l'un des minima correspond à un arc (i, j) retiré du graphe **alors**
    L'instance est insoluble.
fin

pour chaque ligne i **faire**
    Sélectionner un seul zéro qui soit situé dans une colonne où aucun autre n'est sélectionné.
fin
fin

Algorithme 1 : Initialisation et réalisation d'une première affectation

algorithme *résoudre*

j' : **entier**
m : **entier**

tant que il existe une ligne i_0 sans zéro sélectionné **faire**
    Annuler tout repérage des lignes et des colonnes.
    $j' \leftarrow scanner\_ligne(i_0, -1)$
    si j' est un entier désignant la colonne de fin d'une chaine augmentante **alors**
        Alterner la sélection des zéros sur la chaine de i_0 à j' définie par les repères.
    sinon
        $calculer\_de\_nouveaux\_zéros()$
        pour chaque ligne i ne contenant pas de zéro **faire**
            $m \leftarrow \min_{j \in \{1...n\}} a_{ij}$
            $a_{ij} \leftarrow a_{ij} - m \quad \forall j \in \{1...n\}$
        fin
        si l'un des minima correspond à un arc (i, j) retiré du graphe **alors**
            L'instance est insoluble.
        fin
    fin
fin
fin

Algorithme 2 : Augmentation progressive du nombre de zéros sélectionnés

fonction *scanner_ligne* $(i, j$: **entiers**$)$: **entier**

i', j', j'' : **entiers**

Attribuer un repère à la ligne i pour lui affecter j comme prédécesseur.

pour chaque zéro de la ligne i **faire**
    $j' \leftarrow$ indice de la colonne où se trouve le zéro
    si la colonne j' n'est pas repérée **alors**
        Attribuer un repère à la colonne j' pour lui affecter i comme prédécesseur.
        si la colonne j' contient des zéros mais qu'aucun n'est sélectionné **alors**
            retourner j'
        sinon
            $i' \leftarrow$ indice de la ligne où se situe le zéro sélectionné de la colonne
            $j'' \leftarrow scanner\_ligne(i', j')$
            si j'' est un entier désignant la colonne de fin d'une chaine augmentante **alors**
                retourner j''
            fin
        fin
    fin
fin
retourner "aucune chaine augmentante trouvée"
fin

Algorithme 3 : Recherche exhaustive d'une chaine augmentante

algorithme *calculer_de_nouveaux_zéros*

i, j : **entiers**
a_{ij} : éléments de la matrice des coûts réduits
m : **entier**

Soient I l'ensemble des lignes repérées et J l'ensemble des colonnes non repérées.
$m \leftarrow \min_{(i,j) \in I \times J} a_{ij}$

si le minimum correspond à un arc (i, j) retiré du graphe **alors**
    L'instance est insoluble.
fin

pour chaque $i \in I$ **faire**
    $a_{ij} \leftarrow a_{ij} - m \quad \forall j \in J$
fin
pour chaque $j \in \bar{J}$ **faire**
    $a_{ij} \leftarrow a_{ij} + m \quad \forall i \in \bar{I}$
fin
fin

Algorithme 4 : Calcul de nouveaux zéros

2.3 Complexité des algorithmes

2.3.1 Évaluation

Il est constaté et démontré que l'algorithme du simplexe effectue $\Theta(m)$ itérations pour résoudre la plupart des programmes linéaires soumis à m contraintes [10]. Cependant, lorsqu'il doit traiter des programmes tels que le problème d'affectation, il s'éloigne nettement de cette moyenne.

Considérons le problème d'affectation, qui comporte $2n$ contraintes, dont $2n - 1$ sont indépendantes. Nous savons que, à l'optimum, seules n variables vaudront 1. Sur $2n - 1$ variables en base, $n - 1$ seront nulles. Cette situation est défavorable à l'algorithme du simplexe, qui effectuera de nombreuses itérations inutiles. De ce fait, notre problème est dit *dégénéré*.

Il existe cependant d'autres algorithmes de résolution de programmes linéaires. Le plus efficace d'entre eux fut publié en 1984 par un mathématicien indien, Narendra Karmarkar [11]. Il s'agit d'un algorithme *de point intérieur* d'une complexité en $O(n'^{3.5})$ où n' est le nombre de variables. La résolution de notre problème d'affectation par cette méthode présenterait donc une complexité en $O(n^7)$.

Enfin, comme nous l'avons vu à la section 2.2.3, plusieurs algorithmes spécifiques à ce problème existent. Parmi les plus efficaces figure la méthode hongroise améliorée par James Munkres en 1957 [9]. Sur une matrice d'adjacence de taille $n \times n$, Munkres borna la quantité d'opérations requises par sa méthode par

$$\frac{11n^3 + 12n^2 + 31n}{6},$$

soit une complexité en $O(n^3)$. Aucune méthode quadratique ou linéaire générale n'existe à ce jour. Nous avons opté pour un algorithme similaire, à savoir l'algorithme primal dual.

2.3.2 Séparation

Avec les algorithmes dont l'efficacité repose sur un élagage, il est très difficile d'exprimer une complexité moyenne. Nous nous contenterons d'une borne supérieure. Soit un graphe complet de cardinalité n. Dans le pire cas, l'instance est telle qu'en chaque nœud de l'arbre de séparation, tous les sommets n'appartenant pas aux arcs imposés par les ancêtres du nœud sont groupés par paires dans des sous tours de taille 2.

Ainsi, en moyenne, $\frac{1}{2}$ arc est imposé en chaque nœud pour séparer le problème : la première séparation ne fait que retirer un arc, la seconde en impose un. L'édification d'un circuit hamiltonien dans un graphe de n sommets nécessite n arcs. Nous pouvons alors estimer la profondeur moyenne de l'arbre de recherche par $2n$. Les sous tours étant tous supposés de taille 2, chaque nœud possède 2 fils. Par conséquent, au niveau p de l'arbre se trouvent 2^p nœuds. Le nombre de sous problèmes évalués sur notre instance difficile vaut donc approximativement

$$\sum_{p=0}^{2n} 2^p \,,$$

d'où

$$2^{2n+1} - 1 \,.$$

Nous retiendrons que la taille l'arbre de séparation de Carpaneto et Toth est, sur les pires instances, bornée en $\mathrm{O}(4^n)$. En pratique, l'élagage réalisé à l'aide des évaluations et la nature asymétrique du graphe [4] rendent la méthode très efficace.

4. Si l'on traite un TSP symétrique, l'affectation tend à produire $\frac{n'}{2}$ sous tours, où n' est le nombre de sommets n'appartenant pas à des arêtes imposées. Elle place l'algorithme de séparation et évaluation dans son pire cas, dont nous venons de donner la complexité.

Chapitre 3

Pour une résolution efficace

3.1 Sélection

3.1.1 Règle de choix

Les performances d'un algorithme de séparation et évaluation reposent en partie sur la règle qui guide la recherche dans l'arbre. À chaque itération, cette règle sélectionne le sous problème à séparer. Nous avons voulu trouver un compromis entre les deux types d'exploration primitifs, à savoir l'exploration en profondeur (figure 3.1) et l'exploration en largeur (figure 3.2). Pour cela, nous avons observé une propriété élémentaire sur les arbres de recherche par séparation et évaluation : la structure en tas. Pour un problème de minimisation, tout nœud a une valeur supérieure ou égale à son père, car nous passons d'un nœud à ses fils par l'ajout d'une ou plusieurs contraintes.

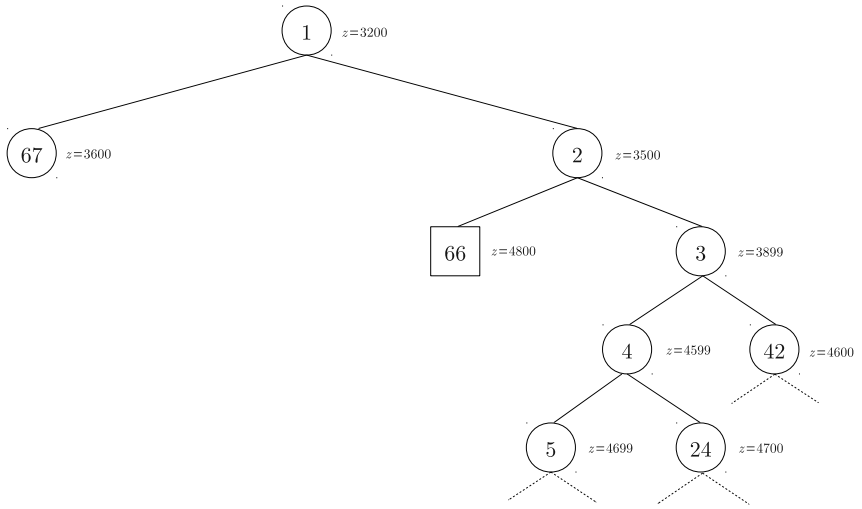

FIGURE 3.1 – Un arbre de séparation et évaluation exploré en profondeur

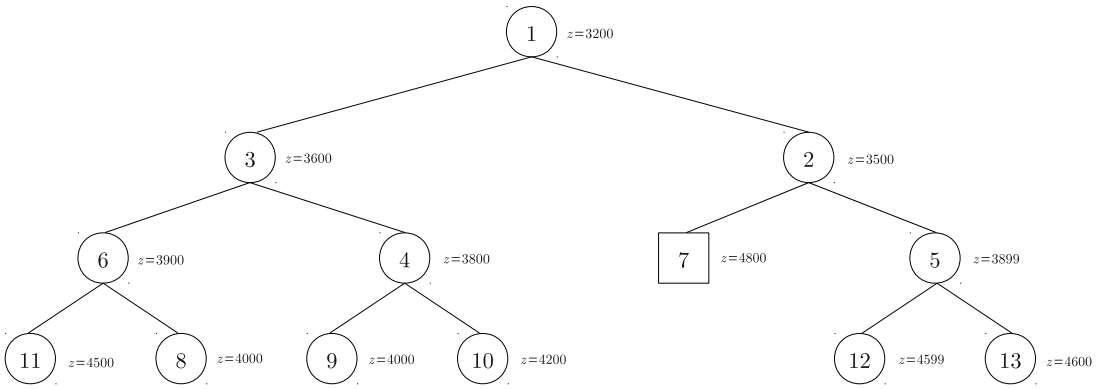

FIGURE 3.2 – Un arbre de séparation et évaluation exploré en largeur

Nous proposons de modéliser l'exploration en profondeur par une règle de choix qui considérerait l'ensemble des feuilles de l'arbre à chaque itération, et élirait celle dont la profondeur est la plus grande. Cette feuille serait séparée et, grâce à notre règle, l'une de ses filles serait élue à l'itération suivante. Les parcours en profondeur sont rarement utilisés, car ils enferment la recherche dans un sous espace de problèmes de plus en plus restreint, si bien que l'on a de grandes chances d'échapper pour longtemps à de bonnes solutions situées ailleurs. Un autre effet négatif en découle, à savoir que l'on traite des nœuds profonds qu'on aurait élagués si une solution de qualité avait été obtenue plus tôt.

Nous proposons d'approcher l'exploration en largeur par une règle de choix qui considérerait l'ensemble des feuilles de l'arbre à chaque itération, et élirait celle dont la fonction objectif est la plus faible. Par la structure en tas, les feuilles de plus faible profondeur seraient le plus souvent choisies et séparées, ce qui tendrait à compléter les niveaux internes et étalerait l'arbre en largeur. L'obtention de solutions admissibles peut être très longue si l'instance est telle que le nombre de séparations pour y parvenir est important.

Première approche

Des deux modélisations ci-avant, nous pourrions élaborer un compromis en classant les feuilles par ordre décroissant de

$$\frac{p}{(1+z-\check{z})^{\alpha}}$$

et en élisant, à chaque itération, la première feuille du classement. p est la profondeur de la feuille, z son coût et $\check{z}$ le coût du problème initial relaxé. Lorsque $\alpha \to 0$, le classement ne dépend que de p et l'on réalise un parcours en profondeur, comme vu ci-dessus. À contrario, lorsque $\alpha \to \infty$, la valeur de z est prépondérante et détermine seule le classement des feuilles [1]. On tend alors à étaler l'arbre en largeur. Dans l'optique d'un compromis, nous nous intéresserons aux valeurs autres que ces deux extrêmes.

1. Nous démontrons cela à travers l'emploi d'une fonction monotone croissante dont la plus adaptée ici est $x \mapsto \sqrt[\alpha]{x}$. De par son caractère monotone croissant, le classement d'éléments selon leur valeur x est identique à leur classement à travers $\sqrt[\alpha]{x}$. Considérant cela, si nous appliquons la fonction à notre expression, nous obtenons un classement selon $\frac{\sqrt[\alpha]{p}}{1+z-\check{z}}$. Cette expression, lorsque $\alpha \to \infty$, est équivalente à $\frac{1}{1+z-\check{z}}$, si bien que le classement ne dépend plus que de z.

Seconde approche

Notons que notre compromis entre les deux types d'exploration primitifs possède un second sens tout aussi fort. Si l'évaluation z_A d'un sous problème A est faible malgré une grande quantité de séparations (p_A élevé), A a une plus grande probabilité d'être proche de la solution optimale qu'un sous problème B doté d'une évaluation z_B élevée malgré peu de séparations (p_B faible). Pour cette raison, notre formule tend à nous faire découvrir de bonnes solutions rapidement.

Cependant, considérer l'évaluation d'un sous problème comme seule estimation de sa proximité avec la solution optimale du problème originel serait une erreur. Nous devons garder à l'esprit qu'une évaluation est d'autant plus proche de la solution optimale du problème originel que les contraintes respectées sont nombreuses. En l'occurrence, s sous tours dans la solution à un sous problème sont autant de contraintes non respectées pour l'ATSP, si bien que le sous problème ne peut pas être considéré à égalité avec un autre de même coût mais comportant moins de sous tours. C'est pourquoi nous évaluerons l'intérêt des sous problèmes par la formule

$$i(p, s, z) = \frac{p}{(1 + z - \check{z})^{\alpha} s^{\beta}} .$$

Néanmoins, un sous problème admet souvent plusieurs affectations, desquelles découlent des sous tours en différentes quantités. Nous verrons plus loin que la prise en compte du nombre de sous tours dans $i(p, s, z)$ n'est pertinente qu'à condition que ce nombre provienne de la solution qui en comporte le moins.

Paramétrisation

Pour s'écarter des cas extrêmes présentés ci-avant, α et β ne doivent ni tendre vers zéro, ni être trop grands. Nous avons déterminé empiriquement que les valeurs donnant les meilleurs résultats sur la plupart des instances de la TSPlib [2] sont $\alpha = 2$ et $\beta = 1$.

3.1.2 Accès au nœud choisi

Pour classer le plus rapidement possible les nœuds de l'arbre par intérêt, mais aussi par évaluation, nous avons utilisé un arbre AVL [12], instancié pour chacun des deux classements. Ce type d'arbres auto équilibrés garantit des temps d'accès et d'insertion logarithmiques en le nombre de sous problèmes ouverts. La figure 3.3 illustre le lien entre les 3 arbres manipulés par notre solveur.

Nous n'avons pas employé des tas binaires car leur implémentation la plus simple, à base de tableaux, ne garantit pas la constance des adresses des éléments. Dans le détail, les opérations d'ajout et de retrait d'éléments déclenchent des mécanismes de *percolation* qui permutent physiquement les nœuds du tas. À la $t^{\text{ième}}$ découverte d'une solution admissible de coût $\hat{z}(t) < \hat{z}(t-1)$, si le tas classant les sous problèmes par ordre décroissant de $\hat{z}$ nous permettrait effectivement d'accéder à ceux qui doivent être fermés, il serait par contre impossible de retirer ces sous problèmes du tas qui les classe par intérêt. Les opérations effectuées sur ce dernier entre le moment où les feuilles en question ont été ajoutées et le moment où nous souhaiterions les retirer ont potentiellement invalidé leurs adresses.

2. bibliothèque d'instances de référence

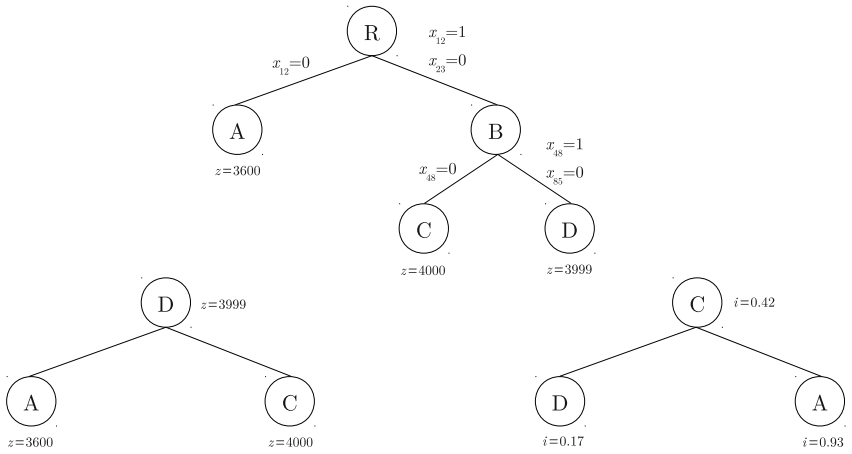

FIGURE 3.3 – Au centre, les feuilles d'un arbre de recherche candidates à la séparation. À gauche, l'AVL de tri selon z. À droite, l'AVL de tri selon leur intérêt.

3.2 Séparation

3.2.1 Obtention des sous problèmes

Dans l'arbre de recherche par séparation et évaluation, les nœuds internes sont des sous problèmes évalués et séparés. Seules les feuilles sont à traiter. Chacune appartient à l'espace de sous problèmes défini par ses prédécesseurs. Pour la traiter, il peut paraître inutile de regénérer son espace en remontant l'arbre de manière à ajouter au problème initial les contraintes de séparation présentes entre la feuille et la racine. Il peut sembler préférable de sauvegarder la matrice des coûts réduits du sous problème en chaque nœud. Lors de la séparation d'une feuille, on ne modifierait que quelques éléments de la matrice, si bien que l'évaluation des sous problèmes fils pourrait être relativement rapide. Plus précisément, sa complexité serait en $\mathrm{O}(n'^2 \log n)$, où n' est le cardinal du graphe du sous problème, qui diminue avec la profondeur de la feuille [13]. En effet, au fil des séparations, de moins en moins de sommets sont présents, du fait de l'imposition d'arcs.

Si cette méthode est élégante et efficace en temps, elle est aussi problématique de par ses besoins en mémoire. Nous avons vu précédemment qu'il est généralement judicieux de ne pas s'enfermer dans un parcours en profondeur dans l'arbre, mais préférable de se donner la liberté de séparer toute feuille qui nous semble prometteuse. Il en résulte un grand nombre de sous problèmes en attente de séparation. La présence d'une matrice d'adjacence et d'une matrice de coûts réduits en chaque feuille consommerait une mémoire importante que nos ordinateurs n'ont pas.

Pour cette raison, nous avons opté pour l'enregistrement des contraintes de séparation en chaque nœud. La génération du sous problème associé à toute feuille se réalisera en remontant l'arbre.

3.2.2 Génération des sous problèmes

Comme nous l'avons vu à la section 2.1.3, la séparation se réalise en retirant des arcs et en en imposant d'autres. Si le retrait d'un arc est trivial à réaliser, l'imposition d'un arc est plus complexe, du moins lorsqu'on la réalise de manière efficace.

Imposition d'un arc

Prenons le graphe d'un sous problème représenté à gauche de la figure 3.4. L'arc en gras est un arc que nous avons imposé. Les arcs (6,7) et (3,4) sont devenus inutiles du fait qu'il n'y ait plus d'autre possibilité que (3,7) pour entrer sur (7) et (3,7) pour sortir de (3). Ainsi, comme le montre le graphe central, un « nettoyage » est réalisé. Cela aura pour effet de diminuer la complexité de l'évaluation du sous problème. Enfin, toujours dans un souci d'efficacité, nous pouvons contracter les sommets (3) et (7) en un seul sommet (a).

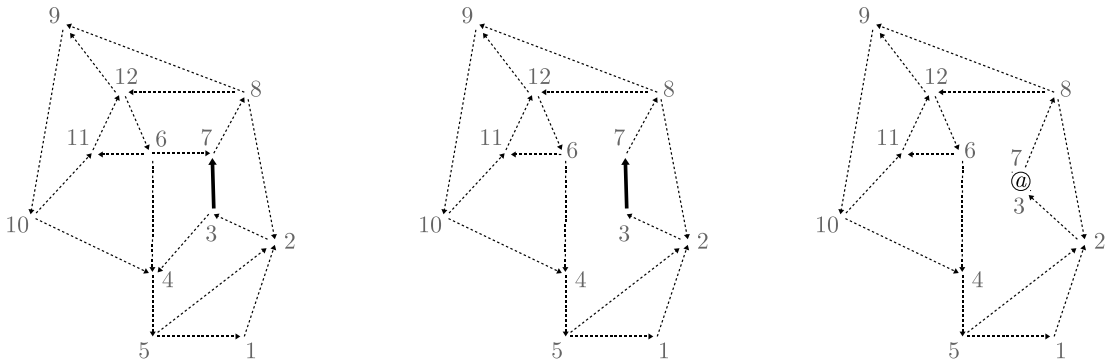

FIGURE 3.4 – Imposition de l'arc (3,7), nettoyage, contraction des extrémités

Plaçons-nous du point de vue de la matrice d'adjacence du graphe. Conformément à la convention adoptée à la section 2.2.2 pour lier le problème d'affectation à l'ATSP, il n'est possible de sortir que des sommets i représentés par les lignes d'indices i de la matrice. De même, il n'est possible d'entrer que sur les sommets j représentés par les colonnes d'indices j de la matrice. Dans la mesure où nous imposons l'arc (3,7), il est inutile de conserver la ligne 3 qui autorise à sortir de (3), tout comme il est inutile de conserver la colonne 7 qui autorise à entrer sur (7). Par conséquent, d'un point de vue matriciel, l'opération de nettoyage se réalise par le retrait d'une ligne et d'une colonne (figure 3.5).

En conséquence des retraits, la contraction des sommets est obligatoire, car nous devons respecter un parallèle fort entre l'instance de l'ATSP et l'instance du problème d'affectation associé ; si les coûts d'arrivée sur le sommet (a), qui contracte (3) et (7), sont situés dans une colonne d'indice a, alors les coûts de sortie doivent être situés dans la ligne d'indice a. Concrètement, sur notre exemple, nous choisissons $a = 3$, suite à quoi nous déplaçons la ligne 7 pour lui faire prendre la place de la ligne 3, retirée lors du nettoyage.

Par souci d'efficacité, nous ne pouvions pas réaliser chaque contraction à l'aide de copies et d'écrasements de valeurs dans la matrice d'adjacence du graphe. Voici notre méthode, en deux phases.

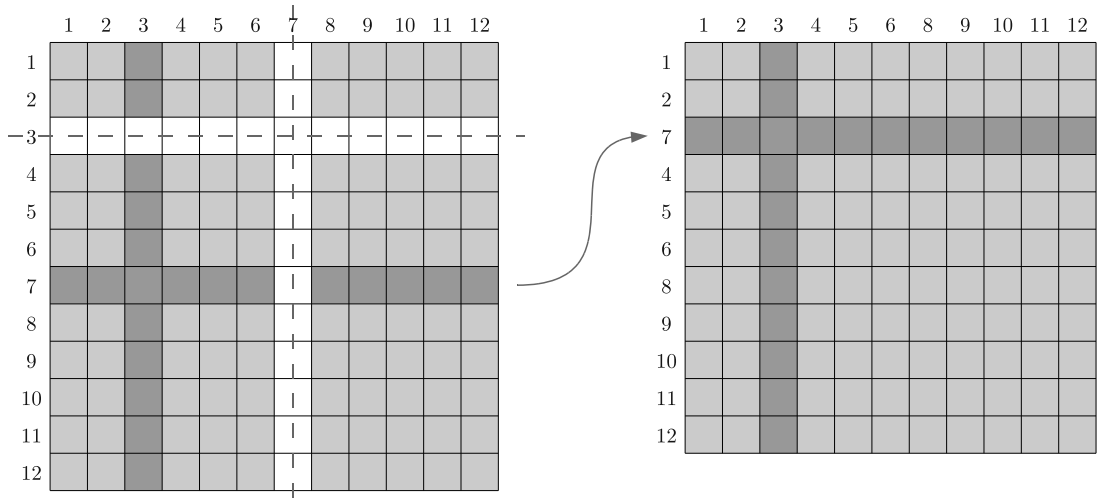

FIGURE 3.5 – Contraction des sommets (3) et (7) dans la matrice d'adjacence

Soit un arc (i, j) dont l'on souhaite contracter les extrémités. Ces dernières peuvent être le début et la fin de chemins précédemment contractés. Nous noterons i' le premier sommet du chemin contracté terminé par i, et j' le dernier sommet du chemin contracté ouvert par j. Pour obtenir i' et j', nous maintenons une *table des chemins contractés*, qui indique les sommets de début et de fin des chemins déjà contractés dans le graphe. La figure 3.6 illustre son contenu. Soient I et J deux listes doublement chainées contenant respectivement les indices des lignes et des colonnes de la matrice. Pour chaque arc (i, j) que nous devons imposer, nous préparons la contraction des sommets i et j comme suit :

- nous retirons le maillon j de la liste J, ainsi que le maillon i de la liste I,
- nous retirons le maillon numéro j' de la liste I, puis le réintégrons en le liant entre le prédécesseur et le successeur de i (cela aura pour effet de déplacer la ligne numéro j' dans la matrice),
- et nous mettons à jour la table des chemins contractés en inscrivant i' et j' en tant que début et fin du nouveau chemin $(i' \cdots j')$.

Au terme de cette étape préparatoire, lorsque l'arbre des séparations a été parcouru jusqu'au sommet, nous entamons la deuxième phase du processus : la génération de la matrice d'adjacence du sous problème. Nous parcourons les listes chaînées I et J, parcours qui nous fournit l'ordre dans lequel les lignes et les colonnes de la matrice M originelle doivent être recopiées dans une matrice M' pour constituer l'instance du sous problème.

Comme l'aura compris le lecteur, lors de la première phase, les opérations de retrait et de réinsertion sont effectuées en O(1). Aucune recherche dans les listes I et J n'est nécessaire car leurs maillons sont les éléments d'une table de taille n. Enfin, la recopie partielle de M vers M', dans l'ordre indiqué par les listes chainées, est réalisée en un minimum de temps puisque seuls les indices présents dans les listes I et J sont énumérés.

Pour imposer un arc (i, j), il aurait été plus simple d'attribuer une valeur excessivement élevée aux poids c_{ix} et c_{yj}, avec $x \in \{1...n\}\backslash\{j\}$ et $y \in \{1...n\}\backslash\{i\}$. Mais, en tout nœud de l'arbre de recherche, nous aurions eu à résoudre un problème d'affectation de taille n. Or, la complexité du problème est cubique, si bien que sa résolution nécessite de l'ordre

	1	2	3	4	5	6	7	8	9	10	11	12
i'	1	2	**3**	4	5	6	**3**	8	9	10	11	12
j'	1	2	**7**	4	5	6	**7**	8	9	10	11	12

	1	2	3	4	5	6	7	8	9	10	11	12
i'	**1**	2		4	5	6	**1**	8	9	10	11	12
j'	**7**	2		4	5	6	**7**	8	9	10	11	12

FIGURE 3.6 – Table des chemins contractés. À gauche, son état suite à la contraction des sommets (3) et (7). À droite, suite à la contraction des chemins (1) et (3-7).

de $O(n^2)$ opérations en moins lorsqu'on contracte deux sommets [3]. Nous ne pouvions pas nous abstraire d'un tel gain.

Retrait d'un arc

Le retrait d'un arc (i, j) s'opère en lui attribuant un poids excessivement élevé. Dans l'élément (i, j) de la matrice d'adjacence, nous inscrivons dans une valeur supérieure à la somme de tous les poids de la matrice. Notons que, sur certaines instances, cela peut mener à des impasses. C'est pourquoi nous avons doté notre algorithme d'évaluation d'une détection de l'impossibilité à affecter certains sommets pour cause d'arcs manquants (*cf.* section 2.2.4).

3.3 Évaluation

3.3.1 Résolution du sous problème d'affectation

Les algorithmes qui résolvent le problème d'affectation recherchent tous des chemins alternés constitués d'arcs consécutifs et opposés dont la sélection engendre un coût optimal. Pour les algorithmes tels que le nôtre, qui travaillent dans la matrice d'adjacence du graphe, les arcs dont la sélection engendre un coût optimal correspondent aux éléments de coût réduit nul (*cf.* section 2.2.4). Il est donc possible de travailler sur l'efficacité de l'implémentation, en permettant un accès en $O(1)$ aux zéros présents dans une ligne ou une colonne, de manière à éviter toute recherche itérative.

Pour ce faire, une liste doublement chainée est attribuée à chaque ligne et à chaque colonne de la matrice. Chacune recense les indices des zéros de sa ligne ou de sa colonne. Les maillons sont alloués dans une table de $n \times n$, image de la matrice des coûts réduits. Soit une ligne i où un zéro apparaît à l'index j. Le maillon m_{ij} de la table est alors inséré dans la liste chainée de la ligne i, ainsi que dans la liste chainée de la colonne j.

Si un zéro vient à disparaître lors d'une diminution des v_j, il n'est nul besoin de le rechercher dans les listes. Les maillons, grâce à leur rangement dans une table, sont accédés en $O(1)$. Quelques écritures de pointeurs permettent de les insérer dans les listes comme de les en sortir.

Des listes chaînées sont également maintenues pour recenser les colonnes et les lignes repérées, dans l'optique d'accélérer le calcul de nouveaux zéros (*cf.* algorithme 4 page 12).

3. $n^3 - (n-1)^3$, que l'on peut approximer par $\frac{\mathrm{d}n^3}{\mathrm{d}n} = 3n^2$ lorsque $n \to \infty$

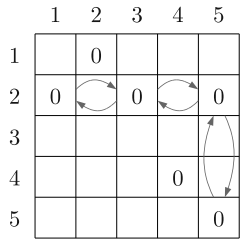

FIGURE 3.7 – Liste doublement chaînée sur deux dimensions

3.3.2 Minimisation du nombre de sous tours

Après résolution du sous problème d'affectation, la matrice des coûts réduits contient souvent des zéros non sélectionnés, grâce auxquels d'autres affectations de coût optimal peuvent souvent se faire. Ces différentes solutions possibles ne produisent pas toutes la même quantité de sous tours. Si nous constatons, par la pratique, que le temps de résolution de l'ATSP est en moyenne indépendant de la solution choisie, nous pouvons tout de même exploiter le fait que le nombre de sous tours correspond au nombre de contraintes du problème originel qui ne sont pas respectées.

Intérêt

En premier lieu, si l'une des affectations possibles produit un unique circuit hamiltonien, il est inutile de séparer le sous problème. Une solution admissible est trouvée et nous évitons l'ouverture d'un nouveau sous espace de recherche.

De plus, au début du chapitre, nous avons proposé d'utiliser le nombre de sous tours que comporte la solution d'un sous problème pour calculer son intérêt. Pour rappel, la borne inférieure fournie par l'évaluation d'un nœud est d'autant plus proche de la solution optimale du problème originel que les contraintes respectées sont nombreuses. Or, lorsque l'on a s sous tours dans la solution d'un sous problème, autant de contraintes de l'ATSP ne sont pas respectées. Par conséquent, pour estimer la qualité des bornes inférieures, nous ne pouvions pas nous satisfaire du nombre arbitraire et souvent pessimiste de sous tours issus de la résolution du problème d'affectation.

Réalisation

Pour élaborer d'autres affectations, nous devons sélectionner de nouveaux zéros dans la matrice des coûts réduits. Cela nous oblige à en désélectionner autant. Nous savons que toute ligne et toute colonne de la matrice possède un zéro sélectionné, de telle sorte que la sélection du premier nouveau zéro nous oblige à en désélectionner 2 ; l'un sur sa ligne, l'autre sur sa colonne. Nous avons donc globalement perdu une sélection. Par exemple, sur la figure 3.8, si l'on sélectionne le zéro en (6,11), nous devons retirer les sélections en (6,7) et en (10,11). De par ce retrait, la colonne 7 n'a plus de zéro sélectionné, tout comme la ligne 10. Nous devons corriger cela par de nouvelles sélections en (i,7) et en (10,j).

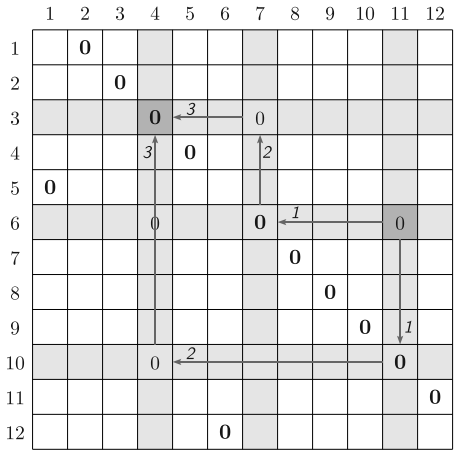

FIGURE 3.8 – Cascade de zéros à sélectionner/désélectionner

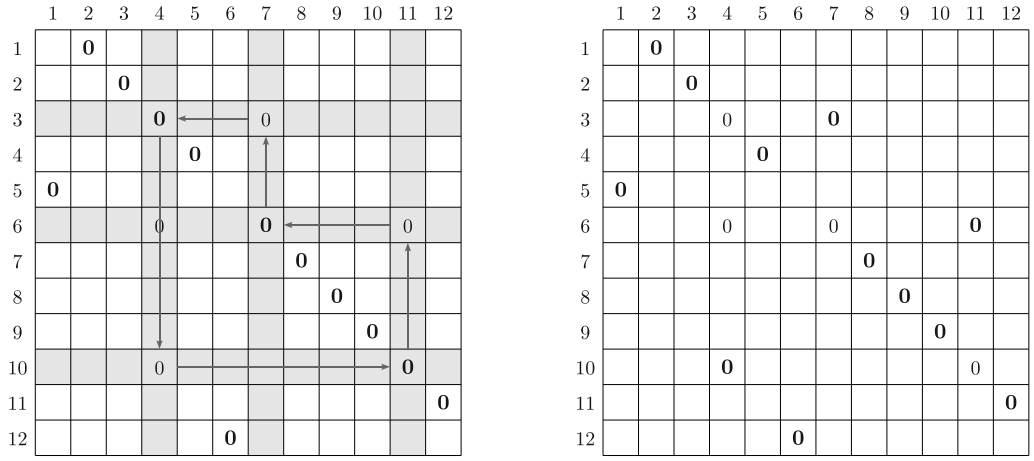

FIGURE 3.9 – Identification puis inversion d'un cycle alterné dans la matrice des coûts réduits

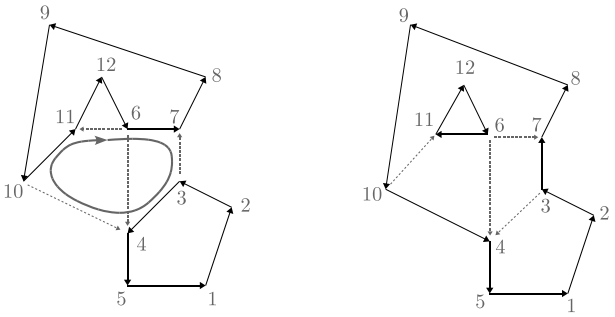

FIGURE 3.10 – Identification puis inversion du cycle alterné dans le graphe correspondant

```
algorithme chercher_cycle
  Compter le nombre de sous tours dans la solution courante.
  si ils sont moins nombreux que dans la meilleure solution trouvée alors
  |   Sauvegarder la solution.
  |   pour chaque colonne j non repérée faire
  |   |   scanner_colonne(j)
  |   |   si le retour de l'appel nous informe qu'au moins un cycle a été trouvé alors
  |   |   |   Quitter la procédure.
  |   |   fin
  |   fin
  fin
fin
```

Algorithme 5 : Recherche du début d'un cycle alterné

```
fonction scanner_colonne (j : entier) : booleen
  i', j' : entiers
  cycle_trouvé : booleen

  cycle_trouvé ← faux
  pour chaque zéro de la colonne j faire
  |   i' ← indice de la ligne où se situe le zéro
  |   si le zéro en (i', j) n'est pas sélectionné et que la ligne i' n'est pas repérée alors
  |   |   j' ← indice de la colonne où se situe le zéro sélectionné de la ligne i'
  |   |   Attribuer un repère à la colonne j pour lui affecter i' comme successeur.
  |   |   Attribuer un repère à la ligne i' pour lui affecter j' comme successeur.
  |   |   si la colonne j' est repérée alors
  |   |   |   cycle_trouvé ← vrai
  |   |   |   chercher_cycle()
  |   |   |   Inverser la sélection sur le cycle passant par (i', j') et défini par les repères.
  |   |   |   chercher_cycle()
  |   |   |   Annuler l'inversion.
  |   |   sinon
  |   |   |   scanner_colonne(j')
  |   |   |   si le retour de l'appel nous informe qu'au moins un cycle a été trouvé alors
  |   |   |   |   cycle_trouvé ← vrai
  |   |   |   fin
  |   |   fin
  |   |   Retirer le repère attribué à la ligne i'.
  |   |   Retirer le repère attribué à la colonne j.
  |   fin
  fin
  retourner cycle_trouvé ?
fin
```

Algorithme 6 : Recherche récursive des cycles alternés

Cependant, la sélection d'un nouveau zéro en (i,7) obligera à désélectionner le zéro qui était sélectionné dans la ligne i. Il en ira de même lors la sélection d'un nouveau zéro en (10,j). Nous entamons ainsi une cascade de modifications dans laquelle il manque toujours un zéro sélectionné, sauf dans un cas : lorsque, des deux côtés de la cascade, nous effectuons une nouvelle sélection qui provoque la désélection d'un même et unique zéro. Sur l'exemple de la figure 3.8, la cascade prend fin de cette manière en (3,4).

En pratique, nous ne progressons pas ainsi. D'une manière tout à fait équivalente mais dont la justification nous semblait moins intuitive pour une première approche, nous recherchons un cycle alterné de zéros sélectionnés et non sélectionnés. Cela est représenté à gauche de la figure 3.9. Le lecteur plus familier avec les graphes en trouvera un parallèle sur la figure 3.10.

La méthode que nous avons conçue pour rechercher les cycles alternés et retenir le nombre minimal de sous tours qui en découle est constituée de deux algorithmes mutuellement réentrants. L'algorithme 5 est le point d'entrée. Il fait appel à l'algorithme 6, chargé de trouver au moins un cycle. Lorsque ce dernier trouve un cycle, il rappelle l'algorithme 5 afin de rechercher d'autres cycles dans la matrice, excepté celui qui vient d'être trouvé.

Chaque cycle trouvé engendre 2 branches dans l'arbre de recherche : soit nous inversons la sélection de zéros sur le cycle, soit nous n'y touchons pas. Dans les deux cas, nous amorçons la recherche d'autres cycles, en dehors de celui qui vient d'être trouvé.

3.4 Optimisation dynamique

3.4.1 Prétraitement de l'instance

Giorgio Carpaneto, Mauro Dell'Amico et Paolo Toth suppriment certains arcs du graphe avant d'entamer la recherche par séparation et évaluation [13]. Pour cela, ils s'appuient sur les faits suivants. La résolution du problème initial relaxé fournit une matrice de coûts réduits. Chaque valeur $\bar{c}_{ij}$ de la matrice renseigne sur l'augmentation du coût de la solution si l'on sélectionne l'arc (i, j). Il existe des arcs dont la sélection engendre un coût si important qu'elle mène de manière certaine à une solution trop mauvaise pour être optimale.

Par l'exécution préliminaire d'une heuristique, ils obtiennent une borne $\hat{z}$ supérieure au coût de la solution optimale de l'ATSP. Suite à cela, ils résolvent le problème relaxé pour obtenir une borne inférieure $\check{z}$. Enfin, en consultant la matrice des coûts réduits obtenue, ils peuvent retirer de l'instance tous les arcs (i, j) tels que $\check{z} + \bar{c}_{ij} > \hat{z}$.

3.4.2 Traitement dynamique

Nous avons choisi de retirer des arcs de l'instance avant d'amorcer le processus de séparation et d'évaluation, comme le proposent les auteurs sus-cités, mais également à tout moment au cours de la résolution. En effet, lorsqu'une solution admissible meilleure que celle qui était connue est trouvée, elle constitue une nouvelle borne supérieure plus basse que la précédente. De ce fait, nous pouvons être en mesure de retirer de nouveaux arcs.

Soit t l'étape où une solution admissible meilleure que celle actuellement connue est trouvée. Soit $\hat{z}(t)$ la valeur de la fonction objectif de cette solution. Nous retirons définitivement de l'instance les arcs (i, j) tels que $\check{z} + \bar{c}_{ij} > \hat{z}(t)$.

Pour que cette opération soit réalisée en un minimum de temps, nous construisons un tas binaire lors de l'évaluation du problème initial pour y placer les différents $\check{z} + \bar{c}_{ij}$. Ensuite, à chaque découverte d'une solution admissible de coût $\hat{z}(t) < \hat{z}(t-1)$, nous extrayons du tas binaire les éléments tels que $\check{z} + \bar{c}_{ij} > \hat{z}(t)$ et supprimons les arcs correspondants de l'instance du problème.

De fait, certains nœuds ouverts peuvent devenir insolubles. Ils seront fermés car nous avons doté notre algorithme d'évaluation d'une détection de l'impossibilité à affecter certains sommets pour cause d'arcs manquants (*cf.* section 2.2.4).

3.5 Allocation mémoire

Au cours de la recherche, des sous problèmes sont ouverts, d'autres fermés, et cela à une cadence élevée. Chaque nœud de l'arbre possède la liste des séparations à effectuer pour générer ses sous problèmes fils. La taille de la liste est liée aux sous tours et est imprévisible. Ainsi, des milliers de blocs de mémoire, de tailles très diverses, sont occupés ou libérés d'une manière désordonnée chaque seconde. Un tel comportement fragmente l'espace utilisateur [4] et oblige le système d'exploitation à effectuer des compactages. Cela est généralement mauvais pour les performances d'un programme et, d'une manière générale, les systèmes d'exploitation n'offrent aucune garantie sur le temps d'obtention d'un bloc mémoire.

Pour cette raison, nous avons conçu une unité d'allocation propre à notre programme. Elle réserve un large espace contigu à l'initialisation, puis y effectue les nombreuses opérations requises par la recherche. Tout à fait classiquement, l'espace est découpé en blocs alloués et en zones libres. Chacun d'entre eux possède à ses extrémités un champ renseignant sur sa taille et son état (alloué ou libre). Pour répondre à une demande d'allocation, nous devons trouver une zone libre suffisamment grande. À cet effet, la règle de choix prise comme base dans tous les systèmes modernes est dite « best fit ». Elle consiste à allouer le bloc demandé par l'utilisateur dans la zone libre la plus petite en mesure de le recevoir.

3.5.1 Organisation de l'espace

Par souci d'efficacité, nous faisons des zones libres les nœuds d'un arbre AVL [12]. La recherche de la plus petite zone d'une taille supérieure ou égale à la demande nécessite alors un temps logarithmique en le nombre de zones libres. La figure 3.11 représente un cas où 3 zones sont libres dans l'espace global et forment un arbre de 3 nœuds.

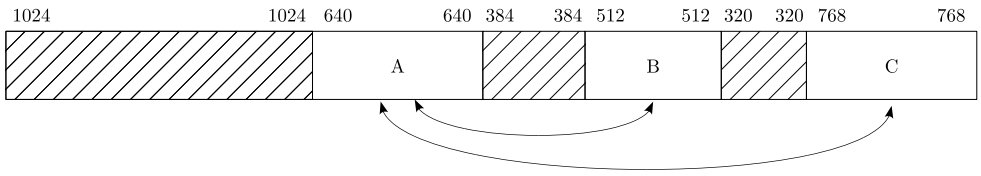

FIGURE 3.11 – Recensement des blocs libres au sein d'un arbre de tri auto équilibré

4. Pour rappel, un fragment est une zone libre dans l'espace mémoire du programme, située entre deux blocs alloués. Un fragment est soit un ancien bloc ayant été libéré, soit le résidu d'une allocation effectuée dans une zone plus large que le bloc alloué.

Les blocs alloués sont souvent entourés d'au moins une zone libre. Lorsque l'utilisateur libère un bloc, nous fusionnons les zones de manière à maximiser les chances de pouvoir répondre aux demandes importantes, et à réduire la profondeur de l'arbre de tri. Cela est illustré sur la figure 3.12.

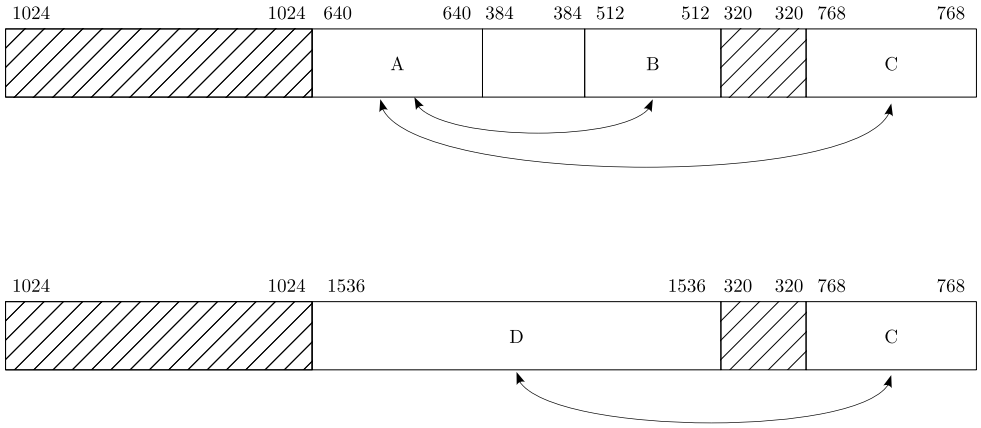

FIGURE 3.12 – Fusion provoquée par la libération du bloc central

3.5.2 Diminution de la fragmentation

Enfin, nous avons souhaité limiter la fragmentation de l'espace global. La règle du *best-fit* et la procédure de recherche dans l'arbre de tri imposent les zones d'allocation. Nous devons donc influencer la fragmentation de l'espace global à l'échelle des zones. Nous proposons une méthode imparfaite mais simple pour y parvenir. Des statistiques glissantes sur les dernières opérations effectuées par l'utilisateur sont maintenues, et ce sur deux critères. Plus précisément, la taille et la durée de vie de tout bloc libéré par l'utilisateur fait l'objet d'un enregistrement dans une file dédiée. Simultanément à l'ajout de ces données dans leur file respective (et au retrait de la dernière valeur de la file), nous mettons à jour deux variables propres à chacune : l'une cumule les valeurs, l'autre cumule leurs carrés. Ainsi, à tout instant, nous sommes en mesure de calculer la moyenne $\bar{x}$ et la variance v_x de chaque critère avec très peu d'opérations.

Nous supposons que la taille et la durée de vie des blocs suivent une loi normale, ce qui est discutable mais demande peu de calculs. De plus, toujours dans l'optique de limiter la quantité de calculs, les critères de taille et de durée de vie sont supposés indépendants. De ces considérations découle une expression simple de la probabilité pour qu'un bloc soit libéré, en fonction de sa taille et de son âge :

$$p(t,d) = p(t)\,p(d)\,,$$

avec

$$p(x) = \frac{1}{\sqrt{2\pi v_x}}\, e^{-\frac{(x-\bar{x})^2}{2v_x}}\,.$$

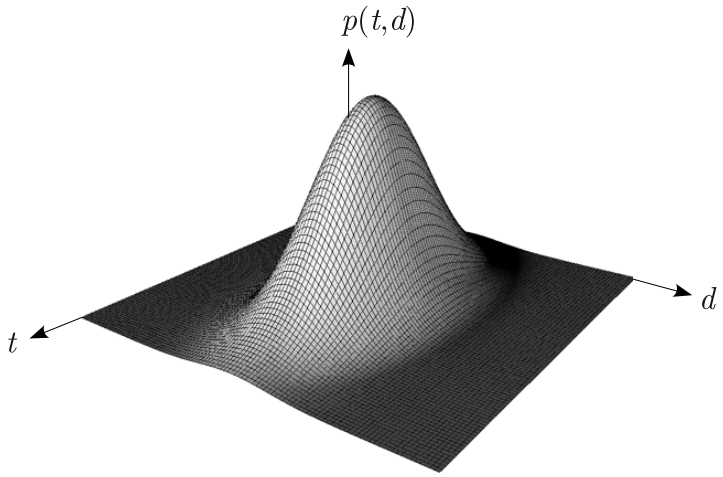

FIGURE 3.13 – Probabilité de libération d'un bloc en fonction de sa taille t et de son âge d

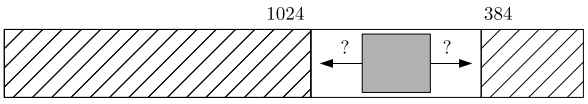

FIGURE 3.14 – Où vaut-il mieux placer l'espace résiduel ?

Nous construisons donc une courbe de Gauss surfacique, représentée sur la figure 3.13 et dont la lecture permet d'estimer une réponse à la question suivante. Sachant que, sauf cas rares, l'allocation d'un bloc génère un espace résiduel, où doit-on le placer ? La réponse s'appuie sur la taille des blocs alloués qui entourent la zone, car nous choisissons de placer l'espace résiduel du côté du bloc ayant le plus de chances d'être libéré [5]. Cela est illustré sur la figure 3.14. L'idée est de fusionner au plus vite cet espace résiduel, souvent trop petit pour être utile, avec une zone plus grande. La probabilité pour que le bloc de gauche soit libéré si l'un des deux devait l'être, s'exprime par

$$\frac{p(t_1, d_1)}{p(t_1, d_1) + p(t_2, d_2)} .$$

Cette expression brute serait lente à calculer, car elle renferme 6 racines carrées et 6 exponentielles. Nous divisons le numérateur et le dénominateur par $p(t_1, d_1)$ et obtenons

$$\frac{1}{1 + \frac{p(t_2)p(d_2)}{p(t_1)p(d_1)}} .$$

Les produits et le quotient autorisent alors une simplification telle qu'il ne subsiste qu'une exponentielle :

$$\frac{1}{1 + \exp\left(\frac{(t_1-\bar{t})^2-(t_2-\bar{t})^2}{2v_t} + \frac{(d_1-\bar{d})^2-(d_2-\bar{d})^2}{2v_d}\right)} .$$

Pour ne pas pénaliser les performances de notre unité d'allocation, les exponentielles sont calculées en O(1) à l'aide d'une approximation suivie d'une correction. L'approximation consiste en la lecture grossière du résultat dans une table de quelques exponentielles précalculées. À l'indice k, la table contient e^{x_k}, avec $x_k = k\,p$, où p est le pas entre deux x_k consécutifs. Suite à cette lecture, nous améliorons la précision du résultat par un développement limité à l'ordre 1 :

$$e^x \approx e^{x_k}(1 + x - x_k) .$$

Notre démarche n'a d'intérêt que si le résultat produit est suffisamment précis. Pour lire la table précalculée, nous arrondissons x au x_k le plus proche, si bien que $|x - x_k| \leq \frac{p}{2}$. L'erreur relative, qui s'exprime par

$$E = \frac{e^x - e^{x_k}(1 + x - x_k)}{e^x} ,$$

vaut donc, au pire,

$$E_{max} = \frac{e^x - e^{x-\frac{p}{2}}\left(1 + \frac{p}{2}\right)}{e^x} ,$$

et se simplifie par

$$E_{max} = 1 - \left(1 + \frac{p}{2}\right) e^{-\frac{p}{2}} .$$

Avec un pas p de 0.5, notre démarche produit résultat précis à $\pm 2.6\%$ près, ce qui est suffisant pour l'utilisation que nous en avons.

5. Pour être exact, nous pondérons la taille des blocs par leur probabilité d'être libérés afin de considérer des espérances.

Chapitre 4

Performances

L'ensemble des mesures qui suivent sont prises sur une machine animée par un processeur Core2Duo d'Intel, cadencé à 3 GHz et doté de 4 Mo de mémoire cache de premier niveau. Un seul de ses cœurs est utilisé. La mémoire vive disponible est d'un peu moins de 2 Go.

4.1 Allocation mémoire

Nous avons testé notre unité d'allocation dans un espace global de 64 Mo, dans lequel nous avons effectué 10 séries d'opérations aléatoires, d'abord sans l'heuristique de diminution de la fragmentation, puis avec elle. Chaque série d'opérations consiste en 210 000 demandes d'allocation de blocs de tailles aléatoires, entrecoupées de 110 000 libérations de blocs choisis aléatoirement. Les blocs alloués ont une taille comprise entre 64 et 3500 octets, avec une probabilité plus forte pour les petits blocs dans la mesure où ils sont les principaux responsables de la fragmentation de la mémoire. Les demandes d'allocation pour des blocs d'une taille inférieure à 500 octets ont une probabilité de $\frac{1}{2}$, les demandes pour des blocs d'une taille inférieure à 1100 octets ont une probabilité de $\frac{1}{3}$, et les demandes pour des blocs d'une taille inférieure à 3500 octets ont une probabilité de $\frac{1}{6}$.

La table 4.1 synthétise les performances de notre allocateur. En haut sont rapportées les mesures prises sans l'heuristique de diminution de la fragmentation. En bas sont rapportées les mesures prises en l'ayant activée.

En moyenne et au total, les 320 000 opérations de chaque série durent 151 ms, ce qui est une performance excellente pour nos besoins. En effet, au cours d'une résolution, nous n'effectuerons pas 2 millions d'ouvertures et fermetures de nœuds par seconde. Concernant l'heuristique, qu'elle soit activée ou pas, le nombre de fragments est sensiblement le même. Nous notons tout de même qu'elle apporte une légère amélioration en moyenne, de l'ordre de 3%, tout en ne ralentissant pas significativement l'unité d'allocation. Nous l'avons donc activée dans notre programme. Dans tous les cas, au terme des 320 000 opérations aléatoires, l'espace global comporte environ 6 000 fragments pour 100 000 blocs alloués, si bien que 94% des blocs sont contigus. Cela est très satisfaisant, du moins pour notre application.

Série	A	B	C	D	E	F	G	H	I	J	moy
Fragments	5981	5869	6201	5688	6059	6213	5932	5581	6529	5916	5997
Temps (ms)	144	143	144	145	144	145	143	145	142	144	144
Fragments	5578	5874	6086	5757	5664	5969	5957	5384	6214	5725	5821
Temps (ms)	151	152	151	151	153	152	152	149	153	150	151

TABLE 4.1 – Performances de l'allocateur mémoire. En haut, sans heuristique. En bas, avec heuristique.

4.2 Évaluation

4.2.1 Résolution des sous problèmes

Le graphique 4.1 synthétise les performances de notre unité de résolution du problème d'affectation lorsqu'elle traite des instances aléatoires. Il a été construit par 9 points de mesure, correspondant au temps de résolution moyen pour 9 tailles d'instances. En effet, chaque point provient de la moyenne du temps de résolution de 10 instances aléatoires de même taille.

La courbe présente sur le graphique a été tracée suite à une régression polynomiale. D'après son résultat, la complexité pratique de notre algorithme serait en $O(n^{2.58})$. Ce résultat, certes approximatif, est sous la borne calculée par James MUNKRES (*cf.* section 2.3) et s'explique par le caractère pessimiste des calculs de complexité.

4.2.2 Minimisation du nombre de sous tours

La table 4.2 synthétise les performances de notre unité de recherche du nombre de sous tours minimal. Les mesures sont prises sur 10 instances. Les 6 premières sont aléatoires et de taille 1000. Les 4 suivantes proviennent de la TSPlib [1].

En haut, la recherche du nombre de sous tours minimal est désactivée et seule la résolution du problème d'affectation est effectuée. En bas, l'affectation produisant le minimum de sous tours est recherchée.

Sur les instances aléatoires, nous observons que le nombre de sous tours est divisé par 2. Sur les instances plus difficiles de la TSPlib, nous obtenons parfois le même résultat, parfois moindre, et quelques instances difficiles semblent ne pas comporter moins de sous tours que ce que propose la première solution du problème d'affectation. Pour les grandes instances, le temps pris par la minimisation du nombre de sous tours est négligeable devant le temps de résolution. Pour les instances de la TSPlib, les durées sont de l'ordre de la précision de notre mesure, si bien qu'il est impossible de se prononcer.

D'un point de vue général, nous constatons qu'il existe bien plusieurs solutions à un problème d'affectation et que certaines d'entre elles respectent un plus grand nombre de contraintes de l'ATSP qu'il n'y paraitrait si l'on retenait la première proposée (*cf.* section 3.3.2).

1. bibliothèque d'instances de référence

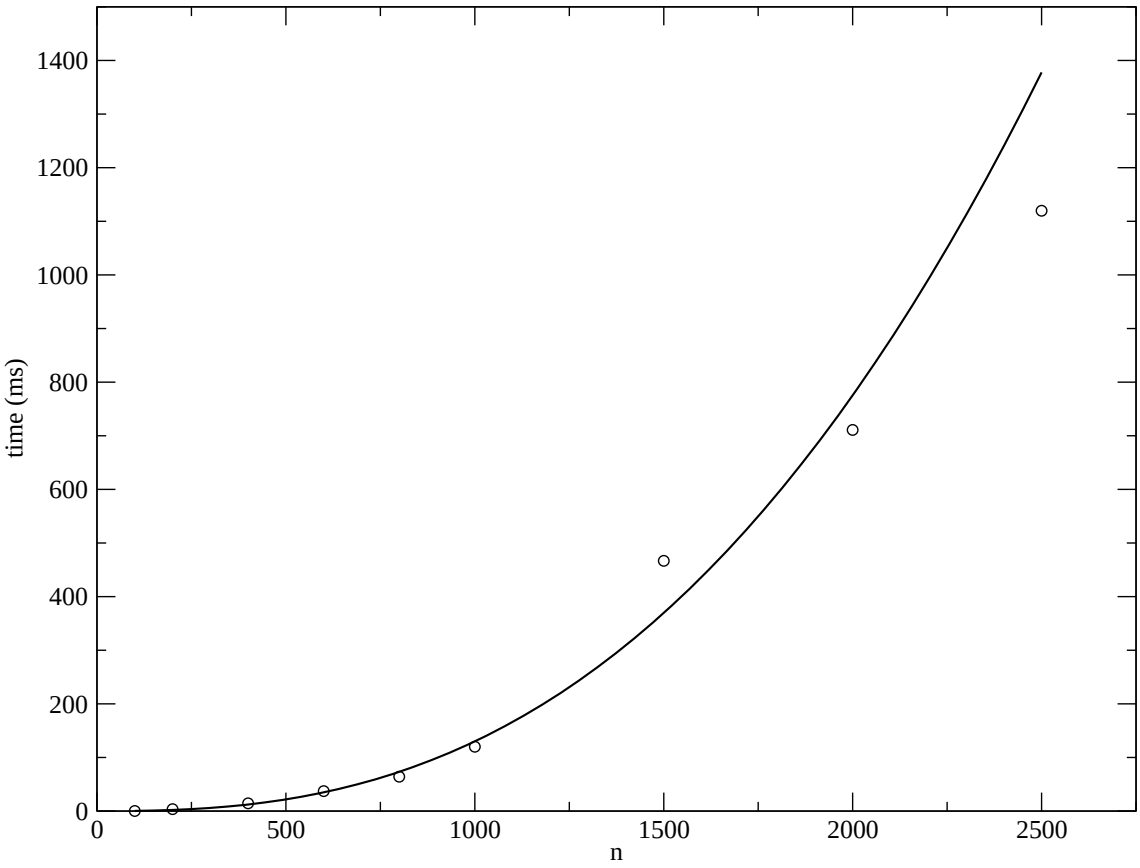

FIGURE 4.1 – Temps de résolution du problème d'affectation

Instance	M1000	N1000	O1000	P1000	Q1000	R1000	P43	FTV170	RBG323	RBG443
Sous tours	10	8	6	5	8	14	43	19	323	443
Temps (ms)	110	152	115	137	127	105	0	2	1	1
Sous tours	8	4	4	2	3	7	28	13	323	250
Temps (ms)	111	152	118	153	224	116	0	3	1	2

TABLE 4.2 – Résolution du problème d'affection. En haut, sans recherche du nombre minimal de sous tours. En bas, avec.

4.3 Résolution complète

Pour les tests globaux de notre solveur, nous avons choisi de procéder à la résolution de 5 séries de 10 instances aléatoires, chaque série étant de taille 100, 300, 600, 800 et 1000. Les arcs ont un poids aléatoire compris dans $[1\,;10^3]$. De cette manière, nous reproduisons les caractéristiques des instances de classe *a1* évaluées par Giorgio Carpaneto, Mauro Dell'Amico et Paolo Toth en 1995 [13].

La première colonne des tableaux ci-après indique la taille des instances de la série considérée. La seconde colonne indique le temps de résolution moyen des 10 instances de la série. La colonne suivante rapporte la durée de résolution la plus longue constatée. La $4^{\text{ème}}$ colonne contient le nombre moyen de sous problèmes d'affectation résolus au cours des recherches. L'avant dernière colonne indique la profondeur moyenne de la solution optimale dans l'arbre de recherche. Enfin, en dernier, nous rapportons la densité moyenne de la matrice des coûts réduits du problème au terme de la résolution [2].

Les temps de résolution sont incomparables. Les ordinateurs utilisés par les auteurs de la table 4.4 étaient équipés de processeurs Intel 486 à 33 MHz, tandis que nous avons pris nos mesures sur un ordinateur d'une puissance très supérieure. Nous pouvons tout de même soupçonner notre solveur d'être lent face au leur, car le ratio des fréquences des processeurs est très supérieur au ratio des temps mesurés. Nous imputons cela à notre algorithme d'évaluation en $O(n^3)$, nécessairement plus lent que celui mis en œuvre par les auteurs sus-cités (*cf.* section 3.2).

Cependant, au-delà du critère temporel, il semble que nous puissions constater l'effet des optimisations que nous avons proposées au chapitre 3 et intégrées à notre solveur. Sur les 3 derniers critères de comparaison, notre solveur semble nettement meilleur que celui pris pour référence. Notamment, notre optimisation dynamique retire des instances de 5 à 30 fois plus d'arcs que le prétraitement de Carpaneto, Dell'Amico et Toth (en dernière colonne). De même, nous trouvons les solutions optimales à une profondeur inférieure, et le nombre de sous problèmes évalués est sensiblement inférieur au leur (sauf pour un cas). Notre retrait d'arcs dynamique et notre règle de choix particulière, s'appuyant sur la recherche du nombre de sous tours minimal, semblent bénéfiques.

2. Dans le tableau 4.4, cette valeur est le quotient du nombre d'arcs restants dans l'instance suite au prétraitement par le nombre d'arcs présents dans le problème initial. Dans le tableau 4.3, cette valeur est le quotient du nombre d'arcs restants suite à notre optimisation dynamique par le nombre d'arcs présents dans le problème initial.

n	Temps moy	Temps max	AP	Prof	Densité de M
100	0.021	0.117	20.1	2.5	0.0343
300	0.327	0.796	46.3	3.5	0.0106
600	0.316	1.040	15.2	2.0	0.0044
800	1.938	8.490	43.5	4.1	0.0041
1000	2.764	5.418	39.7	5.0	0.0037

TABLE 4.3 – Performances de notre solveur

n	Temps moy	Temps max	AP	Prof	Densité de M
100	0.420	0.710	30.2	4.7	0.1550
300	2.080	3.900	27.8	4.5	0.0780
600	19.130	45.160	82.9	6.3	0.1280
800	23.040	42.470	66.9	5.4	0.0640
1000	39.100	121.200	55.4	6.3	0.1050

TABLE 4.4 – Performances du solveur de Carpaneto, Dell'Amico et Toth

Chapitre 5

Conclusion

Si nous obtenons des temps de résolution probablement moins bons que ceux obtenus par les spécialistes du domaine, le faible nombre de sous problèmes que nous évaluons au cours de la recherche et la faible densité de notre matrice d'adjacence sont encourageants. Afin d'exprimer au maximum le potentiel de notre solveur, il faudrait attacher à chaque feuille de l'arbre de recherche la matrice des coûts réduits qui lui est propre, et évaluer les sous problèmes par l'algorithme quadratique utilisé par la plupart des autres solveurs.

Quoi qu'il en soit, le problème du voyageur de commerce est classé *NP Complet* [2], si bien qu'il existe des instances insolubles, trop longues à résoudre à l'échelle d'une vie humaine ou même de l'âge de l'univers. Elles le demeureront tant que n'auront pas été découverts d'hypothétiques algorithmes polynomiaux en mesure de résoudre les problèmes de cette classe sur nos processeurs actuels, ou que le futur ne nous aura pas apporté des machines non déterministes. Sur ces dernières, des algorithmes d'une efficacité extrême existent. On peut citer, à titre d'exemple, l'algorithme proposé par Leonard Adleman en 1994 pour résoudre un TSP à 7 sommets à l'aide de brins d'ADN [14]. Enfin, les processeurs quantiques accéléreraient également la résolution des problèmes de la classe *NP*. Un tel processeur énumérerait la totalité des cycles hamiltoniens d'un graphe en un temps égal à la racine carrée du temps requis par un processeur classique [15].

Bibliographie

[1] Gilbert Laporte : « The traveling salesman problem : an overview of exact and approximate algorithms ». *European Journal of Operational Research* vol 59 (1992).

[2] Michael R. Garey, David S. Johnson : *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co (1979).

[3] Ailsa H. Land, Alison G. Doig : « An automatic method for solving discrete programming problems ». *Econometrica* vol 28 (1960).

[4] Jacques Teghem : *Programmation linéaire*. Éditions de l'Université de Bruxelles (2003).

[5] Giorgio Carpaneto, Paolo Toth : « Some new branching and bounding criteria for the asymmetric travelling salesman problem ». *Management Science* vol 26 (1980).

[6] Harold W. Kuhn : « The Hungarian method for the assignment problem ». *Naval Research Logistics Quarterly* vol 2 (1955).

[7] Carl Gustav Jacob Jacobi : « De investigando ordine systematis aequationum differentialum vulgarium cujuscunque ». *Borchardt Journal für die reine und angewandte Mathematik* vol 64 (1865).

[8] Harold W. Kuhn : « The Hungarian method for the assignment problem and how Jacobi beat me by 100 years ». Concordia University, Montreal (2006).

[9] James Munkres : « Algorithms for the assignment and transportation problems ». *Journal of the society for industrial and applied mathematics* vol 5 (1957).

[10] Karl-Heinz Borgwardt : « The average number of pivot steps required by the simplex-method is polynomial ». *Operations Research* vol 26 (1982).

[11] Narendra Karmarkar : « A new polynomial time algorithm for linear programming ». *Combinatorica* vol 4 (1984).

[12] Georgii M. Adelson-Velskii, Evgenii M. Landis : « An algorithm for the organization of information ». *Proceedings of the USSR Academy of Sciences* vol 146 (1962).

[13] Giorgio Carpaneto, Mauro Dell'Amico, Paolo Toth : « Exact solution of large scale, asymmetric traveling salesman problems ». *ACM Transactions on Mathematical Software* vol 21 (1995).

[14] Leonard Max Adleman : « Molecular computation of solutions to combinatorial problems ». *Science* vol 266 (1994).

[15] Michael A. Nielsen, Isaac L. Chuang : *Quantum computation and quantum information*. Cambridge University Press (2000).