RAIRO. RECHERCHE OPÉRATIONNELLE

MARTIN DESROCHERS
FRANÇOIS SOUMIS

Implantation et complexité des techniques de programmation dynamique dans les méthodes de confection de tournées et d'horaires

RAIRO. Recherche opérationnelle, tome 25, n° 3 (1991), p. 291-310

<http://www.numdam.org/item?id=RO_1991__25_3_291_0>

Recherche opérationnelle/Operations Research
(vol. 25, n° 3, 1991, p. 291 à 310)

IMPLANTATION ET COMPLEXITÉ DES TECHNIQUES DE PROGRAMMATION DYNAMIQUE DANS LES MÉTHODES DE CONFECTION DE TOURNÉES ET D'HORAIRES (*)

par Martin DESROCHERS ([1]) et François SOUMIS ([1])

Résumé. – *Dans les approches par programmation dynamique, nous devons résoudre les équations de récurrence pour tous les états. La solution optimale est un ensemble de chemins entre états initiaux et états finaux. Nous étudions une classe de problèmes de programmation dynamique discrète : les problèmes de plus courts chemins avec plusieurs contraintes supplémentaires. Nous ferons ici la comparaison entre deux classes d'algorithmes résolvant ces problèmes : les algorithmes de « reaching » et de « pulling ». Nous décrirons une implantation de l'algorithme de « reaching » et deux implantations de l'algorithme de « pulling ». Nous comparerons leur complexité et présenterons les résultats de tests numériques sur des problèmes de grande taille. La principale conclusion de ces tests est que le temps d'exécution d'une des implantations du « pulling » ne croît pas avec le nombre d'états mais est plutôt proportionnel au nombre d'états associés à des chemins admissibles.*

Mots clés : Programmation dynamique; complexité; chemin le plus court avec contraintes supplémentaires.

Abstract. – *In all dynamic programming problems, the aim is to solve the recurrence equations for all states. The optimal is a set of paths going from the initial states to the final states. We study a class of discrete dynamic programming problems: the shortest path problems with several constraints. We will compare two classes of algorithms; "reaching" algorithms and "pulling" algorithms. We will describe one "reaching" implementation and two "pulling" ones. We will compare their computational complexity and the results of an empirical study on large scale problems. The main conclusion of this study is that the computation times of the second implementation of the "pulling" algorithm does not grow with the number of states but is rather proportional to the number of feasible paths in the solution.*

Keywords : Dynamic programming; computational complexity; shortest path problem with resource constraints.

(*) Reçu novembre 1986.
([1]) G.E.R.A.D. et École Polytechnique de Montréal, 5255, avenue Decelles, Montréal, Canada, H3T 1V6.

Recherche opérationnelle/Operations Research, 0399-0559/91/03 291 20/$ 4.00

INTRODUCTION

Nous considérons le problème de plus court chemin avec constraintes (PCCC) dans un graphe. Ce problème apparaît comme sous-problème dans les méthodes de décomposition pour la fabrication de tournées et d'horaires de véhicules et de personnel. Le sous-problème correspond à la fabrication d'une tournée admissible.

En général, ces problèmes peuvent se ramener à des problèmes de plus court chemin dans un graphe éclaté en divisant chaque nœud original en plusieurs états. Cette approche est souvent inintéressante à cause de la trop grande taille du graphe éclaté. Nous développons dans cet article des techniques de programmation dynamique permettant de travailler sur le graphe en ne conservant à chaque nœud que les états nécessaires. Ces techniques pourront être particulièrement intéressantes lors de la résolution de problèmes de confection de tournées et d'horaires.

Le PCCC est défini dans la première section. La seconde contient une brève revue de la littérature. Un algorithme de « reaching » est décrit dans la troisième section et deux variantes du « pulling » introduites dans la quatrième. Des résultats numériques pour le cas avec une contrainte supplémentaire sont présentés dans la section 5. La section 6 contient des résultats numériques pour le cas avec plusieurs contraintes.

1. LE PROBLÈME

Soit le réseau $G=(N, A)$ ou A est l'ensemble des arcs et N est l'ensemble de nœuds dont une source p et un puits q. Les variables x_{ij} correspondent aux flots circulant sur l'arc (i, j). Un coût $c_{ij} \in R$ qui peut être positif ou négatif, est associé à chaque arc $(i, j) \in A$ et permet de définir le problème de plus court chemin [(1)-(3)]. La formulation présentée correspond à trouver le chemin parcouru entre la source et le puits par une unité de flot. Il est à noter que la solution de ce problème de plus court chemin n'est pas nécessairement un chemin élémentaire.

$$\text{Minimiser } c \cdot x = \sum_{(i, j) \in A} c_{ij} x_{ij} \tag{1}$$

sujet à

$$\sum_{j \in N} x_{ij} - \sum_{j \in N} x_{ji} = \begin{cases} +1, & i = p \\ 0, & i \in N,\ i \neq p,\ i \neq q \\ -1, & i = q \end{cases} \tag{2}$$

$$x_{ij} \geqq 0, \qquad \forall (i, j) \in A. \tag{3}$$

Dans notre problème, nous avons des contraintes sur l'utilisation de L ressources. A chaque arc (i, j) et à chaque ressource l, est associée $d_{ij}^l \in N$, la quantité de la ressource l utilisée par l'arc (i,j). Lorsque les nœuds sont des tâches, le coût et les ressources utilisées par un nœud i sont ajoutés à ceux des arcs sortant (i, j). Soit T_j^l, la quantité de la ressource l utilisée en poursuivant un chemin depuis la source jusqu'au nœud j; notre problème consiste à trouver le chemin de coût minimum entre p et q tel que T_j^l respecte la fenêtre d'admissibilité $[a_j^l, b_j^l]$ pour tous les nœuds du chemin et pour toutes les ressources $l = 1, \ldots, L$. Nous considérons la problème où il est possible d'atteindre un nœud en utilisant moins d'une ressource l que le minimum de la fenêtre d'admissibilité de cette ressource. En gaspillant la ressource l, il est alors possible de débuter la visite de ce nœud au minimum de la fenêtre d'admissibilité de cette ressource. Le problème du plus court chemin avec fenêtres de temps [7] en est un exemple; il est possible d'arriver à un nœud avec le début de la fenêtre de temps et d'attendre, c'est-à-dire gaspiller du temps, avant de débuter la visite du nœud au début de la fenêtre de temps.

Pour obtenir le problème de plus court chemin avec contraintes (PCCC), nous devons ajouter au problème de plus court chemin [(1)-(3)] les contraintes suivantes :

$$x_{ij} > 0 \quad \Rightarrow \quad T_i^l + d_{ij}^l \leqq T_j^l \qquad (i,j) \in A,\ l = 1, \ldots, L \tag{4}$$

$$a_i^l \leqq T_i^l \leqq b_i^l \qquad i \in N,\ l = 1, \ldots, L \tag{5}$$

où x_{ij} représente le flot sur l'arc (i, j) et T_j^l, la quantité de la l-ième ressource utilisée pour parvenir au nœud i.

Si le gaspillage de ressource n'est pas permis, il faut remplacer l'inégalité de l'équation (4) par une égalité. Le problème résultant est plus contraint et sa solution optimale peut posséder un coût plus élevé. Ce modèle correspond à des problèmes de programmation dynamique bien connus (*voir* Jaffe [11]). La complexité théorique en pire cas des deux problèmes (avec ou sans gaspillage) est équivalente pour certaines implantations. Toutefois il est expérimentalement possible de constater que la résolution d'un problème sans gaspillage est beaucoup plus coûteuse que celle d'un problème avec gaspillage.

Nous considérons que tous les arcs non admissibles sont exclus de l'ensemble A. Un arc (i, j) est non admissible s'il ne permet pas de passer du nœud i au nœud j en respectant les fenêtres d'admissibilité en i et j. Tous les arcs $(i, j) \in A$ respectent donc la condition suivante pour toutes les ressources :

$$a_j^l + d_{ij}^l \leqq b_j^l, \qquad l = 1, \ldots, L \tag{6}$$

Les contraintes supplémentaires (4)-(5) peuvent aussi prendre la forme :

$$\sum_{(i,j) \in A} d_{ij}^l x_{ij} \leqq b_q^l, \qquad l = 1, \ldots, L \tag{7}$$

Aneja *et al.* [1], pour résoudre des problèmes exprimés sous la forme [(1)-(3)]-(7), les transforment en problèmes équivalents exprimés sous la forme (1-5). Toutefois, la transformation inverse n'est pas possible, la forme [(1)-(5)] comportant des contraintes à chaque nœud tandis que la forme [(1)-(3)]-(7) qui ne comporte que des contraintes sur le dernier nœud du chemin (puits).

2. REVUE DE LA LITTÉRATURE

Bien que le problème du plus court chemin avec contraintes supplémentaires et le problème voisin du plus court chemin multicritère, aient fait l'objet de nombreux articles, nous ne discuterons ici que de ceux utilisant notre approche : la programmation dynamique. Pour une revue de littérature plus complète, *voir* Desrochers [3] ou Desrochers et Soumis [4].

Introduisons d'abord une définition des états commune à la plupart des approches par programmation dynamique. A chaque chemin X_{pj} de l'origine à un nœud j, est associé un état correspondant aux L quantités de ressources utilisées T_j^l. Ces états sont représentées par $T_j^{*k} = (T_j^1, T_j^{2k}, \ldots, T_j^{Lk})$ et dénotent les caractéristiques du k-ième chemin X_{pj}^k. Une étiquette (état, coût) est associée à chaque chemin. Le coût est noté C_j^k. Les transitions entre deux états existent si le second état peut être obtenu en prolongeant d'un arc le chemin associé au premier état. Un chemin T_{pj}^k et l'état associé T_j^{*k} sont admissibles si $a_j^l \leqq T_j^{lk} \leqq b_j^l$ pour toutes les ressources ($l = 1, \ldots, L$). Dans le cas des problèmes de plus court chemin multicritère, la définition des états est la même mais l'algorithme optimise par rapport à toutes les composantes de l'état *i. e.* les ressources et le coût. Aucune des composantes n'a priorité.

Dominance et principe d'optimalité

Avant de débuter la définition des deux algorithmes, nous devons définit deux notions : la dominance, introduite par Joksch [12], l'ordre lexicographique et les équations de récurrence du PCCC.

DÉFINITION 1 : *Soit* X^1 et X^2 deux chemins distincts de p à j et leurs étiquettes associées (T_j^{*1}, C_j^1) et (T_j^{*2}, C_j^2), X^1 domine X^2 ou

$$(T_j^1, C_j^1) < (T_j^2, C_j^2)$$

si et seulement si $C_j^1 \leqq C_j^2$, $T_j^{l1} \leqq T_j^{l2}$ pour tout

$$l = 1, \ldots, L \qquad \text{et} \qquad (T_j^{*1}, C_j^1) \neq (T_j^{*2}, C_j^2).$$

La relation de dominance nous permet de conclure qu'un chemin X_{pj} et son étiquette associée (T_j^*, C_j) sont non dominés (efficaces) si et seulement si X_{pj} est le chemin de coût minimal arrivant au nœud j avec des valeurs inférieures ou égales à $T_j^l (l = 1, \ldots, L)$.

DÉFINITION 2 : Soit X^1 et X^2 deux chemins distincts de p à j et leurs étiquettes associées (T_j^{*1}, C_j^1) et (T_j^{*2}, C_j^2), (T_j^1, C_j^1) est lexicographiquement plus petit que (T_j^2, C_j^2) si et seulement

- $T_j^{l,1} < T_j^{1,2}$ ou
- si $T_j^{l1} = T_j^{l2}$ pour tout $l = 1, \ldots, k (k < l)$ et $T_j^{l+1,1} < T_j^{l+1,2}$ ou finalement
- si $T_j^{*1} = T_j^{*2}$ et $C_j^1 \leqq C_j^2$.

L'ordre lexicographique définit un ordre total sur l'espace des états et permet de définir un plus petit état (état lexicographiquement minimum).

Nous utilisons le principe d'optimalité de Bellman (*voir* Denardo [2]), c'est-à-dire que tout sous-chemin d'un chemin efficace doit lui aussi être efficace, pour définir les équations de récurrence suivantes qui caractérisent les chemins efficaces de la source à tous les nœuds pour le PCCC :

Pour tout $j \in N$ et T_j^* admissible :

$$C(j, T_j^*) = \min \{ C(i, T_j) + c_{ij} \mid T_i^l + d_{ij}^l \leqq T_j^l, (i,j) \in A, l = 1, \ldots, L \} \quad (8)$$

$i \in N$, T_j^* état admissible

$$C(p, 0) = 0 \quad (9)$$

L'application du principe d'optimalité permet d'accélérer le calcul des équations de récurrence [(8)-(9)] en évitant de prolonger des chemins X_{pi}^k dominés et de produire ainsi de nouveaux chemins X_{pj}^k dominés.

La programmation dynamique permet d'obtenir une valeur de l'étiquette associée à chaque état, qui satisfait les relations de récurrence pour toutes les transitions. Des variantes sont possibles en fonction des choix suivants. Le sens de l'ajustement des étiquettes pour satisfaire la relation peut se faire soit dans le *sens des transitions* en ajustant l'étiquette à la fin de la transition en conservant fixée l'étiquette du début ou soit dans le *sens inverse des transitions*. Le groupement du traitement des transitions peut se faire en traitant à la suite soit les transitions *incidentes vers l'extérieur* à l'état, soit celles *incidentes vers l'intérieur* à un état.

Quatre variantes sont possibles. La programmation dynamique vers l'avant (« reaching ») est la variante utilisant l'ajustement des étiquettes dans le sens des transitions et regroupant le traitement des transitions incidentes vers l'extérieur à chaque état. La programmation dynamique vers l'arrière (« pulling ») est la variante ajustant les étiquettes dans le sens inverse des transitions en groupant les transitions incidentes vers l'intérieur. Cette deuxième variante est symétrique à la première; le traitement vers l'arrière équivaut à appliquer le traitement vers l'avant après avoir inversé le sens des transitions. Deux autres variantes symétriques sont possibles. Dans cet article, nous comparerons le « reaching » et le « pulling » qui consiste à ajuster les étiquettes dans le sens des transitions en regroupant le traitement des transitions incidentes intérieurement à chaque état.

Le « reaching »

Il existe deux méthodes de ce type. La première méthode, basée sur la généralisation de l'algorithme de Ford-Bellman [12, 17, 9, 7, 11], fonctionne par correction des étiquettes. La seconde méthode, basée sur la généralisation de l'algorithme de Dijkstra [15, 10, 1, 3], fonctionne par marquage permanent des étiquettes. L'utilisation d'un algorithme de type Dijkstra est possible seulement quand il existe un ordre de traitement des étiquettes permettant de traiter une seule fois chaque étiquette. L'algorithme de Dijkstra peut alors être considéré comme une amélioration de l'algorithme de Ford-Belman [15]. L'étape de base dans les deux algorithmes, le « reaching », consiste, à partir d'une étiquette associée à un chemin X_{pi}, à obtenir les étiquettes correspondant aux chemins admissibles X_{pj} prolongeant le chemin X_{pi} par tous les arcs $(i, j) \in A$. Ces algorithmes créent pour chaque nœud i de nouvelles étiquettes à plusieurs nœuds j; ainsi des étiquettes sont créées à plusieurs reprises à chaque nœud j si le nœud j possède plusieurs prédécesseurs. Il faut donc effectuer plusieurs mises à jour des étiquettes en chaque nœud j. Cette mise à jour est coûteuse surtout s'il y a plusieurs contraintes supplémentaires.

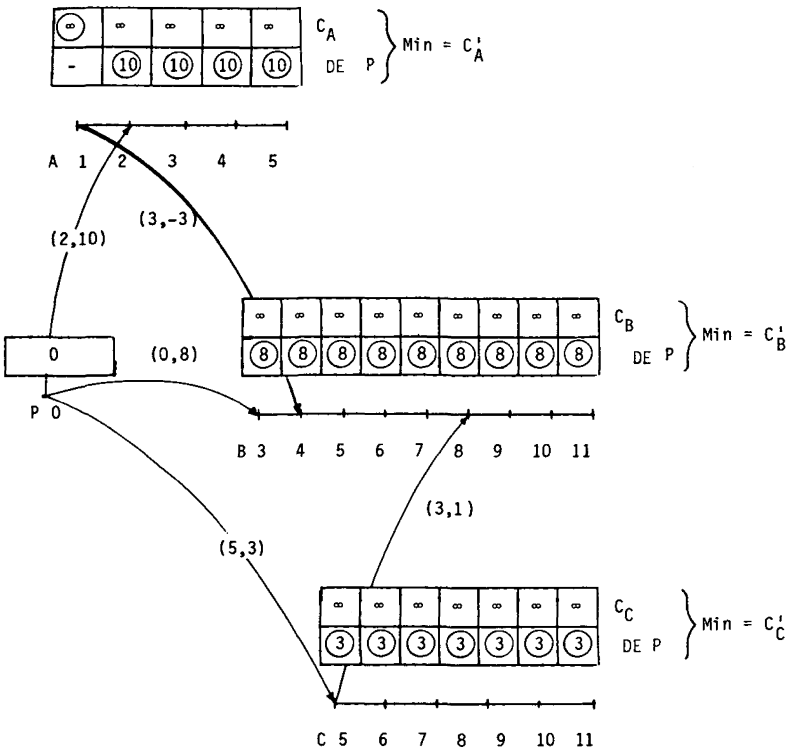

Figure 1. – « Reaching » à partir du nœud *P*.

La figure 1 illustre une étape de l'algorithme de « reaching » pour le cas d'une contrainte supplémentaire. A chaque nœud, nous connaissons la valeur actuelle de C_j pour chaque état admissible. Nous effectuons le « reaching » pour l'état (0, 0) du nœud P. Aux nœuds A, B, et C, le coût C'_j de chaque état est indiqué par un coût encerclé et est le minimum du coût actuel C_j et de $C_p + c_{pj}$. Il y a modification des coûts aux trois nœuds.

Le « pulling »

Cette méthode [3] primale-duale, mise au point pour la réoptimisation du problème du plus court chemin avec fenêtres de temps, calcule les chemins X^k_{pj} associés à une partie ou à tous les états du nœud j. Pour un ensemble S d'états du nœud j, le « pulling » consiste à créer les étiquettes $(T^{*k}_j = (\max\{a^l_j, T^l_i + d^l_{ij}\}, l = 1, \ldots, L), C^k_i + c_{ij})$ associées aux chemins $X^{k'}_{pj}$

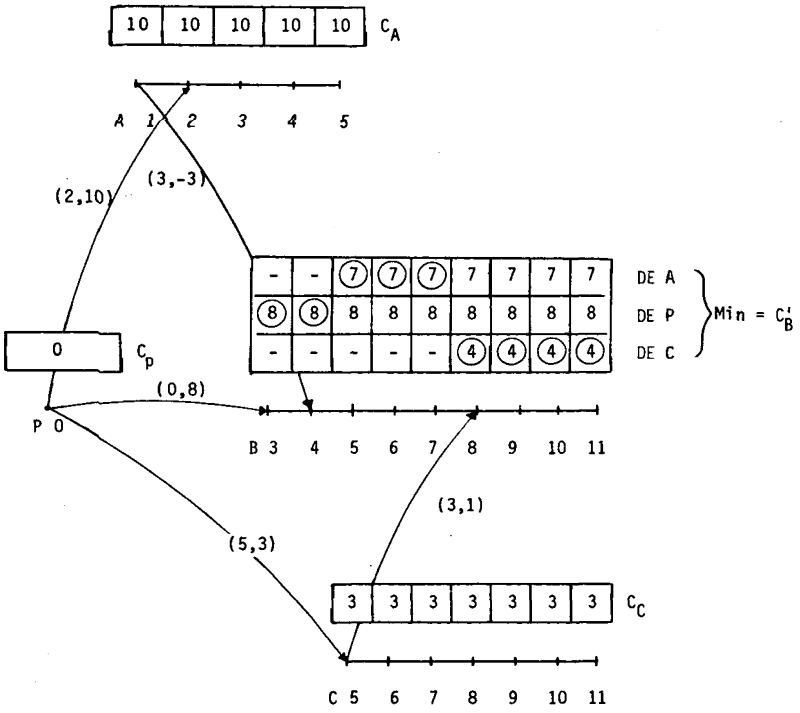

Figure 2. — « Pulling » vers le nœud *B*.

obtenus en prolongeant tous les chemins X^k_{pi} auxquels l'ajout de l'arc $(i, j) \in A$ permet l'arrivée au nœud j en un état $T_j^{*k'} \in S$. Les nouvelles étiquettes étant créées à un seul nœud, cet algorithme demande, au contraire des algorithmes basés sur le « reaching », la mise à jour des étiquettes à un seul nœud.

La figure 2 représente une étape de l'algorithme de « pulling ». Nous effectuons le « pulling » au nœud B pour calculer les nouveaux coûts C'_B des états T_B. Le coût C'_B est indiqué par un coût encerclé et est le minimum des coûts obtenus par prolongements des chemins depuis les nœuds A, C et P.

3. UN ALGORITHME DE « REACHING » MULTIDIMENSIONNEL

Étape 1 : *Initialisation*

$$Q_i = \begin{cases} \{(0..0)\} & i = p \\ \varnothing & \text{pour tout } i \in N, \quad i \neq p \end{cases}$$

$$P_i = 0 \qquad \text{pour tout } i \in N$$

Étape 2 : *Choix de l'étiquette à traiter.*

Trouver l'étiquette $((T_i^{lk}, l=1, \ldots, L), C_i^k)$ de coût lexicographique minimum parmi l'ensemble des étiquettes temporaires $T = \bigcup_{i \in N} (Q_i - P_i)$. Si $T=0$, terminer.

Étape 3 : *Traitement de l'étiquette* $((T_i^{lk} l=1, \ldots, L), C_i^k)$.

Pour tous les successeurs j du nœud i faire
début
si $T_i^{lk} + d_{ij}^l \leqq b_j^l$ (respect des fenêtres d'admissibilité)
alors $(T_j^*, C_j) = ((\max(a_j^l, T_i^{lk} + d_{ij}^l) l=1, \ldots, L), C_i^k + c_{ij})$
$Q_j = EFF\{Q_j \cup (T_j^*, C_j)\}$
fin.
$P_i = P_i \cup \{(T_i^{*k}, C^{kj})\}$
aller à l'étape 2.

EFF { } représente un appel à la méthode permettant d'obtenir les étiquettes efficaces parmi un ensemble d'étiquettes. De façon analogue à l'algorithme de Dijkstra, cet algorithme converge si tous les arcs (i, j) ont une longueur $(d_{ij}^1, \ldots, d_{ij}^L, c_{ij})$ lexicographique positive.

La complexité de l'algorithme présenté est dépendante du nombre d'états admissibles. Soient

L : le nombre de contraintes supplémentaires,

$n = |N|$: le nombre de nœuds dans le graphe G,

$v = \max_{i \in N} \left\{ \prod_{l=1}^{L} (b_i^l - a_i^l + 1) \right\}$: le nombre maximum d'états à un nœud,

$V = \sum_{i=1}^{n} \prod_{l=1}^{L} (b_i^b - a_i^l + 1)$: le nombre total d'états.

L'implantation réalisée utilise une liste pointée triée en ordre lexicographique pour présenter Q_i. L'espace mémoire utilisé est en pire cas $O(|A| + V)$. Toutefois, les listes ne sont efficaces que si une faible proportion des états existe, permettant de réduire la mémoire nécessaire et le travail à effectuer.

L'étape 1 est exécutée une fois et est d'ordre n. L'étape 2 est exécutée au maximum V fois, chaque exécution demande au maximum $O(n)$ opérations. Le traitement de l'étiquette choisie (étape 3) est effectuée au maximum V fois et le nombre de successeurs pour l'étiquette traitée est au maximum de n. Pour chacun de ces successeurs, la fusion et la réduction à un ensemble d'étiquettes non dominées associées au nœud i demande la comparaison de la nouvelle étiquette avec au maximum les $b_i - a_i + 1$ étiquettes éventuelles

présentes. Globalement, pour tous les successeurs, cette opération demande $O(VL)$ opérations. La complexité de cette implantation est d'ordre $O(V^2 L)$. Toutefois, le travail à effectuer est proportionnel au carré du nombre d'étiquettes réellement présentes dans la solution. Cette implantation est intéressante si ce nombre est faible par rapport à V.

4. L'ALGORITHME DE "PULLING" MULTIDIMENSIONNEL

L'algorithme de « pulling » est un algorithme primal-dual. L'ensemble P_j contient des étiquettes primales, c'est-à-dire admissibles et qui sont des bornes supérieures sur le coût des états. L'ensemble Q_j contient des étiquettes duales, c'est-à-dire des bornes inférieures sur le coût des états. Le but de cet algorithme est de toujours réduire l'écart entre la borne supérieure et la borne inférieure. Pour réduire cet écart, l'algorithme effectue un « pulling » primal pour calculer la nouvelle borne supérieure et un « pulling » dual pour calculer la nouvelle borne inférieure. En fait, ces deux étapes sont effectuées simultanément et Desrochers [3] démontre que si tous les arcs ont une longueur lexicographique positive le nombre d'états avec un écart strictement positif entre les bornes inférieures et supérieures diminue à chaque « pulling ».

Étape 1. *Initialisation :*

- préparation des ensembles P_j et Q_j.
- calcul des $(d^{\bullet}_{\bullet j}, c_{\bullet j})$ pour chaque nœud.

Étape 2. *Choix d'une étiquette* $T_j^{\bullet k}, C_j^k$) *à traiter.*

Trouver l'étiquette $(T_j^{\bullet k}, C_j^k)$ de coût lexicographique minimum parmi l'ensemble $S = \bigcup_{j \in N} (Q_j - P_j)$. Si $S = 0$, terminer.

Étape 3. *Choix de la zone à explorer parmi la zone d'incertitude.*

La fin de la zone à explorer TSUP est définie par

$$TR_j = \text{minlex}\{b_j^{\bullet}\} \cup \{T_j^{\bullet m} \mid (T_j^{\bullet m}, C_j^m) \in P_j \text{ et } T_j^{\bullet mL} < T_j^{\bullet k}\}$$

$$TSUP = \text{minlex}\{T_j^{\bullet k} + d^{\bullet}_{\bullet j}, TR_j\}.$$

Étape 4. *Remplacement de l'étiquette* $(T_j^{\bullet k}, C_j^k)$

(A) Calcul de la nouvelle fonction duale dans la zone à explorer

$$R_j = \text{EFF}\{(T_j^{\bullet}, C_j) = ((\max\{a_j^l, T_i^{lk} + d_{ij}^l\}, l = 1, \ldots, L), C_i^k + c_{ij})$$

pour tout $(i, j) \in A$, $(T_i^{\bullet k}, C_i^k) \in Q_i$ et $T_j^{\bullet}$ est dans la zone à explorer.$\}$

(B) Calcul de la fonction primale dans la zone à explorer

$$RP_j = EFF\{(T_j^*, C_j) = ((\max\{a_j^l, T_i^{lk} + d_{ij}^l\}, l = 1, \ldots, L), C_i^k + c_{ij})$$

pour tout $(i, j) \in A$, $(T_j^{*k}, C_i^k) \in P_i$ et T_j^* est dans la zone à explorer. $\}$

(C) Création des nouveaux ensembles Q_j et P_j

$$- P_j = P_j \cup (R_j \cap RP_j)$$
$$- Q_j = (Q_j - \{(T_j^{*k}, C^{kj})\}) \cup (R_j)$$

retourner à l'étape 3.

Nous utiliserons pour exprimer la complexité de l'algorithme de « pulling » les nitations définies dans la section précédente. Nous étudierons la complexité de deux implantations de l'algorithme de « pulling ». Ces implantations ne conservent en mémoire centrale que les étiquettes efficaces dans un SPLAY, structure de donnée proposée par Sleator et Tarjan [16], et qui permet d'accéder à un élément et à son successeur. Sleator et Tarjan montrent que la complexité de toute séquence de m opérations sur un arbre de v étiquettes (avec $m > v$) est d'ordre $m \log v$. Ce résultat permet, en moyenne, d'accéder à une étiquette en temps logarithmique. Selon les résultats obtenus précédemment, [7], seulement 5 à 15 % des étiquettes sont efficaces et l'utilisation de cet arbre permet une économie appréciable de mémoire.

La complexité de l'algorithme de « pulling » dépend de l'implantation utilisée. L'implantation a été effectuée à l'aide de tas (heaps). Toutefois la complexité des trois premières étapes de l'algorithme est indépendante de la structure de données choisie.

L'initialisation, lors de l'étape 1, préparer les ensembles P_i et Q_i. La fabrication d'un tas (heap) comprenant les éléments lexicographiquement minimum de chacun des ensembles Q_i demande $O(Ln \log n)$ opérations. Lors de l'étape 1, il est aussi nécessaire de déterminer, par un examen de tous les arcs, l'arc lexicographiquement minimum à chaque nœud, ce qui demande $O(|A|L)$ opérations. La complexité, en pire cas de l'étape 1, est globalement de $O(L.(|A| + n + n \log n))$ opérations.

L'utilisation d'un tas (heap) permettant l'accès rapide à l'élément lexicographiquement minimum des ensembles Q_i permet de simplifier les étapes 2 et 3. Dans le cas simple, le choix de l'étiquette à remplacer se limite à mettre à jour le tas et à accéder à son sommet.

Cette étape sera exécutée au maximum V fois. Chaque exécution demande $O(L.\log n)$ opérations pour la mise à jour du tas et le choix de l'étiquette à

traiter. Globalement, l'étape 2 demande dans le pire cas $O(V.L.\log n)$ opérations.

La troisième étape peut être exécutée au maximum V fois et demande le calcul de la frontière de la zone à explorer en $O(L)$ opérations. Globalement, l'étape 3 demande en pire cas $O(VL)$ opérations. Globalement, ces trois étapes demandent $O(||A|L+nL+VL.\log n)$ opérations.

La quatrième étape peut être implantée de deux façons, selon l'algorithme utilisé pour trouver les vecteurs efficaces.

L'utilisation d'un SPLAY [16] pour conserver les étiquettes efficaces permet, en moyenne, un temps d'accès logarithmique aux étiquettes. L'utilisation d'une table de dimension V pour représenter les étiquettes associées aux états dans la zone d'incertitude permet de simplifier la vérification de la dominance et de profiter d'une partie des avantages d'une table, tout en conservant l'économie d'espace des arbres. La construction des bornes inférieure et supérieure dans la zone d'incertitude demande le calcul du coût de chacun des chemins en provenance des prédécesseurs du nœud et ayant leur étiquette associée dans la zone d'incertitude. L'accès logarithmique aux étiquettes des prédécesseurs demande en moyenne $O(\log v)$ opérations. Cette moyenne peut être utilisée puisque le nombre d'accès aux étiquettes est $O(v.\text{nombre de successeurs d'un nœud})$ ce qui est plus grand que $O(v)$. La vérification de la dominance s'effectuant dans la table locale, elle se limite à vérifier si le coût du chemin est plus petit que le coût actuel de l'état associé au chemin. Lors de la fabrication des nouveaux ensembles Q_j et P_j, il y a vérification de la dominance entre les étiquettes de la table et reconstruction du SPLAY. Globalement, puisqu'il y a au total V étiquettes et puisque chacune de ces étiquettes est traitée au maximum une fois, l'étape 4 demande pour le remplacement des étiquettes Q_j des n nœuds $O(VL(n \log v+v))$ opérations.

La complexité globale de la première implantation est d'ordre $O(VL.(n \log v+v))$ pour le « pulling » et d'ordre $O(V^2L)$ pour le « reaching ». Puisque $O(n \log v+v)<O(V)$, la complexité de cette implantation du « pulling » est moindre que celle de l'implantation pour le « reaching ».

Le seconde implantation possible utilise aussi un SPLAY pour conserver les étiquettes efficaces. Toutefois, le test de dominance n'est pas effectué dans une table mais plutôt en utilisant un algorithme de calcul du minimum d'un ensemble de vecteurs (c'est-à-dire d'étiquettes multidimensionnelles) [13, 3]. La complexité de cet algorithme est, en pire cas, d'ordre $O(vL\log^2 v)$ à chaque nœud. Globalement, pour les n nœuds, le calcul de la dominance demande $O(VL\log^2 v)$ opérations et l'accès aux prédécesseurs $O(VL\log v)$

opérations. L'étape 5 demande, pour cette implantation,

$$O(VL(n \log v + \log^2 v))$$

opérations.

La complexité globale de la seconde implantation est d'ordre $O(VL(n \log v + \log^2 v))$ pour le « pulling » et d'ordre $O(V^2 L)$ pour le « reaching ». Encore une fois, la complexité de l'implantation du « pulling » est moindre que celle du « reaching ».

5. RÉSULTATS NUMÉRIQUES POUR DES PROBLÈMES AVEC UNE CONTRAINTE SUPPLÉMENTAIRE ($L=1$)

L'algorithme de « pulling » utilisant la première implantation (PULL1) et l'algorithme de « reaching » (REACH) ont été implantés en FORTAN 77. Dans le cas des algorithmes de « pulling », nous utilisons une zone à explorer plus grande que celle définie à l'étape 3. Des tests ont montré qu'en doublant la dimension de la zone à explorer, le temps d'exécution diminue de 25 %. PULL1 et REACH ont été testés sur un ordinateur CDC CYBER 173 pour comparer leurs performances. La comparaison entre les deux algorithmes s'effectuera en faisant varier trois paramètres :

1. Le nombre de nœuds n composant le réseau. Nous résoudrons des problèmes de 500, 1 000 et 2 500 nœuds.

2. Le nombre moyen d'arcs par nœud. Nous choisirons aléatoirement parmi les arcs admissibles selon (6) de façon à obtenir au total environ $10*n$, $25*n$ ou $100*n$ arcs dans le réseau.

3. La largeur des fenêtres de temps. Les débuts de fenêtres a_i seront distribués entre 0 et 100 selon une loi uniforme et la fenêtre de chaque nœud prendra une largeur de 10, 25, 50 ou 100 unités de temps.

Six problèmes différents ont été générés pour chaque combinaison admissible des trois paramètres. Ces problèmes ont été produits en utilisant la même méthode que Desrosiers, Pelletier et Soumis [7].

Les problèmes ont été produits en utilisant les paramètres suivants :

1. les tâches sont dispersées dans un carré de $[0,70] \times [0,70]$ selon une loi uniforme;

2. les durées des tâches d_i sont dispersées dans l'intervalle [5, 15] selon une loi uniforme;

3. le temps inter-tâches t_{ij} est égal à la distance euclidienne entre les tâches i et j additionnée à la durée de la tâche i;

4. le coût inter-tâches c_{ij} est égal à t_{ij} moins une grande constante (33 333). La constante choisie rend le coût de presque toutes les routes de trois nœuds ou plus négatif.

L'ordinateur utilisé ne permettait pas de conserver en mémoire centrale la totalité des nœuds, des arcs et des étiquettes. Les nœuds et les étiquettes sont conservés en mémoire centrale, alors que les arcs sont divisés en pages et une seule page est conservée en mémoire centrale. Lors d'un défaut de page, c'est-à-dire lorsque l'exécution de l'algorithme nécessite des arcs d'une autre page, cette nouvelle page est amenée en mémoire centrale. Le temps de gestion de la mémoire est comptabilisé séparément du temps d'exécution de l'algorithme.

Pour effectuer la comparaison entre les deux algorithmes, les temps d'exécution et de gestion de la mémoire seront utilisés. A titre d'information, le nombre d'étiquettes traitées par REACH, c'est-à-dire celles dont les successeurs sont calculés, et le nombre d'étiquettes remplacées par PULL1 sont indiqués dans les tableaux.

La comparaison des temps d'exécution et de gestion de mémoire pour les problèmes du tableau I permet de déterminer si l'algorithme primal-dual PULL1 est encore compétitif par rapport à l'algorithme primal AMPG pour la résolution de problèmes. L'examen des résultats du tableau I montre que les temps d'exécution des deux algorithmes sont en moyenne les mêmes avec un très léger avantage d'environ 5 % en moyenne pour PULL1. L'algorithme de « pulling » est donc légèrement plus rapide que l'algorithme de « reaching » pour des problèmes avec une contrainte supplémentaire ($L=1$).

6. RÉSULTATS NUMÉRIQUES DU « PULLING » MULTIDIMENSIONNEL

L'algorithme de « pulling » multidimensionnel a été implanté en FORTRAN 77 selon les deux implantations de la section 5. PULL1 utilise des SPLAYS pour conserver les étiquettes efficaces et utilise une table pour vérifier la relation de dominance entre les étiquettes. PULL2 utilise aussi des SPLAYS pour conserver les étiquettes efficaces, mais utilise un algorithme spécialisé pour vérifier la dominance. Les deux implantations partagent les mêmes sous-ritines, sauf celles vérifiant la dominance. Les deux versions ont été compilées sur le compilateur FTN5 avec l'option OPT = 2 sur les ordinateurs CDC CYBER 173 de l'Université de Montréal. La version PULL1 ne peut être utilisée que si le nombre d'états à chaque nœud, c'est-à-dire la taille de la table utilisée pour vérifier la dominance, n'est pas trop grand. La comparaison entre les deux versions ne pourra donc être effectuée que sur de

TABLEAU I

Statistiques d'exécution des deux algorithmes.

|N| réseau	D	|A|	|E_s|	# problèmes résolus	|E| traitées	REACH Temps Gestion mémoire Moy (sec.)	REACH Temps Gestion mémoire Max (sec.)	REACH Temps Exécution Moy (sec.)	REACH Temps Exécution Max (sec.)	# problèmes résolus	|E_r|	PULL1 Temps Gestion mémoire Moy (sec.)	PULL1 Temps Gestion mémoire Max (sec.)	PULL1 Temps Exécution Moy (sec.)	PULL1 Temps Exécution Max (sec.)
500	5500	5044	763	6	763	–	–	.50	.57	6	502	–	–	.69	.70
500	5500	12487	932	6	932	.37	.47	1.12	1.37	6	503	.12	.15	1.10	1.14
500	5500	24982	1045	6	1045	.78	1.13	2.21	2.75	6	517	.38	.68	1.81	1.84
500	5500	49984	1160	6	1160	1.59	1.97	4.51	5.45	6	539	.88	1.20	3.16	3.20
500	13000	4995	1231	6	1231	–	–	.70	.83	6	683	–	–	1.10	1.15
500	13000	12527	1675	6	1675	.65	.93	1.87	2.33	6	850	.38	.39	2.14	2.30
500	13000	24907	2065	6	2065	1.59	2.22	4.17	5.17	6	994	.95	1.04	3.91	4.07
500	13000	49988	2435	6	2435	2.88	3.57	9.33	10.82	6	1112	1.95	2.06	7.40	7.60
500	13000	5033	2167	6	2167	–	–	1.22	1.48	6	1130	–	–	2.15	2.24
500	25500	12501	3309	6	3309	1.56	1.94	3.56	4.42	6	1394	.79	.89	4.92	4.84
500	25500	24975	4132	6	4132	3.25	3.63	8.67	10.56	6	1660	1.75	1.83	8.07	9.34
500	25500	49953	4843	6	4843	6.47	7.19	19.65	22.81	6	1944	3.79	3.93	18.04	19.20
500	25500	4952	4243	6	4243	–	–	2.52	3.12	6	1894	–	–	4.19	4.38
500	50500	12477	6196	6	6196	2.33	3.06	7.69	9.35	6	2502	1.64	1.89	9.37	9.72
500	50500	25074	7961	6	7961	5.59	6.60	20.05	24.17	6	3056	2.99	3.48	19.22	20.71
500	50500	49913	10408	6	10408	10.42	13.23	53.71	60.16	6	3551	5.17	5.52	41.16	43.24
1000	11000	9993	1526	6	1526	–	–	1.01	1.14	6	1004	–	–	1.37	1.39
1000	11000	25015	1827	6	1827	.84	1.01	2.33	2.76	6	1023	.41	.70	2.30	2.34
1000	11000	50129	2061	6	2061	1.95	2.43	4.68	5.55	6	1046	1.07	1.37	3.78	3.83
1000	11000	99739	2393	6	2393	4.19	5.17	10.08	11.90	6	1093	2.43	2.58	6.68	6.74
1000	11000	10018	2434	6	2434	–	–	1.43	1.73	6	1362	–	–	2.30	2.33
1000	26000	25058	3575	6	3575	1.34	1.78	4.33	5.24	6	1745	.94	.96	4.56	4.76
1000	26000	50147	4278	6	4278	2.88	3.93	9.63	10.97	6	2022	2.17	2.35	8.22	8.33
1000	26000	100103	5077	6	5077	6.88	9.20	21.84	24.85	6	2240	4.75	4.98	15.43	16.08
1000	26000	10017	4314	6	4314	–	–	2.59	2.96	6	2254	–	–	4.37	4.51
1000	51000	25080	6523	6	6523	2.78	3.58	8.05	9.30	6	2875	1.95	2.12	9.43	9.83
1000	51000	50016	8287	6	8287	6.75	8.79	19.10	22.53	6	3395	4.19	4.31	18.50	19.47
1000	51000	99959	10322	6	10322	16.58	16.97	45.75	53.62	6	3892	6.22	6.35	37.40	39.95
1000	51000	10012	8242	6	8242	–	–	5.03	5.91	6	3769	–	–	8.54	9.01
1000	101000	25065	12830	5	12557	5.67	7.49	17.33	19.72	6	5150	2.37	2.52	19.61	20.13
1000	101000	50051	13699	1	13699	11.98	11.98	37.79	37.79	1	5993	5.24	5.24	35.96	35.96
2500	27500	24957	3790	6	3778	2.22	3.68	2.61	2.88	6	2535	2.05	2.78	3.86	3.91
2500	27500	62446	4587	6	4580	7.22	9.77	6.08	6.87	6	2571	5.99	7.93	6.28	6.37
2500	27500	124902	5181	6	5179	13.97	19.12	12.70	14.93	6	2645	12.57	18.44	10.10	10.25
2500	27500	249558	5946	6	5943	32.77	39.87	26.57	28.57	6	2741	25.37	35.90	17.81	18.01
2500	65000	25060	6120	6	6098	3.36	4.30	3.57	4.23	6	3564	3.11	3.79	6.19	6.42
2500	65000	62503	8590	6	8570	12.44	13.93	10.36	11.49	6	4409	8.44	10.31	12.11	12.53
2500	65000	125196	10546	6	10498	25.70	30.14	24.36	27.86	6	5056	18.75	25.04	21.54	21.96
2500	65000	249806	12610	0	–	–	–	–	–	6	5527	36.30	45.46	39.87	40.50

tels problèmes. Il n'existe pas d'implantation de l'algorithme de « reaching » multidimensionnel. Toutefois la comparaison effectuée à la section précédente entre le « pulling » et le « reaching » pour le cas d'une seule contrainte supplémentaire montrait que le « pulling » était légèrement plus rapide que le « reaching ». De plus, nous avons vu que la complexité de l'implantation du « reaching » est supérieure à la complexité des implantations du « pulling ».

La comparaison entre les deux implantations se fera sur des problèmes de type fabrication de journées de travail de conducteurs d'autobus urbains. Ce type de problème doit être résolu lors de la fabrication d'horaires de travail par une méthode de génération de colonnes [3]. Dans ce travail, il s'agissait de produire une journée de travail débutant entre les temps A et B, et respectant trois ou cinq contraintes supplémentaires. Une journée de travail comporte de une à plusieurs tâches séparées par des pauses et doit respecter les contraintes suivantes :

1. le nombre de tâches est limité;

2. la durée totale de la journée (c'est-à-dire durée combinée des tâches exécutées et des pauses) est limitée;

3. le nombre d'heures de travail (c'est-à-dire durée totale des tâches exécutées) est limité.

De plus, pour les problèmes avec cinq contraintes supplémentaires, deux autres ressources sont contraintes.

Pour chacune de ces contraintes, la quantité de la ressource déjà utilisée pour atteindre une tâche doit appartenir à la fenêtre d'admissibilité associée à la tâche et à cette ressource. Dans notre type de problèmes, dans deux cas (nombre de tâches exécutées, nombre d'heures de travail) la fenêtre est de la forme $[0\,..\,b\,\text{sup}]$. Dans les trois autres cas (durée totale et les ressources 4 et 5), la fenêtre est adaptée à chaque tâche. La fenêtre sur la durée totale pour une tâche débutant au moment d est $[d-b\,..\,d-a]$.

L'ordre des composantes à l'intérieur des vecteurs $(d_{ij}^1, \ldots, d_{ij}^L)$ est important pour déterminer l'ordre lexicographique et cet ordre détermine l'efficacité de l'algorithme. Nous avons donc choisi, comme première composante du vecteur, la composante l tel que $\Theta^l = \left\{ \dfrac{b_j^l - a_j^l + 1}{\min\limits_{(i,j)\in A} d_{ij}^l} \right\}$ est minimum. Ceci permet de minimiser le nombre de « pulling » à chaque nœud, car Θ^l est le nombre maximum de « pulling » si d_{ij}^l est la première composante du vecteur.

Nous avons produit deux séries de cinq problèmes pour les quatre tailles de problèmes $N = 100$, 250, 500, 1 000. Les caractéristiques de ces problèmes

$[a_i^l, b_i^l]$, $i \in$, $l = 1, \ldots, 5$ et c_{ij}, d_{ij}^l, $l = 1, \ldots, 5$, $(i, j) \in A$, ont été distribuées selon des lois uniformes pour simuler des problèmes de fabrication de journées de façon similaire aux problèmes de la section précédente. La première série représente des problèmes où les données ont été arrondies pour diminuer la quantité de mémoire v requise à chaque nœud, tandis que dans la seconde série, les données sont plus précises. Ceci permet aussi d'étudier l'effet de v sur le comportement des deux implantations. Nous avons doublé les séries en extrayant aussi des problèmes avec $L = 3$.

Le tableau II contient les descriptions des problèmes générés. $|N|$ est le nombre de nœuds dans chaque problème. $|A|$ est le nombre moyen d'arcs admissibles dans le réseau. Il est à noter que le nombre d'arcs admissibles est plus grand dans les problèmes avec trois contraintes que dans ceux avec cinq contraintes. Ceci est dû aux vérifications additionnelles d'admissibilité (6). v est le nombre d'états en un nœud alors que V est le nombre total d'états. V est le nombre maximum d'étiquettes possibles. $|E|$ est le nombre d'étiquettes associées à des chemins efficaces dans la solution finale. Il est fréquent que le nombre d'étiquettes présentes dans la solution lors de l'exécution de l'algorithme soit supérieur à $|E|$.

Les statistiques d'exécution présentées dans le tableau II sont le nombre de « pulling » et les temps d'exécution.

Les statistiques d'exécution des 16 séries de problèmes sont contenues dans le tableau II. La première constatation est que PULL2 a pu résoudre 15 des 16 séries de problèmes, alors que PULL1 n'a pu résoudre, faute de mémoire centrale suffisante, que 11 des 16 séries. La série qu'aucune des implantations n'a pu résoudre demandait beaucoup plus de mémoire centrale que les autres. Les deux implantations semblent être complémentaires. PULL1 est légèrement meilleur que PULL2 sur les problèmes avec $L = 3$ et ou v est petit, cet avantage disparaît lorsque v grandit. Dans le cas ou $L = 5$, la domination est claire, PULL2 demande de 5 à 20 % du temps de calcul de PULL1 et moins de mémoire centrale.

Il est possible d'expliquer la différence de temps d'exécution de PULL1 et PULL2 par la nature des composantes présentes dans l'analyse en pire cas de leur complexité. Pour PULL1, cette complexité est d'ordre $O(VL(n \log v + v))$, et d'ordre $O(VL(n \log v + \log^2 v))$ pour PULL2. La partie $O(VLn \log v)$ est identique pour les deux implantations, seule la partie $O(VLn)$ de PULL1 diffère de la partie $O(VL \log^2 v)$ de PULL2. La vérification de la dominance dans PULL1 demande au moins $O(v)$ étapes alors que dans le cas de PULL2, elle demande $O(k \log^2 k)$ où k est le nombre de vecteurs obtenus ($k \leqq v$). Dans le cas où v est grand et k est petit, $O(v)$ est

TABLEAU II

Statistiques d'exécution des deux implantations.

Description des problèmes					PULL1		PULL2	
$\|N\|$	$\|A\|$	V	$\|E\|$	# PULL	Temps UC (sec.)		Temps UC (sec.)	
					Max	Moy	Max	Moy
Problèmes de trois contraintes avec v variant entre 1 et 200 (moy = 92)								
100	1566	9224	164	101	1,551	1,214	1,011	0,890
250	9096	23086	526	251	6,029	4,612	5,014	4,294
500	36241	46210	1290	501	21,717	16,055	23,071	17,411
1000	144082	92336	2998	1001	86,674	59,645	111,348	69,747
Problèmes de trois contraintes avec v variant entre 1 et 1000 (moy = 427)								
100	1746	42724	348	101	4,953	3,235	1,253	1,097
250	10282	107166	1657	251	17,890	10,509	13,732	6,973
500	40836	214416	4578	501	60,657	34,586	77,735	31,816
1000	espace mémoire insuffisant							
Problèmes de cinq contraintes avec v variant entre 1 et 9200 (moy = 1298)								
100	582	133833	83	101	18,756	10,522	0,602	0,456
250	3364	32658	401	251	47,211	27,211	2,479	2,062
500	12030	646179	1154	501	97,826	55,956	9,677	7,050
1000	46889	1295252	2913	1001	205,787	121,025	39,681	26,371
Problèmes de cinq contraintes avec v variant entre 1 et 46000 (moy = 6012)								
100	641	608212	123	101	espace mémoire insuffisant		0,686	0,517
250	3742	1511336	729	251			3,744	2,592
500	13423	2998202	2513	501			22,712	10,959
1000	52978	6009189	7641	1001			104,709	46,806

souvent plus grand que $O(k \log^2 k)$, ce qui explique que PULL2 soit plus rapide que PULL1 dans le cas où $L = 5$.

Le comportement de l'algorithme PULL2 est surprenant d'un autre point de vue : son temps d'exécution diminue lorsque l'on augmente le nombre de contraintes de 3 à 5 et que le nombre d'états V a considérablement augmenté. Cette diminution s'explique par la réduction du nombre d'arcs admissibles dans les problèmes entraînant une réduction du nombre d'états examinés. Le nombre d'opérations de PULL2 dépend du nombre d'étiquettes examinées plutôt que du nombre d'états. PULL2 permet la résolution de problèmes fortement contraints en un temps raisonnable.

CONCLUSION

L'algorithme de « reaching » est une généralisation évidente des algorithmes de plus court chemin. Dans cet algorithme, il y a un nombre égal des deux opérations de prolongement d'un chemin : 1° calcul du coût d'un

successeur et 2° mise à jour du successeur. Dans le cas des problèmes de plus court chemin, il y a un état par nœud et ces deux opérations s'effectuent en temps constant. Toutefois, dans les problèmes de programmation dynamique où il y a plus d'un état par nœud, le nombre d'opérations nécessaires pour mettre à jour les états du successeur dépend du nombre d'états par nœud.

L'algorithme de « pulling » a été mis au point pour obtenir un meilleur équilibre entre le nombre de chacune de ces deux opérations. L'algorithme de « pulling » cherche à minimiser le nombre de mises à jour des étiquettes du successeur. L'algorithme de « pulling » a donc une meilleure complexité que l'algorithme de « reaching » et les résultats numériques montrent que, dans le cas d'une seule contrainte supplémentaire, le « pulling » est déjà supérieur au « reaching ».

Bien que l'algorithme de « pulling » n'aie pas encore été utilisé pour résoudre des problèmes variés, il a quand même été utilisé avec succès pour résoudre des problèmes de fabrication d'horaires pour les chauffeurs d'autobus [5] et des problèmes de confection de tournées avec contraintes d'horaires et de capacités [6]. Nous croyons donc que cet algorithme a un champs d'application considérable dans le domaine des transports.

REMERCIEMENTS

Ce travail a été subventionné par le Fonds FCAR et le CRSNG. Nous tenons de plus à remercier l'arbitre pour ses nombreuses suggestions.

BIBLIOGRAPHIE

1. Y. P. Aneja, V. Aggarwal et K. P. K. Nair, Shortest chain subject to side constraints, *Networks*, 1983, *13*, p. 295-302.
2. E. V. Denardo, Dynamic Programming, Prentice-Hall, 1982.
3. M. Desrochers, La fabrication d'horaires de travail pour les conducteurs d'autobus par une méthode de génération de colonnes, *Thèse de doctorat*, département d'informatique et de recherche opérationnelle, Université de Montréal, juin 1986, 169 p.
4. M. Desrochers et F. Soumis, A generalized permanent labelling algorithm for the shortest path problem with time windows, *INFOR*, 1988, *26*, p. 193-214.
5. M. Desrochers et F. Soumis, A column generation approach to the urban transit crew scheduling problem. *Transportation Sci.*, 1989, *23*, p. 1-13.
6. M. Desrochers, J. Desrosiers et M. Solomon, A new optimization algorithm for the vehicle routing problem with time windows, Rapport du GERAD G-90-30, 20 p.

7. J. DESROSIERS, P. PELLETIER et F. SOUMIS, Plus court chemin avec contraintes d'horaires, *R.A.I.R.O. Rech. Opér.*, 1983, *17*, p. 357-377.
8. J. DESROSIERS, F. SOUMIS et M. DESROCHERS, Routing with time windows by column generation, *Networks*, 1984, *14*, p. 545-565.
9. W. HABENICHT, Efficient routes in vector-values graphs, Proceedings of the Seventh Conference on Graphtheoretic Concepts in Computer Science, J. R. MÜHLBACHER éd., *Carl Hanser Verlag*, Munich, 1981, p. 349-355.
10. P. HANSEN, Bicriterion path problems, dans G. FANDEL et T. GAL éd., Multiple Criteria Decision Making: Theory and Applications, Lecture Notes in Economics and Mathematical Systems 177, Springer-Verlag, Heidelberg, 1980, p. 109-127.
11. J. M. JAFFE, Algorithms for finding paths with multiple constraints, *Networks*, 1984, *14*, p. 95-116.
12. H. C. JOKSCH, The shortest route problem with constraints, *J. Math. Analysis Applic.*, 1966, *14*, p. 191-197.
13. H. T. KUNG, F. LUCCIO et F. P. PREPARATA, On finding the maxima of a set of vectors, *JACM*, 1975, *22*, 5, p. 469-476.
14. E. Q. V. MARTINS, On a multicriteria shortest path problem, *E. J. Oper. Res.*, 1984, *16*, p. 236-245.
15. M. MINOUX, Plus court chemin avec contraintes : algorithmes et applications, *Ann. Télécommunic.*, 1975, *30*, 12, p. 1-12.
16. D. D. SLEATOR et R. E. TRAJAN, Self-ajusting binary trees, Proc. Fifteenth Annual ACM Symposium on the Theory of Computing, 1983, p. 235-245.
17. P. VINCKE, Problèmes multicritères, *Cahiers Centre Études Rech. Opér.*, 1974, *16*, 425-439.