IFT6580 — Devoir 2

Eric Buist (buisteri@iro.umontreal.ca)

15 mars 2006

Résumé

Dans ce travail, nous testons une heuristique de recherche tabou pour résoudre des problèmes de tournées de véhicules avec fenêtres de temps dures. Cette heuristique effectue plusieurs recherches tabou à partir de solutions initiales différentes et stocke les meilleures solutions dans une mémoire adaptative. Nous examinerons les points forts de cette heuristique et les éléments qui pourraient être améliorés.

1 Introduction

Le problème de tournées de véhicules avec fenêtres de temps est important [4, 6], mais aucun algorithme exact efficace n'a encore été trouvé. Étant donné que ce problème est plus complexe que celui du voyageur de commerce qui est lui-même *NP*-complet, il est improbable de découvrir un jour un tel algorithme. Il faut donc rechercher des heuristiques pour trouver des solutions approximatives.

La recherche tabou [1, 2] est un exemple d'une telle heuristique. Elle part d'une solution initiale et tente de l'améliorer en la transformant itérativement. À chaque itération, le voisinage de la solution courante est généré et la meilleure solution dans ce voisinage est choisie. Contrairement à la descente pure, la recherche tabou prend la meilleure solution du voisinage même si elle est moins bonne que la solution courante. Pour éviter de tourner en rond, une liste tabou est définie afin d'interdire de revisiter les solutions déjà examinées.

Nous allons appliquer la recherche tabou au problème de tournées de véhicules avec fenêtres de temps dures. Nous verrons que la solution initiale revêt une importance capitale pour le succès de la méthode, surtout si le nombre d'itérations est relativement petit. Nous allons implanter cette recherche avec de multiples fils d'exécution afin de tirer parti des nouveaux processeurs à double cœur tels le Pentium D d'Intel et le Athlon64 x2 d'AMD. Nous allons tester l'heuristique sur huit problèmes donnés par [5]. La fonction objectif est hierarchique : minimiser d'abord le nombre de véhicules nécessaires pour ensuite minimiser la distance totale parcourue.

La section suivante définit la notation utilisée pour traiter du problème de tournées de véhicules. La section 3 expose l'heuristique implantée tandis que la section 4 traite de l'implantation que nous

avons faite. La section 5 expose les expériences réalisées avec l'heuristique et la section 6 conclut le travail.

2 Définition d'un problème de tournées de véhicules

Un problème P est composé d'un ensemble de N clients distincts. Chaque client $i = 0,\ldots,N-1$ est caractérisé par une position (x_i, y_i), une fenêtre de service $[e_i, l_i]$, un temps de service s_i et une demande d_i. Le client $i = 0$ est le dépôt à partir duquel tous les véhicules partent au temps e_0 et doivent être revenus avant le temps l_0. Pour tout autre client, le service ne peut débuter avant le temps e_i, mais il doit absolument avoir commencé avant le temps l_i. La distance entre un client i et un client j, notée d_{ij}, est la distance euclidienne $d_{ij} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$. Le temps que prend le véhicule pour aller d'un client vers un autre, noté t_{ij}, correspond à la distance euclidienne d_{ij}.

Une solution $S(P)$ à un problème P est constituée d'un ensemble de R routes, chaque route j étant composée de n_j clients. Soit $r_{j,p}$ l'indice du client de la route $i = 0,\ldots,R-1$ à la position $p = 0,\ldots,n_j - 1$. Par définition, $r_{j,0} = r_{j,n_j-1} = 0$ pour $j = 0,\ldots,R-1$. Nous avons alors

$$n = \sum_{j=0}^{R-1} n_j - 2R + 1.$$

Soit $b_{j,p}$ et $w_{j,p}$ le temps auquel le service du p^{e} client de la route j débute et le temps d'attente du véhicule au client, respectivement. Par définition, $b_{j,0} = 0$ et b_{j,n_j-1} correspond au temps de retour du véhicule au dépôt après sa tournée. Pour alléger la notation, puisque tout client $i > 0$ ne peut se trouver que sur une route, nous définissons respectivement b_i et w_i comme le temps de début de service et le temps d'attente du client i, peu importe la route sur lequel il se trouve. Si le client i n'est pas dans la solution, b_i est alors indéfini. Sinon, il faut que $e_i \leq b_i \leq l_i$. Si le véhicule arrive au temps $a_i < e_i$, il doit attendre pendant $w_i = e_i - a_i$ unités de temps.

Nous définissons également δ_j comme la distance totale parcourue par le véhicule de la route j. Cette distance est donnée par

$$\delta_j = \sum_{k=1}^{n_j-1} d_{r_{j,k-1},r_{j,k}}.$$

Soit également D_j la demande totale sur la route j.

$$D_j = \sum_{k=0}^{n_j-1} d_{r_{j,k}}.$$

Cette demande ne doit pas dépasser la capacité du véhicule.

3 Heuristique utilisée

Pour tenter de résoudre le problème de tournées de véhicules avec fenêtres de temps, nous avons utilisé une recherche tabou avec solutions initiales multiples et mémoire adaptive. Les solutions initiales sont obtenues par deux heuristiques de construction tandis que la recherche tabou effectue des échanges de segments d'une route à une autre. Les solutions produites sont insérées dans une mémoire adaptive servant ensuite à construire de nouvelles solutions initiales pour continuer la recherche.

3.1 Construction des solutions initiales

La première étape de toute recherche tabou consiste à construire une solution initiale à l'aide d'une heuristique. Nous avons implanté deux heuristiques différentes présentées dans [4] : la première heuristique d'insertion et l'heuristique de gain.

L'insertion incorpore itérativement chaque client à la solution. Tout d'abord, une route vide, reliant le dépôt avec lui-même, est créée et marquée comme étant la route courante. À chaque itération, tous les sommets non encore insérés sont testés afin de déterminer la meilleure insertion possible à ce moment. Pour chaque client, chaque point d'insertion dans la route courante est testé. Le client u pourra être inséré entre les clients i et j de la route courante qui minimisent le coût $c_1(i,u,j)$ sans violer les contraintes de temps. L'article [4] expose une méthode permettant de vérifier la contrainte efficacement, par propagation du délai imposé par l'insertion. Le client inséré, u^*, doit minimiser une seconde fonction de coût $c_2(i(u^*),u^*,j(u^*))$, où $i(u)$ et $j(u)$ sont les meilleurs voisins du client u à l'itération courante. S'il reste des clients à insérer et qu'aucun point d'insertion n'est disponible sur la route courante, une nouvelle route est construite et devient la route courante. Une nouvelle route est également créée si la capacité restante du véhicule de la route courante excède la demande de tous les clients pouvant être insérés. Ce processus se poursuit jusqu'à ce que tous les clients soient insérés.

Dans le cas de la première heuristique d'insertion (i1), la fonction $c_1(i,u,j)$ est calculée par une combinaison linéaire de deux fonctions distinctes. Soit $c_{11}(i,u,j) = d_{iu} + d_{uj} - \mu d_{ij}$ le gain occasionné par l'insertion du client entre les clients i et j et soit $c_{12}(i,u,j) = b_j^{\text{new}} - b_j$ le délai occasionné par l'insertion, où b_j^{new} représente le temps auquel le service du client j débuterait si le client u était inséré entre i et j. La fonction de coût est donnée par

$$c_1(i,u,j) = \alpha_1 c_{11}(i,u,j) + \alpha_2 c_{12}(i,u,j),$$

avec $\alpha_1 \geq 0$, $\alpha_2 \geq 0$, $\mu \geq 0$ et $\alpha_1 + \alpha_2 = 1$. La fonction $c_2(i,u,j)$ est quant à elle donnée par

$$c_2(i,u,j) = -\lambda d_{0u} + c_1(i,u,j),$$

avec $\lambda \geq 0$.

Dans le cas de l'heuristique des gains, une route est d'abord construite pour chaque client. À chaque itération, l'heuristique fusionne les deux routes qui maximisent le gain $c_{11}(i,u,j)$ sans

violer les contraintes de temps. Deux routes i et j sont dites compatibles si $b_{i,n_i-2} \leq b_{j,1}$ ou $b_{j,n_j-2} \leq b_{i,1}$ et que l'insertion des clients à la fin de la première route ne viole pas les contraintes de temps ou de capacité.

La valeur habituelle de ces paramètres est $\alpha_1 = 1$, $\alpha_2 = 0$, $\mu = 1$ et $\lambda = 1$. Fixer ces paramètres de façon aléatoire permet d'induire de la variabilité afin de produire des solutions initiales différentes avec la même heuristique.

3.2 Recherche tabou

Maintenant que nous disposons d'heuristiques pour construire des solutions initiales, nous pouvons définir les paramètres de la recherche tabou. Nous avons opté pour des échanges de segments proposés par [6], avec une composante stochastique pour réduire l'espace de solutions à explorer. Contrairement à [6], nous ne permettons pas la violation des contraintes de temps. L'échange de segments (*CROSS exchange*) nécessite de sélectionner deux routes i et j et quatre points $x_1 = 0,\ldots,n_i - 2$, $y_1 \geq x_1$, $x_2 = 0,\ldots,n_j - 2$ et $y_2 \geq x_2$. L'échange permute les clients $r_{r_i,x_1+1},\ldots,r_{r_i,y_1}$ de la route r_i avec les clients $r_{r_j,x_2+1},\ldots,r_{r_j,y_2}$ de la route r_j. Si $x_1 = y_1$, il faut que $x_2 \neq y_2$ pour que quelque chose se produise ; nous avons alors un transfert de sommets d'une route à une autre. De la même façon, si $x_2 = y_2$, il faut que $x_1 \neq y_1$. Ce cas particulier est important, car il permet de réduire le nombre de routes dans une solution. Il est également possible que $i = j$; cela produit un réordonnancement des sommets sur une même route.

Pour initialiser la recherche tabou, l'heuristique a besoin d'une solution initiale et d'un ensemble $\{r_0,\ldots,r_{R'-1}\}$ de routes sur lesquelles il va travailler, avec $r_i = r_j$ si et seulement si $i = j$. Dans notre cas, cet ensemble englobe toutes les routes de la solution si bien que $R' = R$ et l'ensemble est $\{0,\ldots,R-1\}$. Pour chaque paire (i,j) de routes, avec $i = 0,\ldots,R'-1$, $j = 0,\ldots,R'-1$ et $i \leq j$, des longueurs sont générées de façon aléatoire. Si $i = j$, une seule longueur ℓ_i est générée sur $0,\ldots,n_{r_i}$. Si $i \neq j$, deux longueurs sont générées : $\ell_{i,j,1}$ sur $0,\ldots,n_{r_i}$ et $\ell_{i,j,2}$ sur $0,\ldots,n_{r_j}$. Ces longueurs définissent la taille des segments impliqués dans les échanges et sont générés selon une loi de probabilité uniforme discrète.

À chaque itération de la recherche tabou, chaque paire de routes (i,j) est testée. Pour chaque paire, plusieurs échanges de segments sont évalués dans le but de déterminer la meilleure modification à appliquer à la solution. Pour une paire (i,i), l'heuristique teste toutes les valeurs $x_1 = 0,\ldots,n_{r_i} - \ell_i - 4$, $y_1 = x_1 + \ell_i$, $x_2 = y_1 + 1,\ldots,n_i - 3$ et $y_2 = x_2 + 1,\ldots,n_i - 2$. Cela correspond à un segment de longueur ℓ_i et un second segment de n'importe quelle longueur. Dans le cas de paires de routes différentes, l'algorithmme teste $x_1 = 0,\ldots,n_{r_i} - \ell_{i,j,1} - 2$, $y_1 = x_1 + \ell_{i,j,1}$, $x_2 = 0,\ldots,n_{r_j} - \ell_{i,j,2} - 2$ et $y_2 = x_2 + \ell_{i,j,2}$. Cela correspond à tous les segments de longueur $\ell_{i,j,1}$ sur la route r_i et de longueur $\ell_{i,j,2}$ sur la route r_j.

Il existe deux cas particuliers à cette heuristique de test. Si $\ell_i = n_{r_i}$, le test est appliqué avec toutes les valeurs de $\ell_i = 0,\ldots,n_{r_i} - 1$, ce qui explore davantage de solutions. De la même façon, si $\ell_{i,j,1} = n_{r_i}$ ou $\ell_{i,j,2} = n_{r_j}$, toutes les longueurs de segments sont testées. Par contre, si $\ell_i = n_{r_i} - 1$, aucun test n'est effectué. Une paire est également sautée si $\ell_{i,j,1} = \ell_{i,j,2} = 0$ ou si $\ell_{i,j,1} = n_{r_i} - 1$ et $\ell_{i,j,2} = n_{r_j} - 1$.

Pour chaque paire de segments considérée, l'heuristique évalue le gain en distance parcourue. Cela permet de calculer la nouvelle distance parcourue si l'échange de segments était effectué. Cette distance peut être calculée en temps constant, car un nombre fixe de segments sont modifiés par l'échange. Si la diminution de la distance totale est inférieure à celle du meilleur candidat trouvé pendant l'itération, le candidat testé est rejeté sans aucun autre calcul, sans même déterminer si l'échange de segments est permis. Pour les candidats restants après ce test, la recherche écarte les échanges qui violeraient les contraintes de temps. Le test de validité des contraintes est malheureusement linéaire, car les routes impliquées doivent être parcourues pour vérifier chaque client.

Un échange de segments devient le meilleur candidat s'il permet de diminuer le nombre de routes dans la solution ($x_1 = y_1$ ou $x_2 = y_2$) ou s'il diminue la distance totale parcourue par rapport au meilleur candidat précédent. Par contre, il est éliminé si l'échange est tabou. Un échange est dit tabou si les deux listes de clients $(r_{r_i,x_1+1},\ldots,r_{r_i,y_1})$ et $(r_{r_j,x_2+1},\ldots,r_{r_j,y_2})$ sont tabous.

Le statut tabou d'une liste de clients peut être levé à deux conditions. Une liste n'est plus tabou après un nombre fixé d'itérations et un critère d'aspiration permet de lever le statut tabou temporairement si la solution produite par l'échange est meilleure que toutes les autres solutions produites par la recherche jusqu'à présent. Tout échange qui passe ces tests est stocké dans un cache afin d'accélérer les futures itérations.

Lorsque deux routes et quatre positions sont enfin sélectionnées, l'échange de segments est effectué et la nouvelle solution est stockée en mémoire si elle est meilleure que la meilleure solution précédente. Les deux listes de clients impliqués dans l'échange sont déclarées tabou et les caches sont mis à jour de la façon suivante. Pour chaque paire de routes (i', j'), l'échange mis en cache est effacé si $i' = i$, $i' = j$, $j' = i$ ou $j' = j$. De même, étant donné que la longueur des routes modifiées a changé, de nouvelles longueurs sont générées aléatoirement pour les routes impliquées seulement. Si, au cours de la recherche, une route ne contient plus de clients ($n_{r_i} = 2$), cette route est supprimée.

La recherche tabou s'arrête après un nombre maximal d'itérations ou si la meilleure solution trouvée ne change pas pendant un nombre fixé d'itérations. Jusque-là, il est donc possible d'ajuster cinq variables : l'heuristique de construction d'une solution initiale, le nombre d'itérations de conservation du statut tabou, le nombre maximal d'itérations, le nombre maximal d'itérations pendant lequel aucune amélioration n'est observée et la possibilité de fixer $\ell_i = n_{r_i}$, $\ell_{i,j,1} = n_{r_i}$ ou $\ell_{i,j,2} = n_{r_j}$ pour certaines paires.

Après la recherche tabou, il est possible d'appliquer une intensification consistant à tester toutes les paires de routes et tous les segments plutôt que générer des longueurs de façon aléatoire. Cette intensification est appliquée à partir de la meilleure solution trouvée et pendant un nombre maximal d'itérations.

3.3 Diversification et mémoire adaptative

Notre heuristique finale de recherche tabou génère M solutions initiales et applique, en parallèle, une recherche tabou sur chaque solution. Avec une certaine probabilité, la solution est

construite par insertion ou par gains. Pour l'insertion, α_1 est choisi de façon uniforme sur $[0,1)$ et $\alpha_2 = 1 - \alpha_1$. Les paramètres μ et λ suivent une loi exponentielle dont le paramètre est fixé par l'utilisateur. Par exemple, si le paramètre pour μ est λ_μ, la valeur de μ peut être entre 0 et l'infini, mais en moyenne, elle sera $1/\lambda_\mu$. Le paramètre μ pour l'heuristique de gains est lui aussi exponentiel. Il est possible d'ajuster le taux pour les trois paramètres indépendamment.

La recherche est effectuée comme à la section précédente et la meilleure solution obtenue est ajoutée à une mémoire adaptative avant que le fil d'exécution ne soit mis en attente. Lorsque tous les M fils d'exécution ont terminé leur recherche tabou initiale et incorporé leur solution à la mémoire adaptative, la seconde phase de la recherche peut débuter.

Chacun des M fils d'exécution tente alors, indépendamment des autres fils, de construire une solution initiale à partir du contenu de la mémoire adaptative, améliore cette solution par recherche tabou et, si la solution est meilleure que la pire solution dans la mémoire adaptive, l'ajoute à la mémoire. Ajouter la solution seulement si elle est meilleure que la meilleure solution réduit beaucoup la fréquence des ajouts et, par le fait même, la diversité des solutions en mémoire. Si, après l'ajout, la taille de la mémoire dépasse le maximum permis, la pire solution y est supprimée. Chaque fil d'exécution répète ce processus pendant un nombre maximal d'itérations, s'arrêtant si plus aucune amélioration n'est observée après un certain nombre d'itérations. La recherche s'arrête lorsque tous les fils d'exécution ont terminé leur traitement.

La construction d'une solution initiale repose sur une heuristique stochastique de sélection de routes appelé brassage de solutions. D'abord, une permutation de solutions est générée et l'heuristique prend la première solution de cette permutation. Il génère ensuite une permutation de routes dans la solution. Chaque route de la permutation est alors testée afin de trouver la première route compatible. Une route est compatible avec la nouvelle solution si elle ne contient aucun client qui n'est pas déjà dans la nouvelle solution. Lorsqu'une route compatible est trouvée, l'heuristique passe à la solution suivante de la permutation. Cela permet d'obtenir des routes provenant de diverses solutions. Contrairement à [6] qui peut travailler avec des solutions partielles, les clients restants après l'application de l'heuristique de construction sont insérés dans de nouvelles routes à l'aide d'une heuristique de complétion de solutions, à savoir l'heuristique d'insertion de Solomon.

La génération de la permutation de solutions utilise un algorithme particulier qui favorise la sélection des meilleures solutions en premier. À chaque solution de la mémoire adaptative, l'algorithme associe un rang $\rho_i = T - i$, où T est la taille de la mémoire adaptative et $i = 0, \ldots, T-1$. De cette façon, la meilleure solution a le plus haut rang. Une solution i est alors sélectionnée avec probabilité

$$p_i = \frac{\rho_i}{\sum_{j=0}^{T-1} \rho_j}.$$

Lorsque la solution est sélectionné, ρ_i est mis à 0 afin qu'elle ne puisse plus être sélectionnée de nouveau. Pour la permutation de routes, l'algorithme habituel est employé, donnant la même chance à toute permutation.

Nous avons essayé une seconde heuristique qui n'a pas donné de meilleurs résultats : le brassage de routes. À chaque route, nous associons un coefficient donné par le produit de la distance

totale parcourue par le véhicule et du pourcentage de clients du problème qui ne sont pas sur cette route. Une bonne route doit ainsi être courte et contenir un haut pourcentage des clients du problème. L'algorithme précédent est utilisé, cette fois pour générer une permutation de routes. Chaque route de la permutation est testée afin de l'ajouter à la nouvelle solution.

4 Implantation de la recherche tabou

L'heuristique décrite à la section précédente a été implantée dans le langage de programmation Java. Dans le programme, un problème est représenté comme un tableau de clients, chaque client étant représenté par un objet de classe `Customer`. Une solution est, quant à elle, représentée par un tableau de routes, chaque route comportant une liste doublement chaînée de clients. La liste chaînée est utilisée en raison des nombreuses insertions effectuées. En fait, les éléments d'une route sont des instances de la classe `CustomerData` qui comportent une référence vers le client correspondant ainsi que le temps de début de service et le temps d'attente. Nous avons utilisé de tels objets intermédiaires pour permettre à de multiples fils d'exécution de travailler sur le même problème. Une solution comporte également un tableau de booléens permettant de tester en temps constant si un client y est inséré ou pas.

Malheureusement, l'utilisation du cadre des Collections de Java impose quelques restrictions à l'utilisation des listes chaînées. S'il était possible d'accéder aux nœuds internes formant la liste, un échange de segments pourrait se faire en temps constant. Malheureusement, cet accès n'est pas possible et il faut itérer sur tous les clients des segments pour effectuer les échanges. La nécessité d'utiliser de nombreux itérateurs pour ne pas avoir à implanter les structures de données directement a soulevé de nombreux doutes quant à la performance de l'implantation, mais le temps d'exécution est malgré tout demeuré raisonnable pour la taille des problèmes traités. La réduction de performance est largement compensée par la simplification du code. Bien entendu, si le programme devait travailler avec des problèmes de très grande taille, avec des centaines de milliers de clients, il faudrait peut-être davantage optimiser les structures de données.

Pour l'implantation de l'heuristique de gains, nous avons utilisé la file de priorité fournie par Java 5. Cette file implante un monceau qui permet de retrouver le plus grand élément en temps constant et d'ajouter ou supprimer des éléments en temps logarithmique. Dans le cas de la mémoire adaptative, nous avons utilisé un ensemble trié plutôt qu'une file de priorité afin de pouvoir énumérer tout le contenu de la mémoire en ordre. Cette énumération est nécessaire pour notre heuristique de construction de solutions initiales. Nous avons utilisé l'implantation par défaut de Java qui encapsule un arbre rouge-noir permettant des opérations en temps moyen logarithmique.

Pour ce qui est de la génération des variables aléatoires, nous avons opté pour le générateur MRG32k3a fourni par le cadre Stochastic Simulation in Java (SSJ) [3]. L'implantation fournie par SSJ permet de scinder la période du générateur, c'est-à-dire l'ensemble des valeurs qu'il peut produire, en un grand nombre de longues séquences qui peuvent être traitées comme des générateurs indépendants. En associant une séquence différente de nombres aléatoires (*random stream*) à chaque fil d'exécution, il est possible d'améliorer la reproductibilité des expériences de façon

importante. De cete façon, à chaque exécution, chaque fil recevra toujours les mêmes nombres aléatoires. Avec un seul générateur, les valeurs aléatoires s'entre-mêleraient d'un fil d'exécution à l'autre et les résultats dépendraient de l'ordre d'exécution des différentes tranches de temps, donc du système d'exploitation et des programmes en cours d'exécution. Le générateur MRG32k3a nécessite, en guise de germe, un ensemble de six valeurs sur 64 bits que notre programme peut générer aléatoirement en utilisant le générateur LCG de Java.

Il est également important de réaliser une synchronisation de type barrière à la fin de chaque itération afin que les fils d'exécution ajoutent les solutions à la mémoire au même moment et toujours dans le même ordre. Sans cette précaution, le programme produira des résultats différents à chaque exécution et ces résultats dépendront de l'ordre d'exécution des tranches de temps.

5 Expérimentations

Nous avons effectué, avec le problème RC201, quelques expérimentations avec la recherche tabou de base. Ensuite, nous avons appliqué la recherche tabou améliorée sur les huit problèmes avec cinq germes différents. Le tableau 1 donne les paramètres utilisés pour les recherches tabous tandis que le tableau 2 donne les germes pour les cinq expériences effectuées sur chaque problème. Dans le cas de la recherche tabou améliorée, une première séquence de variables aléatoires est employé pour générer les paramètres des constructeurs de solutions initiales et de l'heuristique de complétion de solutions tandis qu'une séquence est dédiée à chaque fil d'exécution.

TAB. 1 – Paramètres des expérimentations

Paramètre	Valeur
Nombre maximal d'itérations de recherche tabou	500
Nombre d'itérations pendant lesquelles un segment est tabou	15
Nombre d'itérations permises sans amélioration	50
Permettre de tester pour tous les segments pour certaines routes	Non
Nombre d'itérations de l'intensification	100
Nombre d'itérations permises sans amélioration	10
Taille de la mémoire adaptative	30
Nombre de solutions initiales	10
Nombre de fils d'exécution	10
Probabilité de l'heuristique de Solomon	1
Paramètre α_1	U(0,1)
Paramètre α_2	$1-\alpha_1$
Paramètres μ et λ	Expo(1)

TAB. 2 – Germes du générateur MRG32k3a pour les expériences

	Germe					
0	12 345	12 345	12 345	12 345	12 345	12 345
1	1 553 932 502	-2 090 749 135	-287 790 814	-355 989 640	-716 867 186	161 804 169
2	1 402 202 751	535 445 604	1 011 567 003	151 766 778	1 499 439 034	-51 321 412
3	1 924 478 780	-370 025 683	-1 554 121 271	496 460 768	679 749 574	-301 730 690
4	-992 618 231	1 128 070 351	-235 907 694	621 908 703	281 685 584	-1 811 800 664

Le germe 0 est le germe par défaut du MRG32k3a de SSJ. Les autres germes ont été obtenu avec le générateur `Random` de Java, avec un germe 12 345.

5.1 Recherche tabou simple

L'heuristique d'insertion avec $\alpha_1 = 1$, $\alpha_2 = 0$, $\mu = 1$ et $\lambda = 1$ produit, pour le problème RC201, une solution avec six routes et une distance totale de $2\ 361,95$. La recherche tabou avec les paramètres du tableau 1 et le germe 0 a pu réduire cette distance à $1\ 671,83$ sans toutefois diminuer le nombre total de routes. Cette recherche s'est arrêtée après seulement 162 itérations et seules 38 itérations ont amélioré la solution. Le tableau 3 présente la route obtenue par insertion et celle obtenue par recherche tabou. La dernière du tableau indique le nombre total de routes ainsi que la distance totale parcourue.

La recherche tabou permet de mieux distribuer les clients entre les six routes, permettant une meilleure utilisation des véhicules. Avec l'heuristique d'insertion, la dernière route a peu de clients tandis que la recherche tabou a équilibré les routes.

L'heuristique permet de réduire la distance totale parcourue, mais peut-elle réduire le nombre de véhicules utilisés ? Si tel n'est pas le cas, elle ne permettra pas d'obtenir une solution intéressante à moins que la solution initiale ait peu de routes. Nous pouvons d'abord tenter d'intensifier la recherche en permettant parfois, même lorsque nous ne sommes pas en phase d'intensification, de tester tous les segments de certaines paires de routes. Étonamment, cela dégrade la solution produite, la distance totale devenant $1\ 697,12$. Si nous tentons d'augmenter le nombre d'itérations à 25 000 et arrêtons la recherche s'il n'y a pas eu d'amélioration après 10 000 itérations, nous obtenons bien entendu une meilleure solution. Pour ce qui est de l'intensification, le nombre maximal d'itérations est augmenté à 1 000 et la recherche s'arrête après 200 itérations sans amélioration. Avec ces nouveaux paramètres, la distance totale passe à $1\ 543,48$, mais le nombre de routes reste à six. De plus, le temps d'exécution passe d'une seconde à quinze secondes. Il semble donc, à priori, que la recherche tabou ne permet pas de réduire le nombre de véhicules.

Essayons à présent de générer une nouvelle solution initiale, cette fois avec l'heuristique des gains de paramètre $\mu = 1$. Comme le montre le tableau 4, cette solution initiale n'est pas très bonne avec ses dix-huit routes et sa distance totale de $1\ 761,78$. Si nous appliquons la recherche tabou avec les paramètres du tableau 1, nous obtenons une meilleure solution avec onze routes et une distance totale de $1\ 408,62$. Ainsi, la recherche tabou peut réduire le nombre de routes. Si nous appliquons cette recherche pendant 25 000 itérations, nous obtenons une solution avec six routes et une distance totale de $1\ 551,98$, ce qui est proche de la solution obtenue par l'heuristique

d'insertion.

Ces expériences montrent que la recherche tabou peut améliorer la solution jusqu'à un certain point, mais sans diversification, elle en vient à stagner. De plus, il est important de bien choisir la solution initiale. En particulier, avec l'heuristique de gains, la recherche tabou travaillera beaucoup inutilement.

5.2 Recherche tabou améliorée

Testons maintenant la recherche tabou améliorée sur nos huit problèmes. Jusque-là, l'exécution des recherches n'avait pris qu'une seconde environ. Avec la recherche tabou améliorée, il faut plus d'une minute sur un processeur Intel Pentium D 2, 8GHz, sous Java 5 x86_64. Comme le programme a été exécuté sous un noyau SMP de Linux Fedora Core 4 et qu'il comporte plusieurs fils d'exécution, il utilise les deux cœurs du processeur. Malheureusement, il n'a pas été possible de contraindre le programme à s'exécuter sur un seul processeur pour étudier l'effet du double cœur sur la performance.

Les tableaux 5 et 6 présentent les meilleures solutions trouvées pour les huit problèmes par notre heuristique. Les temps de traitement sont ici des temps système, car Java ne dispose pas d'une fonctionnalité pour obtenir le temps de processeur. Java 5 fournit certes cette possibilité, mais un bogue sous Linux la rend inutilisable dans une application avec de multiples fils d'exécution. Ces solutions se rapprochent des meilleures solutions reportées sur [5], mais notre heuristique ne bat pas les meilleures heuristiques existantes, comme le montre les solutions du tableau 8.

Le tableau 7 donne la meilleure et la pire solution trouvée pour chacun des problèmes traités avec les cinq germes du tableau 2. La différence de distances entre le meilleur cas et le pire cas n'est pas très grande et le nombre de routes diffère seulement pour un problème. Cela montre que l'heuristique pourra poduire une solution intéressante en moyenne, peu importe le germe du générateur de nombres aléatoires employé.

6 Conclusion

Nous avons implanté une heuristique de recherche tabou qui permet, en un temps raisonnable, de trouver une solution intéressante et qui exploite mieux le potentiel des processeurs à double cœur qu'une recherche tabou simple. Par contre, notre heuristique ne bat pas les meilleures méthodes de résolution de ce problème.

Il serait intéressant de comparer notre variante avec celle décrite dans [6] où les fenêtres de temps peuvent être violées avec une pénalité. Cela modifie le voisinage et, comme nous l'avons découvert en permettant la recherche exhaustive de segments pour certaines routes, modifier le voisinage peut affecter la solution produite. Il serait possible de tirer profit de ce constat pour élaborer une heuristique de changement du voisinage qui pourrait être employée lorsque la recherche tabou ne parvient plus à améliorer la solution.

La recherche tabou pourrait aussi avoir la possibilité d'ajouter de nouvelles routes, pas seulement en supprimer. Lorsqu'il y a peu de routes dans une solution, les échanges de segments sont plus difficiles si la contrainte de temps ne peut pas être violée. Déplacer des clients dans une route temporaire pourrait permettre de résoudre ce problème.

Références

[1] Fred Glover. Tabu search — part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[2] Fred Glover. Tabu search — part ii. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[3] Pierre L'Ecuyer et Eric Buist. Simulation in Java with SSJ. Dans *Proceedings of the 2005 Winter Simulation Conference*, pages 611–620, décembre 2005.

[4] Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, mars 1987.

[5] Marius M. Solomon. VRPTW benchmarking problems, mars 2005. URL `http://web.cba.neu.edu/~msolomon/problems.htm`.

[6] Eric Taillard, Philippe Badeau, Michel Gendreau, François Guertin et Jean-Yves Potvin. A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation Science*, 31:170–186, 1997.

TAB. 3 – Recherche tabou de base sur RC201

Heuristique d'insertion

j	δ_j	D_j	
0	564,884	456,000	0, 92, 63, 65, 83, 82, 98, 11, 47, 15, 16, 52, 86, 87, 9, 99, 57, 90, 84, 49, 66, 20, 56, 89, 48, 24, 25, 91, 93, 80, 70, 0
1	639,808	474,000	0, 59, 14, 5, 45, 69, 2, 88, 7, 12, 78, 73, 79, 8, 3, 46, 6, 37, 43, 68, 4, 60, 55, 1, 17, 58, 100, 0
2	611,103	435,000	0, 72, 36, 39, 42, 27, 29, 31, 71, 61, 44, 40, 38, 41, 81, 94, 85, 50, 26, 34, 54, 96, 32, 35, 77, 0
3	320,381	314,000	0, 95, 62, 33, 64, 19, 21, 23, 18, 76, 51, 22, 53, 10, 97, 13, 74, 0
4	111,761	29,000	0, 30, 28, 67, 0
5	114,018	16,000	0, 75, 0
6	2 361,95		

Recherche tabou (38/162 itérations améliorant la solution, 8/32 itérations d'amélioration pendant l'intensification)

j	δ_j	D_j	
0	269,049	287,000	0, 42, 36, 39, 5, 45, 7, 88, 86, 57, 22, 49, 20, 25, 77, 58, 0
1	307,624	342,000	0, 14, 59, 75, 23, 21, 19, 18, 76, 51, 85, 50, 34, 32, 26, 89, 48, 24, 0
2	221,350	209,000	0, 2, 47, 98, 82, 69, 53, 55, 60, 4, 1, 70, 100, 0
3	302,338	312,000	0, 83, 63, 33, 27, 71, 61, 44, 40, 38, 41, 81, 94, 84, 56, 66, 96, 54, 68, 0
4	322,445	247,000	0, 72, 92, 95, 62, 31, 29, 30, 28, 67, 90, 99, 9, 87, 97, 10, 13, 17, 74, 0
5	249,023	327,000	0, 65, 64, 52, 16, 15, 11, 12, 78, 73, 79, 6, 8, 46, 3, 43, 35, 37, 93, 91, 80, 0
6	1 671,83		

TAB. 4 – Recherche tabou de base sur RC201, solution initiale obtenue par gains

Heuristique de gains

j	δ_j	D_j	
0	84,071	133,000	0, 2, 6, 7, 8, 46, 4, 100, 0
1	80,667	40,000	0, 14, 47, 17, 0
2	92,361	121,000	0, 42, 44, 38, 40, 43, 68, 0
3	84,251	83,000	0, 45, 5, 3, 1, 70, 0
4	100,931	62,000	0, 59, 97, 74, 0
5	86,103	88,000	0, 61, 41, 81, 94, 96, 0
6	111,572	75,000	0, 63, 76, 89, 91, 0
7	94,052	97,000	0, 64, 19, 18, 48, 24, 0
8	129,129	62,000	0, 65, 52, 75, 58, 77, 0
9	64,432	26,000	0, 67, 71, 93, 0
10	108,394	155,000	0, 69, 98, 88, 53, 73, 79, 78, 60, 55, 0
11	97,627	104,000	0, 72, 39, 36, 35, 37, 54, 0
12	92,020	119,000	0, 82, 12, 11, 15, 16, 13, 0
13	95,685	74,000	0, 83, 21, 23, 25, 0
14	102,918	115,000	0, 84, 85, 51, 22, 20, 49, 66, 0
15	97,150	135,000	0, 90, 99, 57, 86, 87, 9, 10, 0
16	126,025	177,000	0, 92, 31, 29, 27, 28, 26, 34, 50, 80, 0
17	114,391	58,000	0, 95, 62, 33, 30, 32, 56, 0
18	1 761,78		

Recherche tabou (386/500 itérations d'amélioration, 29/61 itérations d'amélioration pendant l'intensification)

j	δ_j	D_j	
0	95,085	96,000	0, 71, 67, 94, 84, 56, 91, 0
1	44,044	25,000	0, 81, 61, 90, 0
2	123,214	216,000	0, 5, 45, 2, 6, 7, 8, 46, 3, 1, 4, 100, 70, 0
3	130,207	231,000	0, 69, 82, 11, 15, 16, 12, 73, 79, 78, 60, 55, 0
4	173,300	146,000	0, 95, 63, 85, 51, 76, 89, 48, 25, 77, 58, 0
5	126,982	226,000	0, 65, 52, 83, 64, 19, 21, 23, 18, 49, 22, 20, 66, 0
6	113,463	33,000	0, 33, 30, 32, 93, 0
7	163,317	108,000	0, 14, 47, 59, 75, 97, 74, 24, 0
8	152,811	245,000	0, 72, 36, 39, 42, 44, 41, 38, 40, 43, 37, 35, 54, 68, 0
9	150,262	192,000	0, 88, 98, 53, 99, 57, 86, 87, 9, 10, 13, 17, 0
10	135,939	206,000	0, 92, 62, 31, 29, 27, 28, 26, 34, 50, 96, 80, 0
11	1 408,62		

TAB. 5 – Meilleure solution des huit problèmes (partie 1)

RC201 (germe 1)

j	δ_j	D_j	
0	359,108	396,000	0, 92, 95, 63, 33, 31, 29, 27, 28, 30, 62, 67, 71, 40, 38, 41, 81, 90, 94, 50, 34, 26, 32, 96, 54, 93, 0
1	359,759	379,000	0, 72, 64, 52, 23, 21, 76, 18, 19, 22, 51, 85, 84, 49, 57, 20, 66, 55, 68, 56, 91, 80, 0
2	483,209	494,000	0, 42, 36, 39, 5, 45, 47, 75, 16, 15, 11, 12, 73, 79, 7, 6, 8, 46, 3, 37, 35, 43, 1, 4, 60, 17, 70, 100, 0
3	560,879	455,000	0, 14, 59, 83, 65, 82, 69, 98, 2, 44, 61, 88, 99, 9, 87, 86, 53, 78, 97, 10, 13, 74, 24, 89, 48, 25, 77, 58, 0
4	1762,955		

RC202 (germe 3)

j	δ_j	D_j	
0	310,202	460,000	0, 65, 91, 92, 95, 85, 63, 33, 34, 31, 29, 27, 26, 28, 30, 50, 67, 71, 94, 81, 41, 40, 43, 35, 72, 54, 68, 93, 96, 80, 0
1	435,384	467,000	0, 42, 37, 36, 39, 1, 45, 44, 38, 61, 90, 53, 78, 79, 8, 46, 6, 7, 5, 3, 4, 2, 55, 60, 24, 25, 77, 58, 0
2	348,318	305,000	0, 69, 62, 64, 19, 23, 18, 76, 51, 84, 56, 66, 10, 32, 89, 0
3	420,814	492,000	0, 82, 83, 52, 12, 47, 14, 11, 16, 15, 73, 88, 99, 9, 87, 59, 86, 57, 22, 20, 49, 48, 21, 75, 97, 13, 74, 17, 98, 100, 70, 0
4	1514,718		

RC203 (germe 2)

j	δ_j	D_j	
0	312,092	499,000	0, 61, 45, 1, 42, 39, 37, 36, 44, 43, 40, 38, 41, 2, 6, 7, 8, 46, 4, 5, 3, 35, 72, 54, 81, 96, 71, 93, 94, 0
1	324,729	395,000	0, 69, 12, 11, 15, 73, 88, 99, 49, 22, 10, 16, 47, 17, 13, 74, 86, 58, 77, 25, 57, 52, 0
2	332,350	418,000	0, 91, 92, 95, 50, 34, 28, 32, 33, 26, 27, 29, 31, 30, 62, 64, 76, 67, 84, 51, 89, 63, 85, 56, 66, 90, 68, 80, 0
3	244,025	412,000	0, 65, 83, 20, 24, 19, 18, 48, 21, 23, 75, 97, 59, 87, 9, 14, 78, 79, 60, 53, 82, 98, 55, 100, 70, 0
4	1213,197		

RC204 (germe 2)

j	δ_j	D_j	
0	411,211	732,000	0, 66, 56, 64, 24, 22, 20, 49, 19, 18, 48, 21, 89, 63, 85, 51, 76, 23, 75, 97, 59, 87, 9, 10, 12, 60, 73, 78, 14, 47, 17, 16, 13, 52, 86, 74, 58, 77, 25, 57, 83, 65, 90, 0
1	300,071	620,000	0, 80, 91, 92, 94, 71, 72, 37, 35, 36, 40, 43, 44, 42, 39, 38, 41, 54, 99, 82, 53, 79, 7, 8, 46, 45, 5, 3, 1, 4, 6, 2, 70, 55, 100, 88, 98, 0
2	257,489	372,000	0, 69, 15, 11, 62, 67, 50, 33, 32, 30, 28, 26, 27, 29, 31, 34, 84, 95, 93, 96, 81, 61, 68, 0
3	968,771		

TAB. 6 – Meilleure solution des huit problèmes (partie 2)

RC205 (germe 0)

j	δ_j	D_j	
0	502,585	365,000	0, 72, 36, 39, 42, 45, 69, 61, 44, 71, 67, 38, 40, 41, 81, 53, 86, 74, 17, 13, 24, 25, 77, 58, 0
1	381,021	552,000	0, 82, 12, 14, 47, 16, 15, 11, 73, 88, 7, 5, 3, 2, 6, 8, 79, 78, 98, 90, 91, 50, 32, 26, 34, 94, 96, 54, 70, 100, 80, 0
2	325,005	381,000	0, 92, 95, 31, 29, 27, 28, 33, 62, 30, 63, 85, 76, 18, 21, 19, 22, 84, 51, 49, 20, 66, 56, 68, 55, 0
3	426,605	426,000	0, 65, 83, 64, 23, 57, 52, 99, 9, 87, 59, 75, 97, 10, 60, 46, 1, 4, 43, 35, 37, 93, 89, 48, 0
4	1635,217		

RC206 (germe 0)

j	δ_j	D_j	
0	386,636	472,000	0, 2, 45, 5, 42, 72, 39, 36, 40, 38, 44, 41, 61, 81, 71, 94, 96, 50, 34, 32, 26, 89, 48, 25, 77, 24, 0
1	295,537	490,000	0, 83, 82, 69, 98, 11, 15, 16, 47, 14, 12, 88, 53, 78, 73, 79, 7, 6, 8, 46, 4, 3, 1, 43, 35, 37, 54, 93, 80, 91, 0
2	305,127	399,000	0, 65, 52, 59, 75, 21, 23, 19, 49, 22, 51, 76, 18, 20, 85, 84, 56, 66, 55, 68, 70, 100, 60, 0
3	327,199	363,000	0, 92, 95, 31, 29, 27, 30, 28, 33, 63, 62, 67, 64, 90, 99, 57, 86, 87, 9, 10, 17, 13, 97, 58, 74, 0
4	1314,500		

RC207 (germe 1)

j	δ_j	D_j	
0	169,331	179,000	0, 82, 53, 99, 52, 86, 75, 59, 87, 57, 66, 90, 91, 80, 0
1	385,693	628,000	0, 65, 83, 23, 21, 76, 63, 33, 30, 28, 27, 29, 31, 85, 51, 22, 19, 49, 18, 25, 77, 58, 74, 97, 13, 9, 10, 17, 60, 4, 1, 70, 100, 55, 68, 0
2	359,263	448,000	0, 64, 92, 95, 62, 67, 71, 72, 41, 38, 36, 40, 44, 42, 61, 81, 94, 84, 50, 34, 26, 32, 89, 48, 24, 20, 56, 93, 0
3	254,987	469,000	0, 69, 88, 98, 47, 14, 11, 15, 16, 12, 78, 73, 79, 7, 8, 2, 45, 5, 3, 46, 6, 43, 37, 35, 39, 54, 96, 0
4	1169,273		

RC208 (germe 2)

j	δ_j	D_j	
0	235,941	485,000	0, 90, 61, 81, 94, 92, 95, 67, 93, 71, 72, 41, 42, 44, 39, 38, 37, 35, 36, 40, 43, 1, 3, 5, 46, 4, 6, 70, 100, 55, 68, 0
1	457,118	792,000	0, 69, 98, 12, 14, 47, 16, 15, 11, 59, 75, 23, 21, 18, 19, 22, 51, 76, 89, 33, 30, 28, 26, 27, 29, 31, 34, 32, 50, 62, 84, 85, 63, 48, 49, 20, 24, 25, 77, 58, 74, 97, 13, 17, 60, 0
2	273,884	447,000	0, 65, 64, 83, 52, 99, 82, 78, 73, 79, 7, 8, 45, 2, 88, 53, 10, 9, 87, 86, 57, 66, 56, 91, 80, 96, 54, 0
3	966,943		

TAB. 7 – Variation de la solution en fonction des germes

Problème	Meilleur				Pire				Moyen	
	Germe	R	δ	Temps	Germe	R	δ	Temps	R	δ
RC201	1	4	1762,955	1min29s	0	5	1668,305	1min34s	4,60	1742,508
RC202	3	4	1514,718	1min44s	1	4	1644,766	1min40s	4,00	1586,226
RC203	2	4	1213,197	1min48s	4	4	1351,316	1min29s	4,00	1268,335
RC204	2	3	968,771	2min20s	0	3	1185,983	2min07s	3,00	1054,378
RC205	0	4	1635,217	1min53s	3	5	1596,538	1min37s	4,20	1779,101
RC206	0	4	1314,500	1min46s	3	4	1415,491	1min46s	4,00	1355,837
RC207	1	4	1169,273	1min51s	3	4	1400,390	1min54s	4,00	1278,732
RC208	2	3	966,943	3min29s	1	3	1083,139	3min17s	3,00	1026,544

TAB. 8 – Meilleures solutions trouvées par des heuristiques

Problème	Solution	
	R	δ
RC201	4	1406,91
RC202	3	1367,09
RC203	3	1049,62
RC204	3	798,41
RC205	4	1297,19
RC206	3	1146,32
RC207	3	1061,14
RC208	3	828,14

A Organisation du programme

Le programme implantant l'heuristique est divisé en plusieurs paquetages, chaque paquetage comprenant un certain nombre de classes Java. Le paquetage `data` regroupe les classes nécessaires à la description d'un problème et de solutions et implante les heuristiques nécessaires pour insérer un client dans une solution, supprimer un client et faire des échanges de segments. Ces implantations vérifient que les opérations ne produisent pas de solutions non réalisables, permettant une plus grnde fiabilité d'implantation de la recherche tabou.

Le paquetage `solbuilding` implante les heuristiques de construction ou de complétion de solutions. Il comporte une classe pour l'heuristique d'insertion et une autre classe pour celle des gains.

Le paquetage `tabu` implante quant à lui la recherche tabou ainsi que sa généralisation avec plusieurs fils d'exécution. Il définit également un type d'observateur permettant de suivre la progression de la recherche tabou améliorée au cours de son exécution. Un tel observateur a été utile pour permettre le suivi de la recherche dans une fenêtre graphique sans lier le code d'interface graphique avec le code d'implantation de la recherche tabou.

Finalement, le paquetage `gui` comprend une interface graphique destinée à faciliter l'utilisation du programme et à examiner les solutions produites. Cette interface a entre autre permis de détecter que les échanges de segments produisaient des solutions non réalisables et nous avons pu corriger le problème dans le programme.

La plupart des classes sont documentées avec des commentaires Javadoc. Nous avons choisi l'Anglais comme langue de documentation afin d'éviter tout problème d'encodage des accents dans les fichiers Java.

B Guide d'utilisation du programme

L'exécution du programme nécessite Java 5. Il a été testé avec l'environnement d'exécution de Sun (*Java Runtime Environment* (JRE)) 1.5.0 mise à jour 6, sous Fedora Core 4 x86_64 et sous Microsoft Windows XP. Les fichiers compilés du programme sont dans une archive JAR qui peut être exécutée directement avec la commande

```
java -jar lib/tabuvrsptw.jar
```

L'archive `ssj.jar` doit se trouve dans le même répertoire que `tabuvrsptw.jar`, mais il n'est pas nécessaire de mettre quelqu'archive que ce soit dans le `CLASSPATH` pour une simple exécution. La ligne de commande ci-haut fait apparaître une fenêtre semblable à la figure 1.

Tout d'abord, le menu **Look and Feel** permet de modifier l'apparence de l'interface. En particulier, l'option **Platform look and feel** utilise des fenêtres semblables à celes du système d'exploitation tandis que **Cross-platform look and feel** a été utilisé pour les figures. L'apparence par défaut est **Cross-platform look and feel with large fonts** qui est une variante de l'apparence précédente avec des caractères plus gros.

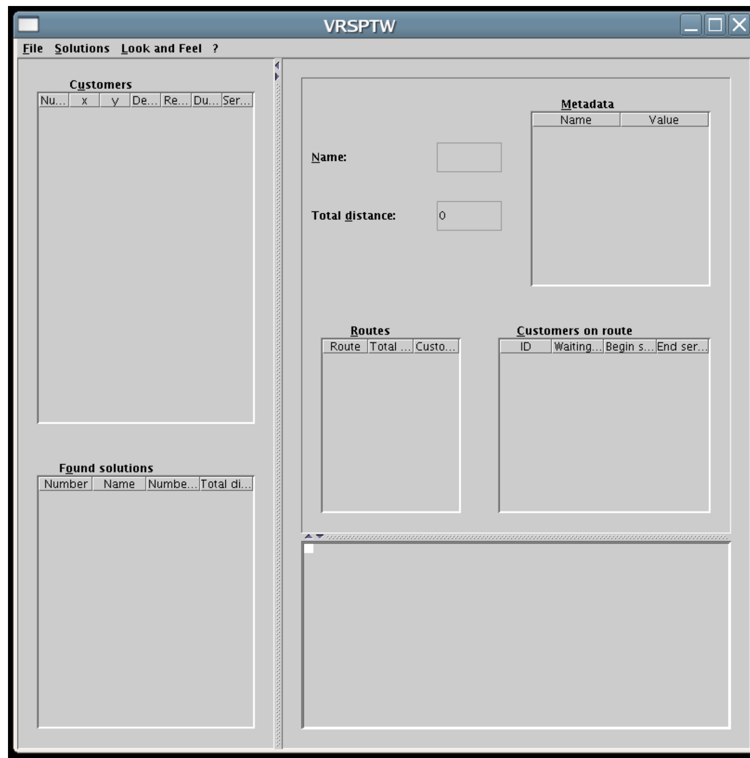

FIG. 1 – Fenêtre principale du programme

Pour commencer une recherche tabou, il faut d'abord charger un problème encodé sous la forme d'un fichier texte. La figure 2 montre les dix premières lignes du problème RC201, encodé de façon à être compatible avec le programme. Les lignes débutant par des dièses sont ignorées et la capacité des véhicules doit être donnée au début du fichier. Chaque ligne subséquente correspond à un client et chaque colonne est obligatoire.

Lorsqu'un problème est chargé, le tableau supérieur gauche affiche la liste des clients. Le tableau inférieur gauche, quant à lui, sert à afficher les solutions trouvées jusque-là. Le menu **Solutions** permet d'insérer ou de supprimer des solutions de cette liste.

Par exemple, la commande **Insertion heuristic** applique l'heuristique de Solomon après avoir demandé à l'utilisateur d'entrer les valeurs des paramètres dans une boîte de dialogue semblable à la figure 3. Lorsqu'au moins une solution est disponible, la partie droite de la fenêtre principale affiche ses constituants. En particulier, toutes les routes construites sont affichées, avec leurs clients.

La partie inférieure droite de la fenêtre affiche la solution sous forme graphique. Il est possible d'agrandir l'image de la solution en cliquant du bouton gauche sur cette dernière ou de la rétrécir en cliquant du bouton droit. Malheureusement, il n'a pas été possible de contraindre l'image à occuper tout l'espace disponible sans désactiver la fonction de zoom. La figure 4 montre un exemple de fenêtre avec une solution et une image. Sur l'image, la route sélectionnée dans la liste est affichée

```
#Vehicles' capacity
1000
#CUST NO.   XCOORD.    YCOORD.    DEMAND     READY TIME   DUE DATE    SERVICE TIME
    1       40.00      50.00        0.00        0.00      960.00        0.00
    2       25.00      85.00       20.00      673.00      793.00       10.00
    3       22.00      75.00       30.00      152.00      272.00       10.00
    4       22.00      85.00       10.00      471.00      591.00       10.00
    5       20.00      80.00       40.00      644.00      764.00       10.00
    6       20.00      85.00       20.00       73.00      193.00       10.00
    7       18.00      75.00       20.00      388.00      508.00       10.00
    8       15.00      75.00       20.00      300.00      420.00       10.00
    9       15.00      80.00       10.00      367.00      487.00       10.00
   10       10.00      35.00       20.00      371.00      491.00       10.00
   11       10.00      40.00       30.00      519.00      639.00       10.00
```

FIG. 2 – Exemple d'une description de problème

en rouge tandis que les autres sont en noir. Un cercle vert représente un point où le véhicule doit attendre avant de servir le client ; cela représente un délai qui peut parfois être exploité pour insérer d'autres clients. Un cercle bleu, non présent sur la figure, représente un client qui est servi exactement au temps e_i. Un cercle jaune représente un client servi après e_i mais avant l_i. Un cercle rouge représenterait une arrivée de véhicule après la fenêtre de temps et donc un bogue dans le programme ! Heureusement, il n'y en a pas sur la figure. Tout client non inclus dans une route sera représenté par un cercle noir. Le dépôt est quant à lui représenté par un carré noir.

La commande **Tabu Search** du menu **Solutions** permet de déclencher une recherche tabou simple sur la solutoin initiale sélectionnée. L'utilisateur devra alors entrer le nombre maximal d'itérations, le nombre d'itérations pendant lequel un échange est tabou, etc. en utilisant la boîte de dialogue montrée sur la figure 5. Le résultat de la recherche apparaît ensuite dans la liste des solutions.

L'heuristique de recherche tabou avec solutions initiales multiples est disponible par le biais de la commande **Improved tabu search** du menu **Solutions**. Une boîte de dialogue semblable à celle de la figure 6 permet de paramétrer la recherche.

Cliquer sur le bouton **Ok** démarre la recherche et affiche une nouvelle boîte de dialogue indiquant la progression du travail des différents fils d'exécution. Cette boîte est montrée sur la figure 7. On y voit, en plus de l'état des fils d'exécution, le contenu de la mémoire adaptative mis à jour en temps réel. À tout moment, il est possible de double-cliquer sur une solution pour en examiner les détails (il faut double-cliquer sur une autre colonne que **Name**) ou la sauvegarder dans un fichier. À tout moment, le bouton **Abort** permet d'interrompre la recherche tandis que le bouton **Close**, qui s'active à la fin du travail, ferme la fenêtre et ajoute la meilleure solution trouvée à la liste des solutions de la fenêtre principale.

Il est également possible de charger et de sauvegarder des solutions de la liste des solutions courantes. Le format d'une solution est textuel, comme celui des problèmes. La figure 8 présente

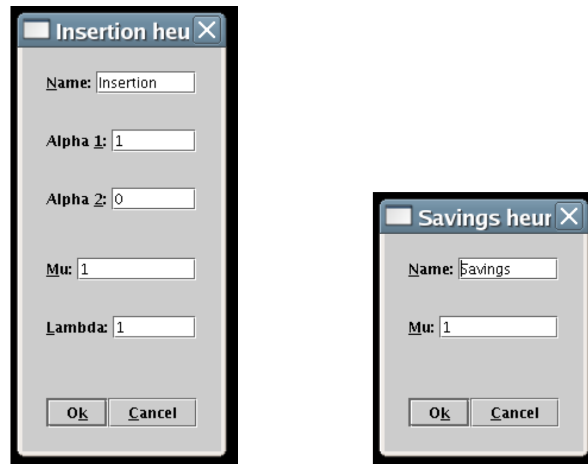

FIG. 3 – Paramètres des heuristiques d'insertion et de gains

un exemple d'une solution du problème RC201. Les barres obliques représentent des fins de ligne typographiques ; elles ne sont pas présentes dans le fichier de solution. Il faut également noter que le dépôt est omis dans les routes du fichier.

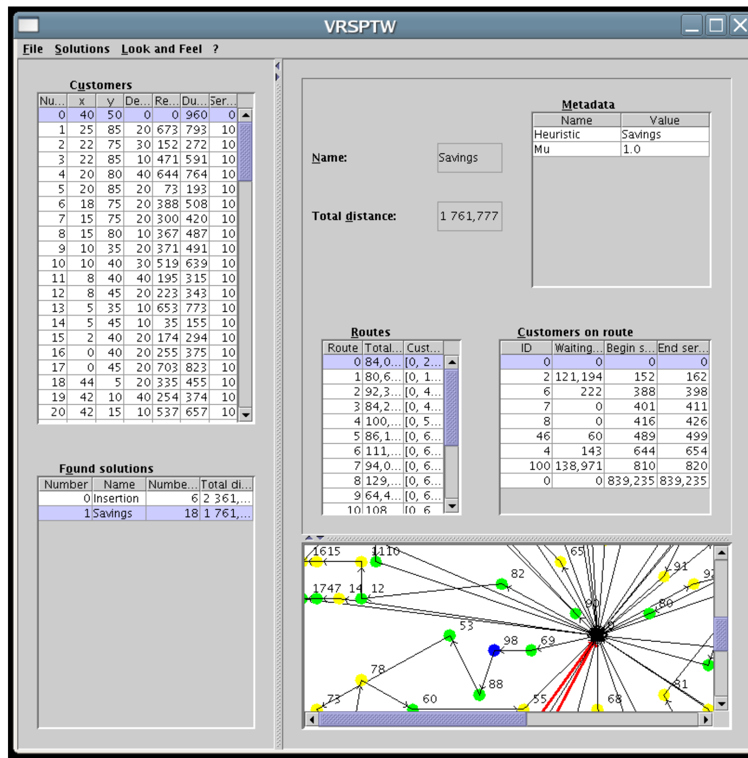

FIG. 4 – Exemple d'une solution affichée dans le programme

Tabu search parameters

Name: Tabu

Maximal number of iterations: 500

Maximal number of iterations the solution does not improve: 50

Number of iteration exchanges remain tabu: 15

Allow full (deterministic) search

Maximal number of iterations during intensification: 100

Maximal number of iterations the solution does not improve during intensification: 10

Random number seed

s1: 12 345 s4: 12 345

s2: 12 345 s5: 12 345

s3: 12 345 s6: 12 345

Generate seed

Ok Cancel

FIG. 5 – Paramètres de la recherche tabou simple

Improved tabu search parameters

Name: Tabu

General parameters

Maximal number of iterations: 50

Maximal number of iterations without improvement: 10

Size of adaptive memory: 30

Solution building from adaptive mem

Solution shuffle   Route shuffle

Initial solutions parameters

Number of initial solutions: 10

Probability of insertion: 1

Mu: 1

Lambda: 1

Mu for Savings: 1

Tabu search parameters

Maximal number of iterations: 500

Maximal number of iterations without improvement: 50

Number of tabu iterations: 15

Allow full (deterministic) search

Intensification

Maximal number of iterations: 100

Maximal number of iterations without improvement: 10

Random number seed

s1: 12 345   s4: 12 345

s2: 12 345   s5: 12 345

s3: 12 345   s6: 12 345

Generate seed

Ok   Cancel

FIG. 6 – Paramètres de la recherche tabou améliorée

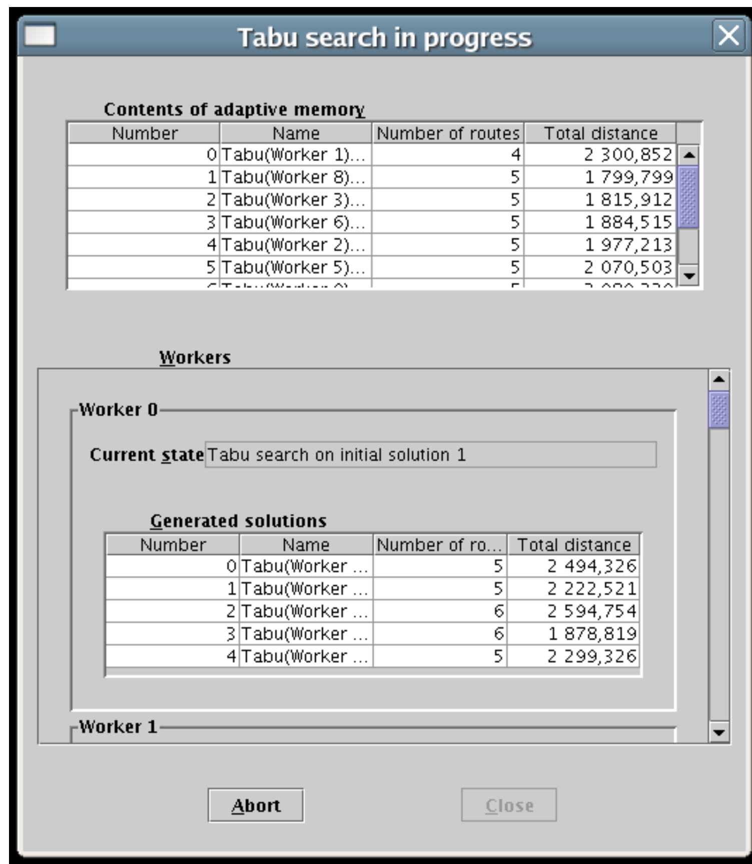

FIG. 7 – Fenêtre d'état de la recherche tabou améliorée

```
"Tabu(Worker 1)_InitialTabu"
# Heuristic=Insertion1
# Alpha1=0.13598841039594017
# Alpha2=0.8640115896040599
# Mu=1.40998165157827
# Lambda=0.8569735971530801
# Tabu_Heuristic=Tabu Search
# Tabu_InitialSolution=Tabu(Worker 1)_Initial
# Tabu_NbMaxIter=500
# Tabu_NbMaxUnimproved=50
# Tabu_NbTabuIter=15
# Tabu_NbIter=2
# Tabu_NbImproved=1
# ITabu_Heuristic=Tabu Search
# ITabu_InitialSolution=Tabu(Worker 1)_Initial
# ITabu_NbMaxIter=100
# ITabu_NbMaxUnimproved=10
# ITabu_NbTabuIter=15
# ITabu_NbIter=2
# ITabu_NbImproved=2
# cpuTime=0:34.410886
# mrg_s1=12345
# mrg_s2=12345
# mrg_s3=12345
# mrg_s4=12345
# mrg_s5=12345
# mrg_s6=12345
[65, 59, 14, 47, 5, 2, 44, 64, 76, 30, 51, 85, 90, 46, 3, 4, 60, 91, \
93, 80, 70, 100]
[45, 36, 39, 42, 72, 28, 27, 29, 31, 62, 67, 71, 61, 38, 40, 41, 81, \
94, 84, 50, 34, 26, 32, 96, 54, 35, 1, 58, 77]
[63, 95, 92, 83, 52, 75, 23, 19, 21, 18, 22, 57, 99, 86, 87, 9, 49, \
37, 43, 68, 13, 74, 24, 25, 48]
[33, 15, 11, 82, 98, 69, 88, 12, 16, 73, 79, 7, 8, 6, 78, 53, 10, 97, \
20, 56, 66, 55, 17, 89]
```

FIG. 8 – Exemple de solution au problème RC201