

Implantation d'heuristiques de résolution du PVC

Projet d'optimisation combinatoire

Matthieu Nottale

1. Introduction

L'objectif de ce projet est d'implanter quelques algorithmes et heuristiques résolvant au mieux le problème du voyageur du commerce.

1.1. Présentation du problème

Le problème du voyageur de commerce (PVC) peut s'énoncer comme suit: étant donné N villes et les distances qui les séparent, trouver le plus court chemin passant une et une seule fois par chaque ville, et revenant à son point de départ. Ce problème fait parti de la catégorie des problèmes NP-complets, c'est à dire qu'aucun algorithme le résolvant en temps polynomial n'a été trouvé, mais que ni l'existence ni la non-existence d'un tel algorithme n'a été prouvé.

1.2. Travail réalisé

Le programme réalisé présente une interface en mode texte afin de sélectionner une instance du problème du PVC, lui appliquer une heuristique de résolution parmi celles implantées et afficher le résultat. Un affichage graphique permet de voir la solution en cours de construction. Les heuristiques implantées sont: "insertion du plus proche voisin", "insertion au moindre cout", "Parcours prefixe d'un arbre recouvrant de poid minimal", "Voisinages 2-Opt", "recuit simulé", "élastique". De plus une méthode de résolution exacte, "séparation-évaluation" est aussi disponible. Le programme est écrit en C++, et fait environ 2000 lignes de code.

2. Utilisation

L'interface est en ligne de commande. Une documentation suffisente est disponible "en-ligne" en tapant '?' ou 'help'. A titre d'exemple, une utilisation typique du programme est:

```
Taper "?" pour avoir la liste des commandes
```

```
>random 10 100 10
```

```
Coordonnees des sommets:
0: 800,420

...

99: 120,340

>output 1

>solve H2Opt

execution de H2Opt(10000)
H2Opt: algorithme de voisinage 2-Opt, démarrage
H2Opt: construction d'un chemin initial
HPlusProcheVoisin: algorithme du plus proche voisin, graphe de 100 sommets
HPlusProcheVoisin: fin, circuit de longueur 7930 trouve
H2Opt: initialisation
H2Opt: boucle principale
H2Opt: fin, circuit de longueur 6820 trouve

>view 0

>show

Id Algorithme      Chemin  Temp
0              H2Opt    6820    19

>x
```

3. Le Code

3.1. Introduction

Le code se divise en trois grandes parties: 'main.cpp' contient le code de l'interface en ligne de commande, 'graphe.cpp' contient les structures de données pour représenter et manipuler un ensemble d'arêtes, une instance du problème ainsi que le code de l'affichage graphique (qui utilise OpenGL), 'algo.cpp' contient le code spécifique de chaque heuristique. De plus du code général sur les listes chaînées, utilisé très fréquemment, se trouve dans lc.cpp et lc.hpp (fonctions génériques: 'templates'), une implantation d'arbre n-aire dans xtree.cpp et xtree.hpp, et un procédé de synchronisation étendant les sémaphores dans synchronizer.cpp. Le reste de ce document est consacré au détail de chacune de ces parties.

3.2. Conventions

On adoptera dans la suite les conventions suivantes:

- N désignera le nombre de villes de l'instance du problème considérée.
- Les termes algorithme et heuristiques seront parfois employés pour désigner l'ensemble des heuristiques et algorithmes implantés.

3.3. Main.cpp

Ce fichier contient la fonction main, qui se contente d'appeler en boucle la fonction command avec comme paramètre une ligne de stdin.

La fonction commande execute les commandes de l'utilisateur en appelant les fonctions correspondantes, en tenant à jour un état via des variables globales (option, instance du problème, résultats déjà obtenus). Cette partie du code ne présente aucune difficulté ou intérêt particulier, et ne sera donc pas détaillé.

3.4. Graphe.cpp

Ce fichier contient des classes utilisés par tous les algorithmes pour manipuler les données relatives au PVC.

3.4.1. Représentation d'une instance du problème

Graphe est la classe de base pour représenter une instance du problème. Les distances entre les villes sont stockés dans un "demi-tableau", ce programme ne traitant que les problèmes symétriques.

EuclidianGraphe hérite de Graphe. Cette classe gère en plus les coordonnées des villes. Elle offre une fonction permettant de générer une instance du problème aléatoirement, et une fonction permettant d'afficher graphiquement ces villes, ainsi qu'un ensemble d'arêtes les reliant. Les coordonnées des villes étant stockés sous forme d'entiers, de même que les distances, il faut prendre garde à ce que les approximations qui en résultent lors du calcul des distances à partir des coordonnées ne mettent pas en défaut l'inégalité triangulaire. Pour éviter ce problème, un paramètre permet de multiplier les coordonnées générés par un facteur donné, avant le calcul des distances.

3.4.2. Représentation d'un ensemble d'arêtes

Tous les algorithmes effectuant des manipulations sur les arêtes, il était nécessaire d'implanter une classe de gestion d'arêtes, ne consommant pas trop de mémoire (un tableau de N par N booléens n'est pas une solution sérieusement envisageable), et permettant l'ajout, la suppression et la consultation rapide de ses

arêtes. La solution retenue utilise une double liste chaînée: chaque arête $(s1, s2)$ $s1 < s2$ est dans une liste chaînée de toutes les arêtes de même $s1$, triée selon $s2$, ainsi que dans une liste chaînée de toutes les arêtes de même $s2$, triée selon $s1$. Des pointeurs sur le premier élément de ces $2*(N-1)$ listes chaînées sont conservés. L'espace mémoire utilisé est ainsi proportionnel au nombre d'arêtes, plus une 'constante' proportionnelle à N . Le temps d'ajout/suppression, proportionnel à N dans le cas général (parcours d'une liste chaînée) est en pratique quasiment constant, car les ensembles d'arêtes utilisés par les algorithmes ont une arité très faible (2 pour tous les algorithmes ne travaillant que sur des chemins). La classe permettant de manipuler cette structure s'appelle `tArêtes`, et implante les opérations de base (ajout, suppression, test) ainsi que deux fonctions permettant d'exporter l'ensemble d'arêtes sous forme de tableau ou d'une liste chaînée circulaire dans le cas où cet ensemble définit un chemin. La classe `tArête` représente une arête dans une instance de `tArêtes`, et expose un certain nombre de fonctions spécifiques à l'algorithme `2Opt` qui seront détaillées plus bas.

3.5. Algo.cpp

Tous les algorithmes ont pour fonction principale une fonction prenant comme paramètre un `EuclidianGraphe` représentant l'instance du problème, et un `unsigned long` indiquant des options d'affichage et de sortie texte (définitions dans `algo.h`), et retournant un `tArêtes` représentant la solution trouvée. La structure `Algorithmes`, tableau de `PVC_Algo`, contient la liste des algorithmes, avec leur nom, une description, et un pointeur sur la fonction principale. Le nom des fonctions principales de chaque algorithme est préfixé par `PVC_H` ou `PVC_A` selon qu'il s'agisse d'une heuristique ou d'une méthode exacte.

La partie du paramètre option concernant l'affichage est un entier codé sur 16 bits, indiquant le temps à attendre entre chaque affichage, par tranche de 10 millisecondes. Une valeur de 0 signifie "désactiver l'affichage".

La partie concernant la sortie texte prend ses valeurs parmi "None", "Important", "Detailed", "UltraDetailed", "Verbose" et "Debug". Typiquement ces valeurs limitent la profondeur maximale à laquelle le code est autorisé à générer un message. Ainsi pour un algorithme s'exécutant en $O(n*n)$, des valeurs de Important, Detailed et UltraDetailed généreront respectivement de l'ordre de $O(1)$, $O(n)$ et $O(n*n)$ messages.

Certaines heuristiques (Recuit simulé, Elastique) utilisent aussi une partie du champ option, passé en paramètre lors de l'appel à "solve", pour paramétrer leur fonctionnement.

3.5.1. PVC_HMoindreCout

Cette heuristique part d'un circuit contenant un seul sommet, et insère à chaque étape le sommet rallongeant le moins possible le circuit.

L'implantation est assez triviale. Les structures utilisés sont un tableau de booléens permettant de tenir compte des sommets déjà insérés, et une liste chaînée circulaire décrivant le sous-circuit en cours, qui

permet de parcourir facilement tous les points d'insertion possibles.

3.5.2. PVC_HPlusProcheVoisin

Cette heuristique triviale part du sommet 0 et insère à chaque étape le sommet le plus proche du dernier sommet inséré.

3.5.3. PVC_HARPM

Cette heuristique consiste à construire le circuit en gardant la première occurrence de chaque sommet lors d'un parcours préfixe d'un arbre recouvrant de poids minimal. L'arbre recouvrant de poids minimal est créé en utilisant l'algorithme de Prim. Puis une fonction récursive, `gen_parcours_prefixe`, est appelée sur un sommet quelconque de cet arbre. Cette fonction prend comme paramètre un arbre, sous forme d'un `tArêtes`, un buffer qui contiendra le parcours préfixe sans redondance, la position actuelle dans le buffer, le sommet actuel et le sommet d'où l'on vient. Elle a pour effet de rajouter le sommet actuel à la fin du buffer, et de s'appeler récursivement pour tous ses sommets fils, c.a.d. tous les sommets qui lui sont reliés par une arête et qui ne sont pas le sommet père. Puis le sommet initial est rajouté à la fin du buffer pour former un circuit, qui est finalement reconverti en `tArêtes`.

Cette heuristique produit généralement de très mauvais résultats, plus mauvais que les deux précédentes, mais elle a le mérite de garantir que la solution trouvée ne sera pas plus de deux fois plus longue que la solution optimale.

3.5.4. PVC_H2Opt

Cette heuristique fonctionne de la manière suivante: on part d'un chemin quelconque. A chaque étape, on considère le voisinage du chemin actuel défini par l'ensemble des chemins que l'on peut engendrer en permutant deux arêtes. Si tous les chemins du voisinage sont plus long que le chemin actuel, l'heuristique s'arrête. Sinon le chemin actuel est remplacé par le chemin le plus court du voisinage, et le processus se répète.

On constate que, lorsque les deux arêtes sont permutées, une partie du chemin va être parcouru dans le sens inverse. Ainsi que l'on stocke le chemin dans un tableau ou dans une liste chaînée circulaire, le temps nécessaire à la permutation de deux arêtes est en $O(n)$.

Cependant il est possible de concevoir une structure de donnée pour laquelle la permutation de deux arêtes s'effectue en temps constant: une liste chaînée circulaire complétée par un tableau indiquant pour chaque sommet s'il s'agit d'un point d'inversion du parcours.

Concrètement, si on dispose des variables "sensactuel" indiquant si le sens de parcours actuel est inversé, "sommetactuel" le sommet actuel dans la liste chaînée, et du tableau `estpointdinversion`, l'algorithme pour passer au sommet suivant s'écrit:

```
si sensactuel est inverse alors prochainsommet=sommeactuel->precedent  
sinon prochainsommet=sommetactuel->suivant  
si estpointdinversion[prochainsommet] alors sensactuel= non sensactuel
```

L'algorithme de permutation de deux arêtes consiste à effectuer la permutation dans la liste chaînée, et à mettre à jour les champs du tableau correspondant à deux des quatre sommets en jeu. Cette opération, bien qu'un peu complexe, s'effectue en un temps constant. Se reporter au code pour les détails de son fonctionnement.

Une fois la structure de donnée représentant l'information trouvée, le reste de l'implantation de cette heuristique est trivial.

3.5.5. PVC_HRecuitSimule

Cette heuristique fondée sur une analogie physique fonctionne sur le même principe que le 2-Opt. A chaque étape une paire d'arêtes est tirée au hasard. Si la permutation engendrée diminue la longueur du circuit, elle est conservée. Sinon, la probabilité qu'elle soit conservée est égale à $\exp(-d/KT)$, d étant la différence de longueur entre les chemins, K une constante, et T un paramètre qui diminue au fur et à mesure des itérations.

Le nombre d'itérations par valeur de T , ainsi que le nombre de valeurs prises par T , sont paramétrables via la partie du paramètre "option" réservée à cet effet.

Les structures de donnée utilisées sont les mêmes que pour le 2-Opt.

On définit une itération par une permutation acceptée, ou alors l'ensemble des couples d'arêtes testés sans qu'aucune ne soit acceptée, ce qui a pour effet d'augmenter le temps de calcul quand T diminue, mais donne de meilleurs résultats que si on avait considéré qu'une itération correspondait à une permutation testée.

Cette heuristique donne généralement de meilleurs résultats que le 2-Opt, car elle permet d'éviter de rester piégé dans un minimum local, mais le temps de calcul est nettement plus important: le gain se situe entre 3% et 10% pour des problèmes de 50 à 500 sommets, alors que le temps de calcul est multiplié par 100 à 1000!

3.5.6. PVC_ASeparationEvaluation

Il s'agit de la seule méthode donnant une solution exacte implantée ici. Elle procède en parcourant l'arbre des solutions par niveau, calculant à chaque sommet la borne inférieure des solutions qui lui sont issues, et élagant ce sommet si cette borne inférieure est supérieure à une solution connue.

La structure de donnée définissant un noeud de l'arbre des solutions est une classe, *Arbre*, dérivant de *xTree*. Chaque instance dispose d'un pointeur vers le noeud père, le premier noeud fils, le frère suivant et le frère précédent, ce qui permet de créer des arbres d'arité quelconque. L'implantation de cette heuristique est divisée en trois fonctions:

La fonction principale se contente d'initialiser les structures de donnée: l'arbre contient initialement un seul noeud, correspondant au sommet 0. Une première solution est cherchée à l'aide d'une heuristique simple (2 Opt) afin d'améliorer l'élégation. Puis elle appelle la fonction de parcours tant que celle-ci renvoie true, ce qui signifie qu'une expansion est toujours possible.

La fonction de parcours étend l'arbre d'un niveau: elle parcourt l'arbre de façon récursive, en élagant tout noeud dont la borne inférieure est supérieure à la meilleure solution connue. A chaque fois qu'elle tombe sur un noeud non développé, elle le développe en créant tous les fils correspondants à une extension du sous-chemin possible, et en appelant la fonction d'évaluation sur chacun de ces fils. Elle renvoie false si aucune expansion n'a été faite, la gestion de la valeur renvoyée étant telle que la valeur finalement renvoyée à la première fonction appelante soit le OU de toutes les valeurs renvoyées par chacun des appels récursifs.

La fonction d'évaluation prend comme paramètre un sous-chemin, défini par l'ensemble des sommets déjà atteints, et renvoie une borne inférieure du plus court sous-chemin étendant ce dernier en chemin valide, ainsi que si cette borne est atteinte ou non. Cette borne est définie par $\text{Somme}(\text{Min}\{w(i,j), j=1..N\}, i \text{ non encore atteint})$. Le code procède de la manière suivante: il regarde si $M = \text{Min}(w(i, \text{currentsommet}), i=1..N)$ est atteint pour au moins un sommet non encore atteint. Si c'est le cas, la fonction s'appelle récursivement sur chacun de ces sommets, renvoie comme borne $M + \text{la valeur renvoyée par l'une de ces fonctions}$ (qui est par définition la même pour toutes). De plus la borne est atteinte si au moins une des fonctions appelées récursivement a renvoyé qu'elle était atteinte.

Dans le cas contraire, c'est à dire si le minimum est atteint en un sommet déjà atteint, alors la borne n'est pas atteignable, et sa valeur est calculée plus rapidement par la formule ci-dessus.

Afin d'accélérer quelque peu, le cas où le minimum des $w(i, \text{currentsommet})$ est atteint pour un seul sommet, cas le plus fréquent car l'égalité de deux distances est rare, est traité de manière itérative.

Malgré le processus d'élégation, la mémoire et le temps de calcul nécessaires à cet algorithme sont tels qu'il est impossible de l'utiliser sur des problèmes dépassant la dizaine de sommets. Bien sûr on ne peut raisonnablement le comparer aux autres heuristiques présentées ici, car il fournit la solution optimale, contrairement aux heuristiques qui, pour la plupart, n'offrent aucune garantie sur la qualité de la solution trouvée.

3.5.7. PVC_HElastique

Autre méthode basée sur la simulation d'un modèle physique, la méthode de l'élastique simule un élastique soumis à une force d'attraction de chacun de ses sommets vers chacune des villes, ainsi qu'à

une force d'attraction entre les sommets voisins, ayant pour effet de diminuer sa longueur.

Les paramètres utilisés sont:

- $M=N$ sommets dans l'élastique initialement, puis augmentation jusqu'à $2.5*N$
- $K=0.2$ à une valeur < 0.02 , décrémentation par pas configurable, défaut 0.98
- Nombre d'itérations par valeur de K configurable, défaut $n=20$
- $\text{Alpha}=0.2$
- $\text{Beta}=2.0$

Le grand avantage de cet algorithme est qu'il est massivement parallélisable: en effet chaque sommet de l'élastique peut être traité indépendamment. J'ai donc opté pour une implantation multithread. Détaillons les aspects importants de cette architecture.

Un thread controleur, le thread principal, est chargé de la synchronisation et de la répartition des tâches entre les threads fils. Chaque thread fils a à sa charge un certain nombre de sommets, et un certain nombre de villes. Les threads fils effectuent de manière synchrone chacune des tâches suivantes:

- Calcul des W_{ij} non normalisés. $W_{ij}=\exp(-\|X_i-Y_j\|^2/(2*K^2))$.
- Calcul des $W_i=\text{Somme}(W_{ij}, j=1..N)$, d'où la nécessité d'attribuer à chaque thread des villes en plus des sommets de l'élastique.
- Normalisation des W_{ij} : $W_{ij}=W_{ij}/W_i$.
- Calcul des nouvelles coordonnées, sans modifier les valeurs, car elles sont utilisés par d'autres threads.
- Remplacement de l'ancienne valeur de coordonnée par la nouvelle.

Le procédé de synchronisation est implanté dans synchronizer.cpp et est utilisé de la manière suivante: Tous les threads partagent le même pointeur vers un objet de synchronisation. A chaque fois qu'un thread fils a terminé une des opérations ci-dessus, il appelle waitforsignal, ce qui le met en attente. Le thread principal, lui a appelé waitforwaiters, ce qui a eut pour effet de le mettre en attente jusqu'à ce que tous les thread fils soient en attente du signal. Une fois que tous les threads fils sont en attente, le thread controleur est débloqué. Il incrémente alors la variable indiquant l'opération courante, puis appelle la fonction signal, ce qui débloque tous les threads fils. Il appelle enfin waitforwaiters à nouveau, et le processus se répète.

Ce mécanisme de synchronisation utilise pour fonctionner trois sémaphores: un pour protéger ses données, un pour bloquer les threads en attente du signal, et le dernier pour bloquer le thread controleur, ainsi qu'un compteur indiquant le nombre de threads en attente du signal.

Etant donné que chacune des opérations effectuées par les threads s'effectue en un temps relativement constant, la répartition des sommets entre les threads ne s'effectue pas de manière dynamique à chaque itération, mais de manière statique, un même nombre de sommets et de villes étant attribué à chaque thread. Etant donné que le nombre de threads et de sommets dans l'élastique varie au cours du temps, un

mécanisme simple a été mis en place pour modifier la répartition rapidement: si on dispose de t threads et de $s=n*t+r$ sommets, $n+1$ sommets seront attribués aux r premiers threads, et n aux threads restants. La valeur de r est conservée, ainsi à chaque fois qu'un sommet est ajouté, il est ajouté au $r+1$ ième sommet et r est incrémenté de 1.

Malheureusement les performances constatées ne correspondent pas aux attentes: en utilisant les paramètres spécifiés par les auteurs de l'article définissant la méthode, le chemin obtenu est généralement plus long de 10% que celui obtenu avec la méthode 2-Opt.

4. Resultats obtenus, conclusion

Le tableau suivant synthétise les résultats obtenus pour les heuristiques implantées, et pour des problèmes de 50 à 1000 villes. Pour chaque valeur de N , chaque heuristique a été testée sur 5 instances du problème générées aléatoirement.

4.1. Resultats

Les longueurs indiquées sont normalisées, en prenant les résultats de 2-Opt comme base.

Table 1. Resultats obtenus en fonction du nombre de villes

Heuristique	Nombre de villes						
	50	100	200	300	400	500	1000
2-Opt	1	1	1	1	1	1	1
Plus proche voisin	1,12	1,19	1,16	1,2	1,19	1,20	
Moindre cout	1,02	1,12	1,14	1,14	1,14	1,16	
Recuit simulé	0,93	0,96	0,97	0,97	0,96	0,97	
Elastique	0,98	1,02	1,04	1,06	1,05	1,07	
ARPM	1,24	1,30	1,30	1,35	1,33	1,35	