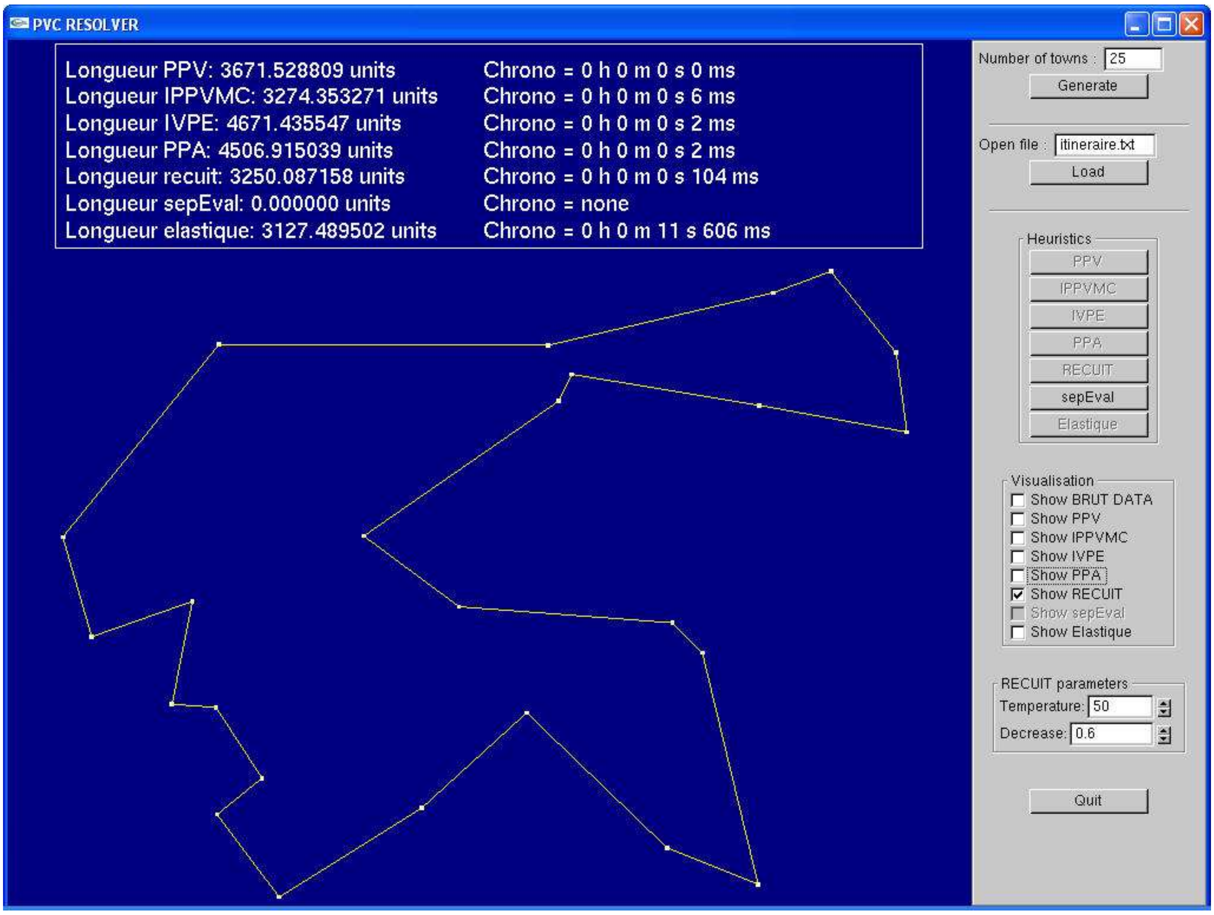
Université de Marne La Vallée
Cité Descartes
5, bd Descartes
Champs Sur Marne
77454 MARNE LA VALLEE CEDEX 2

Projet d'Optimisation Combinatoire :
Problème du Voyageur de Commerce



par
CAZABAN Adrien
PIVETTA Maxime

Master Informatique 1ère année

Enseignants responsables :
Jean Yves Thibon
Frédérique Bassino

Année Universitaire 2004/2005

Sommaire

1 Introduction

1.1 Sujet :

Le projet consiste à implanter différentes heuristiques pour le voyageur de commerce et, dans un deuxième temps, les comparer (qualité de la solution calculée, temps de calcul, taille des problèmes traités…).Enfin une interface permettant une représentation graphique est souhaitée afin de voir l'évolution des solutions calculées.

1.2 Problème du voyageur de commerce :

Un voyageur de commerce cherche la tournée la plus courte entre des villes toutes reliées par des lignes aériennes. Le circuit ne doit passer qu'une fois par chaque ville et revenir finalement à la ville de départ. Le problème (PVC) consiste à trouver un circuit de poids minimal qui passe part toutes les étapes une seule fois exactement.

1.3 Travail réalisé :

Le travail réalisé se présente sous la forme d'une interface facilitant les expérimentations. L'utilisateur a la possibilité d'utiliser un fichier test ou bien de générer des villes aléatoirement enfin de leur appliquer les différentes heuristiques implémentées. En effet, plusieurs algorithmes de résolution du PVC sont mis à disposition: séparation/évaluation, le plus proche voisin (PPV), insertion du voisin le plus éloigné (IVPE), parcourt préfixe de l'arbre recouvrant (PPA), méthode de l'élastique, le recuit simulé et enfin une fusion de l'insertion du plus proche voisin et insertion de moindre coût. Le logiciel est entièrement réalisé en C++, il utilise la librairie openGL pour l'affichage des solutions graphiques et GLUI pour l'interface.

2 Structure des données

2.1 Classes primordiales :

Les classes les plus importantes du projet sont « ville » et « trajet ». Une ville possède 2 coordonnées : en x et en y ainsi qu'un numéro identificateur.

La classe « trajet » contient : le nombre de villes que comporte le parcours, le trajet lui-même sous forme de « vector » et, enfin, le tableau des distance entre chaque ville. Même si la fonction de poids est symétrique pour notre problème, nous avons voulu stocker toutes les longueurs afin de garder la possibilité d'adapter notre programme à un problème asymétrique. Nous avons choisi le type « vector » car il était plus pratique à utiliser grâce notamment à sa surcharge de l'opérateur [ ] et au fait que l'on peut empiler des objets sans savoir la taille finale du vector. Cette structure nous a également permis de réduire la complexité de certains algorithmes en supprimant les villes déjà parcourues du vecteur, afin de ne pas les re-tester. Nous nous sommes rendu compte que cette structure est très lourde et il se pourrait qu'elle augmente les temps de calculs. Il se trouve que cette structure améliore considérablement la compréhension des différentes heuristiques, ce que nous avons recherché en parallèle de bons résultats.

C'est dans cette classe que l'on retrouve toutes les heuristiques sauf l'élastique. En effet, celle-ci a eut droit à une classe extérieure afin de vérifier si notre structure nous faisait gagner du temps de calcul (il s'est avéré que non).

2.2 Classes outils :

Une classe « chrono » a été mise au point afin de calculer les temps de calcul des différentes heuristiques. Sa précision, de l'ordre des millisecondes, s'avère amplement suffisante pour nos expérimentations

La classe « visualisation » gère toute la partie graphique du projet, que se soit pour l'interface ou l'affichage des résultats. Nous avons eut des difficultés à trouver un moyen d'afficher l'évolution des calculs, mais après plusieurs essais nous avons trouvé une technique simple et efficace qui consiste à rappeler notre fonction d'affichage dans les algorithmes.

2.3 Classes intermédiaires :

Les classes « arbre » et « nœud » servent d'intermédiaires pour calculer les arbres recouvrant et leur parcours.

3 Les heuristiques utilisées

3.1 Le plus proche voisin (PPV):

C'est la méthode la plus simple, on part d'une ville et à chaque étape on se déplace vers la ville la plus proche qui n'a pas encore été visitée.

Algorithme :

Initialisation du circuit avec la première ville des étapes à visiter
Pour toutes les villes non visitées
Recherche du minimum entre cette ville et la dernière ajoutée
Ajout de la ville la plus proche
Suppression de la ville ajoutée des villes à visiter
Fin tant que

La solution obtenue par cette heuristique peut être arbitrairement grande par rapport au résultat optimal. En effet, le résultat de cette méthode dépend du premier point inséré, et la dernière distance reliant le premier au dernier point n'est pas prévisible.

De ce fait, on pourrait améliorer la solution en exécutant n fois le PPV et en changeant à chaque fois le point de départ.

Cependant le PPV obtient des résultats rapides avec des problèmes de grosses tailles.

Complexité :

On cherche permis les villes à visiter, laquelle est la plus proche de la dernière ville du cycle. Comme on parcourt entièrement la matrice des distances des villes, on obtient une complexité $O(n^2)$. Cependant, grâce à l'utilisation de vecteurs, on ne revisite pas une ville déjà insérée, ce qui diminue un peu la complexité de cet algorithme (cf. complexité PPA).

3.2 Insertion du voisin le plus éloigné (IVPE) :

On part d'un circuit qui ne comporte que deux villes reliées par une arête de poids maximal. A chaque étape, on calcule pour chaque ville *v* ne se trouvant pas encore dans le circuit, la distance minimale de *v* à une ville du circuit. On insère alors la ville pour laquelle cette distance est maximale juste après l'étape dont elle est le plus proche.

Algorithme :

```
Initialisation du circuit avec les 2 villes les  plus éloignées
Suppression de ces villes des étapes à visiter
Tant que le circuit  n'est pas complet-1
        Pour toutes les villes à visiter
                Pour toutes les villes déjà dans le circuit
                        Calcul de la ville dont la distance est minimale
                Calcul de la ville dont la distance est  maximum des distances minimales
        Insertion de la ville trouvée juste après la ville dont elle est la plus proche
        Suppression de la ville des étapes à visiter
Fin tant que
Circuit + retour
```

Complexité :

On parcourt toutes les villes non traitées dans la matrice des distances. On parcourt donc entièrement toutes les lignes et les colonnes de la matrice pour les étapes à visiter. De manière générale, on obtient une complexité de « nbLignes*nbColonnes » soit O (n^2).

3.3 Résolution par parcourt préfixe de l'arbre (PPA):

L'algorithme se divise en 2 phases : la création de l'arbre recouvrant de poids minimal, puis son parcours.

On utilise l'algorithme de Prim pour créer l'arbre. Pour cela, on utilise deux listes : la liste de toutes les villes à trier, et la liste de toutes les villes que l'on a déjà triées. Le fait d'avoir utilisé des vecteurs pour stocker ces villes nous permet de supprimer les villes déjà triées de la liste de départ pour ne pas avoir à les reparcourir à chaque fois.

Algorithme :

```
Pour toutes les villes que l'on doit trier                          O (n)
        Pour chaque ville que l'on a déjà triée                     O (i)
                Pour chaque ville que l'on a pas encore triée       O (n-i)
                        Relier les villes de chaque liste dont la distance est minimale
```

Complexité :

Avec i allant de 0 à n, on a i(n-i) = (n/2)(n/2), on obtient donc une complexité pour la construction de l'arbre recouvrant de O ($1/4n^3$) au lieu de n^3 si l'on n'avait pas utilisé de vecteurs. Ceci permet de compenser des temps d'accès plus long avec les vecteurs, en bénéficiant d'une structure plus lisible et simple d'utilisation.

Le parcours préfixe de l'arbre obtenu grâce à Prim se fait avec une fonction récursive :

```
Parcours ( nœud courant) :
        Empile nœud courant
        Si courant a un frere
                Parcours(frere);
        Si courant a un fils
                Parcours(fils);
```

Le fait de regarder les frères avant les fils fait que notre parcours est préfixe.

3.4 Méthode de l'élastique :

La méthode de l'élastique a été récemment mise au point par deux chercheurs britanniques qui travaillaient sur un problème de modélisation de réseaux de neurones. Comme son nom l'indique, elle consiste à déformer un « élastique » placé au barycentre des points à relier. Celui-ci, d'abord sous la forme initiale d'un cercle de 2.5 * nombre de villes points, est soumis à 2 types de forces antagonistes : l'attraction par les sommets du graphe, et la tension qui l'incite à minimiser sa longueur. Ainsi, l'élastique est étiré jusqu'à passer par tous les points du graphe.

Première force mise en œuvre : l'attraction des villes

$$\alpha \sum_{i=1}^{n} W_{ij}(V_i - P_j)$$

Alpha est déterminé expérimentalement, Pj est le point de l'élastique sur lequel on calcule le déplacement et Vi sont les villes qui font subir une attraction.

$$W_{ij} = \frac{\phi(dist(V_i P_j), K)}{\sum_{k=1}^{n} \phi(dist(P_k V_i), K)}$$

Où

$$\phi(dist, K) = e^{\frac{-dist^2}{2K^2}}$$

Le dénominateur sert à normaliser la distance obtenue
« dist » est la distance entre la ville courante et le point de l'élastique
« K » est aussi un paramètre déterminé expérimentalement

Deuxième force mise en œuvre : la tension de l'élastique

Elle tend à minimiser la taille de l'élastique en cherchent à le tendre et à maintenir sa cohésion. Chaque point de l'élastique subit une forte attraction de ces 2 proches voisins.

$$\beta K(P_{j+1} - 2P_j + P_{j-1})$$

Beta est un paramètre déterminé expérimentalement.

Après plusieurs expérimentations, nous avons finalement décidé de prendre comme paramètres :

K=0.5
Beta=1.8
Alpha=0.8

Algorithme :

Tant que l'élastique continu à se déplacer (K>0)
Pour tous les points de l'élastique
Pour toutes les villes à visiter
Calcul du dénominateur de Wij
Calcul de Wij
Calcul de Vi-Pj
Calcul du déplacement du point
Mise à jour du point de l'élastique
Diminution de K
Fin tant que

Complexité :

La complexité de cette méthode est très grande, mais on remarque que les calculs du déplacement de chaque point peuvent être fait en parallèles.

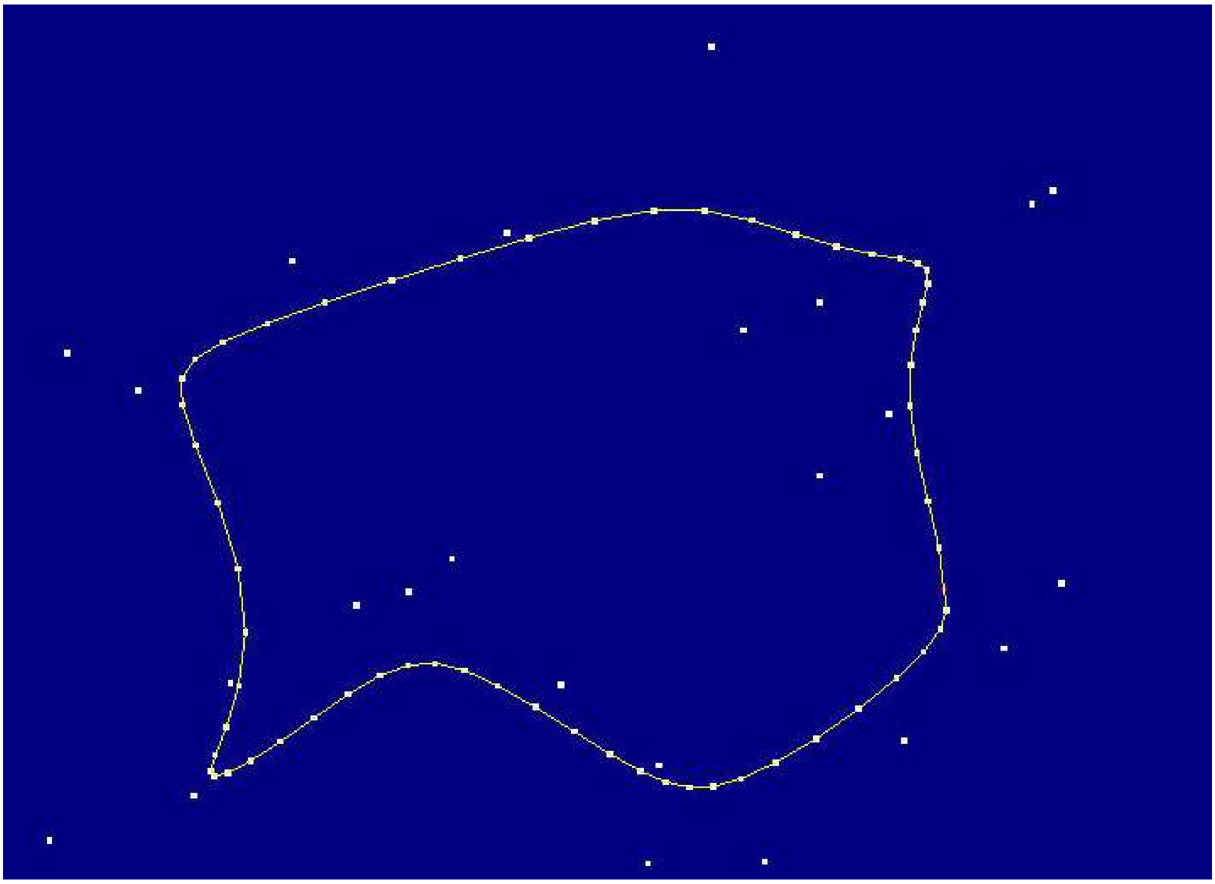


fig. 1 : Elastique en cours de formation

3.5 Le recuit simulé :

Le recuit simulé est une méthode basée sur les principes physiques d'équilibre énergétique lors de la cristalisation des métaux. On introduit un paramètre de température qui est ajusté pendant la recherche.

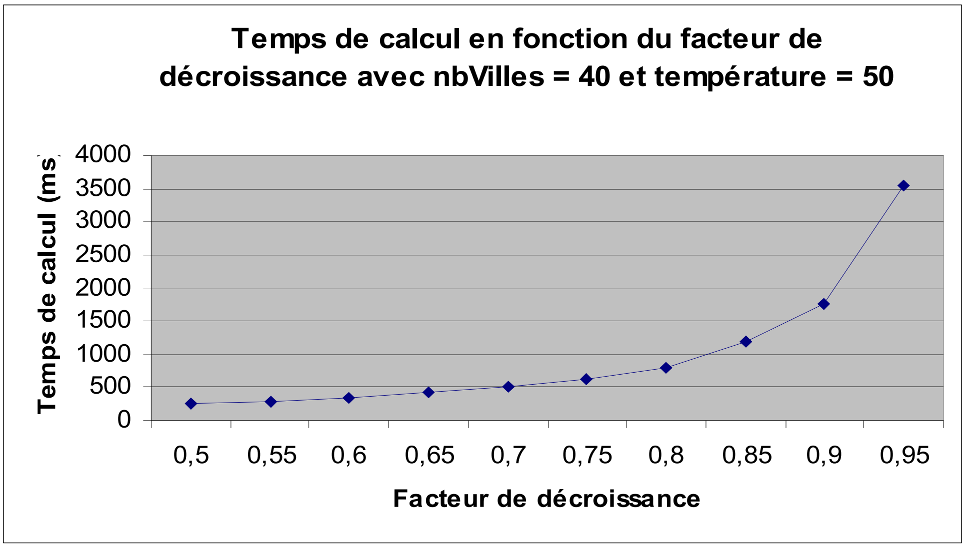
Le principe du recuit simulé est de partir d'un circuit, et d'essayer de l'améliorer en inversant certaines de ses arrêtes. Plus la température est élevée, puis l'algorithme autorisera des transformations couteuses. Sa plus grande particularité est d'autoriser certaines inversions, même si elle n'améliore pas le poids du circuit. En effet, dans le cas où l'inversion est bénéfique au circuit, on la conservera, dans le cas contraire, on guide le hasard vers un choix.

Algorithme :

```
Initialisation aléatoire du circuit
On fixe une température (50), un taux de décroissance (0.9), et une valeur minimale
(0.0001)
Tant que température > valeur minimale
        On intervertit deux arrêtes au hasard
        Si ce changement améliore le circuit
                On le conserve
        Sinon on tire aléatoirement p entre 0 et 1
                Si p < exp(-sqrt(nbVilles) * perte / température)
                        On conserve l'inversion
        On fait décroître température par taux de décroissance
```

Complexité :

Le recuit simulé faisant largement intervenir le hasard, la qualité du résultat obtenu, ainsi que le temps mis pour l'obtenir peut varier énormément d'une fois sur l'autre. Cependant, plus on augmente le nombre de villes, plus cet algorithme se montre efficace.
On a quand même une emprise sur le temps de calcul, en faisant varier la température de départ, et son taux de décroissance, c'est pourquoi ces paramètres se trouvent dans notre interface. On pourrait également faire varier la condition d'arrêt, en l'augmentant un peu ce qui accélère beaucoup l'algorithme mais fait beaucoup perdre de qualité au résultat obtenu.



graph. 1 : Paramètres du recuit simulé

Ce graphique montre à quel point le facteur de décroissance influe sur le temps de calcul. En revanche, sur une série de 20 tests, la qualité du résultat obtenu a varié de moins de 5% en faisant passer le facteur de décroissance de 0.6 à 0.9. (alors qu'elle peut varier de 10% avec les mêmes paramètres, à cause du random)
Ces résultats nous ont poussé à utiliser un taux de décroissance plus bas que prévu (0.6) et une température de 50°.

3.6 Insertion du plus proche voisin à moindre coût (IPPVMC) :

Pour l'heuristique optionnelle à ajouter à notre logiciel, nous avons décidé de fusionner 2 méthodes de résolution du PVC : Insertion du plus proche voisin et l'insertion de moindre coût. En effet, en mélangeant ces 2 heuristiques, nous avons obtenu de très bons résultats.

Le principe est simple, on part d'un circuit trivial qui ne comporte qu'une seule ville. A chaque étape, on calcule pour chaque ville *v* se trouvant dans le circuit, la distance minimale de *v* à une ville des étapes à visiter. On insère alors la ville dont la distance est minimale (cad minimum des minimums) de façon à minimiser l'augmentation de poids du circuit.

Algorithme :

```
Initialisation du circuit  avec la première ville des étapes à parcourir
Insertion de la ville la plus proche de celle débutant le circuit
Insertion de la première ville du circuit  pour le retour
Suppression de ces villes dans les étapes à parcourir
Tant que le circuit n'est pas plein+1
        Pour toutes les villes déjà dans le circuit
                Pour toutes les villes à visiter
                        Calcul de la ville dont la distance est minimale
        Pour toutes les villes déjà dans le circuit-1
                On insère la ville juste après la ville courante
                On calcule le poids du circuit
                Si il est minimal on le garde
                Suppression de la ville du circuit
        Insertion de la ville à la place minimisant le poids du circuit
        Suppression de la ville trouvée des étapes à visiter
Fin tant que
```

Complexité :

On parcourt toutes les villes du circuit et toutes celles des étapes à visiter pour trouver le minimum. Le calcul du poids oblige le parcours de toutes les villes du cycle et ceci pour chaque test de placement. On obtient finalement O ($2n^2$).

3.7 Séparation Evaluation :

La méthode de séparation-évaluation est la seule méthode de notre projet qui fournisse à coup sûr le résultat optimal. Elle se base sur un parcours arborescent de l'arbre des solutions. La subtilité de la méthode consiste à éliminer des solutions possibles avant de l'explorer intégralement. Pour cela on calcule un minorant par une heuristique simple, et on stoppe l'exploration de la branche lorsque le poids intermédiaire de celle-ci dépasse le minorant.

Algorithme :

```
Initialisation du circuit avec une autre heuristique
Fonction récursive ( profondeur, poids_courant, circuit)
        Si on n'est pas sur un niveau terminal
                Si (poids_courant < minorant)
                        Ajoute une ville non vue au trajet courant
                        Relance récursivité avec nivo+1
        Sinon
                Si (poids_courant < minorant)
                        Minorant = poids_courant
                        Circuit = trajet courant

Retourne circuit
```

Complexité :

La complexité du parcours de l'arbre entier est factorielle O (n !). Ce qui dépasse rapidement les capacités de calcul des ordinateurs actuels. Cependant, notre implémentation de « séparation évaluation » nous permet de ne parcourir qu'environ 60% de l'arbre, ce qui permet de monter jusqu'à 17 villes, en conservant des temps de calcul raisonnable (<1h).

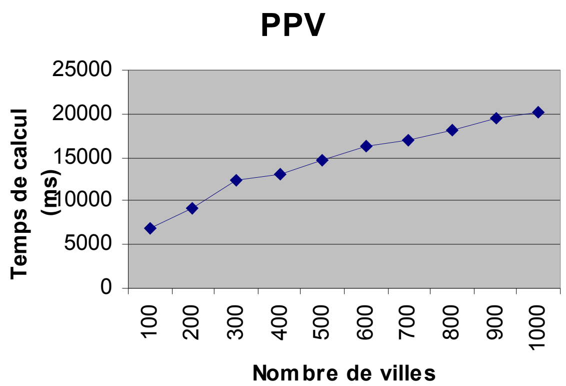
Cette méthode n'étant pas destiné à servir pour des trajets de plus de 20 villes, nous avons choisi de l'initialiser avec un algorithme efficace (le recuit simulé) car le temps d'initialisation avec 20 villes entre le recuit et le PPV est sensiblement identique. Cela nous permet donc d'avoir tout de suite un meilleur comparant et donc de « couper » plus de branches.
Une deuxième amélioration nous a aussi fait gagné 5% de temps de calcul. Lors du test entre le poids courant et le minorant, nous soustrayons au minorant la plus petite des distances possible entre les villes, ce qui rend la condition encore plus contraignante. En effet, le minorant étant calculé à partir d'une heuristique, son poids est le poids d'une boucle (on relie la première ville à la dernière), or dans le cas où on n'effectue pas le test sur une branche feuille, nous n'avons pas de cycle, et donc une arrête de moins.

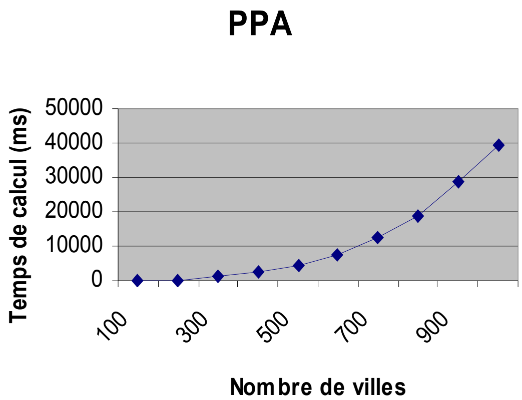
4 Analyses et critiques des résultats obtenus

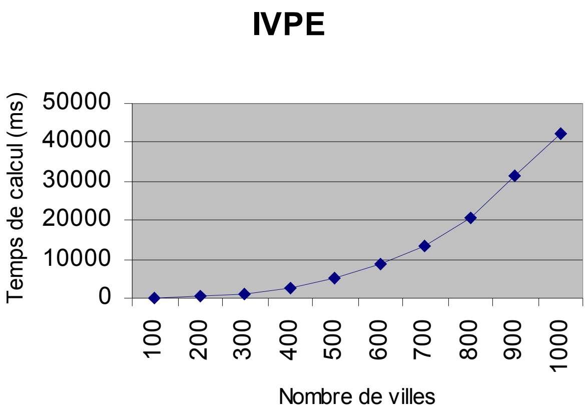
Nous avons effectués de nombreux tests afin de comparer les algorithmes entre eux. Tous les tests ont été réalisés sur une même machine (AMD Athlon XP 2500+). Deux jeux de données allant de 5 à 3000 villes chacun ont été générés, et sauvegardés afin de servir de référence dans les tests comparant les heuristiques. Il est évident que pour obtenir la véritable allure des courbes, il aurait fallu au minimum 10 jeux de données, mais nous obtenons déjà une allure significative pour chaque heuristique.

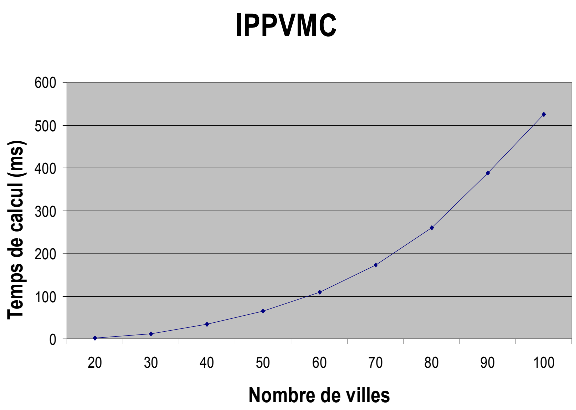
4.1 Courbes de complexité en temps pour les différentes heuristiques

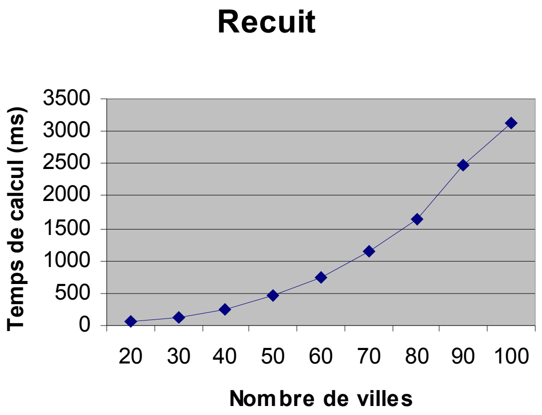
Cete première partie montre comment les temps de calcul évoluent avec le nombre de villes dans chaque algorithme. On ne cherche pas ici à comparer les heuristiques les unes avec les autres, mais plutot à regarder leur complexité.

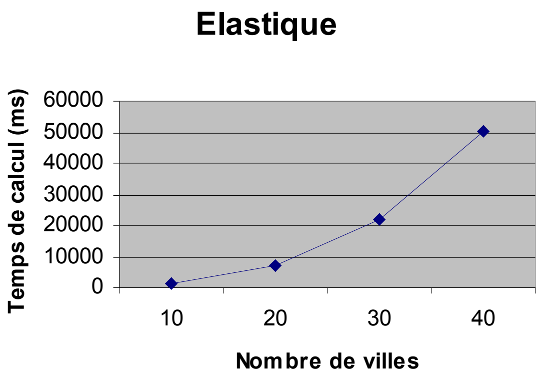


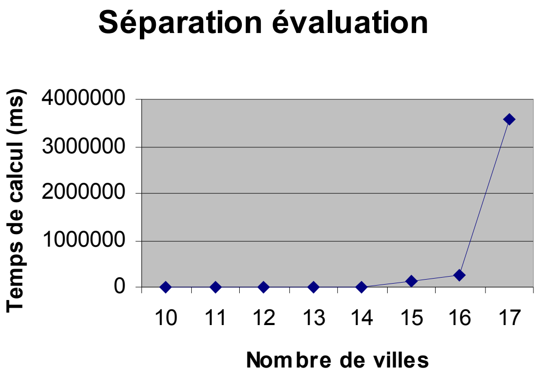








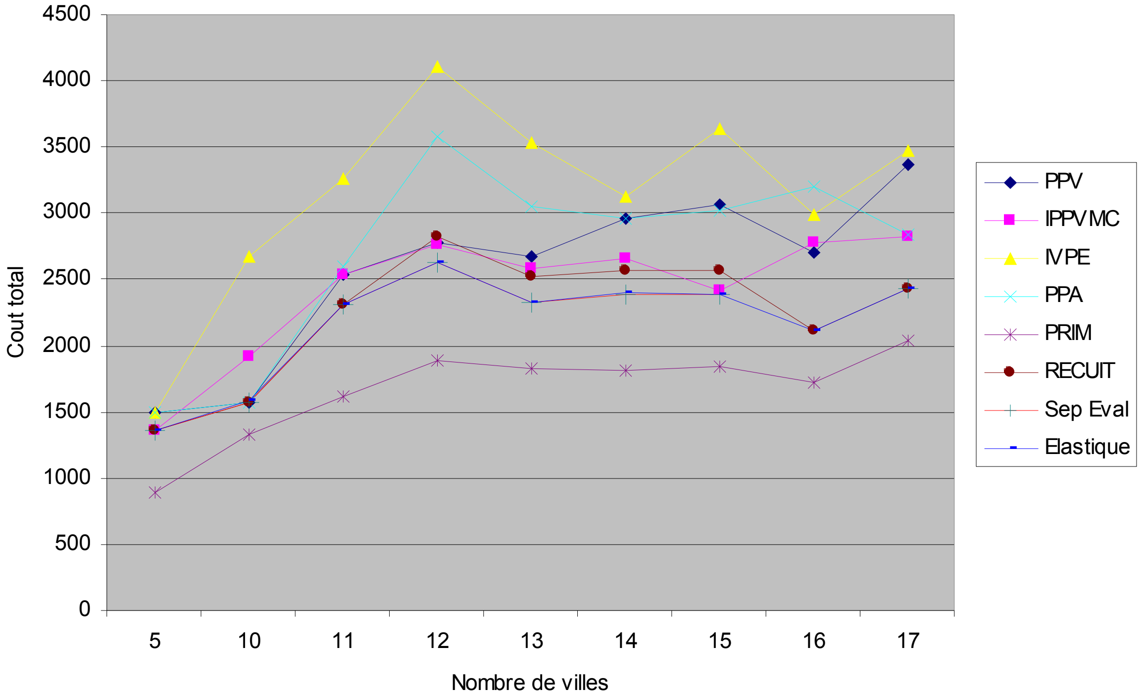




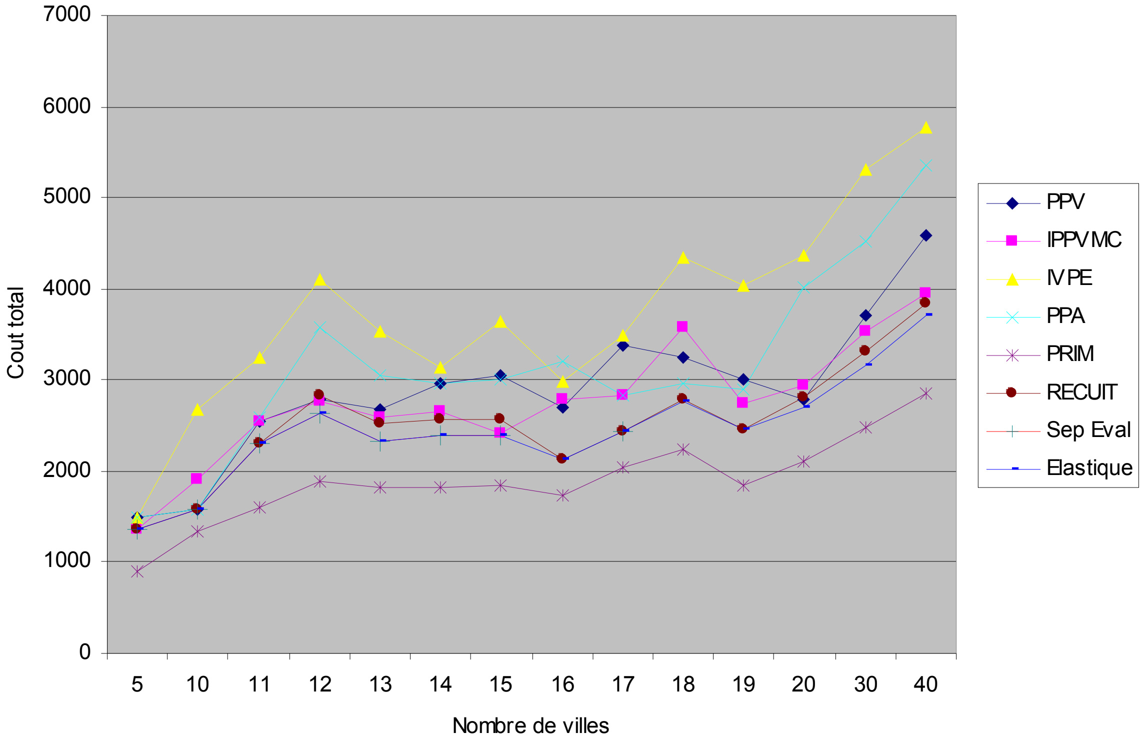
graph. 2 : Courbes de complexité de toutes les heuristiques.

On remarque que la complexité de tous les algorithmes est sensiblement identique (O (n^2) en général), sauf pour le PPV et séparatoin-évaluation. Le PPV évolue de façon quasi-linéaire, tandisque séparation-évaluation se rapproche d'une complexité factorielle.

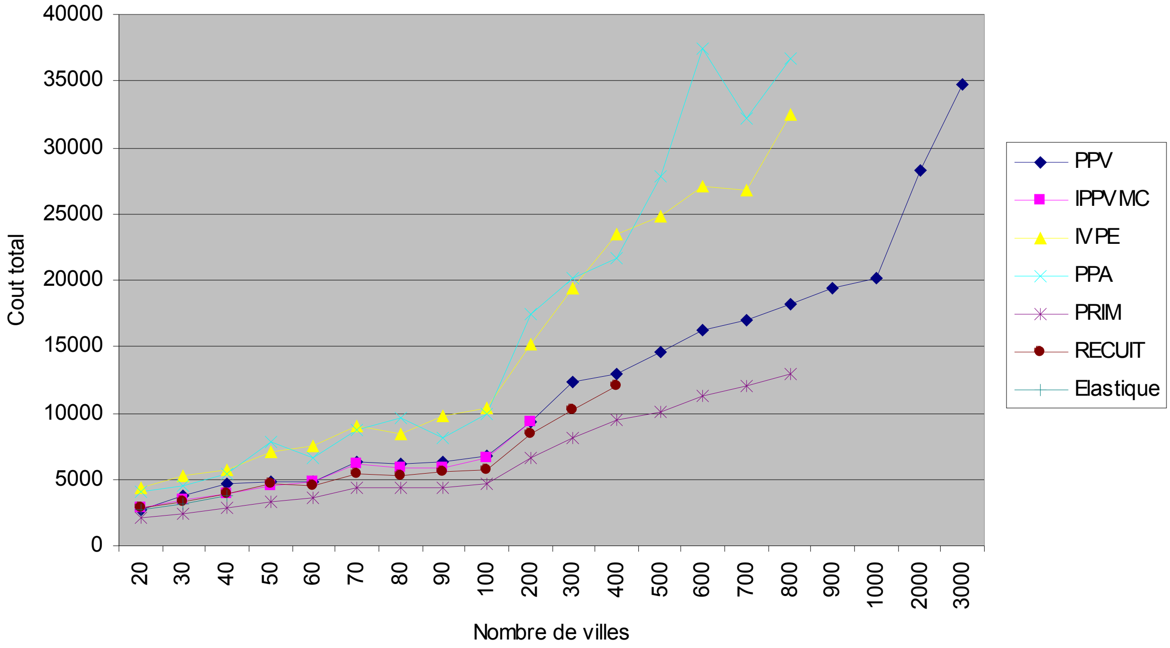
4.2 Tests sur l'efficacité de la solution obtenue en fonction du nombre de villes



graph. 3 : Cout total avec toutes les heuristiques sur un petit nombre de villes



graph. 4 : Coût total avec toutes les heuristiques sur un nombre moyen de villes



graph. 5 : Cout total avec toutes les heuristiques sur un grand nombre de villes

Ces tests nous permettent de voir que tous le classement d'efficacité est modifié en fonction de la taille du jeu de données.
Des algorithmes comme le PPV sont inintéressants sur un petit nombre de villes mais permettent d'obtenir des résultats corrects avec un gros jeu de données.

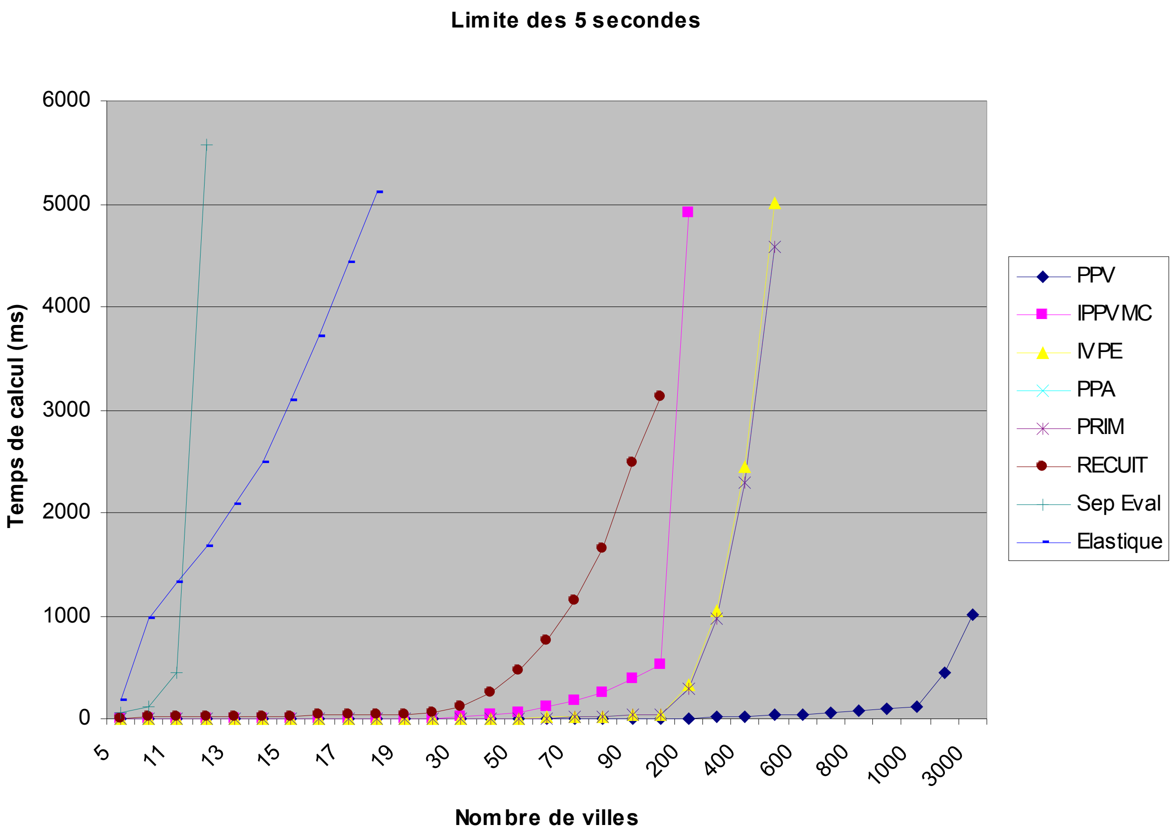
On remarque également que l'arbre recouvrant de poids minimal permet d'obtenir une bonne approximation du résultat optimal. En effet, il semble que son poids soit d'environ 65% du poids de la solution optimale. Mais ce résultat est tout à fait empirique et ne peut de toutes façons pas être vérifié de façon théorique puisque l'on est incapable de prévoir le circuit optimal pour un gros nombre de villes.

4.3 Tests sur le temps de calcul en fonction du nombre de villes

Ce test mesure la plage de données sur laquelle on peut utiliser chaque algorithme. On a fixé pour ce test la limite à 5s, puis on a regardé jusqu' à combien de villes on obtenait un résultat avec chaque algorithme.

Ce test nous permet de voir l'ordre dans lequel les heuristiques « explosent ». Nous obtenons :

- Séparatoin-évaluation : 12 villes en 5'' 568
- Elastique : 18 villes en 5''111
- Recuit en 100 villes en 3''128
- IPPVMC 200 villes en 4''920
- PPA 500 villes en 4''577
- PPV 3000 villes en 1''004



graph. 6 : Temps de calcul « frontière » pour chaque heuristique

4.4 Plages d'utilisation

On a cherché ici à savoir quelle méthode devrait être utilisé sur chaque plage de données. On a donc choisi la méthode donnant la meilleure solution, en moins d'une heure. Il en ressort que :

<u>De 4 à 17 villes :</u> la méthode de séparation évaluation donne le résultat optimal. (9 fois sur 10, l'élastique et le recuit ont donné le même résultat plus rapidement)

<u>De 18 à 40 villes :</u> la méthode de l'élastique est la plus efficace (4 fois sur 10 le recuit donne le même résultat, et 2 fois sur 10 un résultat meilleur, et plus rapidement)

<u>De 40 à 300 villes :</u> le recuit domine toutes les autres méthodes (9 fois sur 10 supérieur à l'IPPVMC, mais l'IPPVMC ne s'écarte pas de plus de 10%, en étant globalement 10 fois plus rapide)

<u>+ 300 villes :</u> le PPV domine largement l'IVPE et le PPA, tout en étant le plus rapide.

5 Interface

L'interface de notre projet se compose d'un espace d'affichage openGL (où se trace le cycle) et d'une barre d'outils construite à partir de la librairie Glui.

L'espace d'affichage :

Les trajets sont dessinés dans cet espace dans un repère allant de 800 pour les x à 600 pour les y. On peut voir l'évolution des tracés ce qui permet de remarquer les choix, fait au cours du temps, par les heuristiques. Pour chaque algorithme de résolution, la longueur du trajet et le temps mis pour le calculer sont affichés. Toutes ces informations sont placées les unes à côté des autres afin de faciliter les comparaisons.

Le chrono défile pour mieux apprécier le temps de calcul, en soustrayant à chaque fois le temps imputable à la visualisation. Le temps affiché est donc le véritable temps de calcul. Pour l'heuristique séparation-évaluation, il se transforme même en pourcentage. En effet, celui étant long à calculer dès 16 villes, il devenait nécessaire de renseigner l'utilisateur de la progression de l'algorithme.

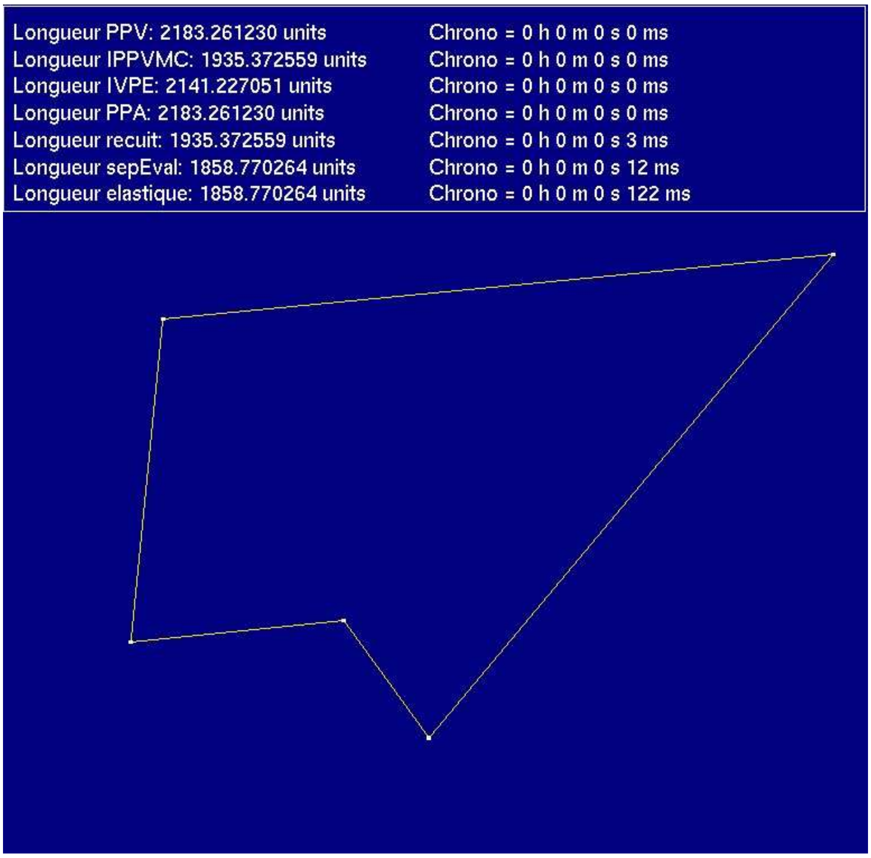


Fig. 2 : Fenêtre de visualisation du trajet obtenu

<u>La barre d'outils :</u>

Celle-ci est simple et intuitive. Elle est réduite à son strict minimum afin de ne pas perdre l'utilisateur dans trop d'informations et de faciliter les expérimentations. L'utilisateur peut au choix, soit charger un fichier du répertoire courant, soit générer aléatoirement un jeu de villes. Nous avons voulu guidé l'utilisateur afin d'éviter au maximum les fausses manipulations. Ainsi, l'application des heuristiques n'est possible que si un fichier de villes a été chargé ou bien si celles-ci sont générées. Une fois calculée, l'heuristique est stockée afin de pouvoir la réafficher a tout moment .Cette technique permet de superposer tous les résultats obtenus et ainsi de bien apprécier les différences de parcours. Pour cela, il suffit de cocher les heuristiques que l'on souhaite voir s'afficher, bien sûr si l'heuristique n'a pas été calculée, la case reste grisée. On peut ensuite à loisir réafficher un trajet, sans avoir à le recalculer.

Des messages apparaissent si on lance un algorithme avec un nombre de villes nécessitant trop de temps de calcul.

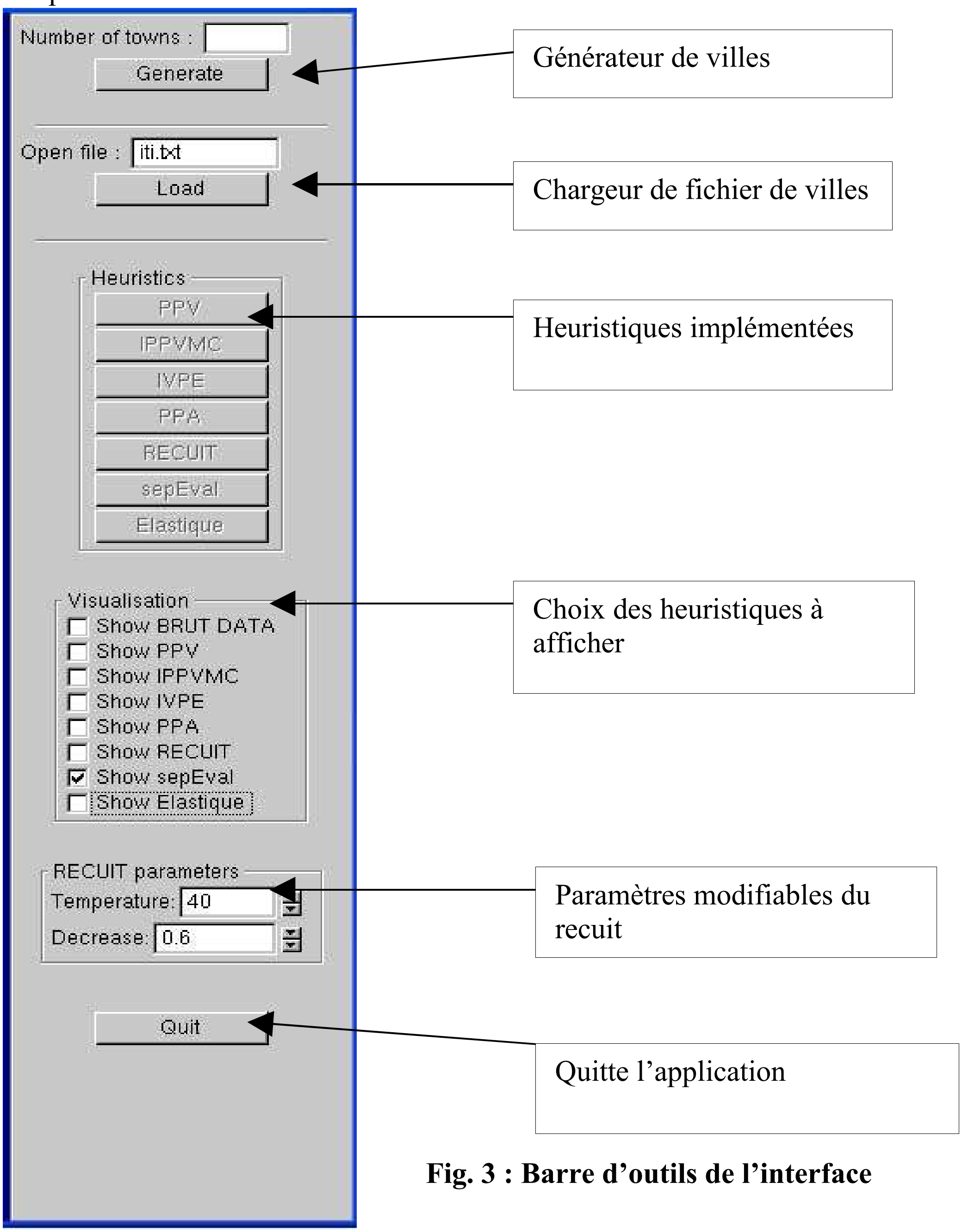


Fig. 3 : Barre d'outils de l'interface

6 Conclusion

Le fait de devoir implémenter nous-mêmes des méthodes de résolution pour le problème du voyageur de commerce, et de les tester, nous a permis de nous rendre compte à quel point il est important de disposer de plusieurs outils, en fonction de la taille du problème.

Le seul algorithme donnant à coup sur un résultat optimal étant bien trop long, il est important de disposer d'autres méthodes. Grâce à l'étude de tous ces heuristiques, nous sommes maintenant capable de choisir, en fonction du temps que l'on a à consacrer aux calculs, quel algorithme sera le plus approprié ainsi que ses paramètres.

7 Annexes

Voici les données qui ont servies à quelques uns de nos tests.

	PPV		IPPVMC		IVPE		PPA		Prim		Recuit		SepEval		Elastique	
Nb de villes	Cout	Temps	Cout	Temps	Cout	Temps	Cout	Temps	Cout	Temps	Cout	Temps	Cout	Temps	Cout	Temps
5	1495	0	1357	0	1495	1	1495	1	898	1	1357	6	1357	58	1357	173
10	1577	0	1915	1	2672	1	1577	1	1334	1	1577	12	1577	124	1585	971
11	2544	0	2543	1	3256	1	2592	1	1612	1	2303	15	2303	440	2303	1323
12	2786	0	2761	1	4110	1	3577	1	1890	1	2828	18	2629	5568	2629	1665
13	2680	0	2580	2	3530	1	3046	1	1832	1	2529	20	2333	14002	2333	2078
14	2954	0	2662	2	3133	1	2954	1	1819	1	2560	24	2387	15297	2395	2487
15	3059	0	2418	2	3638	1	3016	1	1838	1	2567	27	2386	118562	2386	3090
16	2704	0	2776	2	2993	1	3197	1	1726	1	2118	30	2118	246251	2118	3701
17	3372	0	2821	2	3480	1	2835	1	2037	1	2430	35	2430	3591000	2430	4424
18	3242	0	3567	2	4339	1	2969	1	2234	1	2788	39			2767	5111
19	3009	0	2737	2	4045	1	2897	1	1851	1	2464	44			2464	971
20	2781	0	2932	2	4373	1	4020	1	2116	1	2805	50			2703	6848
30	3698	0	3531	12	5320	2	4517	2	2472	2	3315	126			3160	22139
40	4587	0	3948	34	5780	4	5362	3	2851	3	3849	257			3716	50491
50	4819	1	4564	65	7001	7	7823	7	3255	7	4616	462				
60	4852	1	4840	110	7506	11	6668	8	3609	8	4547	748				
70	6275	1	6205	173	8990	16	8678	12	4354	12	5368	1154				
80	6179	1	5863	260	8363	25	9558	19	4293	19	5251	1643				
90	6255	1	5912	388	9835	30	8156	31	4294	31	5499	2486				
100	6776	3	6686	524	10356	46	9896	38	4617	38	5774	3128				
200	9287	4	9262	4920	15212	322	17426	293	6650	293	8448	22123				
300	12327	12			19450	1049	20156	971	8103	971	10219	75470				
400	12974	22			23391	2455	21684	2291	9408	2291	11971	175918				
500	14566	39			24782	5003	27852	4577	10131	4577						
600	16170	45			27065	8557	37427	7766	11315	7766						
700	17046	62			26767	13645	32131	12608	11989	12608						
800	18185	76			32476	20564	36749	18582	12992	18582						
900	19406	96			33304	31671	41334	28570								
1000	20202	113			34836	42108	37023	39549								
2000	28221	450														
3000	34759	1004														