université de BORDEAUX

Collège Sciences et technologies

UNIVERSITÉ BORDEAUX 1 Sciences Technologies

PROBLEME DE VOYAGEUR DU COMMERCE

MIMSE SPE 3

Année universitaire

2014/2015

Réaliser par

BENCHEIKH Ghita

SADIK Meriem

Introduction Générale

Le problème du voyageur de commerce (TSP) a été évoqué pour la première fois en 1930 par le mathématicien viennois Karl Menger et il a intrigué bon nombre de chercheurs depuis ce temps : sans doute parce qu'il est facile à énoncer mais redoutable à résoudre. Ce problème, en apparence anodin et insignifiant, cache de nombreuses finesses et difficultés qui nous emmèneront dans les développements récents de la théorie des algorithmes

On se donne n villes ainsi que les distances (longueur ou temps) entre chaque ville. On peut représenter le problème sous forme d'un graphe et d'une matrice représentant le graphe (les villes sont indicées de 0 à n-1). Le problème du voyageur de commerce, consiste en la recherche d'un trajet minimal permettant à un voyageur de visiter n villes. En règle générale on cherche à minimiser le temps de parcours total ou la distance totale parcourue. Pour résoudre ce problème, nous avons recherché un algorithme : la notion, bien qu'ancienne, n'a été définie et étudiée qu'au XXe siècle, quand il s'est agi pour les mathématiciens de réfléchir aux fondements logiques des mathématiques. Essayons en quelques mots de préciser ce que cette notion recouvre. Un algorithme peut être défini comme un processus, une suite d'instructions permettant la résolution d'un problème donné dans toute sa généralité et en particulier quelle que soient les données du problème.

C'est un concept-clé de l'informatique : l'ordinateur ne peut traiter que les problèmes pour lesquels un algorithme existe ; les langages de programmation, permettent à l'ordinateur de comprendre l'algorithme qu'on veut lui soumettre.

Cette définition, et l'étude que nous avons faite sur le problème du voyageur de commerce, posent une question, auxquelles nous tenterons d'apporter une réponse dans la suite :

Un algorithme doit-il finir dans un intervalle de temps raisonnable ?

D'un point de vue théorique, avoir un nombre fini d'étapes est la seule contrainte que l'on impose à un algorithme ; mais d'un point de vue pratique, la réponse à un problème donné doit pouvoir être obtenue à échelle humaine : à ce titre, l'existence d'un algorithme ne suffit pas à ce que le problème soit concrètement résolu. Ainsi, l'algorithme exhaustif du problème du voyageur de commerce résout le problème posé en théorie mais est incapable de fournir une réponse dans un temps raisonnable pour un nombre de villes pourtant très petit. Reste à définir ce que peut être un temps raisonnable : quelques heures, voire quelques jours, pour

obtenir la réponse à une question, peuvent a priori être considérés comme une attente raisonnable. Mais la notion est éminemment subjective. Elle dépend d'abord de la machine utilisée (plus ou moins puissante), mais aussi du problème. Les questions que nous abordons là concernent ce que l'on appelle l'efficacité des algorithmes : elles ont été étudiées vers le milieu des années 60 par A Cobham et J Edwards. Pour évacuer le facteur temps, par trop subjectif, on fait intervenir le nombre d'étapes élémentaires (comme le nombre de multiplications ou de divisions) nécessaires pour mener l'algorithme à son terme quand il est écrit sur une machine.

Un algorithme est dit à complexité polynomiale s'il existe deux entiers fixes A et k tels qu'à partir des données de longueur n, l'algorithme demande au plus An^k étapes.

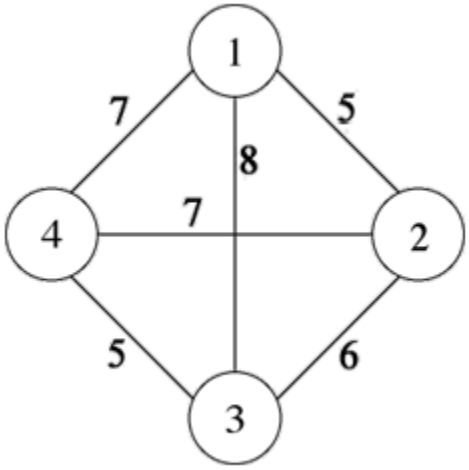
Les algorithmes qui ne sont pas à complexité polynomiale sont dits à complexité exponentielle. Ainsi un algorithme qui requiert 2n ou n! étapes pour des données de longueur n est à complexité exponentielle : la fonction qui donne le nombre d'étapes n'est pas obligatoirement une fonction exponentielle au sens usuel.

La difficulté de résolution du TSP a d'autant plus porté sur lui l'attention des chercheurs. Aussi a-t-il été étudié et retourné sous tous les angles : théorie des graphes, programmation linéaire, programmation dynamique, optima locaux, etc. Les premières formalisations de la recherche exhaustive par la stratégie d'évaluation et séparation seraient même nées de recherches sur le voyageur de commerce. Ce travail nous donnons dans un premier temps une définition du problème de voyageur de commerce. Après avoir présenté le problème, nous proposons des algorithmes, exacts puis approchés. nous finissons par tests affin de vérifier la performance de chacune des méthodes de résolution.

Voyageur de commerce

1. Définition

Le problème du voyageur de commerce peut être modélisé à l'aide d'un graphe constitué d'un ensemble de sommets et d'un ensemble d'arêtes. Chaque sommet représente une ville, une arête représente la distance (ou le temps) séparant une ville à une autre, Ci-contre, un exemple de graphe à 4 sommets.



Exemple de graphe à 4 sommets.

Résoudre le problème du voyageur de commerce revient à trouver le trajet de longueur minimale passant par toutes les villes et revenant au point de départ (un tel cycle est dit "Hamiltonien"). Pour le graphe ci-contre, une solution à ce problème serait le cycle 1, 2, 3, 4 et 1, correspondant à une distance totale de 23. Cette solution est optimale, il n'en existe pas de meilleure.

Comme il existe une arête entre chaque paire de sommets, on dit que ce graphe est « complet ». Pour tout graphe, une matrice de poids peut être établie. En lignes figurent les sommets d'origine des arêtes et en colonnes les sommets de destination ; le poids sur chaque arête apparaît à l'intersection de la ligne et de la colonne correspondantes. Pour notre exemple, cette matrice est la suivante :

$$\begin{pmatrix} \infty & 5 & 8 & 7 \\ 5 & \infty & 6 & 7 \\ 8 & 6 & \infty & 5 \\ 7 & 7 & 5 & \infty \end{pmatrix}$$

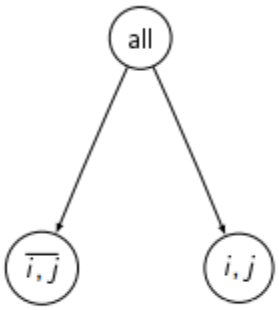
2. Méthodes de résolution

Il existe deux grandes catégories de méthodes de résolution : les méthodes exactes et les méthodes approchées. Les méthodes exactes permettent d'obtenir une solution optimale à chaque fois, mais le temps de calcul peut être long si le problème est compliqué à résoudre. Les méthodes approchées, encore appelées heuristiques, permettent quant à elles d'obtenir rapidement une solution approchée, mais qui n'est donc pas toujours optimale.

Méthode exacte

Pour le problème du voyageur de commerce, l'une des méthodes exactes les plus classiques et les plus performantes reste la procédure par Séparation et Evaluation (PSE) ou Branch and Bound. Cette méthode repose sur le parcours d'un arbre de recherche. Il s'agit de rechercher une solution optimale dans un ensemble de solutions réalisables, c'est-à-dire il consiste à chercher la meilleure solution parmi toutes les solutions possibles. Cette méthode repose d'abord sur la séparation (brunch) de l'ensemble des solutions en sous-ensembles plus petits, puis l'exploration de ces solutions en utilisant une évaluation optimiste (bound) pour trouver la meilleure entre eux.

La séparation permet d'énumérer intelligemment toutes les solutions possibles, pour décrire cette opération il suffit de dire comment on divise un ensemble en sous-ensembles. pour cela nous utilisons un « arbre de recherche » constitué par des nœuds, des arcs et ils sont repartis en niveaux, telle que chaque nœud représente une étape de construction de la solution et chaque arc représente les choix faits pour construire la solution. L'arbre de recherche commence par un seul nœud de niveau 0 appelé racine de l'arborescence où aucune décision n'a été prise. Initialement l'arbre a un seul noeud, représentant tous les cycles. Le noeud (i,j) représente tous les cycles contenant l'arête (i,j) et le noeud $\overline{(i,j)}$ tous les cycles qui ne la contiennent pas. Chaque étape de l'arbre divise ainsi l'ensemble des cycles.



Chaque noeud est un objet de type Noeud, qui contient un champ réserver à la matrice des distances correspondante au noeud courant, un champ qui stock la valeur du noeud, est un dernier champ qui stock les arc qu'on a fixé pour arriver à ce dernier.

Dans le cas du problème du voyageur de commerce, la fonction objectif à minimiser est la longueur du cycle. La réduction de l'espace de recherche repose sur l'utilisation de bornes inférieures et supérieures.

Borne inférieure

C'est une estimation par défaut de la fonction objectif. Autrement dit, c'est une valeur qui est nécessairement inférieure à la valeur de la meilleure solution possible. Dans notre cas, s'il y a un graphe avec N sommets à traverser, le cycle hamiltonien passera obligatoirement par N arêtes. Ainsi, pour avoir une borne inférieure assez intuitive, il suffit d'additionner les plus petits poids de chaque ligne et chaque colonne, toute en le soustrayant de la matrice des distances du noeud courant. Même si cette solution a de forts risques de ne pas être réalisable, la valeur de la fonction objectif ne pourra pas être plus petite.

La fonction qui permet de calculer la borne inférieur est décrite si dessous :

Borne inférieur

Borne_inf ← 0 **Pour** i **allant de** 1 **jusqu'à** n **faire** min ← M[i] [0] **Pour** j **allant de** 1 **jusqu'à** n **faire** **Si (M[i][j] < min)** min = M[i][j] **finsi** **finpour**	**Recherche du plus petit élément de chaque ligne**
Borne_inf ←Borne_inf + min	Ajouter cet élément à la borne inférieure
Pour j **allant de** 1 **jusqu'à** n **faire** M[i][j] ← M[i][j] - min **finsi** **finpour** **Finpour**	Soustraction de cet élément de la ligne i
Pour i **allant de** 1 **jusqu'à** n **faire** min ← M[0] [i] **Pour** j **allant de** 1 **jusqu'à** n **faire** **Si (M[j][i] < min)** min = M[j][i] **finsi** **finpour**	**Recherche du plus petit élément de chaque colonne**
Borne_inf ←Borne + min	Ajouter cet élément à la borne inférieure
Pour j **allant de** 1 **jusqu'à** n **faire** M[j][i] ← M[j][i] - min **finsi** **finpour** **Finpour** Cet algorithme est de complexité polynomile.	Soustraction de cet élément de la colonne i

Choix de l'arc

après le calcul de la borne inférieur, on remarque qu'on obtient au moins un zéro par ligne et par colonne. la première idée est de prend aléatoirement un zéro parmi les zéros existent, et

comme le temps d'exécution joue un rôle très important dans le monde de la recherche opérationnel, c'est à cause de lui que les problèmes d'optimisation combinatoire sont réputés si difficiles, nous avons opté à un choix d'arc qui mène rapidement au résultat, ce choix de l'arc se fait en fonction de la valeur du regret, c'est à dire le coût qu'on payera si on ne choisi pas l'arc (l,c), pour le calculer il suffit d'additionner le plus petit poids de la ligne l et la colonne c sans y tenir en compte l'élément (l,c).

```
Regret : élément ( l , c )
Min1 ← 0
Min2 ← 0
Pour i allant de 1 jusqu'à n faire
   Si i ≠ c et M[ l ][ i ] < M[ l ][ Min1 ]
      Min1 ← i
   Finsi
Finpour
Pour i allant de 1 jusqu'à n faire
   Si i ≠ l et M[ i][ c ] < M[ min2 ][ c ]
      Min2 ← i
   Finsi
Finpour
Revoie M[ l ][ min1 ] + M[ min2 ][ c ]
```

On calcul le regret de chaque 0 figurant dans la matrice du noeud en cours, et on choisi celui ayant le plus grand, et dont on a pas déjà traité, c'est à dire qu'il ne figure pas dans la liste des arcs choisis correspondante au noeud courant.

la fonction qui détermine l'arc à traité est décrite comme suit, cette fonction retourne un élément de type "couple" ayant deux champs : A qui correspond à l'indice de la ligne et B l'indice de la colonne.

```
Arc
Regret ← 0
Pour i allant de 1 jusqu'à n faire
   Pour j allant de 1 jusqu'à n faire
      Si M[ i ][ j ] = 0 et regret < regret ( i , j ) et existe ( i , j ) = false faire
         Regret ← regret (i , j)
         X = ( i , j )
      Finsi
  Finpour
Finpour
Renvoie X
```

Avec " existe " une fonction qui retourne "true" si l'arc (i,j) figure dans la liste des arc fixé menant au noeud en cours. cette fonction est :

```
existe (i , j)
Pour k allant de 1 jusqu'à n faire
  Si liste [ k ] = ( i , j)
     Renvoie false
  Finsi
Finpour
Renvoie true
```

Création des fils

Chaque noeud de arbre possède deux fils sauf les feuilles représentent une solution réalisable au problème. le fils gauche est celui dont on a choisi l'arc, l'autre où l'on interdit l'ajout.

- Fils gauche : on ajoute l'arc (i , j) dans la liste des arc fixé. Afin de s'assurer qu'on aura pas de circuit dans notre solution, on lui interdit tous les arc lié à i et j et évidement l'arc (j ,i), cela on donnant ∞ à tous les éléments de la ligne i et la colonne j sans oublié l'arc (j , i).

  On calcul la borne inférieur correspondante à ce fils, pour cela on recalcule la borne inférieure de la nouvelle matrice. la borne inférieur du fils gauche est égale à l'addition de la valeur trouvé et borne inférieure du noeud courant.

- Fils droit : On interdit d'ajouter l'arc (i , j), pour cela il suffit de changer le poids de l'arc (i , j) en lui donnant une valeur ∞, dans ce cas la borne inférieure de ce fils sera celui du noeud en cours plus le regret de l'élément (i , j).

Fils gauche (i , j)	Fils droit (i , j)

Fils_gauche.liste ← courrant.liste + (i , j)
Matrice ← courrant.Matrice
Pour k allant de 1 jusqu'à n faire
  Si k ≠ i
  Matrice[k][j] ← ∞
  Finsi
Finpour
Pour k allant de 1 jusqu'à n faire
  Si k ≠ j
  Matrice[i][k] ← ∞
  Finsi
Finpour
Val ← Borne_inf (Matrice)
Fils_gauche.Matrice ← Matrice
Fils_gauche.Borne_inf ← courrant.borne_inf + Val
	Fils_gauche.liste ← courrant.liste
	Matrice ← courrant.Matrice
	Matrice[j][i] ← ∞
	Fils_gauche.Matrice ← Matrice
	Val ← regret (i , j)
	Fils_gauche.Borne_inf ← courrant.borne_inf + Val

Borne supérieure

C'est une estimation par excès de la fonction objectif. Autrement dit, la meilleure solution a nécessairement une valeur plus petite. Dans notre cas, un cycle Hamiltonien quelconque dans le graphe fournit une borne supérieure. une idée pour accélérer la recherche de la solution est de stocker la valeur de la meilleur solution trouver et d'interdire l'évaluation d'un nœud de l'arbre lorsque la valeur de sa solution partielle est supérieure à la meilleure solution déjà trouvée. afin d'accélérer la recherche de la solution, on initialise cette borne par une solution initiale évidente qui est le cycle $(x_1,x_2,...,x_n,x_1)$.

Solution réalisable

On dit qu'on est arrivé à une solution réalisable(feuille) lorsque la taille de la liste des arcs fixés est égale au nombre de sommets du graphe.

Feuille

Si la taille de la liste = n
  Renvoie true
Sinon
  Renvoie false
Finsi

Implémentation

Notre arbre de recherche est représenter sous forme d'une pile (Last-in Fisrt-out) de type Noeud.qui contient 3 champs : 1- la matrice des distances 2- la borne inférieure 3- la liste des arcs fixés.

Branch and bound

```
Initialisation de la pile avec un noeud initial
Initialisation de la solution par la solution réalisable (x1,x2,...,xn,x1).
Tant que l'arbre (pile) est non vide faire
   Noeud courant ← sommet de la pile
   Si courant est une feuille alors
      Si solution.Borne_inf < courant.Borne_inf alors
         Solution ← courant
      Finsi
      Supprimer courant de la pile
  Sinon
       X ← Arc
      Supprimer courant de la pile
      Creation des fils ( X )
      Ajouter Fils_gauche à la pile
      Si Fils_droit.Borne_inf < solution.Borne_inf
           Ajouter Fils_droit à la pile
      Finsi
   Finsi
FinTant que
```

Résultats expérimentaux

le programme a été exécuté sur un ordinateur Intel i5, CPU 1.75 GHz et 2,5 Go de RAM. Les jeux de données ont été téléchargés à partir de la page internet :

https://moodle1.u-bordeaux.fr/course/view.php?id=1597

Le tableau ci-dessous résume l'ensemble des résultats obtenus en appliquant les algorithme sur des instances de 2 à 10 sommets, nous avons trouvé les résultats suivants :

NOMBRE DE SOMMETS	RESULTATS	TEMPS D'EXECUTION (s)
2	200	0,02
3	341	0,02
4	400	0,02
5	441	0,06
6	848	0,062
7	890	0,07
8	1173	0,08
9	1338	0,08
10	1169	0,08

Méthodes approchées

Ce problème est plus compliqué qu'il n'y paraît ; on ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on devra donc souvent se contenter de solutions approchées, car on se retrouve face à une explosion combinatoire.

Dans ce travail on se propose deux différents heuristiques permettant de résoudre se type de problème avec des tailles plus ou moins grandes. que l'on comparera expérimentalement.

Pour le problème du voyageur de commerce, une heuristique gloutonne construit une seule solution, par une suite de décisions définitives sans retour arrière, parmi ces méthodes on cite le plus proche voisin et Construction par ajout d'arcs.

Construction par ajout du plus proche

La plus simple et la plus naturelle, consiste à se rendre systématiquement dans la ville la plus proche de celle où l'on se trouve, à la fin de l'algorithme en se retrouve avec une chaîne Hamiltonienne passante par tous les sommets du graphe une et une seule fois, on ajoute l'arc entre le dénier sommet de la chaîne et le sommet du départ pour créer un cycle : on pourrait

penser que cette méthode résout effectivement le problème posé. sauf quelle ne donne pas forcement la solution optimale.

Soit G un graphe de taille n, la matrice des distances du graphes est appelé M. cette méthode paut s'écrire comme suit :

Heuristique 1 : Le plus proche voisin

```
départ ← 1
iter ← 1
tant que iter < n-1 faire
   min ← chercher le plus proche sommet non visité lié au sommet départ
   solution[ iter ] = ( départ, min )
   iter ++
Fintantque
Solution[iter] ← (min , 1)
```

Construction par ajout d'arcs

La deuxième heuristique que nous présentons dans se travaille pour résoudre ce type de problème, consiste à construire le circuit de manière itérative de telle sort qu'on choisit à chaque itération l'arc de plus petit poids qui ne créera pas un circuit.

Afin de savoir si l'arc est valide ou pas, nous avons utiliser un tableau à n dimensions, où chaque éléments "i" désigne le sommet i, est dont chaque cases on stocke son successeur dans le cycle, se tableau se crée au fur et à mesure de la création du cycle. cette fonction est décrite comme suit :

Arc non valide (i , j)

```
Si l'arc est déjà traité
Renvoie true

Depart ← i
Iter ← 1
Tant que Depart ≠ -1 et iter < n - 1
   Si successeur[ depart ] = i
      Renvoie true
   Finsi
   Depart ← successeur [ Depart ]
   iter++
Fin tant que
Renvoie false
```

La deuxième méthode de résolution est donc :

Heuristique 2 : Ajout par arcs

```
tant que iter < n faire
   ( i , j) ← arc valide de plus petit poids
   solution[ iter ] = ( i, j )
   iter ++
Fintantque
```

Résultats expérimentaux

Dans cette section, nous présentons les résultats d'une étude de calcul qui vise à évaluer la performance de chacune des approches présentés précédemment.

A cet objectif, nous avons réalisé deux séries d'expériences, dont la première a pour but d'analyser l'impact de la taille du problème sur la qualité de la solution trouvée par chaque méthode (voir tableau1), et comme le temps d'exécution joue un rôle très important dans le monde de la recherche opérationnel, c'est à cause de lui que les problèmes d'optimisation combinatoire sont réputés si difficiles, nous avons effectué un ensemble de test où nous nous concentrons sur le temps que prend chaque méthode pour donner le résultat (voir tableau 2).

	HEUR1	HEUR2
a280	3157	3083
berlin52	8980	9563
kroA100	27807	26942
kroA150	33633	35047
kroA200	35859	40091
kroB100	29158	30305
kroB150	34499	34055
kroB200	36980	38035
kroC100	26227	29137
kroD100	26947	28799
kroE100	27460	27157

tableau 1 : comparaison des solutions optimales trouvés

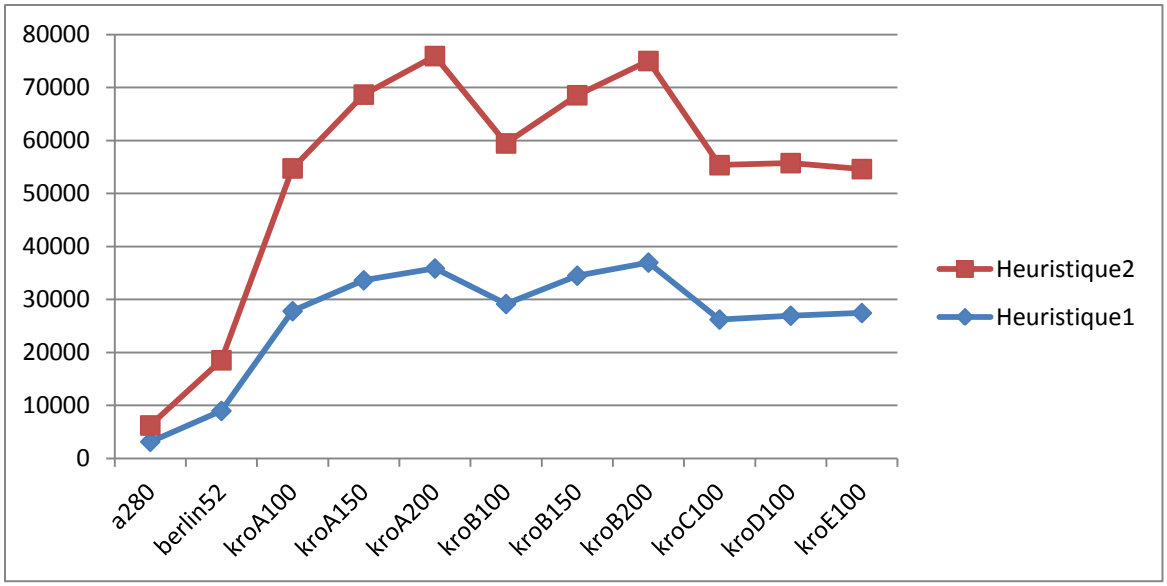
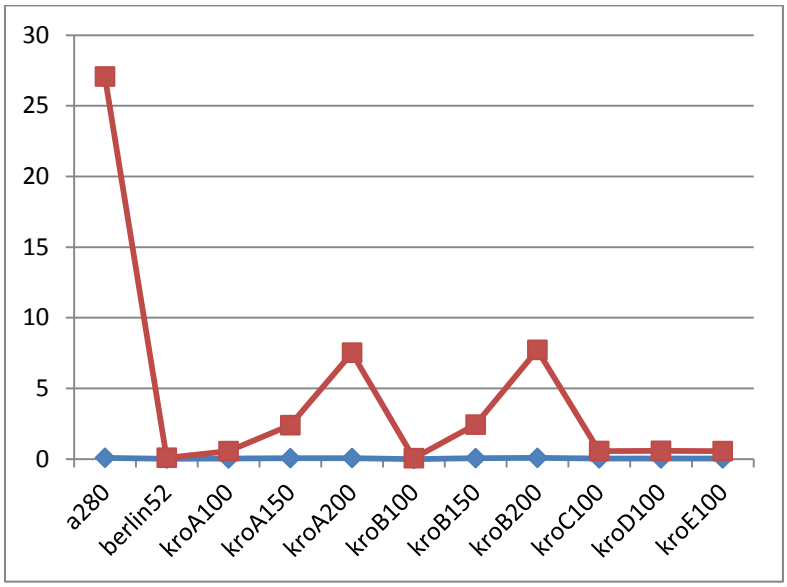


Figure 1 : comparaison des solutions

d'après la figure 1, nous pouvons très bien remarqué que l'heuristique 1 donne des bons résultats par rapport à la deuxième heuristique.

même en terme de temps d'exécution, nous trouve que la 1ère heuristique est plus rapide que la deuxième.



	HEUR1	HEUR2
a280	0,086	26,979
berlin52	0,026	0,064
kroA100	0,034	0,521
kroA150	0,061	2,331
kroA200	0,07	7,453
kroB100	0,0033	0,0537
kroB150	0,061	2,369
kroB200	0,0818	7,364
kroC100	0,045	0,519
kroD100	0,038	0,538
kroE100	0,04	0,518

3. Conclusion

Dans ce travail nous avons présenté 3 différentes méthodes pour résoudre le problème de voyageur de commerce, mais il existe d'autres méthodes fondées sur des principes totalement différents comme les métaheuristiquess.