1 Problème du voyageur de commerce (Traveling Salesman Problem)

Le problème du voyageur de commerce (TSP) est défini par : Soit un ensemble de n villes, et de distances entre chaque ville, il s'agit de trouver le plus court chemin passant exactement une fois par chaque ville et finissant à la ville de départ.

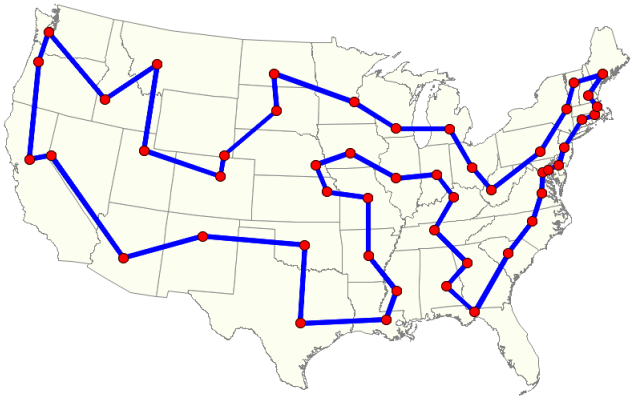


Figure 1: TSP illustration. Source: Robert Allison `http://robslink.com/SAS/democd62/new_94_sas.htm`

Depuis plusieurs dizaines d'année maintenant, le TSP a été au centre de l'attention des mathématiciens à cause de sa difficulté de résolution, malgré sa simplicité d'énonciation. De plus le TSP a une importance capitale en recherche opérationnelle et est représentatif de beaucoup de problèmes d'optimisation. Cela signifie que la résolution du TSP (cas général) aura un impact significatif sur beaucoup d'autres problèmes.

Le plus gros défi cependant est que le TSP appartient à la classe des problèmes NP-complets, pour lesquels il n'y a pas d'algorithmes connus qui permettrait de le résoudre (trouver la solution optimale) en temps polynomial. L'algorithme naïf, qui énumère toutes les routes possibles et choisis la meilleure ne permet pas de trouver une solution en un temps raisonnable (Table 1). Sa complexité est en $O(n!)$, où n est le nombre de villes

n	**Solutions** $(n-1!)$	**CPU time**
5	24	0.000
10	362.880	0.003 sec
15	8.7×10^{10}	20 min
20	1.2×10^{17}	73 years
25	6.2×10^{23}	470 millions years

Votre travail pour ce TP sera de creer un algorithme A* pour résoudre le TSP. Votre algorithme, n'explorera pas toutes les solutions mais priorisera les routes à l'aide d'une heuristique, et évitera les mauvaises routes, ou les routes peu promettantes.

2 Stratégie de recherche

2.1 Problème de représentation

Pour ce TP on utilisera la façon la plus commune pour représenter le TSP. Le TSP peut être représenter à l'aide d'un graphe complet, non orienté $G(V, E)$, où V est l'ensemble des villes, qui sont les sommets du graphe et E l'ensemble des arêtes. Comme G est complet, il a une arête en chaque paire de villes $(u, v), u \neq v \in V^2$. En d'autres termes, il y a chemin qui relie n'importe quelle ville à n'importe quelle autre. A chaque arête $e_{uv} \in E$ est associé un coût $c(e_{uv})$. Comme le graphe est non orienté $c(e_{uv}) = c(e_{vu})$. Le coût c représente une distance et satisfait donc toutes ses propriétés, notamment l'inégalité triangulaire.

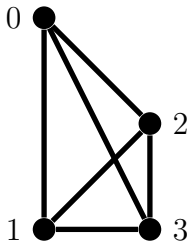
2.2 Arbre de solutions

La stratégie principale de l'algorithme est de faire une recherche sur un arbre de solutions. Chaque nœuds de cet arbre représente une solution partielle S (en construction). Cette solution S est composée d'un sous ensemble $E(S) \subseteq E$ d'arêtes et un sous ensemble $V(S) \subseteq V$ de sommets qui ont déjà été **visités** (atteint pas un sommet).

$$V(S) = \{u | \exists v \in V, (v, u) \in E(s)\}$$

Lorsque $V(S) = V$ alors la solution S contient une route complète et sous coût donné par $C(S) = \sum_{e \in E(S)} c(e)$. Notez que la ville de départ n'est pas comptée comme visitée.

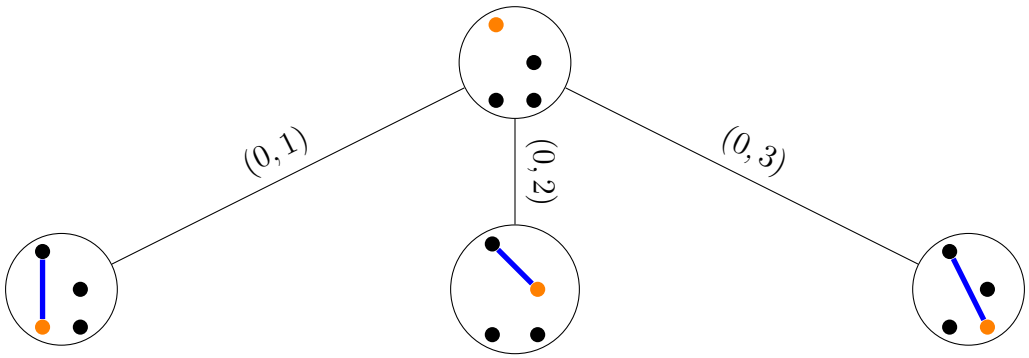
Construisons cet arbre avec un graphe à 4 sommets:



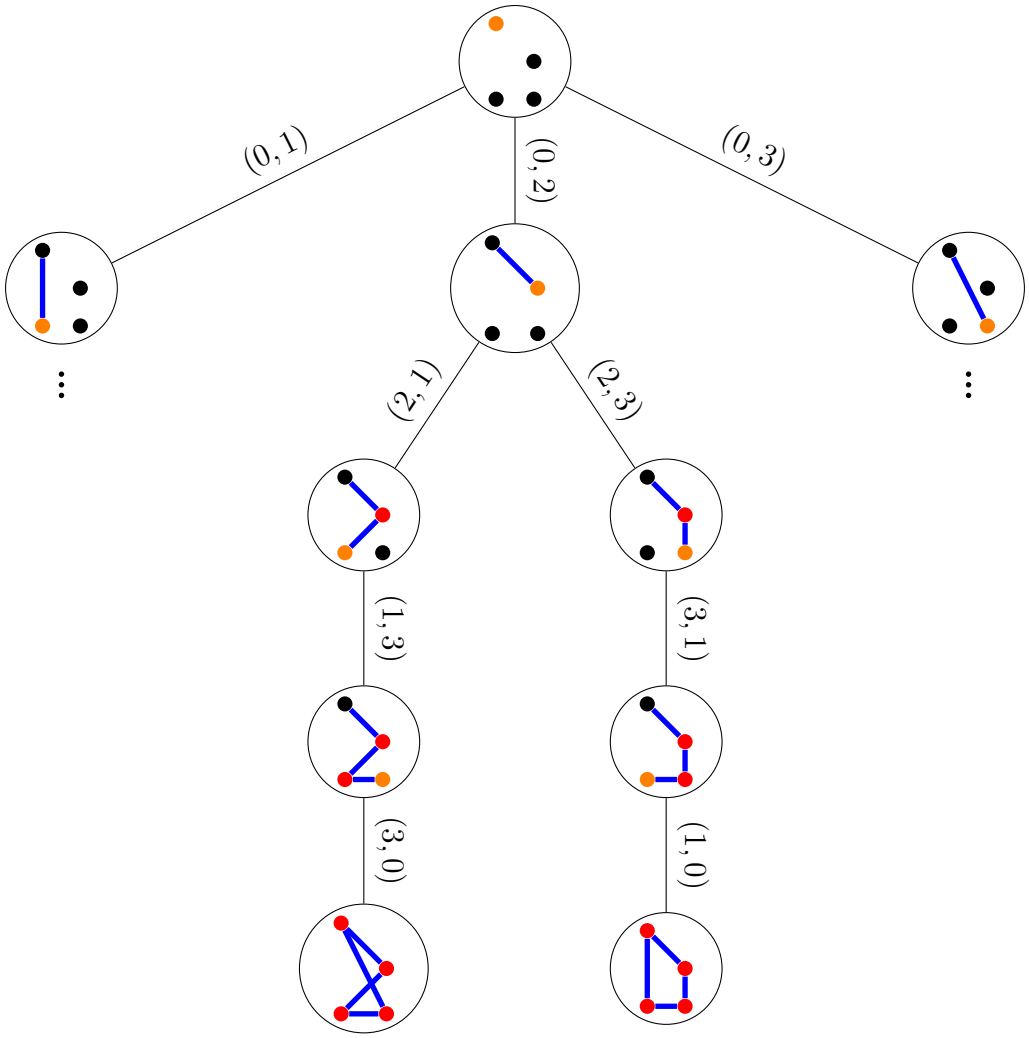
1. Partir de la racine avec une solution *vide* S, avec un ensemble d'arêtes et de sommets vides $E(S) = \emptyset, V(S) = \emptyset$. Sélectionner n'importe quelle ville comme le point de départ v_0 (0 dans notre exemple). Soit v le point courant. v est initialisé à v_0.

2. Pour chaque nœud non visité, $u \notin V(S)$, créer une nouvelle solution 'fille' S' telle que:

   - $E(S') = E(S) \cup \{(v, u)\}$
   - $V(S') = V(S) \cup \{u\}$
   - faire $v = u$



3. L'étape 2 est répétée pour chaque nœuds exploré pour construire des solutions complètes. Notez que l'arête qui retourne au point de départ doit être ajoutée uniquement si c'est le dernier point à visiter.



Si on décide d'utiliser une queue classique (premier entré, premier sorti) (FIFO) pour stocker les solutions, et si on défini un ordre d'exploration des nœuds, on finira par énumérer toutes les solutions possibles, pour trouver la meilleure. Cependant on ne souhaite pas visiter tous les solutions possibles, on va alors utiliser une file de priorité (**heap**) qui permet d'explorer en premier les solutions avec un bon potentiel, et qui permettra dans un second temps d'éliminer les solutions trop mauvaises. Cette file de priorité doit nous toujours nous permettre de trouver la bonne solution (celle de plus petit coût).

3 A vos ordinateurs

Note: Tous les fichiers pour ce TP sont fournis en C++ et python et sont disponible sur la page moodle du cours. Si vous n'êtes pas familiers avec C++ ou avec python n'hésitez pas à utiliser d'autres langages (Java, matlab ou autre) dans lesquels vous êtes plus à l'aise. Les directives de remise sont situés sur la dernière page du document. Des commentaires spécifique à chaque langage seront présents sur le sujet.

- **Graph.** le fichier `Graph.h/Graph.py` est un fichier prêt à être utilisé, il contient la classe graph utilisée pour représenter le problème. Un objet de cette classe a les 4 méthodes suivantes:
  - `Graph(dataset_file)` le constructeur qui prend en argument le nom du dataset (string) et construit le graphe.
  - `getN()` qui renvoie le nombre de sommets n (`.N` en python)
  - `getEdge(i,j)` qui renvoie l'objet edge correspondant à l'arête e_{ij} (`get_edge(i,j)` en python).
  - `getSortedEdges()` qui renvoie le vecteur des arêtes ordonnées par ordre de coût (croissant). On utilisera cette méthode plus tard (`get_sorted_edges(i,j)` en python).

  Un objet **edge** e a 3 attributs :
  - `e.source:` le sommet de départ;
  - `e.destination:` le sommet d'arrivée;
  - `e.cost:` le coût du sommet.

- **Solution.** La classe Solution (définie dans *Solution.h*) a aussi quatre variables :
  - `visited:` la liste de ville visitées,
  - `not_visited:` la liste des villes non visitées,
  - `cost:` le coût de la solution actuelle. On le notera g(i).
  - `graph:` le graphe du problème (de la classe *Graph*). cet variable est utilisée dans certaines méthodes de résolution *Solution*.
  - `source:` en python un champ source est implémenté permettant de garder transmettre la source

  Il s'agit maintenant de commencer à coder. Vous devez implémenter les 4 méthodes de la classe Solution dans le fichier *Solution.cpp* ou *Solution.py*.
  - `Solution(g):` Le constructeur de la classe Solution. IL prend en argument le *Graph* g et initialise tous les attributs de la solution. Cette méthode est utilisée pour créer le premier nœud de l'arbre.
  - `Solution(sol):` Le constructeur d'un nœud fils. Cette méthode utilise un autre nœud solution *sol*, et copie ses valeur dans les champs de la solution en construction. La nouvelle arête n'est pas ajoutée pour l'instant. Attention à bien copier et pas juste référencer.
  - `addEdge(v,u):` cette méthode ajoute une arête (u,v) à la solution et met à jour son coût. v est le sommet déjà visité de l'arête, u est le sommet à ajouter à la liste de sommets visités, (et à enlever de celle des sommets non visités).
  - `print():` affiche les arêtes de la solution et son coût. Cette méthode parcoure la liste *visited* en affichant les villes dans l'ordre de leur visite. le point de départ

devrait apparaître deux fois, en premier et en dernier élément. Voici un exemple d'affichage.

0
2
3
1
0
`COST` = 10

Note: Pour les constructeur de la classe Solution en python, une disjonction de cas est faite en fonction du type d'entrée, ce qui permet de faire les deux constructeurs en un. L'implémentation de addEdge peut pas mal varier en fonction des convention prises et du langage. Faites attention de garder la même et à rester cohérent tout au long du programme. *Il faudra détailler dans le rapport cette convention.*

- **A_star.cpp** C'est le fichier où est implémenté l'algorithme A*.

  La première chose à remarquer est la définition de la classe ***Node*** qui contient 3 attributs :

  - `solution:` un objet de type *Solution*.
  - `v:` le nœuds courant v
  - `heuristic_cost:` le coût de l'heuristique '*heuristic cost*' pour ce nœuds. Par la suite on le notera h(i).

  Pour rappel, dans l'algorithme A* on cherche à choisir le nœuds i qui minimise la fonction

  $$f(i) = g(i) + h(i)$$

  où g et h on été défini précédemment comme le coût actuel de la solution et le coût de l'heuristique. Par conséquent, `heuristic_cost` joue un rôle très important dans l'algorithme A*, puisqu'il aide à décider quel nœuds explorer. Plus le coût h de la solution est petite, plus le pontentiel du nœuds est grand. A partir de ce principe implémentez la méthode suivante pour comparer deux nœuds:

  - `isN2betterThanN1(N1,N2):` qui prend en argument deux objets *Node*, N1 et N2, et renvoie *true* si N2 doit être exploré avant N1 et *false* sinon. Cette fonction sera appelée pour ordonner les nœud entre eux.

  Pour l'instant on considère uniquement le coût actuel de la solution et on suppose donc que h est nulle. Dans la section 4 on changera cette valeur.

  Maintenant en utilisant la stratégie de recherche de la section 2, implémentez les deux méthodes suivantes :

- `main()`: qui crée l'objet solution initiale et le nœud racine de la liste et lance la recherche. Pour cela il parcoure la liste et pour chaque nœud regarde s'il est complet, si oui affiche la solution et termine sinon lance la méthode pour explorer le nœud fils.
- `explore_node(node)`: explore le nœuds en créant ses fils et les rajoutant à la liste.

Pour la liste on utilisera un liste de priorité, je vous invite à vous renseigner sur son fonctionnement. vous pouvez consulter la documentation ici : `http://www.cplusplus.com/reference/queue/priority_queue/`.

En python : `https://docs.python.org/2/library/queue.html` section priority queue. Il faudra en python implémenter la fonction `__lt__()` pour permettre à la queue de marcher

Exécution. Lancez maintenant l'algorithme A* sur le fichier **N10.data**, **N12.data** et reportez vos résultats (tous les résultats sont fournis à l'annexe A.):

- la solution affichée
- le nombre de nœuds explorés
- le nombre de nœuds crées ;
- le temps d'exécution (CPU et/ou ms).

Maintenant essayez de faire de même avec le fichier **N15.data**. Pouvez vous le résoudre? Cela prend probablement un peu de temps. Cela parce que l'heuristique est presque aussi mauvaise que l'algorithme naïf. Uniquement les pires solutions peuvent être évitées.

Une meilleure solution serait de regarder à chaque instant la solution courante et d'estimer son potentiel. On va avoir besoin d'une heuristique pour cela.

Note. Avant de réfléchir à l'heuristique, notez qu'il est possible de comparer des solutions partielles sur d'autres attributs que le coût. On peut en effet comparer le nombre de villes déjà visités. Détaillez dans votre rapport comment utiliser cette information.

4 A* Heuristic

Il faut souligner que pour l'algorithme soit correct, l'heuristique A* doit être admissible. Pour cela elle ne doit jamais surestimer le coût restant avant l'objectif, (ici visiter toutes les villes), en d'autre termes :

$$h(S) \leq min_{S' Solution, S \in S'} \{g(S') - g(S)\}$$

4.1 Arbre couvrant minimum(MST)

Un arbre couvrant minimum d'un graphe est un arbre couvrant (sous-graphe qui est un arbre et qui connecte tous les sommets ensemble) dont la somme des poids des arêtes est minimale. (Wikipedia)

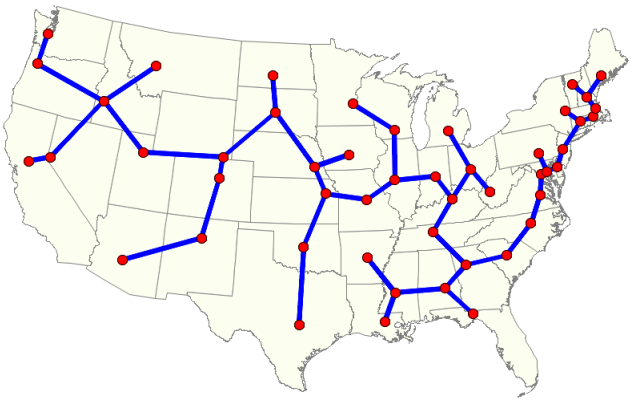
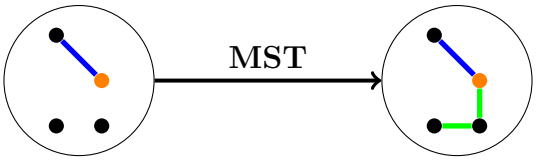


Figure 2: MST illustration. Source: Robert Allison `http://robslink.com/SAS/democd62/new_94_sas.htm`

Soit une solution partielle S, avec $V(S)$ les sommets visités et $E(S)$ les arêtes. Considérez maintenant un arbre couvrant minimum contenant $E(S)$ (qui n'est donc pas un arbre couvrant minimum de tout le graphe).

Montrez que pour parcourir toutes les villes en commençant par les la solution partielle il faut au moins le poids de cet arbre.

Ce poids est donc un minorant du poids restant de la solution et donc un bon indicateur de la pertinence de la solution.



En somme, le coût du MST représentera le coût de la solution (arêtes de $E(S)$) plus le coût minimum pour atteindre tous les autres sommets. Ce coût est une bonne approximaiton du potentiel de la solution et on va l'utiliser pour l'heuristique.

Un des algorithmes les plus connus pour trouver un arbre couvrant minimum, est celui de Kruskal [1], qui prend un temps $O(mlog(n))$, avec m le nombre d'arêtes et n le nombre de sommets. Voici le pseudo code:

Kruskal's algorithm

1: $MST \leftarrow \emptyset$ // initializes the tree as empty
2: Sort the edges
3: **for each** edge e **in** the sorted order **do**
4: **if** e does not create a cycle in MST **then**
5: $MST \leftarrow T \cup \{e\}$
6: **end if**

[1] Kruskal's algorithm animation: `https://en.wikipedia.org/wiki/Kruskal%27s_algorithm`

7: **end for**
8: **return** MST // return the MST

- **Kruskal** une classe *UnionFind* vous est fournie pour trouver un cycle dans l'arbre. Elle contient deux méthodes :
  - `isMakeCycle(e):` qui renvoye true si et seulement si un cycle est formé lors du rajout de e.
  - `add(e):` qui ajoute l'arête e à la structure. Cette méthode doit être appelée après chaque insertion d'une arête dans l'arbre T (ligne 6 de l'algorithme) pour que la structure reste à jour.

  Maintenant implémentez l'algorithme du MST, en complétant la méthode suivante dans le fichier *Kruskal.cpp*.
  - `getMSTCost(sol,sp):` renvoye le coût du MST obtenu pour la solution S (uniquement le coût des arêtes non présentes dans S). N'oublier pas d'ajouter les arêtes au fur et à mesure dans la solution. Implémentez ensuite l'algorithme pour prendre en compte cette heuristique.

Execution. Lancez maintenant l'algorithme sur le fichier **N10.data**, **N12.data**, et **N15.data**. Notez et commentez les améliorations.

4.2 Affiner la borne inférieure

En plus de l'heuristique du MST on peux ajouter deux valeurs pour améliorer le coût de l'heuristique:

1. **le distance de la ville actuelle à v, la ville la plus proche**
   Ce coût va améliorer la borne puisqu'il va évaluer la meilleure arête à ajouter à la solution.
2. **la plus petite distance entre une ville non visitée et le point de départ**
   Ce coût va améliorer le borne puisqu'il va sélectionner la meilleure arête pour revenir au point initial. **Question:** Si vous ajoutez ce coût à l'heuristique, peut on supposer que la première route complète trouvée est la meilleure? Pourquoi?

Implementez ces deux valeurs et incorporez les à l'heuristique, dans la méthode `explore_node`.

Exécution. Refaites vos expériences, que constatez vous?.

5 Bonus

1. Pouvez-vous approcher la solution optimale du problème N17.data avec d'autres techniques de recherches vues en classe (ne garantissant pas nécessairement un résultat optimal)?

2. Pouvez vous améliorer l'heuristique proposée dans le TP?

6 Directives de remise

Le travail sera réalisé en équipe de deux ou trois. Vous remettrez un fichiers .zip par personne, nommés TD1_nom_prenom_matricule.zip (par moodle). Vous devez également remettre un fichier pdf contenant une explication de vos implémentations ainsi que les réponses des questions et les toutes les réflexions que vous trouvez pertinentes. Vous devez montrer que vous avez compris le fonctionnement de l'algorithme. Tout devra être remis avant le **lundi 2 octobre à 23h55**. Tout travail en retard sera pénalisé de 10% par jour de retard.

Barème :

implémentations:	50pts
résultats:	35 pts
rapport :	15pts

A Résultats

Dataset	Optimal Cost
`N10.data`	135
`N12.data`	1733
`N15.data`	291
`N17.data`	2085

*Ce TP a été imaginé par Daniel Aloise, développé par Rodrigo Randel, traduit et corrigé par Pierre Hulot.