**INF8775 – Analyse et conception d’algorithmes**

TP2 – Hiver 2023

| **Nom, prénom, matricule des membres** | CARON, Thomas, 1944066  MESSEDI, Mohamed-Ali, 1946597 |
| --- | --- |
| **Note finale / 13** |  |

# Informations techniques

* Répondez directement dans ce document. Veuillez ne pas inclure le texte en italique servant de directive.
* La correction se fait sur ce même rapport.

Vous devez faire une remise électronique sur Moodle avant le

14 mars à 23h59 pour le groupe B2

21 mars à 23h59 pour le groupe B1

en suivant les instructions suivantes :

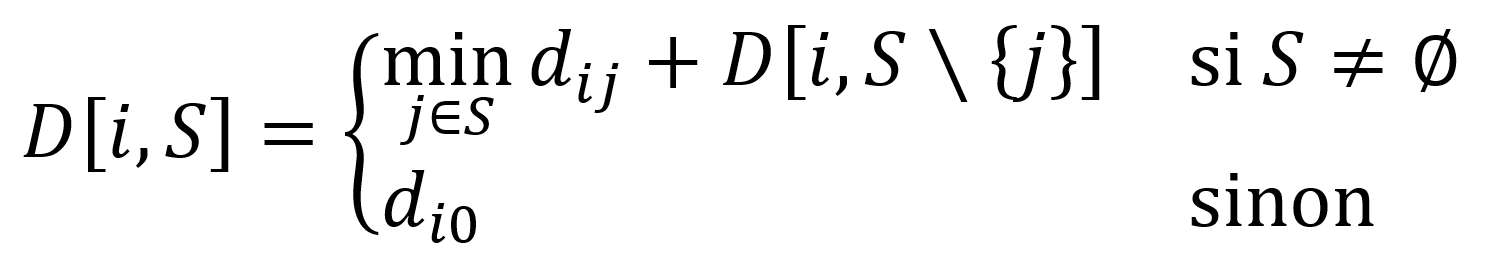
* Vos fichiers doivent être remis dans une archive zip à la racine de laquelle on retrouve :
  + Ce rapport sous format ODT ou docx.
  + Un script nommé *tp.sh* servant à exécuter les différents algorithmes du TP. L’interface du script est décrite à la fin du rapport.
  + Le code source et les exécutables.
* Vous avez le choix du langage de programmation utilisé mais vous devrez utiliser les mêmes langage, compilateur et ordinateur pour toutes vos implantations. **Le code et les exécutables soumis devront être compatibles avec les ordinateurs de la salle L-4714.**
* Si vous utilisez des extraits de codes (programmes) trouvés sur Internet, vous devez en mentionner la source, sinon vous serez sanctionnés pour plagiat.

# Mise en situation

Ce travail pratique se répartit sur deux séances de laboratoire et porte sur l'analyse et la conception d'algorithmes développés suivant différents patrons de conception.

On vous demande d’y résoudre le problème du voyageur de commerce (*Traveling Salesman Problem*, TSP) en utilisant trois patrons de conception distincts :

1. Algorithme glouton : en démarrant d’une ville arbitraire et en faisant une tournée en sélectionnant à chaque fois le plus proche voisin ;
2. Algorithme de programmation dynamique : en résolvant le problème à travers la relation de récurrence suivante :



1. Algorithme approximatif (1-relatif) : en construisant un arbre sous-tendant minimum (Minimum Spanning Tree, MST) du graphe et en le parcourant en préordre à partir d’une ville arbitraire.

On s’intéresse dans ce TP au problème du TSP métrique : les villes appartiennent à un plan N2 muni de la distance euclidienne **arrondie à l’entier le plus proche**.

# Jeux de données

Pour tester les algorithmes, vous devez générer des jeux de données en utilisant le script fourni :

$ ./inst\_gen.py [-h] -s NB\_VILLES [-n NB\_EXEMPLAIRES] [-x PRÉFIXE]

On suggère d’utiliser le script bash suivant pour automatiser la génération des exemplaires :

| #!/bin/bash  # Pour glouton et approx  for n in {"1000","5000","10000","50000","100000"}; do  ./inst\_gen.py -s $n -n 5  done  # Pour tous les algorithmes  for n in {"5","10","15","20","25"}; do  ./inst\_gen.py -s $n -n 5 -x DP  done |
| --- |

La première ligne de chaque exemplaire contient le nombre de villes à visiter (qui est aussi la taille du problème *N*). Les *N* lignes subséquentes contiennent chacune les coordonnées d’une ville à la fois, séparées par des espaces.

Les coordonnées des villes sont chacune comprises entre 0 et 2000 et aucune paire de villes n’est telle que la distance entre elles est de 0.

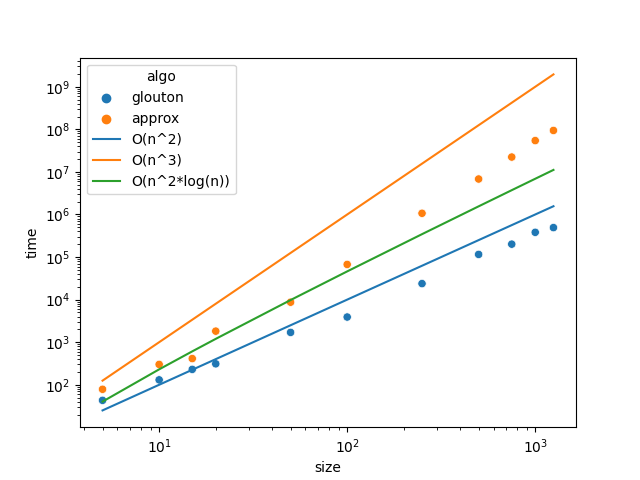
# Présentation des résultats

| 0 | / 4 pt |
| --- | --- |

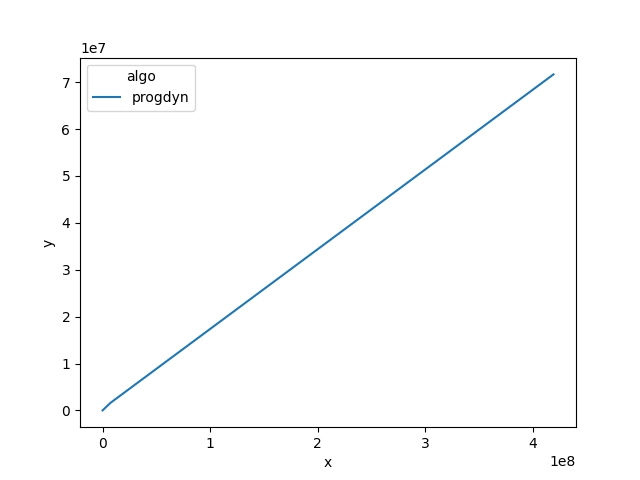
### Tableau des résultats

| **Algorithme** | **Approximatif** | | **Glouton** | | **Dynamique** | |
| --- | --- | --- | --- | --- | --- | --- |
| **Nombre de villes** | **Temps** | **Coût** | **Temps** | **Coût** | **Temps** | **Coût** |
| **5** | 78.33 | 3606.88 | 43.07 | 3784.71 | 198.08 | 3477.15 |
| **10** | 300.90 | 6361.54 | 130.35 | 6433.68 | 19217.88 | 6094.30 |
| **15** | 412.56 | 7709.58 | 229.91 | 7440.36 | 1608405.68 | 6910.18 |
| **20** | 1819.19 | 9738.85 | 310.71 | 8940.56 | 71653286.17 | 7677.28 |
| **50** | 8702.36 | 15036.83 | 1698.83 | 14817.86 | - | - |
| **100** | 67119.57 | 20125.94 | 3901.55 | 19587.45 | - | - |
| **250** | 1067553.25 | 32195.07 | 23800.02 | 29123.67 | - | - |
| **500** | 6813378.42 | 46017.41 | 115356.70 | 42110.23 | - | - |
| **750** | 22345158.59 | 53984.77 | 201024.96 | 50558.20 | - | - |
| **1000** | 54451952.18 | 63871.80 | 381073.83 | 58511.06 | - | - |
| **1250** | 94185657.75 | 71458.62 | 492043.60 | 65462.25 | - | - |

### Graphiques pour analyse hybride

****

Test de puissance (approximatif et glouton)



Test des constantes (dynamique)

### Analyse et discussion

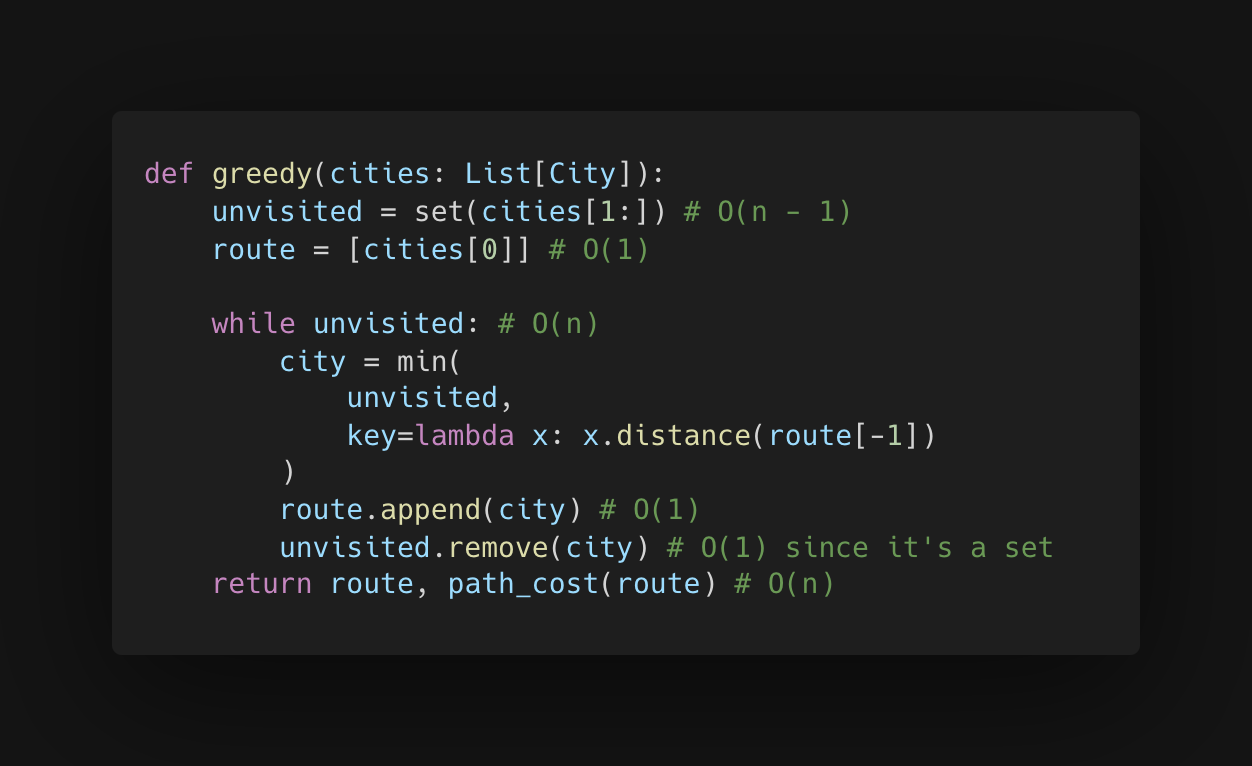
| 0 | / 6 pt |
| --- | --- |

### Faites une analyse asymptotique du temps de calcul pour chaque algorithme.

| **Algorithme** | **Complexité** |
| --- | --- |
| Glouton | O(n^2) |
| Dynamique | O(2^n \* n^2) |
| Approximatif | O(n^3) |

Analyse asymptotique du temps de calcul pour chaque algorithme:

Glouton



En ce qui concerne l'analyse asymptotique de l'algorithme glouton, elle est relativement simple. Ici, on voit que pour la création du set de villes non visitées, c'est une opération qui se fait en O(N-1), donc d’ordre **O(N)**. Ensuite, on a l'opération d'affectation de la route initiale qui se fait en temps constant.

Dans la boucle, pour chaque itération, il faut parcourir toutes les autres villes pour trouver la ville la plus proche non visitée. Comme la boucle principale se fait **n** fois et que pour chaque itération elle effectue **n-1** comparaisons, on a une complexité qui est de l'ordre de O(N\*(N-1)) donc **O(N^2)**.

Il s'agit de la section critique, le reste se fait en O(N), donc on gardera **O(N^2)**.

Dynamique



La complexité temporelle de cet algorithme est O(N^2 \* 2^N), avec N le nombre de villes dans la liste de départ. En effet, la fonction de distance va être appelée de façon récursive pour chaque sous-ensemble de villes et il existe 2^N sous ensembles.

La fonction du min à l'intérieur de la fonction de distance va se faire en O(N) pour trouver le coût minimum. Elle est appelée N fois sur chaque sous-ensemble. On a donc un O(N^2) pour chaque sous-ensemble.

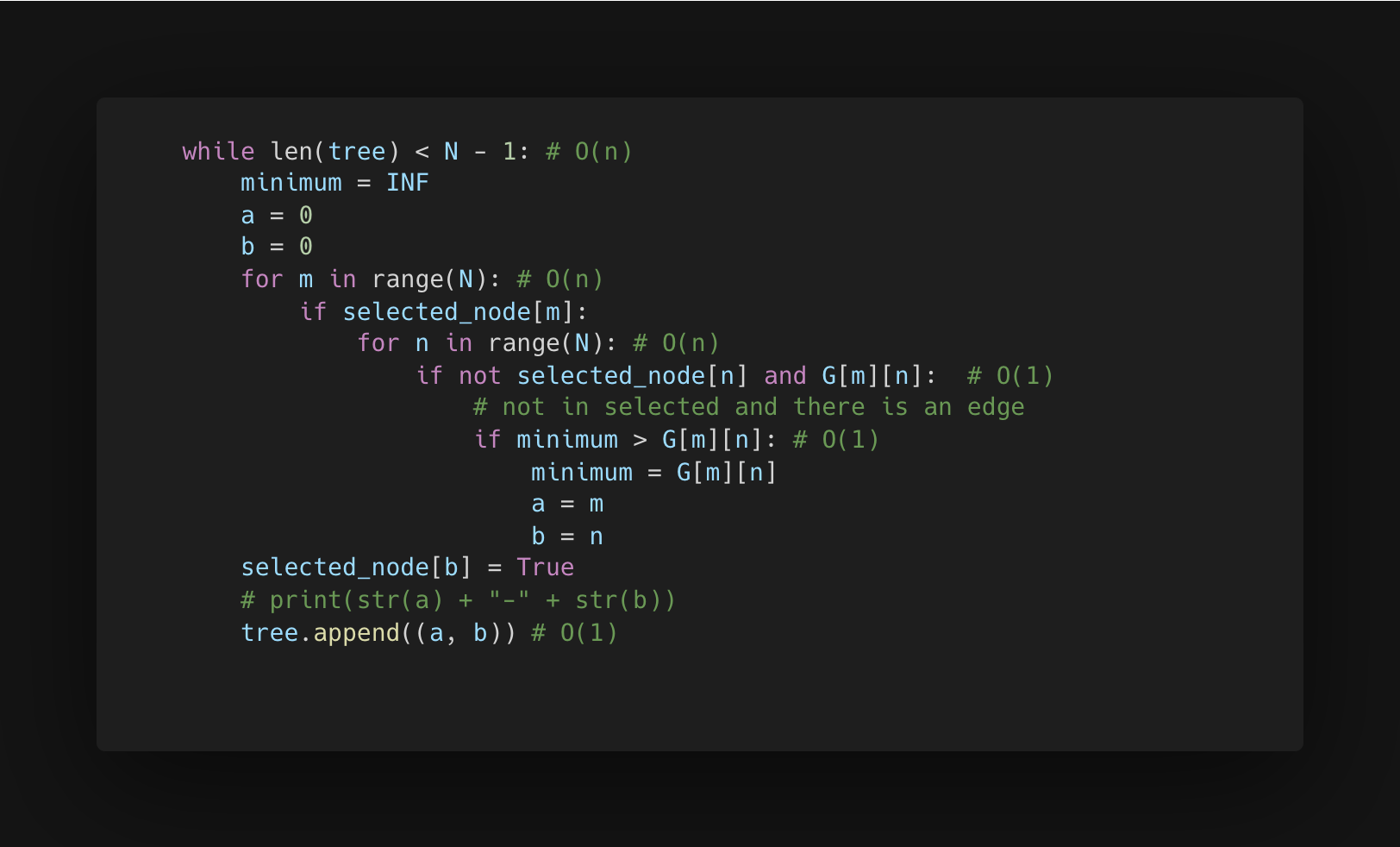
Au final, on aboutit avec 2^N sous-ensembles, et donc une complexité temporelle de O(N^2 \* 2^N). À noter qu'avec le décorateur functools.lru cache, on utilise le principe de memoization et on réduit légèrement le temps d'exécution car les distances précédemment calculées sont ré-utilisées. En mettant *maxsize=None*, on précise qu’on ne veut pas utiliser le fonctionnement LRU (Least Recently Used) de la cache. Ceci fonctionne jusqu’à N = 20 villes, cependant pour un chiffre plus grand, celle-ci n’est pas recommandée car elle devient trop grande (20^2 \* 2^20 entrées).

Algorithme d'approximation

La première ligne consiste à faire l’appel à cette ligne qui s’occupe de faire un parcours préfixe de l’arbre retourné par l’algorithme de prim:



Ici, analysons tout d'abord la fonction prim() :



L'algorithme commence par faire des initialisations pour certaines variables et structures de données, ce qui prend un temps constant.

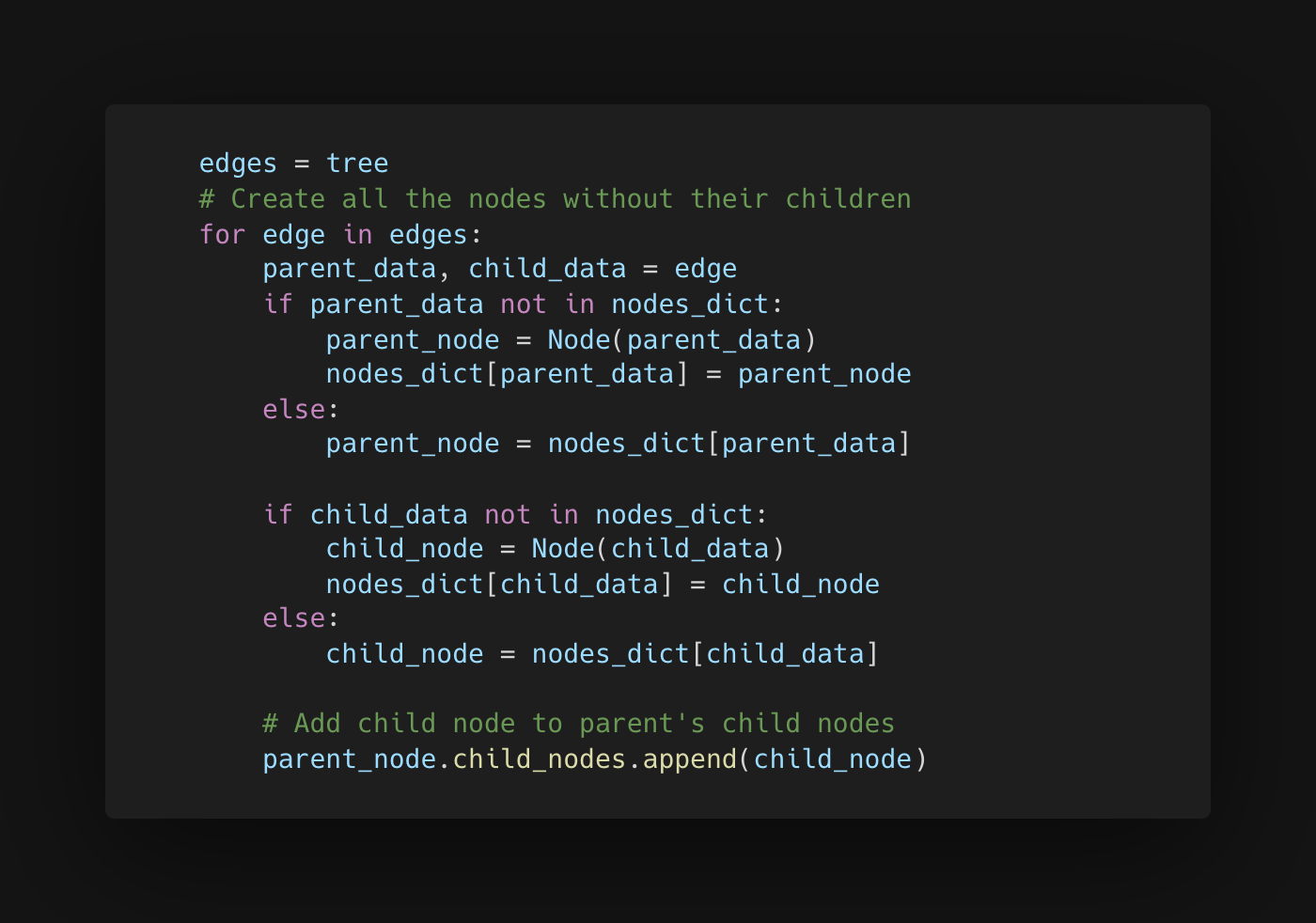
Ensuite, l'algorithme entre dans une boucle while. Elle s'exécutera N-1 fois, car un arbre couvrant minimum possède N-1 arêtes.

Dans la boucle while, l'algorithme va itérer sur chaque paire de nœuds du graphe. Cela va prendre s'exécuter en O(N^2).

Pour chaque paire de nœuds, l'algorithme va vérifier si les nœuds sont sélectionnés et s'il existe une arête entre eux. Si c'est le cas, l'algorithme mettra à jour une valeur minimale et les variables a et b. Cela se fait en temps constant.

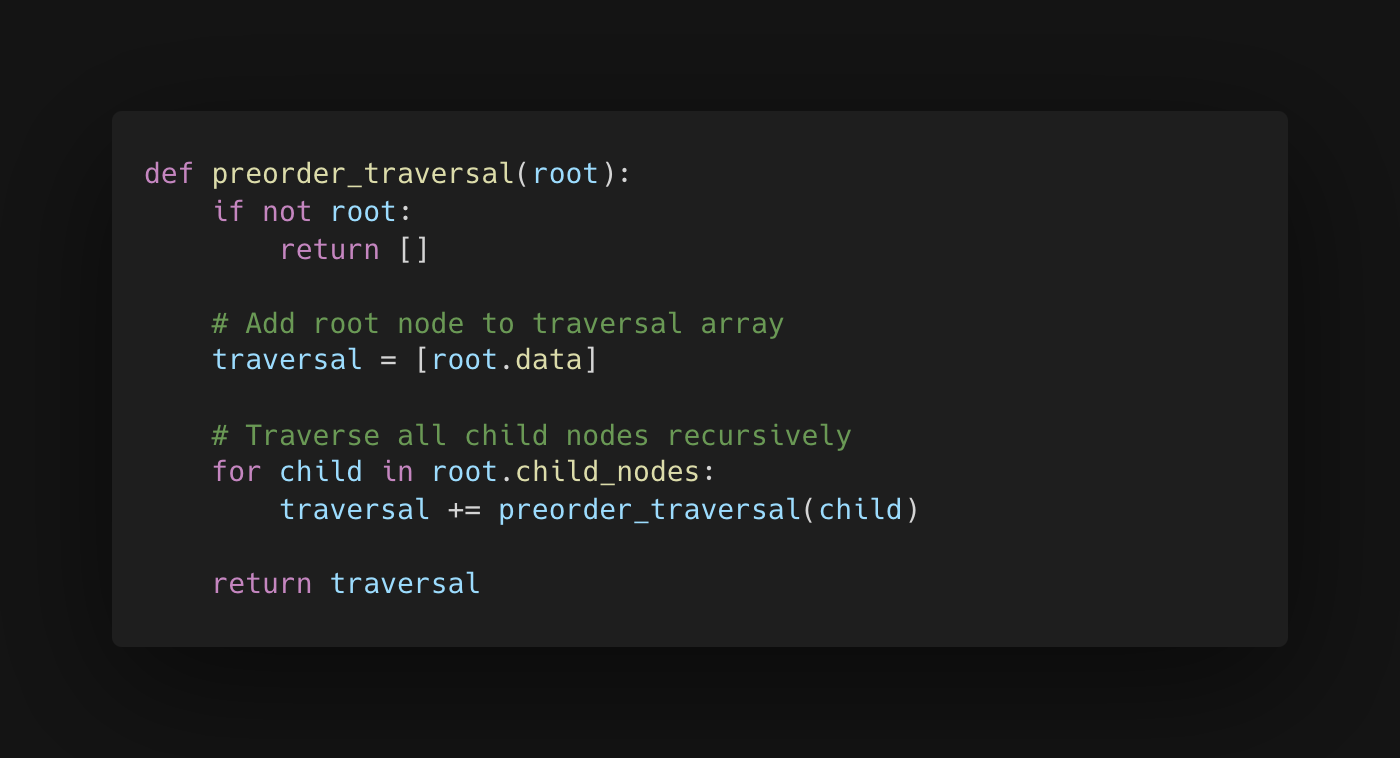
Après la boucle for, l'algorithme sélectionne le nœud b et ajoute l'arête (a,b) obtenue à l'arbre. Tout cela se fait en temps constant.

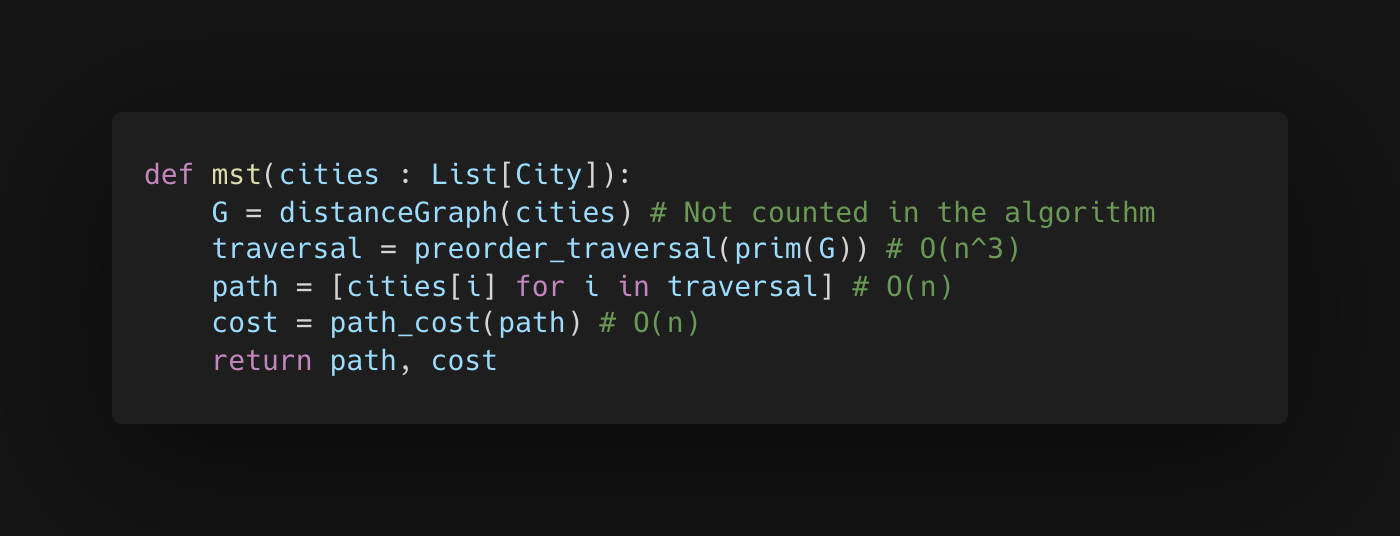
La deuxième partie critique du code est la création de l'arbre :



Ici, l'algorithme opère sur toutes les arêtes de l'arbre, ce qui prend un temps O(N). Ensuite, on vérifie pour chaque arête si les noeuds parent et enfant ont déjà été créés dans le dictionnaire de noeuds. Si ce n'est pas le cas, on les crée, et on ajoute le nœud enfant à la liste child\_nodes du parent. Cette partie prend un temps constant.

En conclusion : la section dominante est la boucle imbriquée à l'intérieur du while. Cela nous donne un O(N^2 \* N), ce qui donne O(N^3). À noter que si une structure de données efficace est utilisée pour trouver l'arête minimale on peut avoir une complexité plus optimale, dans l'ordre du O(Nˆ2 log N).

Finalement, l'objet retourné est la racine de l'arbre sous tendant minimum.   
Il suffit ensuite d'en faire la traversée en préordre : 

La complexité temporelle de cette opération est en O(N), avec N le nombre de nœuds dans l'arbre. En effet, l'algorithme va visiter chaque nœud de l'arbre exactement une fois et le temps nécessaire est constant. 

Finalement, on se retrouve avec une complexité totale en O(N^3).

### Servez-vous de vos temps d'exécution pour confirmer et/ou préciser l'analyse asymptotique théorique de vos algorithmes avec la méthode hybride de votre choix.*.*

Dans cette section, nous allons présenter les graphiques générés à partir des données de temps d'exécution mesurées sur nos implémentations pour chaque algorithme.

Résultat du test de puissance

En ce qui concerne le test de puissance, on a pu calculer les facteurs polynomiaux pour les méthodes d'algorithme Approximatif et de Glouton.

On obtient :

**Approximatif**: y = 0.591312124109065 \* x^2.617813527833828

**Glouton**: y = 2.1144458344633907 \* x^1.7254110780388703

Ici, on voit que les facteurs polynomiaux obtenus sont assez similaires à l'hypothèse théorique émise au départ. On voit que le facteur polynomial de la méthode Glouton est proche de 2, et que celui de la méthode de l'algorithme approximatif est proche de 3.

Notre test de puissance nous permet donc de confirmer les hypothèses de complexité théorique émises pour la méthode glouton et approximative.

Note : On ne considère pas l'algorithme dynamique parce qu'il n'a pas un facteur polynomial donc le test ne s'applique pas. Sa complexité de O(2^N \* N^2) est exponentielle, et donc les résultats seront faussés. On s'attaque à cet algorithme dans le test des constantes (section suivante).

Résultat du test des constantes

En ce qui concerne le test des constantes, il concerne la méthode qui fait appel à l'algorithme de programmation dynamique. Nous avons comparé les données mesurées avec notre programme au temps théorique estimé qui est en O(2^N \* N^2).

Malgré le fait qu'il n'y ait pas beaucoup de points (car au-delà d'un certain seuil l'algorithme prend trop de temps) nous pouvons appliquer la méthode, mais elle est à considérer avec attention, car le résultat aurait probablement été différent avec plus de points.

On sait que quand on fait un test des constantes, si on peut faire passer une droite dans le graphe alors on fait une bonne estimation. Ici, on voit que c'est le cas, car on peut faire passer une droite linéaire par les points obtenus. Ainsi, on a une estimation correcte.

Notre hypothèse selon laquelle le temps d'exécution s'apparente à la fonction que nous avons défini est confirmée, et donc que nos mesures sont correctes.

### Discutez des trois algorithmes en fonction de la qualité respective des solutions obtenues, de la consommation de ressources (temps de calcul, espace mémoire) et de la difficulté d'implantation.

Discussion pour le patron glouton

Le patron glouton est celui qui consomme le moins de ressources. En effet, à part le tableau de NxN villes, il n'utilise qu'un set pour les villes non visitées. L'algorithme fait essentiellement une itération pour chaque ville non visitée et utilise la fonction lambda pour calculer la distance avec la ville précédente. Ensuite, il y a suppression de la ville visitée dans le set initial. Cela nécessite un espace en O(N).

Cette simplicité d'implantation en fait le meilleur au niveau de la mémoire consommée, et donc de la complexité spatiale.

En ce qui concerne la complexité temporelle, l'exécution se fait en O(N^2). De plus, les résultats obtenus avec nos mesures nous permettent de confirmer qu'il s'agit du plus rapide entre les trois méthodes.

Discussion pour la programmation dynamique

Pour la programmation dynamique, on a une complexité spatiale considérable, car on utilise une approche diviser pour régner et une fonction de memoization pour stocker les coûts minimaux pour chaque ville et chaque sous-ensemble, ce qui consomme beaucoup de mémoire.

Comme on considère tous les sous-graphes possibles formés à partir du graphe initial, cela nécessite un espace en O(N\* 2ˆN). En effet, le nombre total de sous-graphes possibles pour N sommets est 2^N.

En termes de complexité temporelle, l'analyse théorique nous a montré qu'il s'agissait de l'algorithme le moins efficace avec un temps en O(2^N \* N^2).

Les temps d'exécution mesurés en laboratoire ont également confirmé cette hypothèse. La récursion divise l'ensemble des villes en sous-ensembles et appelle la fonction de calcul de distance minimale pour chaque ville.

Cela se fait jusqu'à ce que toutes les villes soient visitées. Tout ce processus prend un temps d'exécution exponentiel, et le temps pris devient non viable passé une certaine taille d'échantillon.

Discussion pour l'algorithme d'approximation

Finalement, l'algorithme d'approximation a un temps de complexité temporelle en O(N^3). En effet, en regardant les valeurs, on voit que les temps d'exécution deviennent rapidement supérieurs à ceux de l'algorithme glouton, ce qui confirme notre hypothèse.

En ce qui concerne la complexité spatiale, l'algorithme doit stocker le graphe d'entrée, qui est une matrice NxN. Ensuite, il faut aussi stocker l'arbre sous-tendant minimal. Pour ce faire, il faut garder la liste des sommets visités et non visités, ce qui nécessite un tableau de booléens de taille N. Il faut également stocker les arêtes qui composent l'arbre couvrant minimum, qui sont au maximum N-1 dans un graphe de N sommets. Ainsi, on a une complexité spatiale en O(N^2). La complexité spatiale est également affectée par la fonction de DFS appelée pour faire le parcours en préordre du graphe résultant de l'algorithme de Prim.

### Indiquez sous quelles conditions vous utiliseriez chaque algorithme.

**Algorithme Dynamique :**

L'algorithme dynamique est très précis, mais sa complexité temporelle est très élevée. Ainsi, on peut l'utiliser quand il y a un besoin d'avoir la solution optimale.

Cependant, il ne faut pas que l'échantillon utilisé soit grand, sinon le temps requis peut être hors de proportions. En effet, pour un graphe de taille 15 villes, l'algorithme prend déjà 26 minutes. Ainsi, la taille de l'échantillon doit impérativement être basse.

Ainsi, lorsqu'il faut trouver la solution optimale à tout prix, c'est le bon choix, mais il faut garder en tête la complexité temporelle d'un tel choix.

**Algorithme Glouton :**

L'algorithme glouton est en termes de complexité temporelle et de coût le meilleur choix entre les trois méthodes. En effet, c'est le plus rapide, avec une complexité de O(N^2).

En revanche, en termes d'optimalité, il est moins efficace que les deux autres méthodes. Il faut donc prendre ce facteur en compte lors du choix.

En conclusion, l'algorithme Glouton devrait être favorisé quand les données sont larges pour avoir une exécution efficace, et quand ce facteur est plus critique que d'avoir la solution optimale.

**Algorithme Approximatif :**

Après avoir mené une analyse comparative entre les algorithmes approximatif et glouton pour le laboratoire, on voit que l'algorithme approximatif est plus performant que l'algorithme glouton pour des échantillons de taille inférieure à 15 villes.

Cependant, cette supériorité est relative, car l'algorithme approximatif reste moins efficace en termes de rapidité. Néanmoins, pour un petit échantillon de villes, l'algorithme approximatif pourrait représenter un bon compromis entre l'algorithme glouton et l'algorithme dynamique.

En effet, bien qu'il ne soit pas aussi précis que l'algorithme dynamique, il permettrait d'avoir un coût relativement efficace pour un temps bien inférieur à celui requis par l'algorithme dynamique.

Cependant, pour des échantillons de tailles plus importantes, la méthode glouton reste la meilleure option, car elle permet d'obtenir une solution proche de l'optimale en un temps raisonnable.

**Conclusion :** En somme, le choix de l'algorithme à utiliser dépend de plusieurs facteurs, notamment la taille de l'échantillon et les exigences de précision et de temps de traitement, mais aussi de notre implémentation.

# On vous fournit 5 exemplaires difficiles à résoudre jusqu’à optimalité[[1]](#footnote-0) (fichiers avec préfixe hard). Tentez de les résoudre à l’aide de vos algorithmes glouton et approximatif et discuter des écarts obtenus.

| Fichier | Nombre de villes | Solution optimale (L2 arrondie) |
| --- | --- | --- |
| hard\_N52 | 52 | 551609 |
| hard\_N91 | 91 | 1228726 |
| hard\_N130 | 130 | 1928734 |
| hard\_N169 | 169 | 2600546 |
| hard\_N199 | 199 | 3139778 |

| Nombre de villes | Solution Optimale | Glouton | | Approximation | |
| --- | --- | --- | --- | --- | --- |
| Coût Relatif | Écart (%) | Coût Relatif | Écart (%) |
| 52 | 551609 | 98615 | 18% | 202526 | 37% |
| 91 | 1228726 | 109287 | 9% | 198151 | 16% |
| 130 | 1928734 | 196403 | 10% | 276559 | 14% |
| 169 | 2600546 | 223566 | 9% | 1026034 | 39% |
| 199 | 3139778 | 207826 | 7% | 502542 | 16% |

On voit ici que les résultats sont assez cohérents avec nos hypothèses. En effet, nous avions conclu auparavant que pour des données simples l'algorithme approximatif performait mieux que la méthode Glouton. Cependant, dès que les données se complexifient, l'algorithme Glouton devient meilleur en termes de complexité temporelle mais aussi d'optimalité.

C'est effectivement ce qu'on voit ici. L'algorithme glouton a un écart significativement meilleur que son rival. De plus, la solution du glouton n'est pas très éloignée de la solution optimale, ce qui est également un indicateur de précision de celui-ci.

Cependant, il y a quelques enjeux à prendre en considération. Selon notre réflexion, construire un arbre sous-tendant minimum et le parcourir en préordre devrait au moins être sensiblement moins bon que l'algorithme glouton. Ici il y a une différence trop importante. Plusieurs facteurs pourraient expliquer cette variation élevée, en particulier l'implémentation que nous avons choisi.

# Autres critères de correction

## Respect de l’interface tp.sh

|  | / 1 pt |
| --- | --- |

Utilisation :

$ ./tp.sh -a {glouton, progdyn, approx} -e CHEMIN\_EXEMPLAIRE [-p] [-t]

Arguments optionnels :

-p affiche dans l’ordre, sur chaque ligne, les indices des villes à visiter en commençant par 0 et en finissant par 0, sans texte superflu. Rapportez le chemin tel que la deuxième ville visitée ait un indice inférieur à celui de l’avant dernière ville affichée (voir exemple).

-t affiche le temps d’exécution en millisecondes, sans unité ni texte superflu.

Important : l’option -e doit pouvoir accepter des chemins absolus.

user@host folder $ ./tp.sh -p -e N5\_0 -a progdyn

0

**1**

4

3

**2**

0

## Qualité du code

|  | / 1 pt |
| --- | --- |

## Présentation générale

|  | / 1 pt |
| --- | --- |

* Concision
* Qualité du français

## Pénalité retard

| 0 |
| --- |

* -1 pt / journée de retard, arrondi vers le haut. Les TPs ne sont plus acceptés après 3 jours.

1. Tirés de <http://www.or.uni-bonn.de/~hougardy/HardTSPInstances.html>, les coordonnées peuvent dépasser 2000. [↑](#footnote-ref-0)