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

TP2 – Hiver 2023

| **Nom, prénom, matricule des membres** | YANG, Hee-Min, 1961471  LAURIN, Charles-Antoine 2070729 |
| --- | --- |
| **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 :

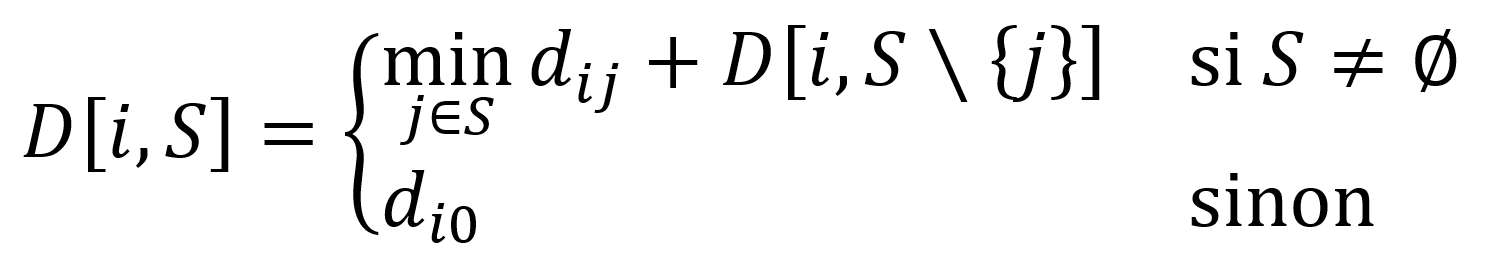
* Vos fichiers doivent être remis dans une archive zip à la racine de laquelle on retrouve :
  + Ce rapport sous format ODT ou docx.
  + Un script nommé *tp.sh* servant à exécuter les différents algorithmes du TP. L’interface du script est décrite à la fin du rapport.
  + Le code source et les exécutables.
* 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 résulats

*Exécutez chacun des trois algorithmes en notant leur temps d'exécution ainsi que la longueur minimale de votre tour à travers les N villes indiquées. Rapportez dans un tableau vos résultats ainsi que la moyenne des temps d’exécution pour chaque série d’une même taille.*

*Pour l’algorithme de programmation dynamique, ne le lancez que sur les tailles d’exemplaires de 25 et moins.*

Tableau 1: Temps d’exécution moyen et longueur minimale en fonction du nombre de ville pour l’algorithme glouton

| *Nombre de villes (unités)* | *Temps d’exécution (ns)* | *Longueur Minimale (L2 arrondie)* |
| --- | --- | --- |
| *8* | *0* | *5614.893* |
| *10* | *0* | *6439.889* |
| *12* | *0* | *6778.069* |
| *14* | *0* | *7527.588* |
| *16* | *0* | *7748.497* |
| *128* | *3125000* | *21624* |
| *256* | *6250000* | *30687* |
| *512* | *28125000* | *42050* |
| *1024* | *103125000* | *59489* |
| *2048* | *425000000* | *82427* |
| *5000* | *2525000000* | *126968* |
| *10000* | *10340625000* | *176729* |
| *50000* | *267625000000* | *397061* |
| *100000* | *1067150000000* | *558359* |

Tableau 2: Temps d’exécution moyen et longueur minimale en fonction du nombre de ville pour l’algorithme MST

| *Nombre de villes (unités)* | *Temps d’exécution (ns)* | *Longueur Minimale (L2 arrondie)* |
| --- | --- | --- |
| *8* | *0* | *5588.196* |
| *10* | *3125000* | *6524.614* |
| *12* | *0* | *7138.819* |
| *14* | *0* | *7863.969* |
| *16* | *0* | *7973.166* |
| *32* | *3125000* | *11886* |
| *64* | *6250000* | *15646* |
| *128* | *12500000* | *23464* |
| *256* | *31250000* | *33154* |
| *512* | *200000000* | *46470* |
| *1024* | *1275000000* | *64180* |
| *5000* | *60212500000* | *141595* |
| *10000* | *302778125000* | *199088* |

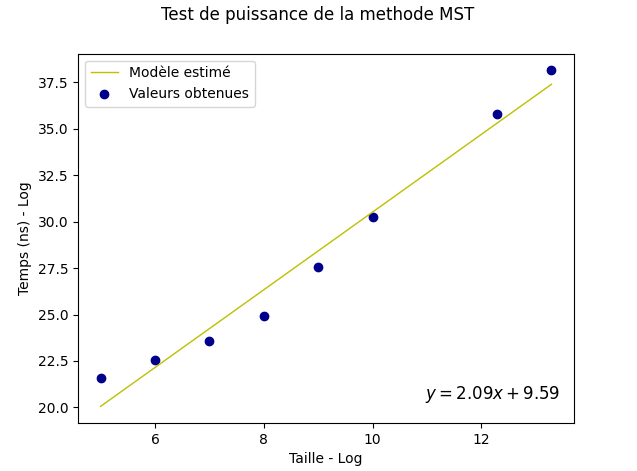
Tableau 3: Temps d’exécution moyen et longueur minimale en fonction du nombre de ville pour l’algorithme dynamique

| *Nombre de villes (unités)* | *Temps d’exécution (ns)* | *Longueur Minimale (L2 arrondie)* |
| --- | --- | --- |
| *8* | *6250000* | *5606* |
| *10* | *71875000* | *6186.201* |
| *12* | *1106250000* | *7570* |
| *14* | *21003125000* | *8414.466* |
| *16* | *399981250000* | *9209* |

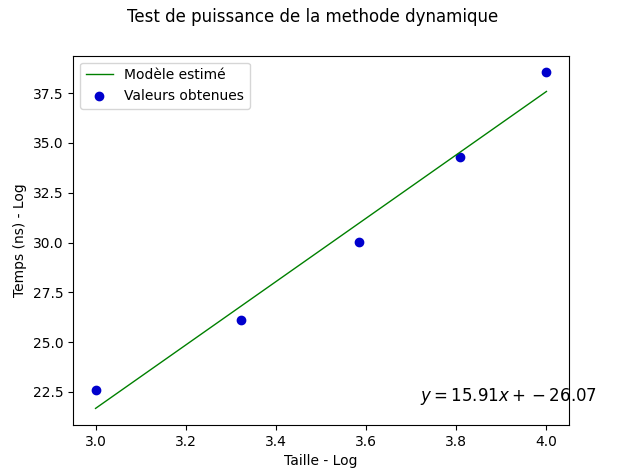
### Graphiques pour analyse hybride

#### 

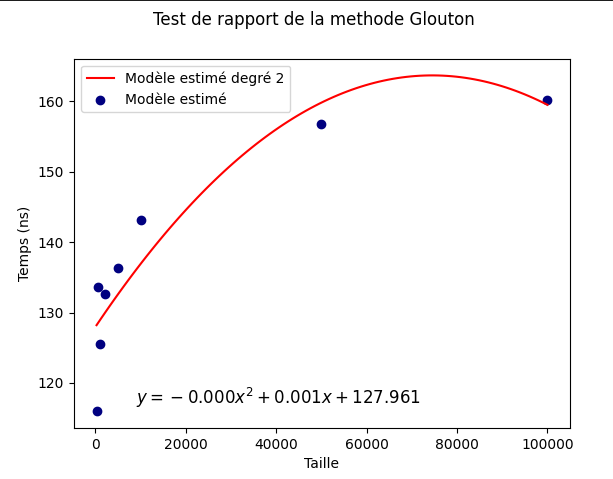
*Figure 1. Test de puissance utilisant l’algorithme glouton.*



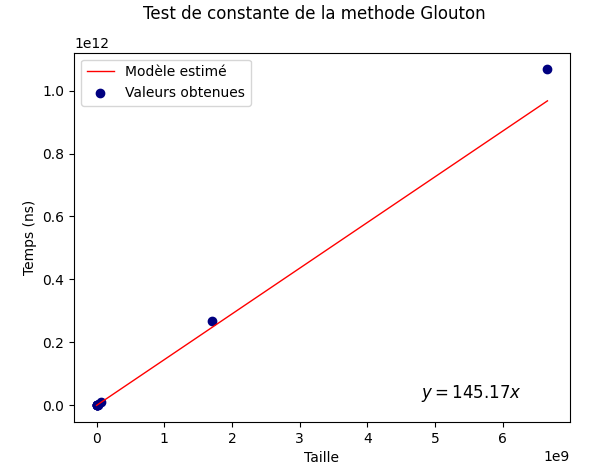
*Figure 2. Test de puissance utilisant l’algorithme MST.*



*Figure 3. Test de puissance utilisant l’algorithme dynamique.*



*Figure 4. Test de rapport utilisant l’algorithme glouton.*



*Figure 5. Test de constante utilisant l’algorithme dynamique.*

# Analyse et discussion

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

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

#### Algorithme Glouton:

En analysant le fichier `algo\_glouton.py`, pour des échantillons de taille 1 et plus, nous avons deux boucles imbriquées. Par contre, à chaque itération de la boucle englobante, nous enlevons un élément de la liste. La boucle englobante est alors appelée n fois et celle à l’intérieur est appelée n-1 fois en pire et moyen cas. En meilleur cas, nous avons une liste de 0 villes où on ne ferait que sortir de la fonction (temps constant). On obtient alors l’analyse suivante:

), ,

#### Algorithme Dynamique:

Pour l’algorithme dynamique, nous procédons en générant tous les sous-ensembles de problèmes possibles. Nous voyons dans le fichier `algo\_dynamique.py` qu’il y a des boucles imbriqués qui itèrent sur chaque sous-ensemble et chaque point.

Le nombre de sous-ensembles considérant qu’il y a n villes est de 2^n. La boucle externe a alors une complexité d’ordre 2^n. La boucle intérieure itère sur chaque ville lui donnant une complexité d’ordre n.

À l’intérieur de la boucle intérieure, la fonction `getPreviousSubset` se fait appelée. Cette fonction appèle la fonction `powerset` qui a une complexité d’ordre 2^n. La fonction `getPreviousSubset` fait appel d’abord à `powerset` et ensuite itère sur chaque sous-ensemble du sous-ensemble passé en paramètre qui a une taille maximale de n-1 puisqu’on n’inclut pas la ville de départ. Nous avons donc une complexité de O(2^(n-1)) pour la fonction `getPreviousSet`.

Sachant la complexité de la fonction `getPreviousSet`, nous pouvons dire que la fonction TSP a une complexité en moyen et pire cas d’ordre O(n\*2^n \* 2^(n-1)) qui se simplifie à O(n\*2^(2n-1)). En meilleur cas, la liste de coordonnées est vide et nous ne faisons que sortir de la fonction. Nous obtenons donc l’analyse asymptotique suivante:

), ,

**Algorithme Minimum Spanning Tree (MST):**

En analysant le fichier `algo\_mst\_py`, nous voyons que pour trouver le “minimum spanning tree”, nous voyons qu’ajouter une nouvelle valeur au tas binaire est de temps constant en meilleur cas et d’ordre log(n) en moyen et meilleur cas. Par contre, cet ajout est imbriqué dans deux boucles imbriquées donnant alors une complexité d’ordre n^2 \* log(n). La partie d’après est très similaire à l’algorithme Prim où on crée le “minimum spanning tree” des villes selon leurs coordonnées. Dans notre algorithme, sachant que nous itérons sur chaque arête de notre graphe et que sortir un élément de notre tas à priorité est d’ordre log(n) où n est le nombre de noeuds, la deuxième partie est d’ordre a\*log(n) où a représente le nombre d’arêtes du graphe. Sachant que chaque noeud visite tous les autres noeuds du graphe, il y a un total de n \* (n-1)/2 arêtes. En pire cas, il n’y a pas de coordonnées et on ne fait que sortir de la boucle. La deuxième partie est alors toujours d’ordre n^2 \* log(n). Nous obtenons alors une complexité de:

,

### 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.

#### Croissance Polynomiale Glouton

Nous voyons que dans notre graphique du test de puissance pour l’algorithme glouton qu’on peut tracer une droite linéaire dans notre graphique log-log pour estimer la croissance temporelle en fonction de la taille de nos échantillons. Cela signifie alors qu’il y a effectivement une croissance polynomiale dans ce cas. Ceci est en concordance avec notre analyse asymptotique théorique.

Pour confirmer que nous avons trouvé une bonne estimation pour le test de puissance, nous observons que le test de rapport nous retourne une courbe qui tend vers une valeur autre que 0 et infini (soit 170). Nous confirmons donc que la droite obtenue lors du test de puissance pour l’algorithme glouton est correcte.

Finalement, dans le test de constantes pour l’algorithme glouton, nous avons trouvé une pente de 145.17 ainsi qu’un travail constant de 0. En obtenant une droite linéaire dans le test de constantes, nous confirmons que l’approche glouton croit polynomialement selon la taille de l’échantillon.

#### Croissance de l’algorithme MST

En effectuant le test de puissance, nous voyons qu’on a été capable d’estimer une certaine droite linéaire. Par contre, on voit que les points forment plutôt une courbe qui tend vers le haut. Ceci implique alors que notre algorithme suit une tendance super-polynomiale. Ceci suit alors notre théorie où on avait estimé une croissance d’ordre n^2 \* log(n).

Bien que les tests de puissance, rapport et constantes ne nous permettent pas de confirmer empiriquement la présence d’une croissance log(n), nous pouvons clairement voir par le temps d’exécution des tableaux 2 et 3 que l’algorithme MST prend beaucoup plus longtemps que l’algorithme glouton dans chacune des situations. Sachant que notre complexité temporelle pour l’algorithme glouton a été estimée à un ordre de grandeur d’ordre n^2, que notre algorithme MST est estimée à un ordre n^2 \* log(n) et que n^2 \* log(n) prend plus longtemps que n^2, le fait que les temps d’exécution pour l’algorithme MST sont plus longs que ceux gloutons laisse supposer que l’algorithme MST suit bien une croissance d’ordre n^2 \* log(n).

#### Croissance de l’algorithme dynamique

Comme nous le voyons sur le graphique du test de puissance pour l’algorithme dynamique, notre estimation linéaire de la croissance temporelle accommode mal les points de notre jeu de données. Effectivement, on obtient plutôt une courbe qui va vers le haut, donnant plutôt une allure de courbe super-polynomiale. On voit alors que ceci concorde avec notre analyse asymptotique qui estimait que notre algorithme était dans l’ordre de n\*2^(2n-1). Nous voyons aussi que les temps d’exécutions sont significativement plus longs comparés à l’approche glouton et mst. Sachant que n\*2^(2n-1) est plus grand que n^2 (glouton) et n^2 \* log(n) (mst), nous voyons que nos résultats respectent l’hypothèse où notre fonction dynamique est réellement dans l’ordre de n\*2^(2n-1).

### 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.

#### Qualité respective des solutions obtenues:

Lorsqu’on compare les solutions, en se fiant aux tableaux 1 à 3, nous voyons que pour de très petites tailles d’échantillons, l’approche dynamique nous donnait de bons résultats. Par contre, plus on augmentait le nombre de villes, plus l’algorithme glouton nous donnait la solution optimale. Par contre, après cela, l’algorithme glouton nous donnait la solution optimale. Il est possible qu’il y ait faute d’implémentation puisque habituellement, la solution gloutonne ne prend pas en compte la solution optimale en considérant la solution au complet. Effectivement, elle ne fait qu’optimiser la solution à chaque étape. En théorie, l’approche dynamique devrait prendre en compte la solution optimale du système au complet puisqu’on recompose le problème entier en le sous-divisant en petits problèmes. L’approche MST garde aussi compte du système au complet en trouvant le “minimum spanning tree” qui passe par tous les nœuds du système. Il se peut alors que notre jeu de solution soit généré de sorte que, par coïncidence, l’approche gloutonne soit la meilleure. Bref, pour des petites tailles de matrices, la solution dynamique est la meilleure et pour toute autre taille de matrice, la solution gloutonne est la meilleure.

#### Ressources:

En termes de ressources, il est clair que l’algorithme dynamique en consomme beaucoup. Effectivement, non seulement faut-il se rappeler des 2^(n-1) sous-ensembles du problème, mais il faut aussi exécuter beaucoup d’opérations pour résoudre tous les sous-ensembles de problème du gros problème. Il est alors évident que l’algorithme dynamique consommait le plus de ressources en termes de temps et d’espace. L’algorithme glouton consommait le moins de ressources temporelles. En regardant les tableaux 1 et 2, nous voyons que plus on augmentait le nombre de villes, plus l’algorithme mst prenait de temps comparé à l’approche glouton. En termes d’espace mémoire, l’approche mst devait se rappeler de tous les arêtes qui existent entre chacun des noeuds, prenant alors n \*(n-1)/2 arêtes aux totale pour cette approche. L’espace mémoire alloué à l’approche mst était alors dans l’ordre de n^2. L’approche glouton n’accédait qu’à la liste originale de coordonnées faisant de sorte que sa consommation en espace mémoire était dans l’ordre de n. En terme d’efficacité de la consommation de l’espace mémoire et du temps, l’algorithme glouton était le plus efficace suivi par l’algorithme mst et finalement l’algorithme dynamique.

#### Difficulté d’implantation

L’algorithme le plus facile à implémenter était l’algorithme glouton. Effectivement, puisqu’on a qu’à se soucier de l’étape présente, l’approche était relativement simple: trouver le nœud le plus proche qui n’a pas encore été visité. L’algorithme MST suivait une approche similaire où on créait un arbre ayant le plus petit poids possible en ajoutant incrémentalement des nœuds en minimisant le poids ajouté à l’arbre existant. Toutefois, pour retourner un chemin à la fin, il fallait implémenter une traversée d’arbre en préordre, ce qui est beaucoup plus compliqué que l’approche glouton où on ne faisait que retourner une liste qui se faisait créer au fur et à mesure que notre algorithme roulait. Finalement, l’algorithme dynamique était le plus compliqué à implémenter. Effectivement, appliquer le concept diviser pour régner avec la construction de tous les sous-ensembles était une implémentation beaucoup plus compliquée. Il fallait d’abord se référer à un tableau pour voir le coût minimal des sous-problèmes composant le sous-problème présent pour alors trouver le coût du présent problème et continuer jusqu’à recomposer le problème entier. Ceci était clairement l’approche la plus difficile.

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

Selon nos résultats, il serait optimal d’utiliser l’approche dynamique pour des petites tailles de villes, soit jusqu’à environ n=8. Après, il serait préférable d’utiliser l’algorithme glouton. Bien que l’algorithme mst pourrait être utilisé, non seulement l’approche est moins efficace en termes de temps, elle ne nous donne pas une meilleure solution. Il se peut très bien qu’il y ait faute d’implémentation ou que notre jeu de test soit biaisé mais selon nos résultats, l’approche MST est simplement sous-optimale en tout cas.

# 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 |

#### Algorithme glouton:

| Fichier | Nombre de villes | Solution (L2 arrondie) | Écart avec solution optimale | % Différence |
| --- | --- | --- | --- | --- |
| hard\_N52 | 52 | 632298 | 80689 | 14.62793 |
| hard\_N91 | 91 | 1338013 | 109287 | 8.894334 |
| hard\_N130 | 130 | 2125138 | 196404 | 10.18305 |
| hard\_N169 | 169 | 2801683 | 25416285 | 7.734414 |
| hard\_N199 | 199 | 3342750 | 202972 | 6.464533 |

#### Algorithme mst:

| Fichier | Nombre de villes | Solution (L2 arrondie) | Écart avec solution optimale | Différence (%) |
| --- | --- | --- | --- | --- |
| hard\_N52 | 52 | 873360 | 321751 | 58.32954 |
| hard\_N91 | 91 | 1879374 | 650648 | 52.95306 |
| hard\_N130 | 130 | 2628664 | 699930 | 36.28961 |
| hard\_N169 | 169 | 3106247 | 505701 | 19.44595 |
| hard\_N199 | 199 | 4216667 | 1076889 | 34.29825 |

#### Nous voyons que l’algorithme glouton performe relativement bien comparé à l’approche mst. Effectivement, l’algorithme glouton a un écart d’environ 10% avec la solution optimale. Au contraire, l’approche MST performe mal avec une différence en moyenne d’environ 40%. Ceci est étonnant considérant que l’approche glouton ne tend qu’à optimiser l’étape courante sans se soucier de la solution optimale globale. Toutefois, il se peut qu’une faute d’implémentation dans notre approche mst fait en sorte qu’on obtienne des résultats moins précis que ceux de l’approche gloutonne.

# 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)