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

Rapport TP2 – Automne 2022

|  |  |
| --- | --- |
| **Nom, prénom, matricule des membres** | Ouali, Ouassim, 1958275  Brunet, Philippe, 1875508 |
| **Note finale / 30** | 0 |

# Informations techniques

* Répondez directement dans ce document DOCX. 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 15 Novembre à 23h59 en suivant les instructions suivantes :
  + Vos fichiers doivent être remis dans une archive zip à la racine de laquelle on retrouve :
    - Ce rapport au format 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.
    - Si le langage que vous utilisez nécessite une phase de compilation, veuillez joindre un Makefile afin que nous puissions le compiler en cas de problème avec vos exécutables. Si nous ne sommes pas en mesure de tester votre code, vous perdrez des points de respect d’interface et de qualité de code !
* 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 afin de résoudre une version simplifiée d'un problème réaliste d'optimisation.

La chaine de restauration rapide W&A songe à élargir sa zone de couverture dans plusieurs villes canadiennes suite à l'engouement général pour ses burgers au “poulet de grain élevé sans antibiotiques et nourri de manière végétarienne”. Une étude de marché lui a permis d'identifier pour chaque ville les sites potentiels où il est possible d'installer de nouveaux restaurants, ainsi que les revenus prévisionnels et la consommation de poulet attendue chaque jour, pour chaque site. Tous les restaurants d'une ville seront reliés à un seul fournisseur, qui livrera la totalité du poulet utilisé dans ses burgers. Afin de pouvoir fournir tous les restaurants de la ville, ce fournisseur ne peut pas livrer une quantité de poulet qui dépasse celle qu'il obtient des producteurs à chaque jour (sa capacité). Tous les fournisseurs sont déjà actifs et chacun d'eux a une quantité de poulet livrable fixe et limitée. Votre rôle est de trouver une solution pour chaque ville qui permet de maximiser la somme des revenus prévus tout en respectant la quantité de poulet livrable par chaque fournisseur. Comment faire ?

# Implantation

Trois algorithmes seront implantés, mettant en pratique les patrons de conception : glouton, programmation dynamique, heuristique d'amélioration locale et probabiliste.

## Algorithme glouton probabiliste

L'algorithme glouton fait son choix d'emplacement en fonction de la rentabilité. Pour chaque emplacement *i* vous calculez sa rentabilité *Ri* comme étant le revenu divisé par la quantité de poulet :

*Ri = ri /qi.*

Nous ajoutons un ingrédient probabiliste en randomisant l'algorithme : à chaque itération, plutôt que d'ajouter l'emplacement qui a la rentabilité maximum, on choisit au hasard un emplacement proportionnellement à sa rentabilité.

La probabilité de choisir l'emplacement *i* est *pi = Ri / ∑j=1..n Rj .*

Un exemple d'implémentation est disponible à la page suivante : <http://en.wikipedia.org/wiki/Fitness_proportionate_selection>

L'algorithme est lancé à dix reprises et la meilleure solution est retournée.

## Algorithme de programmation dynamique

L'algorithme de programmation dynamique remplit un tableau de revenus optimaux en fonction des sites permis et de la quantité livrable du fournisseur : *D[i,j] = max(ri + D[i-1, j-qi], D[i-1, j])* représente le meilleur revenu possible pour un fournisseur ayant la quantité *j* en utilisant des sites parmi les *i* premiers.

Cet algorithme peut être assez gourmand en espace mémoire.

## Heuristique d’amélioration locale

L'algorithme de type amélioration locale démarre sa recherche avec une première solution connue qui est celle obtenue par l'algorithme glouton. Ensuite, pour améliorer cette solution, il est possible de réaliser un certain nombre d'améliorations locales.

L'heuristique d'amélioration locale entre dans la grande famille des algorithmes de recherche locale. Ces algorithmes utilisent deux concepts : le voisinage et la fonction objectif. Le voisinage est la différence entre deux solutions, dites voisines. Pour cet algorithme, il s'agira d'enlever un ou deux sites et d'en ajouter un ou deux autres. Évidemment, vos nouvelles solutions doivent demeurer valides. La fonction objectif est le revenu généré par l'ensemble des sites choisis. Par conséquent, à chacune des itérations de l'algorithme, vous devez choisir, parmi tous les voisins, celui qui améliore le plus la fonction objectif. Vous devez évidemment tenir compte de la validité de la nouvelle solution. Cela signifie que si un échange implique un excès de la capacité du fournisseur, il ne peut pas être choisi.

Le critère d'arrêt sera l'optimum local : lorsqu'aucune solution voisine améliorant la fonction objectif est trouvée, l'algorithme s'arrête.

# Jeu de données

Vous trouverez sur le site Moodle du cours tous les exemplaires (villes) du problème à résoudre.

La structure des fichiers d'exemplaires est :

Sur la première ligne : *Nombre d'emplacements*

Une ligne pour chaque emplacement : *i* <espace> *ri*<espace> *qi*

Sur la dernière ligne : *Capacité du fournisseur (quantité livrable)*

# 1. Présentation des résultats

|  |  |
| --- | --- |
|  | / 1,5pts |

## Tableau des résultats

**Temps et revenu moyen pour des combinaisons de nombres de restaurants et capacité**

Tableau 1 : Algorithme glouton probabiliste



Tableau 2 : Algorithme de programmation dynamique



Tableau 3: Algorithme heuristique d’amélioration locale

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Local | Nombre de restaurants | | | | | |
| 100 | | 1000 | | 10000 | |
| Séries (capacité) | Temps (ms) | Revenu | Temps (ms) | Revenu | Temps (ms) | Revenu |
| 10 | 665.21 | 238 | - | - | - | - |
| 100 | 4969.7033 | 4218 | - | - | - | - |
| 1000 | 2012.19 | 4112 | - | - | - | - |

Pour l’algorithme d’amélioration locale avec heuristique, nous n’avons pas pu générer de résultats pour un nombre de restaurants de 1000 ou plus.

# 2. Analyse et discussion

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

|  |  |
| --- | --- |
|  | / 7,5 pt |

Pour toutes les analyses qui suivent, notons N le nombre de restaurants total et Q la capacité totale.

### 2.1 Algorithme Glouton

*Principe : Tant qu’il reste des restaurants à considérer, on choisit celui qui est le plus rentable.*

À chaque itération, on retire d’abord les restaurants avec une quantité plus élevée que la capacité restante. Pour se faire il faut parcourir les *k* restaurants restants. Ensuite, il faut calculer la rentabilité totale, soit un autre parcourt de k itérations. Comme il s’agit d’un algorithme glouton probabiliste, on ne choisit pas le restaurant le plus rentable, mais plutôt selon une certaine probabilité. Sans entrer dans le détail, cela veut donc dire que l’on parcourra un sous-ensemble des k restaurants, jusqu’à trouver celui correspondant à certain nombre aléatoire. On donc trois étapes avec une consommation d’au plus O(k), ce qui veut donc dire une consommation O(k) pour chaque itération.

Comme les revenues et quantités ne sont pas connues d’avance, et qu’en plus le restaurant choisi est proportionnel à un nombre aléatoire, il n’y a pas moyen de savoir combien de restaurants seront retirées à chaque itérations. En particulier, on ne peut connaitre la séquence des valeurs de k. En pire cas, il est possible de choisir tous les restaurant pour la solution finale, ce qui représenterait n itérations avec k allant de (n-1) jusqu’à 1. Le nombre d’opérations total s’exprimerait donc comme :

Pour une complexité globale de O(N2). En pratique, il pourrait toutefois y avoir bien moins que N itérations de restaurants et le nombre de restaurants à explorer (*k*)pour une itération donnée *i* sera surement inférieur à à (N-i). On s’attend donc à une meilleure performance que O(N2).

### 2.2 Algorithme Dynamique

*Principe : remplir un tableau dynamiquement et obtenir le revenu de la solution optimale dans la dernière case du tableau. On retrace ensuite le contenu de cette solution à partir du tableau.*

* Remplir le tableau : se fait en N\*Q
  + On a donc ici une consommation pseudo-polynomiale (pseudo-linéaire, pour être plus précis). Pour des valeurs de Q bien plus petites que N, la performance est donc linéaire en N, mais si N et Q sont comparables, on a une complexité en N2.
* Trouver la bonne valeur dans le tableau : au pire il faut remonter d’autant de ligne que de restaurant, ce qui fait N itérations.
* L’étape coûteuse est donc celle de la construction du tableau dynamique. La consommation est de O(N\*Q) ou simplement O(N) si Q est suffisamment petit.

Une analyse plus détaillée est disponible en annexe A.

### 2.3 Algorithme d’amélioration locale avec heuristique

*Principe : à chaque itération on construit un voisinage en retirant entre 1 et 2 restaurants de la solution courante et en ajoutant entre 1 et 2 restaurants hors de la solution. La boucle se poursuit tant que le voisinage contient au moins une solution meilleure que la solution courante.*

Nous partons d’une solution calculée par l’algorithme glouton probabiliste. Cet algorithme a été analysé en 2.1. Sa complexité maximale est O(N2), ce qui est dominé par les autres opérations de l’algorithme d’amélioration locale, comme nous le présentons ci-bas.

Construction du voisinage :

* + Supposons qu’il y a N restaurants au total et k restaurant dans la solution courante.
  + Il y a k possibilités pour enlever 1 restaurant et possibilités pour en enlever 2.
  + Peu importe la quantité de restaurants qu’on enlève, il y a (N-k) possibilités pour en ajouter 1 autre et possibilités pour en ajouter 2.
  + Au total, cela fait donc un voisinage d’une taille de :

Dans le meilleur cas, il y a un seul restaurant dans la solution (k=1) et on a voisins. La situation est analogue s’il y a k = N-1 restaurants dans la solution => voisins.

Dans le pire cas, il y a autant de restaurants dans la solution courante qu’à l’extérieur (k = N/2) :

Comme nous partons d’une solution glouton probabiliste, il n’y a pas moyen de savoir exactement le nombre de restaurants initiaux dans la solution courante de départ, ni de savoir son avancement. Il n’est pas non plus possible de savoir après combien d’itérations on ne trouveras plus de voisinage.

Dans le meilleur cas, après une itération il n’y aura plus de meilleur voisinage. Dans le pire cas, à chaque itération on trouve un meilleur voisinage, jusqu’à ce que chaque combinaison de restaurants ait été considéré dans un voisinage au moins une fois, ce qui donne un nombre d’itérations exprimés par :

Afin de donner une borne inférieure sur le nombre d’itérations que cela représente, supposons qu’on considère N4 restaurants à chaque itération. Cela ferait donc itérations. À titre d’exemple, pour N=1000, cela représente un nombre d’itération de l’ordre de 10289.

# 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.

|  |  |
| --- | --- |
|  | / 7,5 pt |

## 3.1 Algorithme glouton probabiliste :

Nous avons une borne supérieure pour la consommation de cet algorithme, mais en raison de l’aspect probabiliste qui le compose et d’autres facteurs discutés dans la section 2.1. Par conséquent, le test de puissance est approprié, ce qui nous permettra de déterminer si la consommation est linéaire.

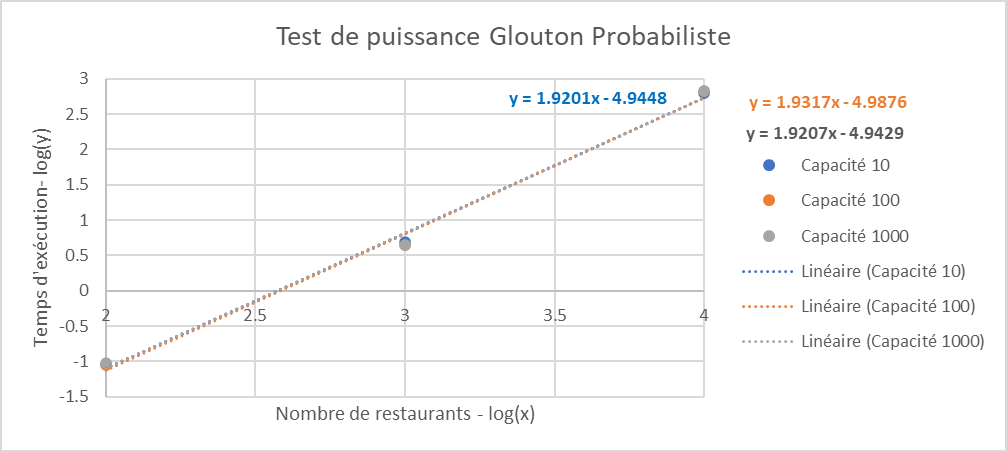


Figure : Test de puissance de l'algorithme glouton probabiliste

En observant la figure 1, contenant le résultat du test de puissance effectuer sur chaque capacité, il est possible de remarquer que les régressions linéaires de chacune des capacités sont très similaires. Et donc qu’il est possible de confirmer que la complexité de l’algorithme glouton probabiliste est de l’ordre de n1.9242.

## 3.2 Algorithme de programmation dynamique :

Pour cet algorithme, nous sommes confiants que la consommation est en O(N\*Q). Un test de constantes est donc spécialement approprié pour préciser cette hypothèse, en plus de faciliter l’analyse de l’aspect multidimensionnelle des paramètres.

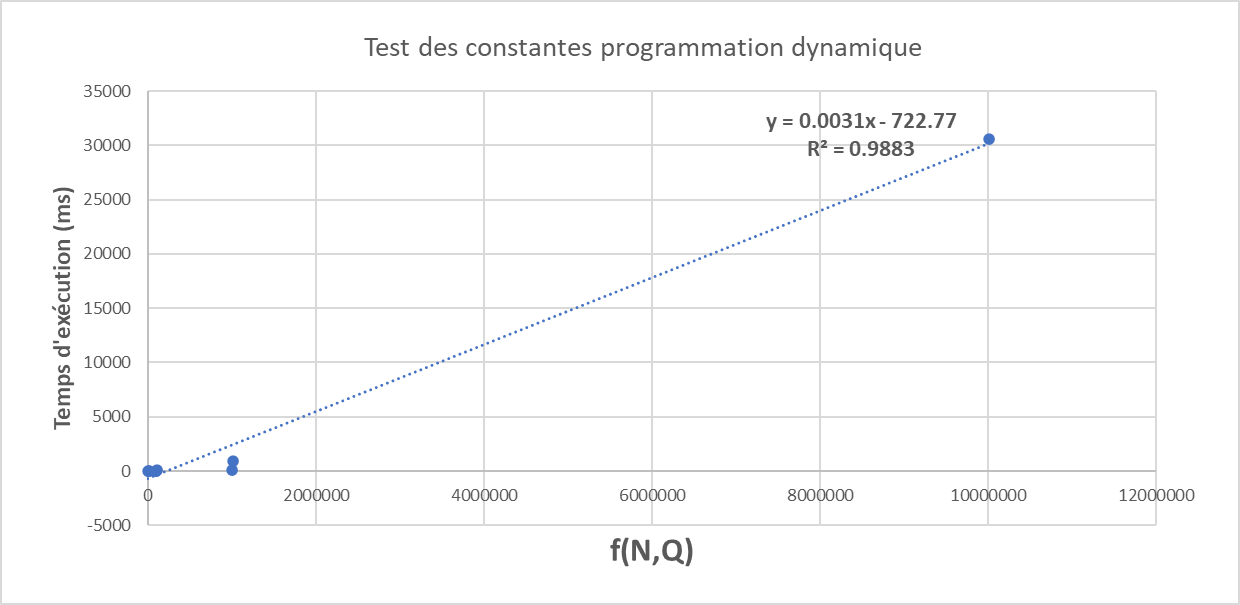


Figure : Test des constantes de l'algorithme de programmation dynamique

À la suite du test des constantes, nous observons que la tendance de la courbe est linéaire et donc que nous pouvons exprimer la consommation de l’algorithme avec où c et b sont des constantes et f(x) est f(N,Q) = N x Q. La consommation peut donc être écrite comme Et nous pouvons conclure que . Ce qui confirme l’analyse asymptotique au point 2.2 précédent.

## 3.3 Algorithme heuristique d’amélioration locale :

De la même façon que pour l’algorithme glouton probabiliste, nous n’avons pas une idée claire sur la consommation de cet algorithme. D’après l’analyse faite en 2.3, nous suspectons toute de même qu’elle soit polynomiale. Il serait donc pertinent d’effectuer un test de puissance. Toutefois, comme nous n’avons pas de résultats pour les séries de données de 1000 et 10000, il nous est impossible de réaliser ce test. La capacité totale a effectivement un impact sur la performance de l’algorithme ici, car plus cette quantité est grande, plus il y a un grand bassin pour des restaurants qui pourraient améliorer la solution et donc impliquer le déroulement d’itérations supplémentaires.

# 4. 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.

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

|  |  |
| --- | --- |
|  | / 6,5 pt |

## 4.1 Algorithme Glouton Probabiliste

Le plus grand avantage de l’algorithme glouton probabiliste est sa simplicité d’implémentation et sa performance. Les solutions obtenues via cette méthode ne sont pas optimales. Puisque cet algorithme utilise un élément probabiliste, il est nécessaire de faire des implémentations nécessaires afin de pouvoir faire le calcul des probabilités. Cette méthode utilise donc un peu plus de ressource de calcul qu’un algorithme glouton non probabiliste. L’espace mémoire utilisé est faible car l’algorithme n’a pas besoin de garder en mémoire les restaurants rejeter par le choix glouton.

## 4.2 Algorithme Dynamique

L’ordre du temps d’exécution de l’algorithme de programmation dynamique est O(*N*\**Q*). Pour ce qui est de l’espace mémoire, l’espace de mémoire à allouer est de l’ordre de n\*m et donc cette méthode utilise une grande quantité de mémoire RAM. L’avantage de la méthode de programmation dynamique est qu’elle donne toujours la solution optimale et cela relativement rapidement. Le problème est que cette méthode utilise beaucoup d’espace mémoire puisse qu’elle doit allouer un tableau de *N* x *Q* éléments. Ce qui, dans le cas de l’exemplaire WC-10000-1000-10.txt, rend impossible l’exécution de l’exemplaire sur les ordinateurs du laboratoire. Pour pouvoir obtenir le résultat, il a fallu utiliser un ordinateur ayant 32 Gb de mémoire vive (RAM) pour obtenir un résultat. Donc, tant que l’exemplaire est d’une taille n’excédant pas la quantité de mémoire pouvant être allouer, cet algorithme est idéal puisque le temps d’exécution est rapide et la solution est toujours optimale. La difficulté d’implémentation est un peu plus élevée que la méthode gloutonne puisqu’il faut tout d’abord définir la récurrence. Après avoir défini la récurrence il faut créer un tableau en utilisant cette récurrence et finalement il faut effectuer une recherche dans le tableau pour trouver la solution optimale. La condition sous laquelle il serait idéal d’utiliser cet algorithme est lorsque le système demande un résultat optimal à tout coup et rapidement le tout sans avoir de contrainte d’espace mémoire (RAM). Un autre cas serait ou la taille des exemplaires n’excèdent pas la mémoire pouvant être allouer.

## 4.3 Algorithme d’amélioration locale avec heuristique

Il s’agit d’un algorithme assez complexe à implémenter, notamment puisqu’il y a plusieurs décisions à prendre en ce qui concerne la façon de générer les voisins (d’abord générer, puis explorer, ou explorer pendant qu’on génère), la façon d’analyser un voisin (calculer le différentiel de la fonction objectif pour chaque voisin, ou recalculer la fonction objective à chaque fois), structures de données à utiliser, etc. Cet algorithme ne garantit pas l’optimalité et peut être très inefficace dépendamment de la solution initiale. Il est aussi problématique qu’il n’y ait aucune garantit sur l’efficacité.

## 4.4 Conclusion

## Si on a besoin d’un algorithme simple et efficace, l’algorithme glouton est le choix à faire.

Si on a besoin d’une solution optimale, l’algorithme dynamique est le seul qui peut le garantir. Il est de plus relativement efficace. Toutefois, si le nombre de restaurants est grand et que la performance est un enjeu important, car il doit être effectué à répétition dans un court laps de temps, alors la complexité de cet algorithme pourrait devenir un problème. Dans la situation actuelle, comme on cherche une seule solution, nous n’aurons pas à calculer la solution plusieurs fois de suite et la performance obtenu est vraisemblablement suffisante. Notons aussi que la quantité totale de poulets peut être un enjeu si elle est du même ordre que N, voire d’un plus grand ordre (Np avec p un entier supérieur à 1). Dans ce cas l’algorithme peut devenir significativement moins performant. Une solution à ce problème serait toutefois de changer l’unité utilisé pour la quantité de façon à avoir moins d’entrée dans le tableau dynamique.

En ce qui concerne l’algorithme d’amélioration locale, celui que nous avons implémenté ne présente pas vraiment d’avantage sur l’algorithme dynamique, car il est moins performant, plus complexe à implémenter et ne garantit pas l’optimalité. Notons néanmoins qu’il aurait été possible de concevoir un algorithme plus simple avec un voisinage moins élaborée. Dans ce cas, l’algorithme d’amélioration locale pourrait être une légère couche par-dessus l’algorithme glouton et ainsi produire des solutions légèrement supérieures à ce dernier, sans trop dégrader la performance.

# Autres critères de correction

## Respect de l’interface tp.sh

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

## Utilisation :

tp.sh –a [glouton|progdyn|local] -e path\_vers\_exemplaire

Arguments optionnels :

-p affiche la solution (emplacements i choisis) sur une ligne, avec chaque emplacement séparé par un espace

Exemple :

1 4 12 20

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

Important : l’option -e doit accepter des fichiers avec des paths absolus.

## Qualité du code

|  |  |
| --- | --- |
|  | / 5 pt |

* + - * 1. Validité des solutions
        2. Qualité de l'implémentation

Présence de commentaires

## Présentation générale

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

* Concision
* Qualité du français

## Pénalité retard

|  |
| --- |
| 0 |

* -15% de la note / journée de retard, arrondi vers le haut. Les TPs ne sont plus acceptés après 3 jours.

# Annexe A – Code annoté pour analyse de l’algorithme dynamique

Text

Description automatically generated

Temps : ɵ(N x *Q* + *Q*).

Temps : ɵ(*N*).

Temps : ɵ(1).

Temps : ɵ(1).

Temps : ɵ(1).

Temps : ɵ(1).

Temps : ɵ(*Q*).

Temps : ɵ(1).

Temps : ɵ(1).

Temps : ɵ(*Q*).

Figure 3: Code du module Initialize Table

A screenshot of a computer

Description automatically generated with medium confidence

Temps : ɵ(N x Q + Q + N).

Temps : ɵ(1).

Temps : ɵ(N).

Temps : ɵ(1).

Temps : ɵ(1).

Temps : ɵ(N x Q+ Q).

Figure 4: Code de l'algorithme dynamique