INF4705 – Analyse et conception d’algorithmes

TP2 – Hiver 2018

|  |  |
| --- | --- |
| **Nom, prénom, matricule des membres** | Tremblay, David, 1748125  Souli, Taleb, 1770491 |
| **Note finale / 13** | 12 |

# Informations techniques

* Répondez directement dans ce document ODT avec LibreOffice. Veuillez ne pas inclure le texte en italique servant de directive.
* La correction se fait à même le rapport.
* Avant votre cinquième séance de laboratoire, vous devez faire une remise électronique en suivant les instructions suivantes:
  + Le dossier remis doit se nommer matricule1\_matricule2\_tp2 et doit être compressé sous format zip.
  + À la racine de ce dernier, on doit retrouver :
    - Ce rapport sous format ODT.
    - Un script nommé *tp.sh* servant à exécuter les différents algorithmes du TP. L’interface du script est décrite à la fin du rapport.
    - Un fichier texte nommé *emails.txt* contenant le courriel de chaque membre de l’équipe
    - 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 compatible 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.

Vous participez à une compétition dont le principal objectif est de construire la tour la plus haute avec les blocs de béton étant à votre disposition. Vous devez les empiler afin que votre tour soit la plus haute possible. Vous disposez d'un ensemble de blocs de dimensions variées, qui sont définies comme suit : hauteur , largeur et profondeur . Afin de garantir la stabilité de votre tour, vous devez vous assurer que le bloc que vous ajoutez sur votre tour repose entièrement sur le précédent.

Plus formellement, vous devez respecter la règle suivante:

qui garantit une inclusion stricte de la base du nouveau bloc dans la surface du bloc receveur.

Puisqu'il vous est permis de tourner vos blocs pour obtenir la configuration la plus avantageuse,

un bloc de dimensions données pourrait être utilisé au plus trois fois, dans chacune de ses orientations.

Donc pour inclure l'orientation des blocs, il vous suffit de faire comme si les trois orientations possibles de votre bloc étaient en fait des blocs différents. Par exemple pour un bloc de taille , vous pourriez créer l'équivalent de ces trois blocs :

Sans perte de généralité, vous pouvez générer vos orientations en considérant uniquement des blocs dont la largeur est toujours inférieure ou égale à la profondeur ().

Votre objectif est donc de maximiser la hauteur de votre tour. Une solution au problème est un sous-ensemble de blocs ordonné, de la base au sommet, tel que chaque bloc peut supporter le suivant.

# Implantation

Trois algorithmes seront implantés, mettant en pratique les patrons de conception vorace, probabiliste, programmation dynamique et recherche avec tabou.

## Algorithme vorace probabiliste

Vous devez concevoir un algorithme vorace de votre cru pour résoudre ce problème. Plutôt qu'être déterministe, il sera probabiliste et devra inclure un aspect aléatoire dans son critère de choix vorace.

## Algorithme de programmation dynamique

Vous devez également résoudre ce problème à l'aide de la programmation dynamique. Tout d'abord, pour que l'approche fonctionne, vous devez trier les blocs disponibles en ordre décroissant de surface. Cela revient à calculer l'aire pour chacun des blocs ().

Soit la hauteur de votre tour avec le bloc au sommet; on la définit récursivement par

Pour un exemplaire à blocs la solution se trouve à .

## Recherche avec tabou

Enfin vous devez implanter l'approche de recherche avec tabou avec le voisinage suivant :

Choisir un bloc parmi ceux ne faisant pas partie de la tour ni de la liste taboue puis le placer sur le bloc le plus haut dans la tour courante qui puisse le recevoir. Ce mouvement permet en particulier de modifier la base de la tour. Cette insertion peut engendrer le retrait de certains blocs directement au-dessus pour conserver l'équilibre de la tour: chacun de ces blocs sera alors tabou pour un nombre d'itérations choisi uniformément au hasard dans l'intervalle . Le voisin choisi est celui qui maximise la hauteur de la tour.

Démarrez avec une tour vide et arrêtez après 100 itérations sans amélioration de la meilleure solution trouvée.

# Jeu de données

Pour tester les algorithmes, vous devez générer un jeu de données avec 10 exemplaires

pour les tailles: 1000, 5000, 10000, 50000, 100000 et 500000. Vous pouvez utiliser le script suivant:

for n in {100,500,1000,5000,10000,50000,100000}; do

for i in {1..10}; do

# Génération d'une permutation de 1 à 3n

shuf -i 1-$((3\*$n)) |

# Regroupement par bloc de 3

awk 'BEGIN {i=0} {printf $1; if (++i%3==0) printf "\n"; else printf " "}' > b${n}\_${i}.txt

done

done

Chaque ligne représente les trois dimensions d’un bloc.

# Présentation des résultats

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

### Tableau des résultats

*NOTE:* Il semble y avoir un problème avec notre algorithme tabou. Celui-ci prend plus de 4 heures pour générer les solutions de tous les exemplaires et il retourne parfois des solutions meilleures que l’algorithme dynamique (anormal en effet). Le reste de ce travail pratique a été réalisé en tenant compte des résultats théoriques que nous devrions obtenir avec l’algorithme de recherche tabou.

**Tableau des résultats - Vorace probabiliste**

|  |  |  |
| --- | --- | --- |
| ***Taille*** | ***Temps (s)*** | ***Hauteur*** |
| *100* | *0,0000512* | *4625* |
| *500* | *0,0002235* | *50352* |
| *1000* | *0,0004327* | *150951* |
| *5000* | *0,0016044* | *1777390* |
| *10000* | *0,0028555* | *5139520* |
| *50000* | *0,0119584* | *18256434* |
| *100000* | *0,0247762* | *25594588* |

**Tableau des résultats - Programmation dynamique**

|  |  |  |
| --- | --- | --- |
| ***Taille*** | ***Temps (s)*** | ***Hauteur*** |
| *100* | *0,0003803* | *6981* |
| *500* | *0,0056600* | *85398* |
| *1000* | *0,0152878* | *247843* |
| *5000* | *0,2714572* | *31112037* |
| *10000* | *1,0497876* | *8422219* |
| *50000* | *34,7314978* | *31112037* |
| *100000* | *146,1505391* | *48418499* |

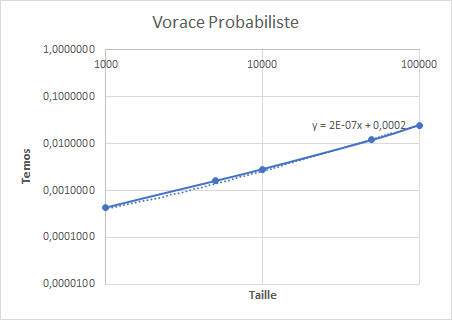
**Tableau des résultats - Recherche avec tabou**

|  |  |  |
| --- | --- | --- |
| ***Taille*** | ***Temps (s)*** | ***Hauteur*** |
| *100* | *0.1420518* | *5178* |
| *500* | *0.8742011* | *48308* |
| *1000* | *1.9239483* | *128706* |
| *5000* | *14.6449014* | *1352789* |
| *10000* | *34.3842085* | *3411121* |
| *50000* | *312.8222711* | *33564692* |
| *100000* | *969.1077856* | *99101692* |

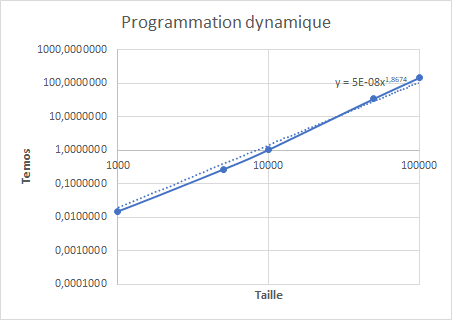
### Graphs pour analyse hybride

***Tests de puissance***

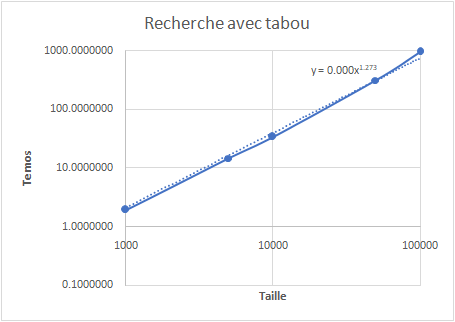
**Test de puissance - Vorace probabiliste**



**Test de puissance - Programmation dynamique**

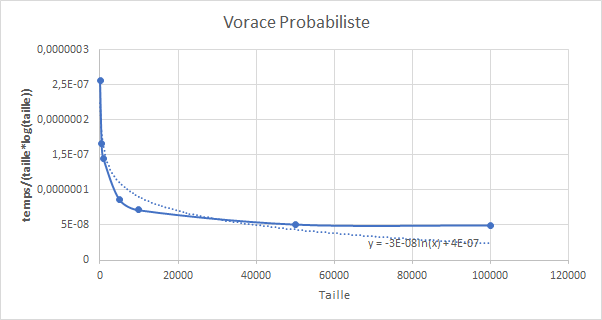


**Test de puissance - Recherche avec tabou**



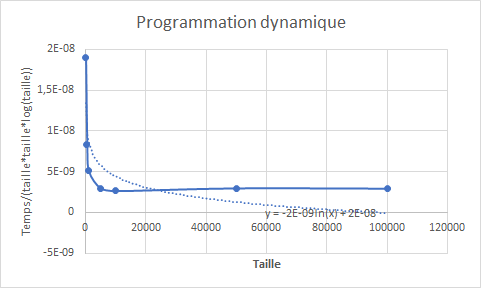
***Tests du rapport***

**Test du rapport - Vorace probabiliste**



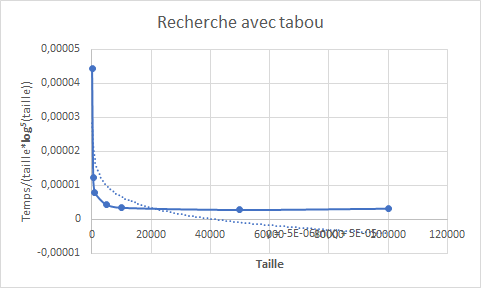
**Test du rapport - Programmation dynamique**

*(avec hypothèse O(n2log(n)) et non O(n2))*



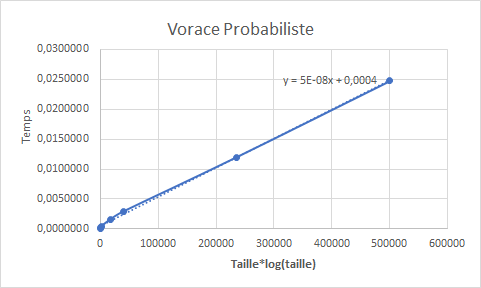
**Test du rapport - Recherche avec tabou**

*(avec hypothèse O(nlog5(n)) et non O(nlog3(n)))*

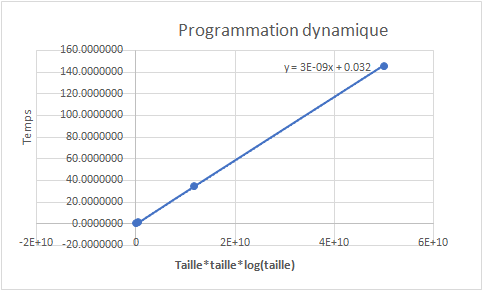


***Tests des constantes***

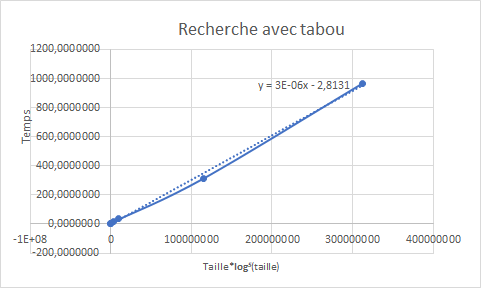
**Test des constantes - Vorace probabiliste**



**Test des constantes - Programmation dynamique**



**Test des constantes - Recherche avec tabou**



# Analyse et discussion

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

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

***Algorithme vorace probabiliste***

Notre algorithme vorace contient tout d’abord la méthode std::sort avec une complexité asymptotique **O(nlog(n))**. Nous avons par la suite une boucle **for** sur l’ensemble des blocs et donc de complexité **O(n)**. L’intérieur de la bloc est en **O(1)**. Nous avons finalement une autre boucle **for** avec une complexité asymptotique identique à la précédente. Au final, notre algorithme est donc de complexité **max(O(nlog(n), O(n), O(1))** et donc **O(nlog(n))**.

***Algorithme de programmation dynamique***

Cet algorithme contient tout d’abord la méthode std::sort de complexité **O(nlog(n))**. Nous avons ensuite une boucle  **for** sur l’ensemble des blocs et donc **O(n)** avec une complexité de **O(1)** à l’intérieur. Nous avons par la suite 2 boucles  **for** imbriquées. La première est encore pour tous les blocs et donc **O(n)** et la seconde est aussi en **O(n)** et l’intérieur des boucles est en **O(1)**. Nous obtenons donc une complexité de **O(*n2*)** pour ces boucles. Une autre boucle for sur l’ensemble des blocs de complexité **O(n)** suit ces boucles et pour finir nous avons une dernière boucle for sur une partie des blocs disponibles (disons m, *où* m ≤ n) et donc de complexité **O(m)**. Notre algorithme aura donc une complexité de **max(O(nlog(n), O(n), O(*n2*), O(1), O(m))**  et donc **O(n2)**.

***Algorithme de recherche avec tabou***

L’algorithme commence tout d’abord avec une boucle **for** sur 100 itérations mais qui peut être réinitialiser. Si cette réinitialisation est réalisée k fois, alors la complexité de la boucle est de **O(k)**. Nous avons ensuite une boucle for imbriquée dans la précédente sur l’ensemble des blocs (**O(n)**) qui contient la méthode std::find en **O(m)** *où* m ≤ n (la méthode est appliquée sur une partie des blocs). Ensuite, nous avons la méthode swipeData qui regarde tous les blocs au-dessus d’un bloc inséré pour analyser si des blocs doivent être ajouter dans la liste tabou. Cet algorithme contient un **while**  sur une partie des blocs (disons m, *où* m ≤ n) et donc de complexité **O(m)** et l’intérieur de la boucle while est en **O(1)**. Il y a ensuite un appel à la méthode getHeight() qui retourne la hauteur d’une tour en **O(m)** (m ≤ n puisque la tour est constituée de tous les blocs ou moins). Finalement, il y a un appel à la méthode decrementIteration() qui décrémente les itérations de la liste tabou à chaque itération qui est en **O(m)** (puisque le tabou contient tous les blocs ou moins). Les deux boucles imbriquées ont donc une complexité de **O(k\**n\*m2 + n\*m + k\*m*)** aveck ≤ n et m ≤ n. Si on considère k et m comme log(n), on obtient une complexité asymptotique finale de **O(log(n)\*n\*log*2*(n)) => O(n\*log*3*(n))**. Je ne suis pas vraiment d’accord avec votre cheminement. Pour une itération, il faut essayer chaque bloc (pire cas: n blocs) et pour chaque bloc il faut essayer de l’insérer en pire cas à toutes les positions de la tour (pire cas: tour de hauteur n). Donc nous avons déjà n^2 pour une seule itération. -0.5

**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.**

La meilleure méthode pour confirmer ou infirmer nos analyses asymptotiques théorique est de réaliser des tests de rapports. Nous avons tout de même réaliser les tests de puissance, rapport et constantes pour pouvoir approfondir l'analyse de nos algorithme.

***Algorithme vorace probabiliste***

Nous avons émis l’hypothèse que cette algorithme avait une complexité O(nlog(n)). Notre analyse semble être confirmé par le test du rapport puisque notre graphe tend vers une constante qui n’est pas zéro (environ 5E-08).

***Algorithme de programmation dynamique***

Nous avons émis l’hypothèse que cette algorithme avait une complexité O(n2). Toutefois, notre analyse semble erronée. En effet, avec une hypothèse de complexité O(n2), la courbe obtenue ne ressemble nullement au résultat désiré (sous-estimation). On observe le même résultat avec une hypothèse de complexité O(nlog(n)). Par contre, on obtient une courbe qui tend vers une constante positive qui n’est pas zéro avec une hypothèse de complexité O(n2log(n)). Notre analyse sous-estimait donc la complexité asymptotique réelle de l’algorithme.

***Algorithme de recherche avec tabou***

Nous avons émis l’hypothèse que l’algorithme de recherche avec tabou avait une complexité O(n\*log*3*(n)). Par contre, un test de rapport a permi de constater que cette hypothèse sous-évaluait la complexité réelle puisque nous obtenons une droite qui croît. Une hypothèse O(n\*log*5*(n)) a aussi été testée et une courbe qui tend vers une constante positive qui n’est pas zéro a été obtenue. Au final, il semble que notre hypothèse initiale sous-évaluait la complexité réelle de l’algorithme de recherche tabou. Votre analyse ici n’est pas convaincante car vous faites des tests à l’aveuglette pour voir si quelque chose semble fonctionner. Ce n’est pas à dire que ça fonctionnerait sur un n plus grand. -0.5

### 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.

***Algorithme vorace probabiliste***

L’algorithme vorace probabiliste est de loin l’algorithme le plus simple à implémenter. Il suffit de choisir la meilleure solution locale et d’y ajouter une certaine décision aléatoire pour décider si cette solution sera retenue ou non. Le temps de calcul est aussi de loin le plus rapide des trois et l’algorithme a la meilleure complexité asymptotique. Par contre, cette algorithme est aussi celui qui donne en moyenne la plus mauvaise des solutions au problème puisque la simplicité de l’algorithme est au détriment de certains aspects du problème qui faudrait considérer afin d'améliorer la solution. L’aspect probabiliste permet d’obtenir un algorithme non déterministe qui pourrait au final mener à un meilleur résultat que un l’algorithme vorace classique mais il n’y a aucune garantie d’amélioration. À ce qui a trait de l’espace mémoire, nous n’avons besoin que des blocs et de la solution en mémoire, ce qui en fait l’algorithme qui consomme le moins de mémoire parmis les trois.

***Algorithme de programmation dynamique***

L’algorithme de programmation dynamique est plus complexe à implémenter que l’algorithme vorace mais il reste quand même simple à réaliser. Son temps de calcul est par contre celui qui est le plus long allant jusqu’à 2,5 minutes pour fichiers de 100 000 blocs (comparativement à 0.02 secondes pour l’algorithme vorace). Il demande aussi plus d’espace mémoire pour garder les solutions temporaires en mémoire, ce qui n’est pas nécessaire pour l’algorithme précédent. Par contre, ce qui est la force de cet algorithme est que la solution obtenue est la meilleure solution au problème. On obtient une tour de près de 50% plus haute (25 000 000 plus haut) que l’algorithme vorace dans le cas d’un fichier de 100 000 blocs.

***Algorithme de recherche avec tabou***

L’algorithme tabou est un compromis entre l’algorithme vorace trop simple et l’algorithme de programmation dynamique qui est très lent mais qui donne toujours le meilleur résultat. Il est fort possible que cet algorithme donne le meilleur résultat mais rien n’est garanti. De plus, les résultats ne sont pas déterministes puisqu’on assigne un nombre d’itérations aléatoires au temps passé dans le tabou. Notre équipe a trouvé que cet algorithme est de loin le plus difficile à implémenter. L’espace mémoire requise est aussi beaucoup plus grande que l’algorithme de programmation dynamique puisqu’on doit réaliser de multiples copies de nos tours temporaires afin de trouver le meilleur candidat à sélectionné. Il est par contre plus rapide que l’algorithme de programmation dynamique et il fournit des solutions meilleures que l’algorithme vorace d’où le compromis entre les deux algorithmes.

### Indiquez sous quelles conditions vous utiliseriez l'un de ces algorithmes plutôt que les deux autres.

Si on désire obtenir une solution rapidement sans se soucier si le résultat pourrait être meilleur, si les blocs sont de telle sorte que la meilleure solution locale à chaque itération fournit la meilleure solution, si on veut implémenter l'algorithme rapidement afin d’obtenir un résultat ou si l’espace mémoire est restreinte, alors l’algorithme vorace probabiliste serait l’algorithme à utiliser. Nous obtiendrions des tours qui pourrait fort probablement être bien plus grande mais si ce n’est pas un critère important alors cet algorithme simple sera suffisant. Il pourrait aussi être intéressant d’utiliser d’utiliser un tel algorithme sur un très grand ensemble de données afin d’obtenir une solution rapidement.

Si, au contraire, l’obtention de la meilleure solution est un critère très important et le temps de calcul n’est pas important, alors l’algorithme de programmation dynamique devrait être choisi puisque c’est le seul algorithme qui assure la meilleure solution malgré un temps d’exécution très long comparativement aux deux autres algorithmes. Il est plus compliqué à implémenter que l’algorithme vorace mais il reste aisé de l’implémenter dans un court délai. Il pourrait aussi être utilisé sur des exemplaires de petite taille puisque son temps d’exécution reste court dans un tel cas.

Finalement, si la solution est un critère important mais que le temps d’exécution l’est encore plus, alors on devrait plutôt opter pour un compromis entre les deux algorithmes précédent et choisir l’algorithme de recherche avec tabou. Par contre, l’espace mémoire ne doit pas être un critère important puisque de multiples copies sont nécessaire pour cet algorithme. La réponse obtenue ne sera pas forcément la meilleure solution mais devrait toutefois être bien meilleure que l’algorithme vorace avec un temps d’exécution moindre que l’algorithme de programmation dynamique. La recherche tabou sur des exemplaires de très grande taille est donc déconseillée puisque la consommation de ressources serait alors très grande avec toutes les copies de tours et les tabous nécessaires pour ce dernier algorithme.

# Autres critères de correction

### Respect de l’interface tp.sh

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

Utilisation

tp.sh -a [vorace | progdyn | tabou] -e [path\_vers\_exemplaire]

Arguments optionnels

-p affiche les blocs utilisés dans la construction de la tour chacun sur une ligne (hauteur, largeur, profondeur) en commençant par le bas.

-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

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

### Présentation générale

|  |  |
| --- | --- |
| 1 | / 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.