Chapitre 2 : Techniques de conception d'algorithmes,

Exemple algorithmes glouton

1. Les techniques de conception d'algorithmes

Les techniques de conception d'algorithmes font référence à des approches et des méthodologies utilisées pour développer des solutions efficaces et optimales à des problèmes informatiques, économiques, industriels, etc. La conception d'algorithmes joue un rôle essentiel dans le développement de logiciels et dans la résolution de problèmes complexes.

L'élaboration d'un algorithme implique de formuler une séquence d'instructions bien définies et structurées, qui permettent de résoudre un problème spécifique. Les techniques de conception d'algorithmes visent à produire des solutions qui sont à la fois **correctes**, **efficaces** en termes de temps d'exécution et d'utilisation des ressources, et **faciles à comprendre et à maintenir**.

Parmi les méthodes de conception les plus utilisées, on peut citer :

- **Force brute :** Cette technique implique de tester toutes les possibilités de manière exhaustive jusqu'à ce que la solution soit trouvée. Bien que souvent inefficace pour des problèmes complexes, elle peut être utile pour des problèmes simples et petits.

- **Diviser pour régner :** Ce concept consiste à diviser un problème en sous-problèmes plus petits, résoudre ces sous-problèmes de manière récursive, puis combiner leurs solutions pour obtenir la solution du problème initial. L'exemple classique est l'algorithme de tri fusion.

- **Programmation dynamique :** Cette technique consiste à résoudre un problème en le décomposant en sous-problèmes plus petits et à mémoriser les solutions de ces sous-problèmes pour éviter de recalculer. Cela permet de réduire d'une manière considérable le nombre de calculs répétés. L'algorithme de calcul de la séquence de Fibonacci est un exemple fréquemment utilisé pour illustrer cette technique.

- **Algorithmes gloutons :** contrairement aux algorithmes dynamiques, les algorithmes gloutons font des choix localement optimaux à chaque étape dans l'espoir que ces choix mèneront à une solution globale optimale. Le problème du rendu de monnaie et le problème du sac à dos fractionnaire en sont des exemples.

- **Recherche et exploration :** Cette technique est utilisée pour explorer un espace de solutions à la recherche d'une solution optimale. Les algorithmes de recherche en profondeur et de recherche en largeur sont des exemples classiques de cette approche.

- **Recherche binaire :** Lorsqu'une liste est triée, la recherche binaire est une technique efficace pour trouver un élément particulier dans la liste en réduisant à chaque étape l'intervalle de recherche de moitié.

- **Algorithme de rétrogradation (Backtracking) :** C'est une technique qui consiste à essayer toutes les possibilités, en revenant en arrière dès qu'une possibilité s'avère être une impasse. Les problèmes de Sudoku et les problèmes de voyageur de commerce peuvent être résolus à l'aide de cette technique.

- **Programmation linéaire et entière :** Ces techniques sont utilisées pour résoudre des problèmes d'optimisation sous contraintes linéaires. La programmation linéaire résout des problèmes continus, tandis que la programmation entière résout des problèmes où les variables doivent prendre des valeurs entières.

- **Méthodes heuristiques et métaheuristiques :** Ces techniques sont utilisées pour résoudre des problèmes d'optimisation difficiles pour lesquels il n'existe pas d'algorithme exact efficace. Les algorithmes génétiques, les algorithmes de recuit simulé et les algorithmes fourmis sont des exemples de métaheuristiques.

- **Décomposition structurelle :** Analyser la structure et les propriétés du problème pour le décomposer en sous-structures plus simples.

- **Simuler le problème :** Imitation du comportement du problème par un programme pour trouver une solution.

Ces techniques de conception d'algorithmes sont des outils puissants pour résoudre différents types de problèmes, mais il est important de choisir la technique la mieux adaptée en fonction des caractéristiques et des contraintes spécifiques du problème que vous essayez de résoudre.

2. Comment choisir la meilleure technique de conception d'algorithme pour un problème donné ?

Parfois, il est très difficile de choisir le bon algorithme de résolution pour des raisons différentes. Voici quelques pistes pour choisir la meilleure technique de conception d'algorithme pour un problème donné :

- Analyser la structure et les propriétés du problème : est-il divisible en sous-parties? existe-t-il des sous-structures récurrentes?
- Identifier si certains sous-problèmes se répètent : la programmation dynamique permet d'éviter les calculs répétés.
- Le problème implique-t-il de nombreuses combinaisons à explorer?: le backtracking est adapté.
- Peut-on trouver une solution optimale en faisant des choix locaux optimaux?: l'approche gloutonne a des chances de réussir.

- Le problème possède-t-il une structure récursive évidente?: la récursivité est alors naturelle.
- Le problème peut-il se traduire sous forme de modèle mathématique?: les outils mathématiques peuvent résoudre directement le problème.
- L'espace des solutions est-il gigantesque ? la force brute n'est pas réaliste, il faut décomposer.
- Le temps de calcul autorisé est-il court ? privilégier les techniques les plus efficaces même si moins optimales.
- Les données du problème se prêtent-elles bien à une division en sous-parties?: l'approche diviser pour régner convient.

Il s'agit d'identifier la structure du problème et d'opter pour la technique de conception d'algorithme la mieux adaptée. Il peut y avoir plusieurs bonnes réponses possibles.

3. Les algorithmes gloutons (greedy algorithms)

Les algorithmes gloutons (greedy algorithms), également connus sous le nom d'algorithmes voraces, sont une catégorie d'algorithmes de conception qui résolvent des problèmes en effectuant des choix localement optimaux à chaque étape, dans l'espoir que ces choix mèneront à une solution globale optimale. En d'autres termes, à chaque étape, l'algorithme prend la meilleure décision possible sans se soucier des conséquences à long terme. Bien que les algorithmes gloutons ne garantissent pas toujours la solution optimale, ils sont souvent rapides et simples à mettre en œuvre, ce qui les rend utiles pour de nombreux problèmes.

L'algorithme générique pour la construction d'un algorithme glouton est le suivant :

```
Fonction Greedy( C : Ensemble) : Ensemble
S ← ∅
TANT QUE ¬Solution(S) ∧ C ≠ ∅ FAIRE
x ← élément de C maximisant Select(X)
C ← C – {x}
SI Réalisable( S ∪ {x}) ALORS FAIRE
S ← S ∪ {x}
SI Solution(S) RETOURNER S
SINON RETOURNER «Pas de solution avec cette approche»
```

Où les composants suivants doivent être présents

a) Un ensemble de candidats disponibles : **C**
b) Un ensemble de candidats déjà choisis : **S**
c) Un test de solution : ***Solution(.)***

d) Un test de réalisabilité : ***Réalisable(.)***
e) Une fonction de sélection : ***Select(.)***
f) Une fonction objective (fonction à optimiser)

3.1 Caractéristiques des Algorithmes Gloutons :

- **Principe de Choix Local :** Les algorithmes gloutons prennent des décisions en se basant uniquement sur l'information disponible localement à chaque étape. Ils ne tiennent pas compte des conséquences à long terme de leurs choix.
- **Solutions Localement Optimaux :** À chaque étape, un algorithme glouton choisit la meilleure option locale, c'est-à-dire celle qui semble être la meilleure parmi les choix immédiats disponibles. Cela conduit à des solutions localement optimales.
- **Stratégie de Construction Incrémentale :** Les solutions sont construites progressivement, étape par étape, en ajoutant ou en sélectionnant des éléments de manière incrémentale. Chaque étape est guidée par la recherche du meilleur choix local.
- **Absence de Retour en Arrière :** Les algorithmes gloutons ne reviennent généralement pas en arrière pour réévaluer ou changer une décision prise précédemment. Une fois qu'un choix est fait, il est définitif.

3.2 Propriétés des problèmes résolubles par un algorithme glouton

- Le problème doit être d'optimisation.
- Sous-structure optimale : Le problème peut se décomposer en une série de choix locaux.
- Le choix optimal local mène parfois à une solution optimale globale.
- Les décisions locales sont indépendantes.

3.3 Les principaux avantages des algorithmes gloutons sont:

- **Simplicité de conception :** Les algorithmes gloutons sont souvent plus simples à concevoir que d'autres techniques car ils construisent directement la solution sans recourir à des calculs complexes.
- **Efficacité :** De nombreux algorithmes gloutons ont une complexité algorithmique faible (linéaire, logarithmique...) et passent à l'échelle sur de grands jeux de données.
- **Garantie de résultat :** Si le problème vérifie les propriétés d'optimalité locale, un algorithme glouton est assuré de fournir une solution réalisable même si elle n'est pas forcément optimale.

- **Implémentation simple :** Leur conception incrémentale en fait des algorithmes souvent plus simples à coder que des approches divisées ou par force brute.

3.4 Limites des algorithmes gloutons

Voici quelques inconvénients potentiels des algorithmes gloutons :

- **Non-optimalité de la solution :** Un algorithme glouton ne garantit pas de trouver la solution optimale globale du problème, seulement une solution réalisable.
- **Sensibilité aux choix initiaux:** Les algorithmes gloutons sont sensibles aux choix initiaux ou à l'ordre dans lequel les décisions sont prises. Un ordre différent des choix initiaux peut conduire à une solution différente, voire sous-optimale.
- **Problèmes d'indépendance locale :** Si les choix locaux ne sont pas complètement indépendants, la solution gloutonne peut être très éloignée de l'optimum.
- **Impossibilité de remettre en cause les décisions :** Contrairement à d'autres techniques comme le backtracking, un algorithme glouton ne peut revenir sur ses choix précédents pour améliorer la solution.
- **Difficulté de preuve :** Il est souvent compliqué de prouver formellement qu'un algorithme glouton donnera bien une solution optimale pour un problème donné.
- **Complexité accrue sur certains problèmes :** La gloutonnerie n'est pas toujours la technique la plus efficace en termes de complexité algorithmique.

En résumé, les principaux inconvénients sont le manque d'optimalité garantie de la solution et l'impossible reconsidération des décisions locales. Mais pour de nombreux problèmes réels, le rapport simplicité/efficacité des gloutons est très intéressant par rapport à des techniques plus complexes.

3.5 Variantes et améliorations possibles

- **Algorithmes gloutons avec lookahead** :ils prennent en compte des informations à l'avance sur les conséquences potentielles de chaque décision. Cette approche tente d'anticiper les résultats futurs des choix actuels, ce qui peut conduire à des solutions de meilleure qualité
- **Algorithme glouton randomisé:** Cette variante consiste à introduire une composante de hasard dans l'algorithme glouton. À chaque étape, plusieurs choix sont générés de manière aléatoire, puis le meilleur parmi eux est sélectionné. Cela peut aider à éviter d'être coincé dans un optimum local.

- **Algorithmes gloutons à deux phases :** Les algorithmes gloutons à deux phases sont une approche de résolution de problèmes d'optimisation combinatoire qui combine deux étapes distinctes : la phase de construction et la phase d'amélioration. Cette approche vise à obtenir une solution initiale grâce à une stratégie gloutonne, puis à l'améliorer davantage à l'aide d'une méthode d'optimisation locale ou d'une métaheuristique.

Exemples d'Algorithmes Gloutons :

3.6 Problème du rendu de monnaie :

Supposons que vous deviez rendre une somme d'argent en utilisant le moins de pièces et de billets possibles.

Résolution par un algorithme glouton

On suppose que les clients ne vous donnent que des sommes entières en euros (pas de centimes pour simplifier) ;

Les valeurs des pièces et billets à votre disposition sont : 1, 2, 5, 10, 20, 50, 100, 200 et 500. On suppose que vous avez autant d'exemplaires que nécessaire de chaque pièce et billet. Dans la suite, afin de simplifier, nous désignerons par « pièces » à la fois les pièces et les billets.

Un algorithme glouton consiste à toujours choisir la pièce ou le billet de la valeur la plus élevée qui ne dépasse pas le montant restant.

Exemple :

Un client nous achète un objet qui coûte 53 euros. Il paye avec un billet de 200 euros. On doit donc lui rendre 147 euros. Une façon de lui rendre la monnaie est de le faire avec un billet de 100, deux billets de 20, un billet de 5 et une pièce de 2.

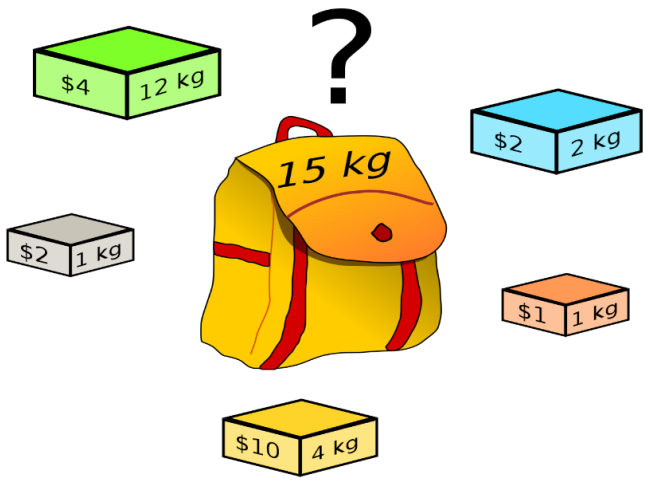
Pour minimiser le nombre de pièces à rendre, il apparaît la stratégie suivante :

- On commence par rendre la pièce de plus grande valeur possible ;
- On déduit cette valeur de la somme (encore) à rendre ;
- On recommence, jusqu'à obtenir une somme nulle.

TP à faire : écrire un programme en python pour résoudre le problème du rendu de monnaie par un algorithme gloutons

3.7 Problème du sac à dos (knapsack problem) :

Dans ce problème, vous avez un sac à dos de capacité **C** et une liste d'objets avec des poids et des valeurs. L'objectif est de remplir le sac de manière à maximiser la valeur totale des objets tout en respectant la capacité du sac. Ce problème est très important possédant des variantes et applications multiples dans plusieurs domaines.



Voici les principaux algorithmes gloutons utilisés pour résoudre le problème du sac à dos:

- **Algorithme du rapport valeur/poids:**
  - Calcul du rapport valeur/poids pour chaque objet
  - Trie par ordre décroissant des rapports
  - Remplissage du sac en prenant les objets dans cet ordre
  - S'arrête dès que la capacité est atteinte

Cet algorithme donne une solution presque optimale pour le sac à dos en valeurs.

- **Algorithme du plus petit poids restant :**
  - Trie les objets par ordre de poids croissant
  - Remplissage du sac en prenant les objets dans cet ordre
  - S'arrête dès que la capacité est atteinte

Cet algorithme permet de maximiser le nombre d'objets dans le sac.

- **Algorithme du plus grand profit:**
  - Trie les objets par ordre décroissant de valeur
  - Remplissage du sac en prenant les objets dans cet ordre
  - S'arrête dès que la capacité est atteinte

Cherche à maximiser la valeur totale même si le nombre d'objets n'est pas maximal.

Exemple d'application de l'algorithme valeur/poids

Soit un sac de capacité 10kg et les objets suivants:

Objet 1: Valeur 4, Poids 5
Objet 2: Valeur 3, Poids 4
Objet 3: Valeur 2, Poids 3
Objet 4: Valeur 1, Poids 1

L'algorithme glouton fonctionne de la manière suivante:

1. **Calcul du rapport valeur/poids pour chaque objet**

Objet 1: 4/5 = 0,8
Objet 2: 3/4 = 0,75
Objet 3: 2/3 = 0,66
Objet 4: 1/1 = 1

2. **Tri par ordre décroissant des rapports**

Objet 4, Objet 1, Objet 2, Objet 3

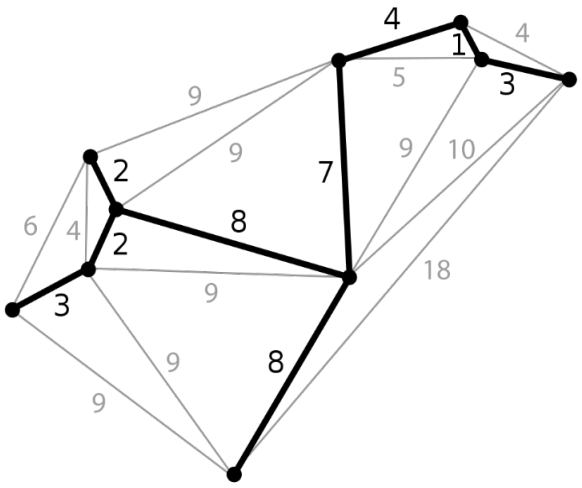
3. **Remplissage du sac en ajoutant les objets un par un dans cet ordre jusqu'à atteindre la capacité limite**

On ajoute Objet 4 (1kg), puis Objet 1 (5kg), le sac est plein à 6kg.
La solution gloutonne est {Objet 4, Objet 1} pour une valeur totale de 4+1 = 5. C'est l'optimum pour ce problème du sac à dos.

TP à faire : écrire les trois algorithmes gloutons en python

3.8 Algorithme de Prim pour l'arbre de recouvrement minimum :

L'algorithme de Prim résout le problème de trouver un arbre de recouvrement minimum (Minimum Spanning Tree) dans un graphe pondéré et non orienté. Il commence par un nœud arbitraire et ajoute itérativement le nœud le plus proche (ou ayant le poids le plus petit) du sous-arbre partiellement construit.

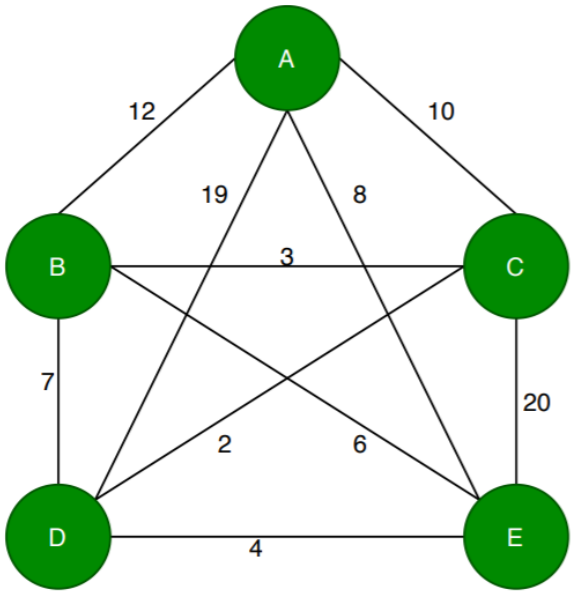


L'algorithme de Prim

1. **Initialisation** :
    1. Choisissez un sommet de départ arbitraire pour commencer la construction de l'arbre couvrant de poids minimum.
    2. Créez un ensemble vide pour stocker les sommets inclus dans l'arbre couvrant.
2. **Étape de Boucle** : Répétez les étapes suivantes jusqu'à ce que **tous les sommets soient inclus dans l'arbre couvrant** :
3. **Trouver l'Arête la Plus Légère** : Parmi les arêtes qui relient un sommet inclus dans l'arbre couvrant à un sommet non inclus, sélectionnez l'arête de poids minimum.
4. **Ajouter le Sommet et l'Arête** :
    1. Ajoutez le sommet connecté par l'arête sélectionnée à l'ensemble des sommets inclus dans l'arbre couvrant.
    2. Ajoutez également l'arête sélectionnée à l'arbre couvrant.
5. **Résultat** : l'arbre de poids minimum du graphe initial.

3.9 Le problème du voyageur de commerce

Le problème du voyageur de commerce (Travelling salesman problem : TSP) consiste à trouver le chemin le plus court qui passe par chaque ville exactement une fois et revient à la ville de départ. Ce problème est très important car il possède plusieurs applications en transport et réseaux et d'autres domaines.



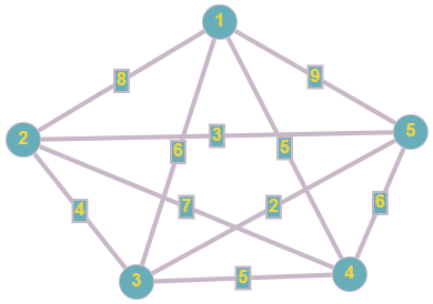
3.9.1 Algorithmes gloutons de résolution

A. Algorithme plus proche voisin

L'algorithme du Plus Proche Voisin est une technique gloutonne qui résout ce problème en choisissant à chaque étape la ville la plus proche non visitée. L'algorithme du Plus Proche Voisin est une technique gloutonne pour résoudre le problème du voyageur de commerce. Son principe est assez simple : à chaque étape, l'algorithme choisit la ville non visitée la plus proche de la ville actuelle et ajoute cette ville à la tournée en cours. L'algorithme continue ainsi jusqu'à ce que toutes les villes aient été visitées, et la dernière étape consiste à revenir à la ville de départ pour compléter le cycle.

Voici le principe de l'algorithme du Plus Proche Voisin en quelques étapes clés :

1. **Initialisation :** Choisissez une ville de départ arbitraire pour commencer la tournée.
2. **Sélection de la ville suivante :** À chaque étape, à partir de la ville actuelle, sélectionnez la ville non visitée la plus proche en termes de distance. Cela signifie que vous calculez la distance entre la ville actuelle et toutes les autres villes non visitées, puis choisissez celle qui a la distance minimale.



3. **Mise à jour de la tournée :** Ajoutez la ville choisie à la tournée en cours et marquez-la comme visitée.
4. **Répétition :** Répétez les étapes 2 et 3 jusqu'à ce que toutes les villes aient été visitées.
5. **Retour à la ville de départ :** Une fois que toutes les villes ont été visitées, ajoutez la dernière étape pour retourner à la ville de départ et fermer la boucle du cycle.
6. **Calcul de la distance totale :** Calculez la distance totale parcourue en additionnant les distances entre les villes dans l'ordre de la tournée.

L'algorithme du Plus Proche Voisin est simple à implémenter et fournit rapidement une solution pour de petites instances du problème du voyageur de commerce. Cependant, il ne garantit pas toujours la solution optimale, car il peut conduire à des tournées qui sont localement optimales mais pas globalement optimales. Dans certains cas, l'algorithme peut conduire à des tournées beaucoup plus longues que la solution optimale.

Exemple : Considérons un ensemble de villes avec leurs distances entre deux villes :

	A	**B**	**C**	**D**	**E**
A	0	8	6	5	9
B	8	0	4	7	3
C	6	4	0	5	2

D 5 7 5 0 6

E 9 3 2 6 0

Supposons que nous partons de la ville 0 (A) comme ville de départ. Voici comment l'algorithme du Plus Proche Voisin serait appliqué :

1. Nous commençons à la ville 0 (A).
2. La ville la plus proche de A est la ville 3 (D) avec une distance de 5.
3. Maintenant, nous sommes à la ville 3 (D). La ville la plus proche de D est la ville 2 (C) avec une distance de 5.
4. Nous sommes maintenant à la ville 2 (C). La ville la plus proche de C est la ville 1 (B) avec une distance de 4.
5. Nous sommes à la ville 1 (B). La ville la plus proche de B est la ville 4 (E) avec une distance de 3.
6. Enfin, nous sommes à la ville 4 (E). Pour compléter la tournée, nous revenons à la ville 0 (A) avec une distance de 9.

Le chemin résultant de l'algorithme du Plus Proche Voisin est donc : A → D → C → B → E → A.

La distance totale parcourue dans cette tournée est : 5 + 5 + 4 + 3 + 9 = 26.

Veuillez noter que cette solution n'est pas garantie d'être optimale. Dans cet exemple, il se trouve que l'algorithme du Plus Proche Voisin a produit une tournée relativement longue comparée à la solution optimale.

TP à faire : écrire l'algorithme du plus proche voisin en python

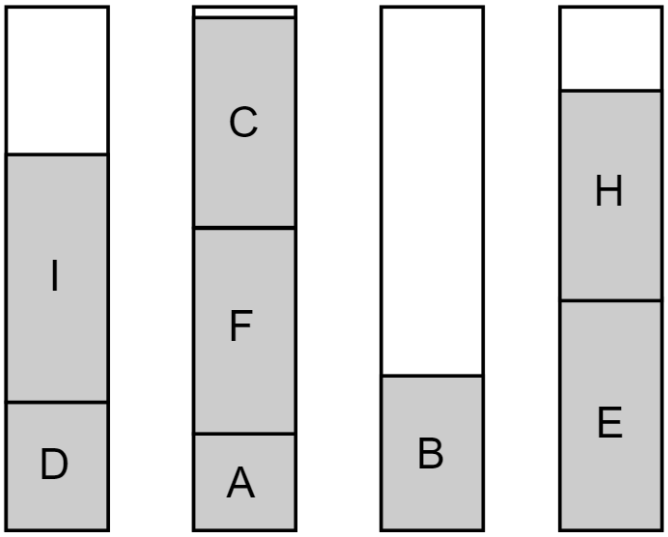
B. Algorithme de l'insertion la moins chère (Cheapest Insertion) :

- Commencez avec un cycle contenant une seule ville.
- À chaque étape, choisissez la ville non visitée qui réduit le plus la distance totale parcourue si elle était insérée dans le cycle existant.
- Insérez la ville sélectionnée à l'emplacement approprié dans le cycle existant pour former un cycle plus long.
- Répétez les étapes précédentes jusqu'à ce que toutes les villes soient visitées.

Cet algorithme est légèrement plus complexe que l'algorithme du plus proche voisin, mais il peut donner de meilleures solutions dans certains cas.

3.10 Les algorithmes de gloutons de remplissage et allocation de ressources

Les algorithmes gloutons First Fit, Best Fit, Worst Fit et Next Fit sont couramment utilisés pour résoudre des problèmes d'allocation de ressources, tels que le placement d'objets dans des conteneurs ou l'allocation de mémoire. Voici une description de chacun de ces algorithmes :



1. ***First Fit (Premier ajustement) :***
    - Parcourez les objets à allouer un par un dans l'ordre où ils sont donnés.
    - Pour chaque objet, recherchez le premier conteneur disponible dans lequel il peut être placé sans dépasser sa capacité.
    - Placez l'objet dans ce conteneur et marquez-le comme utilisé.
    - Si aucun conteneur disponible ne peut accueillir l'objet, créez un nouveau conteneur et placez-y l'objet.

L'algorithme First Fit est rapide et simple à mettre en œuvre, mais il peut conduire à une utilisation inefficace de l'espace, car il peut laisser de petits espaces inutilisés dans les conteneurs.

2. ***Best Fit (Meilleur ajustement) :***
    - Parcourez les objets à allouer un par un dans l'ordre où ils sont donnés.
    - Pour chaque objet, recherchez le conteneur disponible qui a la plus petite taille disponible tout en pouvant accueillir l'objet.
    - Placez l'objet dans ce conteneur et marquez-le comme utilisé.
    - Si aucun conteneur disponible ne peut accueillir l'objet, créez un nouveau conteneur et placez-y l'objet.

L'algorithme Best Fit tente de minimiser les espaces inutilisés en utilisant le conteneur qui convient le mieux à chaque objet. Cependant, il est généralement plus lent que l'algorithme First Fit en raison de la recherche du meilleur conteneur à chaque étape.

3. ***Worst Fit (Pire ajustement) :***
    - Parcourez les objets à allouer un par un dans l'ordre où ils sont donnés.
    - Pour chaque objet, recherchez le conteneur disponible qui a la plus grande taille disponible tout en pouvant accueillir l'objet.
    - Placez l'objet dans ce conteneur et marquez-le comme utilisé.
    - Si aucun conteneur disponible ne peut accueillir l'objet, créez un nouveau conteneur et placez-y l'objet.

L'algorithme Worst Fit vise à maximiser les espaces inutilisés en utilisant le conteneur qui a la plus grande taille disponible. Il peut être utile dans certains scénarios pour gérer des objets de tailles variables.

4. ***Next Fit (Ajustement suivant) :***
    - Commencez avec un conteneur vide.
    - Parcourez les objets à allouer un par un dans l'ordre où ils sont donnés.
    - Recherchez le conteneur disponible le plus récemment utilisé.
    - Si l'objet peut être placé dans ce conteneur sans dépasser sa capacité, placez-le et marquez-le comme utilisé.
    - Sinon, créez un nouveau conteneur, placez-y l'objet et marquez-le comme utilisé **et recommencer à partir de ce conteneur**.

L'algorithme Next Fit est similaire à l'algorithme First Fit, mais il tente de tirer parti de la localité des objets pour réduire le nombre de conteneurs utilisés. Cependant, il peut encore conduire à une utilisation inefficace de l'espace.

Ces quatre algorithmes gloutons fournissent des solutions rapides mais non optimales aux problèmes d'allocation de ressources. Le choix de l'algorithme dépend des contraintes spécifiques du problème et des objectifs recherchés en termes d'utilisation de l'espace.

5. Conclusion

En guise de conclusion sur les algorithmes gloutons de construction, voici les points clés à retenir:

- Les algorithmes gloutons permettent de résoudre de nombreux problèmes combinatoires de façon simple et efficace, en faisant des choix locaux optimaux à chaque étape.
- Ils ne garantissent cependant pas toujours d'atteindre une solution optimale globale, mais souvent une bonne approximation.
- Leur application requiert que le problème puisse se décomposer en choix locaux indépendants et qu'une solution optimale résulte d'une succession de décisions optimales locales.
- Des problématiques telles que le plus court chemin, les problèmes d’assignements, le remplissage du sac à dos se prêtent bien à l'approche gloutonne.
- Des algorithmes classiques comme Prim, Kruskal, Huffman, first fit ont démontré l'efficacité de la technique pour ces problèmes.
- Leur complexité algorithmique est souvent réduite par rapport à des méthodes exactes.
- Des améliorations comme le lookahead permettent parfois d'atteindre l'optimalité.
- La conception gloutonne est simple mais la preuve d'optimalité pour un problème donné reste difficile.
- La programmation dynamique reste plus garantie sur l'optimalité de la solution trouvée.

Les algorithmes gloutons de construction constituent une approche algorithmique intuitive, efficace et largement utilisée pour la résolution approchée de problèmes d'optimisation combinatoire

Références

- https://www.topcoder.com/thrive/articles/Greedy%20is%20Good
- https://dept-info.labri.fr/~gavoille/UE-TAP/cours.pdf
- https://www.geeksforgeeks.org/introduction-to-greedy-algorithm-data-structures-and-algorithm-tutorials/
- https://mblondin.espaceweb.usherbrooke.ca//cours/ift436_a21/notes.pdf
- https://www.dil.univ-mrs.fr/~gcolas/algo-licence/slides/gloutons.pdf
- Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to algorithms." Massachusetts Institute of Technology (2009).
- Jeff Erickson. "Algorithms." University of Illinois at Urbana-Champaign (2022). https://jeffe.cs.illinois.edu/teaching/algorithms/