REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'enseignement supérieur et de la recherche scientifique

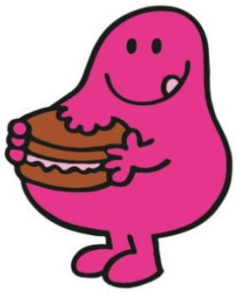
Université Larbi Ben M'Hidi Oum El Bouaghi

Faculté des sciences exactes et science de la nature et de la vie

Département de mathématique d'informatique et

Cours : Méthodes Algorithmiques

Chapitre 1 : Méthode gloutonne



Proposé par Dr. Kenza Belhouchette

Edition 1.0

2025/ 2026

Introduction

Dans le domaine de l'algorithmique, les *méthodes gloutonnes* se distinguent par leur approche simple et intuitive pour résoudre des problèmes d'optimisation combinatoire. Ces méthodes visent à construire une solution optimale en prenant une série de décisions locales, chacune étant le meilleur choix possible à ce moment-là, sans se préoccuper des conséquences futures. Ce principe, bien que séduisant, nécessite une compréhension approfondie des propriétés du problème en question pour garantir que la solution finale sera effectivement optimale.

L'attrait des méthodes gloutonnes réside dans leur capacité à produire des solutions rapidement, souvent en temps polynomial. Leur utilisation est répandue dans des domaines variés tels que la théorie des graphes, la recherche opérationnelle, et l'optimisation. Parmi les exemples emblématiques que nous allons les détailler dans ce chapitre, on trouve le problème du chemin le plus court, les arbres couvrants minimaux et le problème du change de monnaie. Chaque problème présente ses propres défis, mais les techniques gloutonnes offrent des stratégies élégantes et efficaces pour y faire face.

En examinant ces cas, nous mettrons en lumière les principes fondamentaux des méthodes gloutonnes, leurs avantages et leurs limites, tout en démontrant leur pertinence dans la résolution de problèmes complexes dans divers domaines.

1. Définition des algorithmes gloutons

Les algorithmes gloutons sont une classe d'algorithmes qui, pour résoudre un problème, prennent des décisions successives en choisissant à chaque étape l'option qui semble la meilleure, sans revenir en arrière. Utilisés principalement pour des problèmes d'optimisation, ces algorithmes font un choix localement optimal dans l'espoir que ce choix mènera à une solution globalement optimale. Toutefois, il est important de noter que les algorithmes gloutons ne conduisent pas toujours à la solution optimale, car la nature de certains problèmes peut nécessiter une exploration plus approfondie pour garantir une solution satisfaisante

2. La Stratégie Gloutonne

La stratégie gloutonne est une approche algorithmique qui repose sur la prise de décisions séquentielles, qui s'articule autour de trois étapes principales :

2.1 Choix Local Optimal

À chaque étape de l'algorithme, un choix est fait en fonction d'une règle locale d'optimalité. Cela implique de sélectionner l'option qui semble être la meilleure à l'instant présent, sans tenir compte des conséquences futures. Ce choix est généralement basé sur des critères spécifiques, tels que le coût minimal, le profit maximal, ou toute autre mesure pertinente

2.2 Récursivité ou itérativité

Après avoir effectué un choix, le problème initial est réduit. Cela signifie que le problème restant est plus petit ou moins complexe, L'algorithme reprend ensuite le processus en choisissant à nouveau la meilleure option pour le nouveau sous-problème

2.3 Finalité

Le processus continue jusqu'à ce que toutes les décisions aient été prises, aboutissant à une solution complète. La solution finale est le résultat des choix locaux effectués à chaque étape, assemblés pour former une solution globale.

3. Preuve d'Optimalité

Pour qu'un algorithme glouton soit efficace et garantisse une solution optimale, le problème doit satisfaire deux propriétés fondamentales :

Propriété de sous-structure optimale : Cette propriété stipule que le problème peut être décomposé en sous-problèmes dont les solutions optimales peuvent être combinées pour obtenir la solution optimale globale. En d'autres termes, la résolution optimale d'une partie du problème assure l'optimalité de la solution complète.

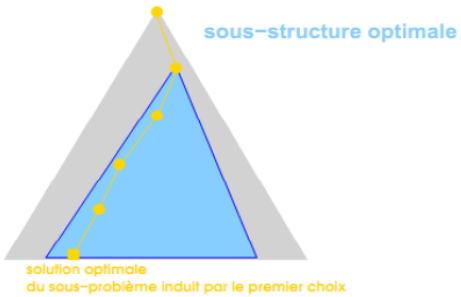


Fig1.**sous-structure optimale**

Propriété du choix glouton : Cette propriété indique que le choix localement optimal à chaque étape doit mener à une solution optimale globale. Ainsi, la meilleure solution peut être atteinte en se basant uniquement sur des décisions locales, sans nécessiter de revenir sur des choix antérieurs ou d'explorer d'autres alternatives.

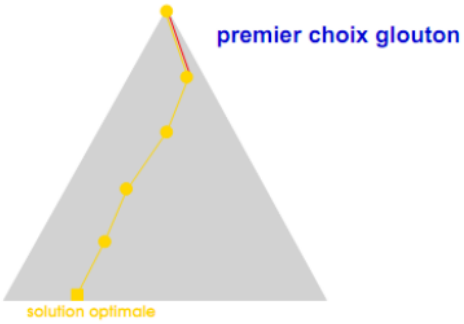


Fig2. **choix glouton**

4. Schéma Général d'un Algorithme Glouton

Ce schéma illustre comment les algorithmes gloutons prennent des décisions itératives basées sur des choix locaux pour construire progressivement une solution optimale

Voici les principales étapes du schéma général :

1. **Fonction de Sélection (select) :** Choisir l'élément optimal parmi les éléments restants en fonction du critère glouton défini. Cette fonction évalue chaque option disponible et retourne celle qui maximise l'optimalité locale.
2. **Ajout à la Solution Partielle :** Intégrer l'élément sélectionné dans la solution partielle, en mettant à jour les variables pertinentes (comme le coût total ou d'autres mesures d'évaluation).
3. **Vérification de Complétude :** Vérifier si la solution partielle constitue une solution complète. Si oui, retourner la solution finale. Sinon, répéter le processus en retournant à l'étape de sélection.

5. L'Arbre Couvrant Minimal (ACM)

Dans le contexte des algorithmes gloutons, l'arbre couvrant minimal (ACM) constitue un sujet fondamental qui mérite une attention particulière. L'ACM est défini comme un sous-ensemble d'arêtes d'un graphe qui relie tous les sommets sans former de cycles, tout en minimisant le poids total des arêtes. Ce concept est d'une importance capitale dans de nombreux domaines, tels que la conception de réseaux, la logistique et l'optimisation des infrastructures. En adoptant une approche gloutonne, des algorithmes tels que ceux de Prim et de Kruskal permettent de trouver efficacement l'ACM, en prenant des décisions basées sur des choix locaux optimaux. Dans cette section, nous explorerons les caractéristiques et les propriétés de l'ACM, ainsi que les algorithmes associés, afin de démontrer leur pertinence et leur efficacité dans la résolution de problèmes d'optimisation complexes.

5.1 Définition d'un arbre couvrant minimal

Un ACM d'un graphe non orienté et pondéré est un sous-ensemble d'arêtes qui relie tous les sommets du graphe sans former de cycles, tout en minimisant la somme des poids des arêtes. Autrement dit, un ACM est un arbre qui couvre tous les sommets du graphe, et dont le coût total, calculé en additionnant les poids des arêtes, est le plus faible possible parmi tous les arbres couvrants possibles.

Formellement, pour un graphe G(V,E), où V est l'ensemble des sommets et E est l'ensemble des arêtes avec des poids associés, un arbre couvrant T⊆E est minimal si :

1. **Cohérence** : T contient tous les sommets de V.
2. **Acyclicité** : T ne contient aucun cycle.
3. **Minimisation du Coût** : La somme des poids des arêtes dans T est minimale, soit

$$\sum_{e \in T} w(e)$$

où w(e) Représente le poids de l'arête e.

5.2 Problèmes Liés aux Arbres Couvrants Minimaux

Il existe plusieurs problèmes importants liés aux arbres couvrants minimaux (ACM), chacun présentant des défis uniques et des applications spécifiques :

1. Problème de Base : Trouver un Arbre Couvrant Minimal
   Ce problème consiste à déterminer l'arbre couvrant minimal d'un graphe non orienté et pondéré. L'objectif est de sélectionner un sous-ensemble d'arêtes qui relie tous les sommets tout en minimisant le poids total des arêtes. Les algorithmes de Prim et de Kruskal sont souvent utilisés pour résoudre ce problème de manière efficace. Ce problème est fondamental dans de nombreux domaines, tels que les réseaux informatiques, la conception de circuits et l'optimisation logistique.
2. Problème du Second Meilleur Arbre Couvrant Minimal
   Ce problème vise à identifier le deuxième arbre couvrant minimal d'un graphe, qui est un arbre couvrant ayant un coût supérieur au premier arbre couvrant minimal, mais inférieur à tous les autres arbres couvrants. Trouver le second meilleur ACM est utile dans des applications où il est nécessaire d'avoir une solution alternative en cas d'échec de la solution optimale ou de changements dans le réseau.
3. Problème de l'ACM avec Contraintes
   Dans ce cas, des contraintes supplémentaires doivent être prises en compte lors de la recherche de l'ACM. Par exemple, certaines arêtes peuvent avoir des limites de capacité, ou il peut être nécessaire d'inclure certaines arêtes spécifiques dans l'arbre couvrant. Ce problème nécessite une approche d'optimisation qui prend en compte ces contraintes tout en cherchant à minimiser le coût total des arêtes.
4. Problème de Steiner : ACM avec Sommets Spécifiques
   Le problème de Steiner consiste à trouver un arbre couvrant minimal qui relie un ensemble

donné de sommets terminaux dans un graphe. Dans ce cas, des sommets supplémentaires appelés sommets de Steiner peuvent être ajoutés à l'arbre pour réduire le coût total. Ce problème est particulièrement pertinent dans des applications telles que la conception de réseaux, où il peut être avantageux d'inclure des points intermédiaires pour optimiser la connectivité et réduire les coûts.

5. Problème de l'ACM dans un Graphe Dynamique

Le problème d'ACM dans un graphe dynamique concerne les situations où la structure du graphe peut changer au fil du temps, par exemple en ajoutant ou en supprimant des sommets ou des arêtes. Ces changements peuvent affecter le poids des arêtes ou même la connectivité du graphe. Trouver un ACM efficace dans ce contexte nécessite des techniques d'algorithmes dynamiques pour recalculer un ACM efficacement à chaque changement plutôt que de recalculer tout depuis le début.

6. Problème de Minimisation des Délais dans un ACM

Le problème de minimisation des délais dans un arbre couvrant minimal (ACM) se concentre sur la conception d'un arbre couvrant qui non seulement minimise le coût total des arêtes, mais prend également en compte les délais associés à chaque arête. Ce problème est particulièrement pertinent dans des applications telles que les réseaux de communication ou de transport, où le temps de transmission ou de parcours est un facteur critique.

5.3 Algorithme de Kruskal

L'algorithme de Kruskal est une méthode efficace pour résoudre le problème de l'arbre couvrant minimal (ACM) dans un graphe non orienté et pondéré. Développé par Joseph Kruskal en 1956, cet algorithme se distingue par son approche basée sur le tri des arêtes et l'utilisation de la structure de données de l'union-find pour éviter la formation de cycles.

L'objectif principal de l'algorithme de Kruskal est de sélectionner un sous-ensemble d'arêtes d'un graphe qui connecte tous les sommets tout en minimisant le poids total des arêtes. Contrairement à d'autres algorithmes, tels que celui de Prim, qui construisent l'arbre de manière croissante à partir d'un sommet initial, Kruskal commence par trier toutes les arêtes du graphe en fonction de leurs poids. Ensuite, il ajoute progressivement les arêtes à l'arbre couvrant, en veillant à ce qu'aucun cycle ne soit formé.

5.3.1Étapes de l'Algorithme de Kruskal

1. ***Initialisation*** *:*

- *Créez un ensemble vide A pour stocker les arêtes de l'arbre couvrant minimal (ACM).*
- *Initialisez une structure de données de type union-find avec n ensembles disjoints,*
- *où n est le nombre de sommets dans le graphe G=(V,E).*
- *Chaque sommet $v \in V$ commence dans son propre ensemble, ce qui peut être représenté par la relation d'appartenance parent[v]=v.*

2. ***Tri des arêtes*** *:*

- *Récupérez toutes les arêtes e de E et triez-les par ordre croissant de poids w(e). Ce tri peut être effectué en utilisant un algorithme de tri efficace tel que le tri par fusion, qui a une complexité de O(E log E).*
- *La notation utilisée ici est $w: E \rightarrow R^+$ ou R^+ représente les poids positifs des arêtes.*

3. ***Ajout des arêtes à l'ACM*** *:*

Parcourez la liste triée des arêtes :

Pour chaque arête (u,v):

- *Vérifiez si u et v appartiennent à des composantes différentes en utilisant la structure de données union-find.*

Cela se fait en calculant :

$$\text{find-set}(u) \neq \text{find-set}(v)$$

Si cette condition est vraie, ajoutez l'arête (u,v) à l'ensemble A :

$$A = A \cup \{(u, v)\}$$

- *Unifiez les deux composantes en utilisant la fonction union : union(u,v*

4. ***Condition d'arrêt*** *:*

- *L'algorithme continue jusqu'à ce que l'ensemble A contienne exactement $n-1$ arêtes, où n est le nombre de sommets. Cette condition est basée sur le fait qu'un arbre couvrant minimal d'un graphe avec n sommets a exactement $n-1$ arêtes.*
- *L'arbre couvrant minimal résultant est un sous-ensemble $T \subseteq E$ qui relie tous les sommets et minimise le poids total W(T), donné par :*

$$W(T) = \sum_{(u,v) \in T} w(u, v)$$

5. ***Résultat*** *:*

Le résultat final est l'ensemble A contenant les arêtes de l'ACM et le poids total W(T)

5.3.2Exemple illustratif de l'Algorithme de Kruskal

Considérons le graphe suivant G avec les sommets V={A,B,C,D,E} et les arêtes E avec leurs poids :

Arête	*Poids*
(A, B)	4
(A, C)	1
(B, C)	2
(B, D)	5
(C, D)	8
(C, E)	10
(D, E)	3

Déroulement de de l'Algorithme

1. *Initialisation*

- **ACM** : A={}(ensemble d'arêtes de l'ACM).
- **Ensembles disjoints** : Créer des ensembles disjoints pour chaque sommet :

make-set(A),make-set(B),make-set(C),make-set(D)make-set(E)

2. *Tri des arêtes*

Les arêtes triées par poids (du plus léger au plus lourd) :

Arête	Poids
(A, C)	1
(B, C)	2
(D, E)	3
(A, B)	4
(B, D)	5
(C, D)	8
(C, E)	10

3. Ajout des arêtes à l'ACM

Vérifiez chaque arête triée et appliquez la méthode find-set pour déterminer si l'ajout de l'arête formera un cycle :

- Ajouter (A, C) :

Vérifiez si A et C sont dans la même composante :

find-set(A) ≠ find-set(C)

Ajoutez à l'ACM : A={(A,C)} (aucun cycle).

- Ajouter (B, C) :

Vérifiez si B et C sont dans la même composante :

find-set(B) ≠ find-set(C)

Ajoutez à l'ACM : A={(A,C),(B,C)} (aucun cycle).

- Ajouter (D, E) :

Vérifiez si D et E sont dans la même composante :

find-set(D) ≠ find-set(E)

Ajoutez à l'ACM : A={(A,C),(B,C),(D,E)} (aucun cycle).

- Ignorer (A, B) :

Vérifiez si A et B sont dans la même composante :

find-set(A) = find-set(B) (déjà dans la même composante).

- Ignorer (B, D) :

Vérifiez si B et D sont dans la même composante :

find-set(B) ≠ find-set(D)

Ajoutez à l'ACM : A={(A,C),(B,C),(D,E),(B,D)}(aucun cycle).

- Ignorer (C, D) :

Vérifiez si C et D sont dans la même composante :

find-set(C) = find-set(D) (déjà dans la même composante).

- Ignorer (C, E) :

Vérifiez si C et E sont dans la même composante :

find-set(C) = find-set(E) (déjà dans la même composante).

4. Condition d'arrêt

Nous avons maintenant 4 arêtes, et comme il y a 5 sommets, nous devons avoir $n-1=5-1=4$ arêtes dans l'ACM.

5. Résultat

L'arbre couvrant minimal est constitué des arêtes suivantes :

(A, C) : 1 , (B, C) : 2 , (D, E) : 3, (B, D) : 5

5.4 Algorithme de Prim

L'algorithme de Prim fonctionne en partant d'un sommet initial et en ajoutant progressivement les arêtes les moins coûteuses qui connectent un sommet de l'arbre à un sommet qui n'est pas encore dans l'arbre. L'algorithme s'arrête lorsque tous les sommets sont inclus dans l'ACM.

1. ***Initialisation*** *:*

Choix d'un sommet de départ *: Choisir un sommet s à partir duquel l'ACM sera construit.*

Création de l'ensemble des arêtes de l'ACM *: Initialiser un ensemble A pour stocker les arêtes de l'ACM, tel que A={}.*

Création de l'ensemble des sommets visités *: Initialiser un ensemble V contenant le sommet de départ, tel que V={s}.*

2. ***Sélection de l'arête minimale*** *:*

Répéter jusqu'à ce que tous les sommets soient inclus dans l'arbre :

Identification de l'arête de poids minimal *: Pour chaque sommet u dans V, examiner toutes les arêtes (u,v)adjacentes à u qui mènent à un sommet v qui n'est pas encore dans V.*

Si w(u,v) est le poids de l'arête (u,v), trouver l'arête (u,v) telle que w(u,v) soit minimal parmi toutes les arêtes qui connectent V à l'extérieur.

3. ***Ajout de l'arête à l'ACM*** *:*

Ajouter l'arête (u,v) à l'ensemble A.

Ajouter le sommet v à l'ensemble des sommets visités V.

4. ***Condition d'arrêt*** *:*

L'algorithme se termine lorsque tous les sommets du graphe sont inclus dans l'ensemble V

5.3.1 Preuve de correction de l'algorithme de Prim par l'absurde

L'objectif de la preuve est de montrer que l'arbre couvrant trouvé par l'algorithme de Prim est bien un arbre couvrant minimal. La méthode utilisée ici consiste à supposer que l'algorithme n'est pas correct, c'est-à-dire qu'il ne produit pas un arbre couvrant de poids minimal, puis à démontrer que cette hypothèse mène à une contradiction.

Hypothèse

Supposons que l'algorithme de Prim génère un arbre couvrant T qui n'est pas un arbre couvrant minimal, c'est-à-dire qu'il existe un autre arbre couvrant T* tel que :

$$w(T)>w(T^*)$$

où w(X) représente le poids total de l'arbre X.

Construction des arbres

1. Initialisation :

Soit V l'ensemble des sommets du graphe, et soit A l'ensemble d'arêtes de l'arbre T généré par l'algorithme de Prim.

2. Établissement de la contradiction :

Considérons l'ensemble des arêtes E qui sont ajoutées à T au fur et à mesure que l'algorithme progresse.

Soit e l'arête de T^* qui n'est pas incluse dans T mais qui relie deux sommets u et v (où u appartient à l'arbre T et v est un sommet non encore ajouté à T au moment de la construction).

Puisque l'algorithme de Prim choisit toujours l'arête de poids minimal reliant un sommet de T à un sommet en dehors de T, cela signifie que le poids de e ne peut pas être inférieur à celui de l'arête choisie à cette étape. On peut donc écrire :

$$w(e)\geq w(a)$$

où a est l'arête de poids minimal choisie par l'algorithme.

3. Construire une nouvelle solution :

Remplaçons l'arête a dans T par e. Cela nous donne un nouvel arbre T′, qui contient toutes les arêtes de T sauf a et inclut e.

Puisque T était connecté, T′ est également connecté et couvre tous les sommets de T.

4. Poids de la nouvelle solution :

Le poids de T′ est donné par :

$$w(T')=w(T)-w(a)+w(e)$$

5. Comparaison des poids : Comme nous avons supposé que $w(T)>w(T^*)$, et que $w(e)\geq w(a)$, il en résulte :

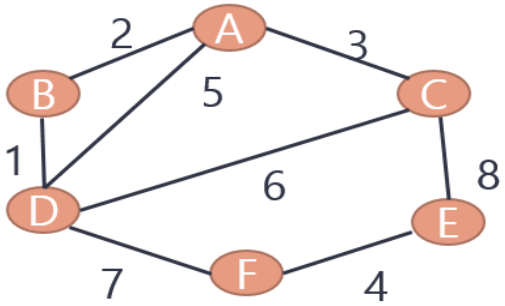
$$w(T')=w(T)-w(a)+w(e) \leq w(T)-w(a)+w(a)=w(T)$$

Par conséquent, nous avons w(T′)≤w(T) mais aussi w(T′)≥w(T*)w. Donc :

w(T)>w(T)ce qui est une contradiction !*

5.4.2 Exemple illustratif de l'algorithme de Prim

Appliquer l'algorithme de Prim pour trouver l'arbre couvrant minimal



	A	B	C	D	E	F
A		2 A	3 A	5 A		
B			3A	1 B		
D			3A			
C					8C	
E						4E

L'arbre couvrant minimal obtenu avec l'algorithme de Prim est constitué des arêtes : (A,B), (C,A), (D,B), (E,C), (F,E) de poids respectivement 2 + 3 +1 +8 + 4 =18

6. Problème du rendu de monnaie

Le problème du rendu de monnaie consiste à rendre un montant donné M à l'aide d'un ensemble de pièces de monnaie de différentes valeurs. L'objectif est de minimiser le nombre total de pièces utilisées pour atteindre ce montant. L'une des approches les plus efficaces pour résoudre ce problème, dans de nombreux cas pratiques, est l'approche gloutonne. Cette méthode se base sur la prise de décisions séquentielles, où à chaque étape, la pièce de la valeur la plus élevée est choisie, tant qu'elle ne dépasse pas le montant restant à rendre.

Énoncé formel :

On dispose d'un ensemble de pièces avec des valeurs $C=\{c_1,c_2,\ldots,c_n\}$, où $c_1<c_2<\cdots<c_n$ sont des entiers positifs. L'objectif est de rendre une somme S avec un nombre minimal de pièces issues de cet ensemble. Chaque pièce peut être utilisée autant de fois que nécessaire.

Le but est de déterminer le nombre minimal de pièces pour atteindre exactement la somme S.

1. ***Initialisation :***

➢ *On a une somme S à rendre avec un ensemble de pièces $C=\{c_1,c_2,\ldots,c_n\}$, triées en ordre décroissant $c_n \geq c_n-1 \geq \cdots \geq c_1$*

On *initialise une variable count pour comptabiliser le nombre de pièces utilisées.*

2. Sélection gloutonne :

➢ *On choisit la pièce de valeur ci la plus grande telle que $c_i \leq S$*
➢ *On détermine combien de fois cette pièce peut être utilisée sans dépasser S, soit $k= S/ c_i$*
➢ *On met à jour count avec count=count+k et on réduit la somme restante S de $k \times c_i$*

1. ***Répétition :***

Tant que $S \neq 0$, on recommence en sélectionnant la pièce la plus adaptée à la nouvelle somme.

4. Arrêt :

L'algorithme se termine quand S=0. Le nombre total de pièces utilisées est donné par la variable count.

Étapes de l'algorithme :Exemple de résolution

Nous disposons de pièces de valeurs 1, 5, et 10 unités, et devons rendre une somme de S=28.

Étape 1 : La somme à rendre est S=28.

➢ La pièce la plus grande inférieure ou égale à 28 est 10.
➢ On utilise 2 pièces de 10, ce qui laisse une somme restante de $28-2\times10=8$
➢ Le nombre total de pièces est de 2.

Étape 2 :

➢ La somme restante est S=8.
➢ La pièce la plus grande disponible est 5.
➢ On utilise 1 pièce de 5, il reste donc $8-5=3$.
➢ Le nombre total de pièces est maintenant de 3.

Étape 3 :

- La somme restante est S=3.
- On utilise 3 pièces de 1 (car ⌊3/1⌋=3.
- La somme est maintenant réduite à 0.
- Le nombre total de pièces est donc de 6.

Correctitude de la méthode gloutonne

La méthode gloutonne est efficace dans de nombreux cas, mais ne garantit pas toujours une solution optimale pour tous les ensembles de pièces. Par exemple, si nous aons des pièces de valeurs 1, 3 et 4 et un montant de 6, l'algorithme glouton choisirait 4 et 1 (total 2 pièces), alors qu'une solution optimale serait d'utiliser 3 et 3 (total 2 pièces).

7. Problème de la location du camion

Le problème de la location de camion consiste à sélectionner un ensemble de camions pour transporter une quantité donnée de marchandises tout en minimisant les coûts. Chaque camion a une capacité maximale de transport et un coût de location. Le but est de minimiser le coût total tout en assurant que l'ensemble des camions choisis puisse transporter la quantité totale de marchandises.

Ce type de problème est un exemple classique d'optimisation combinatoire qui peut être résolu à l'aide d'algorithmes gloutons ou d'autres approches comme la programmation dynamique, selon les caractéristiques des camions et du problème.

7.1 Formulation du Problème

Le but est de minimiser le coût total de location pour transporter exactement M unités de marchandises. Cela peut être exprimé comme une fonction objectif :

$$\text{Minimiser} \sum_{i=1}^{n} x_i \cdot cost_i$$

où x_i représente le nombre d'unités de marchandises transportées par le camion i, et $cost_i$ est le coût de location du camion i.

On doit respecter la contrainte suivante :

$$\sum_{i=1}^{n} x_i \cdot cap_i = M$$

c'est-à-dire que la somme des marchandises transportées par tous les camions doit être égale à M, la quantité totale de marchandise

Propriété de solution optimale

Le problème de location de camion satisfait la propriété dite de:

- Greedy choice property : À chaque étape, en choisissant le camion avec le plus bas coût par unité de marchandise, nous ne compromettons pas la possibilité d'obtenir une solution optimale globale.
- Optimal substructure : Un sous-problème du problème original (par exemple, transporter M−capi unités de marchandises après avoir choisi un camion c_i) est lui-même un problème de même nature, et donc la même approche gloutonne peut être appliquée de manière récursive ou itérative.

Stratégie gloutonne

L'idée principale de cet algorithme est de toujours choisir le camion avec le meilleur rapport coût/capacité (coût par unité de marchandise transportée) jusqu'à ce que tout le volume soit transporté

Algorithme glouton pour la location de camion

1. ***Entrée*** *:*

Une liste de camions C, chaque camion c_i *ayant :*

- *une capacité* cap_i *(quantité de marchandise qu'il peut transporter),*
- *un coût* $cost_i$

Une quantité totale de marchandises M à transporter.

2. ***Sortie*** *: Un sous-ensemble de camions S et leur nombre associé, qui minimise le coût total de transport.*

Etapes de l'algorithmes

1. *Calculer le* ***rapport coût/capacité*** $r_i = cost_i / cap_i$ *pour chaque camion.*

2. *Trier les camions C par ordre croissant de* r_i

3. *Initialiser une variable* $M_{restant} = M$ *(la marchandise restante à transporter).*

4. *Initialiser un sous-ensemble vide de camions sélectionnés S.*

5. *Pour chaque camion* c_i *dans l'ordre trié :*

Si $cap_i \leq M_{restant}$ *, ajouter le camion* c_i *à la solution S et mettre à jour* $M_{restant} = M_{restant} - cap_i$

Si capi > Mrestant , fractionner la capacité du camion c_i *pour transporter* $M_{restant}$*, ajouter cette fraction à S, et mettre* $M_{restant} = 0$

6. *Sortir de la boucle quand* $M_{restant} = 0$

7. *Retourner le sous-ensemble S des camions et le coût total correspondant.*

Exemple illustratif :

Supposons que vous avez trois camions avec les caractéristiques suivantes :

Camion 1 : Capacité = 10 tonnes, Coût = 100 unités

Camion 2 : Capacité = 20 tonnes, Coût = 180 unités

Camion 3 : Capacité = 30 tonnes, Coût = 270 unités

Vous devez transporter 50 tonnes.

Calcul des rapports r_i :

Camion 1 : r_1=10 unités/tonne

Camion 2 : r_2=9 unités/tonne

Camion 3 : r_3=9 unités/tonne

Tri des camions par r_i : C_2,C_3,C_1

Sélection :

On choisit le camion 2 pour transporter 20 tonnes (coût = 180).

Il reste 30 tonnes à transporter.

On choisit ensuite le camion 3 pour 30 tonnes (coût = 270).

Coût total = 180 + 270 = 450 unités.

8. Avantages et inconvénients des méthodes gloutonnes :

Les avantages :

1. Simplicité :

Les algorithmes gloutons sont généralement simples à comprendre et à implémenter. Ils fonctionnent en effectuant des choix locaux optimaux à chaque étape, ce qui rend leur structure algorithmique intuitive.

2. Efficacité temporelle :

En faisant des choix immédiats, les algorithmes gloutons peuvent être très rapides. Leur complexité est souvent inférieure à celle d'autres techniques comme la programmation dynamique ou les approches par backtracking. Par exemple, l'algorithme de Dijkstra pour le plus court chemin dans un graphe à poids positifs est très performant avec une complexité O((V + E) log V).

3. Solutions raisonnablement bonnes :

Pour de nombreux problèmes pratiques, les méthodes gloutonnes fournissent des solutions acceptables en un temps relativement court. Cela les rend utiles dans des situations où une solution rapide est plus importante qu'une solution exacte ou optimale.

4. Applications diversifiées :

Les algorithmes gloutons sont largement utilisés dans des domaines variés comme la planification, l'optimisation des ressources, les réseaux et les graphes (par exemple, Kruskal et Prim pour les arbres couvrants minimaux).

Inconvénients:

1. Absence de garantie d'optimalité :

Un des principaux inconvénients des méthodes gloutonnes est qu'elles ne garantissent pas toujours d'atteindre la solution optimale. Les choix locaux optimaux peuvent mener à des solutions globalement sous-optimales. Par exemple, pour le problème du sac à dos 0/1, les solutions gloutonnes peuvent ne pas être optimales.

2. Dépendance aux conditions spécifiques :

Les algorithmes gloutons ne fonctionnent de manière optimale que sous certaines conditions bien définies (comme la propriété de sous-structure optimale ou la propriété d'échange). S'il manque ces propriétés, l'algorithme glouton échoue souvent à trouver la solution optimale.

3. Impossibilité de réajustement :

Les décisions prises par un algorithme glouton sont irréversibles. Une fois qu'un choix est fait, il n'y a généralement pas de retour en arrière, ce qui limite la flexibilité. Si un mauvais choix est fait, il peut compromettre toute la solution.

4. Non-applicabilité pour certains types de problèmes :

Les méthodes gloutonnes ne sont pas adaptées aux problèmes complexes comme ceux comportant des cycles de poids négatifs (comme dans certains graphes), ou dans des problèmes de programmation linéaire où une optimisation globale est nécessaire.

Conclusion

Les algorithmes gloutons sont des outils précieux dans la boîte à outils de tout programmeur ou théoricien des algorithmes. Ils fournissent des solutions rapides et relativement bonnes dans des situations où l'efficacité temporelle est cruciale. Toutefois, leur applicabilité doit être soigneusement évaluée au cas par cas, en fonction des caractéristiques du problème. Les méthodes gloutonnes s'intègrent ainsi dans une famille plus large d'outils algorithmiques, comprenant des approches comme la programmation dynamique ou les algorithmes d'approximation, qui peuvent être plus appropriées lorsque les conditions d'optimalité locale ne sont pas satisfaites.