Algorithmes gloutons

Lycée Thiers

Présentation

- Un algorithme *glouton* (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand, ou goulu) est un algorithme qui suit le principe de réaliser, étape par étape, un choix optimum local, afin d'obtenir un résultat optimum global (Wikipedia).

Présentation

- Un algorithme *glouton* (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand, ou goulu) est un algorithme qui suit le principe de réaliser, étape par étape, un choix optimum local, afin d'obtenir un résultat optimum global (Wikipedia).
- Exemples classiques :

Présentation

- Un algorithme *glouton* (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand, ou goulu) est un algorithme qui suit le principe de réaliser, étape par étape, un choix optimum local, afin d'obtenir un résultat optimum global (Wikipedia).
- Exemples classiques :
  - Rendu de monnaie ;

Présentation

- Un algorithme *glouton* (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand, ou goulu) est un algorithme qui suit le principe de réaliser, étape par étape, un choix optimum local, afin d'obtenir un résultat optimum global (Wikipedia).
- Exemples classiques :
  - Rendu de monnaie ;
  - Coloration des sommets d'un graphe ;

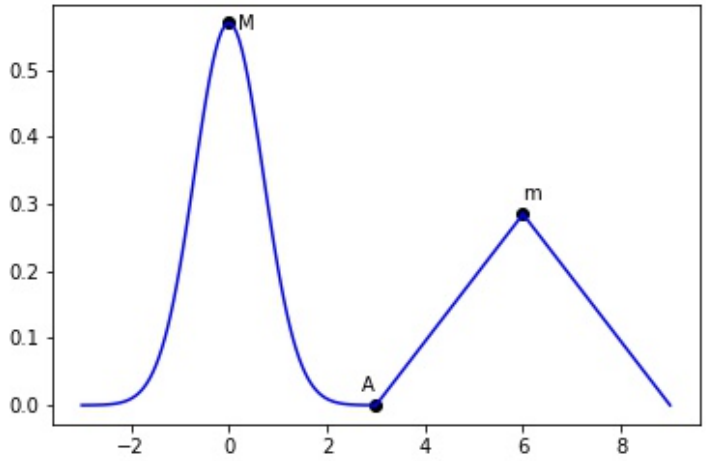
Présentation

- Un algorithme *glouton* (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand, ou goulu) est un algorithme qui suit le principe de réaliser, étape par étape, un choix optimum local, afin d'obtenir un résultat optimum global (Wikipedia).
- Exemples classiques :
  - Rendu de monnaie ;
  - Coloration des sommets d'un graphe ;
  - Algorithme de Dijkstra pour la recherche de PCC ;

Présentation

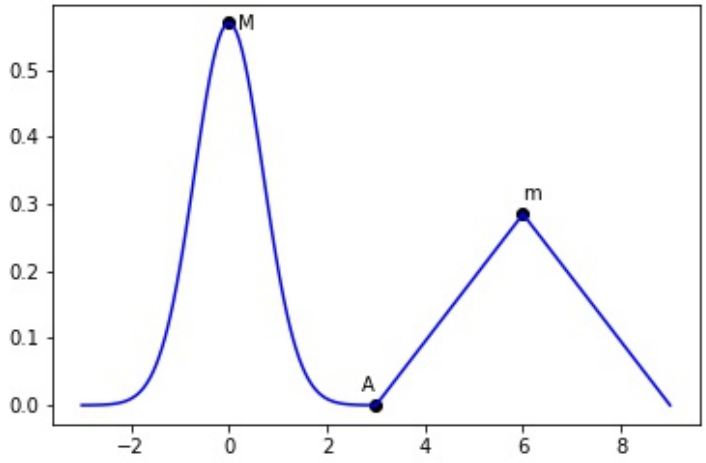
- Un algorithme *glouton* (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand, ou goulu) est un algorithme qui suit le principe de réaliser, étape par étape, un choix optimum local, afin d'obtenir un résultat optimum global (Wikipedia).
- Exemples classiques :
  - Rendu de monnaie ;
  - Coloration des sommets d'un graphe ;
  - Algorithme de Dijkstra pour la recherche de PCC ;
- Un algorithme glouton fournit le plus souvent une solution au problème. Dans les cas où il ne donne pas systématiquement la solution optimale, il est appelé une *heuristique* gloutonne.

Exemple d'heuristique gloutonne



- Un algorithme glouton peut retourner une solution sous-optimale : en partant du point A et en cherchant à monter selon la plus forte pente, un algorithme glouton trouvera le maximum local m, mais pas le maximum global M.

Exemple d'heuristique gloutonne



- Un algorithme glouton peut retourner une solution sous-optimale : en partant du point A et en cherchant à monter selon la plus forte pente, un algorithme glouton trouvera le maximum local m, mais pas le maximum global M.
- Il faut bien comprendre que même si elle ne fournit pas toujours de solution optimale, une stratégie gloutonne est souvent adoptée en raison de la simplicité de sa mise en œuvre.

Présentation

- Soit un ensemble C (pour « coins ») de n valeurs entières de billets et pièces de monnaies $v_1 < v_2 < \cdots < v_n$. Par exemple $C = \{1€,2€,5€,10€,20€,100€,200€\}$

Présentation

- Soit un ensemble C (pour « coins ») de n valeurs entières de billets et pièces de monnaies $v_1 < v_2 < \cdots < v_n$. Par exemple $C = \{1€,2€,5€,10€,20€,100€,200€\}$
- Le problème du *rendu de monnaie* consiste à déterminer le nombre minimal de billets et de pièces pour rendre une somme donnée.

Présentation

- Soit un ensemble C (pour « coins ») de n valeurs entières de billets et pièces de monnaies $v_1 < v_2 < \cdots < v_n$. Par exemple $C = \{1€,2€,5€,10€,20€,100€,200€\}$
- Le problème du *rendu de monnaie* consiste à déterminer le nombre minimal de billets et de pièces pour rendre une somme donnée.
- Par exemple, la somme de 49€ peut être rendue en utilisant 49 pièces de 1€, ou 2 billets de 20€, un billet de 5€ et deux pièces de 2€.

Présentation

- Soit un ensemble C (pour « coins ») de n valeurs entières de billets et pièces de monnaies $v_1 < v_2 < \cdots < v_n$. Par exemple $C = \{1€,2€,5€,10€,20€,100€,200€\}$
- Le problème du *rendu de monnaie* consiste à déterminer le nombre minimal de billets et de pièces pour rendre une somme donnée.
- Par exemple, la somme de 49€ peut être rendue en utilisant 49 pièces de 1€, ou 2 billets de 20€, un billet de 5€ et deux pièces de 2€.
- Donc 5 billets/pièces rendues VS 49. Ce nombre 5 est d'ailleurs le plus petit qu'on puisse trouver pour le système de pièces C.

Précisions

- Pour raison de concision, nous emploierons dans toute la suite le terme ≪ pièce ≫ au lieu de ≪ pièce ou billet ≫.

Précisions

- Pour raison de concision, nous emploierons dans toute la suite le terme ≪ pièce ≫ au lieu de ≪ pièce ou billet ≫.
- De plus nous supposons que le stock de chaque valeur de pièce est illimité, ce qui ne reflète que partiellement la réalité (dans un DAB, il y a un nombre fini de billets de 10,20,50 et 100€).

Précisions

- Pour raison de concision, nous emploierons dans toute la suite le terme ≪ pièce ≫ au lieu de ≪ pièce ou billet ≫.
- De plus nous supposons que le stock de chaque valeur de pièce est illimité, ce qui ne reflète que partiellement la réalité (dans un DAB, il y a un nombre fini de billets de 10,20,50 et 100€).
- La solution calculée par l'algorithme que nous présentons et donc une solution théorique qui ne tient pas compte de la réalité du stock.

Stratégie

- On choisit d'abord les pièces qui permettent de rendre la plus grande valeur possible sur la somme à rendre. Dans l'exemple des 49€, il s'agit de deux billets de 20€.

Stratégie

- On choisit d'abord les pièces qui permettent de rendre la plus grande valeur possible sur la somme à rendre. Dans l'exemple des 49€, il s'agit de deux billets de 20€.
- Il reste alors à rendre 9€. On choisit la plus grande valeur de pièce plus petite que 9, soit 5€. On rend donc un billet de 5 (et pas 2 car $2 \times 5 > 9$).

Stratégie

- On choisit d'abord les pièces qui permettent de rendre la plus grande valeur possible sur la somme à rendre. Dans l'exemple des 49€, il s'agit de deux billets de 20€.
- Il reste alors à rendre 9€. On choisit la plus grande valeur de pièce plus petite que 9, soit 5€. On rend donc un billet de 5 (et pas 2 car $2 \times 5 > 9$).
- Enfin la plus grande valeur de pièce plus petite que les 4€ à rendre est 2€. On peut en rendre deux, ce qui ramène la somme à rendre à 0€. On s'arrête donc là.

Code (version impérative)

```
1 def greedy_change(coins, v):
2     #coins : tab des valeurs de pièces par ordre croi
3     n = len(coins)
4     # le tableau de la somme rendu :
5     # exemple 0 pièce de 1, 3 de 2, 1 de 5 etc :
6     change = [0] * n
7     cur = v # somme restant à rembourser
8     i = n-1 # indice de valeur de pièce courante
9     while cur > 0:
10        c = coins[i]
11        if cur < c:
12            i-=1 # changer de valeur de pièce
13        else:
14            change[i]+=1
15            cur = cur -c
16    return change
```

Paramètres et variables

- Paramètres :

Paramètres et variables

- Paramètres :
  - `coins` : tableau des valeurs de pièces.

Paramètres et variables

- Paramètres :
  - `coins` : tableau des valeurs de pièces.
  - `v` : valeur à rembouser

Paramètres et variables

- Paramètres :
  - `coins` : tableau des valeurs de pièces.
  - `v` : valeur à rembouser
- Variables locales :

Paramètres et variables

- Paramètres :
  - `coins` : tableau des valeurs de pièces.
  - `v` : valeur à rembouser
- Variables locales :
  - `i` : numéro de valeur courante de pièce ;

Paramètres et variables

- Paramètres :
  - `coins` : tableau des valeurs de pièces.
  - `v` : valeur à rembouser
- Variables locales :
  - `i` : numéro de valeur courante de pièce ;
  - `change[i]` : nb de pièces de la valeur courante ;

Paramètres et variables

- Paramètres :
  - `coins` : tableau des valeurs de pièces.
  - `v` : valeur à rembouser
- Variables locales :
  - `i` : numéro de valeur courante de pièce ;
  - `change[i]` : nb de pièces de la valeur courante ;
  - `cur` : somme restant à rendre.

Correction du programme

- Un variant de boucle est `cur + i`. Terminaison OK.

Correction du programme

- Un variant de boucle est `cur + i`. Terminaison OK.
- La condition `coins[0]==1` assure la correction (principe : apcr, on peut rendre autant de pièces de 1€ que la somme restante).

Correction du programme

- Un variant de boucle est `cur + i`. Terminaison OK.
- La condition `coins[0]==1` assure la correction (principe : apcr, on peut rendre autant de pièces de 1€ que la somme restante).
- Le fait que `coins[0]==1`, assure aussi que `i` reste positif et donc l'accès valide au tableau `coins`.

Optimalité

Proposition

Si `coins` *décrit le système monétaire de la zone euro, alors le programme* `greedy_change` *cacule un rendu de monnaie avec le nombre minimal d'éléments.*

Remarque

Un système de pièces qui, tel celui de la zone euro, permet un rendu optimal est qualifié de *canonique*.

Optimalité (preuve)

Certaines combinaison de pièces ne peuvent se trouver dans une solution optimale :

contrainte 1 Une pièce de valeur **val** = 1,5,10,50 ou 100 n'est jamais utilisé deux fois dans une solution optimale. En effet, pour chacune de ces valeurs il est plus avantageux de rendre une pièce de valeur **2val** plutôt que deux de valeur **val**.

Optimalité (preuve)

Certaines combinaison de pièces ne peuvent se trouver dans une solution optimale :

contrainte 1 Une pièce de valeur **val** = 1,5,10,50 ou 100 n'est jamais utilisé deux fois dans une solution optimale.

contrainte 2 Une pièce de valeur 2 ou 20 n'est jamais utilisée 3 fois dans une solution optimale. En effet 3 pièces de valeur 2 sont avantageusement remplacée par par une pièce de valeur 1 et une de 5 ; 3 pièces de valeur 20 sont remplacées par une 10 et une de 50.

Optimalité (preuve)

Certaines combinaison de pièces ne peuvent se trouver dans une solution optimale :

contrainte 1 Une pièce de valeur **val** = 1,5,10,50 ou 100 n'est jamais utilisé deux fois dans une solution optimale.

contrainte 2 Une pièce de valeur 2 ou 20 n'est jamais utilisée 3 fois dans une solution optimale.

contrainte 3 Une pièce de valeur 1 n'accompagne jamais deux pièces de valeur 2 : on pourrait remplacer l'ensemble par une pièce de 5. Une pièce de valeur 10 n'accompagne jamais deux pièces de valeur 20 : on pourrait remplacer l'ensemble par une pièce de 50.

Optimalité (preuve)

- Écrivons un petit programme qui prend en compte les contraintes précédentes et faisons le tourner pour explorer exhaustivement toutes les combinaisons possibles de pièces de moins de 200 euros (voir TP).

Optimalité (preuve)

- Écrivons un petit programme qui prend en compte les contraintes précédentes et faisons le tourner pour explorer exhaustivement toutes les combinaisons possibles de pièces de moins de 200 euros (voir TP).
- On constate que la plus grande somme possible remboursable avec ces pièces est

$$2 \times 2 + 5 + 2 \times 20 + 50 + 100 = 199$$

Optimalité (preuve)

- Écrivons un petit programme qui prend en compte les contraintes précédentes et faisons le tourner pour explorer exhaustivement toutes les combinaisons possibles de pièces de moins de 200 euros (voir TP).
- On constate que la plus grande somme possible remboursable avec ces pièces est

$$2 \times 2 + 5 + 2 \times 20 + 50 + 100 = 199$$

- Ainsi, la solution optimale ne pourra JAMAIS rembourser plus de 199 € avec des pièces de moins de 200 €. Dit autrement, la solution optimale doit rembourser toute somme $S > 200$€ avec le maximum possible de pièces de 200€ (qui est en fait $k = S/200$).

Optimalité (preuve)

- Écrivons un petit programme qui prend en compte les contraintes précédentes et faisons le tourner pour explorer exhaustivement toutes les combinaisons possibles de pièces de moins de 200 euros (voir TP).
- On constate que la plus grande somme possible remboursable avec ces pièces est

$$2 \times 2 + 5 + 2 \times 20 + 50 + 100 = 199$$

- Ainsi, la solution optimale ne pourra JAMAIS rembourser plus de 199 € avec des pièces de moins de 200 €. Dit autrement, la solution optimale doit rembourser toute somme $S > 200$€ avec le maximum possible de pièces de 200€ (qui est en fait $k = S/200$).
- Or, notre algorithme calcule exactement k.

Optimalité (preuve)

- Pour une somme inférieure à 199 euros reprenons notre petit programme et faison le tourner pour trouver le nombre maximum de pièces de moins de 100 euros.

Optimalité (preuve)

- Pour une somme inférieure à 199 euros reprenons notre petit programme et faison le tourner pour trouver le nombre maximum de pièces de moins de 100 euros.
- On trouve alors que la solution optimale ne peut rembourser qu'une somme de 99 euros avec ces pièces. Pour rembourser une somme entre entre 100 et 199€, il faut un billet de 100.

Optimalité (preuve)

- Pour une somme inférieure à 199 euros reprenons notre petit programme et faison le tourner pour trouver le nombre maximum de pièces de moins de 100 euros.
- On trouve alors que la solution optimale ne peut rembourser qu'une somme de 99 euros avec ces pièces. Pour rembourser une somme entre entre 100 et 199€, il faut un billet de 100.
- C'est exactement la quantité que trouve notre programme dans ce cas là !

Optimalité (preuve)

- Pour une somme inférieure à 199 euros reprenons notre petit programme et faison le tourner pour trouver le nombre maximum de pièces de moins de 100 euros.
- On trouve alors que la solution optimale ne peut rembourser qu'une somme de 99 euros avec ces pièces. Pour rembourser une somme entre entre 100 et 199€, il faut un billet de 100.
- C'est exactement la quantité que trouve notre programme dans ce cas là !
- etc.