Algorithmes gloutons vs Force brute !

Objectif : L'objectif de ce tp est de rendre l'élève capable :

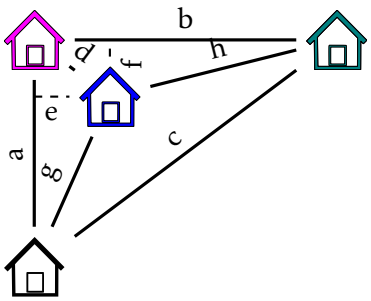
- d'écrire en langage **Python** des algorithmes gloutons
- de chercher la solution par force brute
- comparer les vitesses de convergence

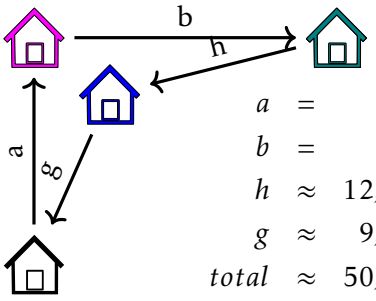
1 Voyageur de commerce

1.1 Présentation

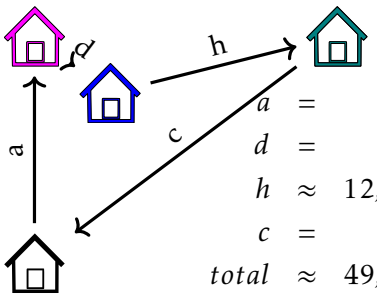
Enoncé: Problème du voyageur de commerce

Trouver le plus court chemin pour qu'un voyageur de commerce puisse passer exactement une fois chez ses clients et rentrer chez lui.





$a = 12$ km
$b = 16$ km
$h \approx 12{,}36$ km
$g \approx 9{,}84$ km
$total \approx 50{,}22$ km



$a = 12$ km
$d = 5$ km
$h \approx 12{,}36$ km
$c = 20$ km
$total \approx 49{,}34$ km

Principe :

On part du part du premier point, on calcule toutes les distances vers les autres points. On prend le plus proche et on recommence en calculant les distances vers les points encore disponibles, jusqu'à ce qu'il n'y ait plus de points disponibles.

Q - 1 : *Écrire une fonction* `distance(P1, P2)` *prenant en argument deux tuples des coordonnées cartésiennes des points P1 et P2 et retournant la distance entre ces points.*

Q - 2 : *Affecter à une liste* `maisons`*, les tuples formées des coordonnées des 4 maisons de l'exemple ci-dessus. On prendra comme premier élément* `0, 0` *qui correspond aux coordonnées de la maison noire.*

1.2 Résolution par algorithme glouton

Q - 3 : *Écrire une fonction* `trajet(L, dep)` *prenant en argument une liste* `L` *de tuples contenant les coordonnées cartésiennes de points ainsi que* `dep` *l'indice du point de départ et renvoyant une liste des indices de* `L` *permettant, à partir du premier tuple, d'obtenir la plus petite longueur de courbe passant par tous les points une et une seul fois avant de revenir au point initial.*

Algorithm 1 Voyageur de commerce

```
    entrée: L une liste de coordonnées cartésiennes
    résultat: une liste d'indices correspondant au parcours des éléments de L minimisant le chemin
    trajet(L, dep)
 1: n, pos ←taille(L), dep
 2: dispo ←[Vrai]*n
 3: parcours ←[dep]
 4: pour i = 1 à n-1 faire
 5:     j ←0
 6:     tant que non dispo[j] faire
 7:         j ←j + 1
 8:     fin tant que
 9:     d_min ←distance(L[pos], L[j])
10:     plus_proche ←j
11:     pour k = j+1 à n faire
12:         si dispo[k] alors
13:             d ←distance(L[pos], L[k])
14:             si d_min > d alors
15:                 d_min ←d
16:                 plus_proche ←k
17:             fin si
18:         fin si
19:     fin pour
20:     pos ←plus_proche
21:     dispo[pos] ←Faux
22:     parcours.append(pos)
23: fin pour
24: renvoi: parcours
```

Q - 4 : *Exécuter la fonction* `trajet` *avec la liste* `maison` *et changer le point de départ à chaque exécution.*

1.3 Force brute

Q - 5 : *Déterminer par force brute, le plus court chemin.*

Q - 6 : *Comparer le temps de calcul entre l'algorithme glouton et la méthode par force brute.*

Q - 7 : *Tirer « 10 maisons » de façon aléatoires dans un rectangle de largeur 16 et de hauteur 10.*

Q - 8 : *Comparer le temps de calcul entre l'algorithme glouton et la méthode par force brute.*

2 Autres exemples

2.1 Problème du sac à dos

Enoncé: Problème du sac à dos

Comment choisir quels objets placer dans un sac à dos limité au niveau du poids total ? Chaque objet a une masse et une valeur données. Comment emporter la plus grande quantité de valeurs possible dans le sac ?

Principe :

Trier les objets en fonction du critère de sélection : par ordre décroissant pour la valeur et le ratio valeur/-masse ; par ordre croissant pour la masse. Parcourir la liste des objets triés et prendre l'objet si la masse accumulée en le prenant ne dépasse pas la masse totale autorisée.

Q - 9 : *Faire de même avec le problème du sac à dos avec une stratégie d'optimisation de la valeur.*

Q - 10 : *Faire de même avec le problème du sac à dos avec une stratégie d'optimisation du ratio valeur/masse.*

Algorithm 2 Sac à dos

entrée: L une liste de 3-uplets triée par ordre décroissant sur une des colonnes (valeur ou valeur/masse) et m_{max} un flottant représentant la masse maximale admissible
résultat: *choix*, *val*, *m* avec *choix* la liste des objets choisis, *val* la valeur totale emportée et *m* la masse totale dans le sac
selection(L, m_{max})

```
1:  choix ←[]
2:  val, m, i ←0, 0, 0
3:  tant que i < taille(L) et m ≤ m_max faire
4:      v_i, invm_i, ratio_i ←L[i]
5:      si m+ 1/invm_i ≤ m_max alors
6:          choix.append(L[i])
7:          val ←val + v_i
8:          m ←m + 1/invm_i
9:      fin si
10:     i ←i + 1
11: fin tant que
12: renvoi: choix, val, m
```

2.2 Problème du rendu de la monnaie

Enoncé: Rendu de la monnaie

> *Étant donnée une monnaie avec sa déclinaison en pièces et billets de différentes valeurs, l'objectif du banquier pressé est de donner au client la somme désirée en prenant le moins d'éléments.*

Principe :

> *Il s'agit d'utiliser en premier lieu les plus gros supports puis compléter avec les plus petits.*

Q - 11 : *Programmer l'algorithme glouton du rendu de la monnaie en euros avec un porte feuille infini.*

Algorithm 3 Rendu de la monnaie

entrée: L une liste des valeurs des différents supports de la monnaie, triés par ordre décroissant et *som* le montant à rendre
résultat: *support* une liste des nombres de supports utilisés par valeur
merci(L, som)

```
1: i, support ←0, []
2: tant que i < taille(L) faire
3:     ni ←entier(som / L[i])
4:     som ←som - ni*L[i]
5:     support.append(ni)
6:     i ←i + 1
7: fin tant que
8: renvoi: support
```