Chapter 1

Algorithmes Gloutons

1.1 Approche et définition

En programmation, pour résoudre un problème, diverses techniques sont possibles. En première approche, et considérant la puissance de nos machines qui n'a de cesse de croître, on peut penser que calculer toutes les solutions pour choisir la meilleure est effectivement le meilleur choix. Cette technique dite par *force brute*, c'est-à-dire en testant tous les cas possibles peut-être utilisée mais ce type de résolution peut aussi présenter un problème d'efficacité voire de mise en oeuvre.

1.1.1 Le voyageur

Imaginons une personne visitant le département du Pas-de-Calais, celle-ci part de Lens où elle visitera sans doute le Louvre, la zone naturelle des terrils 11/19. Puis on lui conseille de voir le grand site national des deux caps, Lille, Arras et la baie de Canche au Touquet pour voir la Manche et ses phoques. Elle reviendra à Lens la dernière journée de son séjour pour voir un match au stade Bollaert. Comment passer son temps à visiter plutôt que sur la route ? On décide de calculer toutes les possibilités de circuits partant de Lens et y revenant.

1. Construire un arbre représentant tous les circuits possibles. Combien en dénombre-t-on ?
2. On décide d'effectuer le trajet en se rendant vers la ville immédiatement la plus proche à chaque étape. Donner le parcours correspondant.

	Lens	2 Caps	Arras	Le Touquet	Lille
Lens		111	18	104	30
2 Caps	111		118	51	116
Arras	18	118		95	48
Le Touquet	104	51	95		124
Lille	30	116	48	124	

Figure 1.1: Distance entre les étapes

3. La *figure 1.2* présente un tableau regroupant le calcul de la longueur des circuits *différents*. Le parcours suivi est-il le plus court ?

   (à partir de `https://www.geoportail.gouv.fr/`).

4. Nous n'avons du gérer que 4 villes hors la ville de départ.

   Combien y aurait-il de trajets si nous avions eu 10 villes ? 13 villes? n villes?

2 Caps-Arras-Le Touquet-Lille	=111+118+95+124+104
2 Caps-Arras-Lille-Le Touquet	505
2 Caps-Le Touquet-Arras-Lille	335
2 Caps-Le Touquet-Lille-Arras	352
2 Caps-Lille-Arras-Le Touquet	474
2 Caps-Lille-Le Touquet-Arras	464
Arras-2 Caps-Le Touquet-Lille	341
Arras-2 Caps-Lille-Le Touquet	480
Arras-Le Touquet-2 Caps-Lille	310
Arras-Lille-2 Caps-Le Touquet	337
Le Touquet-Arras-2 Caps-Lille	463
Le Touquet-2 Caps-Arras-Lille	351

Figure 1.2:

Conclusion Il devient clair que nous ne pouvons utiliser la technique dite de la force brute. La complexité de l'algorithme par force brute est proportionnelle à $n!$, au delà d'une douzaine de valeurs, il devient déraisonnable de l'utiliser. Pour vous donner une idée, le tableau suivant indique le nombre raisonnable d'opérations envisageables pour un tableau de longueur n selon la complexité de l'algorithme:

Nombre d'opération	n max
$log(n)$	$10^{300\ 000\ 000}$
$\sqrt{n}$	10^{18}
n	10^9
n^2	31600
n^3	1000
2^n	30
$n!$	12

La technique glouton (*greedy* en anglais) propose une solution à ce problème : c'est celle utilisée pour le parcours de la question 2.

1.1.2 Algorithme glouton

Un algorithme glouton est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local dans l'espoir d'obtenir un résultat optimum global.

On ne remet donc jamais en cause un choix qui a été fait. L'autre caractéristique des algorithmes gloutons est que lorsqu'un choix est fait, on tente de r**ésoudre un problème plus petit.** Le principal avantage des algorithmes gloutons est leur facilité de mise en œuvre. En outre, dans certaines situations dites canoniques, il arrive qu'ils renvoient non pas un optimum mais l'optimum d'un problème, cela donnera finalement la meilleure solution. On parlera d'algorithmes gloutons exacts. Dans d'autres, non, on parlera d'heuristiques gloutonnes.

Heuristique Rappelons qu' en optimisation combinatoire, théorie des graphes et théorie de la complexité, une heuristique est un algorithme qui fournit rapidement (en temps polynomial) une solution réalisable, mais pas nécessairement optimale, pour un problème d'optimisation difficile. Une **heuristique**, ou **méthode approximative**, est donc le contraire d'un algorithme exact qui trouve une solution optimale pour un problème donné. L'usage d'une heuristique est pertinente pour calculer une solution approchée d'un problème et ainsi accélérer le processus de résolution exacte.

1.2 Exercices

1.2.1 Rendu de monnaie

Donner une somme avec le moins de pièces possibles. On veut donner la somme de 14 euros dans les systèmes de pièce suivant:

$$s_1=\{1;\ 2;\ 5;\ 10\}\ ;\ s_2=\{1;\ 2;\ 7;\ 10\ \}\ ;\ s_3=\{3;\ 5;7;10\ \}$$

On commence par ordonner les pièces dans l'ordre décroissant: il suffira de choisir les pièces dans l'ordre pour faire un choix optimum local. Compléter les tableaux avec ce choix optimum local.

Système 1

Pièce	nombre
10	
5	
2	
1	

Système 2

Pièce	nombre
10	
7	
2	
1	

Système 3

Pièce	nombre
10	
7	
5	
3	

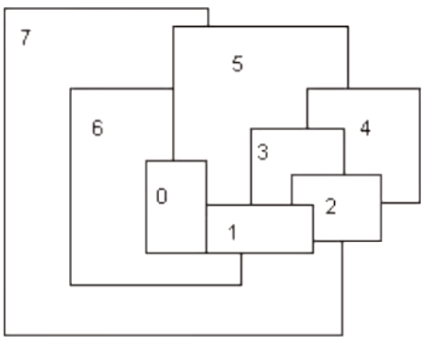
1. Comparer les résultats et commenter.
2. Ecrire en pseudo-code un algorithme de rendu de monnaie.
3. Implémenter une fonction `rendu_monnaie(m, systeme)` et la tester pour le système de pièce européen (le billet de plus forte valeur est 500) et pour un montant m donné.
4. Modifier le code de la fonction `rendu_monnaie` pour qu'elle renvoie un dictionnaire où les clés sont les valeurs des pièces rendues et les valeurs le nombre de pièces à rendre.

Exemple

```
>>>rendu_monnaie(14, [10, 5, 2, 1])
{1:0, 2:2, 5:0, 10:1}
```

1.2.2 Coloriage de carte

On veut colorier une carte en s'assurant que deux zones limitrophes n'ont pas la même couleur avec le minimum de couleurs possible. On choisit de trier les zones par nombres de zones limitrophes, par ordre décroissant. utilisez pour cela le tableau ci-dessous. La méthode gloutonne est le choix de colorier la zone ayant le plus d'adjacent, puis la zone ayant le plus d'adjacent non limitrophe à cette zone et ainsi de suite.



Zone	Limitrophe
0	
1	
2	
3	
4	
5	
6	
7	

Effectuer le coloriage de cette carte avec cette méthode.

1.2.3 Ranger ses affaires

On dispose de n objets $x_1, x_2, ...x_n$ ayant chacun une valeur v_i et une masse m_i. On possède un sac à dos dans lequel on ne peut mettre qu'une masse maximale M. On veut remplir le sac de façon à ce que la valeur des objets emportés soit maximale.

Optimiser le rangement On prend M=10 kg. Il s'agit de choisir les objets à emporter dans le sac afin maximiser la valeur totale tout en respectant la contrainte du poids maximal. C'est un problème d'optimisation avec contrainte.

Considérons les objets suivants et un sac de capacité maximale 10 kg. Quels objets faut-il prendre ?

Objet	A	B	C	D	E	F
masse en kg	7	6	4	3	2	1
valeur en euros	9100	7200	4800	2700	2600	200

Il y a plusieurs choix possibles :

Stratégie 1 : prendre toujours l'objet de plus grande valeur n'excédant pas la capacité restante (il faut trier préalablement par valeur décroissante)

Stratégie 2 : prendre toujours l'objet de plus faible masse (il faut trier préalablement par masse croissante)

Stratégie 3 : prendre toujours l'objet de plus grand rapport $\frac{valeur}{masse}$ n'excédant pas la capacité restante (il faut trier préalablement en suivant ce rapport de façon décroissante).

1. Tester ces trois stratégies et donner la valeur emportée dans les trois cas.
2. On dispose d'une liste d'objets de masses m = [9, 10, 12, 14, 11, 5, 7, 5, 6, 2] ainsi que de leurs valeurs associées v = [10, 8, 7, 7, 5, 4, 3, 2, 2, 1].

   Par exemple, le premier objet d'indice 0 a pour masse 9 kg et pour valeur 10 euros. Programmer un algorithme pour le *knapsack problem*. Tester le programme pour une masse maximale de 22 kg.

Bien sûr, pour manipuler plus finement les objets, il faudrait changer les structures de données utilisées en utilisant une seule liste, un dictionnaire ou de la programmation objet pour représenter l'ensemble des objets considérés.

Conclusion La **stratégie gloutonne** ne donne pas forcément un résultat optimal. On peut alors se demander s'il n'est pas possible de trouver la meilleure solution, à coup sûr, pour résoudre un problème d'optimisation. Une telle approche existe, il s'agit de la stratégie de **force brute** (ou énumérative) qui consiste à passer en revue toutes les options possibles et retenir la meilleure.

Pourquoi n'utilise-t-on pas toujours la force brute ?

Sur le problème du sac à dos, chaque objet est pris ou pas et avec 3 objets, il y a donc 2^3 combinaisons d'objets possibles, c'est-à-dire 8, ce qui est tout à fait acceptable. De manière générale, avec n objets, il y aurait 2^n combinaisons à énumérer et tester. On obtient une complexité dite *exponentielle* et c'est là le problème : avec 80 objets, on obtient 2^{80} combinaisons à tester, c'est-à-dire environ 10^{24} combinaisons, soit de l'ordre de grandeur du nombre d'étoiles dans l'Univers observable, ou de gouttes d'eau dans la mer, ou du nombre de grains de sables au Sahara... (*référence* : `https://fr.wikipedia.org/wiki/Ordres_de_grandeur_de_nombres`).

La stratégie **force brute** est donc inapplicable si trop d'objets sont en jeu. Il en est de même pour les autres problèmes d'optimisation dès que la taille des données est trop importante.

1.3 Un TP: retour sur le problème du voyageur

Le problème du voyageur de commerce - Traveling Salesman Problem TSP -, étudié depuis le 19e siècle, est l'un des plus connus dans le domaine de la recherche opérationnelle. William Rowan Hamilton a posé pour la première fois ce problème sous forme de jeu dès 1859.

1.3.1 Enoncé

Le problème du TSP sous sa forme la plus classique est le suivant :

Un voyageur de commerce doit visiter une et une seule fois un nombre fini de villes et revenir à son point d'origine. Trouvez l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur .

Ce problème d'optimisation combinatoire appartient à la classe des problèmes NP-Complets.

Les domaines d'application sont nombreux : problèmes de logistique, de transport aussi bien de marchandises que de personnes, et plus largement toutes sortes de problèmes d'ordonnancement. Certains problèmes rencontrés dans l'industrie se modélisent sous la forme d'un problème de voyageur de commerce, comme l'optimisation de trajectoires de machines outils : comment percer plusieurs points sur une carte électronique le plus vite possible ?

Pour un ensemble de n points, il existe au total $n!$ chemins possibles. Le point de départ ne changeant pas la longueur du chemin, on peut choisir celui-ci de façon arbitraire, on a ainsi $(n-1)!$ chemins différents. Enfin, chaque chemin pouvant être parcouru dans deux sens et les deux possibilités ayant la même longueur, on peut diviser ce nombre par deux. Par exemple, si on nomme les points, a, b, c, d, les chemins $abcd, bcda, cdab, dabc, adcb, dcba, cbad, badc$ ont tous la même longueur, seul le point de départ et le sens de parcours change. On a donc $\frac{1}{2}(n-1)$ chemins candidatsà considérer. Par exemple, pour 71 villes, le nombre de chemins candidats est supérieur à 5×10^{80} qui est environ le nombre d'atomes dans l'univers connu.

Voir Le voyageur de commerce (Wikipedia).

Figure 1.3: Le voyageur

1.3.2 Heuristique gloutonne

L'objectif de ce TP est de réaliser un algorithme glouton pour résoudre le TSP.

Pour cela vous avez à votre disposition en page d'accueil *Algorithmes gloutons* :

- un jeu de données *exemple.txt* contenant les coordonnées de différentes villes à raison d'une par ligne sous la forme nom_de_la_ville latitude longitude, vous pouvez bien sur l'étendre ou en générer un nouveau avec vos propres villes.

  Exemple `Annecy 6,082499981 45,8782196 Auxerre 3,537309885 47,76720047 Bastia 9,434300423 42,66175842`

- Un fichier `TSP_biblio.py` contenant un ensemble de fonctions permettant la lecture des données et la visualisation d'un tour réalisé par le voyageur (ici pour le moment dans l'ordre d'apparition). Voici les principales fonctions :

```
def get_tour_fichier(f):
    """
    Lit le fichier de villes: format ville,latitude,longitude
    Renvoie un tour contenant les villes dans l ordre du fichier
    : param f: nom de fichier
    : return : (list)
    """

def trac(tour) :
    """
    Trace la tournée realisée
    : param tour: liste de ville à parcourir
    """


def distance (tour, i, j) :
    """
    Dist.euclidienne entre deux villes i et j
    : param tour: sequence de ville
    : param i: numero de la ville de départ
    :param j: numero de la ville d arrivee
    : return: float
    CU: i et j dans le tour
    """

def longueur_tour (tour) :
    """
    Longueur d'une tournée ville de départ -> à la ville de départ
    : param tour: tournee de ville
    n villes = n segments
    : return: float distance totale
    """
```

Afin de créer l'algorithme glouton pour résoudre le problème du TSP, nous allons réaliser certaines étapes.

1. Définir l'heuristique de choix de la solution optimale locale
2. Réaliser un programme Python utilisant les fonctions définies pour la lecture et l'affichage permettant de mettre en œuvre l'heuristique. Pour cela vous pouvez :
   - réaliser une fonction qui génère une matrice qui stocke les distances 2 à 2 entre toutes les villes afin de ne faire le calcul de distance qu'une seule fois.
   - réaliser une fonction qui retourne l'indice de la ville la plus proche étant donnée une ville, une liste de ville sous forme d'indice, une matrice de distance.
   - réaliser l'heuristique gloutonne donnant le tour parcouru par le voyageur de commerce à partir d'une ville donnée en paramètre, la liste des villes et la matrice de distance ville à ville (on passera par un système d'indice).

1.4 Références

Peertube Académie de Bordeaux : algorithme gloutons.