Algorithmes gloutons

1 Problème de rendu de monnaie (1)

Imaginons que nous devons rendre 43 euros et que nous avons à notre disposition des pièces de 1,2,5,10 euros **avec la contrainte supplémentaire de rendre le moins de pièces possible (problème d'optimisation)** (pour simplifier il n'y a pas de billets mais que des pièces)
Une façon courante de procéder est :

1. Combien de fois **la plus grande valeur à notre disposition**, c'est à dire 10 y-a-t-il dans 43 ?
   $43 = 4 \times 10 + 3$
   Je vais donc rendre au moins 4 pièces de 10 euros.
   Après avoir divisé 43 par 10,j'exclus définitivement la valeur 10 des valeurs disponibles et je recommence le travail de division du reste c'est à dire 3 avec la plus grande des valeurs restantes c'est à dire 5
2. Combien de fois **la plus grande valeur à notre disposition**, c'est à dire 5 y-a-t-il dans 3 ?
   $3 = 0 \times 5 + 3$
   On n'utilisera donc pas de pièces de 5 euros et on exclut la valeur 5 des diviseurs
3. Combien de fois **la plus grande valeur à notre disposition**, c'est à dire 2 y-a-t-il dans 3 ?
   $3 = 1 \times 2 + 1$
   Donc pour l'instant on utilisera 4 pièces de 10 et une pièce de 2 on exclut la pièce de 2 et on continue
4. Combien de fois **la plus grande valeur à notre disposition**, c'est à dire 1 y-a-t-il dans 1 ?
   $1 = 1 \times 1$
   On rend donc 4 pièces de 10 et une pièce de 2 et une pièce de 1 euros

Problème de rendu de monnaie (2) : Imaginons que nous devons toujours rendre 43 euros et que cette fois ci nous avons à notre disposition des pièces de 1,3,7,21,30 euros avec la contrainte supplémentaire de rendre le moins de pièces possible
Procédons comme précédemment :

1. Combien de fois **la plus grande valeur à notre disposition**, c'est à dire 30 y-a-t-il dans 43 ?
   $43 = 1 \times 30 + 13$
   Après avoir divisé 43 par 30,j'exclus définitivement la valeur 30 des valeurs disponibles et je recommence le travail de division du reste c'est à dire 13 avec la plus grande des valeurs restantes c'est à dire 21
2. Combien de fois **la plus grande valeur à notre disposition**, c'est à dire 21 y-a-t-il dans 13 ?
   $13 = 0 \times 21 + 13$

On n'utilisera donc pas de pièces de 21 euros et on exclut la valeur 21 des diviseurs

3. Combien de fois **la plus grande valeur à notre disposition**, c'est à dire 7 y-a-t-il dans 13 ?

   $13 = 1 \times 7 + 6$

   Donc pour l'instant on utilisera 1 pièce de 30 et une pièce de 13 on exclut la pièce de 7 et on continue

4. Combien de fois **la plus grande valeur à notre disposition**, c'est à dire 3 y-a-t-il dans 6 ?

   $6 = 2 \times 3$

   On rend donc 1 pièce de 30 et une pièce de 7 et deux pièces de 3 euros.

   On a donc rendu 4 pièces **mais on peut faire mieux** avec 3 pièces car $43 = 2 \times 21 + 1$

Dans un algorithme lorsqu'on choisit à chaque itération le plus grand ou le plus petit d'un ensemble de valeurs et que l'on ne revient plus jamais sur ce choix on dit que c'est un algorithme **glouton** ou que l'on a adopté une stratégie gloutonne pour résoudre notre problème
La stratégie gloutonne fait partie d'un ensemble de stratégies que vous apprendrez au Lycée. Cette année vous voyez la stratégie gloutonne et la stratégie "diviser pour régner" (vue dans la recherche dichotomique), l'année prochaine en Algorithmique on approfondira la stratégie "diviser pour régner" en étudiant le tri par fusion et on découvrira une autre stratégie "la programmation dynamique"
A ce stade une question légitime nous vient à l'esprit : Comment être sûr que la stratégie gloutonne résout bien le problème ?
On a montré que cette stratégie ne fonctionne pas dans le deuxième cas en trouvant un **contre-exemple**, par contre il faut **prouver** le programme pour être sûr qu'il fonctionne dans le premier cas (On admettra que cela fonctionne)
Voici un algorithme pour le premier cas :

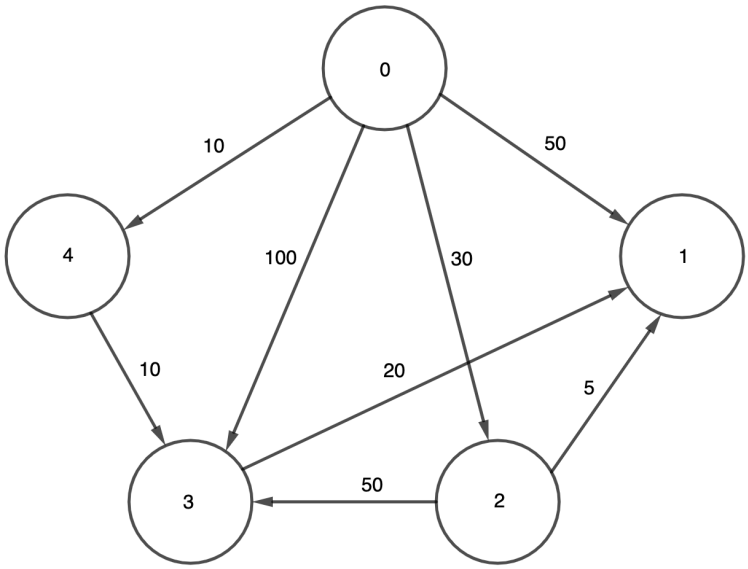
On dispose d'une fonction `div(a,b)` qui retourne le tuple (q,r) tel que $a = bq + r$ est le résultat de la division euclidienne de a par b

Algorithme 1 : Rendu de monnaie (algorithme glouton)

`renduMonnaie` (P,v)
Données : Un tableau de pièces P = [1, 2, 5, 10] et une valeur v
Résultat : Un tableau d'entiers T tel que
$v = T[0] + 2 \times T[1] + 5 \times T[2] + 10 \times T[3]$

```
1  début
2      s ← v
3      T ← [0, 0, 0, 0]
4      i ← 3
5      tant que s > 0 faire
6          T[i], s ← div(s, max(P))
7          i ← i − 1
8      fin
9      retourner T
10 fin
```

2 Plus court chemin dans un graphe (Algorithme de Dijkstra)



On appelle graphe, un ensemble de sommets, ici les éléments 0, 1, 2, 3 et 4 et d'arêtes (les flèches) reliant certains éléments entre eux
Ces arêtes sont orientées ainsi il y a une arête qui part de 0 vers 1, et valuées avec des nombres positifs par exemple, l'arête qui part de 0 vers 1 a pour valeur 50
On peut imaginer qu'un tel graphe peut représenter les temps de transport entre les différentes stations de métro d'une ville
Les distances entre les sommets sont mémorisés dans un tableau 2D, lorsqu'il n'y a pas d'arête d'un sommet vers lui-même ou un autre on met le symbole ∞

L	0	1	2	3	4
0	∞	50	30	100	10
1	∞	∞	∞	∞	∞
2	∞	5	∞	50	∞
3	∞	20	∞	∞	∞
4	∞	∞	∞	10	∞

Un des sommets est l'origine ou **la source** (on choisit ici le sommet 0)
Le but du problème est de trouver les valeurs des plus courts chemins entre la source et les autres sommets
Pour cela on dispose du tableau D = [50, 30, 100, 10] donnant dans l'ordre les distances de 0 à 1, puis de 0 à 2, puis de 0 à 3 et enfin de 0 à 4
Ce tableau va évoluer tout au long de l'exécution de l'algorithme car on va trouver une distance plus courte pour aller de 0 à 4
L'équivalent de l'ensemble des pièces est ici l'ensemble des sommets différents de la source 0, c'est à dire C = [1,2,3,4]

1. On cherche le sommet le plus proche de la source pour les éléments de C :
   C'est le sommet 4, on l'enlève de l'ensemble C, donc C devient [1,2,3] et on regarde maintenant si on peut faire mieux dans le tableau D, en passant par le

sommet 4 pour aller vers 1, 2 ou 3
Oui car le chemin 0-4-3 prend moins de temps que 0-3 donc on remplace 100 par 10 + 10 = 20 et D = [50,30,20,10]

2. On cherche encore le sommet le plus proche de la source pour les éléments de C :
   C'est le sommet 3, on l'enlève de l'ensemble C, donc C devient [1,2] et on regarde maintenant si on peut faire mieux dans le tableau D pour aller de la source vers 1, 2 via le sommet 3
   Oui car le chemin 0-4-3-1 prend moins de temps que 0-1 donc on remplace 50 par 20 + 20 = 40 et D = [40,30,20,10]
3. On cherche encore le sommet le plus proche de la source pour les éléments de C :
   C'est le sommet 2, on l'enlève de l'ensemble C, donc C devient [1] et on regarde maintenant si on peut faire mieux dans le tableau D pour aller de la source vers 1 via le sommet 2
   Oui car le chemin 0-3-1 prend moins de temps que 0-1 donc on remplace 40 par 30 + 5 = 35 et D = [35,30,20,10]

D'où l'algorithme de Dijkstra

Algorithme 2 : Algorithme de Dijkstra (algorithme glouton)

`dijkstra` (L,s)
Données : Un tableau 2D de longueurs L de dimension $n \times n$, la source s
Résultat : Un tableau D de distances les plus courtes de la source s aux autres sommets

```
1  début
2      C ← [1, 2, ..., n − 1]
3      D ← L[0]
4      pour i ∈ ⟦0; n − 1⟦ faire
5          // choix glouton
6          s ← l'élément de C qui minimise D
7          //l'élément choisi est exclu de C
8          C ← C − {s}
9          //D[w] est la meilleure distance connue à l'instant entre 0 et w
10         //D[s] est la meilleure distance connue à l'instant entre 0 et s
11         //L[s][w] est la longueur de l'arête s'il y en a une entre s et w, l'infini
            sinon
12         //On mémorise la plus petite valeur entre D[w] et D[s] + L[s][w]
13         pour chaque élément w de C faire
14             D[w] ← min(D[w], D[s] + L[s][w])
15         fin
16     fin
17     retourner D
18 fin
```

Exécution de l'algorithme avec un tableau

C	v	C	D
-	-	[1, 2, 3, 4]	[50,30,100,10]
[1, 2, 3, 4]	4	[1,2,3]	[50,30,20, 10]
[1,2, 3]	3	[1,2]	[40,30, 20, 10]
[1, 2]	2	[1]	[35, 30, 20,10]

La première ligne du tableau est l'initialisation des variables C et D
Ensuite pour chaque ligne en allant de la gauche vers la droite on prend dans C le sommet qui minimise D (à chaque fois les valeurs entourées en rouge)
Ensuite on exclu cette valeur v de C
Puis on met à jour D en regardant pour chaque sommet w dans C si le chemin 0 - - > v -> w n'est pas plus court que le chemin 0 - ->w (le symbole - - > désigne un chemin c'est à dire une succession d'arêtes du graphe alors que le symbole -> est une arête

Preuve de l'algorithme

On note $\overline{C}$ l'ensemble des sommets visités et C les sommets candidats à être visités, et pas encore visités
Au début dans $\overline{C}$ on met la source 0
On appelle chemin **connu** de la source 0 à un autre sommet x un chemin de la source vers x passant par des sommets **déjà visités**
Visiter le sommet s , par la méthode de Dijkstra cela signifie que :

1. s a été choisi parce que $D[s] = \min_{w \in C}(D[w])$ (choix glouton)
2. Ensuite s est exclu de C et donc appartient à $\overline{C}$
3. Ensuite **on met à jour** D **pour les éléments** w **de** C **en prenant le plus petit des deux nombres** $D[w]$ **ou** $D[s] + L[s][w]$

Qu'y-a-t-il dans D ?
Au début D contient toutes les distances **directes** de la source 0 aux autres sommets. S'il n'y a pas de liaison directe on dit que la distance est infinie
Ensuite D évolue suivant 1) 2) et 3)

Montrons que les instructions 1) 2) et 3) laissent invariante la propriété suivante :

Si s est dans $\overline{C}$ alors $D[s]$ donne la longueur du plus court chemin de la source vers s
Si s est dans C alors $D[s]$ donne la longueur du plus court chemin **connu** de la source vers s

1. Vrai à l'état initial
2. Montrons le **lemme** suivant : Pour deux sommets x et y entrant successivement dans $\overline{C}$ (x d'abord, puis y) alors $D[x] \leqslant D[y]$
   En effet choisir x c'est appliquer 1) donc $D[x] \leqslant D[y]$. Mais ensuite on met à jour D donc rien nous assure à priori que après la mise à jour $D[y] < D[x]$
   Or $D[y] = min(D[y], D[x] + L[x][y]) \geqslant D[x]$
3. Supposons que la propriété est vraie avant de visiter s montrons qu'elle reste vraie après 1) 2) et 3) :

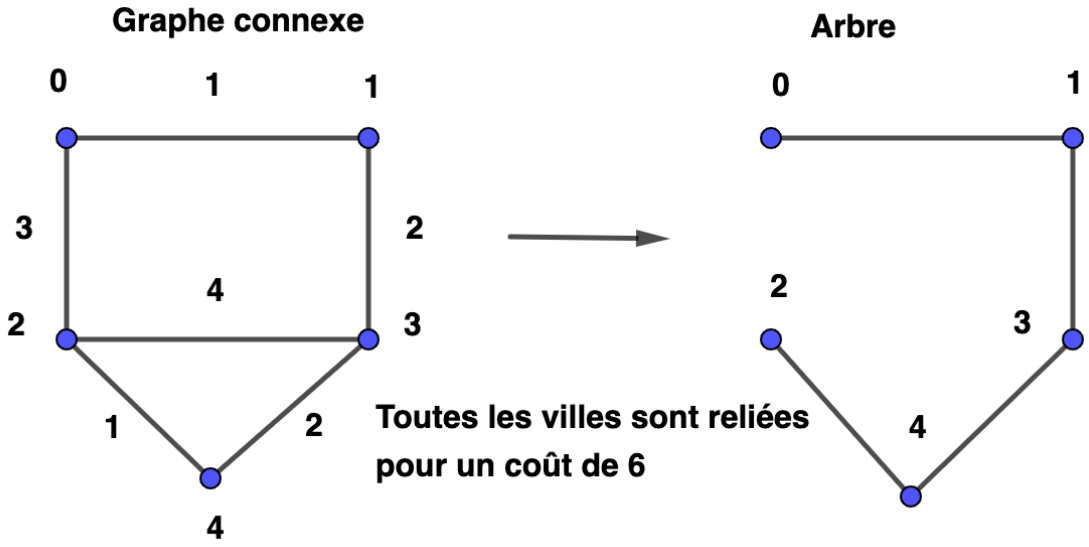
(a) Si $x \in \overline{C} - s$ puisque x est entré dans $\overline{C}$ avant s donc le lemme appliqué plusieurs fois entraîne $D[x] \leqslant D[s]$ et donc $D[s] + L[x][s] > D[x]$ donc le chemin de 0 à x le plus court reste le même malgré l'ajout de s

(b) Pour s le chemin connu reste le même

(c) Pour $x \in C$, un chemin connu peut contenir s ou pas, mais s'il contient s cela ne peut pas se faire n'importe comment, c'est le chemin connu jusqu'à s puis le chemin direct de s à x

A la fin lorsqu'on a visité tous les sommets ,D contient donc les longueurs des chemins les plus courts de la source aux autres sommets

3 Algorithme de Prim



Problème
Etant donné un graphe dont les sommets sont des villes numérotées 0,1,...,$n-1$ et les arêtes des routes à double sens **que l'on projette de construire et de telle sorte qu'entre deux villes quelconque il y aura un chemin (on dit que le graphe est connexe)**
A chaque route $\{i, j\}$ est associée un coût de construction $C(i, j) = C(j, i)$
On veut effectivement construire un réseau routier reliant toutes les villes sous la forme d'un arbre et de coût **minimal**
L'algorithme de Prim est un algorithme **glouton** qui fournit une solution exacte à ce problème
Exécution de l'algorithme sur le cas particulier
On va construire un ensemble T d'arêtes pas à pas formant l'arbre minimal
On part du sommet 0 (on peut partir de n'importe quel sommet mais prenons le sommet 0) pour l'instant le seul élément d'un ensemble S
Un arbre est un graphe connexe sans cycle c'est à dire ne possédant pas pour un sommet quelconque i de chemin commençant par i et finissant par i
Ainsi pour éviter de construire un cycle au fur et à mesure que l'on construit T, une arête que l'on ajoute dans T à chaque choix glouton a un sommet dans S et un autre

dans le complémentaire de S

1. De toutes les arêtes qui partent de 0 il y en a 2 , 0-1 et 0-2, celle qui est de coût minimal est l'arête 0 - 1 de coût 1 donc le sommet 1 est mis dans S et l'arête 0-1 est mise dans T
2. Les arêtes qui sortent de S sont 0-2 de coût 3 et 1-3 de coût 2, le choix glouton est donc l'arête 1-3 et le sommet 3 est mis dans S
3. Les arêtes qui sortent de S sont 0-2 de coût 3 et 3 - 4 de coût 2, le choix glouton est donc l'arête 3 - 4 et le sommet 4 est mis dans S
4. Les arêtes qui sortent de S sont 0-2 de coût 3 et 4-2 de coût 1, le choix glouton est donc l'arête 4-2 et le sommet 4 est mis dans S

   L'algorithme est terminé car S contient tous les sommets du graphe

   T contient les arêtes 0-1, 1-3, 3-4, 4-2 ce qui correspond à l'arbre dessiné ci-dessus

Voici l'algorithme de Prim

Algorithme 3 : Algorithme de Prim (algorithme glouton)

```
prim (N,A,C)
Données : Un graphe connexe (N,A), une fonction coût C :A → ℝ+
Résultat : Une liste T d'arêtes formant un arbre sous-tendant minimal
1  début
2      T ← φ
3      S ← {0}
4      tant que S ≠ N faire
5          // choix glouton
6          parmi les arêtes {u, v} qui "sortent" de S , u ∈ S et v ∈ N − S
7          prendre celle de coût minimal
8          T ← T ∪ {{u, v}}
9          S ← v
10     fin
11     retourner T
12 fin
```

4 Exercices - Algorithmes gloutons

Ex 1

On dispose du système de pièces suivant S = $\{1,\ 2,\ 2^2,\ ...,\ 2^7\}$
Rendre les sommes suivantes avec le système S

1. 127
2. 15
3. 255
4. A quoi cela correspond il d'un point de vue numérique ?

Ex 2

Implémenter en Python l'algorithme glouton du rendu de monnaie

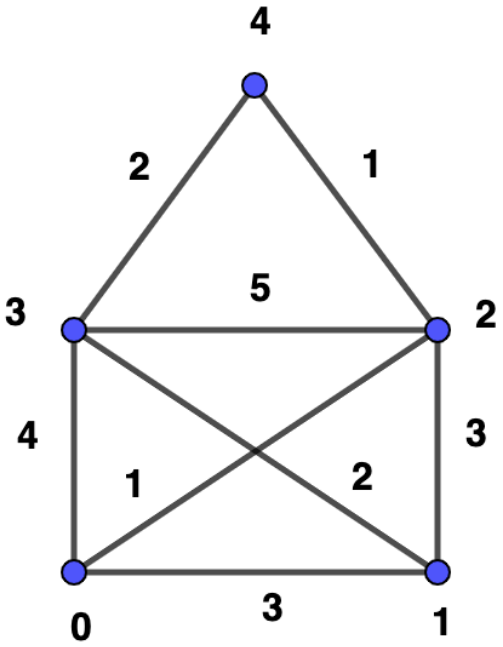
Ex 3

Exécuter l'algorithme de Dijkstra sur le graphe du cours

1. en prenant comme source le sommet 2
2. en rendant le graphe non orienté avec les mêmes valeurs et pour source 0
3. en rendant le graphe non orienté avec les mêmes valeurs et pour source 2

Ex 4

Exécuter l'algorithme de Dijkstra sur le graphe suivant

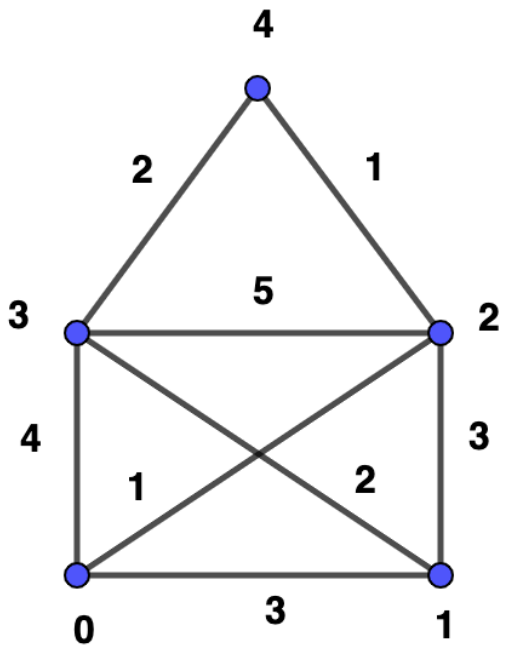


Ex 5

Implémenter l'algorithme de Dijkstra en Python

Ex 6

Exécuter l'algorithme de Prim sur le cas particulier suivant en partant de n'importe quel sommet



Ex 7

Implémenter l'algorithme de Prim en Python