

VI Annexes

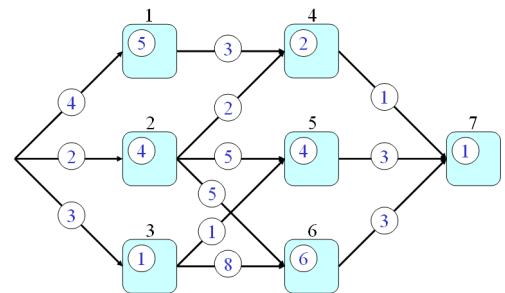
VI-1 Principes de la programmation dynamique (PrD)

- La programmation dynamique prend sa source dans le principe d'optimalité (de Richard Bellman).
- Une stratégie optimale pour résoudre un problème (représenté par un espace d'états dans lequel se trouvera un état = une solution optimale) est telle que quelque soit l'état initial et les premières décisions prises, les décisions restantes doivent constituer une stratégie optimale par rapport à la toute première décision.
- Par exemple, pour trouver un chemin optimal d'un point de départ à un point d'arrivée (p.ex. le chemin le plus court), ce principe stipule que tout sous-chemin partiel de la solution finale est forcément le chemin le plus court entre les deux étapes reliées, étant donné le départ et l'arrivée (peut être démontré par l'absurde).
- Pour pouvoir appliquer le principe de la PrD, trois conditions / étapes doivent être réalisées :
 - Définir une fonction optimale pour le problème à résoudre (p.ex. la longueur d'un chemin optimal ou le nombre minimal de pièces),
 - Décrire une formulation récursive de la fonction optimale ainsi que la condition initiale,
 - Exprimer la solution au problème en termes de la fonction optimale.

• Exemple : soit le graphe de villes ci-contre où les valeurs sur les arcs $route(x, y)$ sont les durées des trajets entre une ville x et la ville voisine y et où la valeur à l'intérieur de chaque noeud représente le temps de la traversée de chaque ville.

• Pour trouver le chemin le plus court (en terme de durée) entre le départ (à gauche) et le noeud 7, on pose :

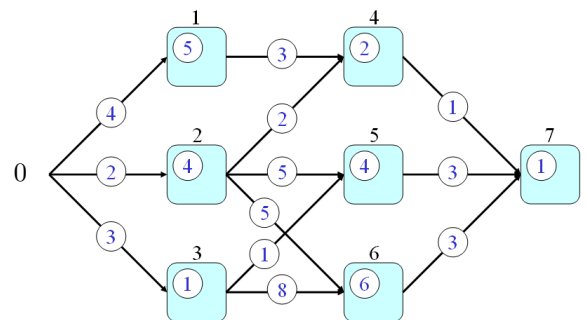
- La fonction optimale $t(v)$ représente le minimum du temps pour aller du début au noeud v (qui contiendra le temps de traverser la ville v)
- Pour arriver à la ville v en passant par une des villes la précédents p_1, \dots, p_k , la formulation récursive de la fonction optimale sera : $t(v) = \min\{t(p_1) + route(p_1, v), t(p_2) + route(p_2, v), \dots, t(p_k) + route(p_k, v)\}$
- $t(p_i)$ contient bien entendu le temps pour atteindre p_i depuis le départ.



Exemple :

- Dans ce graphe, on aura (en partant de la ville de départ 0) :

- $t(0) = 0$
- $t(1) = t(0) + 4 + 5 = 9$
- $t(2) = t(0) + 2 + 4 = 6$
- $t(3) = t(0) + 3 + 1 = 4$
- $t(4) = \min(9 + 3, 6 + 2) + 2 = 10$
- $t(5) = \min(6 + 5, 4 + 1) + 4 = 9$
- $t(6) = \min(6 + 5, 4 + 8) + 6 = 17$
- $t(7) = \min(10 + 1, 9 + 3, 17 + 3) + 1 = 12$



- L'efficacité de la PrD est basée sur le fait d'utiliser des résultats de calculs (partiels) précédents. Après avoir trouvé le chemin le plus court pour arriver à un noeud, cette information est stockée et réutilisée pour calculer le chemin optimal final.

VI-2 Graphes en Python

- Un Dictionnaire en Python est représenté sous la forme de couples *clef* \times *valeur*. Par exemple (voir cours également) :

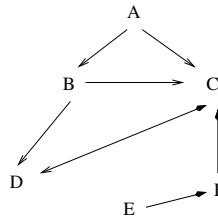
```
params = {"server": "CRI", "database": "master", "uid": "chef", "pwd": "secret"}
params.keys()
# ['server', 'uid', 'database', 'pwd']

params.values()
# ['CRI', 'chef', 'master', 'secret']

params.items()
# [('server', 'CRI'), ('uid', 'chef'), ('database', 'master'), ('pwd', 'secret')]
```

- Pour représenter le graphe suivant à l'aide d'un Dictionnaire en Python, on notera (voir cours également) :

A -> B
A -> C
B -> C
B -> D
C -> D
D -> C
E -> F
F -> C



On peut utiliser

```
graph = { 'A': { 'B': None, 'C': None },
          'B': { 'C': None, 'D': None },
          'C': { 'D': None },
          'D': { 'C': None },
          'E': { 'F': None },
          'F': { 'C': None }
        }
```

- Plusieurs autres représentations sont possibles. Par exemple, le même dictionnaire utilisant des listes donnera (remarquer que la valeur équivalent à None dans les listes est la liste vide []):

```
graph = { 'A': [ 'B', 'C' ],
          'B': [ 'C', 'D' ],
          'C': [ 'D' ],
          'D': [ 'C' ],
          'E': [ 'F' ],
          'F': [ 'C' ]
        }
```

- Différentes méthodes de parcours des graphes et arbres sont données dans le support du cours.

VI-3 Solution par le chemin minimal dans un arbre (par un Dictionnaire)

La méthode Gloutonne ne garantit pas que $Q(S,M)$ soit minimal comme les tests l'ont montré pour $M=28$ et $S=(1, 7, 23)$.

Pour ces données, la méthode Gloutonne choisira d'utiliser une pièce de 23 (la plus grande) puis 5 pièces de 1. Donc un total de 6 pièces. Mais, on peut aisément constater que 4 pièces de 7 auraient pu satisfaire la même demande !

Remarque : même si ces hypothèses ne correspondent pas à la réalité courante, du point de vue Mathématique, nous ne pouvons pas ne pas en tenir compte !

La méthode Gloutonne utilise un optimum local (choix de la plus grande pièce) qui ne débouche pas forcément sur un optimum global. Cependant, elle est très simple à calculer.

La méthode suivante est plus complexe à mettre en oeuvre mais donne une solution optimale.

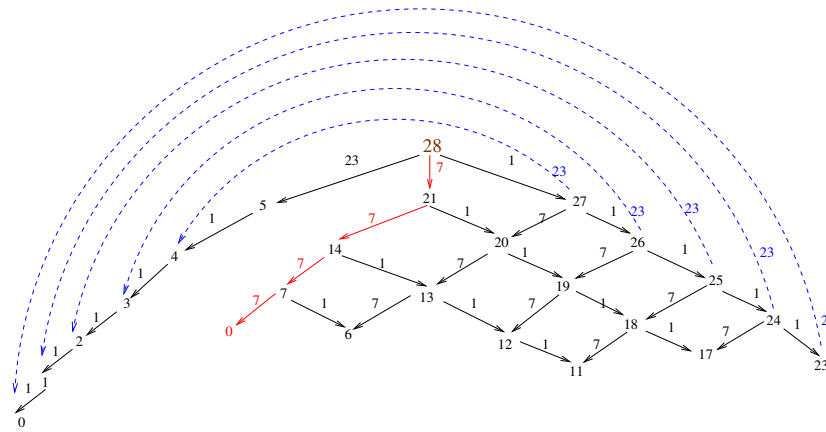
Explications :

Pour illustrer cette méthode, on suppose $M=28$ et $S=(1,7,23)$. C'est à dire, on a seulement des pièces de 1, 7 et 23 (€s ou cents) en nombre suffisant.

On construit un arbre de recherche (arbre des possibilités) dont la racine est M . Chaque noeud de l'arbre représente un montant : les noeuds autres que la racine initiale représentent $M' < M$ une fois qu'on aura utilisé une (seule) des pièces de S (parmi 1, 7 ou 23 €s). La pièce utilisée pour aller de M à M' sera la valeur de l'arc reliant ces deux montants.

Cet arbre est donc développé par niveau (*en largeur*, voir cours). On arrête de développer un niveau supplémentaire dès qu'un noeud a atteint 0 auquel cas une solution sera le vecteur des valeurs (des arcs) allant de la racine à ce noeud.

Détails de M=28 :



Dès qu'on atteint 0, on remonte de ce 0 à la racine pour obtenir la solution minimale donnant le nombre minimal d'arcs entre ce 0 et la racine. Dans la figure, on remarque qu'on utilisera 4 pièces de 7c pour avoir 28c.

Par ailleurs, on remarque que le choix initial de 23 (à gauche de l'arbre) laisse le montant $M'=5c$ à satisfaire lequel impose le choix de 5 pièces de 1c.

Le principe de l'algorithme correspondant à un parcours en largeur dont la version itérative utilise une *File d'attente* (comme devant un guichet de cinéma, voir cours, les parcours de graphes et arbres) est :

Algorithme de principe du parcours en largeur :

```

Fonction Monnaie_graphe
% on suppose qu'il y a un nombre suffisant de chaque pièce/billet dans la tableau S
Entrées: la somme M
Sorties: le vecteur T et Qopt(S,M) le nombre de pièces nécessaires
File F = vide;
Arbre A contenant un noeud racine dont la valeur = M
Enfiler(M) % la file F contient initialement M
Répéter
    M' = défiler()
    Pour chaque pièce vi ≤ M' disponible dans S :
        S'il existe dans l'arbre A un noeud dont la valeur est M' - vi
        Alors établir un arc étiqueté par vi allant de M' à ce noeud
        Sinon
            Créer un nouveau noeud de valeur M' - vi dans A et lier ce noeud
            à M' par un arc étiqueté par vi
        Enfiler ce nouveau noeud
    Fin Si
Jusqu'à (M' - vi = 0) ou (F = vide)

Si (F est vide Et M' - vi ≠ 0)
Alors il y a un problème dans les calculs! STOP.
Sinon Le dernier noeud créer porte la valeur 0
On remonte de ce noeud à la racine et on comptabilise dans T où Ti = le nombre d'occurrences des
arcs étiquetés vi de la racine au noeud v de valeur 0
     $Q(S, M) = \sum_{i=1}^n T_i$  % somme des valeurs du vecteur T
Fin si
Fin Monnaie_graphe
  
```

☞ Voir en annexe une petite introduction aux graphes. Voir également le cours pour plus de détails.

☞ Contrairement à une implantation de graphes avec adressage dispersé (et pointeurs), l'utilisation d'un dictionnaire de Python (Dict) pour représenter le graphe permet de disposer en permanence de la totalité du graphe (moyennant un coût en espace évidemment!).

Dans l'exemple suivant, on a un graphe dont les noeuds de différents niveaux sont visibles et disponibles.

```

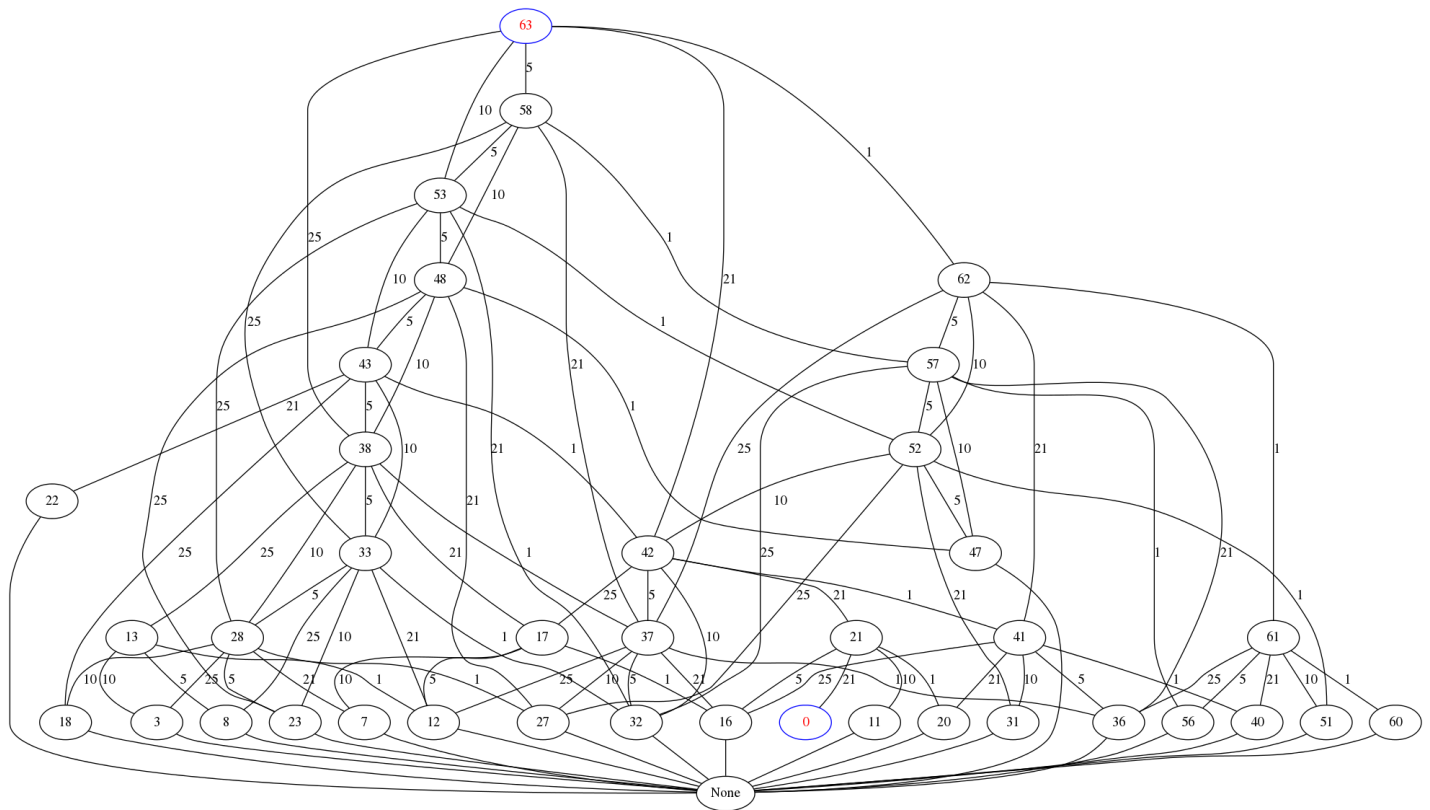
graph = { 'A': { 'B': None, 'C': None },
          'B': { 'C': None, 'D': None },
          'C': { 'D': None },
          'D': { 'C': None },
          'E': { 'F': None },
          'F': { 'C': None }
        }
  
```

De ce fait, il n'est plus besoin d'une file d'attente (utilisée dans l'algorithme ci-dessus). L'accès à l'ensemble des fils d'un niveau est directement disponible dans un dictionnaire Python. Voir cours pour les implantations alternatives.

Proposer une solution Python de cette méthode. Dans une première étape, développer le graphe pour atteindre la feuille zéro puis construisez le chemin de la racine à cette feuille.

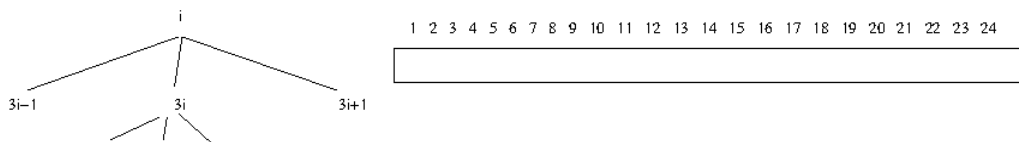
VI-4 Arbre d'un autre exemple

- L'arbre du système $Q(S,M) = Q([1,5,10,21,25], 63)$:



VI-5 Utilisation d'un arbre ternaire représenté par une liste

- On peut appliquer la solution par un Dictionnaire Python à un arbre n -aire où n est la taille de S .
- Ci-dessous, on développe le cas de $n=3$. Voir ci-après pour $n=4,5, \dots$
- Comme pour le BE2 (arbre de la partie "fusion"), il est possible d'utiliser un arbre représenté par une liste plate :



- Pour le cas particulier où on a seulement $k = 3$ pièces de monnaies différentes (l'ensemble S donnant ici une arbre ternaire), la représentation ci-dessus permet de passer :
 - de l'indice `pere` du père aux fils : `3*pere-1, 3*pere, 3*pere+1` (for `j in range(k) : 3*pere+j-1`)
 - de l'indice `fils` au père : `pere= fils // 3; if fils % 3 == 2 : pere +=1`
 - La somme initiale (ici 28 à rendre) est à l'indice 1 (l'indice 0 de la liste non utilisé).

☞ La **faiblesse** de cette méthode est que si l'ensemble S contient k pièces, l'équation ci-dessus devra être changée. Par exemple, pour $k=2$, les fils seront en $2i, 2i+1$ (la racine à l'indice 1). Pour $k=4$ et $k=5$, voir ci-après.

Pour $k > 4$, il faudra modifier l'équation (voir ci-après). Noter que Python permet de passer une fonction en paramètre et on pourra donc préparer et transmettre une fonction qui permet de retrouver les fils / père en fonction de k à nos calculs.

- Pour plus de souplesse, on peut paramétrer la solution par :
 - k le nombre de pièces (taille de S) et le fait de savoir si pour cette valeur de k , l'emplacement d'indice 0 est utilisé ou pas (pour $k=4$, il est utilisé mais pas pour $k = 2, 3, 5$)
 - Une fonction donnée en argument d'appel qui permet de calculer les indices des fils à partir de l'indice du père. Cette fonction donnera un couple si $k = 2$, un triplet si $k = 3$, un quadruplet si $k = 4$, etc.

◦ Une fonction donnée en argument d'appel qui permet de calculer l'indice du père à partir de l'indice d'un fils. Cette fonction accepte un entier et renvoie un entier.

→ Par exemple, pour $k = 3$

```
pere2fils_k_is_3 = lambda p : (3*p+j-1 for j in range(3))
list(pere2fils_k_is_3(2)) # [5, 6, 7]
list(pere2fils_k_is_3(21)) # [62, 63, 64]

fils2pere_k_is_3 = lambda f : f//3+1 if f % 3 == 2 else f//3
fils2pere_k_is_3(37) # 12
fils2pere_k_is_3(21) # 7
fils2pere_k_is_3(20) # 7
```

• Pour $k = 4$ et l'indice *pere* donné (cf. BE2), le premier fils est à l'indice $4pere + 1$, le 2e à l'indice $4pere + 2$, le 3e à $4pere + 3$ et le dernier à $4pere + 4$.

◦ Pour retrouver le parent (*pere*) d'un noeud d'indice *fils* :

```
pere=fils // 4
if fils % 4 == 0 : pere -= 1
```

☞ Rappelons que pour $k = 4$, l'indice 0 est utilisé (pas pour $k = 2$).

• Pour $k = 5$ et l'indice *pere* donné, les fils sont en indice $5pere - 3, 5pere - 2, 5pere - 1, 5pere, 5pere + 1$

◦ Pour retrouver le parent (*pere*) d'un noeud d'indice *fils* :

```
pere=fils // 5
if fils % 5 > 1 : pere += 1
```

☞ Rappelons que pour certains k , l'indice 0 n'est pas utilisé.

• Vous pouvez chercher la relation pere-fils pour $k > 5$.

VI-6 Arbre avec des tableaux

• Parmi les représentations possible des arbres et graphes, on choisit ici la représentation de l'arbre avec des tableaux.

• Le graphe $G(V,E)$ (avec V : ensemble des noeuds, E : ensemble d'arcs/arêtes) est représenté par ces deux tables.

• Habituellement, la table **V** contient toutes les informations sur les noeuds (p. ex. une ville, sa population, sa superficie, ...).

La table **E** n'a pas besoin de répéter ces informations et se contente de contenir le nom du successeur, ou mieux, l'indice du successeur dans la table **V**.

Les flèches rouges dans la figure ci-contre renvoient vers le noeud successeur dans **V** : ce renvoi se fait via le nom du *successeur_i* (le même nom que dans **V**) ou via l'indice du successeur dans **V**.

Si une pondération des arcs (arêtes) est présente, chaque case de **E** sera un couple (*noeud_succ*, *poids*).

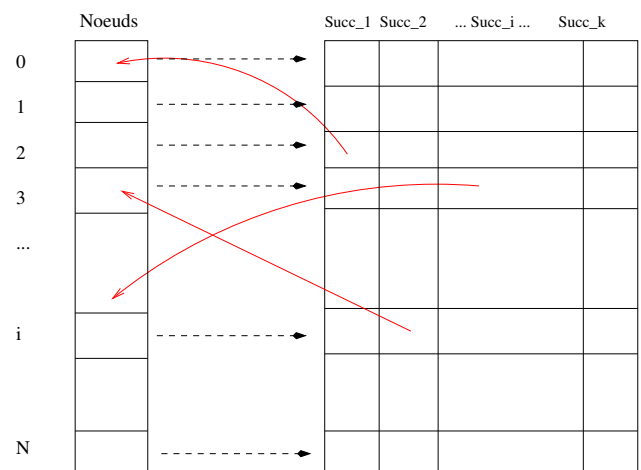


Table V

Table E (les arcs/arêtes)

• Dans le cas du problème de la monnaie, sachant que l'information d'un noeud est un simple entier (un montant), il est plus simple que chaque successeur contienne également un montant.

- Par exemple, pour $Q(S,M) = Q([1,7,23], 28)$, on peut avoir :

Les 3 colonnes de la table E correspondent aux 3 valeurs :

28-1, 28-7 et 28-23.

La table E contient 3 colonnes car $|S| = 3$.

La table E étant initialisée par *None*, ces valeurs ne se modifient pas si pour un montant (p. ex. 5), on ne peut pas utiliser certaines les pièces de **S**.

- Notons que cette représentation ne modifie en rien le type du parcours (en profondeur ou en largeur) des arbres/graphes.
- Rappelons que nous utilisons un parcours en largeur.

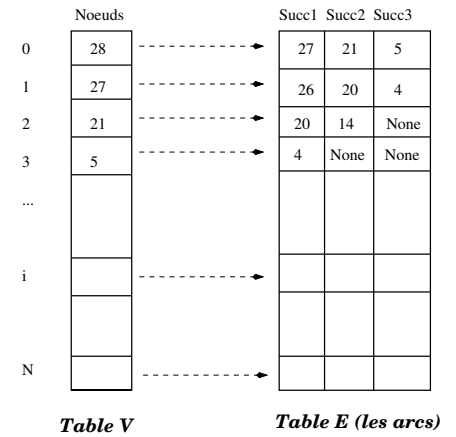


Table des Matières

I. Problème de monnaie	1
II. Approche algorithmique	2
II-1. Technique Gloutonne	2
II-2. Version optimale	3
III. Programmation Dynamique (PrD)	3
IV. Solution par un système d'équations	4
V. Travail à rendre	5
VI. Annexes	6
VI-1. Principes de la programmation dynamique (PrD)	6
VI-2. Graphes en Python	7
VI-3. Solution par le chemin minimal dans un arbre (par un Dictionnaire)	7
VI-4. Arbre d'un autre exemple	9
VI-5. Utilisation d'un arbre ternaire représenté par une liste	9
VI-6. Arbre avec des tableaux	10