Algorithmes et Structures de données
Problème de la monnaie rendue

INF TC1

2015-16 (S6)

Version Élèves

2015-16

I Problème de monnaie

• Fonctionnement de la machine à rendre la monnaie.
• Le problème est connu sous le nom de *"Change making"* (rendre la monnaie)

Définition du problème :

Dans une machine capable de rendre la monnaie, on a des pièces de 1c à 2€s, puis des billets. On suppose pour simplifier qu'il n'y a que des pièces. Un billet de 5€s sera représenté comme une pièce de 500 centimes.
De plus, on suppose qu'il existe un nombre suffisant (autant que nécessaire) de chaque pièces.

Un stock de pièces est alors un tuple $S = (v_1, v_2, ..., v_n)$ où l'entier $v_i > 0$ est la valeur de la ième pièce.
Pour refléter le fait qu'on a des pièces de 1,2 et 5c, S contiendra $v_1 = 1$ (1 centime), $v_2 = 2, v3 = 5$.

Pour un système **S** et un entier positif **M**, *le problème de monnaie* est un problème d'optimisation combinatoire (S, M) permettant de trouver le tuple $T = (x_1, x_2, ..., x_n)$ avec $x_i \geq 0$ qui minimise $\sum_{i=1}^{n} x_i$ sous la contrainte $\sum_{i=1}^{n} x_i.v_i = M$.
Autrement dit, sachant que x_i est le nombre de pièces de valeur v_i utilisées, on veut minimiser le nombre total de x_i utilisés.

Exemple : pour rendre la somme *M*, on utilisera x_i fois la pièce v_i. Pour un stock de pièces contenant :

i	1	2	3	4	5	6	7	8	9	...
$\mathbf{v_i}$	1	2	5	10	20	50	100	200	500	...

donc avec $S = (1, 2, 5, 10, 20, 50, 100, 200, 500, ...)$, si on utilise 3 pièces de 2€s (200c : $v_8 = 200$), on aura dans la réponse T la valeur $x_8 = 3$.

Dans ce problème d'optimisation, l'objectif à minimiser est le nombre total de pièces rendues.
Appelons $Q(S, M) = \sum_{i=1}^{n} x_i$ la quantité de pièces à rendre pour le montant *M* étant donné le système *S* décrit ci-dessus. Une solution optimale à ce problème est telle que *Q(S,M)* soit minimale : $Q_{opt}(S, M) = min \sum_{i=1}^{n} x_i$.

• **Un exemple :** pour rendre 9€s étant donné S ci-dessus, parmi les possibilités (en n'utilisant que les pièces ≥1€=100c) :

- 9 pièces de 1€ et 0 pour toutes les autres — **T=(0,0,0,0,0,0,0,9,0,0,0...0)** → 9 pièces, $Q(S, 9) = 9$
- 5 × 1€+2 × 2€s, 0 pour les autres — **T=(0,0,0,0,0,0,0,5,2,0,0...0)** → 7 pièces, $Q(S, 9) = 7$
- 1 × 1€+4 × 2€s, 0 pour les autres — **T=(0,0,0,0,0,0,0,1,4,0,0...0)** → 5 pièces, $Q(S, 9) = 5$
- 2 × 2€s+1 × 5€s, 0 pour les autres — **T=(0,0,0,0,0,0,0,0,2,1,0...0)** → 3 pièces, $Q(S, 9) = 3$
- 3 × 1 + 3 × 2, 0 pour les autres — **T=(0,0,0,0,0,0,0,3,3,0,0...0)** → 6 pièces,
- 4 × 1 + 1 × 5, 0 pour les autres — **T=(0,0,0,0,0,0,0,4,0,1,0...0)** → 5 pièces,
- etc. sans parler des solutions avec des centimes !

La solution *T=(0,0,0,0,0,0,0,0,2,1,0...0)* minimise le nombre total de pièces rendus est de 3 pièces.
Donc, $Q_{opt}(S, 9) = min\ Q(S, 9) = 3$.

• **Un autre exemple** : pour rendre la somme de à 1989€s pièces (sans les centimes), on aura :
$1989 = 500 \times 3 + 488 = 500 \times 3 + 200 \times 2 + 50 \times 1 + 20 \times 1 + 10 \times 1 + 5 \times 1 + 2 \times 2$
soit $3 + 2 + 1 + 1 + 1 = 8$ grosses pièces (billets) et $1 + 2 = 3$ pièces, tout est en euro.

Résolution du problème

Pour résoudre ce problème (démontré NP-difficile relative à la taille de *S*), plusieurs approches sont possibles dont les approches :

I. Algorithmique : technique Gloutonne (pas toujours optimale)
II. Algorithmique : chemin de longueur minimale dans un arbre de recherche,
III. Algorithmique : Programmation Dynamique, ... et autres méthodes algorithmiques (cf. quasi-Dijkstra)
IV. Système d'équations à optimiser
Etc.

II Approche algorithmique

Soit M la somme à rendre et S un tableau de pièces/billets disponibles. Un exemple de S est donné ci-contre :

On se limite arbitrairement aux pièces/billets 100 €s max.

Dans sa forme complète (cas réel), on gère également le nombre de chaque pièce/billet disponible (la table **D**).

d[i]=k veut dire : il y a **k** pièces/billets du montant v_i disponibles (pièces ou billets du montant *v[i]*) à l'indice *i* dans la table *S*.

Pour satisfaire la somme M, les solutions algorithmiques se déclinent sous différentes formes.
Nous abordons la technique dite Gloutonne, le chemin minimal dans un arbre de recherche et **la Programmation Dynamique**.
☞ Ci-dessous, on ne tient pas compte de la table D.

	la table S	**la table D**
indice	Valeur (v_i)	Disponibilité (d_i)
1	1c	nombre de pièces de 1c disponibles
2	2c	nombre de pièces de 2c disponibles
3	5c	
4	10c	
5	20c	
6	50c	
7	100 (1€)	1€
8	200 (2€s)	pièces 2€s
9	500 (5€s)	billets de 5€s
10	1000	billets de 10€s
11	2000	billets de 20€s
12	5000	billets de 50€s
13	10000	billets de 100€s

On suppose donc qu'il y a un nombre suffisant de chaque pièce/billet dans la tableau S. On suppose également que S est ordonnée croissante.

II-1 Technique Gloutonne

Cette méthode ne donne pas forcément un nombre minimal de pièces mais elle est simple à implanter.
Son principe est simple : on trouve la pièce la plus grande inférieure ou égale à M. Soit V_i cette pièce. On utilise ($x_i = M\ div\ v_i$) fois la pièce v_i ; le reste à traiter sera M'=($M\ mod\ v_i$). Ensuite, on recommence suivant le même principe pour satisfaire M'...
Ce qui donne l'algorithme de principe suivant (on ne tient pas compte ici de D : $d_i = \infty$) :

```
Fonction Monnaie_Gloutonne
Entrées: la somme S, M
Sorties: le vecteur  T = Q(S,M) : le nombre de pièces nécessaires
    M'=M
    Répéter
        Chercher dans S l'indice i tel que Vi =< M'
        Ti = M div Vi                          % on utilisera Ti fois la pièce Vi
        M' = M mod Ti
    Jusqu'à M' = 0

    Constituer T avec les Ti utilisés
    Q(S,M) = sum_{i=1}^{n} Ti          % somme des valeurs du vecteur T
Fin Monnaie_Gloutonne
```

Exemple : pour satisfaire 236,65€s , cet algorithme se déroule de la manière suivante :

$23665 >= 10000$
$T_{13} = 23665\ div\ 10000 = 2$ % T_{13} correspond à 100€s
$M' = 23665\ mod\ 10000 = 3665$ % on utilisera 2 billets de 100€s

$3665 >= 2000$
$T_{11} = 3665\ div\ 2000 = 1$ % T_{11} correspond à 20€s
$M' = 3665\ mod\ 2000 = 1665$ ...

→ On obtient $T_{1..13} = (0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 2)$ et $Q(S, M) = 9$

☞ En Python, les indices commencent à zéro ! Faire -1 sur les indices ci-dessus.

Remarque :
Dans certains systèmes (cf. un parking), l'utilisateur paie une somme et attend sa monnaie (le reste). On peut d'emblée intégrer les pièces introduites (par un client) au tableau S. De cette sorte, on pourra toujours essayer de satisfaire une demande (même si S est initialement vide auquel cas un échec suivra !).

II-1-a Prise en compte de la table D

On a supposé disposer d'une quantité suffisante de chaque pièce (Donc d_i illimité et suffisamment grande). Modifier la solution précédente pour tenir compte du nombre disponible de chaque pièce (table D en paramètre, avec d_i limité).

III Programmation Dynamique (PrD)

• Pour une introduction à la Programmation Dynamique, voir support du cours.

Explications :
Supposons que l'on puisse, pour le montant *M*, savoir calculer une solution optimale pour tout montant $M' < M$. Pour satisfaire *M*, il faudra alors prendre une (seule) pièce v_i supplémentaire parmi les n pièces disponibles. Une fois cette pièce choisie, le reste $M' = M - v_i$ est forcément inférieur à *M* et on sait qu'on peut calculer un nombre optimal de pièces pour M'. Par conséquent : $Q_{opt}(S, M) = 1 + Min\ Q(S, M')$.

Le raisonnement ci-dessus est un raisonnement par Induction Mathématique.
L'inconvénient de cette méthode est que le nombre d'opérations nécessaires est proportionnel à *M*, donc exponentiel en la taille de *M* (sauf si *M* est représenté avec des bâtons - système monadique !).

Dans cette partie, on utilise la programmation dynamique dont les principes sont rappelés ici :
- ◦ identifier une famille de sous-problèmes,
- ◦ exprimer les équations qui relient ces sous-problèmes, puis résoudre ces problèmes dans un ordre ascendant.

Propriétés de la solution :

◦ Pour la somme $m \in [0, M]$, notons $Q_{opt}(i, m) = min\ Q(i, m)$ qui sera le nombre **minimal** de pièces nécessaires pour former la somme **m** lorsque seules les pièces de type $i..1, i \leq |S|$ sont disponibles ($|S|$ est la taille de S).

Pour être *pratique*, sachant que le stock de pièces **S** ne varie pas, on fait varier $i \in 1..|S|$ en raisonnant sur la i^{eme} pièce (le i^{eme} élément de S est une pièce de type i : $S[i]$ vaut v_i cents, cf. plus haut) :

◦ Pour atteindre une somme nulle, aucune pièce n'est nécessaire.
Donc, nous avons : $Q(i, 0) = 0, \forall i \in 1..|S|$ (autrement dit, quelque soit **S**, on a besoin d'aucune pièce si $M = 0$)

◦ Atteindre une somme non nulle est impossible si l'on ne dispose d'aucune pièce (caractérisé ici par $i = 0$).
Donc, nous avons : $Q(0, m) = \infty\ si\ 0 < m \leq M$

◦ Enfin, pour atteindre une somme quelconque **M**,
- soit on utilise au moins une pièce de type *i*, auquel cas les autres pièces sont de types *1* à *i* (i est compris) et doivent constituer une manière optimale d'atteindre la somme $S - v_i$;
- soit on n'en utilise aucune (du type i), auquel cas les pièces utilisées seront de types *1* à $i - 1$.

☞ Remarquons que i décroit : ses valeurs vont de $|S|$ à 0.

Nous avons par conséquent ($i : 1..|S|$ représente une pièce dans le stock S de valeur v_i, la somme $m \in 1..M$) :

$$Q_{opt}(i, m) = min \begin{cases} 1 + Q(i, m - v_i) & si\ (m - v_i) \geq 0 \quad \textit{on utilise une pièce de type i de valeur } v_i \\ Q(i - 1, m) & si\ i \geq 1 \quad \textit{on n'utilise pas la pièce de type i, essayons i-1} \end{cases}$$

Pour être fidèle au principe de *mémorisation* de la PrD, on remplit une matrice $mat[|S|][M]$. Pour ce faire, la version *ascendante* (il existe également une version *descendante*) de la PrD calcule tous les $Q_{opt}(i, m)$, d'abord pour *i* croissant, puis pour *m* croissant (double itérations de l'algorithme ci-dessous).
Ainsi, pour $1 \leq i \leq |S|, 1 \leq m \leq M$, $mat[i][m]$ devient synonyme de $Q_{opt}(i, m)$ et pour calculer la valeur d'une case quelconque $mat[i][m]$, les optima pour toutes les cases depuis $mat[0][0]$ sont déjà calculés [1] [2].

Rappelons que les v_i sont strictement positifs.

On stocke les différentes valeurs de $Q(i, m), m = 0..M, i = 0..|S|$ dans la matrice **mat** [3]. La table S contient les pièces disponibles de valeurs v_i dans l'ordre croissant (la dernière est la plus grosse ; en faisant $i - 1$, on les consulte dans l'ordre décroissant).
La valeur recherchée finale est égale à *mat[M][| S |]* qui sera la toute dernière case de la matrice (en bas à droite).

1. Rappel : i est un indice et S[i] est une pièce du stock S de valeur v_i. On préfère travailler avec les i plutôt qu'avec les v_i qui sont dis-contigües.
2. cf. la fonction $Fib(N) = Fib(N-1) + Fib(N-2)$ et le fait que le calcul de $Fib(N)$ est $O(1)$ si $Fib(N-1)$ et $Fib(N-2)$ sont déjà calculées et disponibles. Si on admet $Fib(0) = Fib(1) = 1$ par hypothèse, $Fib(2)$ n'aura qu'à additionner $Fib(0)$ et $Fib(1)$ déjà disponibles ; $Fib(3)$ d'utiliser $Fib(1)$ et $Fib(2)$, etc. La complexité de $Fib(.)$ devient ainsi linéaire (moyennant $O(N)$ en espace) versus une version récursive collant à la définition de $Fib(.)$ qui sera d'une complexité *exponentielle* quand $N \to \infty$. De plus, si on ne stock que $Fib(i-1)$ et $Fib(i-2), 0 \leq i \leq N$, la complexité en espace descend à $\Theta(2)$.
3. Pour les indices nuls, voir $Q(i, 0)$ et $Q(0, m)$ des propriétés de Q.

Le pseudo-code peut s'écrire de la façon suivante :

```
fonction Monnaie(S,M) :                    S est le stock des pièces, M est le montant (la somme)
    soit mat la matrice d'indices [0, |S|] x [0, M]
    pour i = 0 à |S| faire
        pour m = 0 à M faire
            si m = 0 alors
                mat[i][m] = 0
            sinon si i = 0 alors
                mat[i][m] = infini
            sinon
                mat[i][m] = min(
                                si m - vi >= 0  alors 1 + mat[i][m - vi]  sinon infini
                                si i >= 1       alors mat[i-1][m]         sinon infini
    renvoyer mat[|S|][M]
```

La complexité de l'algorithme en espace est $O(M.|S|)$, c'est-à-dire le nombre d'éléments de la matrice.
Sa complexité en temps est la même, car il suffit d'un nombre constant d'opérations pour calculer la valeur de chaque case de la matrice.

Proposer une solution Python de cette méthode. Dans une première étape, développer trouver simplement le nombre minimal de pièces. Puis modifier votre solution pour donner les pièces utilisées.

IV Solution par un système d'équations

Il suffit de résoudre le système défini ci-dessus (dans la définition du problème) par un solveur quelconque de système linéaires :

$$\text{Minimize } \sum_{i=1}^{n} x_i =$$

$$\text{Subject to}$$

$$\sum_{i=1}^{n} x_i \, v_i = M$$

$$x_i \leq 0$$

$$v_i > 0$$

Cependant, en terme d'efficacité et disponibilité de ressources embarquées, la solution par la Programmation Dynamique semble mieux adaptée.
On note également que pour résoudre le système ci-dessus, nous avons besoin d'un solveur. Les plus curieux pourraient regarder du côté du : Matlab (commercial), LpSolve, GLPk, MiniZinc, et autres solveurs publics.

V Annexes

V-1 Graphes en Python

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

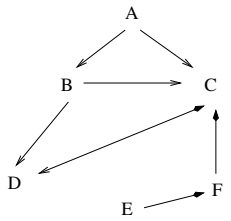
```
params = {"server":"CRI", "database":"master", "uid":"chef", "pwd":"secret"}
params.keys()
# ['server', 'uid', 'database', 'pwd']

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.

V-2 Solution par le chemin minimal dans un arbre

La méthode Gloutonne ne garantie 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 !

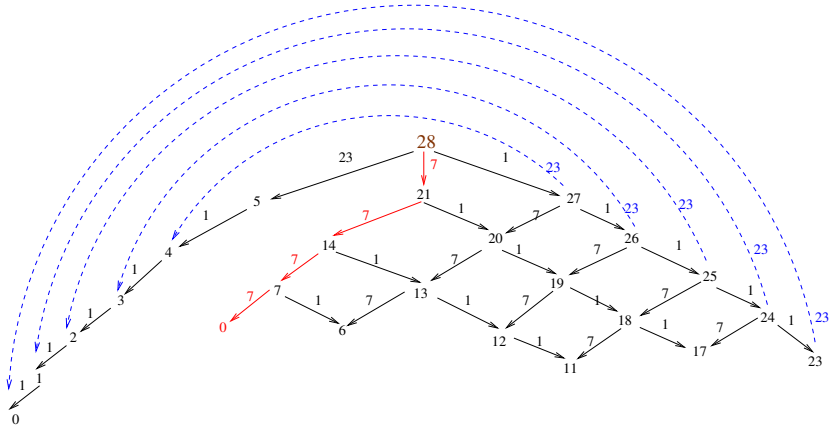
Remarque : même si ces hypothèses ne correspondent pas à la réalité courante, du point de vue Mathématique, nous ne pouvons pas ne pas en tenir compte !
La méthode Gloutonne utilise un optimum local (choix de la plus grande pièce) qui ne débouche pas forcément sur un optimum global. Cependant, elle est 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  Acontenant 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) = ∑_{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 cout 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-dessous). 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.

V-3 Arbre d'un autre exemple

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

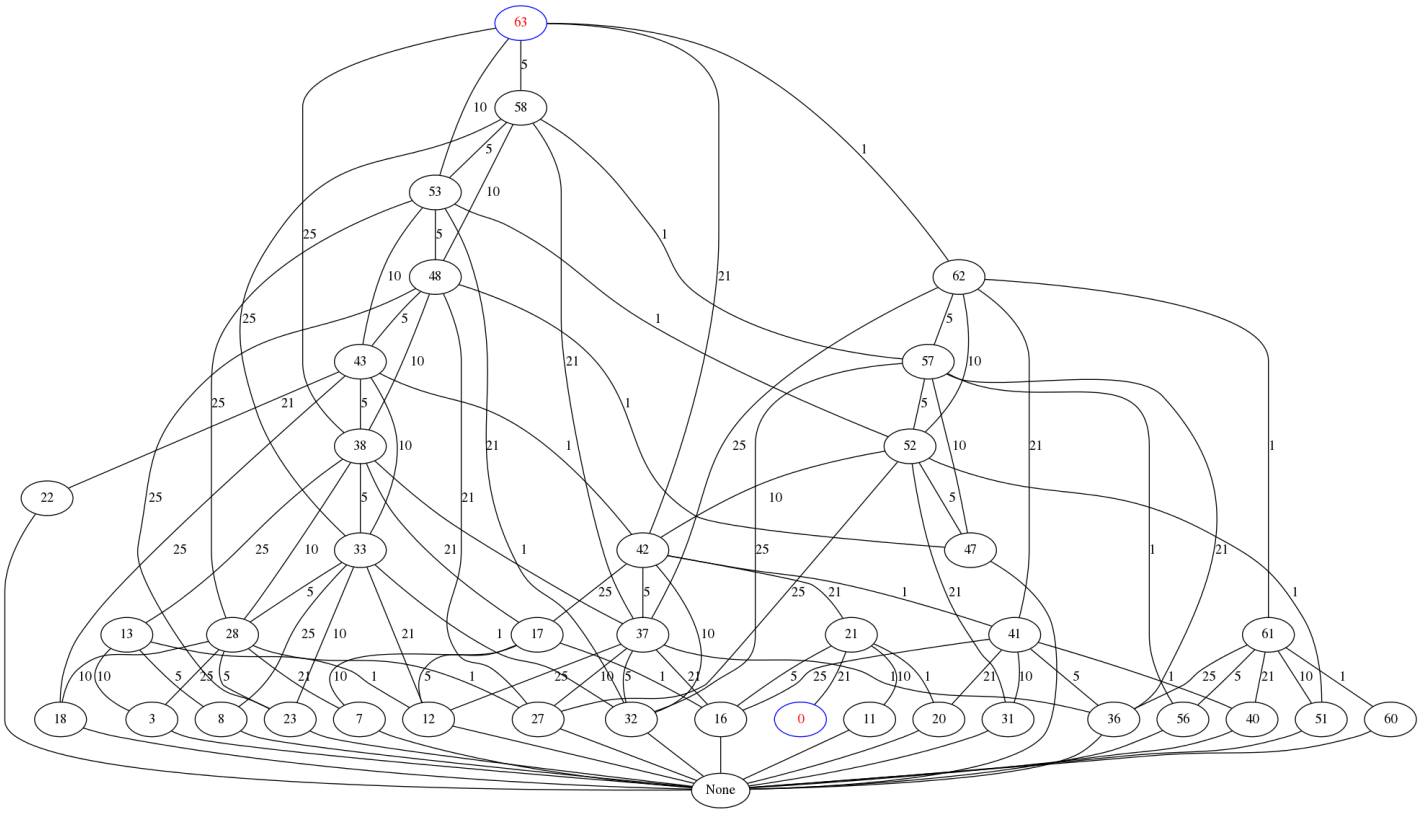


Table des Matières