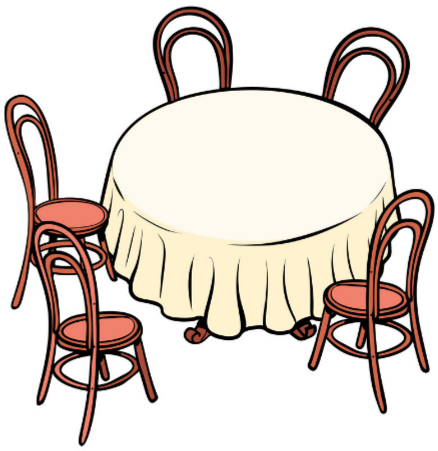
Micro Projet : Dîner des ennemis



UE : Algorithmique générale MAIN

Étudiant :
AMAIRI Tahar

Enseignant :
OUZIA Hacene

Année :
MAIN3

4 Mai 2021

Table des matières

1 Partie A : Modélisation

1.1 Question 1

La première étape est de modéliser informatiquement un invité comme un objet que nous pourrons manipuler très facilement. Mais comment s'y prendre ? Un invité, au vu de l'énoncé du problème, est caractérisé par deux éléments : son identité et celle de ses ennemis. Avec ces informations, transformer un invité en tant qu'objet informatique devient très simple à l'aide des structures. Ainsi, un invité n'est autre qu'une structure composée de deux variables :

— `int number`, un entier strictement positif représentant l'identité de l'invité,
— et `vector<int> ene`, un vecteur d'entiers strictement positifs contenant l'identité des ennemis de l'invité.

1.2 Question 2

Le problème que nous cherchons à résoudre est en réalité un exercice combinatoire revenant à trouver toutes les combinaisons de différentes tailles k parmi N tel que $k = 1, 2, 3, .., n$ et $N = 1, 2, 3, .., n$ avec n le nombre d'invités. En effet, il faut voir une combinaison (par exemple $[1, 2, 3]$) comme une table à laquelle sont assis des invités (ici les invités 1, 2 et 3 sont assis à une table de taille $k = 3$). De plus, nous n'avons aucune contrainte sur la taille des tables, d'où une résolution sur non pas quelques tailles k spécifiques mais sur toutes les tailles k. Néanmoins, nous avons bien une contrainte nous obligeons à sélectionner uniquement les combinaisons contenant des invités qui ne sont en aucun cas ennemis. Ainsi, il est clair qu'avant de commencer à chercher la configuration de tables la plus optimale (i.e avec un nombre de tables minimal), il faut obtenir les combinaisons qui sont "cohérentes" vis à vis de la configuration des invités. C'est ce qu'on appellera l'ensemble des solutions réalisables.

Mais comment pouvons-nous obtenir ces solutions ? Une première approche simple, est de calculer toutes les combinaisons tout en filtrant celles qui sont réalisables mais cela prendra énormément de temps. En effet, pour obtenir par exemple toutes les combinaisons pour $n = 20$, il nous faudra calculer et filtrer plus d'un million de choix possibles ($2^{20} = 1,048,576$).

La seconde approche bien plus intéressante est d'utiliser un algorithme de retour sur trace (dit de "backtracking") nous permettant d'éviter le calcul des combinaisons non solutions du problème. Mais comment cet algorithme fonctionne ? Par exemple, si la combinaison $[1, 2]$ n'est pas solution alors toute combinaison contenant cet sous-ensemble aussi ne l'est pas. Ainsi, en éliminant la combinaison [1,2] dès le début, on ne calculera pas les prochaines combinaisons contenant cet sous-ensemble. Cela résultera en un temps de calcul considérablement plus inférieur que celui de la première méthode.

Pour implémenter cet algorithme, il nous faut une fonction qui construit chaque combinaison à partir de sous-combinaisons de taille k plus petite :

Algorithm 1: back_tracking_comb(res,guests_list,guest_numbers,temp,idx)

Input : - res = liste de liste vide stockant les combinaisons
- guests_list = [StructInv1,StructInv2,...,StructInvn]
- guest_number = [1,2,...,n]
- temp = liste vide et idx = 0

Output: res contenant toutes les tables éligibles

```
1  Pseudo-code :
2  if temp non vide then
3  |   res.push(temp)                                // on push temp dans res
4  end
5  tmp                                               // une liste vide
6  for i = idx; i < n; i++ do
7  |   tmp ← temp
8  |   tmp.push(guest_numbers[i])
9  |   if !IfEne(guest_numbers,tmp) then
10 |   |   back_tracking_comb(res,guests_list,guest_numbers,tmp,i+1)
11 |   end
12 |   tmp.clear()                                   // on vide tmp
13 end
```

On remarque qu'avant chaque appel récursif de la fonction (ligne 9), on vérifie à l'aide d'une fonctionne booléenne, `IfEne`, si la combinaison actuelle respecte la configuration des invités. Si ce n'est pas le cas, alors on ne fait pas d'appel récursif avec et on évite donc de construire toutes les combinaisons ayant pour origine celle-ci. La complexité de cet algorithme (sans celle de `IfEne` qui est totalement aléatoire car variant selon la taille des combinaisons à vérifier) varie selon celle de la configuration des invités : si très peu d'invités sont ennemis alors elle se rapprochera de $\mathcal{O}(2^n)$ sinon elle s'en éloignera, car on aura très peu de combinaisons éligibles et donc moins d'appel récursif.

2 Partie B : Énumération brute

2.1 Question 1

En ayant maintenant la liste des tables respectant la configuration des invités, il est temps de trouver la combinaison optimale de table, telle qu'une condition ne contient pas plus d'invités qu'il y en a et qu'aucun invité soit présent à la fois sur deux tables différentes. Pour cela, nous allons énumérer de façon implicite toutes les combinaisons de tables possibles dans un arbre de recherche. L'idée est d'imaginer la construction d'une combinaison comme le parcours d'une branche spécifique de cette arbre et qu'à chaque étape de la construction on vérifie si les deux conditions du début soient respectées. Si ce n'est pas le cas, alors on sait que cette branche ne mènera pas à une combinaison de tables éligible et on peut donc arrêter de la parcourir. Pour modéliser tout cela, on va créer une structure `abr` qui représentera les nœuds de l'arbre de recherche :

— `int weight`, un entier représentant le nombre de tables validés dans la construction de la combinaison. Celui-ci nous permettra de choisir la combinaison la plus optimale.
— `vector<int> table`, la table du nœud,
— `vector<abr*> under_table`, les nœuds fils (les prochaines combinaisons à tester dans la branche actuelle),
— `abr* before_table`, le nœud père,
— `vector<int> current_guests`, un tableau de suivi permettant de suivre les invités déjà attablés. Celui-ci nous permettra d'arrêter la construction d'une branche.

Pour initialiser l'arbre de recherche, on va créer des nœuds pour chacune des tables obtenues de cette manière :

— `int weight` $= 1$, car on a une table d'ajoutée dans la construction de la combinaison,
— `vector<int> table` sera la table avec laquelle on va créer ce nœud,
— `vector<abr*> under_table`, les nœuds fils seront toutes les autres tables sauf celle du nœud actuel. Les nœuds fils devront aussi satisfaire les conditions citées auparavant.
— `abr* before_table` sera le nœud père, qui sera un nœud initialisé à part avec un poids nul, un vecteur `table` nul et un nœud père pointant vers `NULL`,
— `vector<int> current_guests` équivaudra finalement à `table`.

Après l'initialisation des premiers nœuds fils, on peut construire l'arbre de recherche de façon récursive en appliquant pour chaque nœud :

1. Trouver les invités manquant à la liste `current_guests` du nœud parcouru. Si aucun ne l'est, nous avons alors trouver une combinaison de tables. La fonction s'arrête ici.
2. Filtrer les tables obtenues à la partie A de telle sorte qu'on ait uniquement celles ne contenant que les invités manquants (pour réduire les effets combinatoires on peut exclure les tables avec une taille supérieure à celle de la table du dit nœud). Si nous n'obtenons aucune table après le tri, alors la branche actuelle parcourue n'est pas solution. La fonction s'arrête ici.
3. Créer des nœuds fils avec les tables obtenues à l'étape 2 : avec un poids incrémenter de 1 par rapport à celui du nœud père (du nœud parcouru donc) et avec `current_guests` mis à jour avec la table avec laquelle le nœud fils a été crée.
4. Relancer la fonction avec les nœuds fils.

Pour obtenir maintenant les combinaisons de tables, il suffit de lancer une fonction récursive depuis la racine pour obtenir les feuilles des branches contenant une solution. Pour cela, il suffit de vérifier la taille de `current_guests` pour chacun des nœuds parcourus. Si celle-ci équivaut à n, le nombre d'invités, alors on sait que nous venons de parcourir une branche solution et que nous sommes sur une feuille. Il suffit après d'enregistrer toutes ces feuilles dans une liste. Pour obtenir la combinaison la plus optimale, il suffit de prendre la feuille avec le poids (variable `weight`) le plus faible et remonter sa branche pour obtenir les tables.

Finalement, voici ce que chacune des fonctions implémentent :

— `recur_enum` permet de construire l'arbre de recherche de façon récursive en suivant l'algorithme énoncé auparavant,
— `get_last_nodes` permet d'obtenir toutes les feuilles des branches solutions de l'arbre.
— Finalement, `enum_brute` initialise la racine, les premiers nœuds fils puis construit l'arbre avec l'aide de `recur_enum`. De plus, elle s'occupe de l'affichage de la combinaison optimale avec `get_last_nodes`.

Algorithm 2: recur_enum(node,tables,all_guests)

```
Input  : - pTrnode = pointeur vers le noeud actuel parcouru
         - tables = liste de liste contenant toutes les tables
         - all_guests = [1,2,...,n]
Output: void
Pseudo-code :
diff ← all_guests - pTrnode→current_guests                // Etape 1
if diff vide then
    return
end

checkedTable ← Filtrer(tables,diff)                        // Etape 2

if checkedTable vide then
    return
end

current                                                    // liste vide

// Etape 3
for tab in checkedTable do
    current ← pTrnode→current_guests + tab
    newNode            // abr * (pointeur vers un noeud abr)
    newNode→current_guests ← current
    newNode→table ← tab
    newNode→before_table ← pTrnode
    newNode→weight ← pTrnode→weight + 1
    pTrnode→under_table.pushback(newNode)
    current.clear()                                        // on vide current
end

// Etape 4
for noeudFils in pTrnode→under_table do
    recur_enum(noeudFils,tables,all_guests)
end
```

Algorithm 3: enum_brute(tables,n)

```
Input   : - tables = liste de liste contenant toutes les tables
          - n, le nombre d'invités
Output: Affiche la combinaison de table optimale
Pseudo-code :
temp, all_guests                                // Deux listes vides

for i = 0; i < n; i++ do
    all_guests.pushback(i+1)
    temp.pushback(0)
end

root                    // abr * (pointeur vers un noeud abr)
root→current_guests ← temp
root→table ← temp
root→before_table ← NULL
root→weight ← 0

temp.clear()

   // On peut ignorer les tables singletons, à l'exception
 d'une pour réduire les effets combinatoires
for tab in tables do
    temp ← tab
    newNode             // abr * (pointeur vers un noeud abr)
    newNode→current_guests ← temp
    newNode→table ← temp
    newNode→before_table ← root
    newNode→weight ← 1
    root→under_table.pushback(newNode)
end

for noeudFils in root→under_table do
    recur_enum(noeudFils,tables,all_guests)
end
last_nodes                        // liste de pointeur abr vide
get_last_nodes(root,last_nodes,n)
// on prend la feuille avec le poids le plus petit
OptiLeaf ← minWeight(last_nodes)
// on affiche les tables composant cette branche
Afficher(OptiLeaf)
```

2.2 Question 2

Au delà des quelques boucles for présentes dans chacun de ces algorithmes de complexité de taille n ou bien du nombre de tables possibles obtenues à la partie A, il est très compliqué d'avoir une idée précise de la complexité de ces algorithmes car ils dépendent profondément de la configuration des invités, qui est totalement aléatoire. Cependant, au vu des nombreux appels récursifs, la complexité de cet algorithme d'énumération brute est au moins exponentielle ($\mathcal{O}(2^n)$) et même plus au vu des nombreux nœuds fils crées à chaque appel récursif.

3 Partie C : Optimisation dynamique

3.1 Question 1

L'idée ici, est de partager notre problème en multiples sous-petits problèmes et où la résolution de chacun d'eux participe non pas qu'à la résolution du problème général mais contribue aussi à celle des sous-problèmes. C'est à l'instar de la méthode précédente où la résolution d'une branche ne contribuait pas à celle des autres branches car on n'enregistrait en aucun cas le résultat de chacune des résolutions, mais aussi, on avait aucun moyen de contrôle et de comparaison entre chacune des branches afin d'anticiper si l'une d'elle menait à une combinaison moins optimale qu'une déjà trouvée auparavant.

Pour palier à tout cela, on va utiliser ce qu'on appelle la "mémoïsation", une caractéristique très intéressante de l'optimisation dynamique, mettant en cache les résultats obtenus. La première étape est donc de construire un tableau booléen de taille $M \times M$ avec M le nombre de tables possibles obtenues à la partie A. L'idée générale est de placer sur chacune des colonnes et des lignes les tables et de les comparer afin de savoir si elles peuvent être ensemble dans une combinaison (en vérifiant les mêmes conditions que ceux de la partie B). Si ce n'est pas le cas, alors la case correspondante à cette combinaison aura pour valeur 1 (donc `true` car il y des invités présents simultanément dans les tables comparées ou bien si la somme des deux tailles des tables est strictement supérieure à n), 0 sinon (donc `false`). Le tableau sera au début initialisé à 2, indiquant une case qui n'a pas encore été traité, et sa diagonale à 1 car une table ne peut pas être associée avec elle-même dans une combinaison.

Pour pouvoir comparer deux tables, on va tout d'abord créer une structure `table` :

— `vector<int> guests`, correspond aux invités assis à cette table,
— `int size`, est un entier correspondant à la capacité de la dite table.

Cette structure nous permettra d'avoir accès rapidement à la composition d'une table mais aussi à sa capacité. Pour initialiser et stocker chacune des tables initialement en forme de liste d'entier en struct `table`, on utilisera la fonction `ini_table` qui retournera une liste de struct `table`. De plus, pour savoir si deux tables peuvent être associées, on utilisera la fonction `IfDuplicates` qui retournera 1 si les deux tables ont un invité en commun, 0 sinon. On peut maintenant passer à la construction de la table de mémoïsation avec la fonction `dynamic_prog` :

Algorithm 4: dynamic_prog(l_tab,n)

```
Input  : - l_tab = liste de struct table obtenue avec l_tab ini_table
         - n, le nombre d'invité
Output: Retourne la table de mémoïsation liée à list_tables
Pseudo-code :
size ← l_tab.size()                               // Taille M
DP[size][size] ← 2          // Table 2D de mémoïsation ini. à 2
for i = 0; i < size; i++ do
    D[i][i] ← 1                                   // Diagonale à 1
end
for i = 0; i < size; i++ do
    for j = 0; j < size; j++ do
        if DP[i][j] == 2 then
            if DP[j][i] != 2 then
                DP[i][j] ← DP[j][i]
            else
                if l_tab[i].size + l_tab[j].size ≤ n then
                    DP[i][j] ← IfDup(l_tab[i].guests,l_tab[j].guests)
                else
                    DP[i][j] ← 1
                end
            end
        end
    end
end
return DP
```

Maintenant que nous avons notre tableau de mémoïsation, comment pouvons-nous en tirer la composition de table la plus optimale ? Premièrement, on partira du fait qu'une telle combinaison aura **le plus souvent** une composition avec des tables de grandes tailles car celles-ci permettent d'inclure le plus grand nombre d'invités. Ceci dit, il suffit d'effectuer une lecture ligne par ligne en commençant par les tables avec les plus grandes tailles puis colonne par colonne lorsque nous sommes fixés à une ligne. En effet, on sait qu'une ligne correspond à la relation d'une table spécifique avec les autres tables et en effectuant donc une lecture de cette ligne de gauche à droite (ici par colonne donc) on peut construire une combinaison de cette dite table :

— Imaginons que nous sommes à la ligne i, nous sommes donc en train de construire une combinaison avec la table i+1 initialement,
— En effectuant maintenant une lecture colonne par colonne de la ligne i, si la case $[i][j]$ vaut 0, alors on sait que la table j+1 peut être associée avec la table i+1,
— Avant d'ajouter la table j+1 dans la construction de la combinaison actuelle, il faut effectuer une vérification avec les autres tables présentes déjà dans la combinaison. Par exemple, si la combinaison en construction équivaut à $C = [i, h, k, m]$ alors les cases $[h][j], [k][j]$ et $[m][j]$ doivent valoir aussi 0. Et ici, nous n'avons pas de calcul supplémentaire à faire car nous avons le tableau de mémoïsation.

De plus, comme nous n'effectuons aucun appel récursive, nous avons une meilleure flexibilité. Par exemple, on peut ajouter un cache stockant la dernière table ajoutée lors de la construction, ce qui peut nous éviter d'effectuer des vérifications inutiles lors de l'ajout d'une table : si on veut ajouter la table j+1 à $C = [i, h, k, m]$ alors avant d'initier la vérification entière de toute la combinaison il faut que $[i][j]$ et $[m][j]$ soient nulles. Nous pouvons aussi stocker dans une autre variable cache la cardinalité de C, et si lors de la construction d'une autre combinaison C_1, sa cardinalité est supérieure à celle de C alors on peut arrêter la construction de C_1 car on sait que celle-ci nous ne donnera pas une meilleure combinaison. Et au contraire, si on a une cardinalité inférieure alors on met à jour le cache.

Néanmoins, on peut se demander si cette méthode couvre bien toutes les combinaisons ? Oui, car si une table j+1 n'a pas eu la chance d'être combinée avec une table i+1, cette dernière le sera quand on parcourra les colonnes de la ligne j du tableau de mémoïsation.

Algorithm 5: get_best_comp(l_tab,n)

```
Input  : - DP = le tableau de mémoïsation
         - l_tab = liste de struct table
         - n, le nombre d'invité
Output: Retourne la combinaison de table optimale
Pseudo-code :
res                                                     // liste vide
resCache                                                // liste vide
size ← DP.size()
cache, sum, test                                        // des entiers
for i = size − 1; i ≥ n − 1; i − − do
    res.pushback(i)
    cache ← i
    sum ← l_tab[i].size
    for j = size − 1; j ≥ 0; j − − do
        if i != size - 1 then
            if sum == n and resCache.size() > res.size() then
                resCache ← res
                break
            else if resCache.size() < res.size() then
                break
            else if DP[i][j] == 0 and DP[j][cache] == 0 then
                test ← 0
                for k = 0; k < res.size(); k + + do
                    if DP[j][res[k]] == 1 then
                        test ← 1
                        break
                    end
                end
                if !test then
                    cache ← j
                    res.pushback(j)
                    sum ← sum + l_tab[j].size
                end
            else
                pass
            end
        else
            if DP[i][j] == 0 and DP[j][cache] == 0 then
                test ← 0
                for k = 0; k < res.size(); k + + do
                    if DP[j][res[k]] == 1 then
                        test ← 1
                        break
                    end
                end
                if !test then
                    cache ← j
                    res.pushback(j)
                    sum ← sum + l_tab[j].size
                end
            end
            if sum == n then
                resCache ← res
                break
            end
        end
    end
    res.clear()
end
return resCache
```

3.2 Question 2

La complexité de ces deux algorithmes est ($\mathcal{O}(M^2)$) avec M le nombre de tables possibles obtenues à la partie A car dans le premier cas, on effectue deux boucles for M fois pour construire le tableau de mémoïsation. De même pour le second cas, on parcourt celui-ci en entier pour le lire. Bien plus, on omet les complexités des appels de la fonction de `IfDuplicates` pour le premier cas et pour le second cas, la boucle for de vérification à la ligne 36 car elles sont inhérentes au hasard de la configuration des invités.

3.3 Question 3

Pour l'optimisation dynamique, nous avons une complexité polynomiale quant à l'énumération, une complexité exponentielle. L'optimisation dynamique performera bien mieux en terme de temps d'exécution.

4 Partie D : Mise-en-œuvre informatique

4.1 Question 1

La mise-en-œuvre informatique a été faite en C++ dû à la présence de librairies plus complètes par rapport au C et pour sa rapidité d'exécution du fait qu'il est compilé et non interprété (à l'instar de Python). Pour générer de façon aléatoire les différentes instances, on a utilisé un générateur Mersenne Twister couplé avec une distribution uniforme. Voici les étapes :

- On définit `ratio`, un float choisi par l'utilisateur compris entre $[0, 1]$ permettant de donner le nombre d'invités initialisés avec des ennemis, i.e, si $ratio = 0.5$, alors la moitié des invités seront initialisés avec des ennemis,
- Les invités sont choisi de façon aléatoire pour l'attribution des ennemis,
- Le nombre d'ennemi par invité est aussi choisi de façon aléatoire,
- Finalement, on finit par attribuer les ennemis manquants, car si u est ennemi de v alors ce dernier l'est aussi par rapport à u.

4.2 Question 2

Voici le temps d'exécution de chacun des algorithmes en secondes (avec n le nombre d'invités et M celui des tables possibles obtenues à la partie A). Par ailleurs, le temps d'exécution comprend pour les deux dernières parties l'affichage des résultats ce qui n'est pas le cas pour la partie A :

n	**M**	**Par. A**	**Par. B**	**Par. C**
4	15	0.000064	0.0.000493	0.000066
6	47	0.000266	0.041202	0.000480
8	135	0.001002	+INF	0.003498
10	267	0.003320	+INF	0.015909
12	1028	0.017598	+INF	0.259653

Table 1 – Ratio : 0.1

n	**M**	**Par. A**	**Par. B**	**Par. C**
4	7	0.000044	0.000167	0.000042
6	20	0.000138	0.008654	0.000156
8	51	0.000434	1.110856	0.000636
10	57	0.000927	70.678970	0.000760
12	123	0.001818	+INF	0.003635

Table 2 – Ratio : 0.5

n	**M**	**Par. A**	**Par. B**	**Par. C**
4	5	0.000089	0.000066	0.00003
6	10	0.000250	0.002486	0.000053
8	24	0.000465	0.358019	0.000297
10	33	0.000465	35.129760	0.001330
12	72	0.001115	+INF	0.259653

Table 3 – Ratio : 0.9

n	M	Par. A	Par. C
16	326	0.0085994	0.025754
18	750	0.0175368	0.140184
20	3164	0.063331	2.803785
22	8881	0.184563	28.980799
24	13810	0.385463	65.168031
26	15488	0.508579	82.884276

TABLE 4 – Ratio : 0.5 - Uniquement Part. A et Part. B avec n plus élevé

r = 0.1 n = 24 m = 4325423 98.429449 sec	r = 0.9 n = 50 m = 30376 3.213119 sec

TABLE 5 – Impact du ratio sur Part. A

La première chose qu'on remarque est l'efficacité de l'algorithme basé sur la programmation dynamique comparé à celui de l'énumération brute et ce, quel que soit le ratio, le nombre d'invités ou bien de tables disponibles. Celui-ci résout instantanément le problème combinatoire dès lorsque la fonction `back_tracking_comb` termine. De plus, celui-ci est capable de résoudre pour des M élevés en un temps acceptable (tableau 4) alors que l'algorithme d'énumération brute sature déjà à $M \approx 70$ (tableau 2).

Avec les tableaux 1, 2 et 3, on remarque qu'en dépit différentes valeurs de ratio, la fonction `back_tracking_comb` n'est pas le maillon limitant la résolution mais bien les autres algorithmes car on obtient les bonnes tables dans un temps toujours inférieur à celui de l'exécution des deux autres parties. Bien plus, cela se voit plus particulièrement avec le tableau 4 là où la programmation dynamique met 83 secondes avant de résoudre la combinaison alors que l'algorithme de backtracking met uniquement une demi-seconde pour trouver les bonnes tables.

Étudions maintenant l'impact du paramètre `ratio` sur le temps de résolution. On remarque, comme évoqué dans la partie A, quand celui-ci est proche de 0 alors on obtient un M plus élevé et cela impacte directement le temps d'exécution de chacune des fonctions (voir tableau 1 où l'énumération implicite sature déjà à n = 8 alors qu'aux tableaux 2 et 3 elle l'est à n = 10). Et au contraire, lorsque le ratio est proche de 1, la configuration des invités devient plus complexe, i.e que tout le monde est ennemi de tout le monde, ce qui résulte en un nombre M inférieur et un temps d'exécution plus faible et ce malgré un n plus élevé (voir tableau 5).

Finalement, il est très difficile d'avoir une idée précise de la complexité de chacun de ces algorithmes car comme nous l'avons vu celle-ci dépend de la configuration des invités et de la valeur du ratio qui sont totalement aléatoires. Néanmoins, on voit bien que l'énumération brute explose très rapidement à partir d'un très petit M, synonyme d'un comportement exponentiel là où la programmation dynamique met plus de temps mais finit éventuellement par exploser aussi (d'où une complexité de $(\mathcal{O}(M^2))$).

4.3 Question 3

Pour réduire les effets combinatoires durant l'énumération brute, il suffit de fournir à chaque nœud crée uniquement des nœuds fils ayant des tables de tailles inférieures ou égales à celle de sa table. Cela permet d'éviter de parcourir des branches qui l'ont été déjà. De plus, on peut exclure les tables singletons $([1], [2]...)$ à l'exception d'une durant l'initialisation des premiers nœuds fils dans `enum_brut`.

Concernant la partie C, il n'y a vraiment peu d'optimisation à faire mais lors de la construction du tableau de mémoïsation, on peut utiliser les résultats obtenus auparavant car si une table i+1 n'est pas compatible avec une table j+1 alors c'est aussi le cas dans l'autre sens donc DP[i][j] = DP[j][i]. Ainsi, il suffit de remplir la moitié du tableau pour obtenir l'autre et c'est là qu'on remarque l'intérêt de l'optimisation dynamique évoqué précédemment : la résolution d'un sous-problème participe aussi à celle de l'ensemble qu'il soit général ou subalterne.