Le jeu de Röckse

Corrigé UPSTI

PARTIE I : SAUTS ET CHEMINS

Question 1 Ecrire une fonction `poids(T, chemin)` qui, étant donné un plateau de jeu `T` et un chemin `chemin`, renvoie le poids de ce chemin. Donner la complexité de cette fonction.

```
def poids(T, chemin) :
    p = 0                   # initialisation
    for i, j in chemin :
        p += T[i][j]        # ajout de la pénalité
    return p
```

Complexité : soit n la longueur du chemin. Il y a exactement n itérations, et à chaque itération, les opérations réalisées ont une complexité en $O(1)$. On en déduit que la complexité totale est en $O(n)$.

Question 2 Ecrire une fonction `appliquer_sauts(i, j, sauts)` qui, étant donné une case (i, j) et une liste de sauts `sauts`, applique les sauts dans l'ordre donné par la liste `sauts`, en partant de la case (i, j) et renvoie la case atteinte. On suppose que le chemin indiqué par `sauts` reste dans la grille.

```
def appliquer_sauts(i, j, sauts) :
    for saut in sauts :
        i += saut[0]        # MAJ de la ligne
        j += saut[1]        # et de la colonne
    return (i, j)
```

Question 3 Ecrire une fonction `sauts_corrects(sauts, bonus, chemin)` qui, étant donné l'ensemble des sauts par défaut représenté par la liste `sauts`, les sauts associés aux cases bonus `bonus` et un chemin `chemin`, renvoie `True` si les sauts utilisés dans le chemin sont corrects et `False` sinon. On suppose que le chemin reste dans la grille.

```
def sauts_corrects(sauts, bonus, chemin) :
    for c in range(len(chemin)-1):              # déterminer les sauts dans chemin
        case, case_suivante = chemin[c], chemin[c+1]
        di, dj = case_suivante[0] - case[0], case_suivante[1] - case[1]
        if case in bonus:                       # modifier sauts si passage par une case bonus
            sauts += bonus[case]
        if (di, dj) not in sauts:               # si saut dans chemin n'est pas correct
            return False
    return True                                 # tous les sauts dans chemin sont corrects
```

Comme on concatène `sauts` à `bonus[(i, j)]` puis on l'affecte à `sauts`, on ne modifie pas la liste `sauts` originale.

Question 4 Ecrire une fonction `sauts_bien_formes(sauts, bonus)` qui, étant donné la liste des sauts par défaut `sauts` et les sauts associés aux cases bonus `bonus`, vérifie que chaque saut de ces listes satisfait la condition (*). La fonction renvoie `True` si c'est le cas et `False` sinon.

```
def sauts_bien_formes(sauts, bonus):
    for case in bonus:
        sauts += bonus[case]
    for (di, dj) in sauts:
        if not (di > 0 or (di == 0 and dj > 0)):
            return False
    return True
```

PARTIE II : RECHERCHE EXHAUSTIVE

Question 5 Quelle est la longueur maximale L d'un chemin de (0, 0) à (N-1, N-1) pour n'importe quel ensemble de sauts satisfaisant (*) ? Donner un exemple d'ensemble de sauts par défaut pour lequel se réalise ce chemin de longueur L.

Un chemin est de longueur maximale s'il utilise le plus de sauts l'éloignant de sa cible, donc ici ↙. Un exemple serait un chemin qui emprunterait la première ligne jusqu'à la case $(0, N-1)$ avec $N-1$ sauts →, puis la diagonale jusqu'à la case $(N-1, 0)$ avec $N-1$ sauts ↙, puis la dernière ligne jusqu'à la case $(N-1, N-1)$ avec $N-1$ sauts →.

Cela correspond à une longueur maximale de $L = 3(N-1)$.

Question 6 Ecrire la fonction auxiliaire `trouve_complet_rec(T, sauts, bonus, sauts_max, i, j)` et la fonction principale `trouve_complet(T, sauts, bonus, sauts_max)` qui renvoie le couple `(poids, sauts)` où `poids` est le poids du chemin trouvé et `sauts` est la liste des sauts du chemin trouvé. Quelle est la complexité de cette fonction ?

```
def trouve_complet_rec(T, sauts, bonus, sauts_max, i, j) :
    N = len(T)
    if sauts_max==0 or (i==N-1 and j==N-1):     # condition d'arrêt
        return (T[i][j], [])
    if (i,j) in bonus :                         # ajout des bonus dans sauts
        sauts += bonus[(i,j)]
    poids_min = INFINI                          # initilisation poids
    sauts_min = []                              # initialisation liste des sauts
    for (di, dj) in sauts:                      # pour chaque saut possible
        ni, nj = i + di, j + dj                 # nouvelle position
        if ni < N and 0 <= nj < N :             # si le saut permet de rester dans la grille
            poids_rec, sauts_rec = trouve_complet_rec(T, sauts, bonus, sauts_max-1, ni, nj)
            if poids_rec + T[i][j] < poids_min: # MAJ du meilleur poids et des sauts associées
                poids_min = poids_rec + T[i][j]
                sauts_min = [(di,dj)] + sauts_rec
    if poids_min == INFINI :                    # aucun saut ne permet de rester dans la grille
        return (T[i][j], [])
    return (poids_min, sauts_min)


def trouve_complet(T, sauts, bonus, sauts_max) :
      # Recherche du chemin optimal partant de (0, 0)
      return trouve_complet_rec(T, sauts, bonus, sauts_max, 0, 0)
```

La complexité de la deuxième fonction est la complexité de la première fonction appliquée à i=0 et j=0. Calculons la complexité dans le pire cas de la première fonction, en posant $n = sauts\_max$ et m =nombre total de sauts pour alléger les notations. On aura la relation de récurrence suivante :

$$C(m,n) = O(1) + O(m) + O(m \times C(n-1))$$

$$= O(m \times C(n-1))$$

Sachant que $C(0)$ est une constante, alors on peut écrire :

$$C(m,n) = O(m^n)$$

Question 7 La fonction `trouve_complet_rec` perd beaucoup de temps à refaire les mêmes calculs. On peut l'améliorer en enregistrant les résultats déjà calculés pour une valeur fixée de `sauts_max`. Expliquer en quelques lignes comment procéder. Quelle serait alors la complexité ?

Après chaque calcul, on enregistre chaque poids optimal et le chemin correspondant dans une structure (souvent un dictionnaire). De plus, si la longueur du chemin optimal est plus petit que sauts_max, on peut l'enregistrer pur

des valeurs plus petites que sauts_max. Avant de faire un nouveau calcul, on vérifie dans la structure si le résultat est déjà mémorisé ; si c'est le cas, on le réutilise directement. C'est la mémoïsation.

Il y aurait alors au plus $N \times N \times (n+1)$ appels différents à la fonction (où n =sauts_max) (une par case de départ dans la grille et par nombre de sauts restants), et pour chaque appel, on parcourt au plus m sauts possibles (où m = nombre total de sauts) : la complexité devient donc $C(m,n) = O(N^2 \times m \times n)$. Il s'agit d'une complexité polynomiale, ce qui présente une nette amélioration par rapport à la complexité exponentielle de la recherche exhaustive.

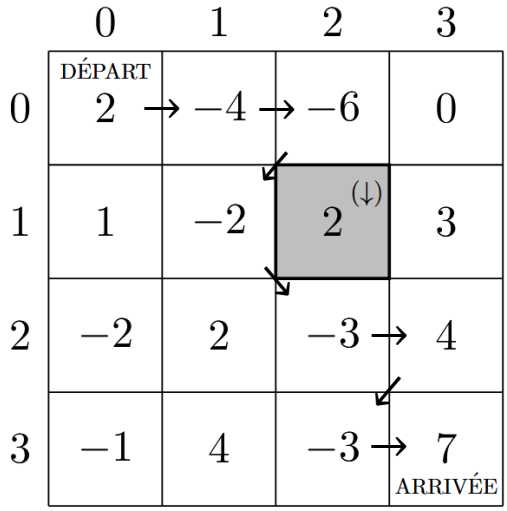
PARTIE III : RECHERCHE GLOUTONNE

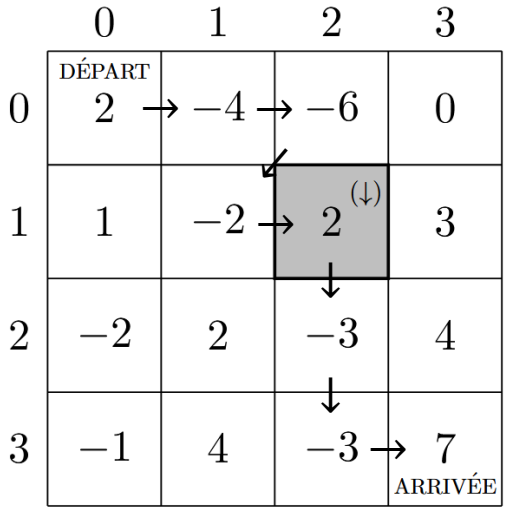
Question 8 Sur l'exemple donné en introduction, quel est le chemin trouvé pour k = 2 ? Est-il optimal ? Mêmes questions pour k = 3. En augmentant k, obtient-on forcément un chemin de poids plus petit ?

- Pour k = 2 : le chemin trouvé est donné sur la figure de gauche, ci-dessous. Un autre chemin possible emprunterait le détour par la case (3, 1) plutôt que (2, 3) car cela ne change pas le poids. Le poids de ce chemin est -5, ce qui n'est pas optimal.

- Pour k = 3 : le chemin trouvé est donné sur la figure de droite, ci-dessous. Le poids de ce chemin est-7, ce qui est optimal.

Si on augmente k, on trouvera forcément un chemin de poids plus petit car l'algorithme glouton se transforme alors en recherche exhaustive.





Question 9 Ecrire une fonction `trouve_glouton(T, sauts, bonus, k)` qui, étant donnés la grille de jeu `T`, la liste des sauts par défaut `sauts`, les sauts associés aux cases bonus `bonus` et l'horizon `k`, effectue cette recherche gloutonne et renvoie le couple `(poids, sauts)` où `poids` est le poids du chemin trouvé et `sauts` est la liste des sauts du chemin trouvé. Quand la recherche exhaustive avec l'horizon k ne trouve pas de suite locale, la fonction renvoie `(INFINI, [])`.

```
def trouve_glouton(T, sauts, bonus, k) :
    N = len(T)
    i, j = 0, 0                          # initialisation
    poids, sauts = 0, []
    while (i, j) != (N-1,  N-1) : # tant qu'on n'est pas arrivé
        p, s = trouve_complet_rec(T, sauts, bonus, k, i, j)
        if len(s) == 0 :                 # aucun saut possible
            return (INFINI, [])
        poids += p                       # MAJ du poids
        sauts += s                       # MAJ des sauts
        i, j = appliquer_sauts(i, j, s) # MAJ de la case
        if (i, j) in bonus :             # MAJ des sauts
            sauts += bonus[(i, j)]
```

```
    return (poids, sauts)
```

PARTIE IV : RECHERCHE PAR PROGRAMMATION DYNAMIQUE

I - Encodage des cases bonus activées

Question 10 Ecrire la fonction `code_bonus(masque_bonus)` qui renvoie le code associé au masque binaire `masque_bonus` représentant l'ensemble des cases bonus activées.

```
def code_bonus(masque_bonus) :
    c = 0                                # initialisation
    for k in range(len(masque_bonus)) :
        if masque_bonus[k] :             # si b_k est True
            c += 2**k
    return c
```

II - Récurrence

Question 11 Ecrire la définition de `poids_opt[i][j][<`b_0`…  `b_{n-1}`>]`. On notera Γ l'ensemble des sauts par défaut et Δ_k l'ensemble des sauts associé à la k-ième case bonus.

- Si (i, j) = (N-1, N-1) :

$$poids\_opt[N-1][N-1][<b_0 \dots b_{n-1}>] = T[N-1][N-1]$$

- Sinon, si (i, j) n'est pas une case bonus :

$$poids\_opt[i][j][<b_0 \dots b_{n-1}>] = T[i][j] + \min\left\{ poids\_opt[i+\delta i][j+\delta j][<b_0 \dots b_{n-1}>] \,/\, (\delta i, \delta j) \in \Gamma \cup \bigcup_{\substack{\ell=0 \\ tel\ que\ b_\ell \neq 0}}^{n-1} \Delta_\ell \right\}$$

- Sinon (i.e. (i, j) est une case bonus) : notons k le numéro de cette case bonus. Alors :

$$poids\_opt[i][j][<b_0 \dots b_{n-1}>] = T[i][j] + \min\left\{ poids\_opt[i+\delta i][j+\delta j][<b_0 \dots b_{n-1}>] \,/\, (\delta i, \delta j) \in \Gamma \cup \bigcup_{\substack{\ell=0 \\ tel\ que\ b_\ell \neq 0}}^{n-1} \Delta_\ell \cup \Delta_k \right\}$$

III - Algorithme

Question 12 Ecrire une fonction `combinaisons_bonus(nb_bonus)` qui, étant donné le nombre total de cases bonus `nb_bonus`, renvoie la liste de masques bonus ordonnée de manière décroissante dans le sens de l'inclusion, en codant l'algorithme ci-dessus.

```
def combinaisons_bonus(nb_bonus) :
    liste_masques = [[True for i in range(nb_bonus)]]# initialisation
    file = liste_masques[:]                          # file des choix possibles
    while len(file) != 0 :                           # tant qu'il y a des choix possibles
        choix = file.pop(0)                          # on prend le premier entré
        for k in range(nb_bonus) :                   # on parcourt ses b_k
            if choix[k] :                            # si b_k est égal à True
                choix[k] = False                     # on le met à False
                if choix not in file :               # si le nouveau masque n'a pas encore été vu
                    file.append(choix)               # on l'enregistre
                    liste_masques.append(choix)
                choix[k] = True                      # on remet b_k à True pour ne pas affecter les
                                                     prochains choix
```

```
    return liste_masques
```

Question 13 Ecrire une fonction `trouver_sauts_possibles(sauts, bonus_au_rang, masque_bonus)` qui, étant donnés les sauts par défaut `sauts`, la liste `bonus_au_rang` renvoyée par `ranger_bonus(bonus)` et le masque des bonus activés `masque_bonus`, renvoie l'ensemble des sauts possibles.

```
def trouver_sauts_possibles(sauts, bonus_au_rang, masque_bonus) :
    nb_bonus = len(masque_bonus)                    # nombre de cases bonus
    for k in range(nb_bonus) :                      # on parcourt l'ensemble des cases bonus
        if masque_bonus[k] :                        # si la case bonus n°k est activée
            sauts += bonus_au_rang[k]               # on ajoute les sauts associés
    return sauts
```

Question 14 Le code Python incomplet de la page suivante implémente la fonction `trouve_dynamique(T, sauts, bonus)` qui, étant donnés une grille de jeu `T`, la liste des sauts par défaut `sauts` et les sauts associés aux cases bonus `bonus`, calcule le chemin optimal par programmation dynamique en utilisant la récurrence trouvée en question 11. Le résultat de la fonction est le couple `(poids_opt, sauts_opt)` où `poids_opt` est le poids du chemin optimal trouvé et `sauts_opt` est la liste des sauts de ce chemin.

1. Donner les sept parties manquantes indiquées par « … » dans le code.

2. Quelle est la complexité de cette fonction ?

1. On complète les différentes parties :

- Partie 1 : le nombre de codes bonus possibles, sachant que chaque code bonus est un mot binaire constitué de n bits, est donc de 2^n. Ainsi, on écrit :

```
nb_code_bonus = 2 ** nb_bonus
```

Remarque : on pourrait aussi écrire l'instruction suivante, mais sa complexité est en $O(n)$ au lieu de $O(1)$:

```
nb_code_bonus = len(combinaisons_bonus(nb_bonus))
```

- Partie 2 : le poids du chemin optimal partant de (N-1, N-1) (et arrivant sur (N-1, N-1)) est tout simplement la pénalité dans cette case.

```
poids_opt[N-1][N-1][code_bonus_actifs] = T[N-1][N-1]
```

- Partie 3 : on cherche tous les sauts possibles pour ce code bonus.

```
sauts_possibles = trouver_sauts_possibles(sauts, bonus_au_rang, code_bonus_actifs)
```

- Parties 4 & 5 : on doit itérer « à l'envers » sur les cases car on a d'abord besoin des poids des successeurs :

```
for i in range(N-1, -1, -1) :
    for j in range(N-1, -1, -1) :
```

- Parties 6 & 7 : on met à jour le saut optimal et le poids optimal associés à ce code bonus et cette case de départ (i, j) :

```
poids_opt[i][j][code_bonus_actifs] = poids_opt_dest
saut_opt[i][j][code_bonus_actifs] = (delta_i, delta_j)
```

2. Complexité de la fonction : sachant qu'il y a 2^n codes bonus possibles, la complexité est en $O(2^n \times N^3)$ car il y a trois boucles imbriquées de `bonus_actifs` sur 2^n itérations, de `i` sur N itérations et de `j` sur N itérations, et à chaque itération, on doit aussi vérifier que `i_dest` et `j_dest` restent dans [0, N-1].

Si le nombre de sauts m est strictement supérieur à 2, alors la complexité de l'algorithme par programmation dynamique est bien meilleure que celle de la recherche exhaustive.

Question 15 Ecrire la fonction `solution_dynamique(saut_opt, N)` qui, étant donnés la structure `saut_opt` calculée dans la question précédente et la dimension `N` de la grille, renvoie le chemin optimal correspondant. La fonction renvoie le chemin vide `[]` s'il n'existe pas de chemin entre la case de départ et celle d'arrivée.

Si les données associées à la case (0, 0) et au code bonus 0…0 (aucune case bonus activée, ce qui correspond au départ) ont été remplies dans la liste `saut_opt`, cela veut nécessairement dire qu'il y a en effet un chemin qui existe entre cette case et la case d'arrivée (N-1, N-1) puisque cette liste a été construite à partir de cette case d'arrivée.

Je ne comprends pas pourquoi cette fonction aurait besoin de N (on pourrait le récupérer avec la première dimension de sauts_opt) et pourquoi elle n'aurait pas besoin de bonus pour utiliser la fonction ranger_bonus… Je propose ci-dessous une version où la fonction prend en entrée la liste bonus :

```
def solution_dynamique(saut_opt, bonus) :
    N = len(saut_opt)                                   # récupère la taille de la grille
    rang_du_bonus = ranger_bonus(bonus)[1]              # récupère les numéros des cases bonus
    i, j, code_bonus = 0, 0, 0                          # initialisation
    chemin = [(0, 0)]
    while (i, j) != (N-1, N-1) :                        # tant qu'on n'est pas arrivé
        if (i, j) in bonus :                            # si c'est une case bonus
            k = rang_du_bonus[(i, j)]                   # on récupère son numéro
            code_bonus += 2 ** k                        # MAJ du code bonus
        if len(saut_opt[i][j][code_bonus]) == 3 :       # valeur initiale (0, 0, 0)
            return []                                   # alors pas de chemin du tout
        i, j = saut_opt[i][j][code_bonus]               # sinon, on récupère le saut optimal
        chemin.append((i, j))                           # MAJ du chemin
    return chemin
```

Une autre version, qui ne nécessiterait pas d'utiliser sauts_opt, requerrait d'enregistrer le numéro de l'éventuelle case bonus en tant que troisième élément du tuple… Mais il faudrait rajouter des lignes dans le code fourni dans le sujet pour implémenter cela.