I Sujet 0 CCP

On considère le programme suivant, ici en OCaml, dans lequel n fils d'exécution incrémentent tous un même compteur partagé.

```
let n = 100
let compteur = ref 0
let f i = compteur := !compteur + 1

(* Création de n fils exécutant f associant à chaque fil son numéro *)
let threads = Array.init n (fun i -> Thread.create f i)

(* Attente de la terminaison de tous les fils *)
Array.iter Thread.join threads
```

On rappelle que l'on dispose en OCaml des trois fonctions `Mutex.create : unit -> Mutex.t` pour la création d'un verrou, `Mutex.lock : Mutex.t -> unit` pour le verrouillage et `Mutex.unlock : Mutex.t -> unit` pour le déverrouillage, du module `Mutex` pour manipuler des verrous.

1. Quelles sont les valeurs possibles que peut prendre le compteur à la fin de ce programme?
2. Identifier la section critique et indiquer comment et à quel endroit ajouter des verrous pour garantir que la valeur du compteur à la fin du programme soit n de manière certaine.

Dans la suite de l'exercice, on suppose que l'on ne dispose pas d'une implémentation des verrous. On se limite au cas de deux fils d'exécution, numérotés 0 et 1. Nous cherchons à garantir deux propriétés :

- Exclusion mutuelle : un seul fil d'exécution à la fois ne peut se trouver dans la section critique;
- Absence de famine : tout fil d'exécution qui cherche à rentrer dans la section critique pourra le faire à un moment.

On utilise pour cela un tableau `veut_entrer` qui indique pour chaque fil d'exécution s'il souhaite entrer en section critique ainsi qu'une variable `tour` qui indique quel fil d'exécution peut effectivement entrer dans la section critique. On propose ci-dessous deux versions modifiées `f_a` et `f_b` de la fonction `f`, l'objectif étant de pouvoir exécuter `f_a 0` et `f_a 1` de manière concurrente, et de même pour `f_b`.

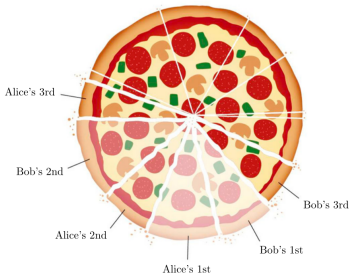
```
let veut_entrer = Array.make 2 false
let tour = ref 0

let f_a i =
    let autre = 1 - i in
    veut_entrer.(i) <- true;
    while veut_entrer.(autre) do () done;
    (* Section critique *)
    veut_entrer.(i) <- false

let f_b i =
    let autre = 1 - i in
    veut_entrer.(i) <- true;
    tour := i;
    while veut_entrer.(autre) && !tour = autre do () done;
    (* Section critique *)
    veut_entrer.(i) <- false
```

4. Expliquer pourquoi aucune de ces deux versions ne convient, en indiquant la propriété qui est violée.
5. Proposer une version `f_c` qui permet de garantir les deux propriétés. Il s'agit de l'algorithme de Peterson.
6. Connaissez-vous un algorithme permettant de généraliser à n fils d'exécution? Rappeler très succinctement son principe.

II Partage de pizza

Alice et Bob se partagent une pizza, qui est découpée en n parts. Chaque part possède un poids. Ils choisissent une part chacun leur tour, avec les règles suivantes :

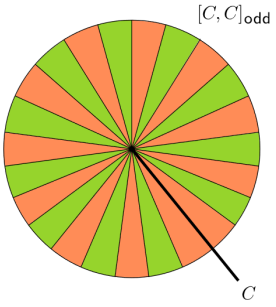
- Alice commence et choisit la part de son choix.
- Ensuite, ils alternent en choisissant à chaque fois une part adjacente à une part déjà choisie.



L'objectif pour chaque joueur est de manger le plus de pizza possible.

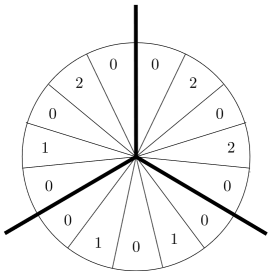
1. Combien y a t-il de configurations possibles pour ce jeu ?
2. Donner une stratégie gloutonne pour Alice et montrer qu'elle n'est pas optimale.
3. On suppose que n est pair. Montrer qu'Alice a une stratégie lui garantissant de manger au moins la moitié de la pizza.

Dans la suite, on suppose que n est impair.
Une coupe désigne la zone entre deux parts de pizza adjacente. Pour chaque coupe C, on définit un coloriage rouge/vert comme sur l'exemple suivant :



On appelle $R(C)$ les parts rouges suivant ce coloriage.

4. Montrer que Alice a une stratégie lui garantissant de manger toutes les parts de $R(C)$, pour un certain C.
5. En déduire qu'Alice a une stratégie lui garantissant de manger au moins $\frac{1}{3}$ de la pizza.
6. Montrer que Bob a une stratégie lui permettant de manger au moins $\frac{5}{9}$ de la pizza suivante (et donc que Alice peut manger au plus $\frac{4}{9}$ de la pizza), où on a indiqué le poids de chaque part :



III Diner des philosophes

Rappel sur l'utilisation de la bibliothèque `pthread` en C :

- `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER` : définit un mutex.
- `pthread_mutex_init(pthread_mutex_t*)` : initialise un mutex.
- `pthread_mutex_lock(pthread_mutex_t*)` : verrouille le mutex `m`.
- `pthread_mutex_unlock(pthread_mutex_t*)` : déverrouille le mutex `m`.
- `sem_t s` : définit un sémaphore.
- `sem_init(sem_t*, int)` : initialise un sémaphore avec un compteur.
- `sem_wait(sem_t*)` : décrémente un sémaphore. Si le compteur est nul, le thread est bloqué.
- `sem_post(sem_t*)` : incrémente un sémaphore et réveille éventuellement un thread bloqué.

On considère n philosophes répartis autour d'une table, chaque philosophe étant un thread. Pour manger, un philosophe a besoin des deux couverts adjacents. L'objectif est d'écrire un programme qui permette à chaque philosophe de manger, si possible en parallèle.

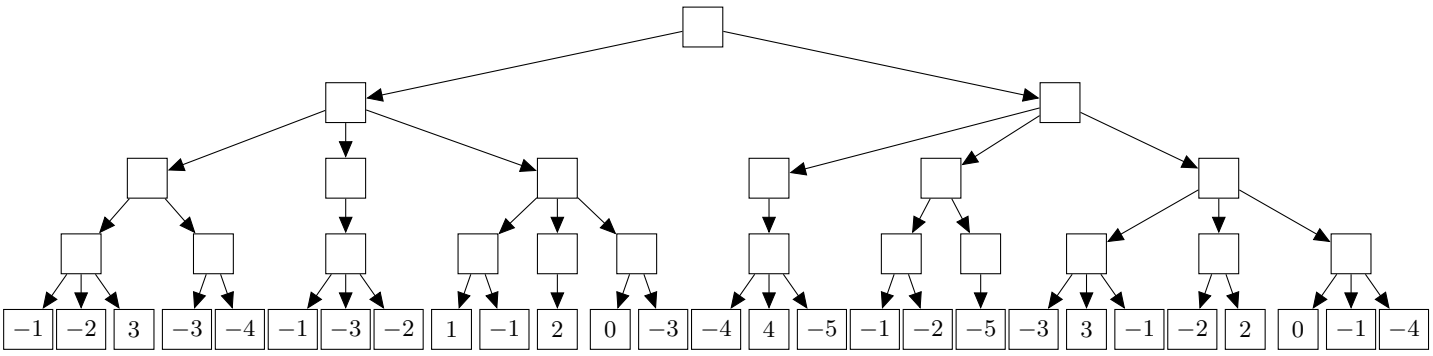
Décrire un algorithme en français pour résoudre ce problème, et l'implémenter en C.

IV Algorithme min-max

Déterminer le score de la racine étant donnée les valeurs de l'heuristique aux feuilles de l'arbre suivant :

1. en supposant que la racine est un état du joueur 1 ;
2. en supposant que la racine est un état du joueur 2.

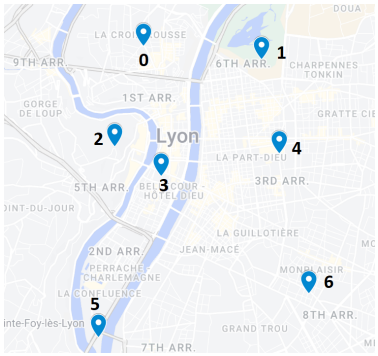
On appliquera l'élagage Alpha-Beta dans les deux cas.



V Algorithme de Christofides

On considère un graphe $G = (V, E)$ **non orienté**, **complet** et **pondéré** par une fonction $w : E \longrightarrow \mathbb{R}^+$. On supposera dans toute la suite que $V = \{0, ..., n-1\}$ où n est le nombre de sommets. Si G' est un graphe ou un ensemble d'arêtes, son **poids** $w(G')$ est la somme des poids des arêtes de G'.

On utilisera comme exemple le graphe G_{ex} suivant, représenté par matrice d'adjacence pondérée, dont les sommets sont des points d'intérêts de Lyon (numérotés de 0 à 6) et chaque arête correspond à une distance euclidienne entre les deux points :



$$G_{ex} = \begin{pmatrix} 0.0 & 2.3 & 1.75 & 2.2 & 2.9 & 4.8 & 5.1 \\ 2.3 & 0.0 & 2.9 & 2.6 & 1.5 & 5.4 & 4.0 \\ 1.75 & 2.9 & 0.0 & 0.9 & 2.9 & 3.2 & 4.2 \\ 2.2 & 2.6 & 0.9 & 0.0 & 2.1 & 2.9 & 3.2 \\ 2.9 & 1.5 & 2.9 & 2.1 & 0.0 & 4.4 & 2.5 \\ 4.8 & 5.4 & 3.2 & 2.9 & 4.4 & 0.0 & 3.7 \\ 5.1 & 4.0 & 4.2 & 3.2 & 2.5 & 3.7 & 0.0 \end{pmatrix}$$

On suppose que G est **métrique**, c'est-à-dire que w vérifie l'inégalité triangulaire : $\forall x, y, z \in V, \quad w(x, y) \leq w(x, z) + w(z, y)$

Un **cycle hamiltonien** dans un graphe est un cycle qui passe exactement une fois par chaque sommet. Dans la suite, on représente un cycle hamiltonien par sa liste de sommets, dans l'ordre (contenant donc chaque sommet exactement une fois).

Le **problème du voyageur de commerce (TSP)** consiste à trouver un cycle hamiltonien de poids minimum.

L'**algorithme de Christofides** donne une $\frac{3}{2}$-approximation du TSP. Voici son fonctionnement :

- Trouver un arbre couvrant de poids minimum T de G.
- Considérer V_1 l'ensemble des sommets de degré impair de T.
- Trouver un couplage parfait de poids minimum M de $G[V_1]$.
- Considérer le multigraphe H obtenu par union des arêtes de T et de M.
- Trouver un circuit eulérien C dans H.
- Transformer C en cycle hamiltonien en « sautant » les sommets répétés.

V.1 Recherche d'un arbre couvrant de poids minimum

1. Quel algorithme peut-on utiliser pour trouver un arbre couvrant de poids minimum dans un graphe ? L'appliquer sur G_{ex} en donnant les arêtes d'un arbre couvrant de poids minimum T_{ex} obtenu par l'algorithme, dans l'ordre.

V.2 Sommets de degré impair

Soit V_1 l'ensemble des sommets de degré impair dans T (en ne considérant que les arêtes de T).

2. Donner V_1 dans le cas où $T = T_{ex}$.
3. Montrer qu'un graphe G' non orienté possède un nombre pair de sommets de degré impair.

La question précédente montre donc que V_1 est de cardinal pair.

V.3 Couplage parfait de poids minimum

On définit $G[V_1]$ (**graphe induit** par V_1) comme le graphe dont l'ensemble de sommets est V_1 et dont deux sommets sont adjacents si et seulement si ils sont adjacents dans G. Les poids des arêtes de $G[V_1]$ sont les mêmes que ceux de G.

Un **couplage parfait de poids minimum** est un couplage parfait dont le poids est minimum parmi tous les couplages parfaits possibles.

4. Montrer l'existance d'un couplage parfait de poids minimum de $G[V_1]$.

5. On suppose que les sommets de G sont des points de $\mathbb{R}^2$ (comme c'est le cas pour G_{ex}). On peut alors considérer chaque arête comme un segment dans $\mathbb{R}^2$. Montrer que si M est un couplage de G dont deux arêtes se croisent (c'est-à-dire que les segments correspondants s'intersectent), alors M n'est pas de poids minimum.

6. Avec $G = G_{ex}$ et $T = T_{ex}$, donner les arêtes d'un couplage parfait de poids minimum M_{ex} de $G[V_1]$, ainsi que son poids.

V.4 Cycle eulérien

Un **multigraphe** est comme un graphe, sauf qu'il peut y avoir plusieurs arêtes entre deux sommets.
On définit le multigraphe $H = (V, E_H)$ dont les sommets sont les mêmes que G et tel que E_H contienne l'union des arêtes de T et de M (avec répétition : E_H est un multiensemble).

7. Dessiner H dans le cas où $G = G_{ex}$, $T = T_{ex}$ et $M = M_{ex}$.

Dans un graphe G' non orienté, un **cycle eulérien** est un cycle passant par chaque arête exactement une fois (mais qui peut passer plusieurs fois par un sommet, contrairement à un cycle hamiltonien).

8. Donner un cycle eulérien dans H dans le cas où $G = G_{ex}$, $T = T_{ex}$ et $M = M_{ex}$.

9. Montrer que si G' possède un cycle eulérien C, alors G' est connexe et tous les sommets de G' sont de degré pair.

On veut écrire un algorithme en OCaml permettant de trouver un cycle eulérien dans un graphe.

10. Écrire une fonction `supprime e l` supprimant la première occurence de `e` dans la liste `l`. Si `e` n'apparaît pas dans `l`, on renverra `l` inchangée. Par exemple, `supprime 3 [6; 3; 2; 3; 5]` doit renvoyer `[6; 2; 3; 5]`.

Dans la suite, on suppose que G' est connexe et que tous les sommets de G' sont de degré pair.

11. On part d'un sommet u quelconque de G', puis, tant que possible, on se déplace suivant une arête adjacente à u qui n'a pas encore été utilisée. Montrer que ce processus termine et que la liste des sommets visités forme un cycle C.

12. Écrire une fonction `cycle g u` qui renvoie le cycle C obtenu par l'algorithme précédent sous forme d'une liste de sommets, où G' est représenté par une liste d'adjacence `g`. On supprimera de `g` les arêtes utilisées par le cycle.

13. Montrer que G' possède un cycle eulérien, en raisonnant par récurrence.

14. En déduire une fonction `euler g u` qui renvoie un cycle eulérien (sous forme d'une liste de sommets) en partant depuis le sommet `u` dans le graphe G' donné sous forme de liste d'adjacence.

V.5 Cycle hamiltonien

Le cycle eulérien C dans le graphe H de la section précédente peut passer plusieurs fois par le même sommet. Pour le transformer en cycle hamiltonien, on supprime les sommets apparaissant plusieurs fois dans C (à part la première occurence).

15. Donner le cycle hamiltonien obtenu avec $G = G_{ex}$ ainsi que son poids. Comparer avec le cycle hamiltonien de poids minimum.

16. Écrire une fonction `supprime_doublons l` qui renvoie la liste obtenue en supprimant les doublons dans la liste de sommets `l`, sauf la première occurence de chaque sommet.
Par exemple, `supprime_doublons [1; 0; 1; 4; 6; 4]` doit renvoyer `[1; 0; 4; 6]`.

V.6 Preuve de la $\frac{3}{2}$-approximation

Soit C^* un cycle hamiltonien de poids minimum de G et T un arbre couvrant de poids minimum de G.

17. Montrer que $w(T) \leq w(C^*)$.

Soit V_1 l'ensemble des sommets de degré impair de T. Soit C_1^* un cycle hamiltonien de poids minimum de $G[V_1]$.

18. Montrer qu'on peut séparer les arêtes de C_1^* en deux couplages parfaits M_1 et M_2 de $G[V_1]$.

19. Soit M un couplage parfait de poids minimum de $G[V_1]$. Montrer que $w(M) \leq \min(w(M_1), w(M_2))$.

20. Montrer que $\min(w(M_1), w(M_2)) \leq \dfrac{w(C_1^*)}{2}$.

21. Montrer que $w(C_1^*) \leq w(C^*)$.

22. Soit C le cycle hamiltonien obtenu par l'algorithme de Christofides. Montrer que $w(C) \leq \dfrac{3}{2} w(C^*)$.

```
BFS()
    A ← {e_0}
    p ← 0
    tant que A ≠ ∅
        B ← ∅
        pour tout x ∈ A
            si x ∈ F alors
                renvoyer VRAI
            B ← s(x) ∪ B
        A ← B
        p ← p + 1
    renvoyer FAUX
```

FIGURE 1 – Parcours en largeur.

Partie I. Jeu à un joueur, parcours en largeur

Un jeu à un joueur est la donnée d'un ensemble non vide E, d'un élément $e_0 \in E$, d'une fonction $s : E \to \mathcal{P}(E)$ et d'un sous-ensemble F de E. L'ensemble E représente les états possibles du jeu. L'élément e_0 est l'état initial. Pour un état e, l'ensemble $s(e)$ représente tous les états atteignables en un coup à partir de e. Enfin, F est l'ensemble des états gagnants du jeu. On dit qu'un état e_p est à la *profondeur* p s'il existe une séquence finie de $p+1$ états

$$e_0 \; e_1 \; \ldots \; e_p$$

avec $e_{i+1} \in s(e_i)$ pour tout $0 \leq i < p$. Si par ailleurs $e_p \in F$, une telle séquence est appelée une *solution* du jeu, de profondeur p. Une solution *optimale* est une solution de profondeur minimale. On notera qu'un même état peut être à plusieurs profondeurs différentes.

Voici un exemple de jeu :

$$\begin{aligned} E &= \mathbb{N}^\star \\ e_0 &= 1 \\ s(n) &= \{\, 2n,\, n+1 \,\} \end{aligned} \tag{1}$$

Question 1. Donner une solution optimale pour ce jeu lorsque $F = \{\, 42 \,\}$.

Parcours en largeur. Pour chercher une solution optimale pour un jeu quelconque, on peut utiliser un parcours en largeur. Un pseudo-code pour un tel parcours est donné figure 1.

Question 2. Montrer que le parcours en largeur renvoie VRAI si et seulement si une solution existe.

Question 3. On se place dans le cas particulier du jeu (1) pour un ensemble F arbitraire pour lequel le parcours en largeur de la figure 1 termine. Montrer alors que la complexité en temps et en espace est exponentielle en la profondeur p de la solution trouvée. On demande de montrer que la complexité est bornée à la fois inférieurement et supérieurement par deux fonctions exponentielles en p.

```
DFS(m, e, p)
    si p > m alors
        renvoyer FAUX
    si e ∈ F alors
        renvoyer VRAI
    pour chaque x dans s(e)
        si DFS(m, x, p + 1) = VRAI alors
            renvoyer VRAI
    renvoyer FAUX
```

FIGURE 2 – Parcours en profondeur (partie II), limité par une profondeur maximale m.

Programmation. Dans la suite, on suppose donnés un type `etat` et les valeurs suivantes pour représenter un jeu en Caml :

```
initial: etat
suivants: etat -> etat list
final: etat -> bool
```

Question 4. Écrire une fonction `bfs: unit -> int` qui effectue un parcours en largeur à partir de l'état initial et renvoie la profondeur de la première solution trouvée. Lorsqu'il n'y a pas de solution, le comportement de cette fonction pourra être arbitraire.

Indication : On pourra avantageusement réaliser les ensembles A et B par des listes, sans chercher à éliminer les doublons, et utiliser une fonction récursive plutôt qu'une boucle `while`.

Question 5. Montrer que la fonction `bfs` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Partie II. Parcours en profondeur

Comme on vient de le montrer, l'algorithme BFS permet de trouver une solution optimale mais il peut consommer un espace important pour cela, comme illustré dans le cas particulier du jeu (1) qui nécessite un espace exponentiel. On peut y remédier en utilisant plutôt un parcours en profondeur. La figure 2 contient le pseudo-code d'une fonction DFS effectuant un parcours en profondeur à partir d'un état e de profondeur p, sans dépasser une profondeur maximale m donnée.

Question 6. Montrer que DFS(m, e_0, 0) renvoie VRAI si et seulement si une solution de profondeur inférieure ou égale à m existe.

Recherche itérée en profondeur. Pour trouver une solution optimale, une idée simple consiste à effectuer un parcours en profondeur avec $m = 0$, puis avec $m = 1$, puis avec $m = 2$, etc., jusqu'à ce que DFS(m, e_0, 0) renvoie VRAI.

Question 7. Écrire une fonction `ids: unit -> int` qui effectue une recherche itérée en profondeur et renvoie la profondeur d'une solution optimale. Lorsqu'il n'y a pas de solution, cette fonction ne termine pas.

Question 8. Montrer que la fonction `ids` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Question 9. Comparer les complexités en temps et en espace du parcours en largeur et de la recherche itérée en profondeur dans les deux cas particuliers suivants :

1. il y a exactement un état à chaque profondeur p ;
2. il y a exactement 2^p états à la profondeur p.

On demande de justifier les complexités qui seront données.

Partie III. Parcours en profondeur avec horizon

On peut améliorer encore la recherche d'une solution optimale en évitant de considérer successivement toutes les profondeurs possibles. L'idée consiste à introduire une fonction $h : E \to \mathbb{N}$ qui, pour chaque état, donne un minorant du nombre de coups restant à jouer avant de trouver une solution. Lorsqu'un état ne permet pas d'atteindre une solution, cette fonction peut renvoyer n'importe quelle valeur.

Commençons par définir la notion de *distance* entre deux états. S'il existe une séquence de $k+1$ états $x_0\ x_1\ \cdots\ x_k$ avec $x_{i+1} \in s(x_i)$ pour tout $0 \leq i < k$, on dit qu'il y a un chemin de longueur k entre x_0 et x_k. Si de plus k est minimal, on dit que la distance entre x_0 et x_k est k.

On dit alors que la fonction h est *admissible* si elle ne surestime jamais la distance entre un état et une solution, c'est-à-dire que pour tout état e, il n'existe pas d'état $f \in F$ situé à une distance de e strictement inférieure à $h(e)$.

On procède alors comme pour la recherche itérée en profondeur, mais pour chaque état e considéré à la profondeur p on s'interrompt dès que $p + h(e)$ dépasse la profondeur maximale m (au lieu de s'arrêter simplement lorsque $p > m$). Initialement, on fixe m à $h(e_0)$. Après chaque parcours en profondeur infructueux, on donne à m la plus petite valeur $p + h(e)$ qui a dépassé m pendant ce parcours, le cas échéant, pour l'ensemble des états e rencontrés dans ce parcours. La figure 3 donne le pseudo-code d'un tel algorithme, appelé IDA*, où la variable globale *min* est utilisée pour retenir la plus petite valeur ayant dépassé m.

Question 10. Écrire une fonction `idastar: unit -> int` qui réalise l'algorithme IDA* et renvoie la profondeur de la première solution rencontrée. Lorsqu'il n'y a pas de solution, le comportement de cette fonction pourra être arbitraire. Il est suggéré de décomposer le code en plusieurs fonctions. On utilisera une référence globale `min` dont on précisera le type et la valeur retenue pour représenter ∞.

Question 11. Proposer une fonction h admissible pour le jeu (1), non constante, en supposant que l'ensemble F est un singleton $\{t\}$ avec $t \in \mathbb{N}^\star$. On demande de justifier que h est admissible.

Question 12. Montrer que, si la fonction h est admissible, la fonction `idastar` renvoie toujours une profondeur optimale lorsqu'une solution existe.

```
DFS*(m, e, p) ≝
    c ← p + h(e)
    si c > m alors
        si c < min alors
            min ← c
        renvoyer FAUX
    si e ∈ F alors
        renvoyer VRAI
    pour chaque x dans s(e)
        si DFS*(m, x, p + 1) = VRAI alors
            renvoyer VRAI
    renvoyer FAUX
```

```
IDA*() ≝
    m ← h(e_0)
    tant que m ≠ ∞
        min ← ∞
        si DFS*(m, e_0, 0) = VRAI alors
            renvoyer VRAI
        m ← min
    renvoyer FAUX
```

FIGURE 3 – Pseudo-code de l'algorithme IDA*.

Partie IV. Application au jeu du taquin

Le jeu du taquin est constitué d'une grille 4×4 dans laquelle sont disposés les entiers de 0 à 14, une case étant laissée libre. Dans tout ce qui suit, les lignes et les colonnes sont numérotées de 0 à 3, les lignes étant numérotées du haut vers le bas et les colonnes de la gauche vers la droite. Voici un état initial possible :

	0	1	2	3
0	2	3	1	6
1	14	5	8	4
2		12	7	9
3	10	13	11	0

On obtient un nouvel état du jeu en déplaçant dans la case libre le contenu de la case située au-dessus, à gauche, en dessous ou à droite, au choix. Si on déplace par exemple le contenu de la case située à droite de la case libre, c'est-à-dire 12, on obtient le nouvel état suivant :

2	3	1	6
14	5	8	4
12		7	9
10	13	11	0

Le but du jeu du taquin est de parvenir à l'état final suivant :

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	