Informatique

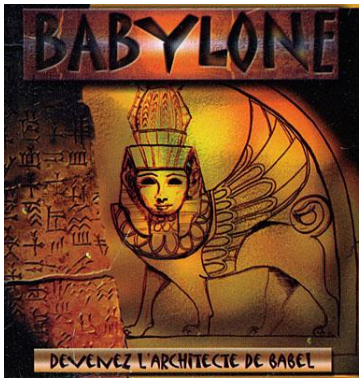
3
Intelligence artificielle

TD3-3

Jeu de Babylone

Idée de sujet tirée de l'ouvrage « Informatique TRONC COMMUN » éditions Vuibert

Exercice 1: Jeu de Babylone



Le jeu

Le jeu de Babylone est un **jeu à deux joueurs jouant chacun à tour de rôle**. Il est constitué de tablettes de couleurs à empiler selon des règles simples jusqu'à ce qu'il ne soit plus possible d'agir, le joueur ne pouvant plus rien faire ayant perdu.
Dans le jeu classique, il y a C=4 couleurs et N=3 tablettes par couleur.
Au départ, chaque tablette est posée seule sur la table. Il est possible d'empiler deux piles à deux conditions :

- Elles ont la même taille
- La tablette supérieure est de la même couleur

Ce jeu a été repris sous le nom SOLUNA dans un contexte de ciel étoilé. Vous pouvez essayer de jouer avec des bouts de papier pour vous entraîner et comprendre le principe du jeu ou aller sur ce site mais vous devrez créer un compte et choisir les options ci-dessous (cliquer « Oui » pour jouer à 2 sur un même ordinateur):

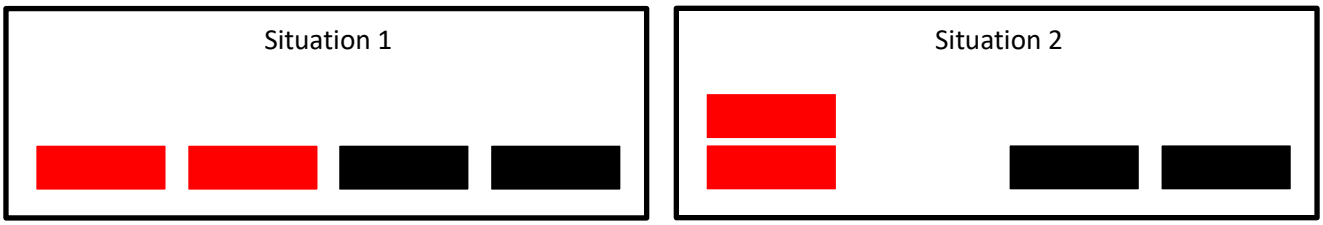
Proposons quelques définitions. A chaque étape du jeu, les joueurs jouant à tour de rôle, le nombre de piles diminue de 1 et conduit donc à une nouvelle situation. Ce jeu ne présente donc pas de cycle (**acyclique**) et toute partie est finie par la présence de positions sans successeurs formant un sous ensemble à atteindre pour chaque joueur, on parle de **jeu d'accessibilité**. Parmi les 3 types d'états finals (J1 gagne, J2 gagne, match nul), on remarque que le match nul n'existe pas dans Babylone. Par ailleurs, on dit que ce jeu est **à information totale** : à tout instant, chacun des joueurs à une information complète de l'état du jeu (rien n'est caché). Les décisions de jeu sont prises en fonction de la situation présente sans tenir compte des situations passées de la partie ou des parties précédentes (**jeu sans mémoire**). Dans une situation, une décision amène toujours à la même situation (**jeu sans hasard ou déterministe**), et ne dépend que de la situation et non du joueur (**jeu impartial**). On dit que c'est **un jeu à somme nulle**, c'est-à-dire que la somme des gains et des pertes est égale à 0. Le gain de l'un constitue obligatoirement une perte pour l'autre. Dans le cas de Babylone, cela veut dire que si l'un gagne, l'autre perd.

Lister les coups possibles

On représente une situation de jeu par une liste x de listes à deux entiers [ci,hi] correspondant à chaque pile pi=x[i] avec :

- ci la couleur (on la définira à l'aide d'un entier positif)
- hi la hauteur de la pile (nombre de tablettes)

Soient les deux situations de jeu suivantes, avec ci=0 pour le rouge, ci=1 pour le noir, et N=2.



Question 1: Donner l'état de la liste x dans les deux situations de jeu proposées

On souhaite déterminer l'ensemble des situations de jeu possibles X stockées dans une liste LX à partir d'une situation initiale x en empilant deux piles pi et pj (pour tous les i et j nécessaires) lorsque c'est possible :

- Empilement de pi sur pj
- Si le résultat est différent du premier empilement, empilement de pj sur pi

On rappelle que la méthode sort des listes L.sort() permet de trier L de manière lexicographique, soit si L contient des listes, selon la première composante, puis pour la même première composante, selon la seconde etc.

Pour rendre facile la détection de situations semblables dans la suite (ex : [[1,2],[0,1],[0,1]]= [[0,1],[0,1],[1,2]]), on triera les listes X avant de les insérer dans les listes résultats.

Question 2: Créer la fonction empile(x,i,j) prenant en argument la liste de situation x et les indices i≠j de deux piles pi et pj, et renvoyant la liste de listes LXij des situations atteignables par empilement lorsque les règles sont respectées de pi sur pj et de pj sur pi si différent

Vérifier :

```
>>> x = [[0,1],[0,1],[1,1],[1,1]]

>>> empile(x,0,1)
[[[0, 2], [1, 1], [1, 1]]]

>>> x = [[0,1],[0,1],[1,1],[1,1]]

>>> empile(x,0,3)
[[[0, 1], [0, 2], [1, 1]], [[0, 1], [1, 1], [1, 2]]]

>>> x = [[0,2],[1,1],[1,1]]

>>> empile(x,0,1)
[]
```

Question 3: Créer la fonction coups(x) prenant en argument une situation x (liste) et renvoyant une liste de listes LX de toutes les situations différentes (pas de doublons) atteignables depuis x

Vérifier :

```
>>> x = [[0,2],[0,3],[1,1],[1,2]]

>>> coups(x)
[[[0, 5], [1, 1], [1, 2]], [[0, 3], [0, 4], [1, 1]],
[[0, 3], [1, 1], [1, 4]], [[0, 2], [0, 3], [1, 3]]]

>>> x = [[0,1],[0,1],[1,1],[1,1]]

>>> coups(x)
[[[0, 2], [1, 1], [1, 1]], [[0, 1], [0, 2], [1, 1]],
[[0, 1], [1, 1], [1, 2]], [[0, 1], [0, 1], [1, 2]]]
```

Création du graphe du jeu

On souhaite réaliser un dictionnaire représentant l'ensemble des situations de jeu possibles à partir d'une situation initiale x0.

Question 4: Créer une fonction init(C,N) prenant en argument le nombre de couleurs C et le nombre de tablettes par couleur N et renvoyant la liste x0 situation initiale du jeu triée par ordre lexicographique

Vérifier :

```
>>> init(3,4)
[[0, 1], [0, 1], [0, 1], [0, 1], [1, 1], [1, 1],
[1, 1], [1, 1], [2, 1], [2, 1], [2, 1], [2, 1]]
```

Les dictionnaires prennent en argument des tuples et il n'est pas possible d'avoir comme argument un tuple de listes, il faut des tuples de tuples de tuples etc. (dans la suite, j'écrirai **T**uple avec une majuscule pour désigner ces tuples). Nous allons donc créer une fonction Tuple transformant une liste de liste de liste etc. en tuple de tuple de tuple etc. Attention à ne pas remplacer la fonction tuple de python (sans majuscule), qui vous sera utile ici.

Question 5: Créer une fonction récursive Tuple(L) réalisant le travail demandé

Vérifier :

```
>>> L = [[[1,2],[3,4,5]],[[6]]]

>>> Tuple(L)
(((1, 2), (3, 4, 5)), ((6,),))
```

Remarque : Il est normal que la transformation d'une liste de listes en tuple fasse apparaître une virgule.

```
>>> tuple([[1]])
([1],)
```

Pour réaliser le dictionnaire représentant le graphe du jeu, on réalise un parcours en largeur. On utilisera une collection deque afin d'améliorer la complexité en temps de ce parcours. Le dictionnaire contiendra :

- Pour clés : Une situation du jeu x transformée en Tuple
- Pour valeurs : Une liste de listes LX de toutes les situations atteignables X à partir de x

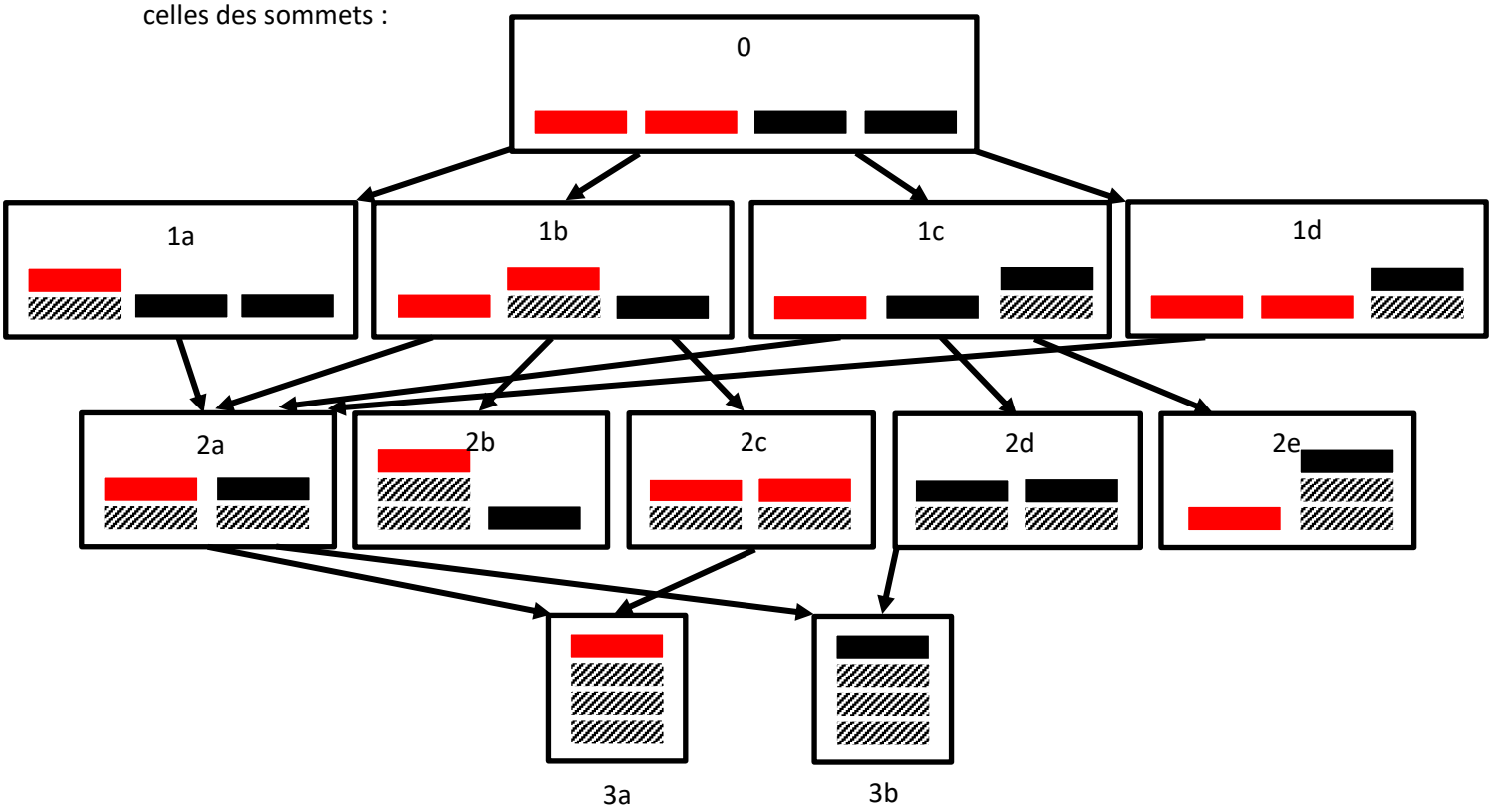
Question 6: Créer la fonction graphe(C,N) réalisant le parcours en largeur des possibilités

Vérifier :

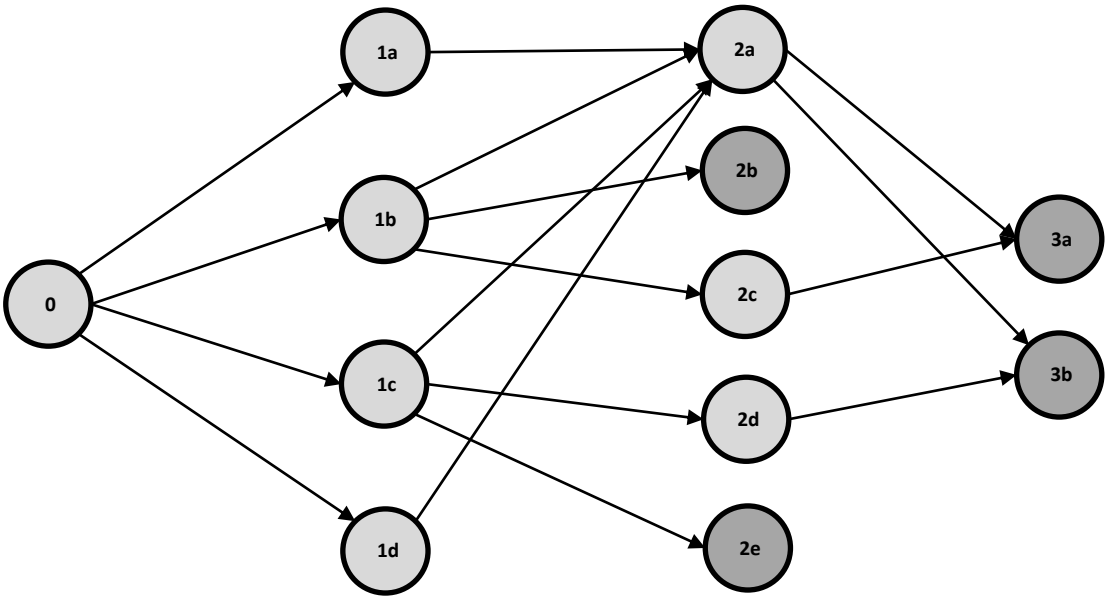
```
>>> graphe(2,2)
{((0, 1), (0, 1), (1, 1), (1, 1)): [[[0, 2], [1, 1], [1, 1]], [[0, 1]
, [0, 2], [1, 1]], [[0, 1], [1, 1], [1, 2]], [[0, 1], [0, 1], [1, 2]]
], ((0, 2), (1, 1), (1, 1)): [[[0, 2], [1, 2]]], ((0, 1), (0, 2), (1,
1)): [[[0, 3], [1, 1]], [[0, 2], [0, 2]], [[0, 2], [1, 2]]], ((0, 1),
(1, 1), (1, 2)): [[[0, 2], [1, 2]], [[1, 2], [1, 2]], [[0, 1], [1, 3]
]], ((0, 1), (0, 1), (1, 2)): [[[0, 2], [1, 2]]], ((0, 2), (1, 2)): [
[[0, 4]], [[1, 4]]], ((0, 3), (1, 1)): [], ((0, 2), (0, 2)): [[[0, 4]
]], ((1, 2), (1, 2)): [[[1, 4]]], ((0, 1), (1, 3)): [], ((0, 4),): []
, ((1, 4),): []}
```

Ce dictionnaire représente le graphe orienté fini G=(S,A) dans lequel des sommets (clés) sont contrôlés par le joueur J1 (S1) et d'autres par le joueur J2 (S2). L'ensemble (G,S1,S2) est appelé **arène**. Chaque **sommet** représente une configuration/**situation/position** du jeu. On appelle **arc/arrête** a=(si,sj)∈A la possibilité pour un joueur Ji de passer du sommet si∈Si au sommet sj∈Sj en un coup, c'est une **décision**. Les arcs relient uniquement des sommets entre S1 et S2. On défini les ensembles F1 et F2 avec F1⋂F2=Ø, les objectifs des deux joueurs (**sommets gagnants/états finals/terminaux** pour J1 et J2), on appelle alors **jeu d'accessibilité** l'ensemble de l'arène et de F1 et F2 ((G,S1,S2),F1,F2).

On peut représenter le graphe pour C=2 et N=2, en ne tenant pas compte des couleurs autres que celles des sommets :



On obtient donc le graphe orienté fini suivant dans lequel les positions cibles des joueurs J1 et J2 (positions sans successeurs) sont représentées en foncé :



Pour la suite, on définit :

```
Pos_0 = [[0,1],[0,1],[1,1],[1,1]]
Pos_1a = [[0,2],[1,1],[1,1]]
Pos_1b = [[0,1],[0,2],[1,1]]
Pos_1c = [[0,1],[1,1],[1,2]]
Pos_1d = [[0,1],[0,1],[1,2]]
Pos_2a = [[0,2],[1,2]]
```

```
Pos_2b = [[0,3],[1,1]]
Pos_2c = [[0,2],[0,2]]
Pos_2d = [[1,2],[1,2]]
Pos_2e = [[0,1],[1,3]]
Pos_3a = [[0,4]]
Pos_3b = [[1,4]]
```

Soient les instructions suivantes :

```
N1 = len(Graphe)
N2 = sum([len(Graphe[x]) for x in Graphe])
```

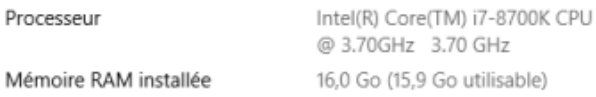
Question 7: Préciser ce que représentent N1 et N2 et vérifier vos propositions sur le graphe

On propose le tableau suivant :

	N =	1	2	3	4
C = 1	Sommets				
	Arêtes				
	Temps (s)				
C = 2	Sommets				
	Arêtes				
	Temps (s)				
C = 3	Sommets				4220
	Arêtes				23 487
	Temps (s)				6560
C = 4	Sommets				
	Arêtes				
	Temps (s)				

Question 8: Utiliser le programme mis en place afin de compléter les cases non grisées

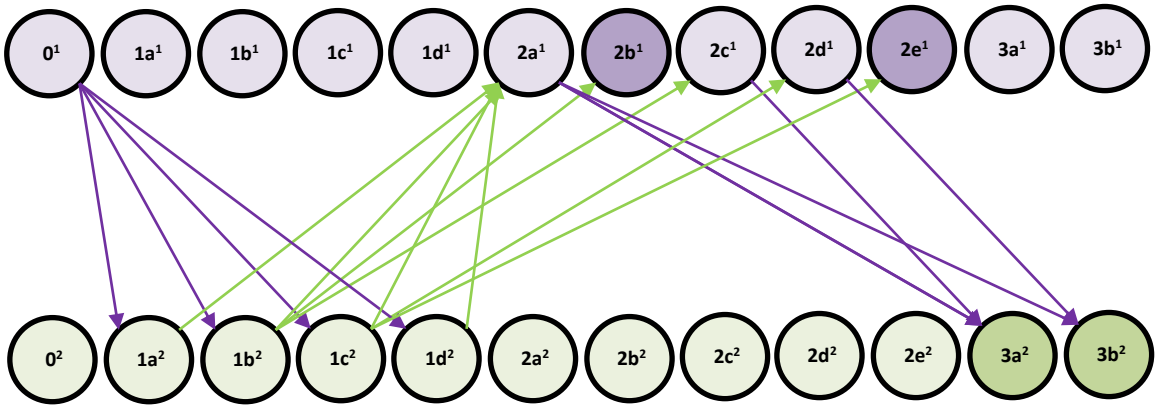
Remarques :



- Le temp indiqués est relatif à l'exécution sur mon ordinateur (caractéristiques sur l'image) ainsi que ceux dans le corrigé PDF
- Le calcul pour C=3 et N=4 est passé sur mon ordinateur avec 16 Go de RAM, mais pas avec un autre PC avec seulement 8 Go de RAM
- Les cases grisées sont des cases pour lesquels mon ordinateur n'a pas été capable de trouver le graphe par manque de mémoire RAM (et après une longue attente) :

Graphe biparti

Dans le cas des jeux d'accessibilité à deux joueurs, on peut représenter le graphe orienté précédent sous la forme d'un graphe **biparti** en doublant les sommets, en indiçant leurs noms par le numéro du joueur (J1 ou J2) et en choisissant le joueur qui commence (J1) :



L'ensemble des sommets S={0,1a,1b,1c,1d,2a,2b,2c,2d,2e,3a,3b} de ce graphe se décompose en deux ensembles S1={0,2a,2b,2c,2d,2e} et S2={1a,1b,1c,1d,3a,3b} tels que S=S1US2 et S1∩S2=Ø avec Si l'ensemble des positions depuis lesquels le joueur Ji jouera. Comme dit précédemment, dans un graphe biparti, les arêtes ne peuvent relier qu'un sommet de S1 à un sommet de S2 et inversement.

Question 9: En remarquant que les sommets du joueur J1 ont un nombre de piles de même parité que x0, créer la fonction sommets_12(G,C,N) prenant en argument le graphe G, N et C, et renvoyant les Tuples des sommets S1 et S2 des joueurs J1 et J2
Question 10: Créer les Tuples S1 et S2 dans le cas C=N=2 et vérifier les résultats avec le graphe biparti ci-dessus

Les parties

Une partie est un parcours du jeu, c'est-à-dire un chemin fini ou infini de sommets du graphe. Une **partie est déclarée gagnée lorsqu'elle se termine dans un état final de J1 (F1) ou de J2 (F2)**, ce qui définit le gagnant.

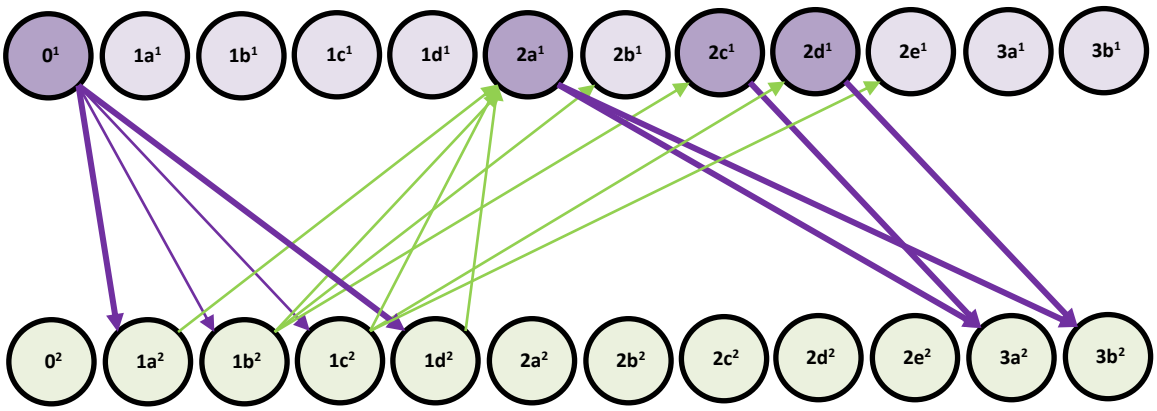
Question 11: Pour C=N=2, et en prenant à chaque fois le premier successeur identifié dans le graphe, afficher une partie et préciser le joueur gagnant

Vérifier :

```
Joueur: 1
Jeu: [[0, 1], [0, 1], [1, 1], [1, 1]]
Joueur: 2
Jeu: [[0, 2], [1, 1], [1, 1]]
Joueur: 1
Jeu: [[0, 2], [1, 2]]
Joueur: 2
Jeu: [[0, 4]]
```

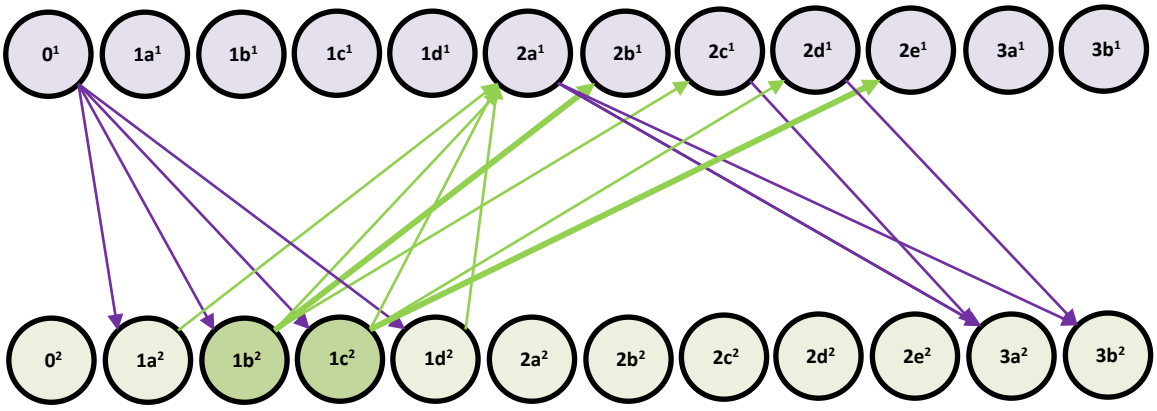
Stratégies et positions gagnantes

Une **stratégie est une méthode à appliquer à chaque coup**. Nous venons de réaliser la stratégie suivante : « à chaque coup, on choisit la première solution que notre algorithme a déterminé dans le graphe ». Nous nous intéressons aux stratégies sans mémoire qui ne dépendent que de l'état actuel du jeu (on ne peut se dire : la dernière fois, l'adversaire a fait ça, alors je fais ça…).
Une **stratégie est dite gagnante pour un joueur si**, en jouant cette stratégie, **toute partie est finie et se termine dans un état gagnant pour ce joueur**. On dit alors que la partie est jouée suivant cette stratégie.
Sur le graphe biparti précédent, nous représentons en gras une stratégie gagnante pour le joueur J1 :



Une **position x est dite gagnante s'il existe une stratégie gagnante depuis ce sommet**. Autrement dit, si le joueur qui y joue a beaucoup de mémoire, quels que soient les coups de son adversaire, il pourra forcer ce dernier à perdre. Cela ne veut pas dire que l'autre joueur est perdant/ne peut pas gagner, cela veut dire que si le joueur disposant d'une position gagnante ne commet aucune erreur, il gagnera.
Les positions {0,2a,2c,2d} sont gagnantes atteignables par le joueur J1.

Sur le graphe biparti ci-dessus, la position 0 est une position gagnante pour le joueur J1. En effet, en suivant la stratégie gagnante depuis 0, il gagnera quel que soit le chemin suivi (positions $1b^2$ et $1c^2$ à proscrire). Si le joueur J1 va sur $1b^2$ ou $1c^2$, le joueur J2 dispose alors d'une position gagnante :



Les positions {1b,1c} sont gagnantes atteignables par le joueur J2.

Finalement, les positions {0,1b,1c,2a,2c,2d} sont gagnantes pour le joueur qui y joue, quel qu'il soit.

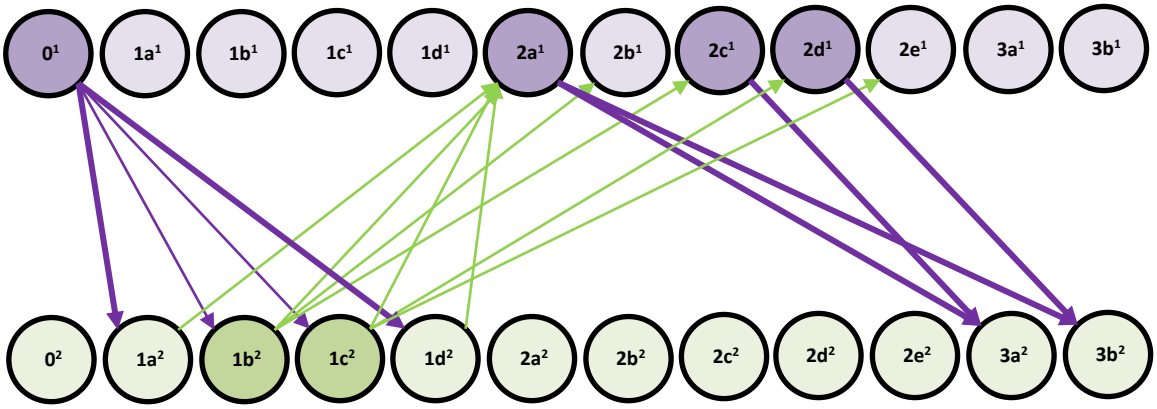
Détermination des positions gagnantes

Dans toute cette partie, je vais essayer de privilégier les termes « elle n'est pas gagnante » plutôt que « elle est non gagnante », ou encore pire « elle est perdante », car ce que l'on peut dire, c'est qu'une position est gagnante ou ne l'est pas.
Pour déterminer si une position est gagnante pour un joueur, on propose une méthode récursive où l'objectif de chacun est identique : ne pas atteindre une position n'ayant pas de successeurs.

Ainsi, une position :

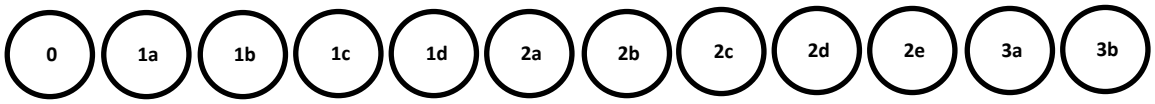
- Est gagnante s'il existe au moins un successeur qui n'est pas gagnant
- N'est pas gagnante si :
    - Elle n'a pas de successeurs
    - Aucun de ses successeurs n'est pas gagnant = Tous ses successeurs sont gagnants

Soit le schéma ci-dessous :



Question 12: En vous aidant du schéma ci-dessus, griser les positions gagnantes pour le joueur qui y joue sur le bilan ci-dessous

Bilan des positions gagnantes :



On remarque que le joueur J1 dispose d'une position gagnante en début de partie dans le jeu à N=C=2.

Question 13: Créer la fonction est_gagnante(G,x) prenant en argument le graphe du jeu et la position x (liste ou Tuple) et renvoyant le booléen True si la position est gagnante pour le joueur qui y joue, et False sinon

Vérifier votre proposition de la question précédente.

On souhaite mettre en place un dictionnaire des positions gagnantes afin de mémoriser si une position du graphe est gagnante ou non pour le joueur qui y joue.

Question 14: Mettre en place une fonction dico_gagnant(G) dont les clés sont les positions du graphe et les valeurs, le booléen True ou False indiquant si la position est gagnante ou non

Vérifier :

```
>>> dico_gagnant(Graphe)
{((0, 1), (0, 1), (1, 1), (1, 1)): True, ((0, 2), (1, 1), (1, 1)):
False, ((0, 1), (0, 2), (1, 1)): True, ((0, 1), (1, 1), (1, 2)): T
rue, ((0, 1), (0, 1), (1, 2)): False, ((0, 2), (1, 2)): True, ((0,
3), (1, 1)): False, ((0, 2), (0, 2)): True, ((1, 2), (1, 2)): True
, ((0, 1), (1, 3)): False, ((0, 4),): False, ((1, 4),): False}
```

La fonction est_gagnante réalisée précédemment :

- Recalcule plusieurs fois le statut est_gagnant de sous-situations lors du calcul du statut d'une seule situation
- Est appelée et refait tout le travail lors du remplissage du dictionnaire

On se propose donc d'optimiser la création du dictionnaire en utilisant la technique de mémoïsation qui retiendra l'état gagnant ou non de chaque situation rencontrée et renverra directement le dictionnaire de tout le graphe.

Question 15: Proposer une fonction dico_gagnant_opt(G) renvoyant le dictionnaire des états gagnants des positions du graphe avec mémoïsation

Vérifier que vous obtenez le même résultat qu'avec dico_gagnant.

Question 16: Comparer les temps d'exécution de dico_gagnant et dico_gagnant_opt pour C=N=3

Remarque : Pour déceler d'éventuelles erreurs de programmation, mettez un `print('mémo')` lorsque la mémoïsation joue son rôle afin de vérifier que c'est bien le cas.

Pour la suite, on écrira : `dico_gagnant = dico_gagant_opt`

Etude des positions gagnantes au départ

On souhaite étudier les cas du tableau proposé ci-dessous afin de compléter chaque case avec le numéro du joueur disposant d'une position gagnante au départ du jeu pour différentes valeurs de N et C.

N =	1	2	3	4
C = 1				
C = 2				
C = 3				1
C = 4			2	1

Question 17: Mettre en place le code nécessaire et remplir le tableau proposé

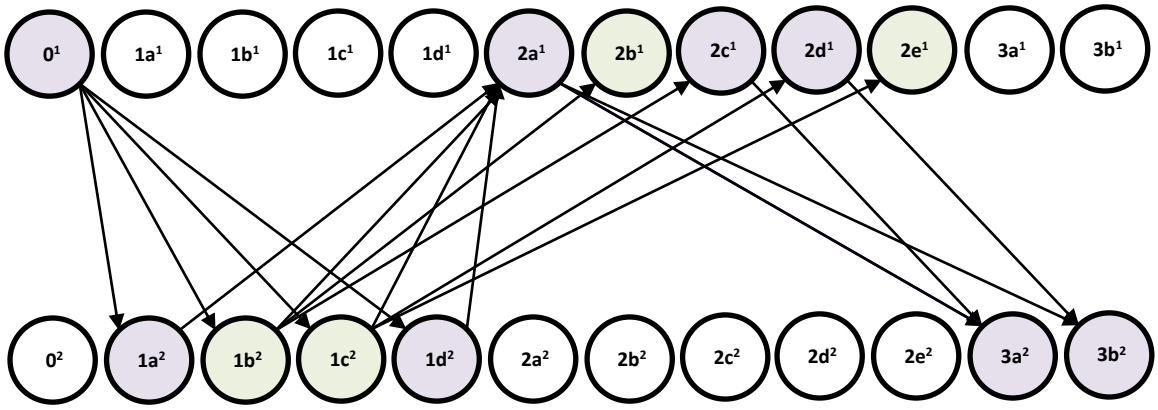
Le jeu classique que l'on peut acheter possède C=4 et N=3. Malheureusement, nous avons vu que nous ne pouvons créer le graphe des cases grises… Nous verrons plus tard qu'un algorithme appelé min-max permet de mettre en place une stratégie de jeu sans avoir de graphe à disposition, ce qui m'a parmi de remplir les cases grises quand même.

Les attracteurs

La définition des positions gagnantes déterminées précédemment ({0,2a,2c,2d}∈S1 gagnantes atteignables par J1 et {1b,1c}∈S2 gagnantes atteignables par J2) peut être étendue aux sommets de S2 gagnants pour J1 et aux sommets de S1 gagnants pour J2. En effet, on remarque qu'une position qui n'est pas gagnante pour le joueur qui y joue signifie que ses successeurs sont tous gagnants pour l'autre joueur. Cette position est donc gagnante pour l'autre joueur. **Ainsi, si aucun match nul n'est possible, toute position est gagnante pour l'un des deux joueurs au cours de la partie**. Les positions qui ne sont pas gagnantes {1a,1d,2b,2e} sont donc des positions gagnantes pour l'adversaire du joueur qui y joue.

On définit ainsi les **attracteurs d'un joueur** (ou **région gagnante**), c'est-à-dire **l'ensemble des positions qui lui garantissent de pouvoir gagner**. Dans notre exemple avec C=N=2 :

- Attracteurs du joueur J1 : A1={0,1a,1d,2a,2c,2d,3a,3b}
- Attracteurs du joueur J2 : A2={1b,1c,2b,2e}



Ces attracteurs sont complémentaires, l'union des deux constitue l'ensemble des sommets : S=A1UA2

On appelle $Attr_1(F)$ l'ensemble des sommets à partir desquels le joueur J1 peut arriver dans F en au plus k coups. Le calcul des attracteurs se fait par itérations sur k (exemple pour le joueur J1) :

- $Attr_1^0(F) = F$
- Tant que $Attr_1^{k+1}(F) \neq Attr_1^k(F)$
- $Attr_1^{k+1}(F) = \begin{cases} Attr_1^k(F) \\ U\{x \in S_1 \mid \exists x' \in Attr_1^k(F), (x,x') \in E\}\ (1) \\ U\{x \in S_2 \mid \forall x'/(x,x') \in E : x' \in Attr_1^k(F)\}\ (2) \end{cases}$, soit :
    - Les sommets actuellement trouvés menant à F
    - (1) Les sommets du graphe du joueur J1 ayant au moins une arête conduisant aux sommets menant à F (le joueur J1 choisira ce coup)
    - (2) Les sommets du graphe du joueur J1 dont les arêtes conduisent toutes à des sommets menant à F (le joueur J1 n'aura pas le choix)

Avec :

- $Attr_1^0(F) \in Attr_1^1(F) \in \cdots \in Attr_1(F)$
- $Attr_1^i(F)$ une suite convergente avec au plus n itérations avec $n = |S|$ le nombre de sommets

Illustration des étapes du calcul des attracteurs $Attr_1(F)$ dans notre exemple :

- Au rang 0, on définit $Attr_1^0(F) = \{3a, 3b\}$, états gagnants pour J1
- Au rang 1, $Attr_1^1(F)$ est défini grâce à $Attr_1^0(F) = \{3a, 3b\}$
    - $U\ \{2a,\ 2c, 2d\}$: Ces 3 positions de J1 mènent à $Attr_1^0(F)$
    - $U\ \emptyset$: Aucun coup de J2 ne mène à $Attr_1^0(F)$
- Au rang 2, $Attr_1^2(F)$ est défini grâce à $Attr_1^1(F) = \{2a, 2c, 2d, 3a, 3b\}$
    - $U\ \emptyset$: Aucun coup de J1 ne mène à $Attr_1^1(F)$
    - $U\{1a, 1d\}$: Ces 2 positions de J2 mènent obligatoirement à $Attr_1^1(F)$
- Au rang 3, $Attr_1^3(F)$ est défini grâce à $Attr_1^2(F) = \{1a, 1d, 2a, 2c, 2d, 3a, 3b\}$
    - $U\{0\}$: Cette position de J1 mène à $Attr_1^2(F)$
    - $U\ \emptyset$: Aucun coup de J2 ne mène à $Attr_1^2(F)$

Finalement, $Attr_1(F) = \{0,1a, 1d, 2a, 2c, 2d, 3a, 3b\}$.

Dans le cas du jeu de Babylone :

- A chaque itération, une des deux lignes (1) ou (2) ne trouvera aucun sommet dans S1 ou S2.
- On pourrait ne pas traiter tous les sommets à chaque itération en se limitant successivement à chaque « étage » du jeu, mais la méthode ci-dessus s'applique à d'autres types de jeux et sera donc programmée telle que proposée.

Calcul des attracteurs

On souhaite créer deux dictionnaires dAi contenant pour clés, les sommets du graphe, et pour valeur un booléen True ou False indiquant si le sommet (la clé) est dans les sommets gagnants du joueur Ji.
Lors de l'initialisation des attracteurs des deux joueurs dans le cas de notre jeu, on a :

- Les attracteurs du joueur J1 sont les sommets de S2 sans successeurs
- Les attracteurs du joueur J2 sont les sommets de S1 sans successeurs

A chaque fois donc, le dernier coup du joueur Ji doit mener dans les sommets sans successeurs de l'autre joueur.

Question 18: Créer la fonction init_attracteurs(G,S1) prenant en argument le graphe G du jeu et le Tuple S1 des sommets du joueur J1 et renvoyant les deux dictionnaires attendus

Dans le cas C=N=2, vérifier :

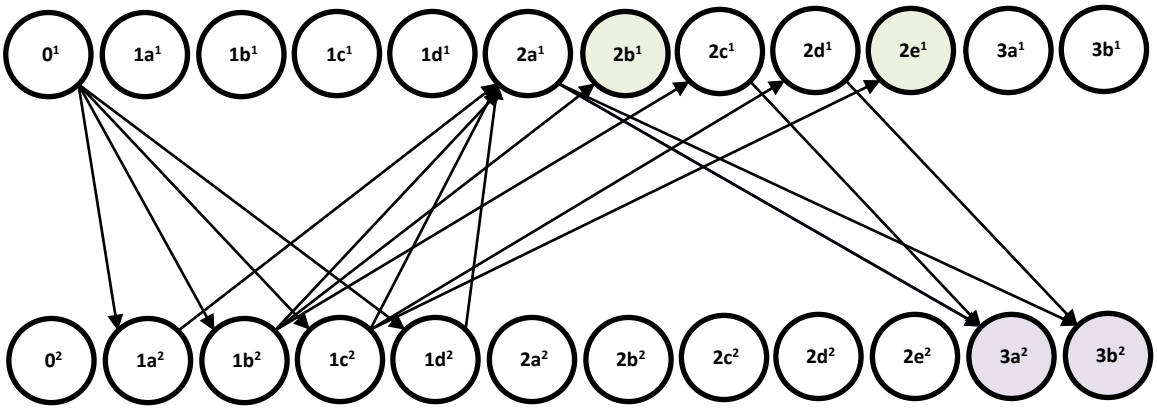
```
>>> dA1,dA2 = init_attracteurs(Graphe,S1)

>>> dA1
{((0, 1), (0, 1), (1, 1), (1, 1)): False, ((0, 2), (1, 1), (1, 1))
: False, ((0, 1), (0, 2), (1, 1)): False, ((0, 1), (1, 1), (1, 2))
: False, ((0, 1), (0, 1), (1, 2)): False, ((0, 2), (1, 2)): False,
((0, 3), (1, 1)): False, ((0, 2), (0, 2)): False, ((1, 2), (1, 2))
: False, ((0, 1), (1, 3)): False, ((0, 4),): True, ((1, 4),): True
}

>>> dA2
{((0, 1), (0, 1), (1, 1), (1, 1)): False, ((0, 2), (1, 1), (1, 1))
: False, ((0, 1), (0, 2), (1, 1)): False, ((0, 1), (1, 1), (1, 2))
: False, ((0, 1), (0, 1), (1, 2)): False, ((0, 2), (1, 2)): False,
((0, 3), (1, 1)): True, ((0, 2), (0, 2)): False, ((1, 2), (1, 2)):
False, ((0, 1), (1, 3)): True, ((0, 4),): False, ((1, 4),): False}
```

On appelle Cond_1 et Cond_2 les conditions appliquées à une position x pour le joueur Ji telles que :

$$\begin{cases} Cond\_1 : \exists x' \in Attr_i^k(F), (x,x') \in E \\ Cond\_2 : \forall x'/(x,x') \in E : x' \in Attr_i^k(F) \end{cases}$$

Question 19: Créer la fonction cond_1(G,di,x) prenant en argument le graphe du jeu G, le dictionnaire di des attracteurs du joueur Ji et un sommet x du jeu (liste), et renvoyant le booléen True si le sommet respecte la condition (1), False sinon
Question 20: Créer la fonction cond_2(G,di,x) prenant en argument le graphe du jeu G, le dictionnaire di des attracteurs du joueur Ji et un sommet x du jeu (liste), et renvoyant le booléen True si le sommet possède des successeurs et respecte la condition (2), False sinon

Les dictionnaires n'étant qu'initialisés ainsi :



On ne peut vérifier ces conditions que pour certains positions particulières pour le moment :

- Pour dA1 : vérifier les résultats pour 2a à 2e
- Pour dA2 : vérifier les résultats pour 1a à 1d

Puis :

- Pour dA2 : vérifier les résultats pour 2a à 2e
- Pour dA1 : vérifier les résultats pour 1a à 1d

Question 21: Créer la fonction attracteurs_it(G,di,Si) prenant en argument le graphe G, le dictionnaire di des attracteurs de Ji, et le Tuple Si des positions du joueur Ji, réalisant une itération de la procédure de détermination des attracteurs du joueur Ji en changeant les valeurs dans di (en place), et renvoyant True si au moins un changement (False vers True) a eu lieu, False sinon

Remarques :

- On veillera à ne traiter que les sommets n'étant pas à True (donc à False) dans di pour ne pas risquer de repasser à False des sommets initialisés à True n'ayant donc pas de successeurs. Et, cela gagnera du temps.
- Cette fonction pourrait être optimisée dans le cas du jeu de Babylone en ne considérant pas tous les sommets False à chaque itération, je ne souhaite pas aller plus loin ici

Question 22: Créer la fonction attracteurs_Ji(G,di,Si) avec les mêmes arguments que attracteurs_it réalisant la procédure complète de création des attracteurs du joueur Ji en complétant di

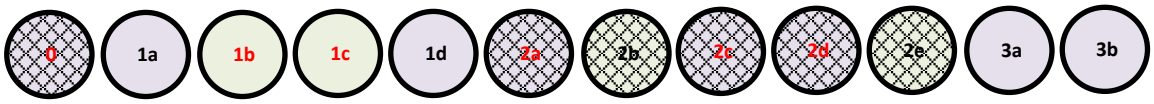
Question 23: Créer enfin la fonction attracteurs(G,C,N) créant et renvoyant les dictionnaires dA1 et dA2 des attracteurs des joueurs J1 et J2

Vérifier vos dictionnaires avec les attracteurs proposés précédemment dans le cas C=N=2.

Les positions gagnantes pour le joueur qui y joue obtenues précédemment peuvent être déterminées à l'aide des attracteurs des deux joueurs. En effet, **une position est gagnante pour le joueur Ji qui s'y trouve si cette position est dans Si et fait partie de ses attracteurs Ai**.

Illustration avec :

- En quadrillé : les positions du joueur 1
- En texte rouge : les positions gagnantes du jeu



Les positions gagnantes du joueur 1 sont les positions quadrillées violettes. Les positions gagnantes du joueur 2 sont les positions non quadrillées vertes.

Question 24: Créer la fonction dico_gagnant_att(G,C,N) prenant en arguments le graphe G, C et N, et renvoyant le dictionnaire des positions gagnantes

Vérifier que vous obtenez le même dictionnaire qu'avec les fonctions dico_gagnant ou dico_gagnant_opt.

Stratégie optimale

On souhaite maintenant créer une fonction qui, pour toute position x du jeu, renvoie la stratégie optimale à jouer, c'est-à-dire le meilleur coup.

On peut définir le meilleur coup à partir de la position x ainsi :

- Si x n'a pas de successeurs, on renvoie une liste vide
- Si x a des successeurs, on choisit de la sorte :
    - Création de la liste des choix L_Choix contenant
        - Les successeurs qui ne sont pas gagnants s'il y en a
        - Tous les successeurs sinon
    - Choix aléatoire d'un élément de L_Choix

Ainsi, **le joueur qui joue choisit à chaque fois une position suivante n'étant pas gagnante pour l'adversaire (donc gagnante pour lui), si elle existe**.
Remarque : cela revient à choisir, pour chaque joueur, un sommet parmi ses attracteurs, mais nous ne le programmerons pas ainsi puisqu'il faut alors différencier le coup de chaque joueur.

Question 25: Créer la fonction strategie_opt(G,dg,x) prenant en argument le graphe G, le dictionnaire gagnant dg et une position x (liste), et renvoyant un choix de successeur respectant le choix du meilleur coup
En utilisant plusieurs fois la fonction, vérifier :

```
>>> C=N=2

>>> G = graphe(C,N)

>>> dg = dico_gagnant(G)

>>> x0 = init(C,N)

>>> strategie_opt(G,dg,x0)
[[0, 1], [0, 1], [1, 2]]

>>> strategie_opt(G,dg,x0)
[[0, 2], [1, 1], [1, 1]]
```

Question 26: Créer la fonction strategies_opt(G,dg) prenant en argument le graphe G et le dictionnaire gagnant dg, et renvoyant un dictionnaire dico_s dont chaque clé est une position x du jeu (Tuple), et chaque valeur la solution issue du meilleur coup à jouer depuis x pour le joueur qui y est

Voici un exemple de résultat obtenu, mais il n'est évidemment pas unique :

```
>>> C=N=2

>>> G = graphe(C,N)

>>> dico_g = dico_gagnant(G)

>>> strategies_opt(G,dico_g)
{((0, 1), (0, 1), (1, 1), (1, 1)): [[0, 2], [1, 1], [1, 1]], ((0,
2), (1, 1), (1, 1)): [[0, 2], [1, 2]], ((0, 1), (0, 2), (1, 1)): [
[0, 3], [1, 1]], ((0, 1), (1, 1), (1, 2)): [[0, 1], [1, 3]], ((0,
1), (0, 1), (1, 2)): [[0, 2], [1, 2]], ((0, 2), (1, 2)): [[1, 4]],
((0, 3), (1, 1)): [], ((0, 2), (0, 2)): [[0, 4]], ((1, 2), (1, 2))
: [[1, 4]], ((0, 1), (1, 3)): [], ((0, 4),): [], ((1, 4),): []}
```

Simulation de jeu en IA

Nous allons maintenant simuler un jeu où chaque joueur est une intelligence artificielle.

Question 27: Créer la fonction jeu(C,N) qui affiche le joueur disposant d'une stratégie gagnante au départ, les étapes du jeu en précisant quel joueur joue, et quel joueur gagne

Question 28: Utiliser la fonction jeu pour différentes situations et conclure

Exemples d'exécutions :

```
>>> jeu(2,2)
Le joueur 1 dispose d'une position gagnante
Départ: [[0, 1], [0, 1], [1, 1], [1, 1]]
Joueur: 1
[[0, 2], [1, 1], [1, 1]]
Joueur: 2
[[0, 2], [1, 2]]
Joueur: 1
[[1, 4]]
Joueur: 2
[]
a perdu
>>> jeu(3,3)
Le joueur 1 dispose d'une position gagnante
Départ: [[0, 1], [0, 1], [0, 1], [1, 1], [1, 1], [1, 1], [2, 1], [2, 1], [2, 1]]
Joueur: 1
[[0, 1], [0, 1], [1, 1], [1, 1], [1, 2], [2, 1], [2, 1], [2, 1]]
Joueur: 2
[[0, 1], [0, 1], [1, 1], [1, 1], [1, 2], [2, 1], [2, 2]]
Joueur: 1
[[0, 1], [1, 1], [1, 2], [1, 2], [2, 1], [2, 2]]
Joueur: 2
[[0, 1], [1, 1], [1, 2], [1, 4], [2, 1]]
Joueur: 1
[[0, 1], [1, 2], [1, 2], [1, 4]]
Joueur: 2
[[0, 1], [1, 4], [1, 4]]
Joueur: 1
[[0, 1], [1, 8]]
Joueur: 2
[]
a perdu
>>> jeu(2,3)
Le joueur 2 dispose d'une position gagnante
Départ: [[0, 1], [0, 1], [0, 1], [1, 1], [1, 1], [1, 1]]
Joueur: 1
[[0, 1], [0, 1], [0, 2], [1, 1], [1, 1]]
Joueur: 2
[[0, 2], [0, 2], [1, 1], [1, 1]]
Joueur: 1
[[0, 4], [1, 1], [1, 1]]
Joueur: 2
[[0, 4], [1, 2]]
Joueur: 1
[]
a perdu

>>> jeu(3,2)
Le joueur 2 dispose d'une position gagnante
Départ: [[0, 1], [0, 1], [1, 1], [1, 1], [2, 1], [2, 1]]
Joueur: 1
[[0, 2], [1, 1], [1, 1], [2, 1], [2, 1]]
Joueur: 2
[[0, 2], [1, 1], [1, 1], [2, 2]]
Joueur: 1
[[1, 1], [1, 1], [2, 4]]
Joueur: 2
[[1, 2], [2, 4]]
Joueur: 1
[]
a perdu
```

Heuristique et min-max

Lorsque l'arbre est trop grand pour être exploré, le graphe n'existe pas. Il existe des stratégies visant à choisir le meilleur coup à jouer à la profondeur p, c'est-à-dire en étudiant les gains/pertes à p étapes.

Dans toute cette partie, on définit :

- Joueur : le joueur qui joue son coup, avec l'objectif de gagner
- Adversaire : l'autre joueur, avec l'objectif de le faire perdre

Dans le cas de Babylone par exemple :

- Pour $p = 1$: On explore toutes les possibilités parmi les successeurs de x et on choisit, s'il existe, celui qui mène l'adversaire à perdre/celui qui fait gagner le joueur.
- Pour $p = 2$: On explore toutes les possibilités parmi les successeurs des successeurs de x et on choisit, parmi les successeurs celui qui ne fait pas perdre le joueur.
- Pour $p = 3$: On explore toutes les possibilités parmi les successeurs des successeurs des successeurs de x et on choisit, s'il existe, celui qui mène l'adversaire à perdre/celui qui fait gagner le joueur.

D'une manière générale, on introduit une fonction d'utilité appelée « **heuristique** » qui, à chaque coup, associe un poids (un réel). Plus le poids est grand, plus le joueur a des chances de gagner, et au contraire, plus il est faible, plus il a de chances de perdre. Une heuristique doit être trouvée avec intuition, elle est différente pour chaque jeu.

Le jeu de Babylone ne se prête pas à la détermination d'une heuristique sophistiquée mais peut au moins nous permettre de choisir une stratégie simple étudiant p sous étapes, et choisissant la prochaine solution menant possiblement au gain après p coups, ou au moins, menant à ne pas perdre après p coups.

Soit l'heuristique suivante :

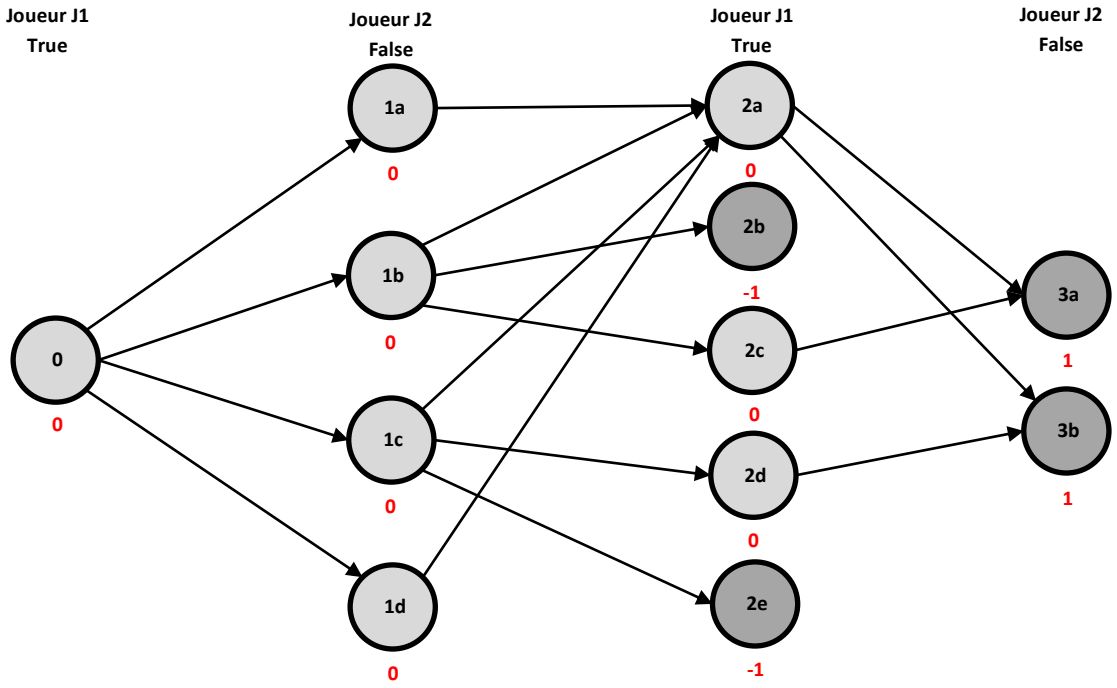
- Si x ne possède aucun successeur (position finale) :
  - Si x appartient au joueur, retourner -1 (le joueur perd)
  - Sinon (x appartient à l'adversaire), renvoyer 1 (le joueur gagne)
- Sinon, retourner 0

Question 29: Mettre en place la fonction h(x,bool) prenant en argument une position x du jeu (liste) et le booléen bool valant True si le joueur joue, False si c'est son adversaire, et renvoyant le résultat de l'heuristique proposée

Vérifier que vous obtenez les résultats suivant et les comprendre à l'aide des explications qui suivent.

Instruction	Résultat
`h(Pos_0,True)`	0
`h(Pos_1a,False)`	0
`h(Pos_2b,True)`	-1
`h(Pos_3a,False)`	1

On reprend le graphe du jeu quand C=N=2 et on le complète des valeurs de l'heuristique proposée lorsque le joueur J1 réalise le premier coup et à son tour :



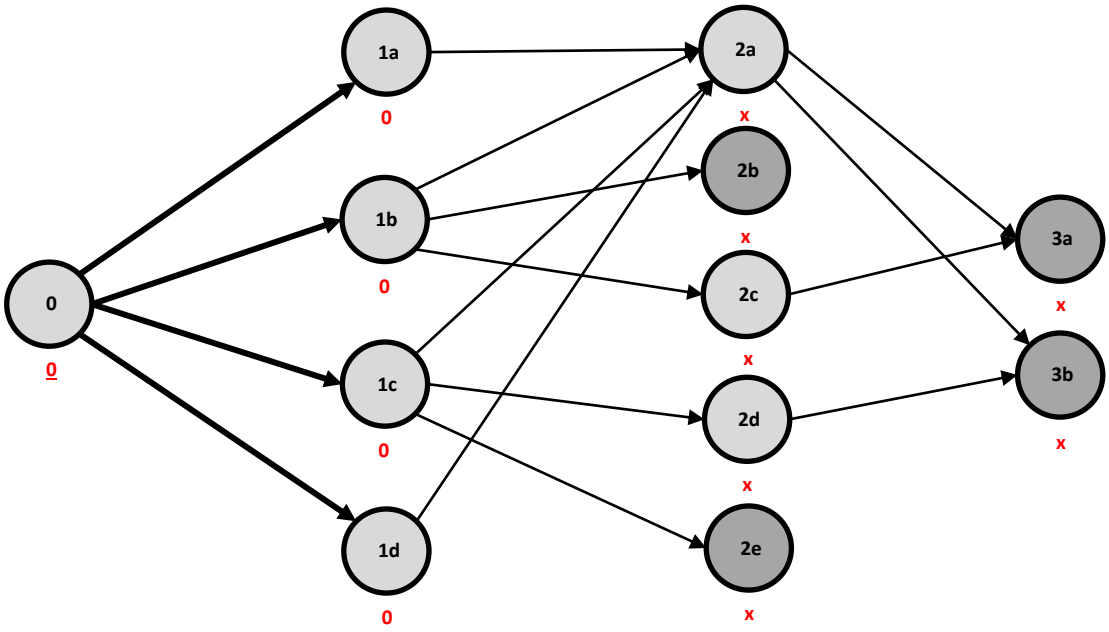
L'objectif du joueur J1 est d'aboutir à une position où l'heuristique est maximale (1), et de ne pas aboutir à une heuristique minimale (-1).

Pour guider ce choix, on introduit l'algorithme **min-max** depuis la position x à la profondeur p pour le joueur qui joue. Le principe est le suivant :

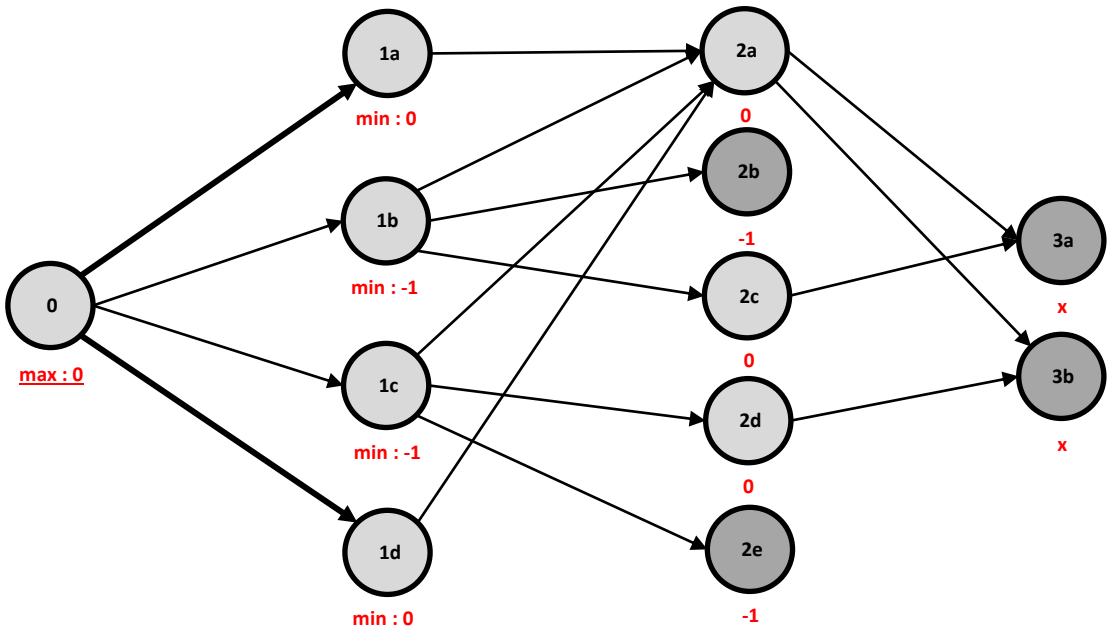
- Si la position x n'a pas de successeurs ou si la profondeur p=0 est atteinte :
  - Renvoyer l'heuristique de la position x
- Si la position x présente des successeurs :
  - Si c'est un coup du joueur :
    - Calculer le résultat de l'algorithme min-max pour tous les successeurs de x à la profondeur p-1 pour l'adversaire
    - Renvoyer le maximum des résultats obtenus
  - Si c'est un coup de l'adversaire :
    - Calculer le résultat de l'algorithme min-max pour tous les successeurs de x à la profondeur p-1 pour le joueur
    - Renvoyer le minimum des résultats obtenus

Reprenons le graphe ci-dessus et supprimons les heuristiques pour inscrire les résultats de l'algorithme min-max :

- Pour le joueur J1 à la position 0, et p=1 :

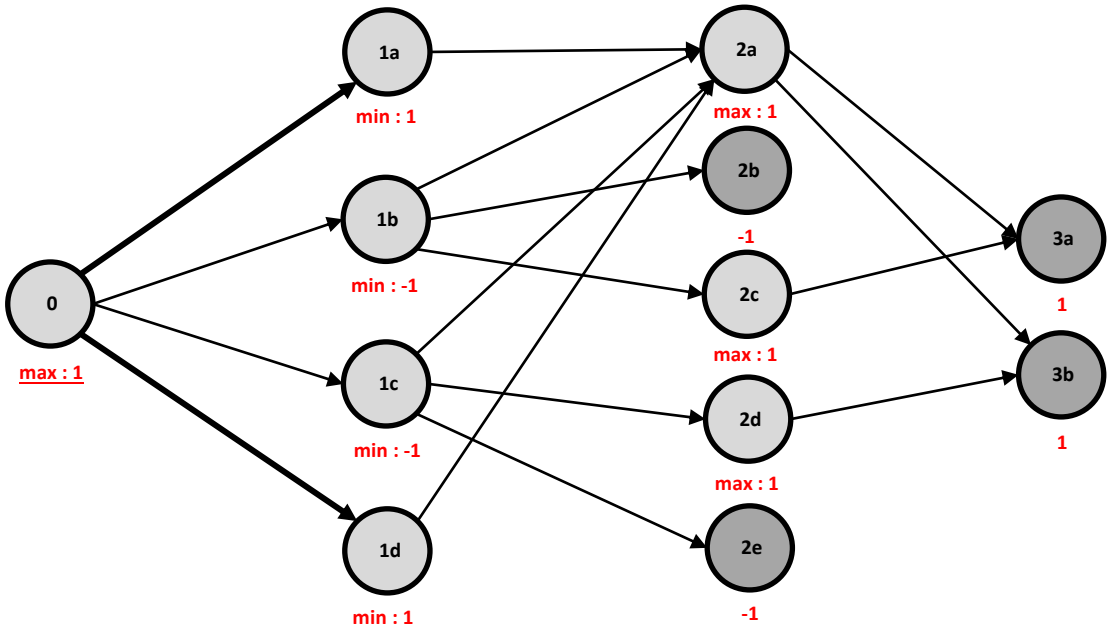


- Pour le joueur J1 à la position 0, et p=2 :



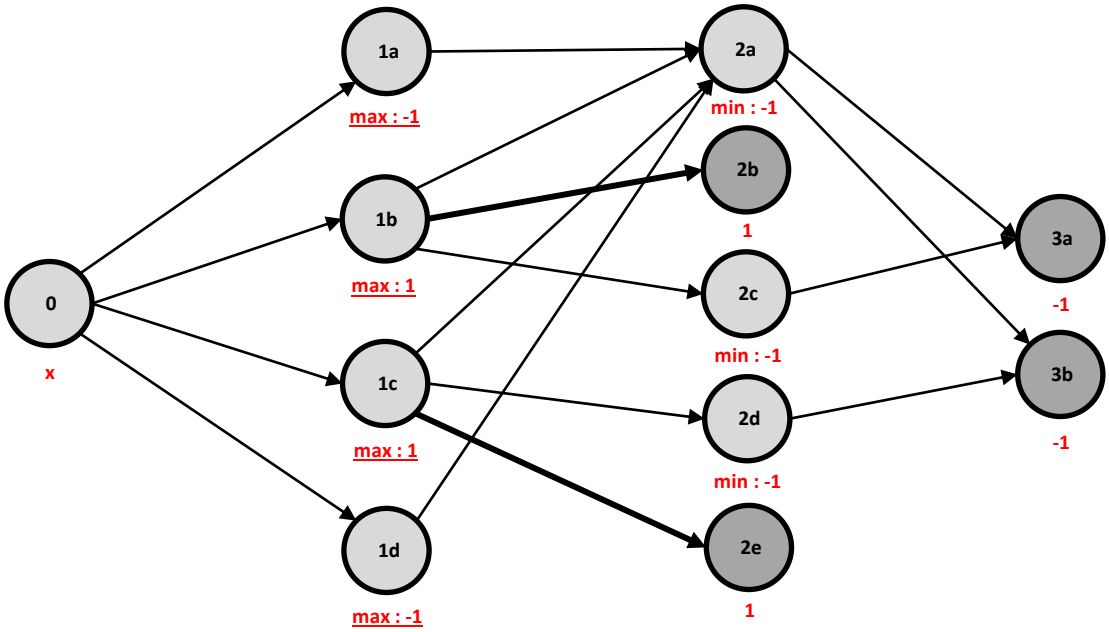
En choisissant d'aller là où le résultat est le plus grand depuis la position 0, on remarque que l'on évitera pour le joueur J1 les positions 1b et 1c, ce qui garantit la victoire du joueur J1 à p=2.

- Pour le joueur J1 à la position 0, et p=3 :



Avec p=3, nous avons finalement étudié tout le graphe et nous voyons que les chemins privilégiés seront les mêmes qu'avec p=2. Si parmi les deux choix à 1, l'un d'eux ne menait pas à la réussite, il vaudrait 0 et le seul chemin menant vers la réussite serait choisi ☺

Etudions les 4 cas de choix sur un seul graphe (selon ce qu'aura choisi le joueur J1) pour le joueur J2 aux positions 1a 1b 1c 1d, et p=2 (remarquer le changement des valeurs 1 et -1 de l'heuristique sur les sommets sans successeurs) :



- On voit qu'en 1b et en 1c, le joueur J2 choisira les positions 2b et 2e le menant à la victoire

Nous allons proposer la fonction min-max telle que présentée précédemment.

Question 30: Créer la fonction min_max(x,p,bool) prenant en argument une position x (liste), une profondeur p (entier) et le booléen représentant si c'est le coup du joueur (True) ou de l'adversaire (False) et renvoyant la valeur min-max attendue

Vous identifierez ce que représentent les instructions suivantes et vérifierez vos résultats :

Instruction	Résultat
`min_max(Pos_0,1,True)`	0
`min_max(Pos_0,2,True)`	0
`min_max(Pos_0,3,True)`	1
`min_max(Pos_1a,2,True)`	-1
`min_max(Pos_1b,2,True)`	1
`min_max(Pos_1c,2,True)`	1
`min_max(Pos_1d,2,True)`	-1

Question 31: Etudier la valeur renvoyée par la fonction min-max sur la position initiale x0 à la profondeur maximale pour les combinaisons de N et C du tableau ci-dessous et conclure

N =	1	2	3
C = 1			
C = 2			
C = 3			

Remarque :

- On admettra qu'avec l'heuristique proposée dans cet exercice, et en calculant le min-max de x0 à la profondeur maximale, on retrouve les positions gagnantes au départ. Cette fois-ci, aucun graphe n'étant créé, il n'y a plus de problèmes de mémoire RAM, mais les calculs restent LONGS ! Voilà comment j'ai pu remplir les cases grisées du tableau vu plus haut
- La fonction min-max programmée ci-dessus recalcul beaucoup de résultats sur des situations identiques et peut être grandement optimisée par mémoïsation

Question 32: Si vous avez du temps, proposer une fonction min_max_opt(x,p,bool) réalisant le même travail que min_max avec mémoïsation, observer le gain de temps pour C=N=3 et remplir le tableau des positions gagnantes au départ pour Cmax=4 et Nmax=4

Pour ma part, j'ai gagné un facteur 50 sur le temps d'exécution pour C=N=3 ! Et obtenu les résultats pour C=4 et/ou N=4 quasiment immédiatement avec cette fonction optimisée. Si vous avez programmé la fonction min_max_opt, écrivez dans la suite : `min_max = min_max_opt`

Dans la suite, nous allons avoir à choisir un maximum dans une liste ayant des exæquos. Nous souhaitons choisir aléatoirement l'un des maximums en renvoyant son indice dans la liste étudiée.

Question 33: Créer la fonction choix_ind_max(L) prenant en argument une liste L et renvoyant aléatoirement l'un des indices python des maximums de L

Vérifier par exemple :

```
>>> L = [2,1,2,1]
>>> choix_ind_max(L)
2
>>> choix_ind_max(L)
2
>>> choix_ind_max(L)
0
>>> choix_ind_max(L)
2
>>> choix_ind_max(L)
2
```

Pour simuler un jeu, il nous faut une fonction qui détermine le meilleur coup à jouer en étudiant une partie du graphe à partir de la position x jusqu'à une profondeur p. Voici quelques indications sur cette fonction :

- Si la liste des coups depuis x est vide, renvoyer une liste vide
- Sinon :
    - Si p=0 :
        - Choisir aléatoirement l'un des successeurs possibles de x
    - Sinon :
        - Choisir le successeur de x à l'aide de l'algorithme du min-max

On remarquera que l'on réalise manuellement la première itération du min-max afin d'identifier le coup à jouer (savoir lequel des successeurs présente le maximum), ce qui devra être traduit dans l'appel de la fonction min-max 😉

Question 34: Créer la fonction strategie_h(x,p) renvoyant le meilleur choix de coup depuis x avec une étude à la profondeur p

Vous identifierez ce que représentent les instructions suivantes et vérifierez vos résultats à l'aide des graphes présentés dans cette partie :

```
strategie_h(Pos_0,1)
strategie_h(Pos_0,2)
strategie_h(Pos_1b,2)
strategie_h(Pos_1c,2)
```

Nous allons maintenant simuler un jeu comme nous l'avons fait avec la stratégie optimale. Vous reprendre cette fonction et l'adapterez.

Question 35: Créer la fonction jeu_h(C,N,p) simulant un jeu pour les valeurs de C et N avec une étude à chaque coup à la profondeur p

Remarques : Encore plus de mémoïsation ?

- Il serait encore possible d'améliorer l'exécution de min-max lors de l'exécution de strategie_h puisque la fonction recalcule des résultats déjà trouvés pour un même joueur, depuis la même position et pour la même profondeur
- Attention toutefois, pour deux exécutions de strategie_h (dans jeu_h), ce n'est pas vrai car min-max est appelée à partir de positions qui changent et pour des joueurs qui changent…

Question 36: Utiliser la fonction jeu_h pour différentes situations et observer les résultats

Vous vérifierez par exemple que le joueur gagnant de l'instruction `jeu_h(2,2,2)` est toujours le bon joueur.

Remarque : sachez qu'il est possible d'améliorer l'algorithme min-max en utilisant la méthode **d'élagage alpha-bêta**, mais ce n'est pas au programme. Voilà de quoi aller plus loin 😊