Informatique Tronc Commun

PTSI-PT*

Séquence 5

Théorie des jeux

Thème : Étude des jeux

Objectifs

- Révisions sommaires sur les graphes ;
- Jeux d'accessibilité à 2 joueurs ;
- Stratégie et stratégie gagnante ;
- Position gagnante ;
- Notion d'attracteurs ;
- Construction de stratégie ;
- Notion d'heuristique ;
- Algorithme mini-max avec une heuristique.

Table des matières

Cours - Théorie des jeux

Dans ce chapitre nous allons nous intéresser à l'étude théorique et algorithmique de jeux à deux joueurs antagonistes jouant alternativement. Les jeux qui nous intéressent sont à information totale : à tout instant d'une partie chacun des joueurs a une vision complète de l'état du jeu. Ceci exclus la plus-part des jeux de cartes (on ne connait pas le jeu de l'adversaire) mais inclus des jeux tels les échecs, les dames, le go, etc. Nous allons nous intéresser à des jeux « simples » pour lesquels il est possible de déterminer (au moins pour de petites configurations) une stratégie gagnante, puis nous verrons, pour les jeux les plus complexes, comment bâtir une stratégie à l'aide d'une heuristique.

1 Introduction et intérêt de la théorie des jeux

1.1 Histoire

La théorie des jeux commence véritablement dans les années 1920 avec von Neumann en chef de file de ces études. Il s'intéresse aux jeux à deux joueurs qui disposent chacun d'un nombre fini d'actions et se poursuit avec le théorème de Nash en 1950. La théorie des jeux a un impact conséquent dans la prise de décision ou la compréhension des phénomènes économiques, politiques, militaires, biologiques, etc. Par exemple, par le prisme de la théorie des jeux, on constate que des comportements animaliers sont complètement différents des comportements humains. Certaines espèces (comme les fourmis ou les abeilles) ont des stratégies collaboratives là où les humains ont un comportement individualiste parce que les objectifs sont différents.

Exemple 1

Dilemme du prisonnier *:*

	Le suspect n°2 se tait	Le suspect n°2 dénonce
Le suspect n°1 se tait	Les deux font 6 mois de prison	1 fait 10 ans de prison ; 2 est libre
Le suspect n°1 dénonce	1 est libre ; 2 fait 10 ans de prison	Les deux font 5 ans de prison

Deux suspects sont arrêtés par la police. Mais les agents n'ont pas assez de preuves pour les inculper, donc ils les interrogent ***séparément*** *en leur faisant la même offre. « Si tu dénonces ton complice et qu'il ne te dénonce pas, tu seras remis en liberté et l'autre écopera de 10 ans de prison. Si tu le dénonces et lui aussi, vous écoperez tous les deux de 5 ans de prison. Si personne ne dénonce l'autre, vous aurez tous deux 6 mois de prison. »*
Chacun des prisonniers réfléchit de son côté en considérant les deux cas possibles de réaction de son complice.

- *« Dans le cas où il me dénoncerait :*
  - *Si je me tais, je ferai 10 ans de prison ;*
  - *Mais si je te dénonce, je ne ferai que 5 ans. »*
- *« Dans le cas où il ne me dénoncerait pas :*
  - *Si je me tais, je ferai 6 mois de prison ;*
  - *Mais si je le dénonce, je serai libre. »*

Si chacun des complices fait ce raisonnement, les deux vont probablement choisir de se dénoncer mutuellement, ce choix étant le plus empreint de rationalité. Conformément à l'énoncé, ils écoperont dès lors de 5 ans de prison chacun. Or, s'ils étaient tous deux restés silencieux, ils n'auraient écopé que de 6 mois chacun. Ainsi, lorsque chacun poursuit son intérêt individuel, le résultat obtenu n'est pas optimal pour chacun deux. Une stratégie collaborative et non individualiste est donc ici bien meilleure.

Le but de la théorie des jeux est d'étudier des notions de stratégie et d'équilibre. La recherche informatique s'est aussi développée avec les jeux sur les graphes qui ont des applications dans plusieurs domaines d'informatique théorique et des mathématiques en permettant de modéliser certains problèmes.

Exemple 2

Modélisation simpliste de la guerre froide :

La Guerre Froide est un bon exemple d'illustration : il s'agit d'un « jeu » dans lequel chaque « joueur » (USA et URSS) peut décider d'attaquer l'adversaire ou non. Avec le formalisme de la théorie des jeux, il apparaît que la meilleure stratégie pour chacun : celle qui minimise les pertes consiste à rester en défense.

		URSS		
		Attaque	Neutre	Défend
USA	Attaque	- - URSS - - USA	- - URSS ++ USA	- URSS + USA
	Neutre	++ URSS - - USA	++ URSS ++ USA	- URSS ++ USA
	Défend	+ URSS - USA	++ URSS - USA	- URSS - USA

1.2 Hypothèses dans le cadre du programme d'ITC

À partir d'une situation donnée, des joueurs prennent à tour de rôle une décision parmi un ensemble fini de décisions possibles, chaque décision amenant une nouvelle situation. Nous considérons principalement dans la suite des **jeux à deux joueurs** satisfaisant certaines conditions :

- chaque joueur a le même vue d'ensemble de la situation (jeu à information complète) ;
- une décision est prise en fonction de la situation présente et pas des situations passées (jeu sans mémoire) ;
- cette décision ne dépend que de la situation et pas du joueur (impartial) ;
- dans une situation donnée, une décision amène toujours a la même nouvelle situation (jeu sans hasard).

Les jeux sont modélisés par des graphes orientés finis. Une situation, ou une **position**, est un sommet du graphe. Une décision est le choix d'un arc amenant à une nouvelle position. Dans un **jeu d'accessibilité**, chaque joueur souhaite atteindre un sous-ensemble de sommets particulier en se déplaçant à tour de rôle sur le graphe. Un jeu d'accessibilité à deux joueurs est modélisé par un graphe biparti.

1.3 Graphes bipartis

Définition 1 : Graphe biparti

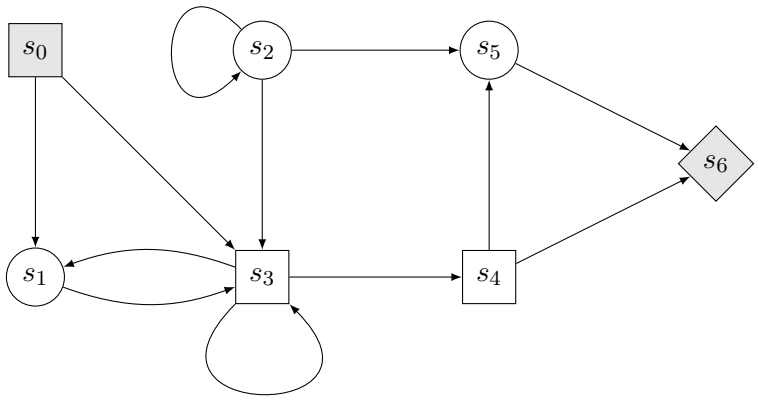
Un graphe G est biparti si l'ensemble de ses sommets peut être divisé en deux sous-ensembles disjoints G_1 et G_2 tels qu'il n'y ait aucune arête distincte entre éléments de G_1 et aucune arête entre éléments de G_2.

2 Vocabulaire de la théorie des jeux

Pour introduire le vocabulaire de la théorie des jeux, considérons le jeu simple suivant :

Exemple 3

C'est un jeu à deux joueurs J_1 et J_2. Un jeton est placé sur le nœud de départ 0 et peut se déplacer en suivant les arcs du graphe. Si le jeton est sur un nœud carré, c'est à J_2 de jouer, sinon c'est au tour de J_1. Pour gagner, J_1 doit atteindre le nœud en forme de losange. Le rôle de J_2 est de l'en empêcher.



2.1 Arène, jeu et partie

Définition 2 : Arène de jeu

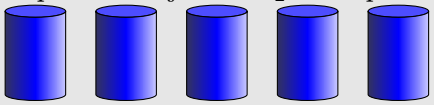
Le graphe de la figure de l'exemple 3 ci-dessus délimite ce que l'on appelle **l'arène de jeu**. On la notera $G(S, A)$ où S désigne l'ensemble de sommets et A l'ensemble des arcs.

Propriété 1 : Arène de jeu

L'arène est dite **bipartite** puisque le graphe est biparti.

Exemple d'application 1

Soit le jeu suivant : Deux joueurs J_1 et J_2 peuvent à tour de rôle retirer 1, 2 ou 3 bâtonnets. Le joueur retirant le ou les derniers bâtonnets perd la partie. (Variante du jeu de Nim). Pour notre exemple, nous disposons de 5 bâtonnets. Le joueur J_1 (sommets circulaires) commence la partie. Le joueur J_2 est représenté par des sommets carrés.



Q1. Construire le graphe de l'arène du jeu.

Retour sur l'exemple 3. Pour gagner la partie, le joueur 1 doit rejoindre le nœud s_6. De façon plus générale, on préfère définir un ensemble Ω contenant les **conditions de gain**. Dans ce cas d'étude, elles sont très simples : $\Omega = \{s_6\}$. On pourrait imaginer des jeux plus compliqués pour lesquels les conditions de gain sont des suites de nœuds à visiter, ou une série d'arêtes à emprunter.

Définition 3 : Jeu mathématique

Un **jeu** mathématique est donc un couple (G, Ω), soit une arène et des conditions de gain

Définition 4 : Partie

Une **partie** est modélisée par une suite d'arcs dans le graphe, correspondant aux actions effectuées par les joueurs, par exemple : $\Lambda = \{(s_0 - s_1), (s_1 - s_3)), (s_3 - s_4)), (s_4 - s_6))\}$ est une partie gagnante pour J_1.

Remarque 1

Le programme d'ITC se limite aux jeux d'accessibilité, c'est-à-dire aux cas où les conditions de gain sont un ensemble de sommets auxquels if faut accéder. Il exsite d'autres types de jeux, comme les jeux de parité dans lesquels la victoire s'obtient lorsque l'autre joueur ne peut plus jouer, ou les jeux de boucles qui consistent à revisiter un nœud déjà visité.

Exemple d'application 2

Q2. Dans notre jeu de bâtonnets, déterminer une partie gagnante pour le joueur J_1.

2.2 Stratégie, positions gagnantes et jeux résolus

Tout l'intérêt de la théorie des jeux consiste à élaborer des **stratégies**. Formellement, une stratégie est une application Ψ qui calcule une action à effectuer en fonction de l'avancée d'une partie. Nous nous limiterons aux stratégies sans mémoire, où l'action choisie ne dépend que de l'action précédente :

$$\begin{array}{rccc} \Psi : & A & \longrightarrow & A \\ & a_{ij} & \longmapsto & \Psi(a_{ij}) = a_{jk} \end{array}$$

avec a_{ij} l'arc entre s_i et s_j

Définition 5 : Stratégie gagnante depuis s_i

On dit qu'une stratégie est gagnante pour le nœud s_i si elle permet au joueur de gagner systématiquement à partir du nœud s_i. Si l'on parvient à trouver une stratégie gagnante pour tous les nœuds $s_i \in S$ de l'arène, le jeu est alors déclaré comme **résolu**.

Définition 6 : Stratégie gagnante

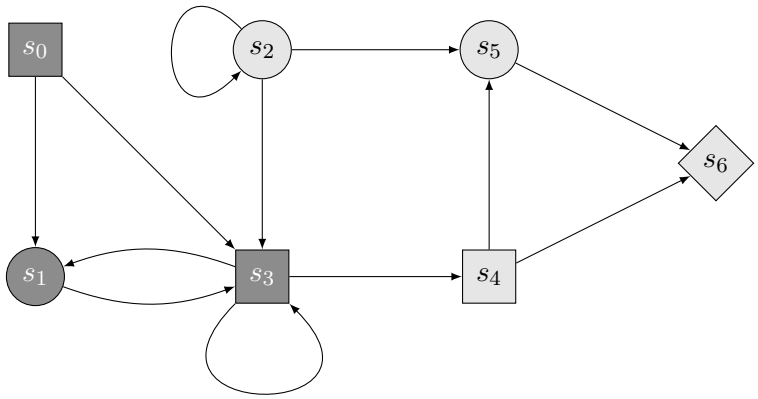
On dit qu'une stratégie est gagnante pour le joueur J_1 si la stratégie est gagnante pour J_1 depuis s_0.

Propriété 2 : Propriété particulière des jeux d'accessibilité

Pour chaque nœud du graphe, il existe une stratégie gagnante pour l'un des joueurs.

Exemple 4

La figure ci-dessous indique les ***positions gagnantes*** *pour J_1 en gris clair, et celles de J_2 en gris foncé. On remarque que tous les nœuds sont coloriés.*



Exemple d'application 3

Q3. Dans notre jeu de bâtonnets, existe-t-il une stratégie gagnante pour le joueur J_2 ? Donner cette stratégie. Existe-t-il une stratégie gagnante pour le joueur J_1 ?

3 Stratégie gagnante sur un jeu d'accessibilité

3.1 Objectif

L'objectif est maintenant de calculer la stratégie gagnante. Sur un jeu d'accessibilité, cela consiste à identifier les positions gagnantes. La stratégie émerge alors rapidement :

- Si le jeton se trouve sur une position gagnante, continuer d'évoluer dans l'ensemble des positions gagnantes jusqu'à atteindre Ω ;
- Sinon jouer aléatoirement (en espérant que l'autre joueur commette une erreur qui vous ramène sur une position gagnante pour vous). C'est d'ailleurs une façon de faire aux dames anglaises. La seule façon de ne pas perdre une partie contre la stratégie gagnante consiste à jouer de manière aléatoire.

Dans ce qui suit, nous allons découvrir une méthode de calcul des positions gagnantes à partir du concept d'**attracteur**.

3.2 Attracteur et rang d'un nœud

On cherche les positions gagnantes de J_1 (celles de J_2 étant faciles à déduire, puisqu'il s'agit du complémentaire à celles de J_1 sur S).

Définition 7 : Attracteur

On appelle **attracteur** la suite (V_n) telle que V_i contienne l'ensemble des nœuds à partir desquels J_1 gagne en moins de i coups.

Propriété 3 : Attracteur

On note S^1, l'ensemble des nœuds jouable par le joueur J_1. On note S^2, l'ensemble des nœuds jouable par le joueur J_2. L'attracteur obéit à la relation de récurrence suivante :

$$V_0 = \Omega$$
$$V_{n+1} = \underbrace{V_n}_{(1)} \cup \underbrace{\{s \in S^1 | \exists \omega \in V_n, (s,\omega) \in A\}}_{(2)} \cup \underbrace{\{s \in S^2 | \forall \omega \in V_n, (s,\omega) \in A\}}_{(3)}$$

- (1) : L'ensemble des nœuds qui permettent à J_1 de gagner en moins de n coups, soit V_n,
- (2) : L'ensemble des nœuds appartenant à J_1 qui permettent de rejoindre un nœud de V_n,
- (3) : L'ensemble des nœuds appartenant à J_2 conduisant obligatoirement à un nœud de V_n.

Le nombre de nœuds étant fini, cette suite converge vers l'**attracteur global** $\lim\limits_{n \in [\![0,|S|]\!]} V_n = V$ qui contient l'ensemble des positions gagnantes pour J_1. On peut en déduire une notion de **rang** pour les positions gagnantes correspondant au nombre de coups minimal permettant de gagner à partir de ce nœud :

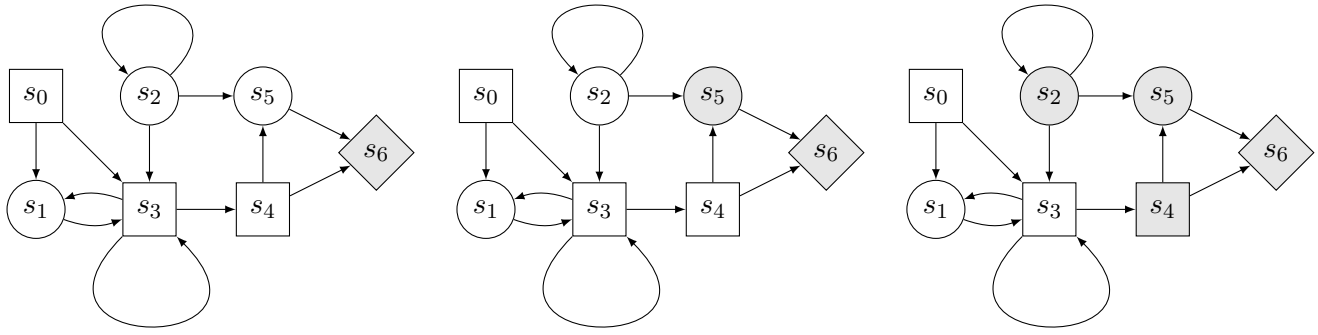
$$\forall s \in V, \qquad \text{rg}\, s = k \text{ tel que } s \in V_k \text{ et } s \notin V_{k-1}$$

Une fois que l'attracteur global V est calculé, la stratégie gagnante consiste à :

- Si le jeton est sur un nœud $s \notin V$, jouer aléatoirement,
- Sinon, si $s \in V$, rejoindre le nœud w tel que $\text{rg}\, w < \text{rg}\, s$.

Exemple 5

Reprenons l'exemple précédent. On calcule les positions gagnantes (ie l'ensemble V) pour le joueur J_1 sur l'arène de démonstration. Pour cela, on applique la formule de récurrence définie plus haut :



(a) Étape 1 : V_0 (b) Étape 2 : V_1 (c) Étape 3 : V_2

- ***Étape 1 :*** *V_0 ne contient que le sommet dans Ω, c'est-à-dire s_6,*
- ***Étape 2 :*** *s_5 permet d'accéder directement à s_6 qui est dans V_0, on l'ajoute donc pour construire V_1. Le nœud s_4 appartient à J_2, et celui-ci peut choisir d'aller soit sur s_6, soit sur s_5 qui n'est pas dans V_0. On ne l'ajoute pas pour le moment.*

- ***Étape 3 :*** *Comme le nœud s_2 appartient à J_1 et permet de rejoindre $s_5 \in V_1$, on l'ajoute à V_2. Cette fois le joueur J_2 en s_4 n'a pas d'autre choix que d'aller dans sur une position gagnante pour J_1 (soit s_6, soit s_5 qui vient de rejoindre A_1), donc on l'ajoute également.*

On a bien trouvé toutes les positions gagnantes pour J_1, ce qui nous donne immédiatement la stratégie gagnante depuis les sommets s_2, s_4 ou s_5. Il n'y a pas de stratégie gagnante depuis s_0. Ainsi, pour ce jeu le joueur J_2 à une stratégie gagnante : Jouer de s_0 à s_3 et boucler sur s_3...

Exemple d'application 4

Q4. Dans notre jeu de bâtonnet, déterminez la stratégie gagnante pour le joueur J_2. Déterminez pour cela l'attracteur.

4 Une IA simple

4.1 Retour sur les graphes et les parcours de ceux-ci

4.1.1 Algorithme de Djisktra

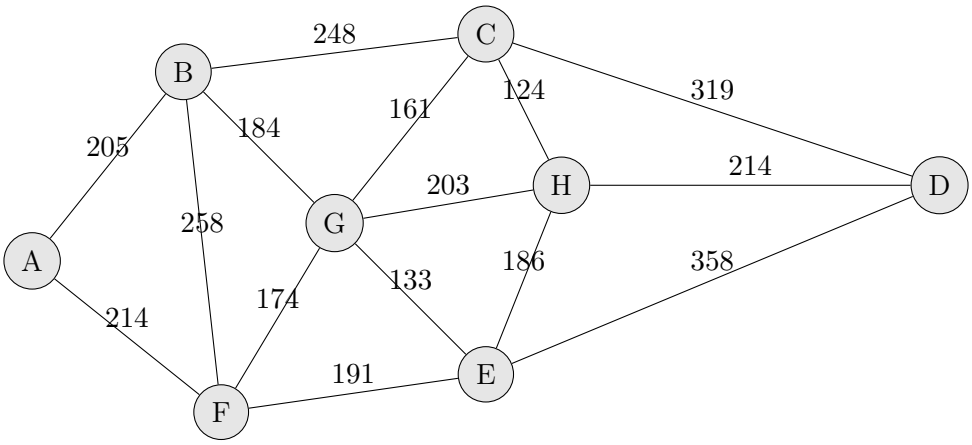


FIGURE 2 – Graphe exemple

A	B	C	D	E	F	G	H	Ét.
								1
								2
								3
								4
								5
								6
								7
								8

A	B	C	D	E	F	G	H	Ét.
								1
								2
								3
								4
								5
								6
								7
								8

A	B	C	D	E	F	G	H	Ét.
								1
								2
								3
								4
								5
								6
								7
								8

A	B	C	D	E	F	G	H	Ét.
								1
								2
								3
								4
								5
								6
								7
								8

A	B	C	D	E	F	G	H	Ét.
								1
								2
								3
								4
								5
								6
								7
								8

A	B	C	D	E	F	G	H	Ét.
								1
								2
								3
								4
								5
								6
								7
								8

A	B	C	D	E	F	G	H	Ét.
								1
								2
								3
								4
								5
								6
								7
								8

A	B	C	D	E	F	G	H	Ét.
								1
								2
								3
								4
								5
								6
								7
								8

L'algorithme de Dijkstra est un algorithme de programmation dynamique.

4.1.2 Algorithme A*

Intérêt

Dans le cadre de la recherche d'un plus court chemin entre un sommet `depart` et un sommet `arrivee`, l'algorithme de Dijkstra peut être légèrement modifié afin de se stopper lorsque le chemin pour arriver au sommet `arrivee` est connu. Si les sommets du graphe représentent des positions géographiques, l'algorithme de Dijsktra ne parait pas très bien adapté. En effet, si l'on souhaite établir un trajet entre Paris et Lille (ville du nord de la France, pour ceux qui l'ignoreraient), il est peu vraisemblable que l'on doive prendre un trajet passant par Orléans (ville au sud de Paris, pour ceux qui l'ignoreraient). Dans l'algorithme de Dijkstra cette possibilité va être testée. En effet, l'exploration des nœuds s'appuie sur un parcours en largeur qui ne se focalise pas sur le sommet à atteindre.

L'algorithme A* va permettre de rendre cette recherche plus efficiente grâce à une **heuristique**.

Définition 8 : Heuristique

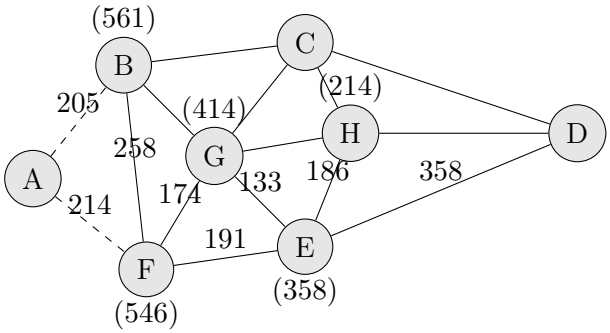
« Une heuristique est un raisonnement formalisé de résolution de problèmes (représentable par une computation connue) dont on tient pour plausible mais non pour certain qu'il conduira à la détermination d'une solution satisfaisante du problème » [a]. En fait, une heuristique est une règle que l'on suit lors de l'application d'un algorithme, règle qui doit permettre d'aboutir à une solution « acceptable » sans que le coût de calcul ne soit prohibitif.

a. *la modélisation des systèmes complexes* (1991), Jean-Louis Le Moigne

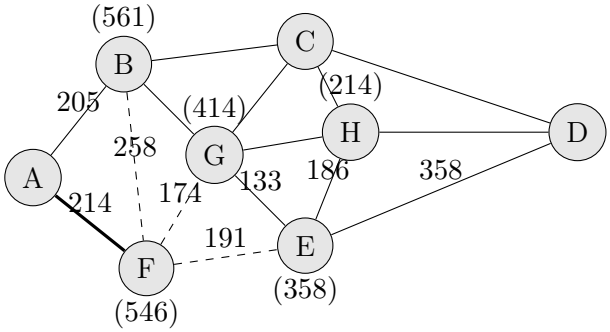
Heuristique de l'algorithme A*

Dans notre cas, l'heuristique choisie va permettre de déterminer à partir d'un sommet i, un sommet $i+1$ dont la distance entre le sommet `depart` et l'estimation du reste à parcourir entre $i+1$ et `arrivee` est minimale. Tout l'« art » de l'algorithme est donc de réussir à bien estimer la distance restante entre un sommet $i+1$ et `arrivee`.

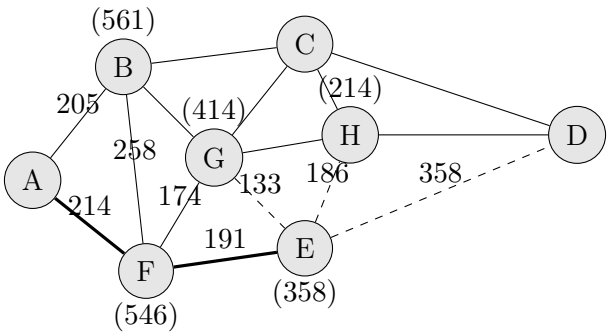
Première illustration de l'algorithme



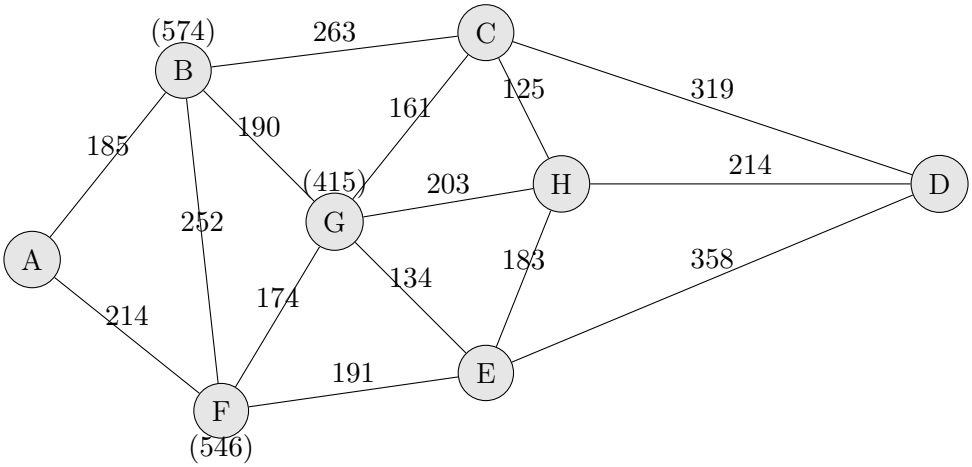
Le prochain sommet choisi sera F car, la distance estimée entre A et D en passant par F est de 760. Tandis que celle en passant par B est 766.



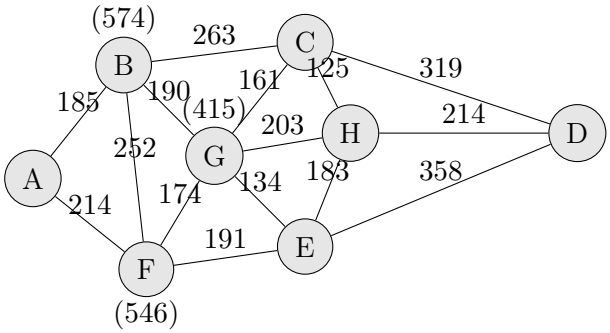
Le prochain sommet choisi sera E car, la distance estimée entre A et D en passant par F et E est de 763. Tandis que celle en passant par F et B est de 1033 et celle passant par F et G est de 802. De plus, le chemin de A à D en passant par B est estimé avoir une distance de 766, il n'est donc toujours pas choisi.

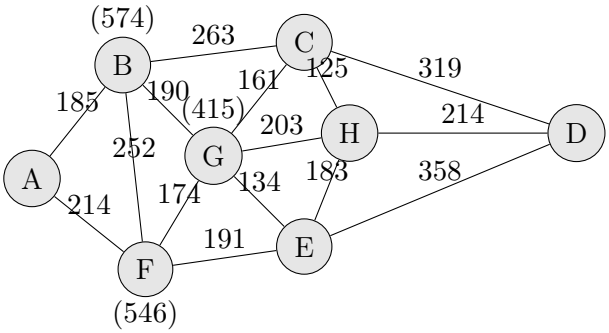


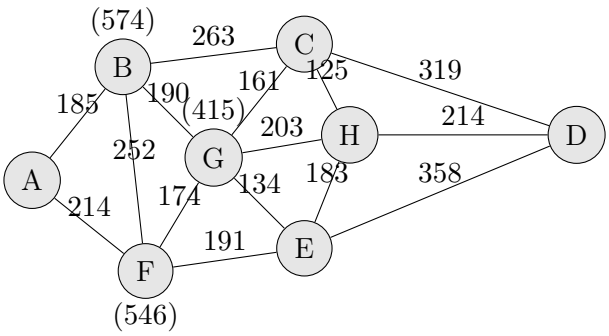
Le prochain sommet choisi sera D, car la distance entre A et D en passant par F, E et D est de 763. Tandis que la distance estimée en passant par F, E et H est 805 et celle passant par F, E et G est estimée à 952. Les autres chemins estimés précédemment, ont toujours une distance supérieure à 763.

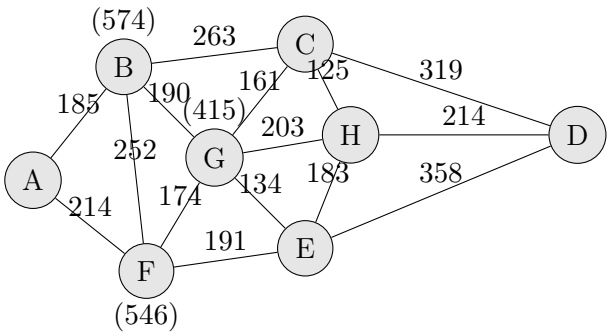


Seconde illustration de l'algorithme









Conclusion

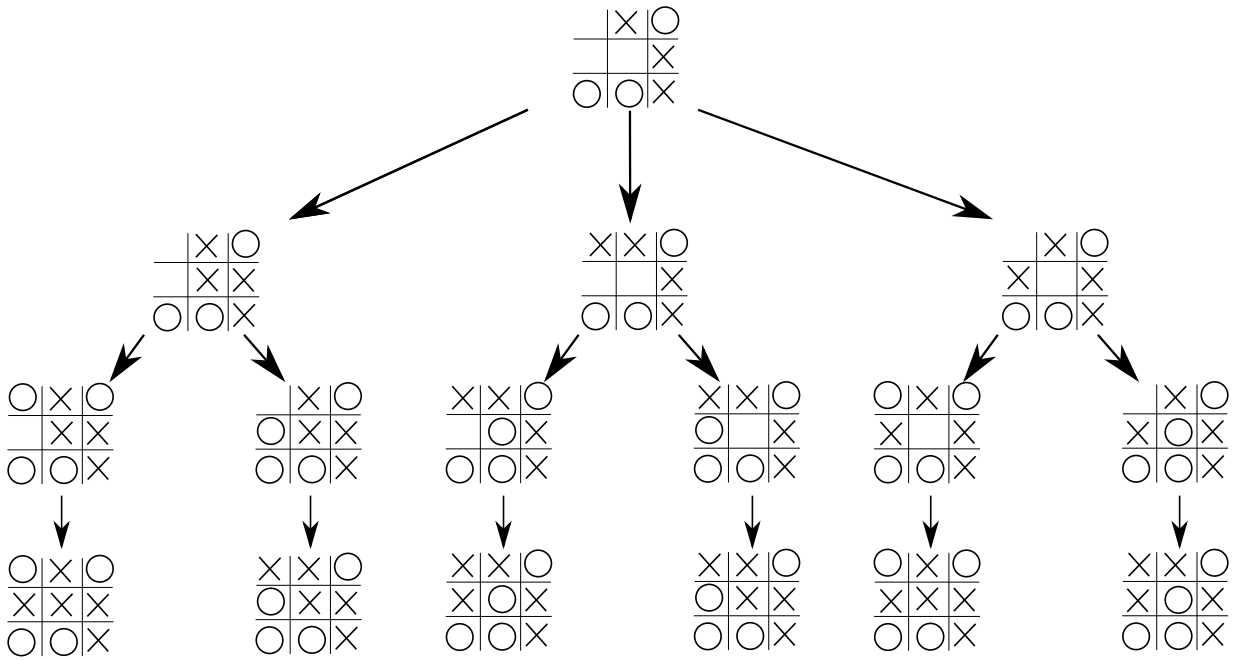
On constate sur cet exemple, que l'on trouve effectivement la bonne réponse et ayant fait bien moins d'exploration qu'avec Dijkstra. (4 itérations contre 8 avec Dijkstra pour le premier exemple). Cependant, son efficacité n'est pas forcément toujours aussi intéressante (voir second exemple).Vous pouvez retrouver des explications et illustrations de l'ensemble des algorithmes de parcours étudiés sur l'excellent site `http://mpechaud.fr/scripts/parcours/index.html`.

4.2 Mettre en place la meilleure stratégie : Algorithme Minimax

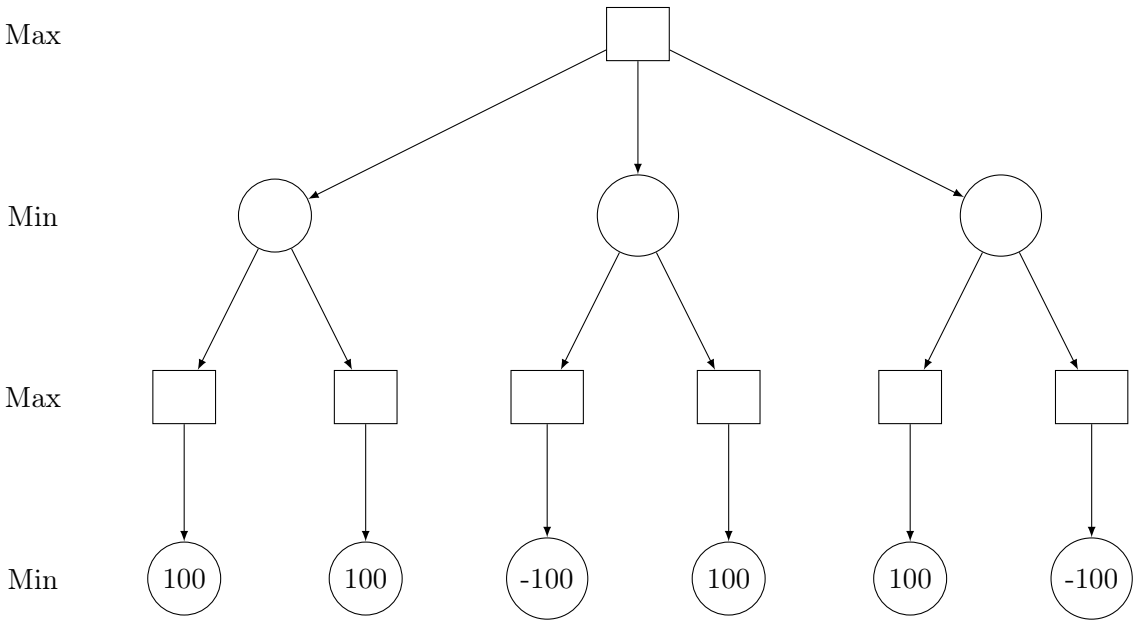
4.2.1 Jeu du Morpion

Le jeu du morpion ou Tic-Tac-Toe est le bon exemple pour illustrer le concept qui va être présenté. Un joueur J_1 (X) (l'ordinateur par exemple) joue contre un joueur J_2 (O) (vous par exemple). Le but pour le joueur J_1 va être de minimiser sa perte maximale sur le jeu (c'est-à-dire dans le pire des cas). D'où le nom algorithme Minimax. Minimiser sa perte maximale peut se retourner en Maximiser son gain minimum si le joueur est dans une position favorable (mais l'algorithme restera le même).

Étudions, cette situation de jeu, où c'est au joueur J_1 (ordinateur) de jouer.



Clairement le joueur J_1 doit jouer dans la situation présentée sur la gauche de l'arbre décisionnel. Dans les autres situations, il n'est pas sûr de gagner. Représentons la situation sous forme de graphe. Le joueur J_1 est représentée sous forme carrée et le joueur J_2 représentée sous forme de cercle. En cas de gain du joueur J_1, la note attribuée est 100, en cas de victoire de J_2, la note est de -100 (défaite de J_1), en cas de match nul, la note attribuée est 0.



Exemple d'application 5

Appliquer l'algorithme Minimax sur le graphe précédent pour montrer que le joueur J_1 est en situation de gagner la partie.

4.2.2 Principe de base de l'algorithme

L'algorithme peut s'écrire sous forme récursive. Détaillons son fonctionnement à partir d'un sommet (quelconque) v dans l'arbre. On notera, $w_1, \ldots, w_n$ ses sommets fils.

- Si v est une feuille de l'arbre, alors la partie est terminée, on renvoie le **score** associé au résultat de celle-ci.
- Si c'est au tour de l'adversaire J_2, on renvoie le score associé au sommet fils w_k qui minimise les gains pour J_1 (ou alors qui maximise les pertes pour J_1), puisque c'est l'objectif de J_2.
- Si c'est au tour de J_1, on renvoie le score associé au sommet fils qui maximise les gains (ou minimise les pertes).

Algorithme 1 : Algorithme Minimax pour J_1

```
Données : v : sommet de l'arbre ;
f : fonction d'évaluation d'une feuille ;
J : nom du joueur possédant v ;
w_1, ..., w_n : les sommets fils de v ;
si v est une feuille de l'arbre alors
    return f(v)
fin
sinon si J est J_1 alors
    Calculer Minimax(w_i) ∀ i;
    return arg max Minimax(w_i)
fin
sinon si J est J_2 alors
    Calculer Minimax(w_i) ∀ i;
    return arg min Minimax(w_i)
fin
```

La fonction d'évaluation est généralement le point le plus difficile à définir : elle doit associer un score à une situation finale (une feuille de l'arbre). On peut imaginer renvoyer un score positif si le joueur J_1 gagne, négatif s'il perd et nul en cas d'égalité.

Lors du jeu du Morpion, il peut y avoir plusieurs situations de jeux amenant le joueur J_1 a gagné la partie. Une fonction d'évaluation permettant de ne pas faire un choix aléatoire entre plusieurs situations est de noter la victoire par : $100-$ nombre de coups pour gagner. En cas de défaite : $-100+$ nombre de coups pour perdre.

4.2.3 Vers une version un peu plus évaluée, l'exemple de Deep Blue

Deep Blue est un superordinateur spécialisé dans le jeu d'échecs par adjonction de circuits spécifiques, développé par IBM au début des années 1990. Il a perdu un match en 1996 (2-4) contre le champion du monde d'échecs de l'époque Garry Kasparov.

Explosion combinatoire

L'algorithme de Deep Blue s'appuie sur l'algorithme Minimax. Deep Blue peut anticiper jusqu'à 11 coups. Il se heurte alors au phénomène d'explosion combinatoire. Le nombre de calculs augmente comme une exponentielle et devient inimaginable. Selon les estimations, 32 ans de calculs serait nécessaires pour tester l'ensemble des combinaisons sur 11 coups... Ce n'est donc pas possible... (et cela n'est que pour 11 coups). Une heuristique adaptée et un élagage de l'arbre de résolution (éviter de calculer des coups inutiles) sont alors alors nécessaires.

Fonction heuristique

Puisqu'il n'est pas possible d'étudier l'ensemble de l'arbre de décision, il est nécessaire de « juger » une situation intermédiaire. Comment évaluer le gain d'un joueur dans une situation donnée ? En effet, après 11 coups joués, beaucoup d'entre eux n'aboutissent pas à une fin de partie (et celles qui aboutissent peuvent être des successions de coups « idiots » : une IA se faisant avoir sur le coup du Berger ne serait pas vraissemblable...). Il faut donc être capable d'évaluer une situation de jeu pour savoir si elle est intéressante ou non. Il faut donc associer une fonction mathématique à une situation du jeu. **Cette fonction mathématique est une heuristique.**

Exemple 6

Sur le jeu du Morpion, l'algorithme Minimax doit anticiper 9 coups si le joueur J_1 commence à jouer ou 8 si c'est le joueur J_2, ce qui fait 362 880 *ou* 40 320 *coups à anticiper. Ces nombres sont raisonnables et la partie peut être jouée par un ordinateur dans un temps raisonnable. Mais dans ce cas, le jeu est imbattable, ce qui est un peu dommage pour un jeu...*
Par contre, pour rendre la partie intéressante, on peut essayer de ne faire anticiper à l'IA que 3 ou 4 coups. Dans ce cas, l'ordinateur pour faire des choix intelligents, il est nécessaire d'évaluer une situation. Dans le cas du Morpion, on peut, pour chaque ligne, chaque colonne et chaque diagonale calculer un score (8 scores au total à calculer)

- *(*100$-$*nombre de coups à jouer pour gagner) points si les 3 mêmes pions de J_1 sont alignés ;*
- 10 *points si 2 pions de J_1 sont sur la même ligne, colonne ou diagonale et que la dernière case est vide ;*
- 1 *point si 1 pion de J_1 est seul sur une ligne, colonne ou diagonale ;*
- *0 si la ligne est vide ;*
- *(*$-$100$+$*nombre de coups à jouer pour gagner) points si les 3 mêmes pions de J_2 sont alignés ;*
- -10 *points si 2 pions de J_2 sont sur la même ligne, colonne ou diagonale et que la dernière case est vide ;*
- -1 *point si 1 pion de J_2 est seul sur une ligne, colonne ou diagonale.*

Elagage alpha-beta

Dans cet algorithme Minimax, on peut coupler l'algorithme d'élagage alpha-beta (hors programme). L'élagage alpha-beta consiste à ne pas explorer certaines des branches de l'arbre dont on sait qu'elles n'interviendront pas dans l'évaluation de la position courante.

Peut-on alors coder un algorithme Minimax pour tous les jeux ?

L'algorithme Minimax s'applique pour les jeux à deux joueurs à somme nulle (autant de gain que de perte) et à information complète. Il peut néanmoins être étendu à d'autres type de jeux.

Cependant, bien que plutôt facilement codable, il nécessite une intelligence qui elle pour le coup n'était au départ pas artificielle : celle de créer une fonction heuristique vraissemblable. Ainsi, pour le jeu d'échecs, pour réaliser le code de Deep Blue, il a nécessairement fallu des joueurs d'échecs de haut niveau pour établir le score à des situations de plateaux données. Ce qu'un excellent programmeur informatique mais novice au jeu d'échecs **était** incapable de faire...Je dis bien « était » car maintenant par une intelligence artificielle de type renforcement, on pourrait créer cette fonction heuristique (mais bon autant faire directement l'intelligence artificielle sur le jeu... comme il a été fait pour AlphaGo).

TD Info - Théorie des jeux

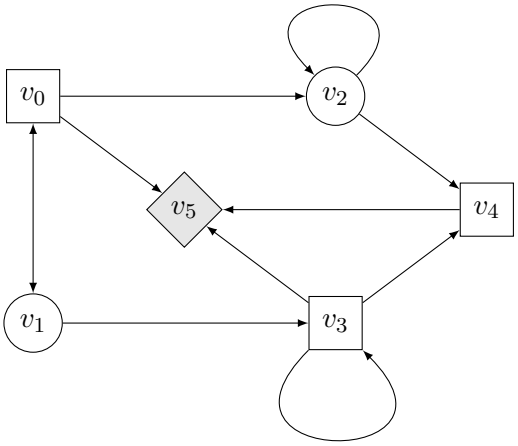
I - Recherche d'attracteur

1 Présentation

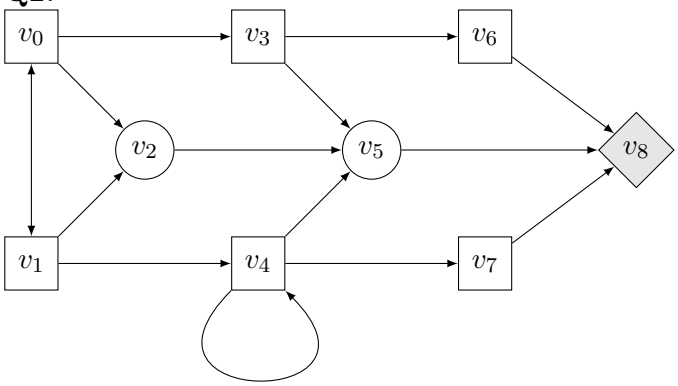
Objectif

On considère des jeux d'accessibilité pour lesquels les conditions de gain sont représentées par des nœuds en forme de losange. Déterminer les positions gagnantes du joueur 1 (jouant les nœuds ronds) sur les arènes ci-dessous en détaillant votre méthode.

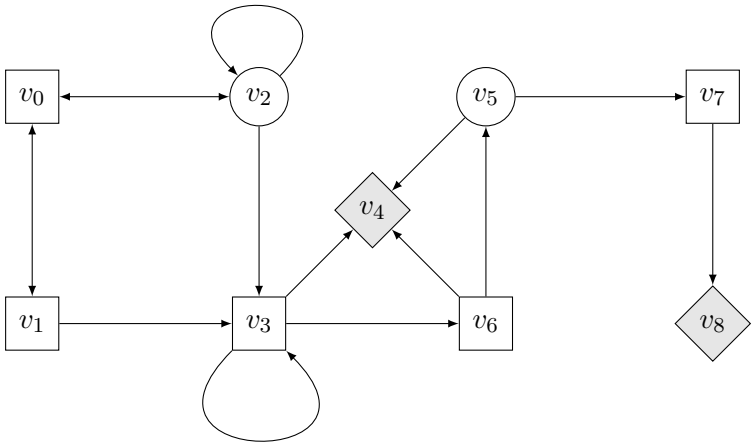
Q1.



Q2.



Q3.



II - Programme de recherche d'attracteur

1 Présentation

Objectif

On considère des jeux d'accessibilité pour lesquels les conditions de gain sont représentées par des nœuds en forme de losange. On souhaite déterminer à l'aide d'un programme Python, les positions gagnantes du joueur 1 (jouant les nœuds ronds) sur les arènes ci-dessus en détaillant votre méthode.

On se propose de représenter l'arène de jeu sous forme de graphe renseigné par un dictionnaire d'adjacence. La clé d'un élément du dictionnaire correspondra au numéro de nœud et la valeur associée sera un tuple de deux éléments : le premier élément sera un entier indiquant le numéro du joueur auquel appartient le nœud et le deuxième élément sera la liste des successeurs de ce nœud.

Q1. Reprendre le premier jeu proposé dans l'exercice précédemment et indiquez le dictionnaire d'adjacence de celui-ci.

La recherche d'attracteurs nécessite, un parcours de graphe en s'intéressant aux prédécesseurs des nœuds faisant partie de l'ensemble des attracteurs.

Q2. Programmez une fonction `recherche_predecesseur`, prenant en argument d'entrée le dictionnaire d'adjacence créé précédemment, le numéro du nœud dont on cherche les prédécesseurs. Cette fonction renverra la liste des prédécesseurs du nœud passé en argument.

Q3. Créez une fonction `ajout_attrac_J1` permettant d'ajouter un nœud du joueur J_1 à la liste des attracteurs V s'il est admissible à en faire partie. Les arguments d'entrée de la fonction sont le dictionnaire d'adjacence, la liste des attracteurs V actuelle et le nœud que l'on veut tester pour l'ajouter à la liste des attracteurs.

Q4. Créez une fonction `ajout_attrac_J2` permettant d'ajouter un nœud du joueur J_2 à la liste des attracteurs V s'il est admissible à en faire partie. Les arguments d'entrée de la fonction sont le dictionnaire d'adjacence, la liste des attracteurs V actuelle et le nœud que l'on veut tester pour l'ajouter à la liste des attracteurs.

Afin de déterminer la liste d'attracteur, il faut réaliser un parcours de graphe.

Q5. Réalisez une fonction s'appuyant sur le principe d'un parcours de graphe en largeur permettant d'obtenir la liste des attracteurs. Les paramètres d'entrée de la fonction seront le dictionnaire d'adjacence du graphe et l'initialisation de la liste des attracteurs.

Annexe : Utilisation de deque

- Initialiser une `deque` : `D=deque(iterable)`. Par exemple, `D=deque([5])` ou `D=deque("pile")` (dans ce cas le premier élément est `"p"` et le dernier élément est `"e"`) ;
- Ajouter un élément en fin de `deque` : `D.append(elem)`. Par exemple, `D.append(4)` ;
- Ajouter un élément en début de `deque` : `D.appendleft(elem)`. Par exemple, `D.appendleft(4)` ;
- Ajouter plusieurs éléments en fin de `deque` : `D.extend(iterable)`.
  Par exemple, `D.extend([1,2,3])` ;
- Ajouter plusieurs éléments en début de `deque` : `D.extendleft(iterable)`.
  Par exemple, `D.extendleft([1,2,3])` ;
- Récupérer un élément en fin de `deque` : `elem=D.pop()`, on précise que cet élément `elem` est automatiquement supprimé du `deque` D.
- Récupérer un élément en début de `deque` : `elem=D.popleft()`, on précise que cet élément `elem` est automatiquement supprimé du `deque` D ;
- Vider le `deque` créé : `D.clear()` ;

III - Intelligence artificielle et jeu du Morpion

1 Présentation

Objectif

Dans cet exercice, on va mettre en place l'algorithme de Minimax sur le jeu du Morpion. L'objectif n'est pas de coder entièrement l'algorithme mais principalement de l'adapter à un code de jeu déjà existant.

Le jeu du Morpion (ou jeu Tic-Tac-Toe) est largement répandu et consiste à aligner sur une ligne, une colonne ou une diagonale un même symbole `'X'` et `'O'`. Ici, vous allez jouer contre votre ordinateur. Celui-ci joue avec le symbole `'X'` et vous avec le symbole `'O'`.

Q1. Ouvrir le fichier `jeu_morpion.py`. Lancez une partie et jouez contre l'ordinateur. Normalement vous devriez gagner assez facilement.

Q2. Expliquez le fonctionnement global du code proposé. Vous devez notamment expliquer :

- comment est assuré l'alternance Joueur/Ordinateur ;
- comment l'ordinateur joue un coup et l'intérêt de la boucle `while` ;
- pourquoi, on utilise également une boucle `while` lors de la proposition de jeu du joueur ;
- ce qui se passe si le joueur se trompe en tapant au clavier son choix de jeu ;
- comment se termine la partie.

Afin d'améliorer le choix de jeu de l'ordinateur, on se propose de coder l'algorithme Minimax. La fonction heuristique choisie permettant d'évaluer une situation de jeu pour réaliser l'algorithme Minimax est la suivante :

- (100−nombre de coups à jouer pour gagner) points si les 3 mêmes pions `'X'` sont alignés. ;
- 2 points si 2 pions `'X'` sont sur la même ligne, colonne ou diagonale et que la dernière case est vide ;
- 1 point si 1 pion `'X'` est seul sur une ligne, colonne ou diagonale ;
- 0 si la ligne est vide ;
- (−100+nombre de coups à jouer pour gagner) points si les 3 mêmes pions `'O'` sont alignés ;
- −2 points si 2 pions `'O'` sont sur la même ligne, colonne ou diagonale et que la dernière case est vide ;
- −1 point si 1 pion `'O'` est seul sur une ligne, colonne ou diagonale.

Q3. Ouvrir le fichier `jeu_morpion_minimax_eleve.py`. Ce fichier est à compléter pour faire fonctionner l'algorithme.

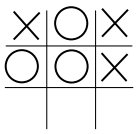
Q4. En vous inspirant de la fonction `detection_victoire_ligne_colonne_diag`, créez la fonction `score_ligne(line,nb_coups)` avec `line` une ligne du jeu qui a été extraite de la grille et `nb_coups` le nombre de coups qui a été joué depuis une situation initiale.

Q5. En vous inspirant de la fonction `detection_victoire`, créez la fonction `score_total(grille,nb_coups)` avec `nb_coups` le nombre de coups qui a été joué depuis une situation initiale.

Pour cette intelligence artificielle, il est nécessaire d'indiquer à l'ordinateur la profondeur de jeu qu'il peut anticiper. Cette variable est indiquée par `profondeur`. Cependant, vers la fin de partie, on ne peut pas anticiper autant de coups avec la profondeur souhaitée qu'il n'en reste à jouer.

Q6. Donnez la relation permettant de mettre à jour la valeur de la profondeur `profondeur=min(...,...)`. Complétez alors le code de la fonction `main_jeu` du code `jeu_morpion_minimax_eleve.py`.

Dans l'algorithme du Minimax, il est nécessaire de déterminer les feuilles de l'arbre de décision. Pour bien comprendre comment déterminer ce qu'est une feuille de l'arbre nous allons traiter un exemple. Voici, l'exemple à traiter pour une profondeur de 2 et 3 :



Q7. Dessiner l'arbre de jeu d'une profondeur de 3. C'est au tour du joueur (`'O'`) de jouer. Conclure sur comment une feuille est déterminée.

Q8. D'après l'algorithme présenté dans votre cours, compléter l'algorithme `minimax` permettant à l'ordinateur de jouer un coup.

Q9. Vérifiez que votre algorithme fonctionne bien en effectuant plusieurs parties avec l'ordinateur pouvant jouer jusqu'à à une profondeur de 5.