Projet ALGO1

Philippe Rannou

1 Algorithme A^*

1.1 Définitions, notations et énoncé du problème

Définitions et notations

- On note $G = (S, V)$, le graphe orienté où S est l'ensemble des sommets (représentés par des entiers allant de 0 jusqu'à $|S| - 1$) et V l'ensemble des arcs orientés de G.
- On définit une fonction de coût $c : V \to \mathbb{R}^+$ sur un graphe G comme une fonction qui associe un coût réel positif à chaque arc du graphe.
- Un *chemin* de G est une suite de sommets $s_1, \ldots, s_n$ telle que $\forall i \in [1, n-1]\, s_i \to s_{i+1} \in V$.
- Le coût d'un chemin est la somme des coûts des arcs $s_i \to s_{i+1}$.

Problème du plus court chemin Étant donné un graphe $G = (S, V)$ connexe muni d'une fonction de coût $c : V \to \mathbb{R}^+$, ainsi que deux sommets *source* et *cible*, on souhaite trouver le chemin de coût minimum reliant *source* à *cible* à travers G.

1.2 Principe de l'algorithme

L'algorithme A^* est un algorithme de recherche de meilleur chemin de type *meilleur d'abord*. L'idée est de parcourir le graphe G depuis le sommet *source* jusqu'au sommet *cible* en se deplaçant dans les 'meilleurs' sommets ; i.e. ceux ayant le plus petit coût global : coût du chemin depuis la *source* + coût estimé jusqu'à la *cible*. Au départ, on se place dans le sommet *source* puis, tant qu'on ne se trouve pas dans le sommet *cible*, on évalue le coût global des successeurs du sommet courant : le coût depuis la *source* est la somme des coût des arcs rencontrés et on estime le coût jusqu'à la *cible* à l'aide d'une heuristique. On choisit alors le sommet non visité le moins coûteux parmi ceux rencontrés et on répète la boucle.

Remarque : l'algorithme de Dijkstra est un cas particulier de A^*, dans lequel l'heuristique est la fonction nulle.

Une *heuristique* est une fonction évaluant (approximativement) un coût. Dans notre problème, l'heuristique nous permettra d'évaluer approximativement le coût du chemin minimum à partir d'un sommet jusqu'au sommet *cible*. On dit qu'une heuristique est *exacte* si les coûts estimés des sommets sont les coûts réels, qu'elle *sous-estime* si les coûts estimés sont inférieurs aux coûts réels et qu'elle *sur-estime* s'il existe un sommet pour lequel le coût estimé est supérieur au coût réel.

On dira qu'une heuristique est *admissible* pour un graphe G si elle permet à l'algorithme A^* de trouver le chemin de coût minimum.

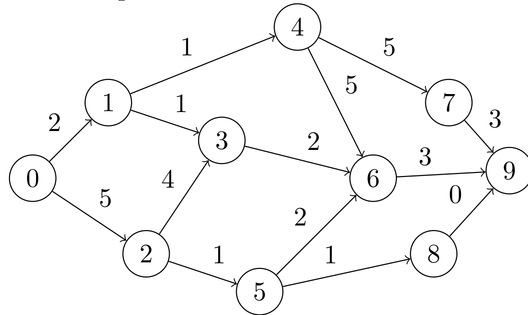
Algorithm 1 $A^*(G, s, t, h)$: retourne le chemin minimal du sommet s au sommet t dans G avec une heuristique h

```
sommet_courant ← s
valeurSommets ← tableau(|S|,∞)
valeurSommets[s] ← 0
peres ← tableau(|S|,−1)
while sommet_courant ≠ t do
   marquer(sommet_courant)
   miseAJourSuccesseurs(G,valeurSommets,sommet_courant,peres)
   sommet_courant ← u | valeurSommet(u)+h(u, t) minimal et u non marqué
end while
resultat ← [t]
sommet_courant ← t
while sommet_courant ≠ s do
   sommet_courant ← peres[sommet_courant]
   resultat ← sommet_courant : : resultat
end while
return resultat
```

Algorithm 2 *miseAJourSuccesseurs*(G,valeurSommets, *sommet*, *peres*) : met à jour la valeur des successeurs de *sommet* ainsi que *peres*

```
for s ∈ Successeurs(sommet, G) do
   val ← valeurSommets[sommet]+cout(sommet, s, G)
   if valeurSommets[s] > val then
      valeurSommets[s] ← val
      peres[s] ← sommet
      if marqué(s) then
         démarquer(s)
      end if
   end if
end for
```

Question 1 – Appliquer l'algorithme A^* sur le graphe G avec comme source le sommet 0, comme cible le sommet 9 et l'heuristique h.



Graphe G

$(1, 9)$	5
$(2, 9)$	5
$(3, 9)$	5
$(4, 9)$	4
$(5, 9)$	3
$(6, 9)$	3
$(7, 9)$	3
$(8, 9)$	0
$(9, 9)$	0

heuristique h

Question 2 – h est-elle exacte/sous-estimante/sur-estimante ? admissible ?

Question 3 – Démontrez que si h sous-estime alors elle est admissible.

Question 4 – Dans le cas où le graphe est un arbre binaire complet, avec pour source la racine pour cible une feuille et pour heuristique la fonction nulle, quelle est la complexité en fonction du nombre de sommets de l'arbre dans le pire cas d'exécution ? Que devient la complexité avec une heuristique exacte ?

2 Un algorithme pour les jeux à deux joueurs : *minimax*

Lors d'un jeu contre un adversaire, on élabore une stratégie qui nous permettra de remporter la partie (ou la plus grande récompense possible) quelles que soient les actions de l'autre joueur. Le but ici est d'étudier un algorithme qui automatise le calcul d'une stratégie pour un certain type de jeux à deux joueurs.

Nous nous plaçons dans le cadre des jeux à deux joueurs, à information complète (les deux joueurs ont accès à toutes les informations), non stochastiques (sans hasard), à somme nulle (si l'un des joueurs gagne, l'autre perd, il peut y avoir des matchs nuls) et où les joueurs jouent chacun leur tour. Citons les échecs, les dames, le puissance 4, le morpion,...

2.1 Principe de l'algorithme

Définitons et notations

- On dénote les deux joueurs par $Joueur1$ et $Joueur2$, $Joueur1$ commence la partie.
- Un *état* est une configuration de la partie avant un coup d'un joueur, il est soit terminal (la partie est finie) soit contrôlé par l'un des joueurs.

L'algorithme *minimax* a pour but de déterminer dans un état donné quel coup permettra au joueur contrôlant cet état d'avoir la meilleure récompense possible. Il s'appuie sur le principe suivant : le coup de l'adversaire est potentiellement le pire possible. Aussi, le joueur qui cherche le meilleur coup évalue les états terminaux :

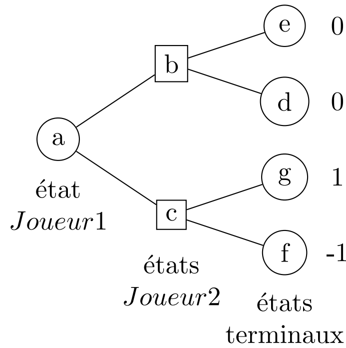
- 1 si l'état est gagnant pour lui,
- -1 s'il est perdant,
- 0 si c'est un match nul.

Il choisit ensuite le coup qui lui assure la récompense minimale la plus grande (voir figure 1). Ce principe est vérifié pour les jeux à somme nulle car quand l'un des joueur gagne, l'autre perd ; il est donc sensé de considérer que le joueur adverse jouera toujours le pire coup pour lui.

L'algorithme est du type 'bottom-up' car pour déterminer quel coup jouer, on part de l'évaluation des états terminaux du graphe et on fait remonter l'information jusqu'à l'état qui nous intéresse.

En pratique, l'algorithme crée un graphe avec trois type d'états :

- les états contrôlés par $Joueur1$;
- les états contrôlés par $Joueur2$;
- les états terminaux (sans successeurs).



Dans cet exemple, $Joueur1$ cherche le meilleur coup à jouer dans l'état a (pour arriver en b ou c). Il évalue les états terminaux d, e, f et g. Par exemple, après le coup de $Joueur2$, s'il se retrouve en d, il aura une récompense de 0, s'il se retrouve en g, il aura une récompense de 1. Dans l'état b, $Joueur2$ peut l'envoyer en d ou en e, avec donc une récompense minimale de 0, dans l'état c, il peut l'envoyer en f ou en g, avec donc une récompense minimale de -1. $Joueur1$ jouera alors en b, car la récompense minimale qu'il obtiendra sera supérieure à celle obtenue en c.

FIGURE 1 – Exemple minimax

Les transitions entre états représentent les coups possibles. Les joueurs jouent chacun leur tour, il y a donc toujours alternance entre états du $Joueur1$ et états du $Joueur2$. La construction d'un tel graphe est possible car c'est un jeu à information complète : on sait quels coups sont possibles dans tel état.

Lors de l'exécution de l'algorithme par le $Joueur1$ par exemple, il va évaluer les états terminaux à -1, 0 ou 1 suivant s'ils sont perdants, nuls ou gagnants pour le $Joueur1$. Ensuite, de manière récursive, dans un état contrôlé par le $Joueur1$, l'algorithme va évaluer ses successeurs (des états contrôlés par le $Joueur2$) et choisir le coup qui maximise la récompense. Pour évaluer un état contrôlé par le $Joueur2$, l'algorithme va évaluer ses successeurs et choisir le coup qui minimise la même fonction récompense (on évalue le pire cas).

Remarque : on note $etat + c$ l'état du jeu après avoir jouer c dans $etat$

Algorithm 3 minimax($etat$,$joueur$) : retourne le couple $meilleurCoup$,$meilleureValeur$ pour l'adversaire de $joueur$

```
if EstTerminal(etat) then
  return •,val(etat,joueur)
else
  meilleurCoup ← •
  _,meilleureValeur ← +∞
  for c ∈ coupsPossibles(etat) do
    _,valeur ← maximin(etat + c,autreJoueur(joueur))
    if valeur < meilleureValeur then
      meilleureValeur ← valeur
      meilleureCoup ← c
    end if
  end for
  return meilleurCoup,meilleureValeur
end if
```

Algorithm 4 maximin($etat$,$joueur$) : retourne le couple $meilleurCoup$,$meilleureValeur$ pour joueur

```
if EstTerminal(etat) then
  return •,val(etat,joueur)
else
  meilleurCoup ← •
  _,meilleureValeur ← −∞
  for c ∈ coupsPossibles(etat) do
    _,valeur ← minimax(etat + c,autreJoueur(joueur))
    if valeur > meilleureValeur then
      meilleureValeur ← valeur
      meilleureCoup ← c
    end if
  end for
  return meilleurCoup,meilleureValeur
end if
```

Algorithm 5 minimaxStrategy($etat$,$joueur$) : retourne le $meilleurCoup$ à jouer dans l'état pour $joueur$

```
meilleurCoup,_ ← maximin(etat,joueur)
return meilleurCoup
```

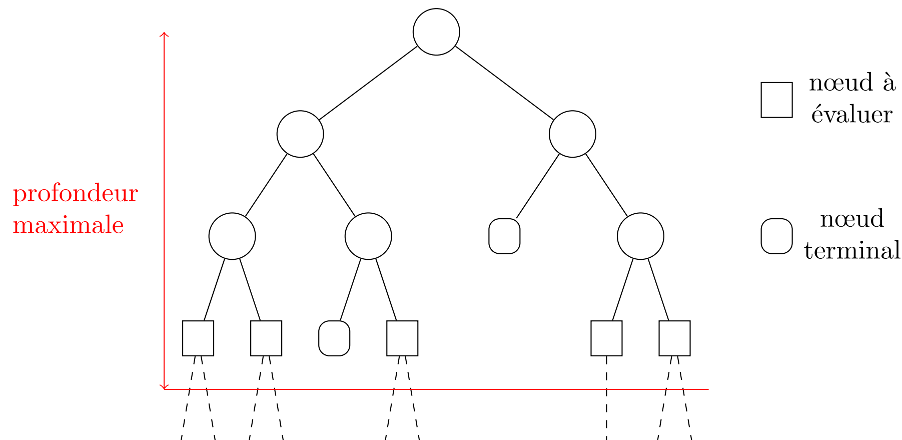
Question 5 – Cet algorithme termine-t-il ? Si oui pourquoi ? Si non pourquoi et quelle condition faut-il ajouter pour qu'il termine ?

Question 6 – Quelle est la complexité, en fonction de la taille (nombre d'états) du graphe, de cet algorithme ?

Question 7 – Dans le cas d'un puissance 4 de taille $n \times m$ quelle est l'ordre de grandeur de la taille du graphe construit par l'algorithme (on ne demande pas une évaluation précise, juste un ordre de grandeur) ?

2.2 Introduction d'une fonction d'évaluation

Pour éviter de parcourir tout le graphe, nous allons nous fixer une profondeur maximale de recherche et nous allons estimer la valeur des états de profondeur maximale quand ils ne sont pas terminaux.



De même que l'évaluation des états terminaux, l'évaluation des états de profondeur maximale se fera toujours par rapport au joueur jouant à la racine et elle sera d'autant plus élevée que l'état sera avantageux pour ce joueur.

Remarque : dans la suite, on évaluera une situation perdante à $-\infty$, une gagnante à $+\infty$ et une fonction d'évaluation sera une fonction de $S \to \mathbb{R}$

Question 8 – Réécrivez cet algorithme en pseudo-code puis implémentez-le en Caml pour le Morpion et le Puissance4 (vous pouvez utiliser les fonctions de *Mechanism.mli : *getPossiblesMoves* pour les coups possibles depuis l'état courant, *play* pour joueur un coup et *unplay* pour revenir en arrière).

2.3 Élagage alphabeta

Quand on parcourt l'arbre de jeu, on s'aperçoit qu'il n'est pas nécessaire de parcourir toutes ses branches.

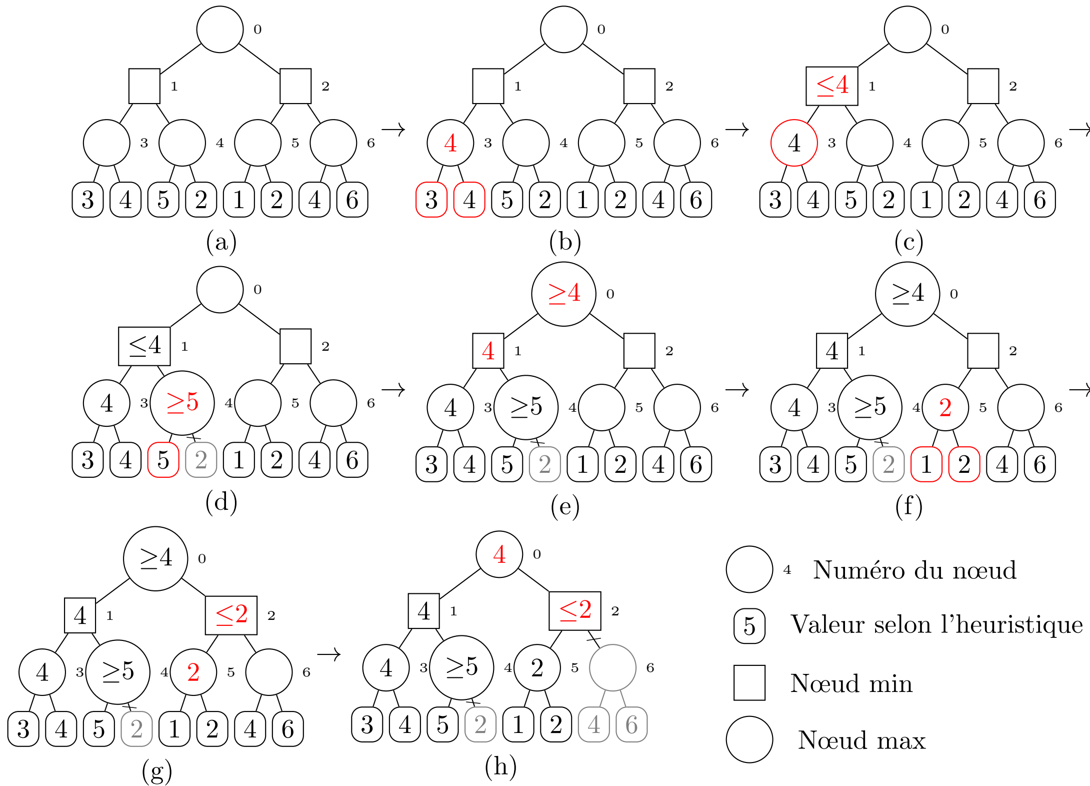


Figure 2 – Exemple d'exécution de l'algorithme alphabeta

2.3.1 Principe de l'algorithme

Dans la figure 2, on cherche à trouver le meilleur coup pour le joueur qui essaie de maximiser sa récompense à la racine $_{(0)}$. On se fixe comme profondeur de recherche 2, le problème se ramenant donc à la figure (a) :

1. dans un premier temps, on cherche à évaluer le premier fils de la racine $_{(1)}$;
2. cela nous amène à évaluer son premier fils $_{(3)}$(lui-même évalué en faisant le max des valeurs selon l'heuristique de ces fils) : figure (b) ;
3. on peut en déduire que la valeur de $_{(1)}$ sera inférieure à celle de $_{(3)}$(puisque c'est un nœud min) : figure (c) ;
4. on évalue ensuite le deuxième fils $_{(4)}$, et on s'aperçoit que la première valeur rencontrée est 5, on peut donc en conclure que la valeur de $_{(4)}$ sera supérieure à 5 et ce sans parcourir les autres branches (figure (d)) ;

5. ainsi dans la configuration (1), le joueur min préfèrera le premier fils (3) : la valeur de (1) sera donc 4 (figure (e)) ;
6. de même, on en déduit que la valeur de la racine sera supérieure à 4 et on commence à parcourir la branche du nœud (2) ;
7. après avoir calculer la valeur de (5), on s'aperçoit que la valeur de (2) sera inférieure à 2 (figure (g)) ;
8. on en déduit, sans parcourir la dernière branche, que la valeur de la racine sera 4.

On a donc pu trouver la valeur de la racine en n'explorant qu'une partie de l'arbre.

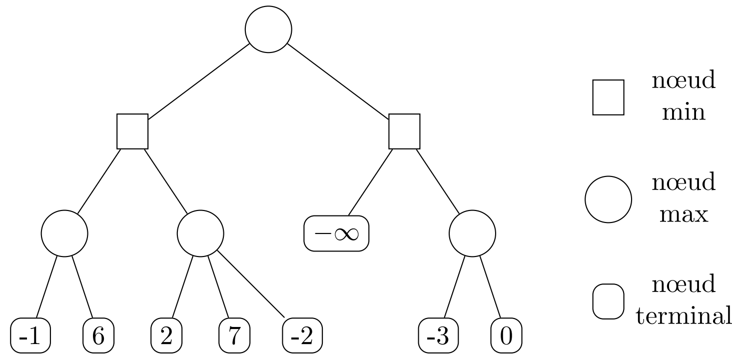
2.3.2 pseudo-code

Lors du parcours de l'arbre, nous gardons en mémoire un encadrement de la valeur de chaque nœud. Au début de l'exécution, l'encadrement sera $[-\infty, \infty]$ et au fur et à mesure du calcul, nous réduirons cet encadrement.

Algorithm 6 alphabeta($etat$,$alpha$,$beta$) : retourne un couple $meilleurCoup$,$meilleureValeur$ pour le joueur dans l'état $etat$

```
if EstFinal(etat) then
  return •,val(etat)
else
  if EstMin(etat) then
    Val ← +∞
    meilleurCoup ← •
    for c ∈ coupsPossibles(etat) do
      _,v ← alphabeta(etat + c,alpha,beta)
      if Val > v then
        meilleurCoup ← c
        Val ← v
        if Val ≤ alpha then
          return c,Val {coupure alpha}
        end if
      end if
      beta ← Min(beta,Val)
    end for
    return meilleurCoup,Val
  else
    Val ← −∞
    meilleurCoup ← •
    for c ∈ coupsPossibles(etat) do
      _,v ← alphabeta(etat + c,alpha,beta)
      if Val > v then
        meilleurCoup ← c
        Val ← v
        if Val ≥ beta then
          return c,Val {coupure beta}
        end if
      end if
      alpha ← Max(alpha,Val)
    end for
    return meilleurCoup,Val
  end if
end if
```

Question 9 – Effectuer l'élagage alphabeta de l'arbre suivant :



Question 10 – Implémentez cet algorithme en Caml et testez-le sur les jeux fournis (Puissance4 et Morpion)

3 Heuristique

Question 11 – Élaborez et implémentez une heuristique pour le puissance 4.

4 Apprentissage

L'apprentissage par renforcement est une autre forme d'élaboration de stratégies. C'est une forme 'naturelle' d'apprentissage du type échec - pénalité / réussite - récompense. D'un point de vue informatique, on simule des expériences, on évalue le résultat numériquement et on fait en sorte de choisir des coups menant à des résultats élevés. Dans le cas d'un jeu à somme nulle, on peut par exemple considérer que le gain d'une partie est une récompense positive (1), un match nul, une récompense nulle et une partie perdue une récompense négative (pénalité) (-1).

4.1 Formalisation

On considérera que l'environnement est déterministe : jouer un coup donné dans un état donné mène toujours au même état ; à information complète : on connait tous les coups possibles dans toutes les configurations.

Définition-notations

- On note S l'ensemble des états du jeu.
- On note C l'ensemble des coups.
- On définit $A : S \to 2^C$ la fonction qui à un état s associe l'ensemble des coups possibles dans s.
- On définit $R : S \times C \to \mathbb{R}$ la fonction de récompense/punition qui à un état s et un coup $c \in A(s)$ associe la récompense dans le jeu du coup c dans l'état s

Définition 1 *Une* politique *$\pi : S \to C$ est une fonction qui à un état du jeu s, associe le prochain coup à jouer $c \in A(s)$.*

Notre but est d'obtenir une politique *optimale*, i.e. une politique qui maximise les récompenses sur le long terme. En fait, nous allons 'apprendre' cette politique en jouant un grand nombre de parties, dans lesquelles nous évaluerons et modifierons notre politique de manière à préférer les états gagnants aux perdants.

Nous allons construire une fonction $Q_\pi : S \times C \to \mathbb{R}$ qui à chaque couple $(s, c \in A(s))$ associera la valeur à long terme du coup c en s. Nous allons prendre pour valeur d'un couple (s, c) la somme des récompenses

que l'on obtiendra si on suit la politique π à partir de l'état où nous mènera le coup c dans l'état s. Ainsi pour une partie :

$$s \xrightarrow{c} s_1 \xrightarrow{\pi(s_1)} s_2 \xrightarrow{\pi(s_2)} s_3 \xrightarrow{\pi(s_3)} s_4 \xrightarrow{\pi(s_4)} s_5 \ldots$$

Ainsi, on aura :

$$Q_\pi(s,c) = R(s,c) + \sum_{k\geq 1} R(s_k, \pi(s_k))$$

Nous voulons les coups qui donnent une récompense plus forte à court terme, on introduit donc dans cette somme un facteur $\gamma < 1$:

$$Q_\pi(s,c) = R(s,c) + \sum_{k\geq 1} \gamma^k R(s_k, \pi(s_k))$$

Que l'on peut réécrire en :

$$Q_\pi(s,c) = R(s,c) + \gamma Q_\pi R(s_1, \pi(s_1))$$

Nous admettons le résultat suivant : nous avons une politique optimale π^*, si et seulement si la fonction associée Q^*_π vérifie :

$$\forall s \in S, \forall c \in A(s)\, Q^*_\pi(s,c) = R(s,c) + \gamma \max_{c' \in A(s')} Q^*_\pi(s',c') \text{ avec } s' \text{ le successeur de } s \text{ par le coup } c$$

Apprentissage Pour apprendre, on va consécutivement fixer Q_π pour mettre à jour π et fixer π pour calculer Q_π. Pendant l'élaboration de notre politique, il faut que l'on puisse explorer des coups. On va donc introduire une probabilité $\epsilon < 1$ de jouer un coup aléatoire.

$$\pi_\epsilon(Q_\pi, s) = \begin{cases} \text{un coup aléatoire parmi } A(s) \text{ avec une probabilité } \epsilon \\ argmax_{c\in A(s)} Q_\pi(s,c) \text{ avec une probabilité } 1-\epsilon \end{cases}$$

On va également introduire un facteur $\alpha < 1$ qui permettra, lors de la mise à jour de Q_π de prendre plus ou moins en compte les valeurs passées :

$$Q_\pi(s,c) = (1-\alpha)Q_\pi(s,c) + \alpha(R(s,c) + \gamma \max_{c' \in A(s')} Q_\pi(s',c'))$$

Algorithm 7 Apprentissage(π,Q,n) : retourne la fonction Q après n apprentissages

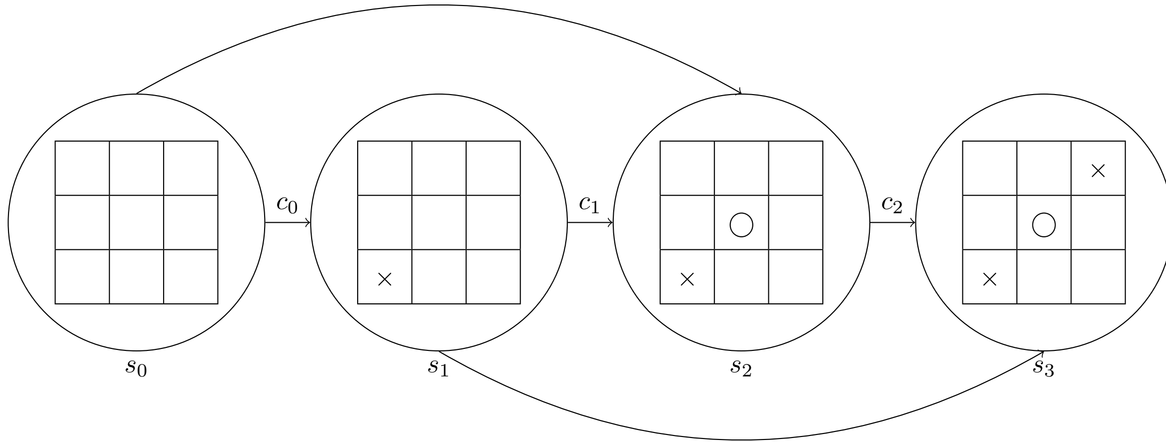
```
for i = 0 → n do
  s ← EtatInitial()
  while !EstFinal(s) do
    c ← π_ε(Q, s)
    r ← JOUER(c) {r est la récompense reçue après avoir jouer c}
    Q(s, c) ← (1 − α)Q(s, c) + α(r + γ max_{c'∈A(s')} Q*(s', c'))
  end while
  return Q
end for
```

Remarque : comme pour l'algorithme minimax, on considère que le joueur adverse a la même politique que nous.

4.2 Application aux jeux à deux joueurs et somme nulle

Pour appliquer l'algorithme *Apprentissage* à un jeu à deux joueurs, on considère qu'un coup correspond à deux mouvements consécutifs (un de chaque joueur). Au départ, nous partons avec une fonction Q nulle. On joue ensuite une partie contre nous même en mettant à jour les valeurs de Q. Si le premier joueur gagne la partie, les valeurs des coups qu'il a joués seront augmentées et celles du deuxième seront diminuées, etc.

La mise à jour de Q se fait de la manière suivante :

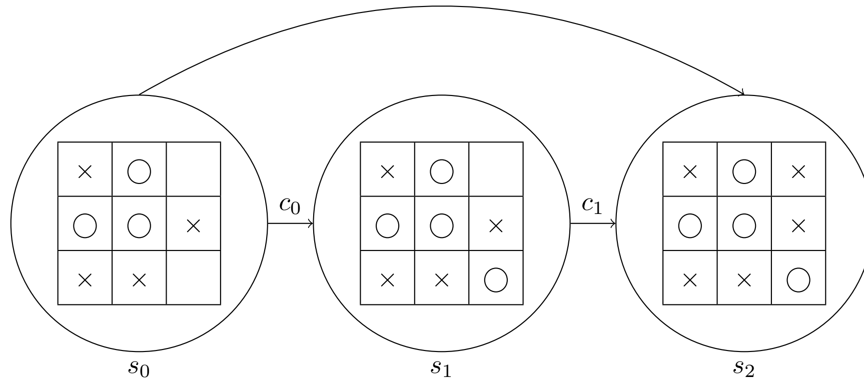


Mise à jour cas 1

Dans le cas 1, il n'y a pas d'état terminal rencontré :

$$Q(s_0, c_0) = (1 - \alpha)Q(s_0, c_0) + \alpha(0.0 + \gamma \max_{c \in A(s_2)} Q(s_2, c))$$

$$Q(s_1, c_1) = (1 - \alpha)Q(s_1, c_1) + \alpha(0.0 + \gamma \max_{c \in A(s_3)} Q(s_3, c))$$

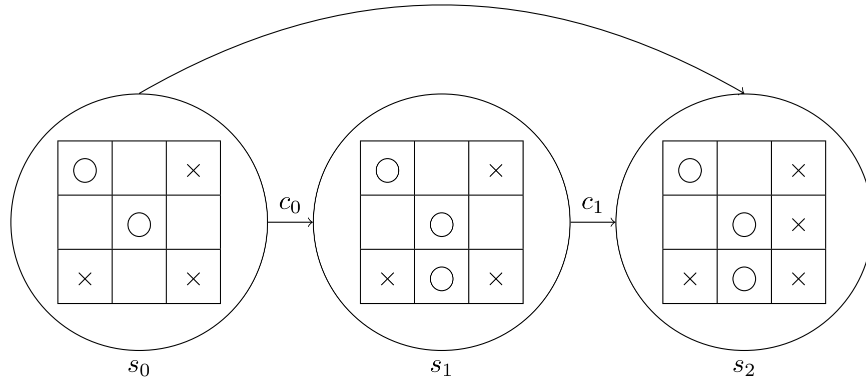


Mise à jour cas 2

Dans le cas 2, l'état terminal rencontré est un match nul :

$$Q(s_0, c_0) = (1 - \alpha)Q(s_0, c_0) + \alpha(0.0 + 0.0)$$

$$Q(s_1, c_1) = (1 - \alpha)Q(s_1, c_1) + \alpha(0.0 + 0.0)$$



Mise à jour cas 3

Dans le cas 3, l'état terminal rencontré est gagnant pour un joueur :

$$Q(s_0, c_0) = (1 - \alpha)Q(s_0, c_0) + \alpha(-1.0 + 0.0)$$

$$Q(s_1, c_1) = (1 - \alpha)Q(s_1, c_1) + \alpha(1.0 + 0.0)$$

Question 12 – Implémentez cet algorithme en Caml en suivant la signature du fichier 'MorpionLearning.mli'.

Question 13 – (Subsidiaire) Démontrez la convergence de π vers π^* grâce à cet algorithme.