Agent Oriented Learning

Introduction apprentissage par renforcement

Stéphane Airiau

Université Paris-Dauphine

Agent oriented Learning

- un agent autonome interagit avec son environment et essaye de s'adapter

⇨ méthode d'apprentissage basée sur l'interaction avec l'environnement, sans avoir nécessairement de connaissances préalables : ex : α-Go Zero

- l'agent peut-être seul et doit savoir apprendre à s'adapter
- il peut y avoir d'autres agents dans l'environnement. L'interaction entre les agents peut être de nature
  - coopérative ⇨ apprendre à travailler ensemble
  - compétitive (ou non coopératif) ⇨ il faut essayer de "gagner contre" les autres agents.
    - si on a un seul adversaire ⇨ battre cet adversaire
    - si on deux adversaires ou plus ⇨ plus compliqué : on peut former des coalitions avec certains agents, changer de coalitions, etc...

Souvent, tout cela est fait sans communication explicite ou sans negotiation.

Agent oriented Learning

- une bonne partie du cours sera une introduction (succincte) à l'apprentissage par renforcement (~ 3 séances)
  ⇨ apprendre à optimiser son comportement en évoluant dans un environnement.
  apprendre **seul**, c'est la base.
- l'apprentissage par renforcement dans un contexte multiagent
  - ⇨ aspects théorie des jeux
  - d'autres méthodes d'apprentissage multiagent

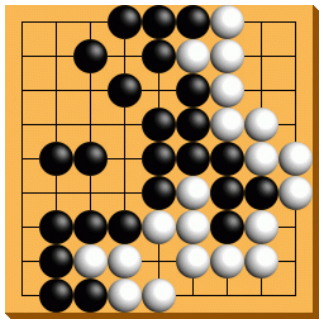
Pour ce premier cours, on s'intéresse d'abord à jouer à un jeu (donc contexte compétitif), puis on généralisera la méthode pour apprendre à "jouer" dans un environnement.

Types de jeux

On s'intéresse pour le moment aux jeux :

- à deux joueurs
- où les joueurs jouent un seul coup en alternance
- où chaque joueur peut observer le coup de son adversaire
- à somme constante (si un joueur gagne, l'autre perd)

exemples : les échecs, le jeu de go



General Game playing : le serveur donne une spécification du jeu (dans un langage "logique") et le programme doit jouer à ce jeu (2005–2020 - ?)

Modélisation

- s_0 état initial
- `joueur(s)` : définit quel joueur doit effectuer une action
- `actions(s)` : donne les coups légaux dans l'état `s`
- `resultat(s,a)` : fonction de transition qui définit le résultat de l'action `a` prise dans l'état `s`
- `terminal?(s)` retourne si le jeu est terminé
- `utilité(s)` : retourne une valeur numérique à chaque joueur lorsque le jeu est terminé

Par exemple pour les échecs :

- le gagnant reçoit un score de +1
- le perdant reçoit 0
- en cas d'égalité, chaque joueur reçoit $\frac{1}{2}$

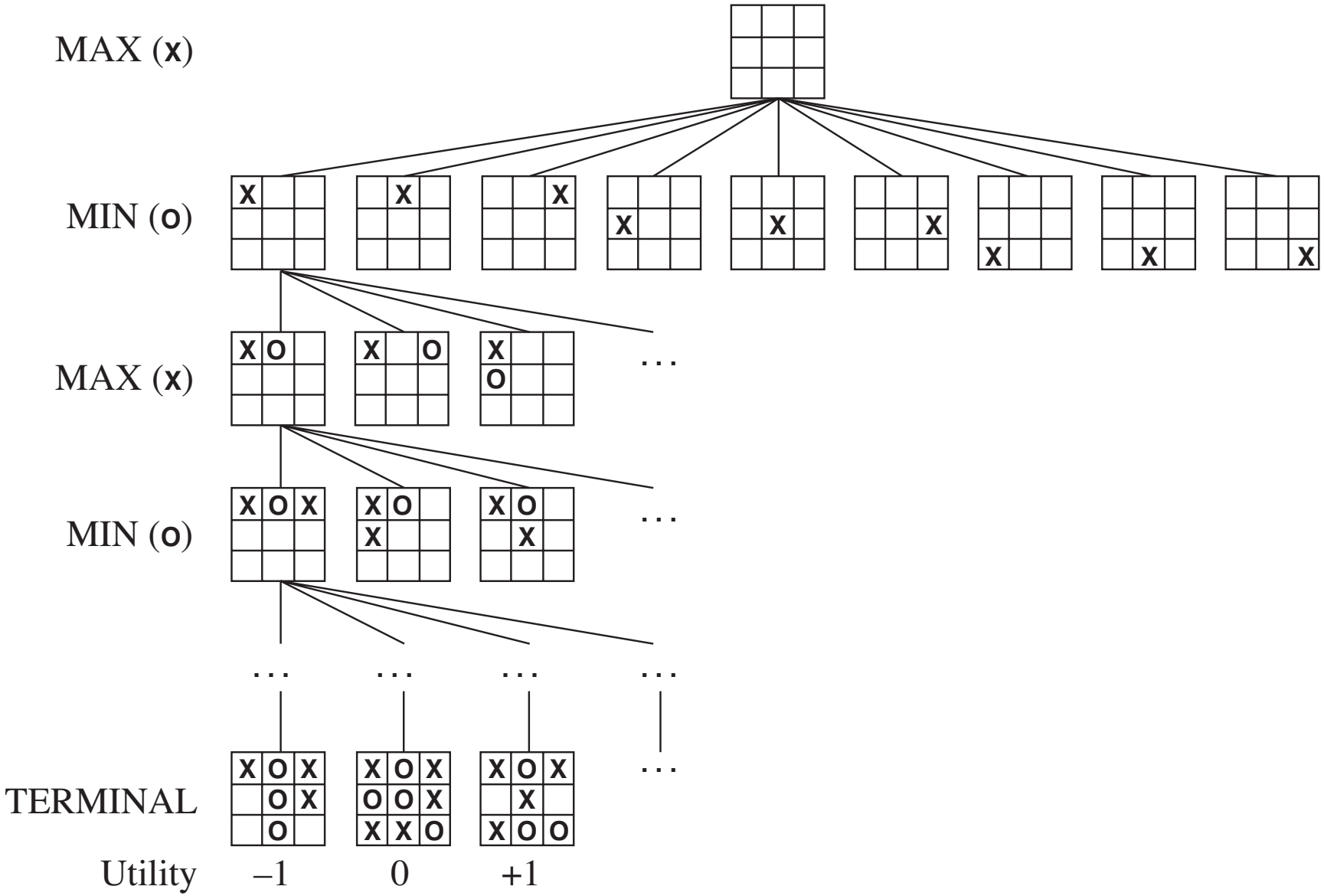
On parle de jeu à somme constante
(les joueurs ont des objectifs complètement contraires)

Représentation avec un arbre

On peut représenter une partie à l'aide d'un arbre :

- à chaque niveau de l'arbre, un joueur choisit un coup
- chaque joueur cherche à gagner
  ⇨ le jeu est à somme constante
  ⇨ un joueur cherche a maximiser son score
  ⇨ son adversaire cherche à le faire perdre ⇨ cherche à minimiser le score
- on se place du point de vue d'un des joueur
  - on appelle ce joueur `MAX` (il cherche à maximiser son score)
  - l'autre joueur est appelé `MIN` (il cherche à minimiser le score de `MAX`)

Exemple du morpion



le morpion

- combien y-t-il de configurations différentes ?

le morpion

- combien y-t-il de configurations différentes ?
- moins que 9 != 362,880 feuilles dans l'arbre

le morpion

- combien y-t-il de configurations différentes ?
- moins que 9 != 362,880 feuilles dans l'arbre
- certaines configurations vont se retrouver plusieurs fois dans l'arbre
  ⇨ table de transposition pour se rappeler qu'on a déjà vu cette configuration

le morpion

- combien y-t-il de configurations différentes ?
- moins que 9 != 362,880 feuilles dans l'arbre
- certaines configurations vont se retrouver plusieurs fois dans l'arbre
  ⇨ table de transposition pour se rappeler qu'on a déjà vu cette configuration
- généralement, les jeux sont trop gros pour raisonner sur l'arbre en entier
  ⇨ on peut le voir comme une abstraction théorique que l'on **n**'utilisera **pas** en pratique.

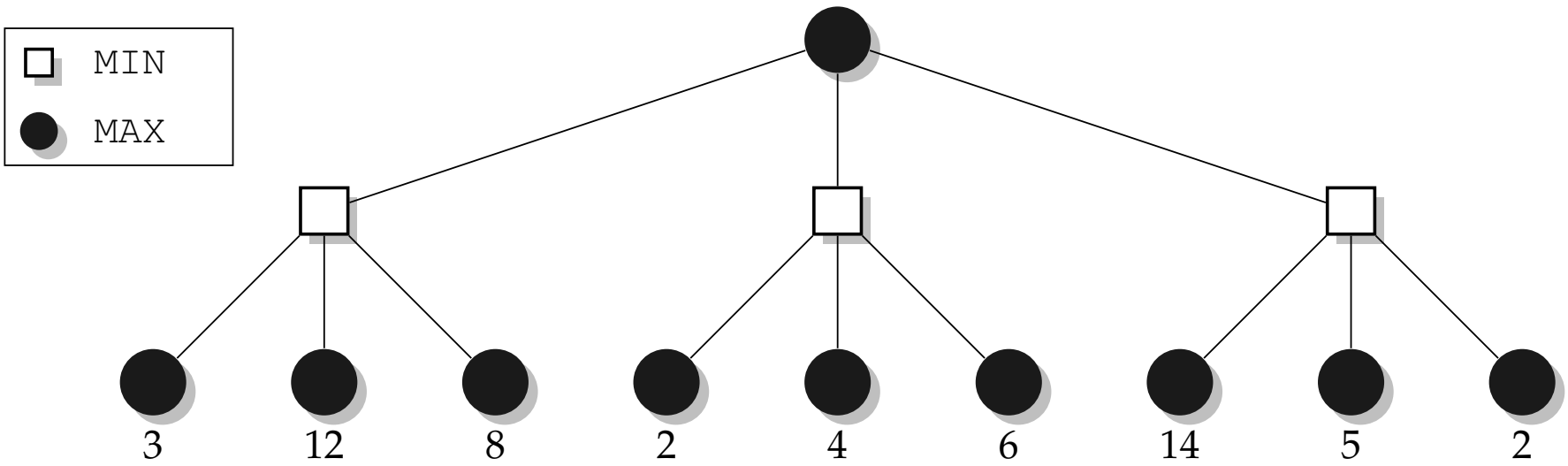
Résolution Optimale

- séquence de coups qui mènent à un état gagnant !
- maix `MIN` joue aussi !
  ⇨ une stratégie doit prendre en compte `MIN` et prévoir un coup pour chaque action possible de `MIN`
- on peut supposer que `MIN` joue de façon optimale aussi !

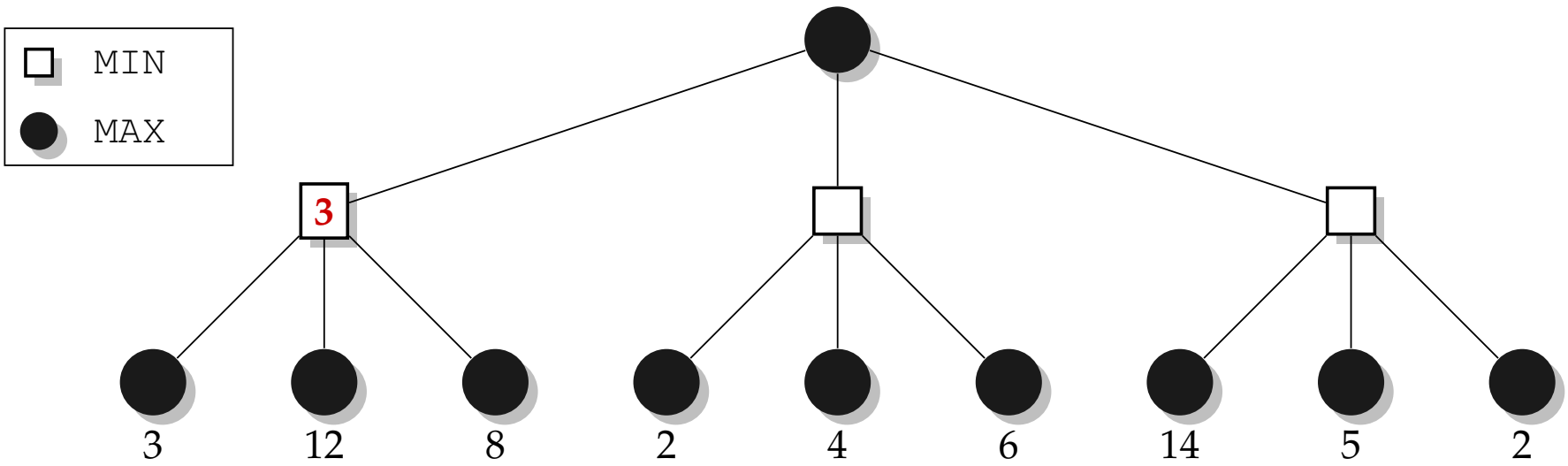
`MINIMAX(s)` =

- `utilité(s)` si `s` est un état final
- $\max_{a \in actions(s)}$ `MINIMAX(resultat(s,a))` si `MAX` joue
- $\min_{a \in actions(s)}$ `MINIMAX(resultat(s,a))` si `MIN` joue

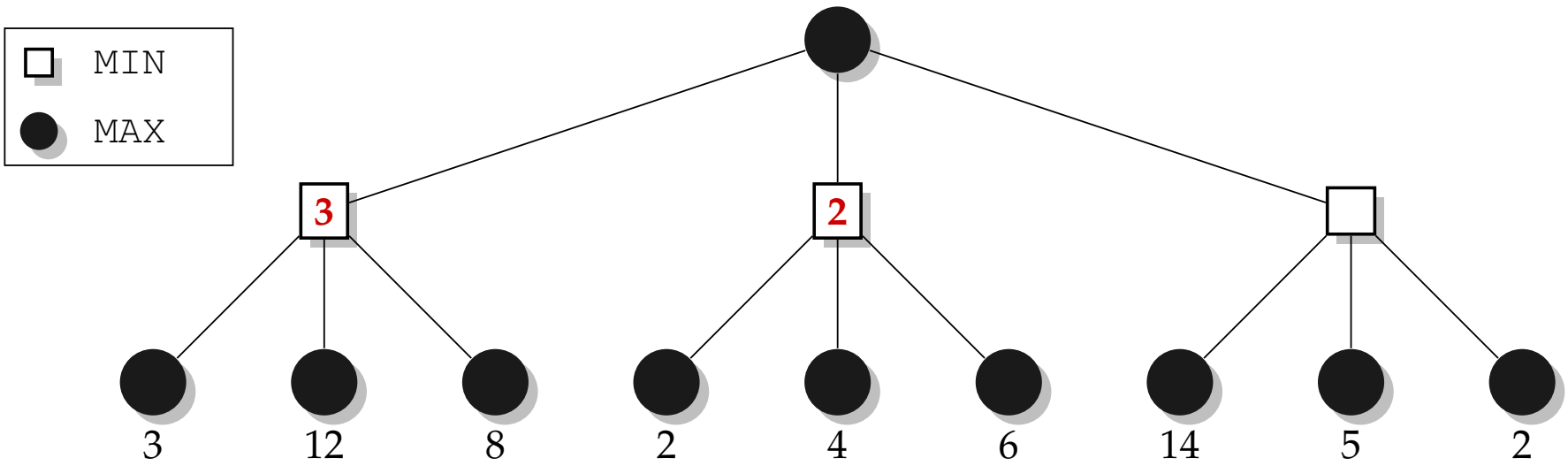
Exemple



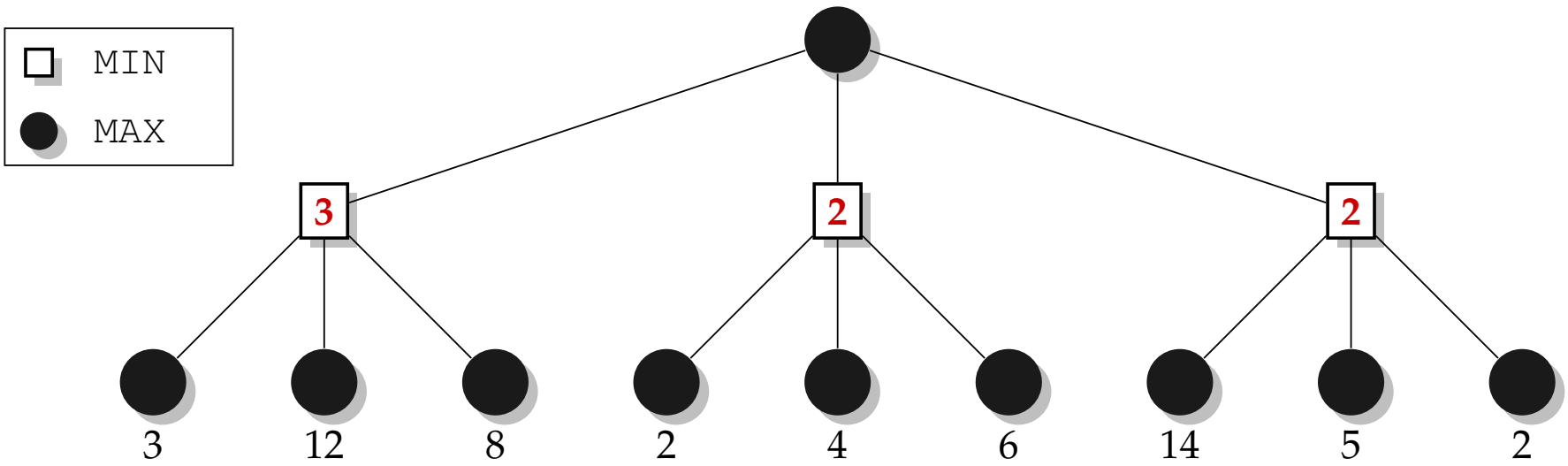
Exemple



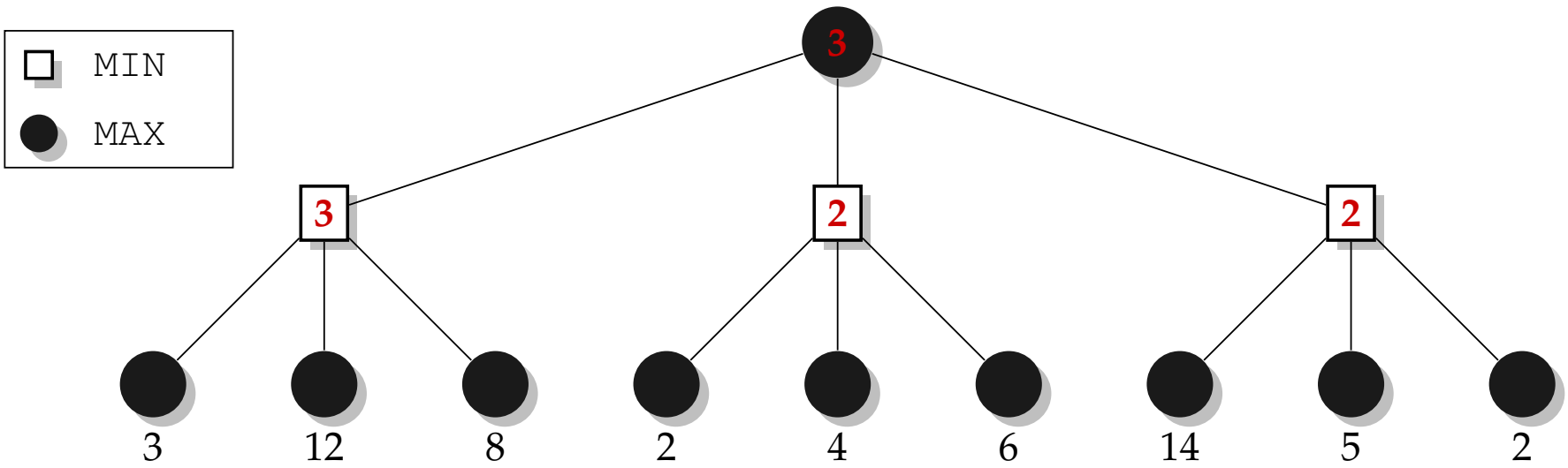
Exemple



Exemple



Exemple



Algorithmes

```
1 function MINIMAX_DECISION(s) returns an action
2 arg max_{a∈actions(s)} MIN_VALUE(result(s,a))
```

```
1 function MIN_VALUE(s) returns a utility value
2     if terminal?(s) then return utility(s)
3     v ← ∞
4     for each a ∈ actions(s) do
5         v ← min{v, MAX_VALUE(result(a,s))}
6     return v
```

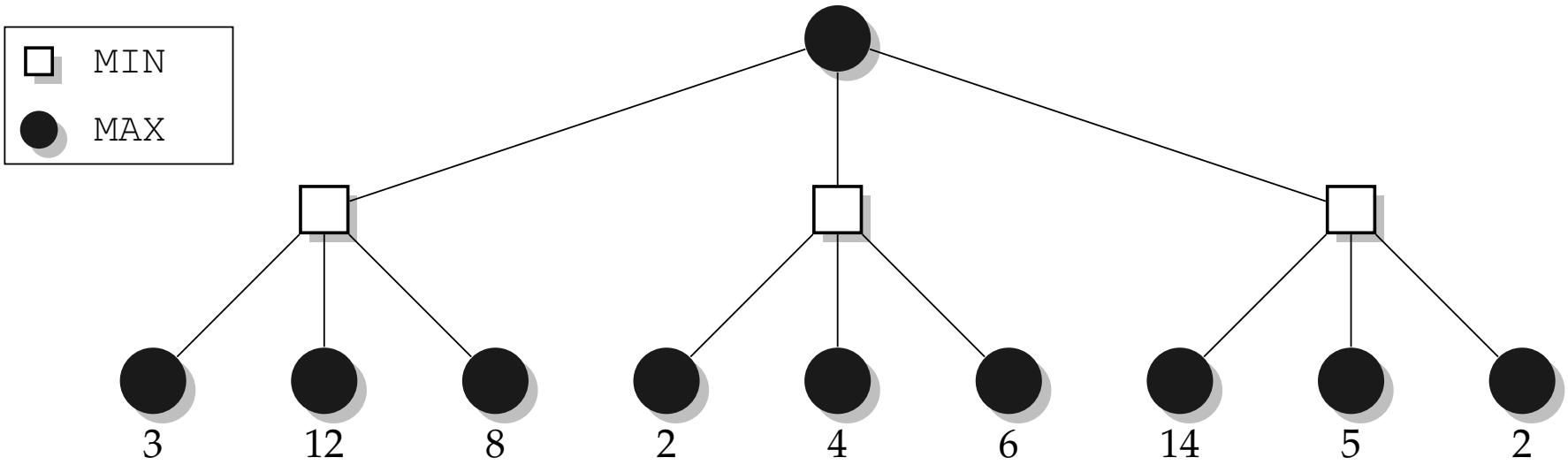
```
1 function MAX_VALUE(s) returns a utility value
2     if terminal?(s) then return utility(s)
3     v ← −∞
4     for each a ∈ actions(s) do
5         v ← max{v, MIN_VALUE(result(a,s))}
6     return v
```

Jeux à plus de deux joueurs

- l'algorithme peut marcher
- cependant, les autres joueurs n'ont pas toujours le même but (de minimiser le score du même joueur)
- il faut analyser plus finement le jeu
  ⇨ théorie des jeux (économie, Academy Awards)

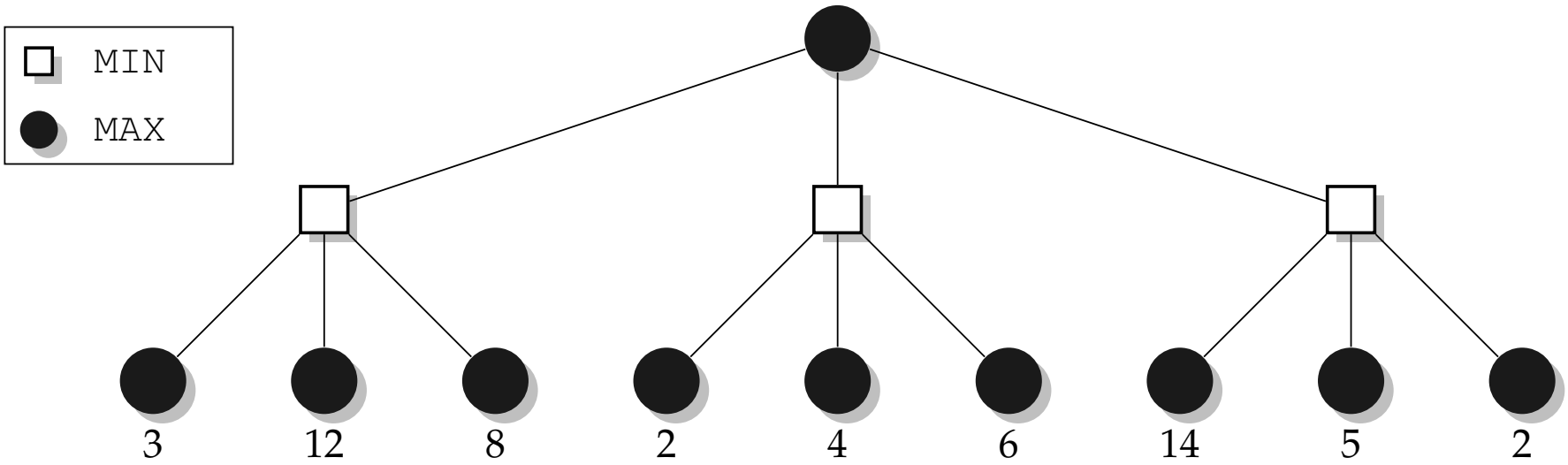
Elagage

Peut-on éviter de visiter des sous arbres entiers
sans risque de manquer une solution optimale ?



Elagage

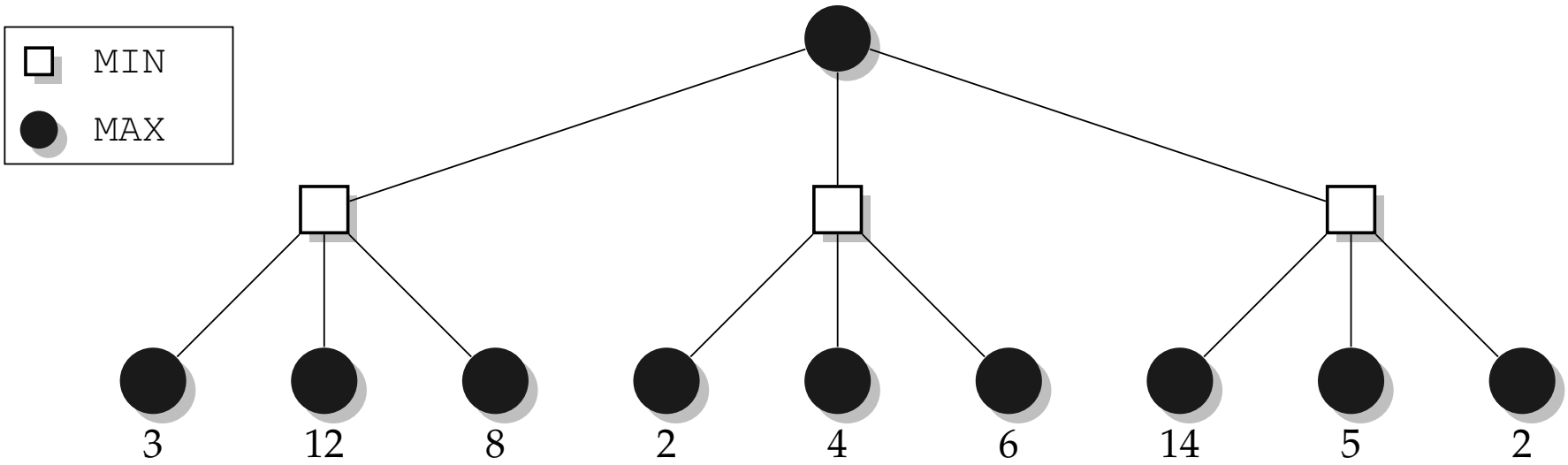
Peut-on éviter de visiter des sous arbres entiers
sans risque de manquer une solution optimale ?



$$\texttt{MINIMAX(racine)} = \max\{\min\{3,12,8\},\min\{2,x,y\},\min\{14,5,2\}\}$$
$$= \max\{3,z,2\} \text{ où } z = \min\{2,x,y\} \leq 2,$$
$$= 3$$

Elagage

Peut-on éviter de visiter des sous arbres entiers
sans risque de manquer une solution optimale ?

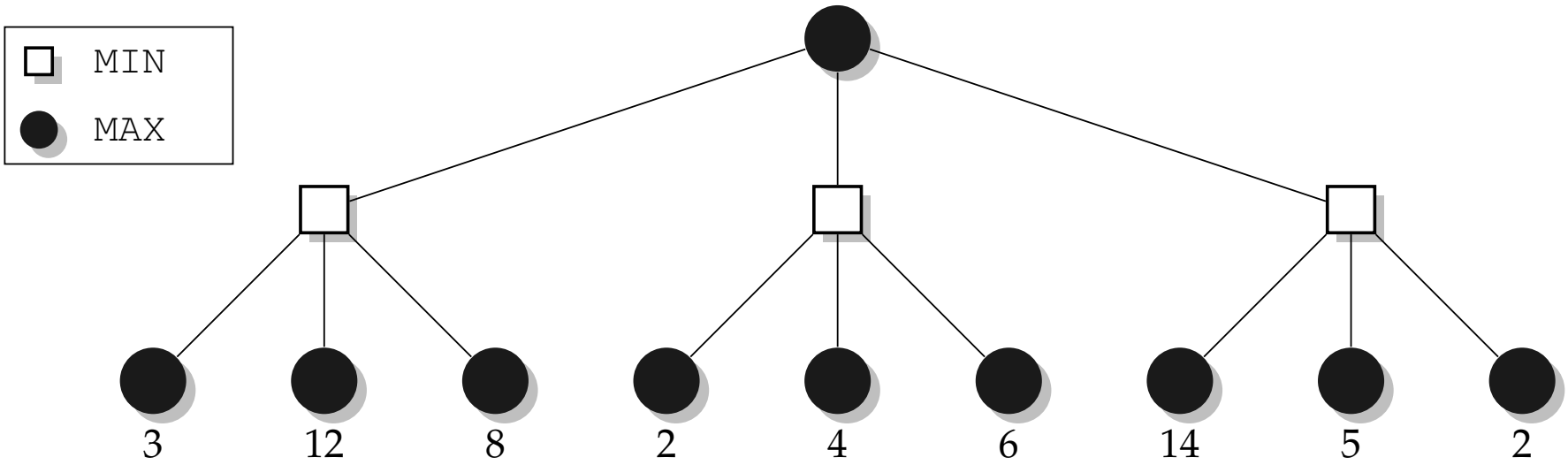


$$\texttt{MINIMAX(racine)} = \max\{\min\{3,12,8\},\min\{2,x,y\},\min\{14,5,2\}\}$$
$$= \max\{3,z,2\} \text{ où } z = \min\{2,x,y\} \leq 2,$$
$$= 3$$

⇨ Quelles que soient les valeurs de x et y, on connait la valeur de `MINIMAX(racine)` !

Elagage

Peut-on éviter de visiter des sous arbres entiers
sans risque de manquer une solution optimale ?

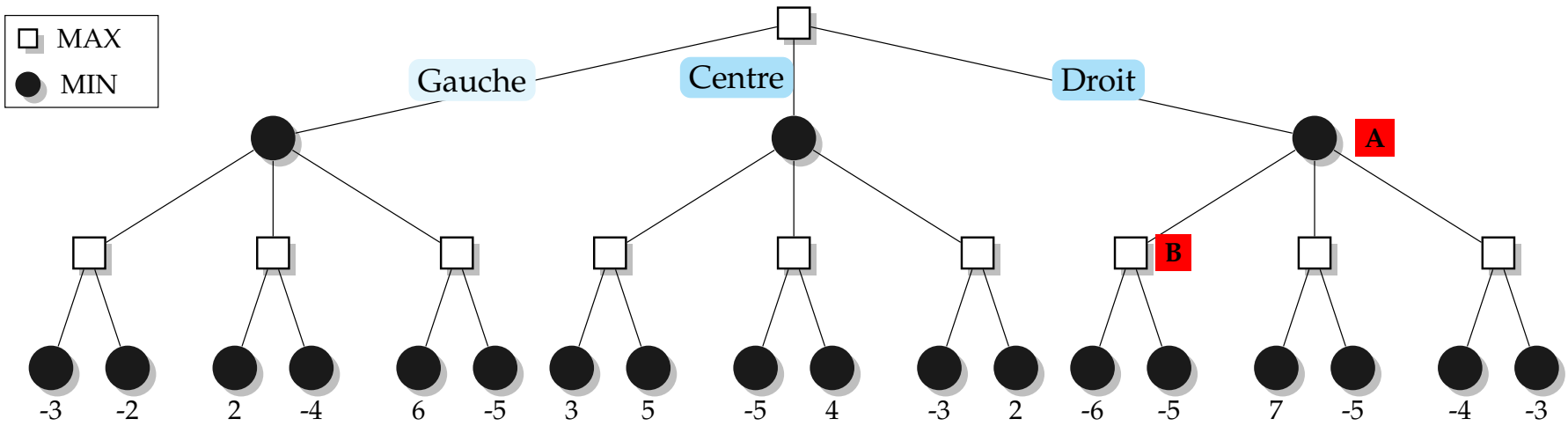


$$\texttt{MINIMAX(racine)} = \max\{\min\{3,12,8\},\min\{2,x,y\},\min\{14,5,2\}\}$$
$$= \max\{3,z,2\} \text{ où } z = \min\{2,x,y\} \leq 2,$$
$$= 3$$

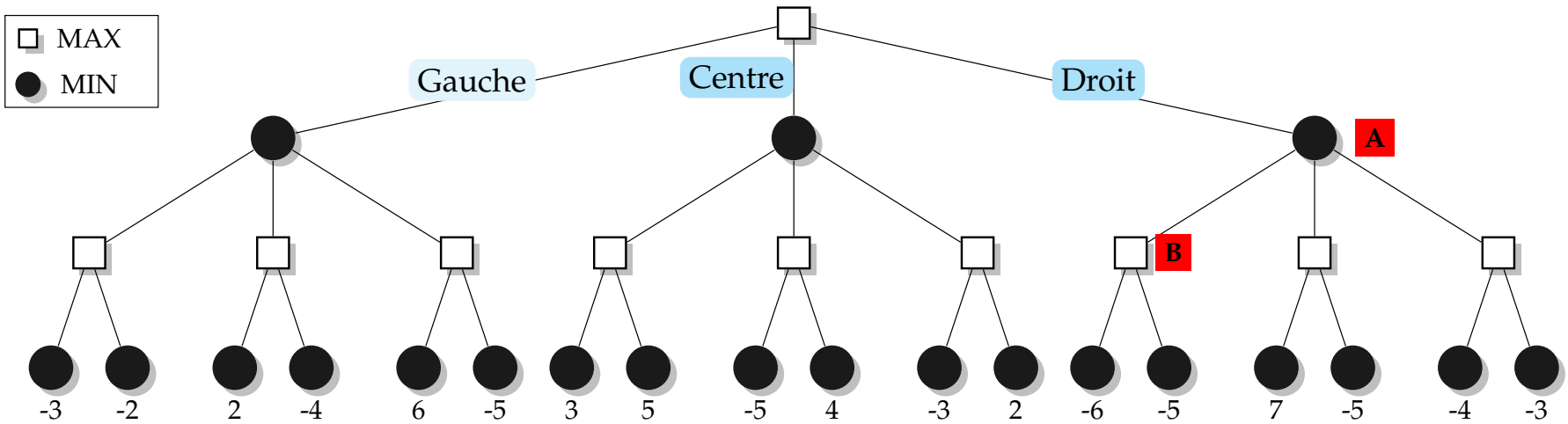
⇨ Quelles que soient les valeurs de x et y, on connait la valeur de `MINIMAX(racine)` !

⇨ élagage pour `MAX`

Elagage

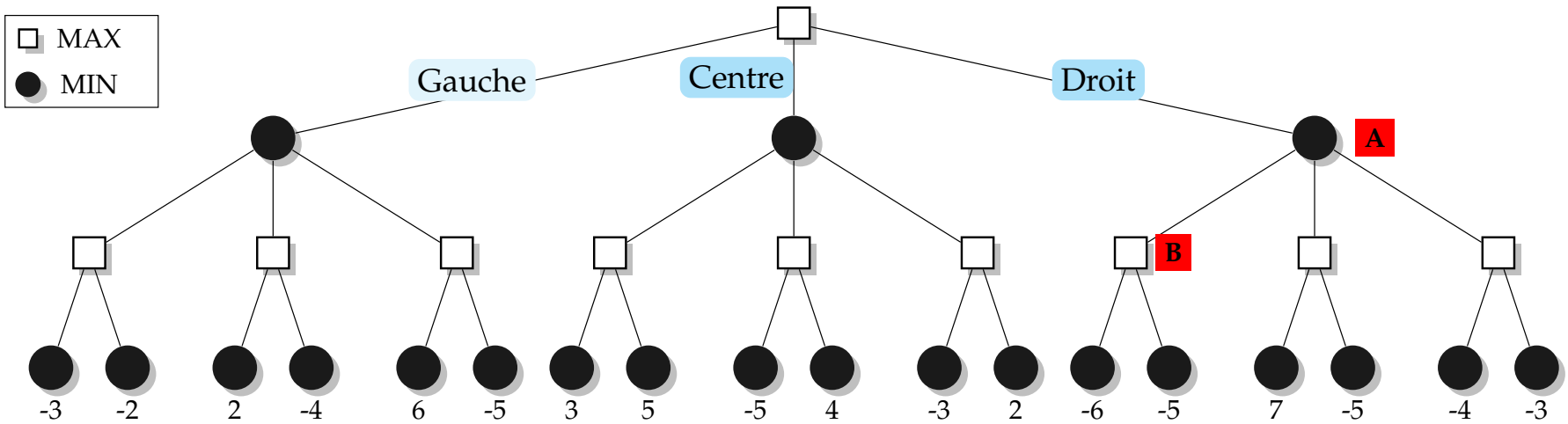


Elagage



- au moment d'examiner le noeud `A`, le joueur `MAX` a la garantie d'avoir au moins 2

Elagage



- au moment d'examiner le noeud `A`, le joueur `MAX` a la garantie d'avoir au moins 2
- après examen du sous-arbre sous `B`,
  on sait que si `MAX` joue l'action `Droit` au premier coup alors `MIN` aura au moins -5
  ⇨ on n'a pas besoin d'étudier les autres fils du noeud `A`
  ⇨ élagage pour `MIN`.

Elagage α-β

L'idée est donc de maintenir deux valeurs :

- α valeur de la meilleure (plus haute) valeur jusqu'ici pour `MAX`
- β valeur de la meilleure (plus basse) valeur jusqu'ici pour `MIN`

Le but est donc de mettre à jour α et β et d'élaguer la recherche quand la valeur du noeud considéré est pire que α pour `MAX` ou β pour `MIN`.

Algorithmes

```
1  function ALPHA_BETA(s) returns an action
2  v ← MAX_VALUE(s, −∞, +∞)
2  return a ∈ actions(s) with value v
```

```
1  function MIN_VALUE(s, α, β) returns a utility value
2      if terminal?(s) then return utility(s)
3      v ← +∞
4      for each a ∈ actions(s) do
5          v ← min{v, MAX_VALUE(result(a, s), α, β)}
6          if v ≤ α then return v
7          β ← min{β, v}
8      return v
```

```
1  function MAX_VALUE(s, α, β) returns a utility value
2      if terminal?(s) then return utility(s)
3      v ← −∞
4      for each a ∈ actions(s) do
5          v ← max{v, MIN_VALUE(result(a, s), α, β)}
6          if v ≥ β then return v
7          α ← max{α, v}
8      return v
```

Propriétés

- décision optimale si l'adversaire joue de façon optimale
  si l'adversaire ne joue pas parfaitement, on va faire mieux
  mais la solution ne sera peut être pas optimale dans ce cas
- complexité (temps) $O(b^m)$ où b est le facteur de branchement et m la profondeur maximale
- complexité (mémoire) $O(b \cdot m)$ (recherche en profondeur d'abord)

pour les échecs $b \approx 35$ et $m \approx 100$ pour des parties "raisonnables"
⇨ solution exacte impossible !

si on pouvait choisir optimalement l'ordre des actions à jouer pour élaguer au maximum, on a en général une complexité autour de $O(b^{\frac{m}{2}})$
cela reste insuffisant pour les échecs !

Utiliser des fonctions d'évaluation si on ne peut pas finir la partie

- utiliser la connaissance du jeu pour bâtir une heuristique mesurant la qualité de l'état
  ex : donner des points pour les différentes pièces : 1 point pour pion, 3 pour une tour, etc..
  utiliser une somme pondérée comme fonction d'évaluation
- utiliser des critères plus complexes (les poids peuvent être en fonction de la présence de certaines combinaisons de pièces)
- utiliser des patterns de positions
- utiliser des simulations Monte Carlo pour évaluer les états

Variante

- **`CutOff-Test`**`(s, profondeur)` va retourner vrai lorsqu'on atteint une certaine profondeur ou si l'état `s` est un état final

⇨ il faut déterminer la profondeur seuil pour bien utiliser le temps.

- Autre solution : tant qu'il reste du temps, exécuter `Iterative-Deepening-Search`
- il ne faudrait pas s'arrêter à des états où l'évaluation a des risques de changer drastiquement
  ex : si on compte les pièces au échecs, évaluer juste avant une capture peut être trompeur.
- l'effet d'horizon peut nous tromper (on peut imaginer une situation dans laquelle on est sûr de perdre, mais on peut repousser l'inévitable en faisant des actions qui ne vont rien changer)

Simulation de Monte Carlo

- Monte Carlo : nom générique utilisé lorsqu'on effectue des simulations pour obtenir des statistiques
- ici, on veut estimer la valeur des états

⇨ ensuite, il suffira de choisir le coup qui mène au meilleur état.

- *idée :* au lieu de visiter tout un sous arbre pour connaître la valeur d'un état
  on va simplement jouer "quelques parties" complètes
  ⇨ on fabrique des statistiques sur ce noeud

UCT

selection On utilise une heuristique (bas pour choisir les coupsee sur l'intervall de confiance)

expansion Lorsqu'on arrive à un noeud avec des satistiques incomplètes

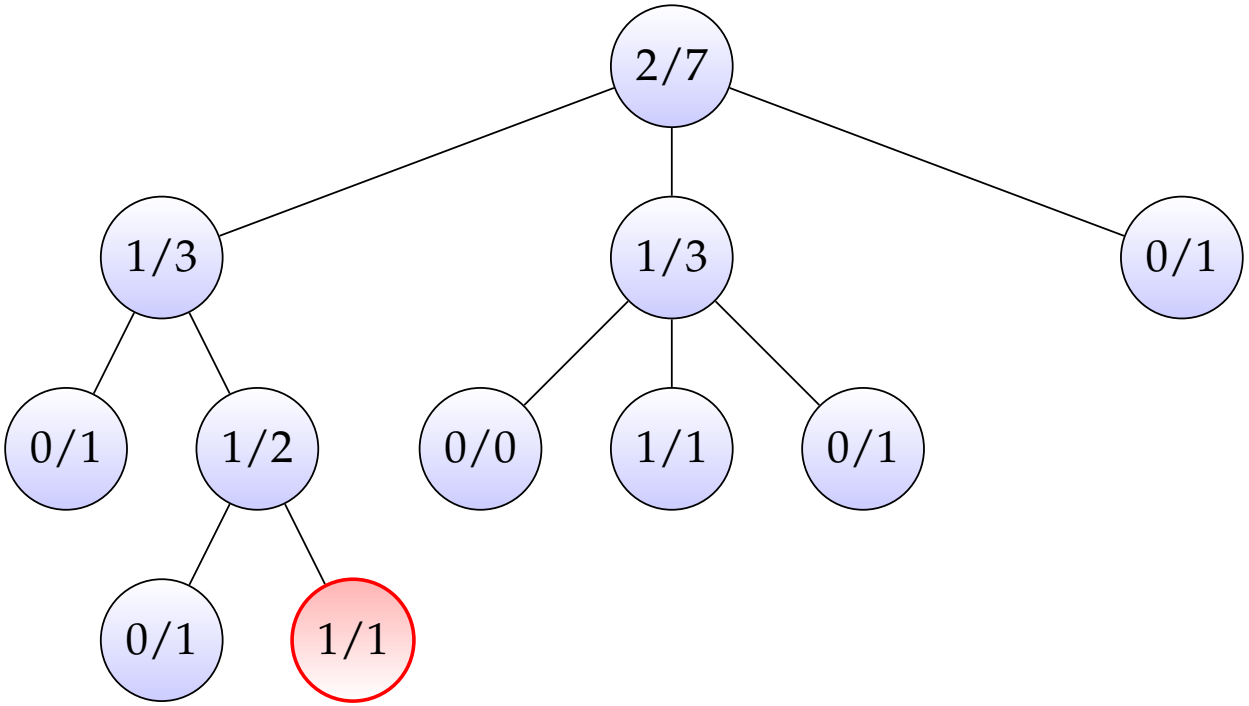
- on choisit un noeud suivant au hasard

simulation On effectue une "partie complète" au hasard
on descend au hasard jusqu'à un état final

propagation on propage les résultats ⇨on met à jour les statistiques pour les états parents

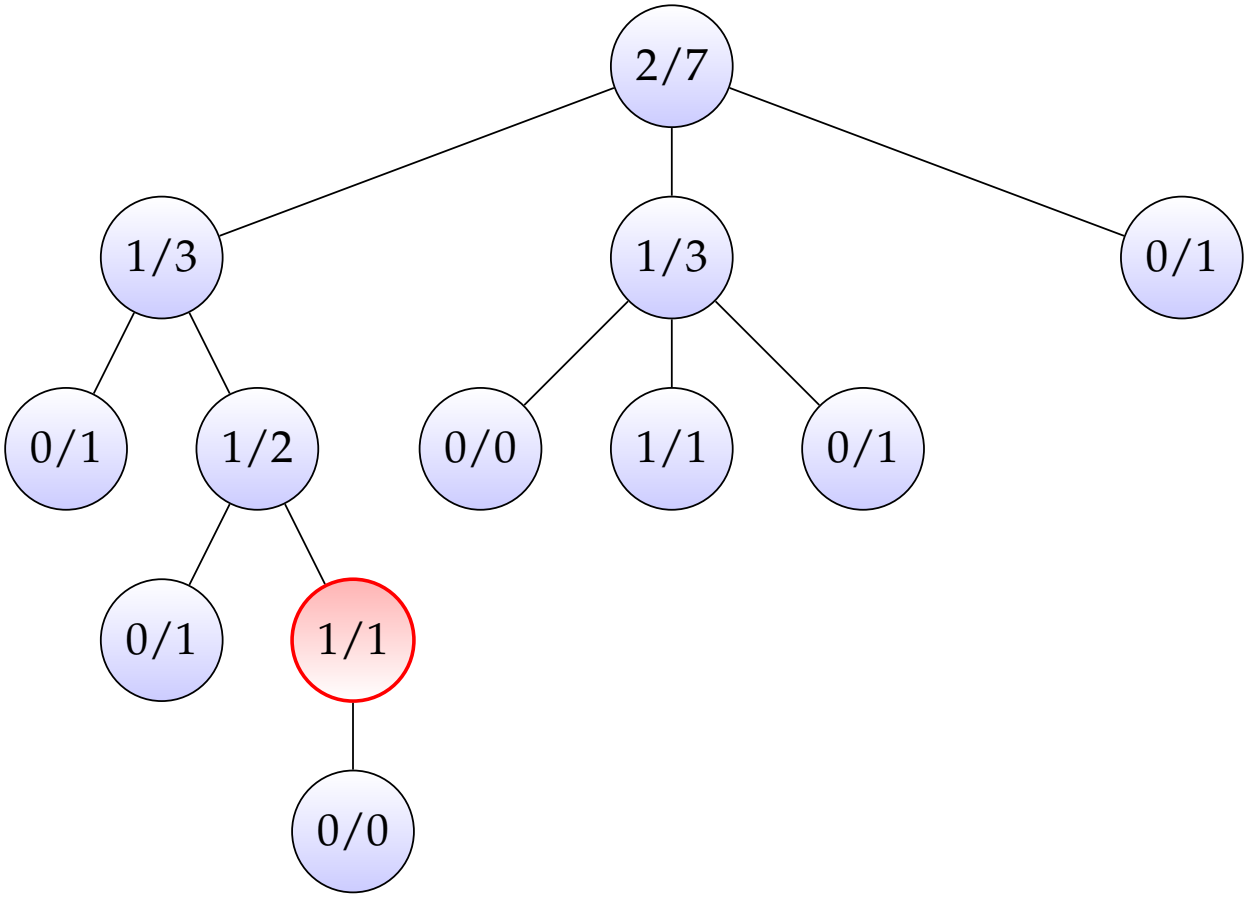
- On construit peu à peu l'arbre de recherche.
- On joue de façon guidée en conservant un bon équilibre exploration / exploitation

UCT : exemple



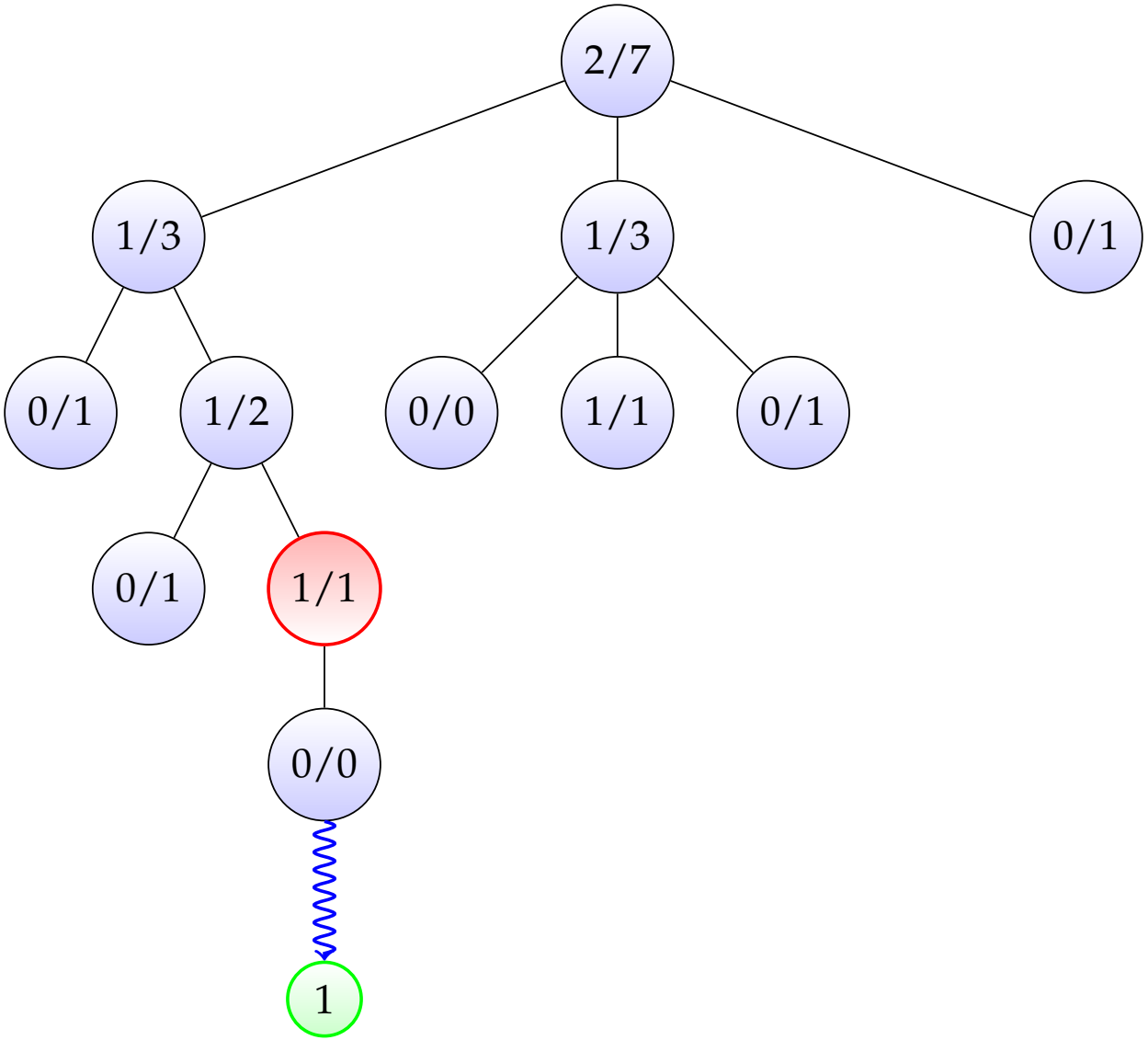
Selection : On choisit les actions avec UCB, dans l'exemple, cela nous mène au noeud rouge.
Le noeud en rouge n'a pas de fils avec des statistiques

UCT : exemple



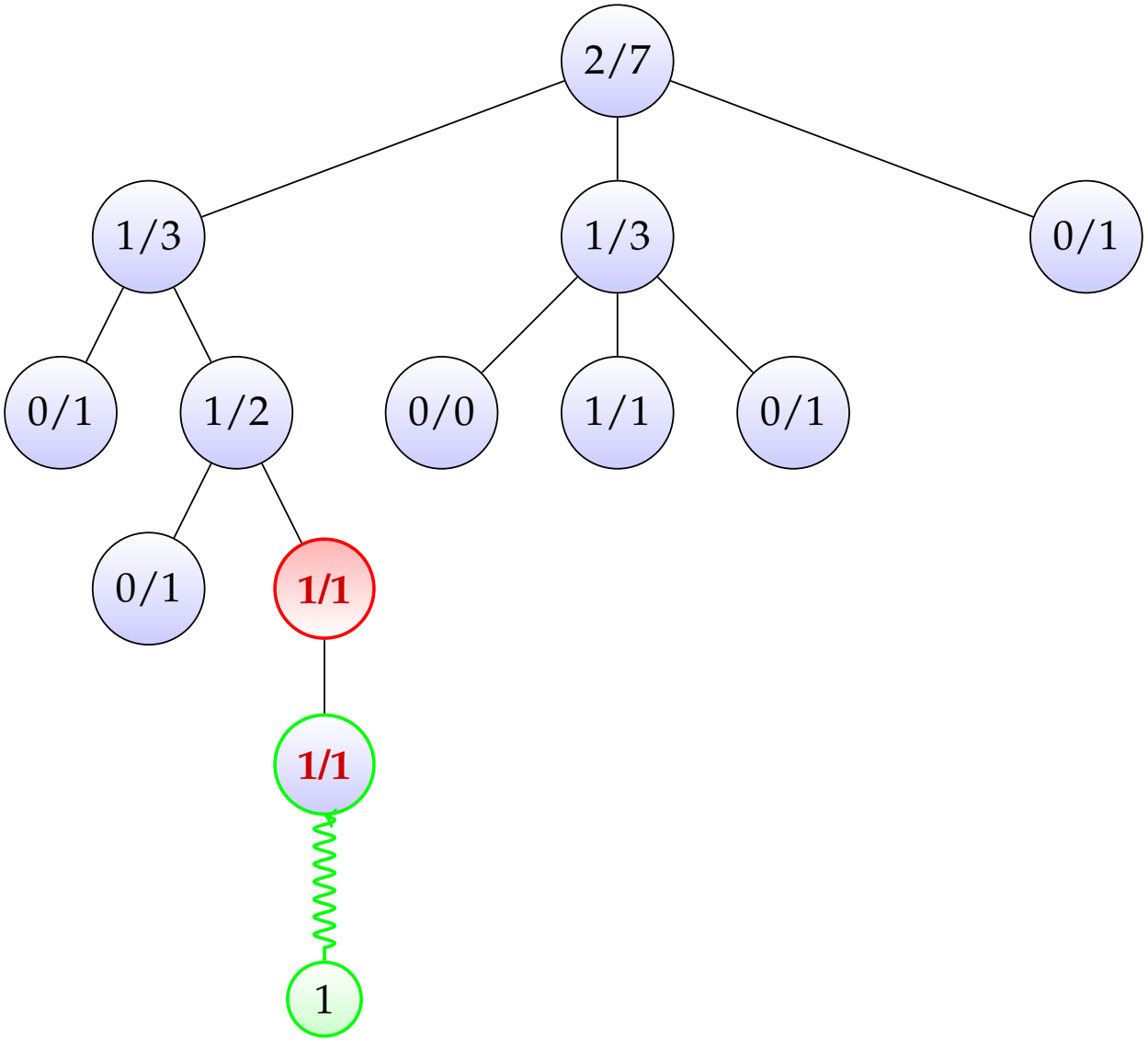
Expansion : on choisit un fils au hasard

UCT : exemple



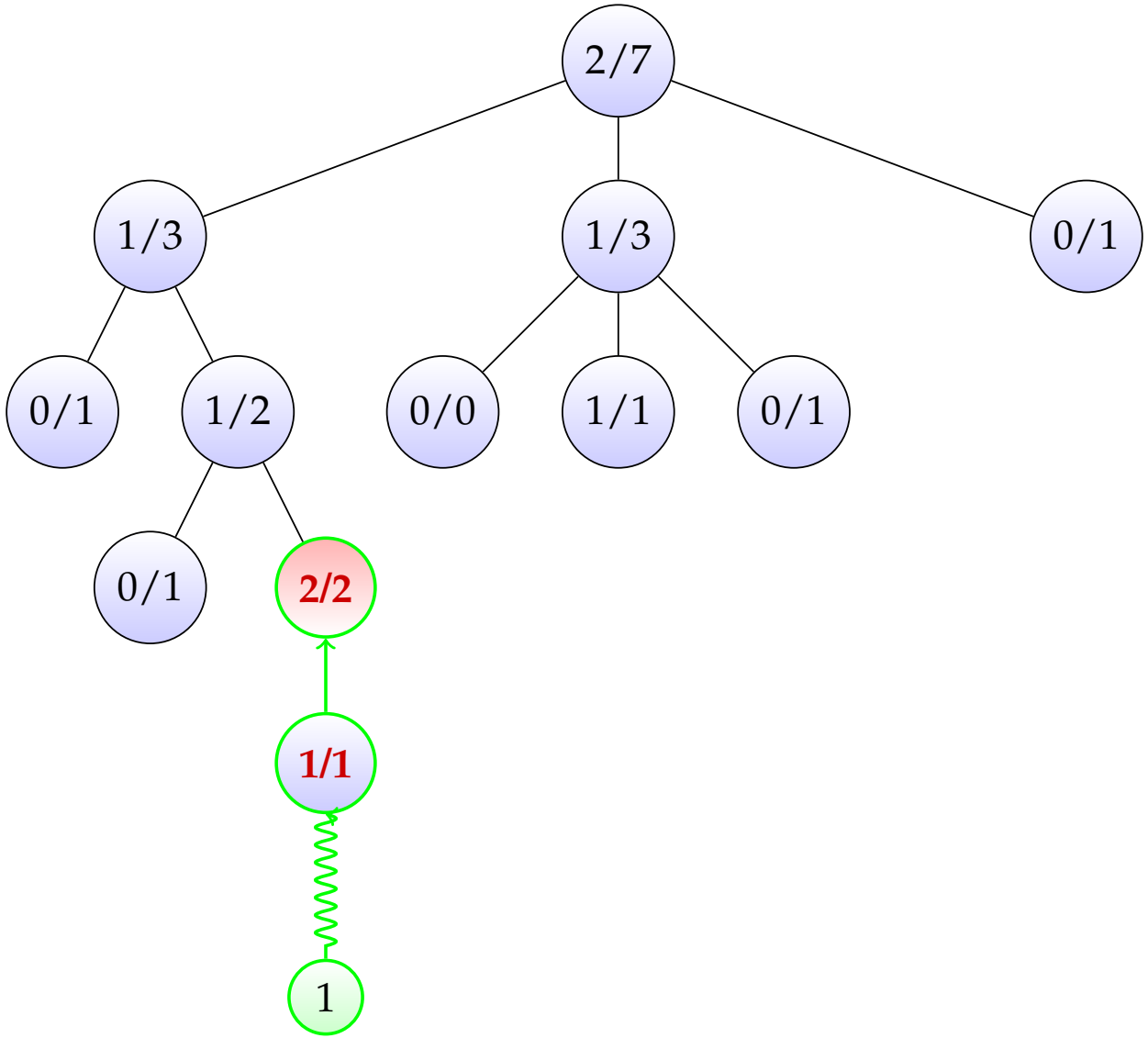
Simulation : on complète une partie au hasard, dans l'exemple on gagne

UCT : exemple



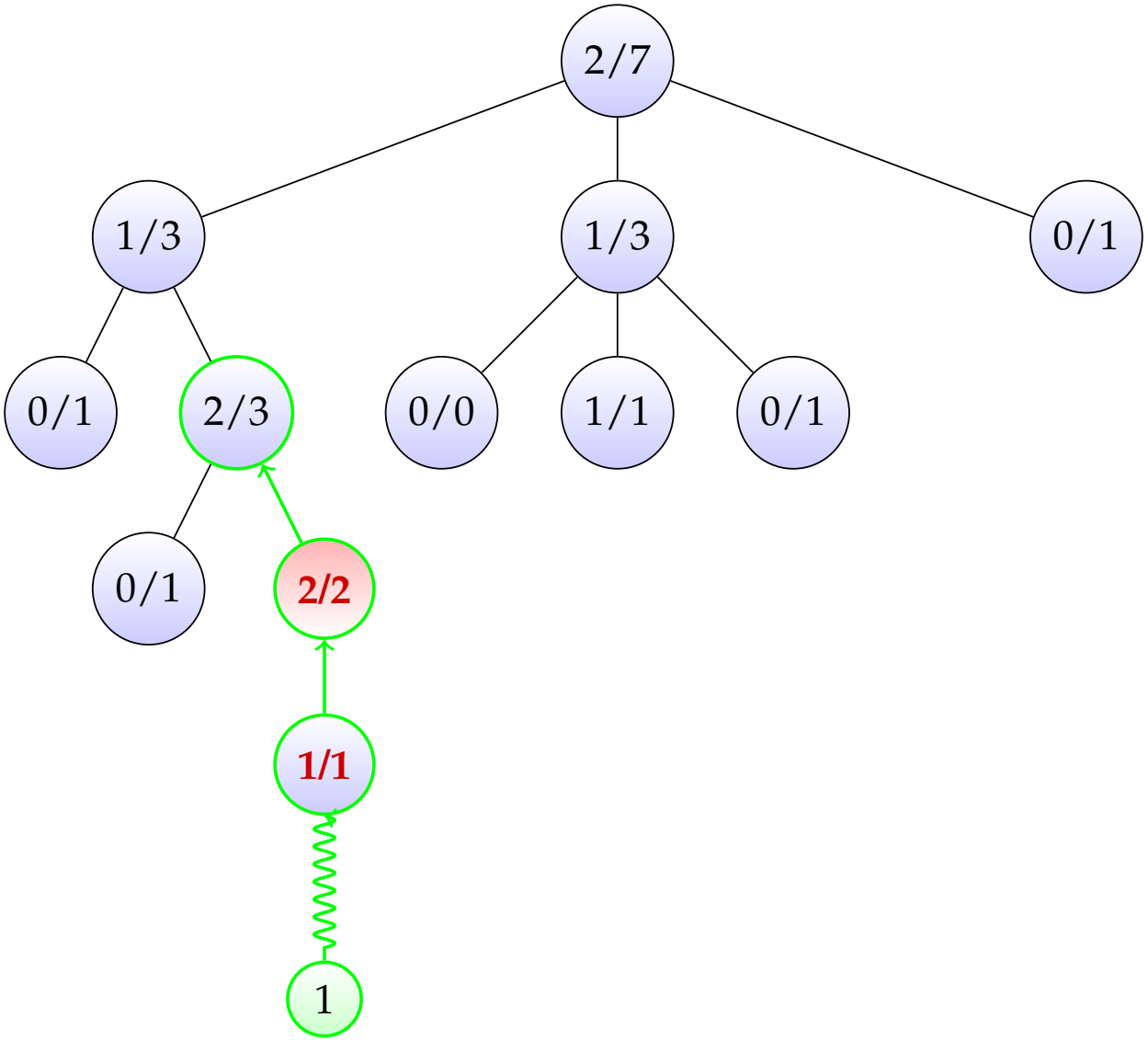
Retropropagation : on met à jour les statistiques

UCT : exemple



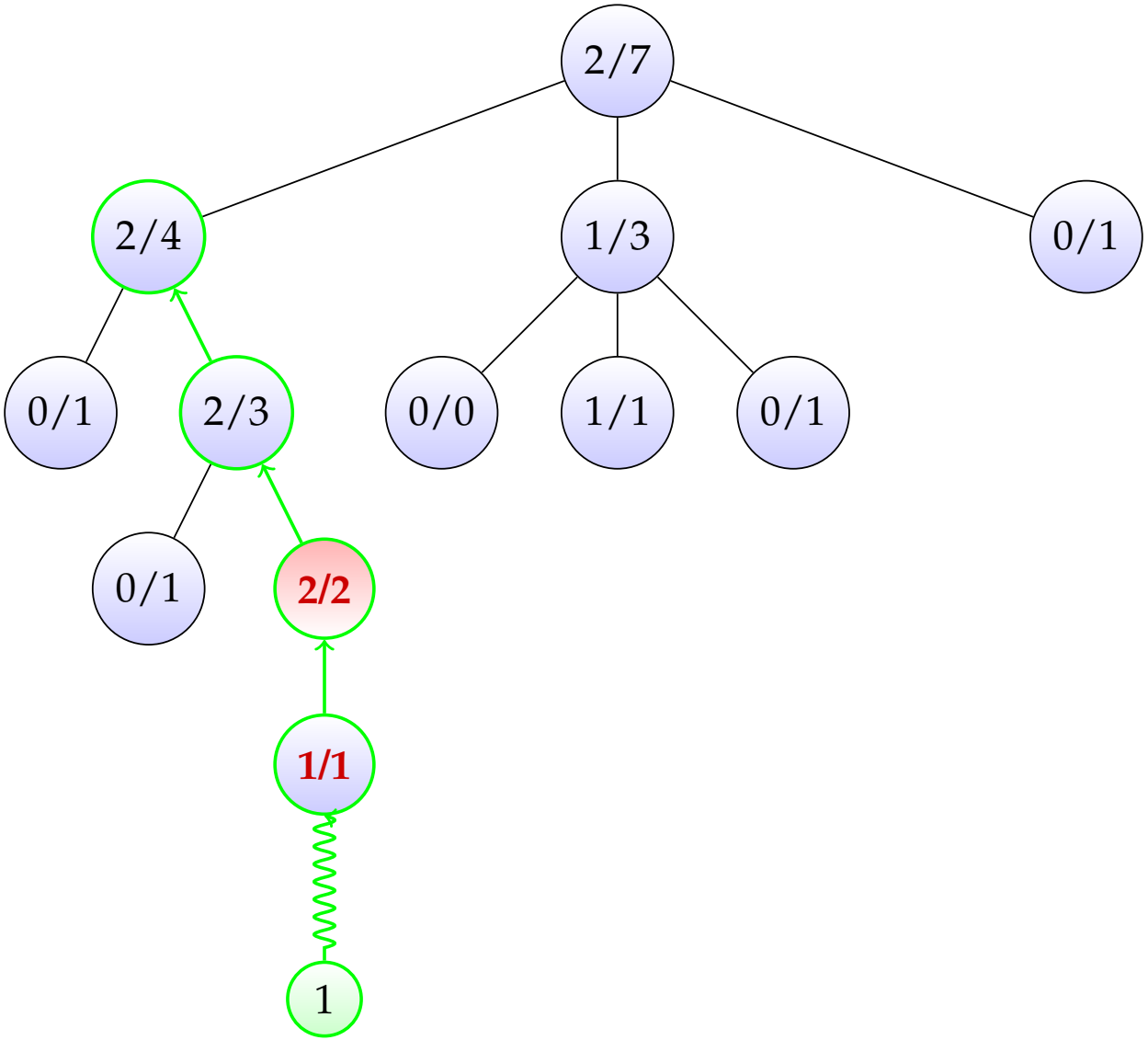
Retropropagation : on met à jour les statistiques

UCT : exemple



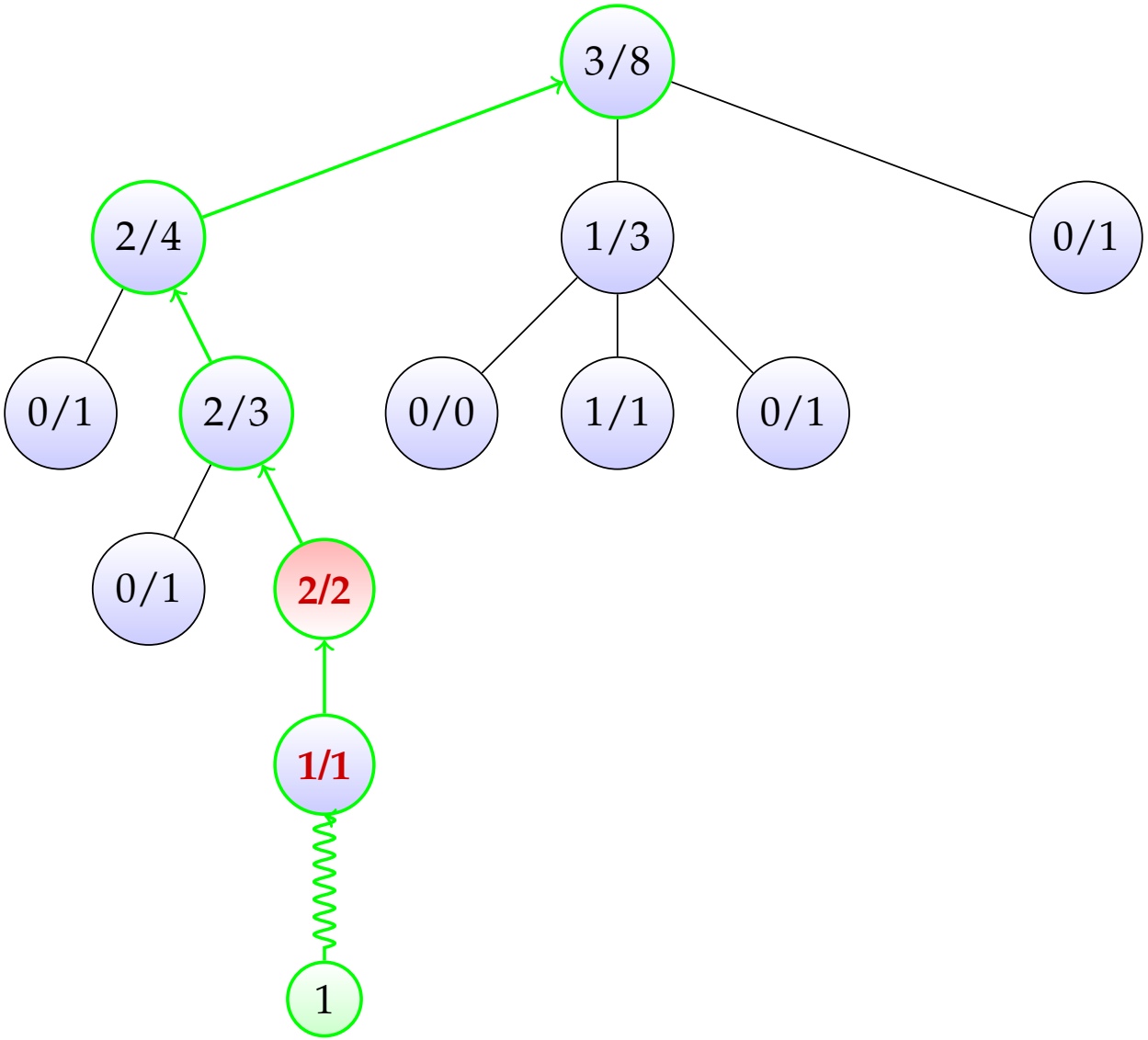
Retropropagation : on met à jour les statistiques

UCT : exemple



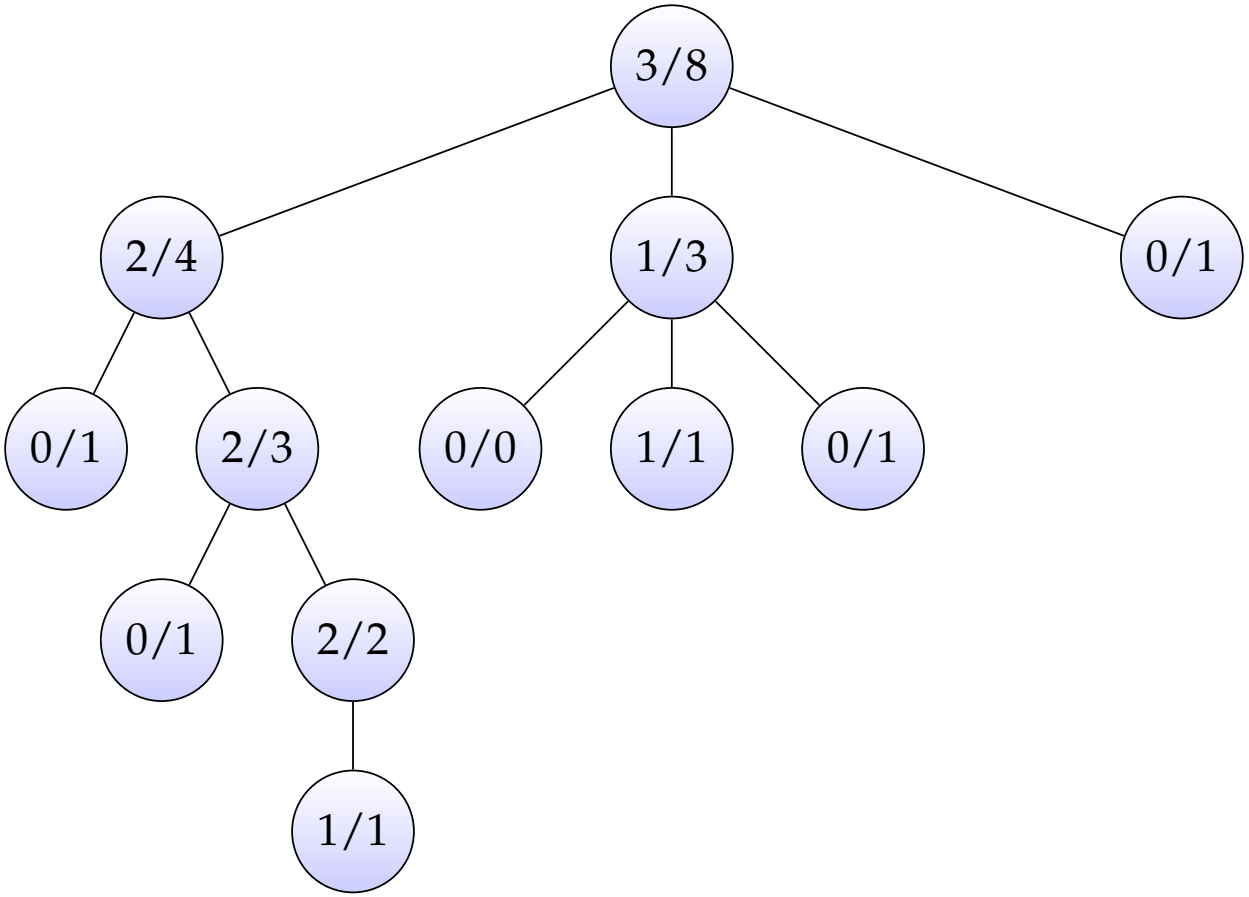
Retropropagation : on met à jour les statistiques

UCT : exemple



Retropropagation : on met à jour les statistiques

UCT : exemple



Fini, on peut recommencer ![a]

a. Le point de départ de l'exemple n'est peut être pas un état atteignable avec UCT

Monte Carlo Tree Search

- UCT (Upper Confidence Tree) est une variante d'une famille plus large appelée Monte Carlo Tree Search.
- il existe d'autres variantes pour gérer l'exploration et l'exploitation

⇨ ces méthodes sont à l'origine d'un saut de performance pour le jeu de go dans les années 2006–2008
avant, des variantes de α-β étaient utilisées

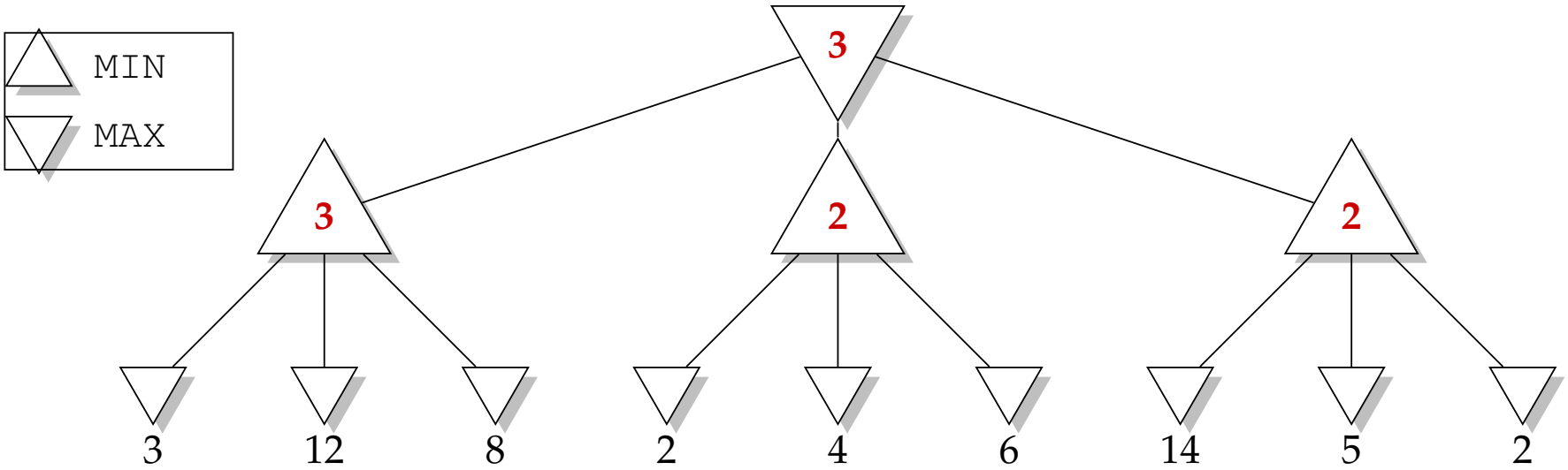
On verra à la fin du cours comment fonctionne alphago, le programme champion du monde de DeepMind.


Kocsis, Levente and Szepesvári, Csaba. Bandit based monte-carlo planning. In Machine Learning : ECML 2006 pp. 282–293. Springer, 2006.


Silver et al. Mastering the game of go with deep neural networks and tree search. Nature, 2016.

Jeux à deux joueurs

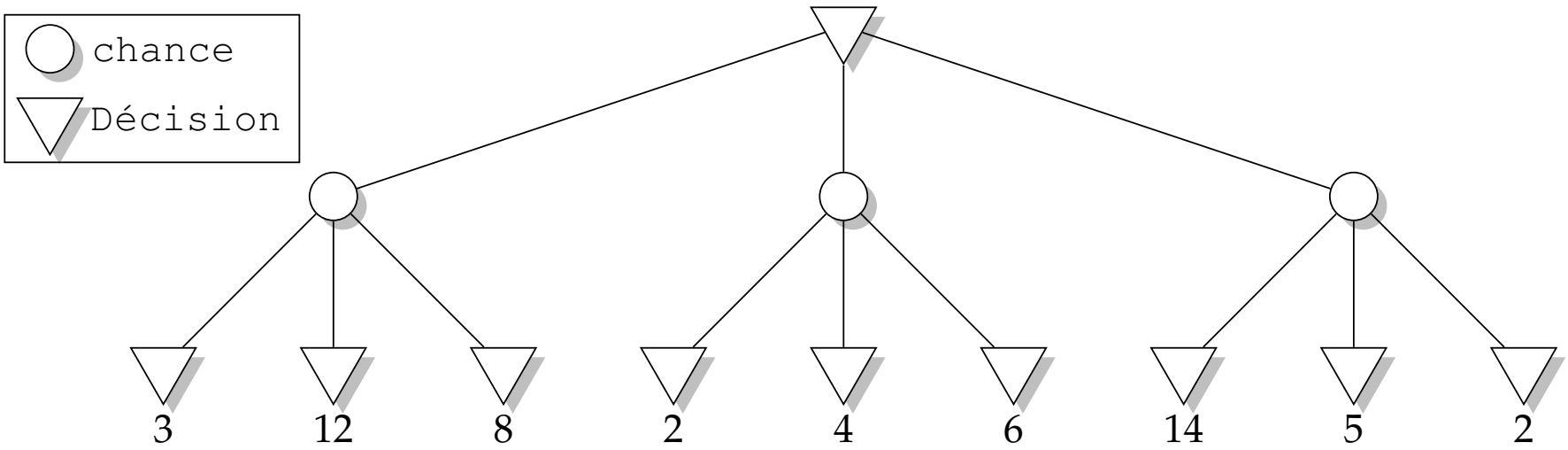
on vous a présenté l'algorithme `minimax` pour jouer à des jeux à deux joueurs, avec information complète.



On raisonne sur les coups possibles de l'adversaire et on choisit son premier coup en fonction.

Que se passet-t-il si l'adversaire est "la nature" et qu'on ne sait pas précisemment comment elle joue ?

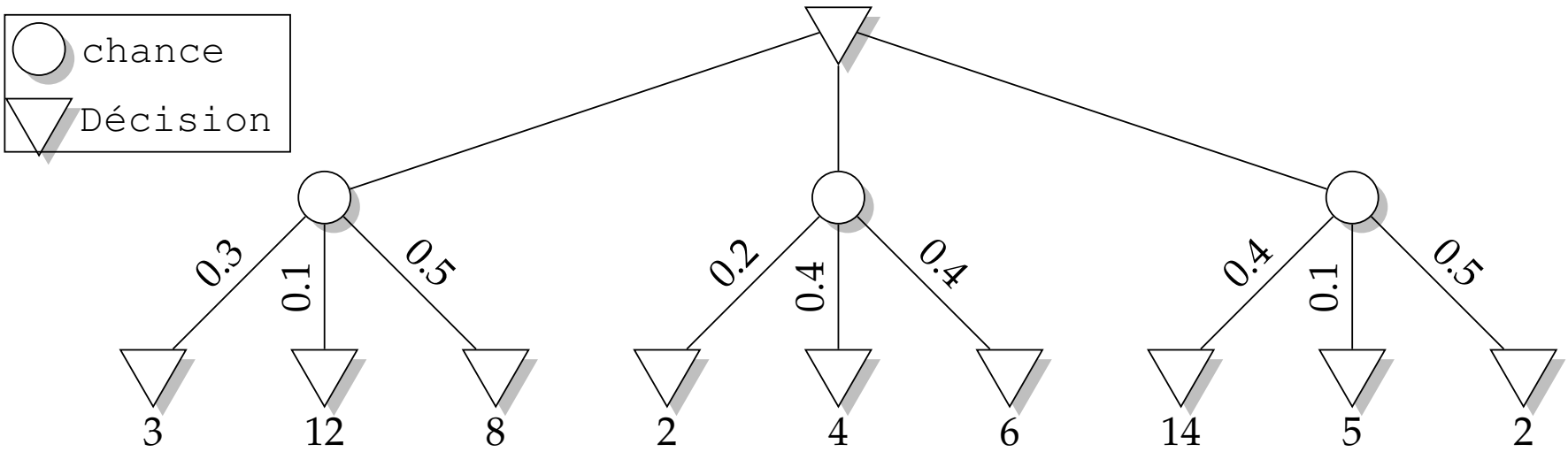
Arbres de décision



Comment jouer maintenant ?

- Le joueur MIN est remplacé par "la nature"
- si on a des probabilités, on peut choisir de prendre l'action qui donne la meilleure espérance.
- si on n'a pas d'information ⇨ approche dans le pire des cas (on considère que la nature est un joueur MIN)

Arbres de décision



On va utiliser EXPECTIMAX

- pour les noeuds `décision` ⇨ comme les noeuds MAX dans maxmin
- pour les noeuds `chance` ⇨ on calcule la valeur moyenne des valeurs des enfants.

Algorithmes

```
1 function EXPECTIMAX(s) returns an action
2 arg max_{a∈actions(s)} EXPECTEDVAL(result(s,a))
```

```
1 function EXPECTEDVAL (s) returns a utility value
2     if terminal?(s) then return utility(s)
3     v ← 0
4     for each n ∈ next(s) do
5         v ← v + ℙ(n) × EXPECTIMAX(n)
6     return v
```

élagage pour la recherche expectimax ?

A part les valeurs des feuilles, on n'observera pas (peu) les valeurs des noeuds pour une partie donnée !

élagage pour la recherche expectimax ?

A part les valeurs des feuilles, on n'observera pas (peu) les valeurs des noeuds pour une partie donnée !

Si on une borne sur les valeurs des feuilles, on peut peut-être faire de l'élagage, mais sinon NON !

élagage pour la recherche expectimax ?

A part les valeurs des feuilles, on n'observera pas (peu) les valeurs des noeuds pour une partie donnée !

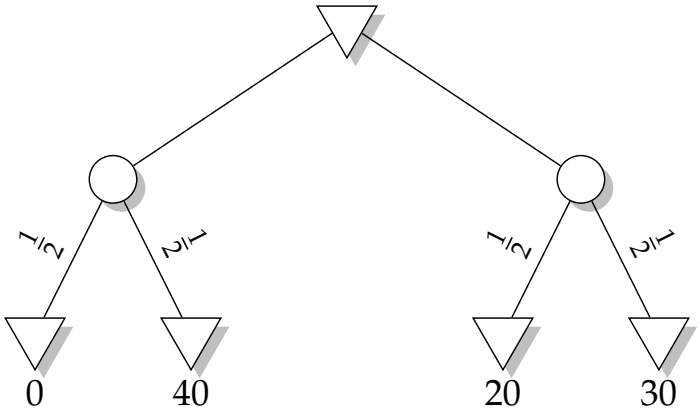
Si on une borne sur les valeurs des feuilles, on peut peut-être faire de l'élagage, mais sinon NON !

On peut utiliser depth limited expectimax avec des fonctions d'évaluation si on n'a pas le temps d'étudier tout l'arbre

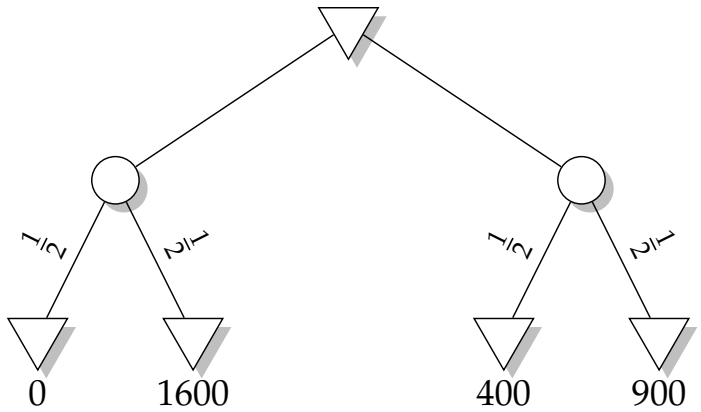
Danger des échelles d'utilités

Pour minimax, l'échelle des valeurs des feuilles n'était pas importante, tant que l'ordre entre les valeurs est conservé, le raisonnement sera correct.

pour expectimax, ce n'est pas le cas !



⇨ on prend le carré des valeurs



Récapitulatif

- si on ne connait pas les probabilités : pour le moment, pas mieux que MINIMAX ! approche pessimiste !
- si on connait les probabilités
  - l'approche pessimiste est toujours disponible
  - approche en moyenne
  - ⇨ cela dépend comment on modélise la situation !

Généralisation de expectimax

Avec les arbres de décisions, on a vu un premier type de décision séquentielle dans l'incertain.

On va maintenant voir une généralisation.

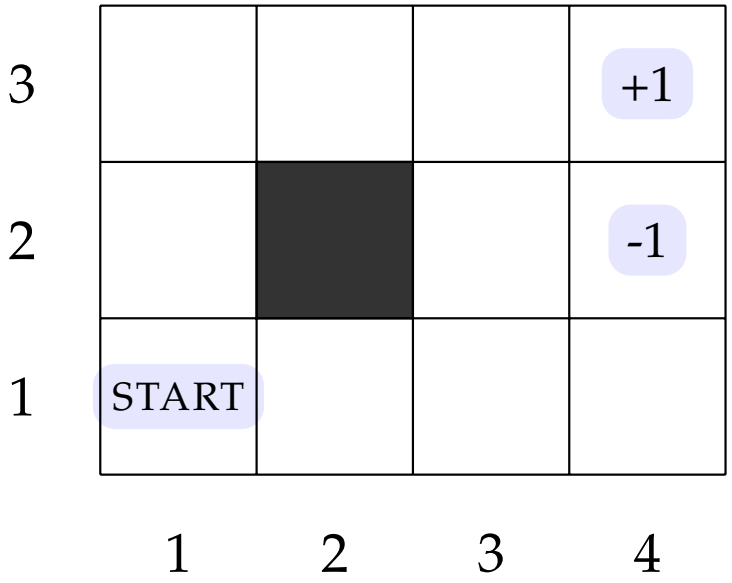
Exemple canonique : un robot se déplace dans une grille.

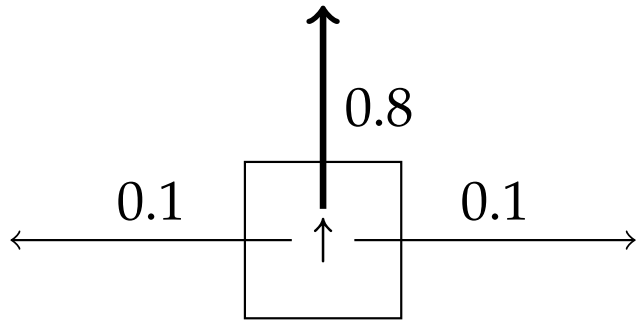
- Les murs bloquent le déplacement de l'agent. Il ne connait pas a priori la position de tous les murs
- Il peut tomber dans des trous, il ne connait pas la position de chaque trou ou s'ils existent
- le robot est un peu plus réaliste : ses actions peuvent échouer (les roues peuvent patiner, ou non)
  - on estime que l'action D mène bien le robot dans la case au D,
  - mais dans 10% des cas, il dévie vers la gauche
  - mais dans 10% des cas, il dévie vers la droite
  - Si $D = Nord$, il a 80% de chance d'aller dans la case au nord, 10% d'aller dans la case à l'est, 10% d'aller dans la case à l'est.

Exemple : gridworld

- l'agent paie une pénalité pour chaque déplacement (il dépense de l'energie)
- certaines cases peuvent contenir une grosse récompense

cf Opportunity et Curiosity sur Mars...





exemple action vers le haut

Définition (Processus décisionnel de Markov)

Un *Processus décisionnel de Markov* est un tuple $\langle S, A, T, R, \gamma \rangle$ où

- S est un ensemble fini d'états
- A est un ensemble fini d'actions
- T est une matrice de transition
  $T^a_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$ probabilité d'arriver dans l'état s' à l'instant +1 quand on a pris l'action a dans l'état s à l'instant t
- R est le vecteur de récompenses
  $R^a_s = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$ valeur moyenne obtenue après avoir pris l'action a dans l'état s
- un ensemble d'état initial
- parfois un ensemble d'états terminaux

Hypothèse de Markov

Si on exécute l'action a dans l'état s :

- On obtient une récompense r
- On arrive dans un état s'

En principe, r et s' peuvent dépendre de tout l'historique !

Définition (Etat de Markov)

Un état S_t est dit de **Markov** ssi

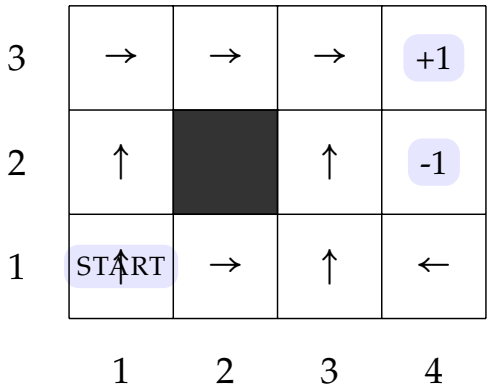
$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t]$$

- "Etant donné le présent, le futur est indépendant du passé"
- Une fois que l'état est connu, on peut effacer son historique !
- *ex :* aux échecs, l'état du jeu ne dépend pas de l'historique des coups !

Les composants d'un agent : politique

C'est ce qui gouverne le comportement de l'agent

- **politique déterministe** : La fonction associe à chaque état **une action** $\pi : S \mapsto A$



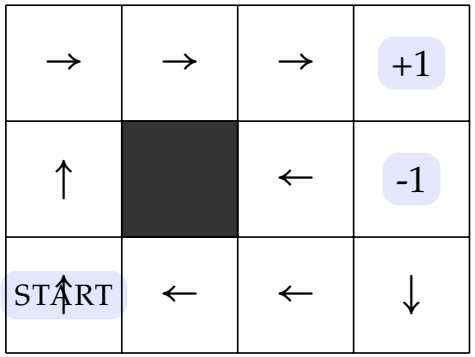
politique optimale pour un pénalité de 0.03 par déplacement

- **politique stochastique** : une distribution de probabilité sur les actions possibles

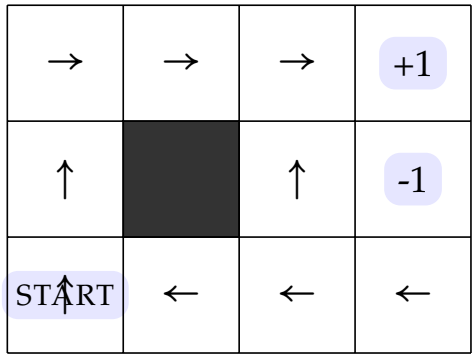
$$\pi : S \mapsto \Delta(A)$$

où $\Delta(N)$ désigne une distribution de probabilité sur l'ensemble (fini) N.
$p \in \Delta(N)$ ssi $\forall i \in N, p(i) \in [0,1]$ et $\sum_{i \in N} p(i) = 1$

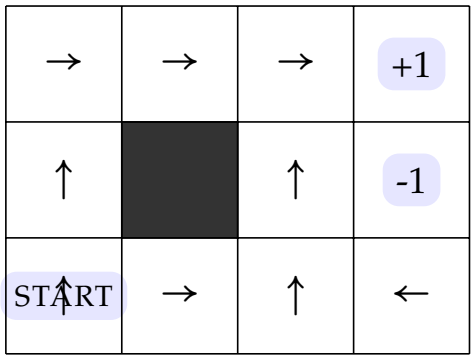
Politiques optimale



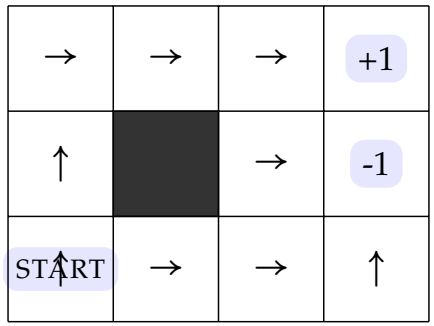
pénalité de 0.01 par déplacement
aucun risque : on fait le tour!



pénalité de 0.02 par déplacement
petit risque



pénalité de 0.04 par déplacement
on prend le chemin le plus court

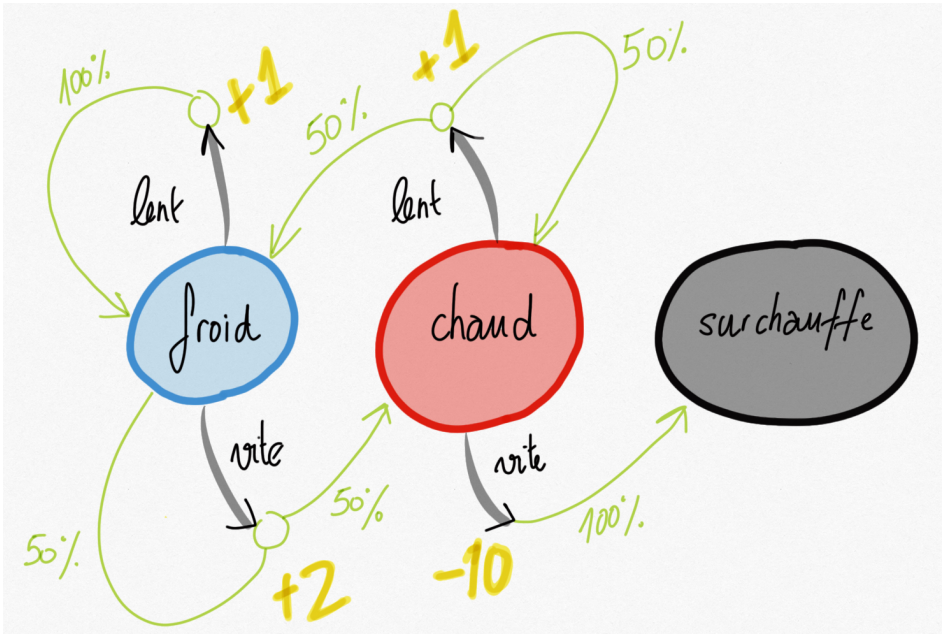


pénalité de 2 par déplacement
on prend des risques pour terminer au plus vite

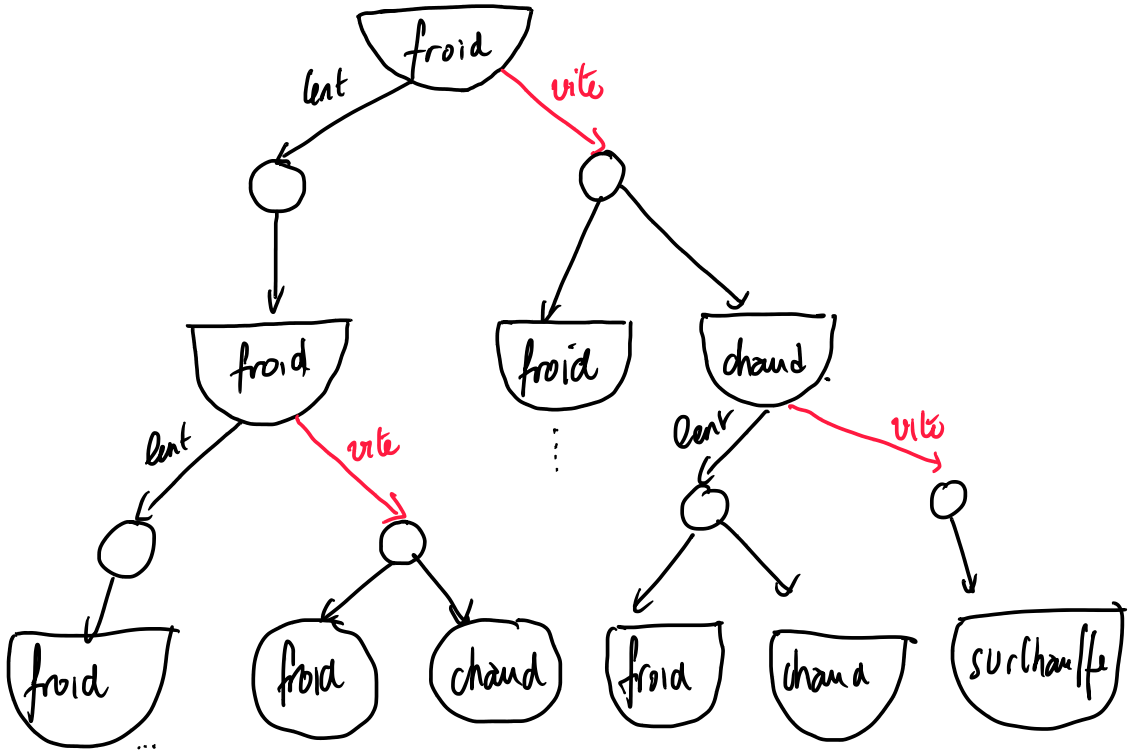
Exemple de la voiture de course

un robot voiture veut aller vite et loin !

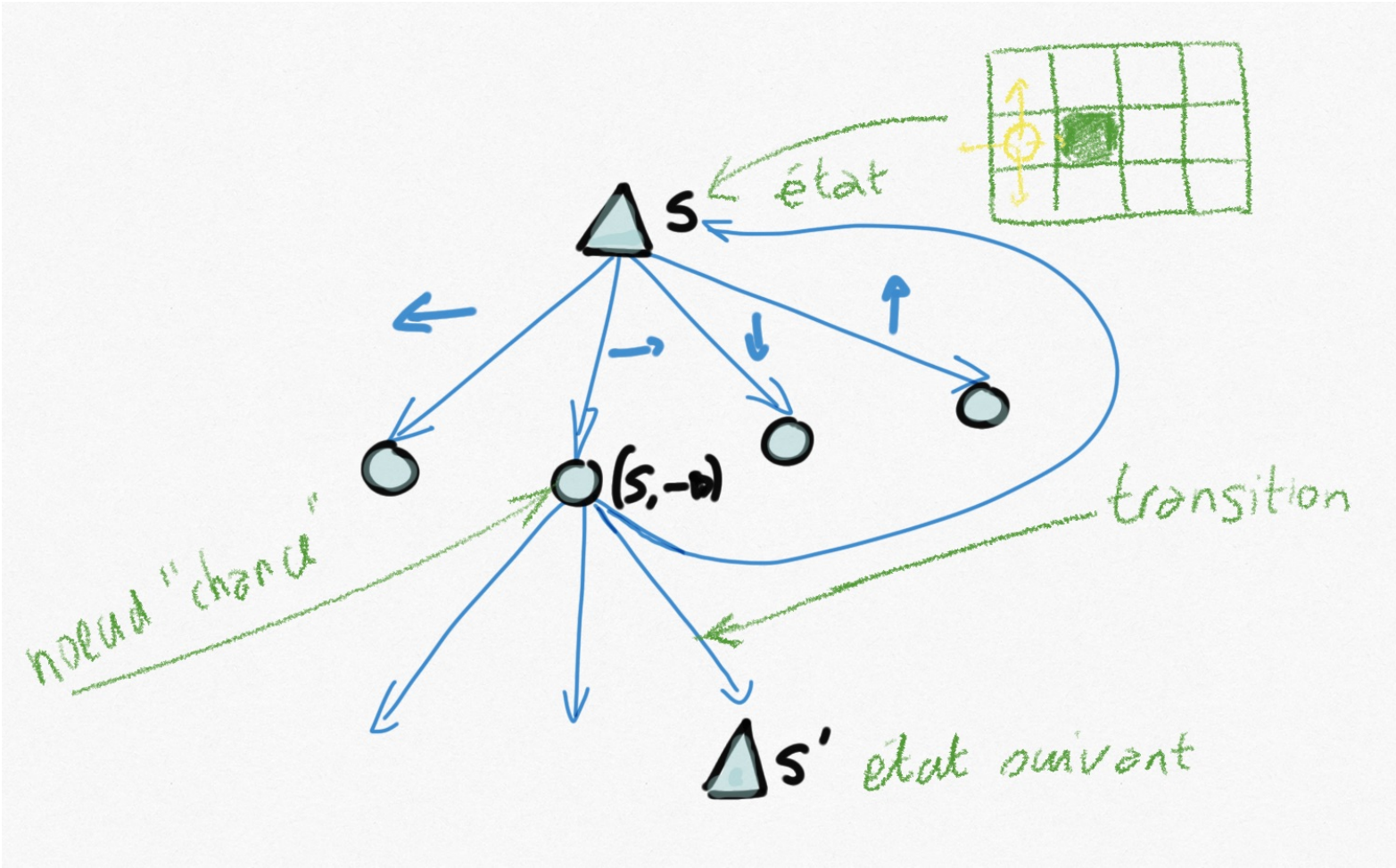
- trois état du moteur : normal, chaud, surchauffe
- deux actions : lent, vite
- en allant plus vite, on double la récompense
- +1 pour les action lent, +2 pour action vite, -10 pour atteindre un état surchauffe.
  la description ne suit pas exactement le modèle formel



Exemple de la voiture de course & expectimax



Exemple de la voiture de course & MDP



Avec les MDP, on ne va pas raisonner sur un arbre, mais sur un graphe.

Quel est notre but

On va obtenir une séquence de valeurs d'utilité. Que préfère-t-on ?

- maintenant, plus tard ? (ex $\langle 0,0,10 \rangle$ ou $\langle 8,2,0 \rangle$)
  vous préférez 10 dans 3 jours, ou 8 aujourd'hui, 2 demain, et 0 dans 3 jours ?

Quel est notre but

On va obtenir une séquence de valeurs d'utilité. Que préfère-t-on ?

- maintenant, plus tard ? (ex $\langle 0,0,10 \rangle$ ou $\langle 8,2,0 \rangle$)
  vous préférez 10 dans 3 jours, ou 8 aujourd'hui, 2 demain, et 0 dans 3 jours ?
- généralement, on préfère avoir plus d'utilité en tout, mais une distribution plus régulière peut être appréciable !

Quel est notre but

On va obtenir une séquence de valeurs d'utilité. Que préfère-t-on ?

- maintenant, plus tard ? (ex $\langle 0,0,10 \rangle$ ou $\langle 8,2,0 \rangle$)
  vous préférez 10 dans 3 jours, ou 8 aujourd'hui, 2 demain, et 0 dans 3 jours ?
- généralement, on préfère avoir plus d'utilité en tout, mais une distribution plus régulière peut être appréciable !
- Dans certains problèmes où il y a une **fin**, on pourra chercher à maximiser la somme des utilités

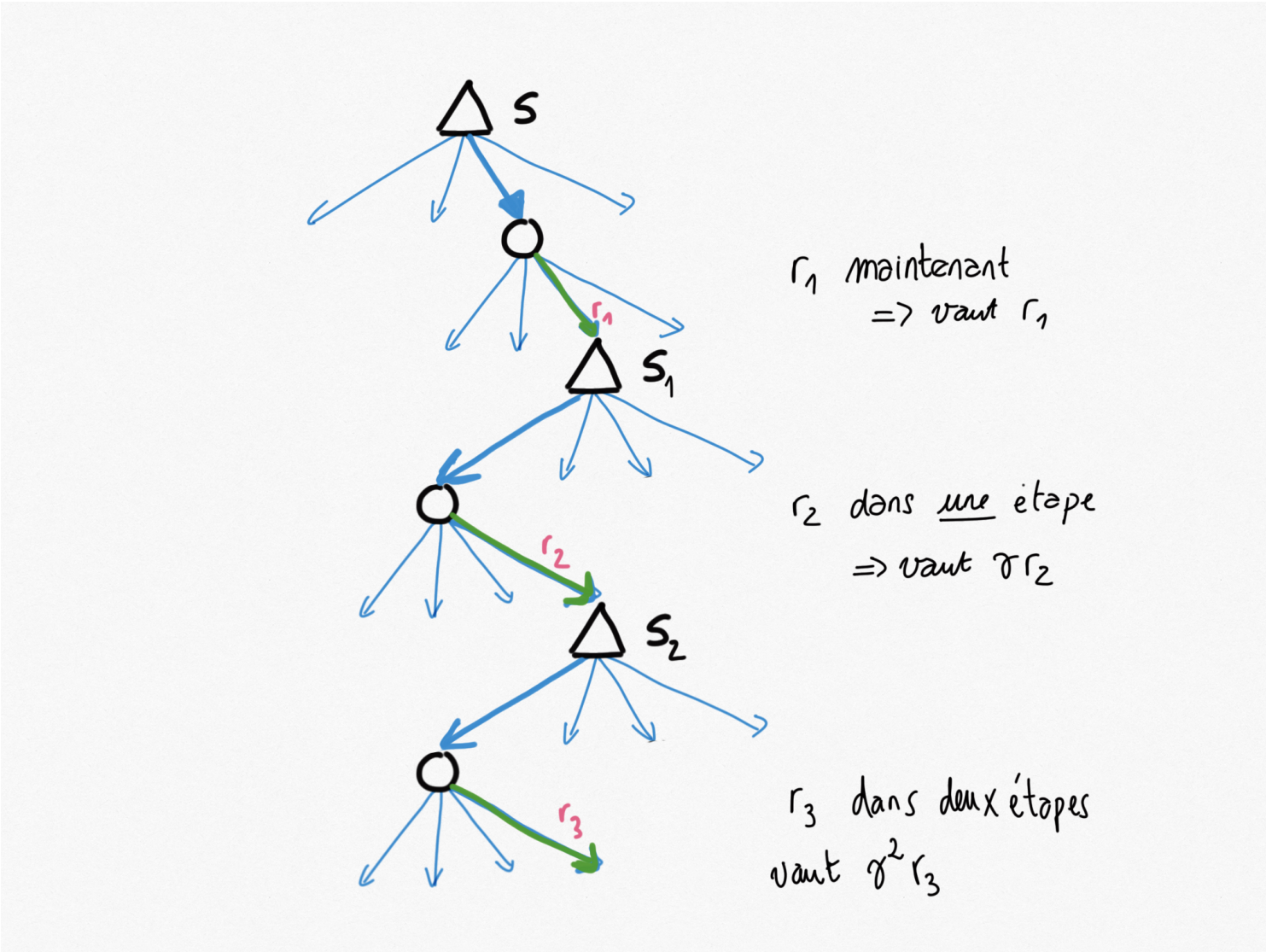
Quel est notre but

Dans tous les autres cas, on peut utiliser un taux d'escompte γ

- avoir 1 aujourd'hui vaut 1 aujourd'hui !
- avoir 0 aujourd'hui et 1 demain vaut aujourd'hui γ aujourd'hui
- avoir 0 aujourd'hui, 0 demain et 1 dans deux jours vaut γ^2 aujourd'hui

- $\gamma = 0$ l'agent est "myope" : il n'est intéressé que par la récompense immédiate
- $0 < \gamma < 1$ l'agent cherche un équilibre entre la récompense immédiate et celle qu'il obtiendra dans le futur

Récompense avec escompte



Si on raisonne sur ce chemin, la récompense sera $r_1 + \gamma r_2 + \gamma^2 r_3$.

Quel est notre but : version formelle

- pour des tâches épisodiques
  - il y a des états terminaux et initiaux
  - on repart dans un état initial une fois qu'on atteint un état terminal

  ⇨ maximise le cumul des récompenses sur *un épisode*

  (ici de longueur T, mais la longueur de chaque épisode peut différée)

$$G_T = r_1 + r_2 + \cdots + r_T$$

- pour des tâches en continue

  ⇨ maximise une récompense "escomptée" $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$

  γ est le taux d'escompte
  - $\gamma = 0$ l'agent est "myope" : seule la récompense immédiate importe
  - $0 < \gamma < 1$ quand $\{r_t, t \in \mathbb{N}\}$ est bornée, alors R_T est bien définie.

    ⇨ l'agent cherche un équilibre entre la récompense immédiate et celle qu'il obtiendra dans le futur

- maximiser une récompense "en moyenne" $G_t = \lim_{T \to +\infty} \frac{1}{T} \cdot \sum_{k=0}^{T} r_{t+k+1}$

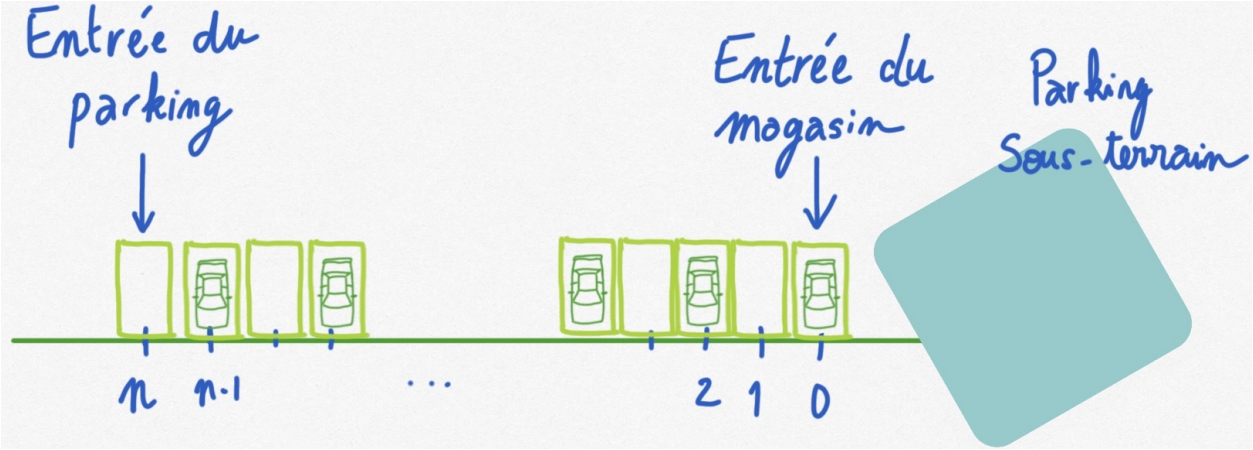
Focus pour le cours

- problèmes itératifs en continue.
- objectif : maximiser la somme "avec dévaluation" $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$
  - Pour éviter une récompense infinie si on tombe dans des cycles
  - Le futur reste uncertain ! Bon compromis entre court et long terme
  - Tendance naturelle vers le court terme
  - Mathématiquement, c'est quand même pratique !
- la fonction de transition est stochastique
- la fonction de récompense est connue

Comment trouver la meilleure politique ?

Exercice de Modelisation

Le parking d'un magasin est très long, construit le long d'une voie à sens unique qui se termine au magasin et à un parking souterrain. L'objectif des clients est de marcher le moins possible après s'être garé, mais les places sont rares et ils cherchent à éviter de rentrer dans le parking souterrain.



- chaque place est libre avec une même probabilité p
- le conducteur ne peut voir si la place est libre que lorsqu'il est devant.
- le conducteur ne peut pas faire marche arrière
- les places sont numérotées de 0 à n, l'entrée du magasin est devant le 0, l'entrée du parking est situé au niveau de la place n.
- se garer dans le parking souterrain "coûte" $C > 0$, une place à une distance i de l'entrée "coûte i"

Mise en bouche : le problème des k-bandits manchots

k bandits manchots $\Leftrightarrow$ k machines à sous (en anglais *k-armed bandit problem*)

Supposons qu'on ait à notre disposition un saut de 1000 jetons pour jouer aux machines à sous.

Supposons qu'il y a k machines à sous, et qu'elles ont des propriétés différentes sur les gains.

On suppose que chaque le fonctionnement machine est indépendente de chacunde des autres.

Quelle est notre politique pour utiliser nos 1000 jetons ?

Autres applications des bandits

- Essais cliniques :
  on a k molécules pour un symptôme et l'efficacité est incertaine.
  ⇨ étant donnée la réponse sur les précédents patients ayant le même symptôme, quel traitement est à proposer ?
- Publicité en ligne (€€€) k publicités peuvent être affichées
  Laquelle doit-on afficher pour un utilisateur, basé sur le comportement d'autres utilisateurs (similaires) ?
- dans des jeux (ex Go), on a k coups possibles, lequel choisir pour nous mener à la victoire ?

problème des k-bandits manchot

On note A_t la machine choisie pour jouer le t^{ième} jeton et R_t la récompense obtenue.

On suppose que chaque machine a utilise une distribution de probabilité de moyenne $q_\star(a) = \mathbb{E}[R_t | A_t = a]$

Si on connaissait $q_\star(a)$ pour chaque machine, le problème est trivial

⇨ on passe notre temps à jouer $\operatorname{argmax}_a q_\star(a)$!

peut être pas complètement trivial : si deux machines ont la même valeur moyenne mais des variances différentes, etc...

Mais on peut former une estimation.

Supposons qu'on ait une estimation pour chaque machine

- vous jouez la machine qui a la meilleure estimation ⇨"action gloutonne"
  ⇨ on exploite sa connaissance actuelle.
- Vous ne jouez pas pas l'action gloutonne ⇨ exploration.
- Durant l'exploration, la récompense est moins bonne à court terme. Mais elle peut permettre de plus grands gains à long terme !

⇨ il faut trouver *un équilibre* entre l'exploitation et l'exploration.

- ce problème a été étudié en mathématique, et on peut construire des méthodes sophistiquées pour trouver le meilleur équilibre, mais cela nécessite des hypothèses précises sur les machines.

Une estimation simple

On va utiliser la moyenne des gains obtenus jusqu'au moment présent.

$$
\begin{aligned}
Q_t(a) &= \frac{\text{Somme des récompenses obtenues en jouant } a}{\text{nombre de fois où on a joué } a} \\
&= \frac{\sum_{i=1}^{t-1} R_i 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}}, \text{ où } 1_{cond} \begin{cases} 1 \text{ si cond est vraie} \\ 0 \text{ sinon} \end{cases}
\end{aligned}
$$

en utilisant la loi des grands nombres, $Q_t(a)$ va converger vers $q_\star(a)$.

$$\lim_{t \to +\infty} Q_t(a) = q_\star(a)$$

Acton gloutonne et exploration ϵ-gloutonne

Une action gloutonne est donc

$$\operatorname*{argmax}_a Q_t(a)$$

ϵ-glouton : une variante est d'être glouton presque tout le temps, et de temps en temps, d'explorer aléatoirement

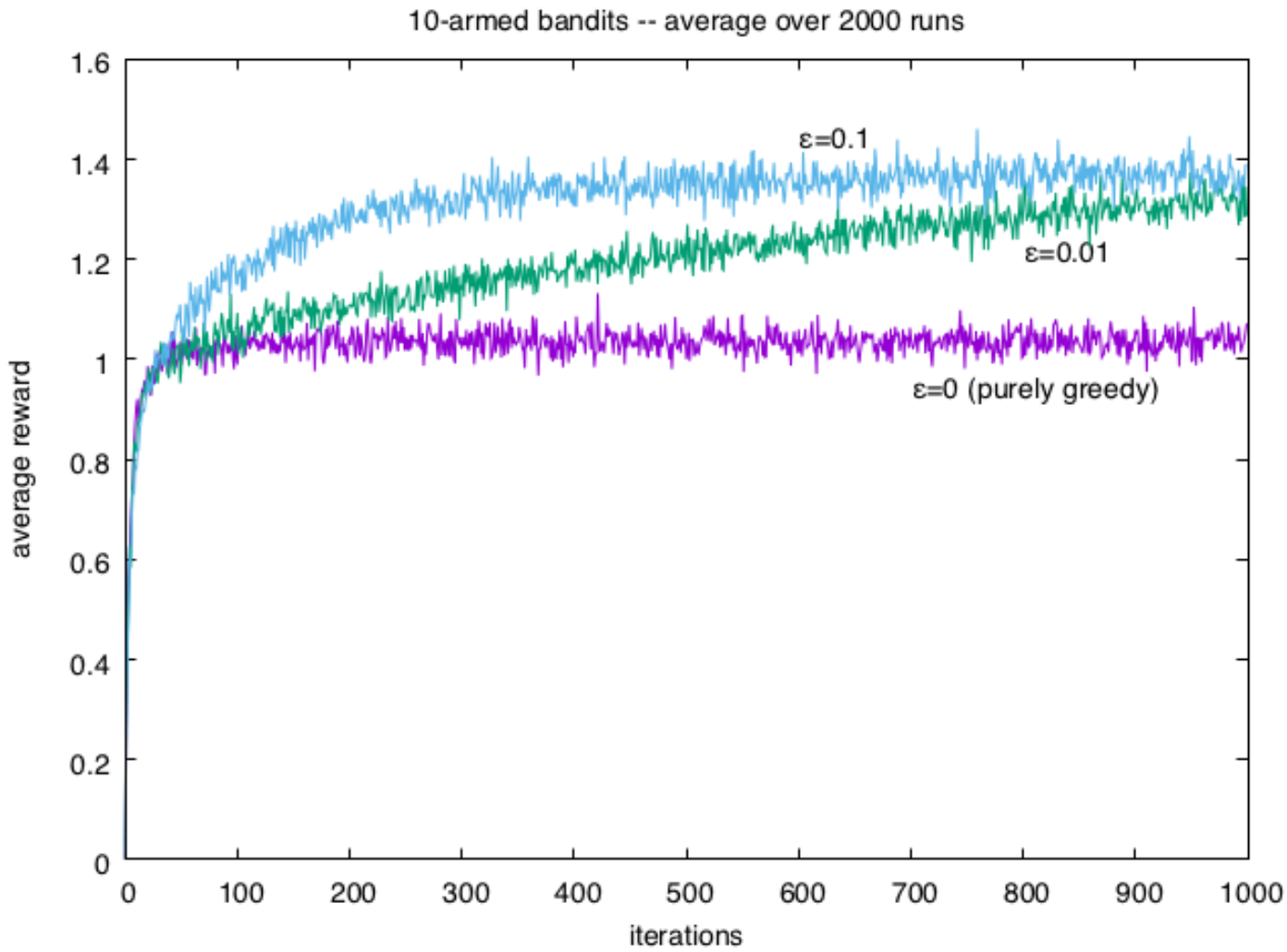
- avec une probabilité ϵ, on choisit une action au hasard (en utilisant une probabilité uniforme sur les machines)
- avec une probabilité $1 - \epsilon$, on choisit l'action gloutonne

Evaluation empirique

On utilise une simulation avec 10 machines à sous.

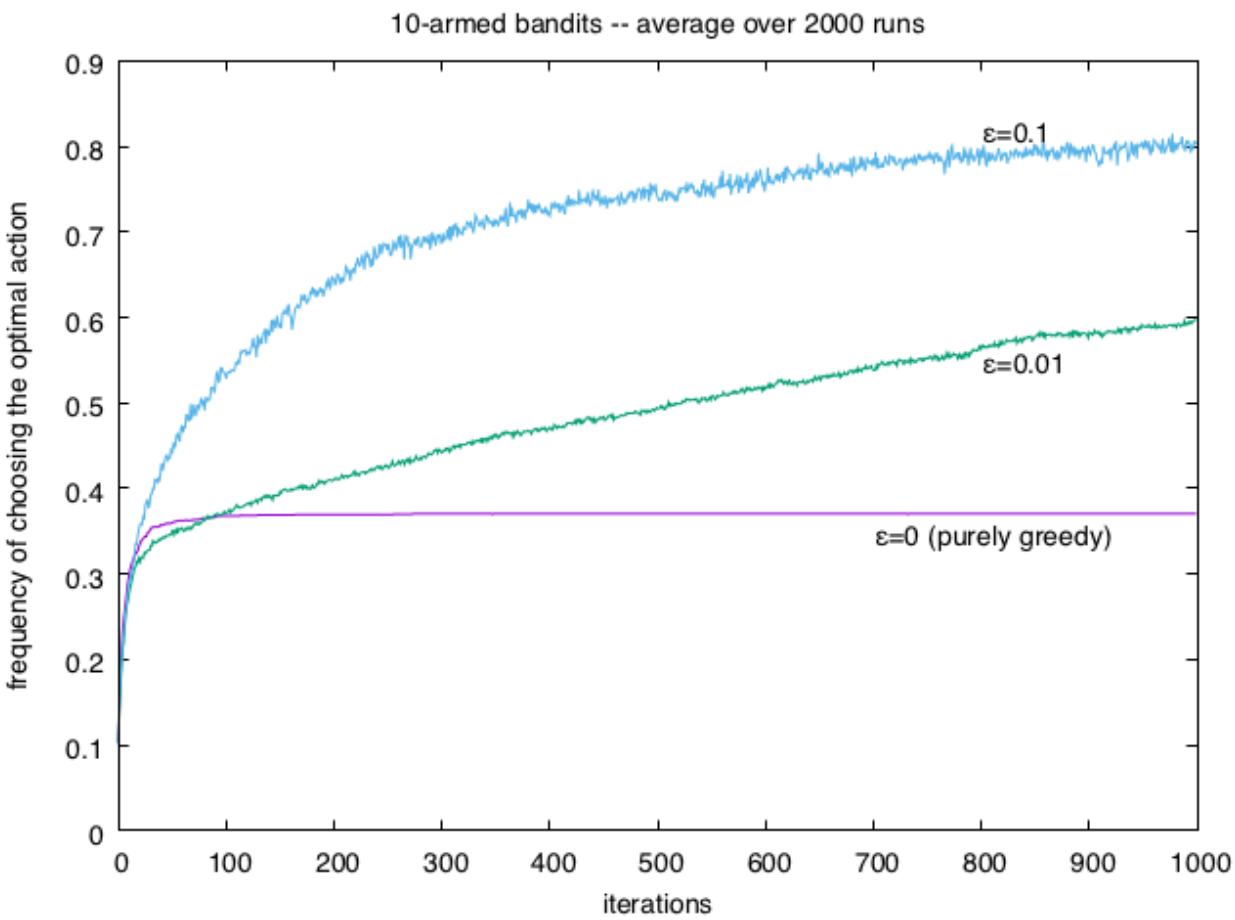
- Pour chaque machine a, on génère $q_\star(a)$ en tirant un échantillon d'une distribution normale de moyenne 1 et de variance 1 ($N(0,1)$).
- à chaque fois qu'on joue à la machine a, on tire un échantillon en suivant une loi normale de moyenne $q_\star(a)$ et de variance 1.
- on utilise un algorithme en choisissant 1000 fois une machine.
  ⇨ un "run"
- pour obtenir des informations plus fiables, on répète 2000 "runs" (on tire au sort les valeurs $q_\star$, puis on choisit 1000 fois une machine)

Evaluation empirique : Glouton, ϵ-glouton



récompenses en moyenne

Evaluation empirique : Glouton, ϵ-glouton



fréquence du choix de la meilleure machine

Evaluation empirique : Glouton, ϵ-glouton

- la méthode glouton se "bloque" trop vite sur une machine non optimale.
- 0.1-glouton explore assez rapidement, mais va plafonner à la limite (en théorie, devrait atteindre 91% de bon choix)
- 0.01-glouton explore plus lentement au début, mais avec plus d'itérations, devrait atteindre un meilleur résultat.
- en regardant ce graphe, et si vous avez 1000 jetons, quelle stratégie utilisez vous ?

Implémentation efficace

On tient à calculer l'estimation de la valeur des machines de manière computationnellement efficace !

- on veut éviter de stocker toutes les données
- on veut éviter de faire une boucle pour calculer la moyenne

C'est trivial, mais c'est une préoccupation à avoir !

Implémentation efficace

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\
&= \frac{1}{n}\left(R_n + nQ_n - Q_n\right) \\
&= Q_n + \frac{1}{n}\left(R_n + -Q_n\right)
\end{aligned}
$$

Dans ce cours, on va beaucoup utiliser une règle de mise à jour

$$estimate_{new} \leftarrow estimate_{old} + \alpha \cdot (observation - estimate_{old})$$

Mise à jour

$$estimation_{new} \leftarrow estimation_{old} + \alpha \cdot (observation - estimation_{old})$$

Etant donnée la nouvelle observation

- s'il semble que notre estimation est surestimée
  ⇨ on baisse notre estimation
- s'il sembe que notre estimation est sousestimée
  ⇨ on augmente notre estimation
- $(observation - estimate_{old})$: estimation de l'erreur
- la réduction ou l'augmentation se fait proportionnellement à l'erreur, avec un coéfficient α (généralement petit).
- α peut dépendre du nombre de mises à jour effectuée
  dans notre exemple $\alpha = \frac{1}{n}$ où n est le nombre de fois qu'on a choisi une machine en particulier.

Et si le réglage des machines à sous varie au cours du temps ?

Le problème est **non-stationnaire** et dépend donc du temps.
Idée : on peut donner un poids plus *lourd* aux récompenses *récentes* par rapport aux récompenses obtenues il y a longtemps.
Par exemple :

$$Q_{n+1} = Q_n + \alpha [R_n - Q_n]$$

où le paramètre $\alpha \in]0,1]$.

$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha [R_n - Q_n] \\
&= \alpha R_n + (1-\alpha) Q_n \\
&= \alpha R_n + (1-\alpha) [\alpha R_{n-1} + (1-\alpha) Q_{n-1}] \\
&= \alpha R_n + \alpha(1-\alpha) R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + \alpha(1-\alpha) R_{n-1} + (1-\alpha)^2 [\alpha R_{n-2} + (1-\alpha) Q_{n-2}] \\
&= \dots \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1-\alpha)^{n-i} R_i
\end{aligned}$$

Et si le réglage des machines à sous varie au cours du temps ?

$$Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$$

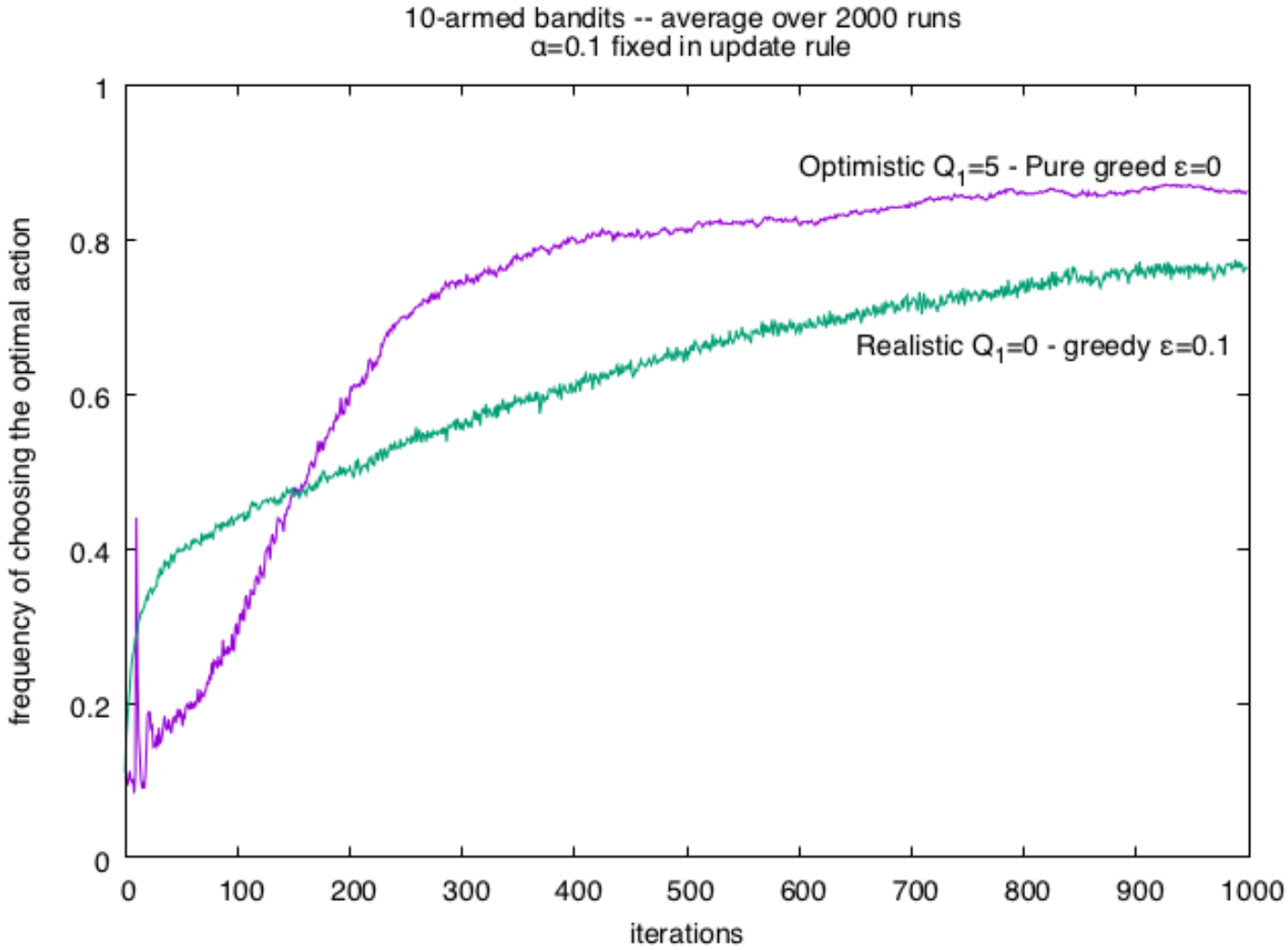
- c'est une moyenne pondérée
  on peut vérifier que la somme des poids est égale à 1
- $0 < \alpha \leq 1$ donc on a une chute exponentielle du poids en fonction de l'ancienneté.

Initialisation de l'estimation

Quelle valeur pour $Q_1(a)$?

- le choix de $Q_1(a)$ peut introduire un biais
- pas un problème en soi, et peut même être utile ⇨ on peut *forcer* l'exploration
- **idée** : initialisation optimiste des valeurs initiales.
- ⇨avec nos méthodes gloutonnes, on va explorer ces états qui apparaissent prometteurs
  - si ce ne sont pas de bons états ⇨ l'estimation va vite baisser
  - on va forcer que chaque état soit exploré un certains nombre de fois, ce qui va aider la convergence !

Evaluation empirique



C'est une ruse qui fonctionne généralement bien pour les problèmes stationnaires.

Utiliser un intervalle de confiance

- On a besoin d'explorer
- ϵ-glouton traite les actions non-gloutonne de la même manière.

Cependant :

- Certaines actions ont peut-être été utilisées plus souvent.
- On n'a peut être pas intérêt à tester plusieurs fois une action vraiment mauvaise.
- Il faut peut être tester plus vite des actions qui paraissent plus prometteuses.

⇨ confiance de l'estimation

⇨ Méthode des intervalles de confiance

Utiliser un intervalle de confiance

- idée : essayer de mesurer l'incertitude sur la valeur Q(a) qu'on estime
- but : on calcule une borne supérieure probable de la valeur Q(a) qu'on estime
- La borne supérieure est calculée par l'expression

$$Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}$$

- plus la borne est élevée, plus a est prometteuse !

On va donc choisir l'action $\underset{a}{\text{argmax}}\left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}\right]$

⇨ On explore les actions les plus prometteuses
⇨ meilleure utilisation des échantillons.

Utiliser un intervalle de confiance

$$Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}$$

- $N(a)$ est le nombre de fois où l'action a a été choisie
- c est une constante qui détermine la mesure de confiance
- t est le nombre total d'itérations

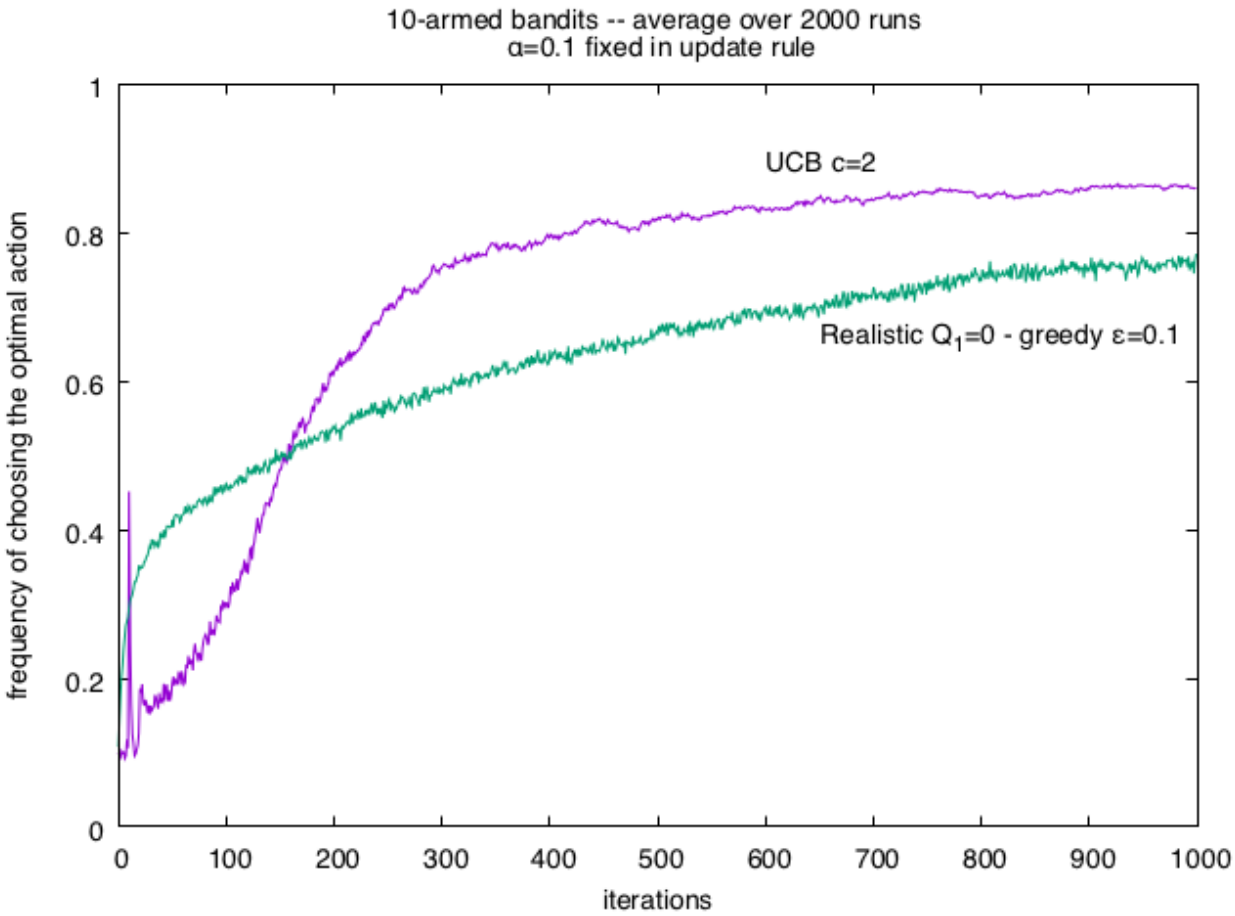
⇨ si a est choisie ⇨ l'incertitude baisse (donc la borne aussi)

⇨ si a n'est pas choisie, t croît, et l'incertitude croît.

- avec le log, l'accroissement devient de plus en plus petit.

Peter Auer, Nicolò Cesa-Bianchi, Paul Fischer. (2002). Finite-time Analysis of the Multiarmed Bandit Problem, *Machine Learning*, 47, 235—256

Evaluation empirique



fréquence du choix de la meilleure machine

Utiliser un intervalle de confiance

Marche avec des problèmes de bandits stationnaires.

Personne n'a encore réussi à adapter ce type d'idée pour les modèles plus compliqués d'apprentissage par renforcement que l'on va voir dans le cours.

certains d'entre vous on peut-être vu son utilisation dans les jeux à deux joueurs (Monte Carlo et UCB ont donné de bons résultats).

Montée de gradient stochastique

- jusqu'ici : on apprend une estimation, puis on utilise cette estimation pour choisir une action
- maintenant, on veut apprendre une fonction de préférences : plus on préfère une action, plus souvent on va la prendre.
  On note $H_t(a)$ la préférence pour l'action a au pas de temps t.

⇨ idée de distribution de Boltzmann

$$Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_{b=1}^{10} e^{H_t(b)}} = \pi_t(a)$$

- On peut utiliser l'idée de la montée de gradient stochastique $\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}$

$$H_{t+1}(a) = \begin{cases} H_t(a) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(a)) & \text{if } a = A_t \\ H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a) & \text{otherwise} \end{cases}$$

où $\bar{R}_t$ est la moyenne des récompenses jusqu'au pas de temps t, comme calculé avant.

Montée de gradient stochastique