SCALIAN

ALGORITHMES DE RECHERCHE DE GRAPHES

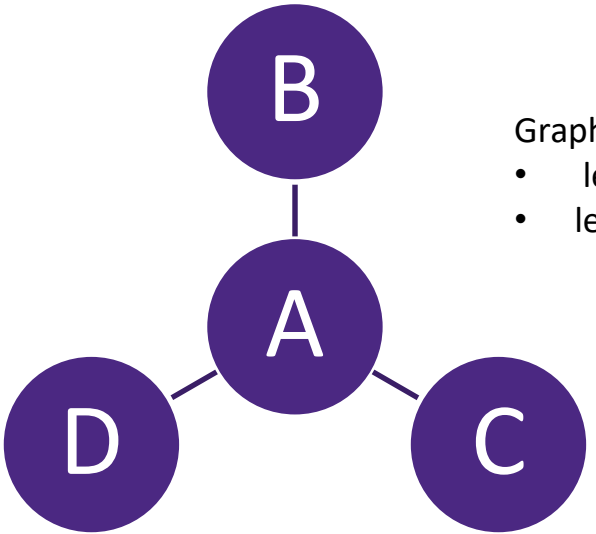
RAPPELS SUR LES GRAPHES

RAPPEL SUR LES GRAPHES

Un graphe peut être défini comme un ensemble de “points” pouvant être reliés entre eux.

Les “points” sont appelés **noeuds ou sommets** (node ou vertex en anglais)

Les connections sont appelées **arêtes ou arcs** (edge en anglais)



Graphe défini par :

- les nœuds [A,B,C,D]
- les arêtes [(A,B), (A,C), (A,D)]

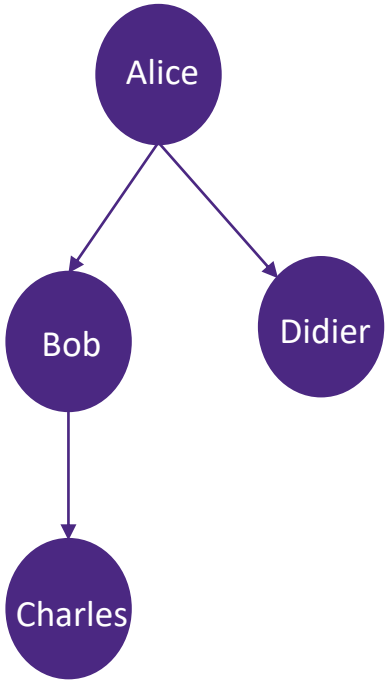
Il s’agit d’un graphe particulier dit « **en étoile** »

LES GRAPHES EN RL

SCALIAN

Les graphes utilisés partout en datascience

- **Bases de données graphes**
- **Tensorflow et réseaux de neurones**
- **Pipeline de votre projet de ML !**
- **Réseaux sociaux**
- **Arbre de decisions**
- **Clustering hierarching**
- **…**



LES GRAPHES EN RL

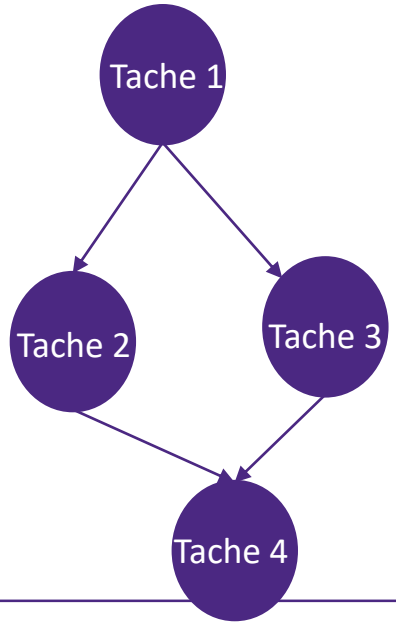
SCALIAN

Les graphes sont une des structures de données les plus utilisées dans les problemes de RL.

- **Pour représenter les transitions état-action-état :**
- Un noeud est un état
- Il y a une arête entre A et B si une action permet de passer de l'état A à l'état B
- L'algorithme de renforcement "réfléchit" alors sur le graphe, par exemple en essayant de trouver le chemin permettant d'accéder au sommet avec la plus grande valeur.
- Environnements **discrets uniquement**

- **Pour représenter l'environnement**
- Labyrinthe
- Réseau routier
- Relations entre personnes

- **Pour représenter des dépendances dans des processus à optimiser**
- Ordonnancement de taches sur des serveurs et super-calculateurs
- Optimisation de tâches pour de la gestion de projet



DÉFINITION SUR LES GRAPHES

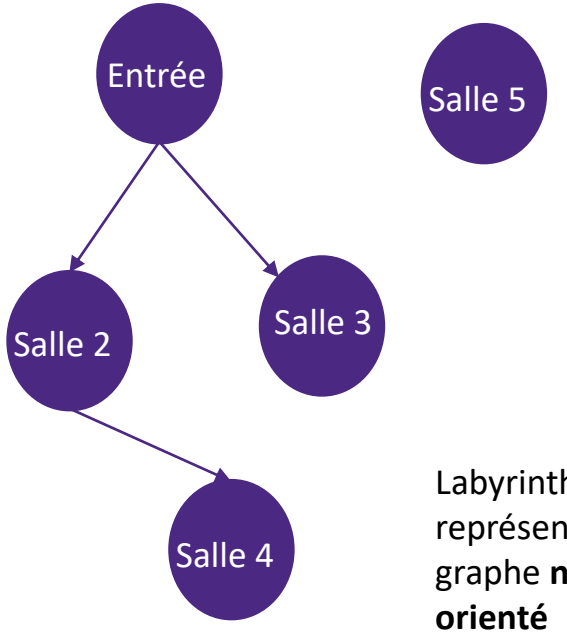
SCALIAN

Un graphe est **orienté** si connections sont **orientées**.

- On parle d'arêtes pour un graphe non orienté, et **arcs pour un graphe orienté**.
- Dans un graphe orienté, la presence d'un arc de A vers B **n'implique pas** la presence d'un arc de B vers A.

Tout les sommets d'un graphe ne sont pas forcément relies à d'autres sommets

- Un graphe est **connexe** si quel soit le sommet A de depart et B d'arrivée, il existe un **chemin** d'arêtes menant de A vers B.



Labyrinthe représenté par un graphe **non connexe orienté**

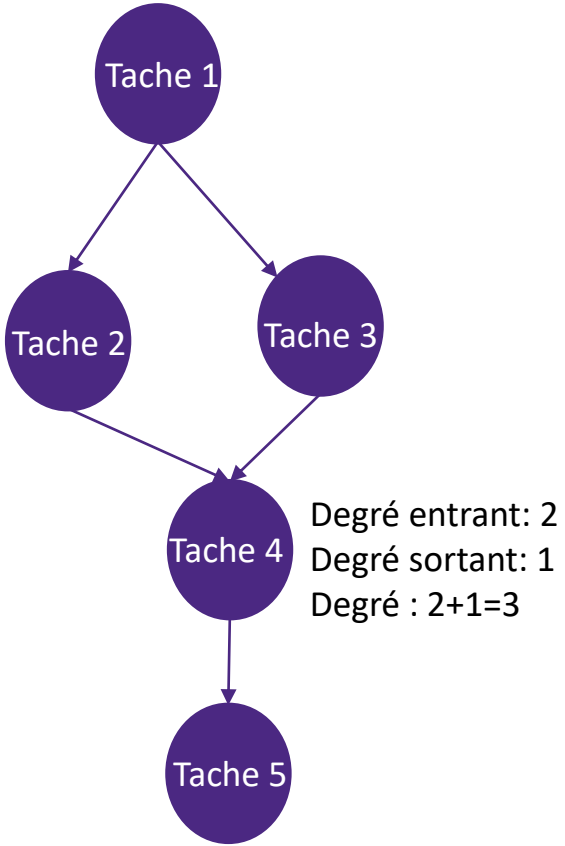
DÉFINITION SUR LES GRAPHES

Le **degré d'un sommet** est le nombre d'arêtes connecté à ce sommet.

Le **degré d'un graphe** est maximum du degré de ses sommets

Si le graphe est orienté, on parle de **degrés entrants et degré sortants**.

- Le degré est alors la **somme des degrés entrants et sortants**.



DÉFINITION SUR LES GRAPHES

SCALIAN

Tâche de RL	Connexe	Cyclique	Orienté	Degré
Labyrinthe				
Une partie d'échecs				
Ordonnancement de tâches				

DÉFINITION SUR LES GRAPHES

SCALIAN

Tâche de RL	Connexe	Cyclique	Orienté	Degré
Labyrinthe	ça dépend	ça dépend	ça dépend	Le nombre maximum de sortie d'une salle
Une partie d'échecs	Oui	Oui	Oui	Le nombre maximum d'actions possibles sur un état
Ordonnancement de tâches	Non	Non	Oui	Degré entrant : nombre maximum de tâches dont une tâche peut dépendre. Degré sortant : nombre maximum de tâches pour lequel une même tâche est nécessaire

LES ARBRES

SCALIAN

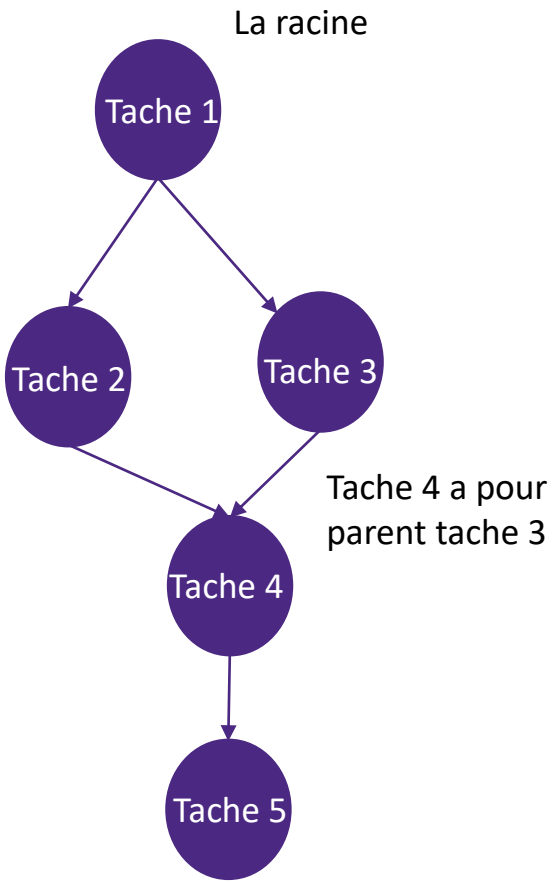
Un arbre est un **graphe connexe acyclique**

- En pratique on utilise presque toujours des arbres **orientés**

- S'il y a un arc de A vers B :
  - *B est un **fils** de A*
  - *A est un **parent** de B*

- Un sommet peut être désigné comme **racine**
  - *Celui qui en partant de lui permet d'accéder à tout les sommets*
  - *Généralement, **c'est l'état/ la position de depart en RL***

- La **profondeur** d'un sommet est sa **distance** (le nombre d'arêtes à parcourir) **depuis la racine**.
  - *Contre un adversaire : état de profondeur paire : c'est à vous de jouer.*
  - *Contre un adversaire : état de profondeur impaire : c'est à l'adversaire de jouer*



REPRÉSENTER LES GRAPHES

SCALIAN

Il existe deux manière de représenter les graphes en informatique :

La représentation sous forme de **matrice** : par **matrice d'adjacence**

La représentation sous forme de **liste** : par **liste d'adjacence.**

- **Tout graphe ou arbre peut être représenté par liste ou matrice d'adjacence**
- **Suivant la situation, l'une ou l'autre des implementation sera plus rapide.**

Motivation supplémentaire : il n'y a pas de bibliothèque de graphes immediate en python.

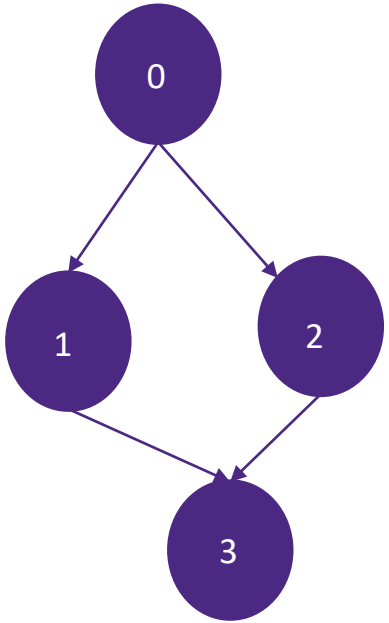
- **Bibliothèque networkx mais plus adaptée à modéliser des réseaux**

LA REPRESENTATION MATRICIELLE.

SCALIAN

Pour un graphe à N sommets, on le représente par une matrice N*N.

Une valeure différente de 0 dans la case (i,j) indique la presence d'une arête entre I et J (non orienté) ou d'un arc de I vers J (orienté).



Représentation par matrice d'adjacence

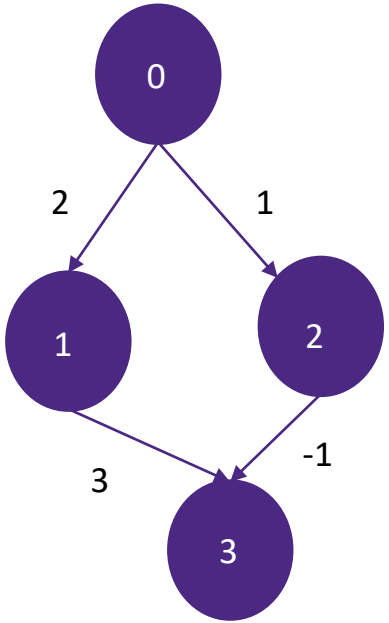
0	1	1	0
0	0	0	1
0	0	0	1
0	0	0	0

LA REPRESENTATION MATRICIELLE.

SCALIAN

Pour un graphe à N sommets, on le représente par une matrice N*N.

Lorsqu'une distance est associée aux arrêtes (par exemple la distance entre deux villes, …), celle-ci peut être codée par la valeur de la case de la matrice



Représentation par matrice d'adjacence

0	2	1	0
0	0	0	3
0	0	0	-1
0	0	0	0

LA REPRESENTATION MATRICIELLE.

SCALIAN

Pour un graphe à N sommets, on le représente par une matrice N*N.

Lorsqu'une distance est associée aux arrêtes (par exemple la distance entre deux villes, …), celle-ci peut être codée par la valeur de la case de la matrice

- **Trouver si une arrête existe** ou si un sommet est fils d'un autre **est instantané**
- Peut demander **une taille importante si beaucoup de sommets**
  - Matrice sparse si beaucoup de sommets et peu d'arêtes
- **Ajouter ou enlever des sommets peut être couteux.**
- **Lister les fils d'un sommet peut être couteux.**

Représentation par matrice d'adjacence

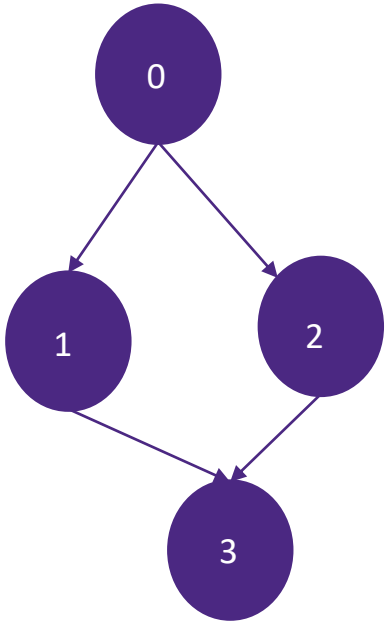
0	2	1	0
0	0	0	3
0	0	0	-1
0	0	0	0

LA REPRESENTATION LISTE D'ADJACENCE

SCALIAN

Pour un graphe à N sommets, chaque sommet possède la liste de ses fils/les sommets auquel il est connecté.

Si un sommet A possède la liste [C,D], cela veut dire qu'il y a un arc/arête entre A et C ainsi qu'entre A et D.



Représentation par liste d'adjacence

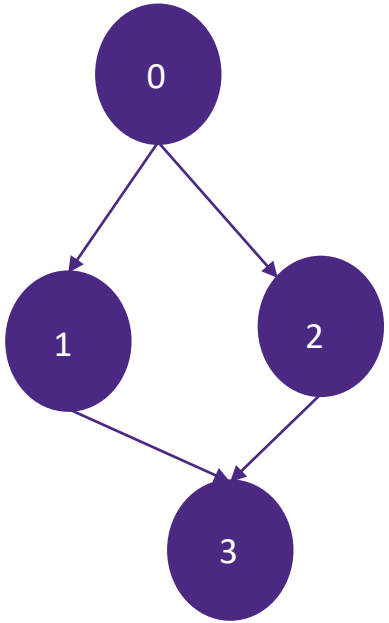
0: [1,2]
1: [3]
2: [3]
3: []

LA REPRESENTATION LISTE D'ADJACENCE

SCALIAN

Pour un graphe à N sommets, chaque sommet possède la liste de ses fils/les sommets auquel il est connecté.

Si un sommet A possède la liste [C,D], cela veut dire qu'il y a un arc/arête entre A et C ainsi qu'entre A et D.



Astuce : en python, utiliser un dictionnaire :

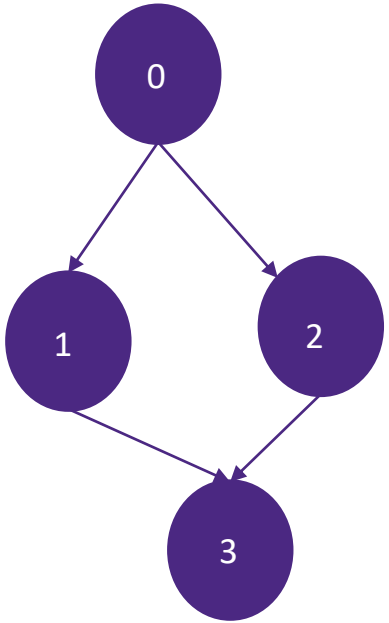
g = {"0": ["1", "2"], "1": ["3"], "2": ["3"], "3": [] }

g[« non du sommet »] nous retourne la liste des noms des sommets auquel il est connecté.

LA REPRESENTATION PAR LISTE D'ADJACENCE

Pour un graphe à N sommets, chaque sommet possède la liste de ses fils/les sommets auquel il est connecté.

Si un sommet A possède la liste [C,D], cela veut dire qu'il y a un arc/arête entre A et C ainsi qu'entre A et D.



- Ajout et suppression de sommet rapide.
- Accès à la liste des fils immédiat avec le dictionnaire en python.
- Efficace en mémoire
- Trouver si deux sommets A et B sont connectés **peut être long** (demande de parcourir une à une toutes les arêtes partant de A)
- **Souvent mieux adapté au RL**
  - **Elagage facile**
  - **En RL, degré << nombre d'état**
  - **Souvent, l'opération que l'on désire est la liste des fils**

LES PARCOURS DE GRAPHES

LES PARCOURS DE GRAPHES

SCALIAN

Bien parcourir un graphe est critique, notamment en RL.

Comment trouver la sortie **la plus proche de l'entrée** ? **Le chemin qui minimise l'essence ?**

Comment **parcourir un labyrinthe de façon "réelle" ?** (sans se teleporter à l'autre bout du labyrinthe pour explorer la sale suivante). **Sans tourner en rond ? En garantissant d'avoir exploré toutes les salles ?**

Comment trouver **"rapidement" la sortie ?**

Comment trouver le meilleur coup à jouer quand on a un adversaire ? (quand on ne contrôle pas quel déplacement il va faire)

LE PARCOUR EN PROFONDEUR

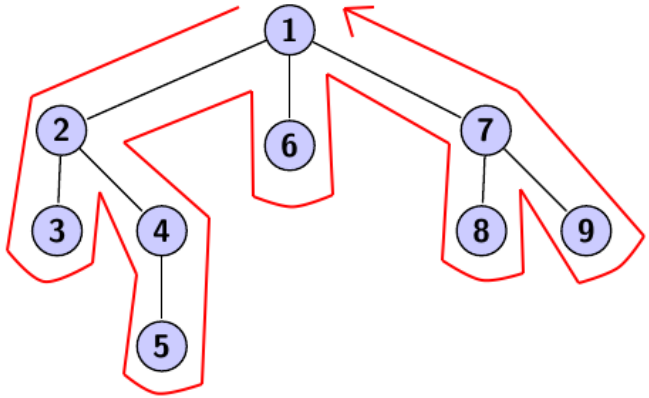
SCALIAN

Algorithme de parcours de **graphes (et pas seulement d'arbres)**

Une definition du parcours en profondeur est **"la méthode intuitive pour trouver la sortie sans tourner en rond".**

Illustration du fonctionnement

Principe : on poursuit un chemin, et dès que l'on rencontre un cul-de-sac, on rebrousse chemin jusqu'au dernier endroit où on a laissé un chemin encore non-exploré, et ainsi de suite.



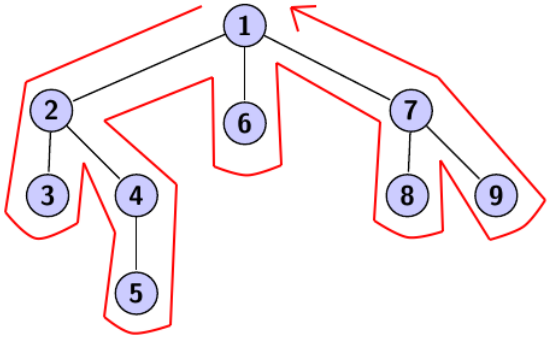
LE PARCOUR EN PROFONDEUR

SCALIAN

Algorithme intuitivement implementable de manière recursive.

Mais c'est généralement plus efficace (et plus simple) de l'implémenter de manière iterative.

```
A_visiter <- [depart]
Déjà_visite <- []
Tant qu'il reste des sommets dans a_visiter:
        sommet_actuel <- dernier élement de a_visiter
        ajouter à la fin de a_visiter les fils de sommet_actuel jamais encore
                rencontrés
        enlever sommet_actuel de a_visiter
```



LE PARCOUR EN PROFONDEUR

SCALIAN

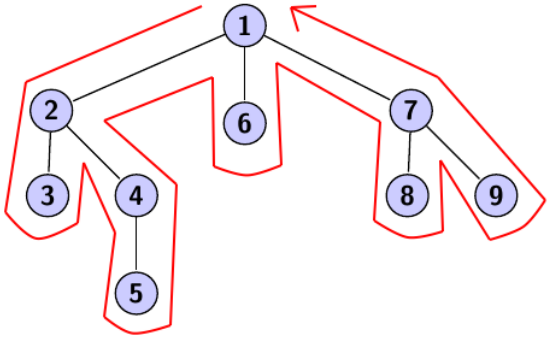
Le parcours en profondeur :

- Adapté pour trouver la sortie du labyrinthe

- Ne parcours pas plusieurs fois le même sommet
- ***Fonctionnera même si le graphe est cyclique***

- **Ne garantit pas de trouver la sortie "rapidement",** sans explorer une majorité des sommets

- S'il y a plusieurs sortie, **ne garantit pas de retourner la plus proche de l'entrée**
- *Pas adapté à tout ce qui est problème de minimisation de distance parcourue.*

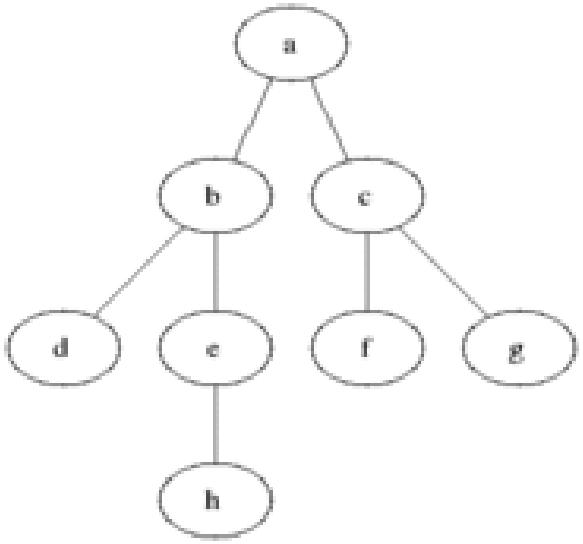


LE PARCOUR EN PROFONDEUR

SCALIAN

Algorithme de parcours de **graphes (et pas seulement d'arbres)**

Idée : parcourir d'abord tout les sommets de profondeur 1, puis ceux de profondeur 2, etc.



LE PARCOUR EN HAUTEUR

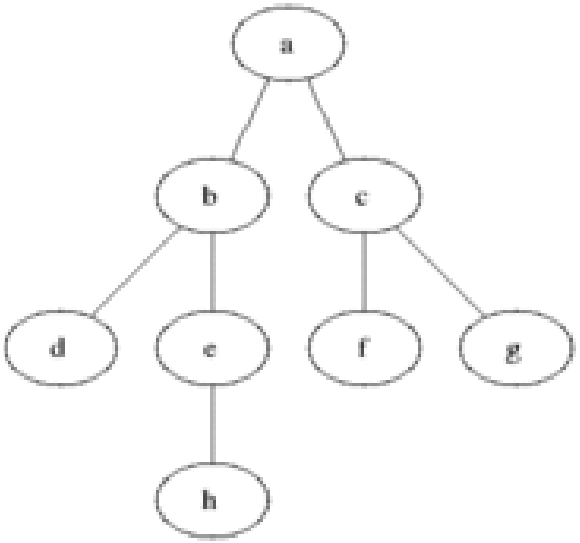
SCALIAN

Algorithme de parcours de **graphes (et pas seulement d'arbres)**

Idée : parcourir d'abord tout les sommets de profondeur 1, puis ceux de profondeur 2, etc.

Astuce : Avec l'implementation précédente du parcours en profondeur, ne demande de changer qu'une ligne de code !

```
A_visiter <- [depart]
Déjà_visite <- []
Tant qu'il reste des sommets dans a_visiter:
        sommet_actuel <- premier élement de a_visiter
        ajouter à la fin de a_visiter les fils de sommet_actuel jamais encore
                rencontrés
        enlever sommet_actuel de a_visiter
```



LE PARCOUR EN HAUTEUR

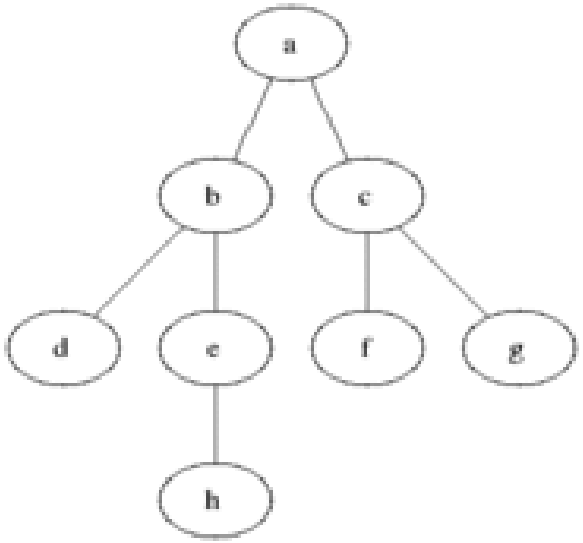
SCALIAN

Algorithme de parcours de **graphes (et pas seulement d'arbres)**

Idée : parcourir d'abord tout les sommets de profondeur 1, puis ceux de profondeur 2, etc.

En règle générale, adapté aux problemes de minimisation.

- Sortie la plus proche
- Gagner une partie le plus rapidement possible



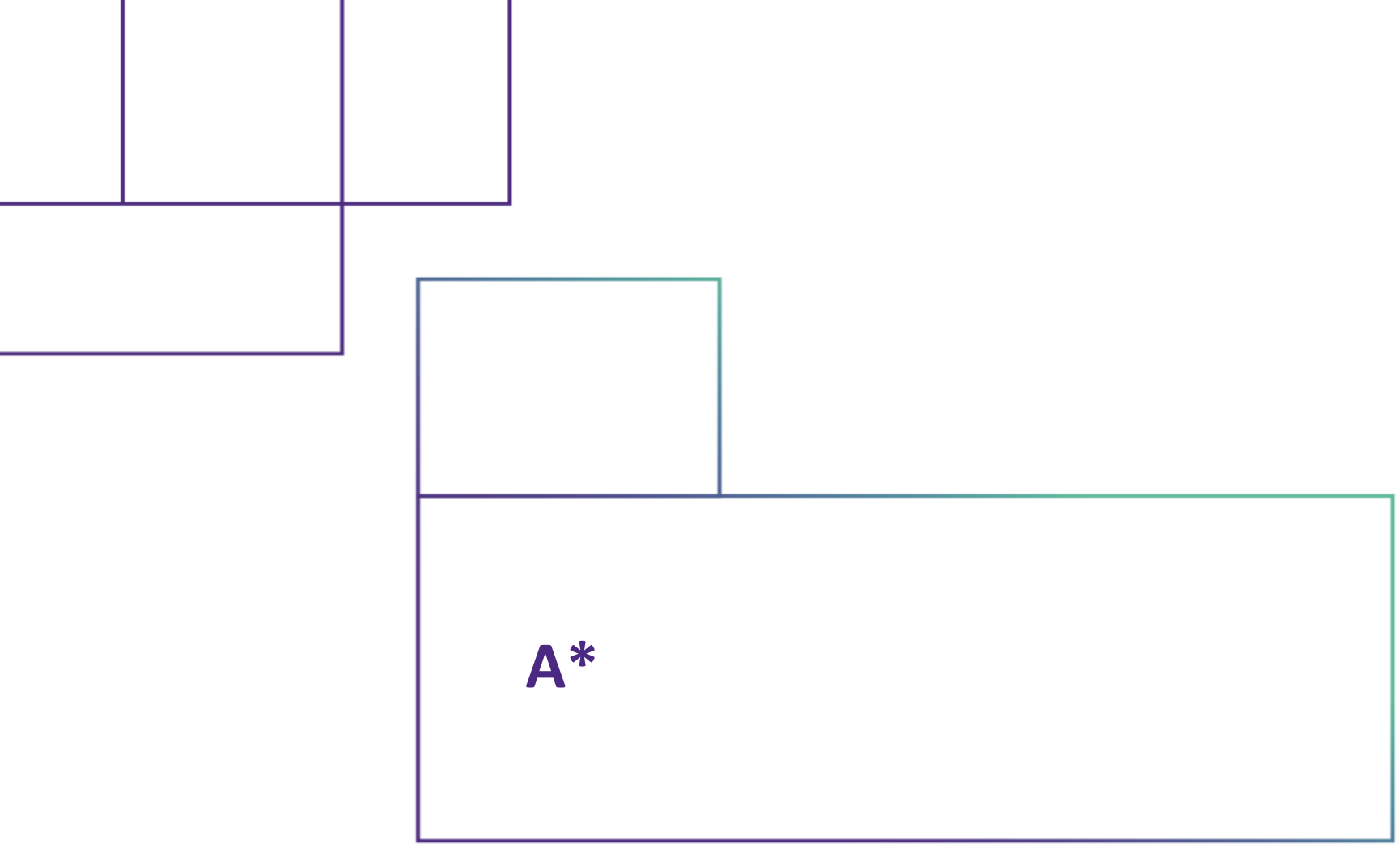
Mais pas toujours:

- **Peut ne pas fonctionner lorsque le poids des arrêtes est différent** !

LES PARCOURS DE GRAPHES

SCALIAN

Graphes et parcours : à vos notebooks !


A*

L'ALGORITHME A*

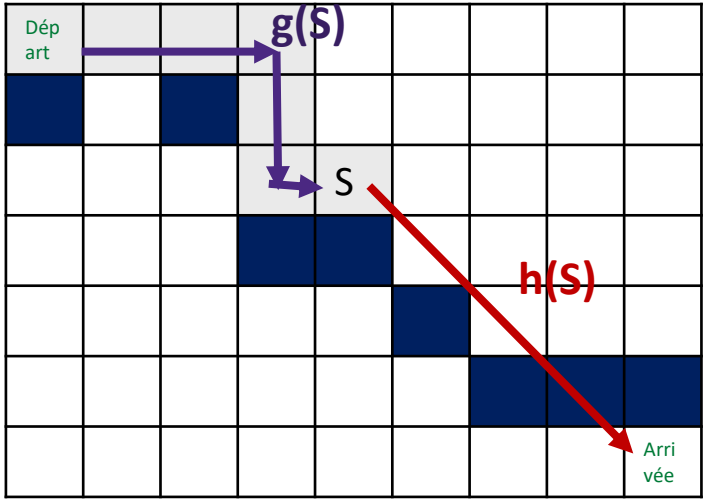
SCALIAN

Algorithme A* (A-star) : évolution du parcours en hauteur.

Idée : quand vous rencontrez une intersection en explorant un labyrinthe, ou cherchez à aller à Ynov, et savez où est la sortie, vous allez d'abord **essayer les chemins qui ont l'air d'aller dans la bonne direction !**

Principe : visiter à chaque step le sommet s qui minimise f(s), où f est **:**

$$f(s) = g(s) + h(s)$$



Avec:

- **G(s) : distance du depart à s (connue)**
- **h(s) : distance de s à l'arrivée (estimée** par un **heuristique)**

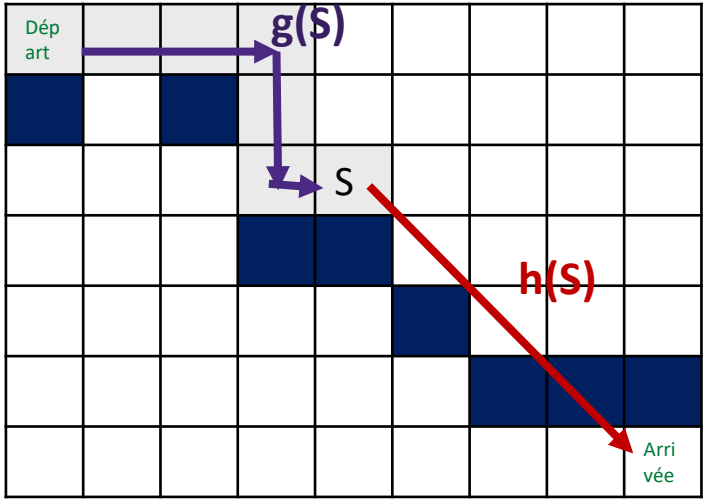
Exemple : h(s) : distance à vol d'oiseau entre s et l'arrive

L'ALGORITHME A*

SCALIAN

Algorithme A* (A-star) : en action

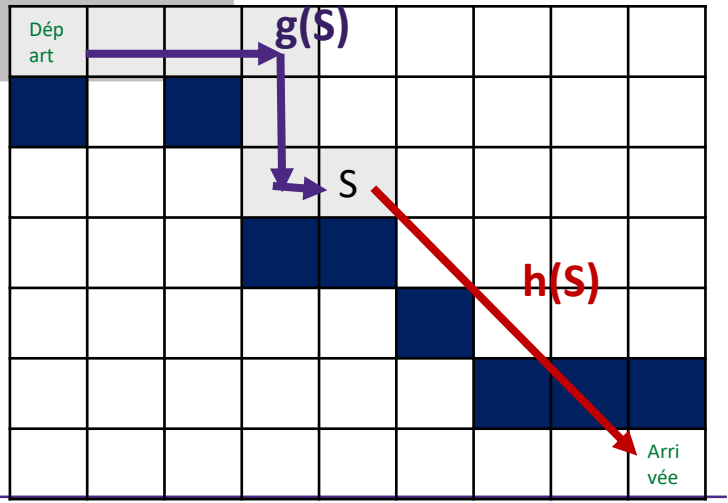
- **Dépend de la pertinence de l'heuristique**
- **Quasi toujours plus rapide que le parcours en hauteur**
- **Capable de gérer des arbres avec des distances différentes sur les arêtes**
- **Le plus utilisé dans les jeux vidéos**
- **Il a été prouvé qu'il est optimal à heuristique égal**
- *Il ne peut pas y avoir d'algorithme qui puisse garantir de trouver plus rapidement sans avoir de meilleur heuristique.*
- **Il peut être long si votre heuristique l'est !**



L'ALGORITHME A*

SCALIAN

A_visiter <- [depart]
Déjà_visite <- []
Tant qu'il reste des sommets dans a_visiter:
    sommet_actuel <- élément de a_visiter qui minimise f
    ajouter **à la fin** de a_visiter les fils de sommet_actuel jamais encore **rencontrés**
    pour chaque fils c non **visité**
        mettre à jour g(c)
        calculer h(c) et f(c) de chaque fils
    enlever sommet_actuel de a_visiter



LES PARCOURS DE GRAPHES

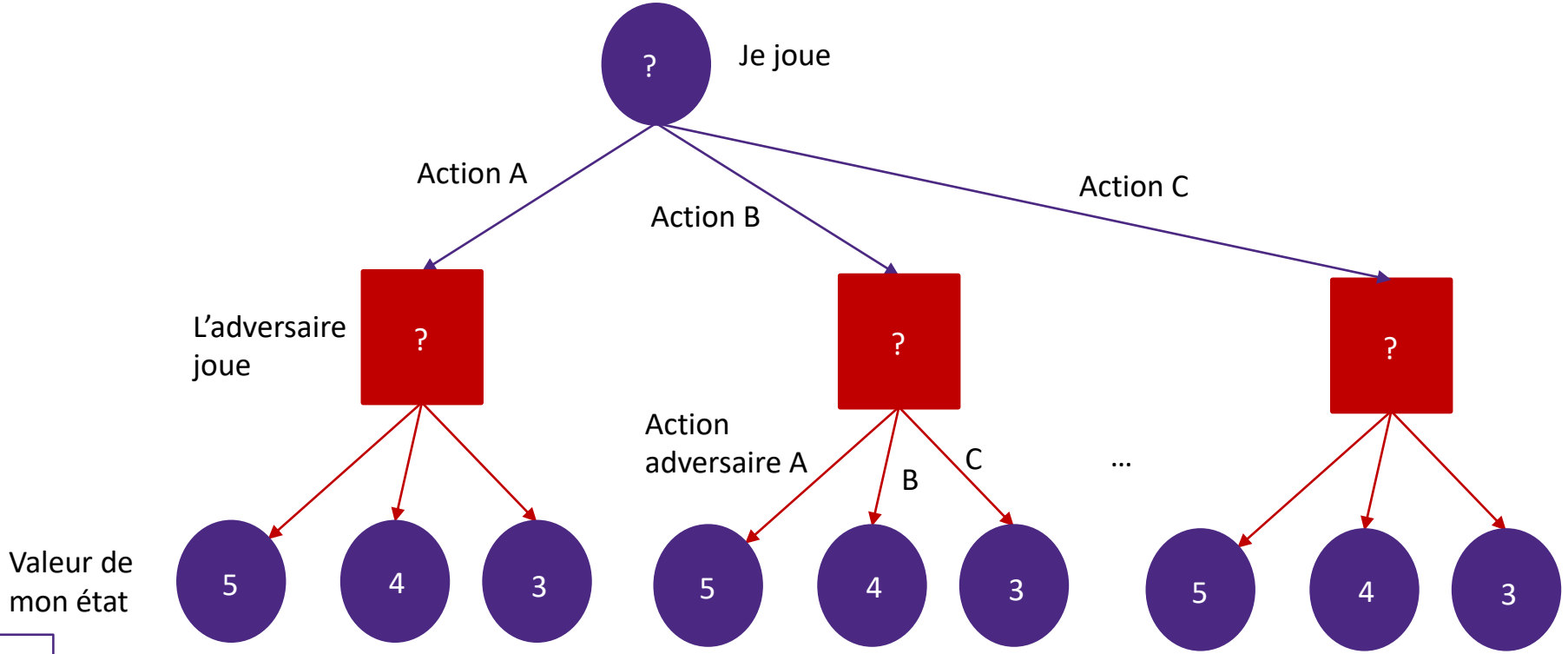
SCALIAN

à vos notebooks !

MINIMAX

L'ALGORITHME MINIMAX

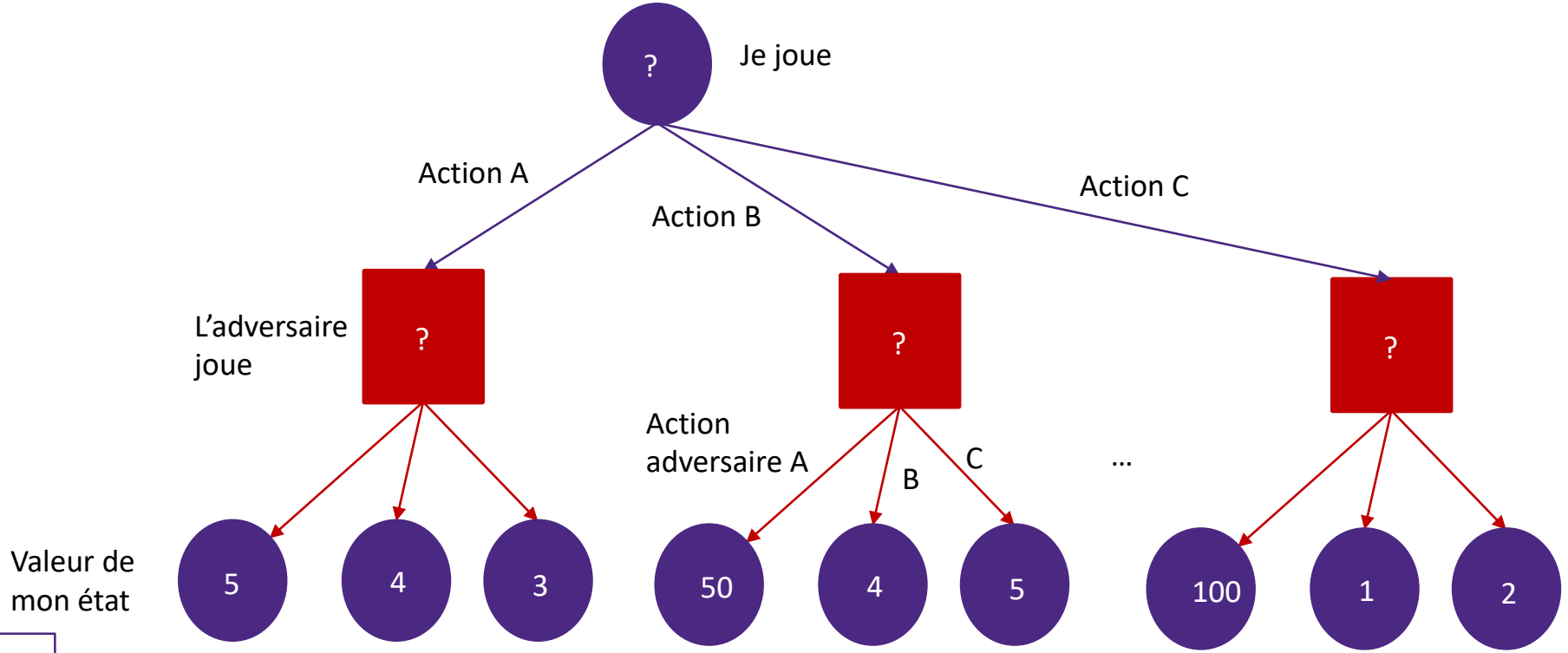
Imaginons que je joue aux échecs ou au morpion
Je ne sais pas ce que l'adversaire va jouer.
Comment choisir que action effectuer, et quelle valeur à donner à chaque action/état ?



L'ALGORITHME MINIMAX

SCALIAN

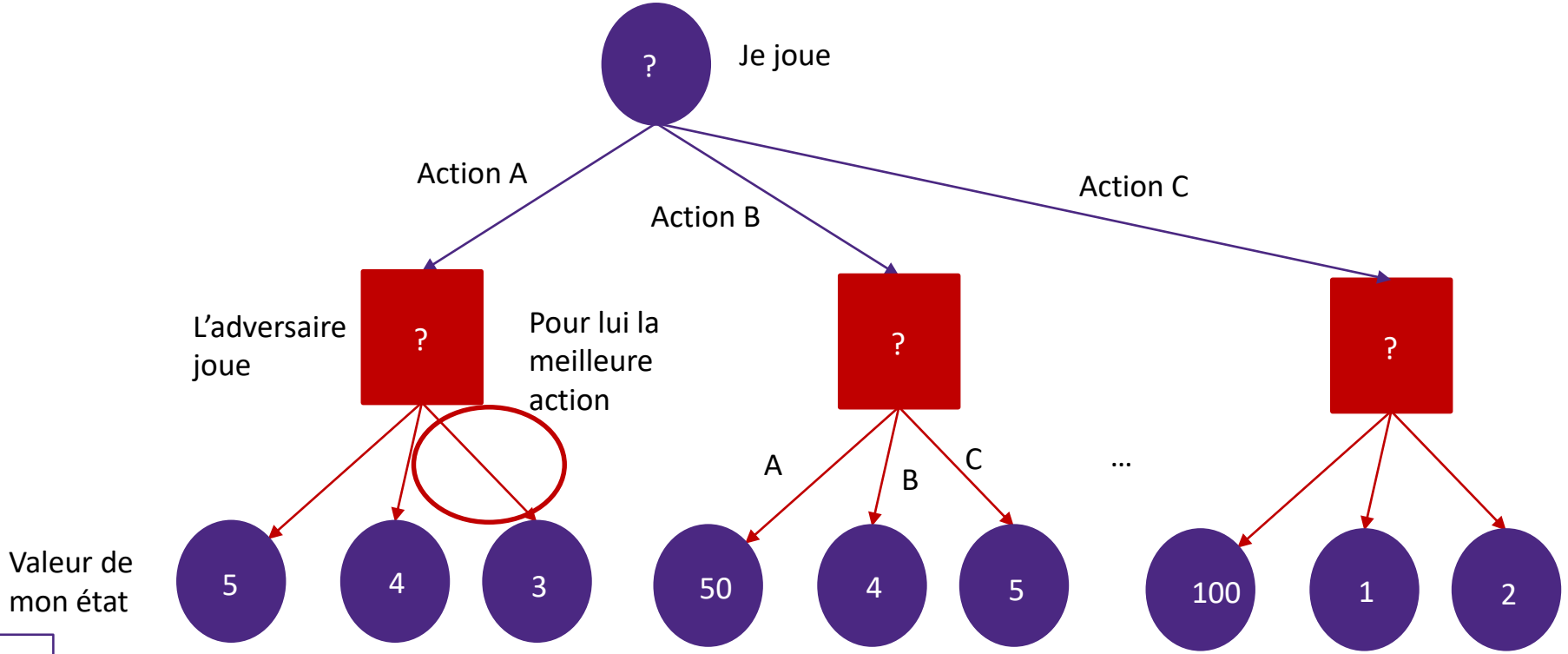
Imaginons que je joue aux échecs ou au morpion
Je ne sais pas ce que l'adversaire va jouer.
Comment choisir que action effectuer, et quelle valeur à donner à chaque action/état ?
Idée : supposer que l'adversaire **joue le mieux possible**.



L'ALGORITHME MINIMAX

SCALIAN

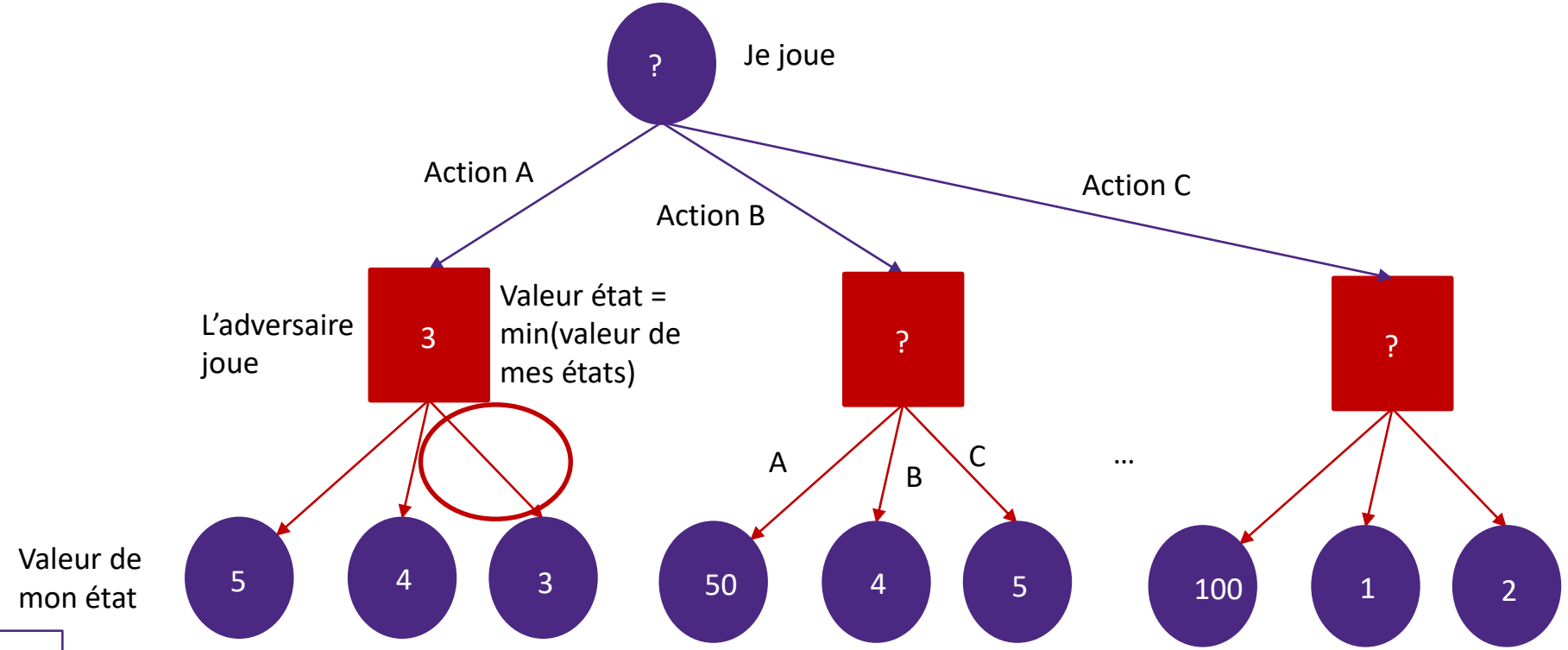
Imaginons que je joue aux échecs ou au morpion
Je ne sais pas ce que l'adversaire va jouer.
Comment choisir que action effectuer, et quelle valeur à donner à chaque action/état ?
Idée : supposer que l'adversaire **joue le mieux possible**.



L'ALGORITHME MINIMAX

SCALIAN

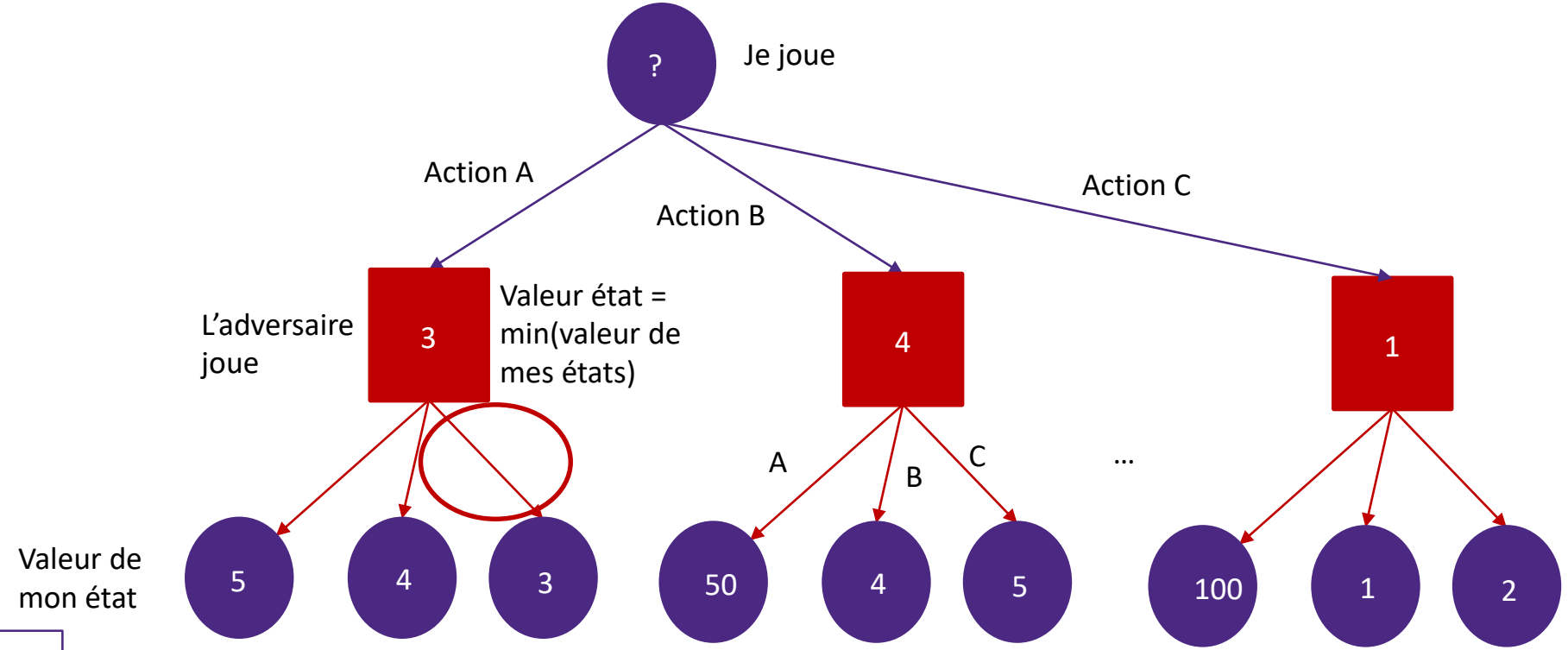
Imaginons que je joue aux échecs ou au morpion
Je ne sais pas ce que l'adversaire va jouer.
Comment choisir que action effectuer, et quelle valeur à donner à chaque action/état ?
Idée : supposer que l'adversaire **joue le mieux possible**.



L'ALGORITHME MINIMAX

SCALIAN

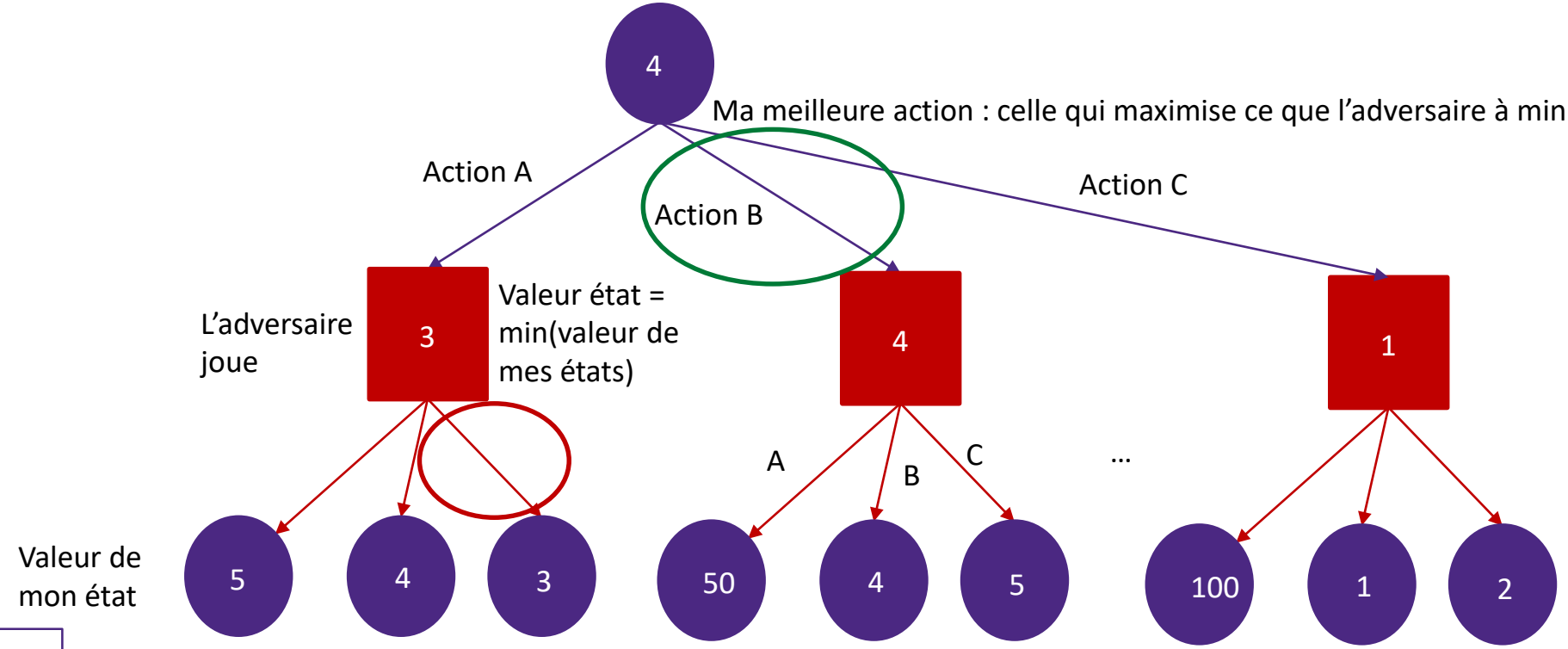
Imaginons que je joue aux échecs ou au morpion
Je ne sais pas ce que l'adversaire va jouer.
Comment choisir que action effectuer, et quelle valeur à donner à chaque action/état ?
Idée : supposer que l'adversaire **joue le mieux possible**.



L'ALGORITHME MINIMAX

SCALIAN

Imaginons que je joue aux échecs ou au morpion
Je ne sais pas ce que l'adversaire va jouer.
Comment choisir que action effectuer, et quelle valeur à donner à chaque action/état ?
Idée : supposer que l'adversaire **joue le mieux possible**.

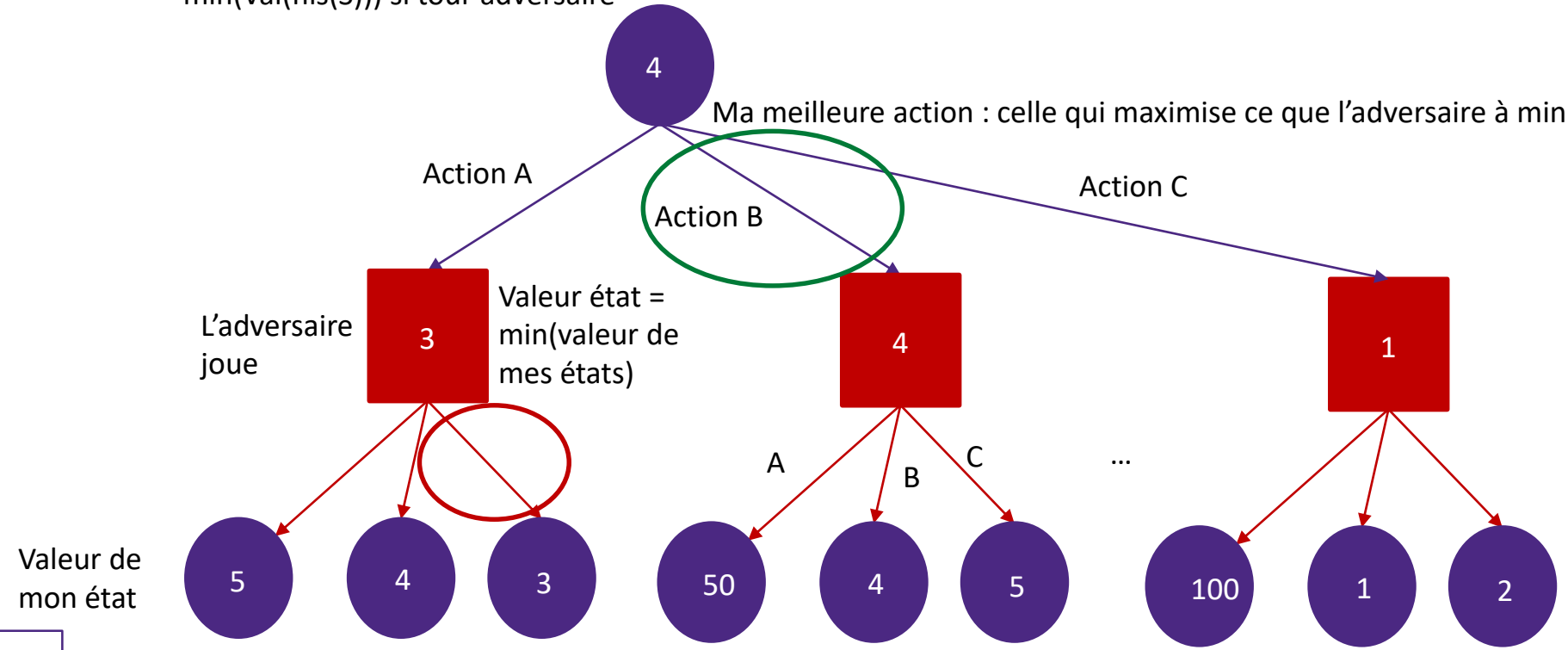


L'ALGORITHME MINIMAX

L'algorithme minimax cherche à **minimiser la perte maximum (maximiser le gain minimal, dans le pire des cas)**

Définition pratique :

Val(S) = max(Val(fils(S))) si tour joueur
min(Val(fils(S))) si tour adversaire



ALGORITHME MINIMAX

SCALIAN

Mais le minimax à un petit problème :

Morpion : environ 300 000 parties possibles

Echecs : estimées à 10^{120} parties possibles

Go : estimées à 10^{600} parties possibles

Nombre d'atome de l'univers : 10^{80}

Comment calculer la valeurs des états fils, sachant que l'on devra s'arrêter avant d'atteindre une feuille ?

- **On définit une profondeur maximale**
- **On utilise un heuristique pour estimer la valeur lorsque l'on atteint la profondeur maximale.**

ALGORITHME MINIMAX

PSEUDO CODE

```
Fonction minmaxrec(state, joueur, depth):

        si est_terminé(state) ou depth == 0:
                valeur_estimée = heuristique(state)
                returner (no_action, valeur_estimée)


        si joueur = ai:
                valeur_etat_actuel = -inf
                prochain_joueur = adversary
        sinon:
                 valeur_etat_actuel = +inf
                prochain_joueur = ai

        pour chaque action possible a dans state:
                new_state = appliquer(a, state)
                action_fils, valeur_estimee_fils = minmaxrec(new_state, prochain_joueur, depth-1)
                si joueur == ai et valeur_etat_actuel < valeur_estimee_fils:
                        valeur_etat_actuel = valeur_estimee_fils
                        best_action = a
                si joueur == adversary et valeur_etat_actuel > valeur_estimee_fils:
                        valeur_etat_actuel = valeur_estimee_fils
                        best_action = a


        retourner best_action, valeur_etat_actuel
```

ALGORITHME MINIMAX

PSEUDO CODE

```
Fonction minmaxrec(state, joueur, depth):

        si est_terminé(state) ou depth == 0:
                valeur_estimée = heuristique(state)
                returner (no_action, valeur_estimée)

        si joueur = ai:
                valeur_etat_actuel = -inf
                prochain_joueur = adversary
        sinon:
                 valeur_etat_actuel = +inf
                prochain_joueur = ai

        pour chaque action possible a dans state:
                new_state = appliquer(a, state)
                action_fils, valeur_estimee_fils = minmaxrec(new_state, prochain_joueur, depth-1)
                si joueur == ai et valeur_etat_actuel < valeur_estimee_fils:
                        valeur_etat_actuel = valeur_estimee_fils
                        best_action = a
                si joueur == adversary et valeur_etat_actuel > valeur_estimee_fils:
                        valeur_etat_actuel = valeur_estimee_fils
                        best_action = a

        retourner best_action, valeur_etat_actuel
```

Condition d'arret

Initialise la recherche du max/min suivant le cas

Appel recursif

Recherche du max si ai

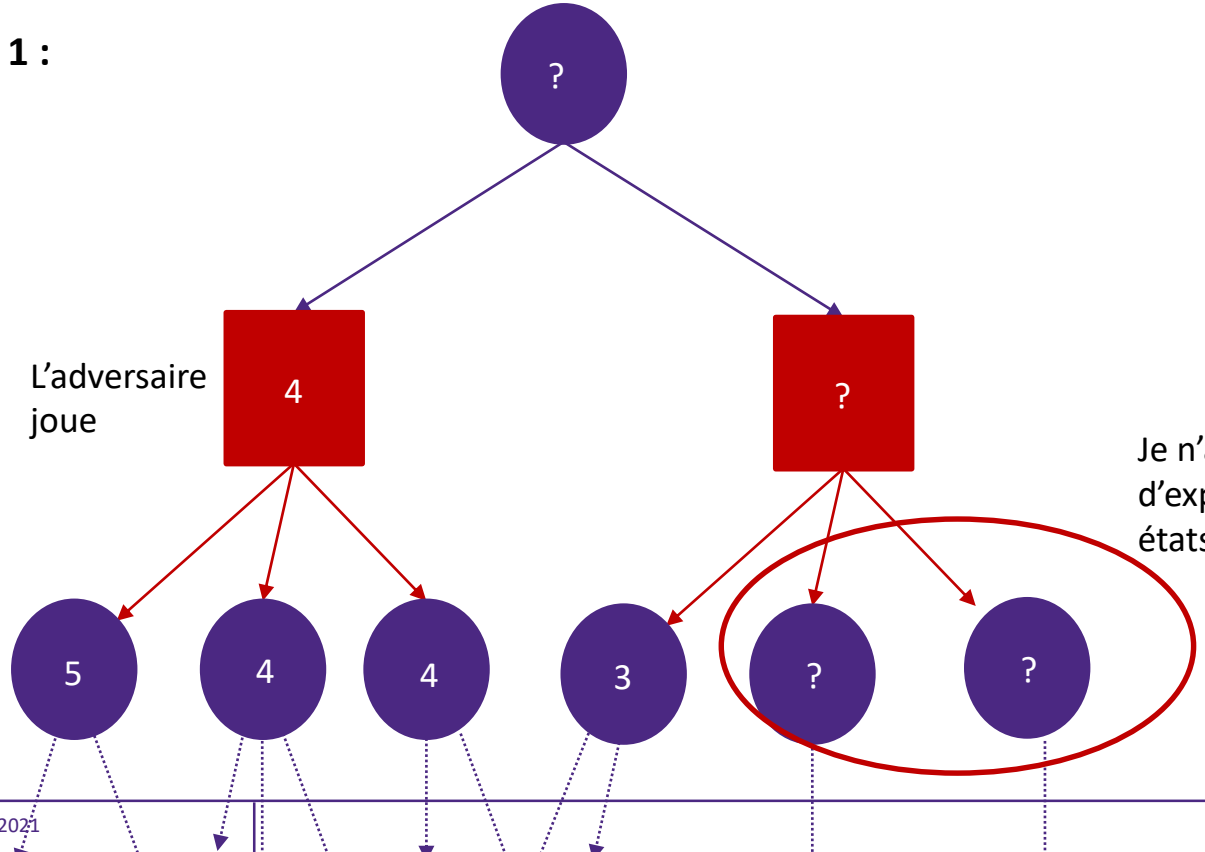
Recherche du min si adversaire

ALGORITHME MINIMAX

SCALIAN

L'élagage alpha-beta (alpha-beta pruning) : deux case de figures sont élagués

Cas 1 :

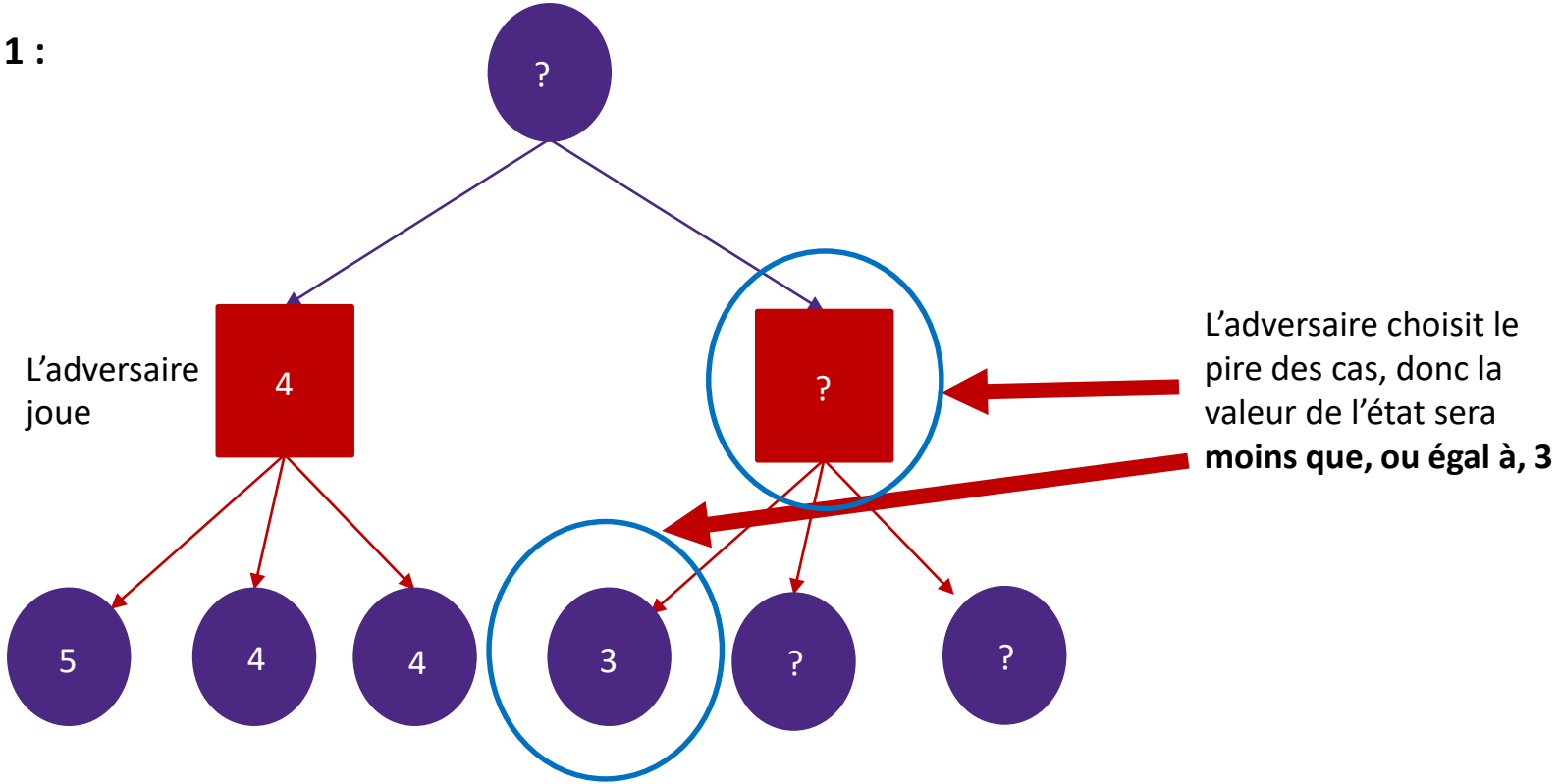


ALGORITHME MINIMAX

SCALIAN

L'élagage alpha-beta (alpha-beta pruning) : deux case de figures sont élagués

Cas 1 :

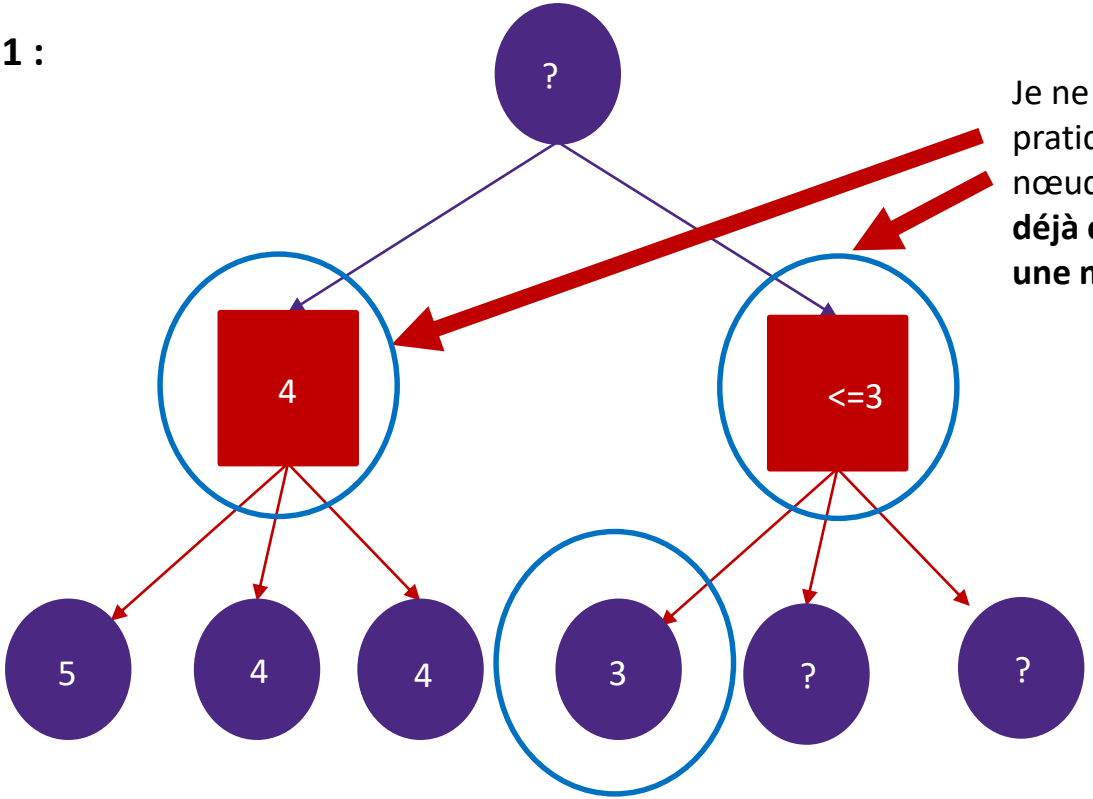


ALGORITHME MINIMAX

SCALIAN

L'élagage alpha-beta (alpha-beta pruning) : deux case de figures sont élagués

Cas 1 :



Je ne choisirais donc en pratique jamais ce nœud, comme un **nœud déjà exploré me garantit une meilleure valeur**

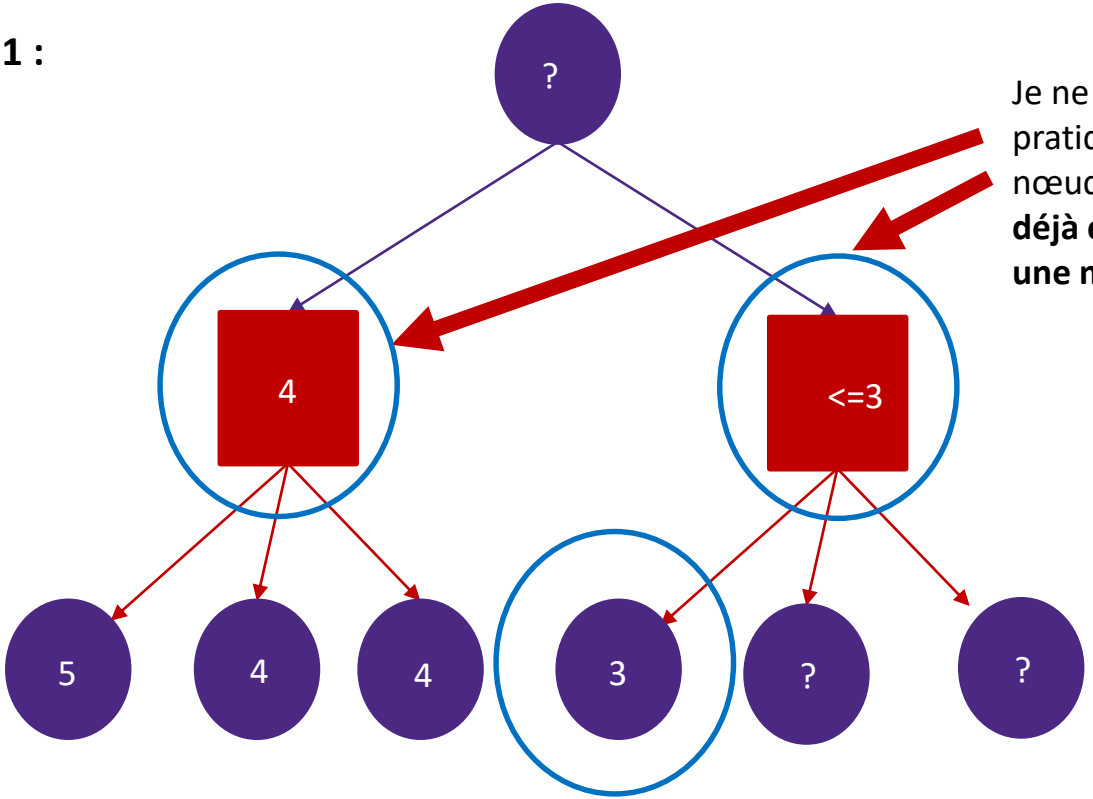
Valeur de mon état

ALGORITHME MINIMAX

SCALIAN

L'élagage alpha-beta (alpha-beta pruning) : deux case de figures sont élagués

Cas 1 :



Je ne choisirais donc en pratique jamais ce nœud, comme un **nœud déjà exploré me garantit une meilleure valeur**

Valeur de mon état

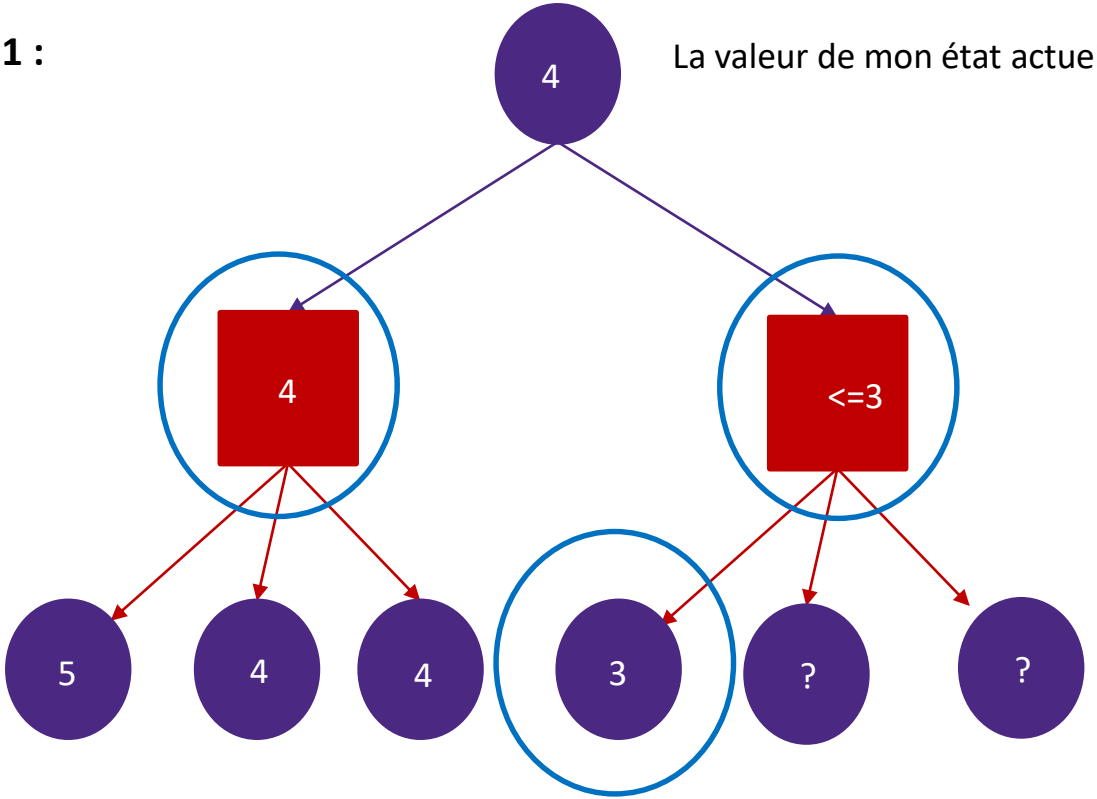
ALGORITHME MINIMAX

SCALIAN

L'élagage alpha-beta (alpha-beta pruning) : deux case de figures sont élagués

Cas 1 :

La valeur de mon état actuel est donc 4



Valeur de mon état

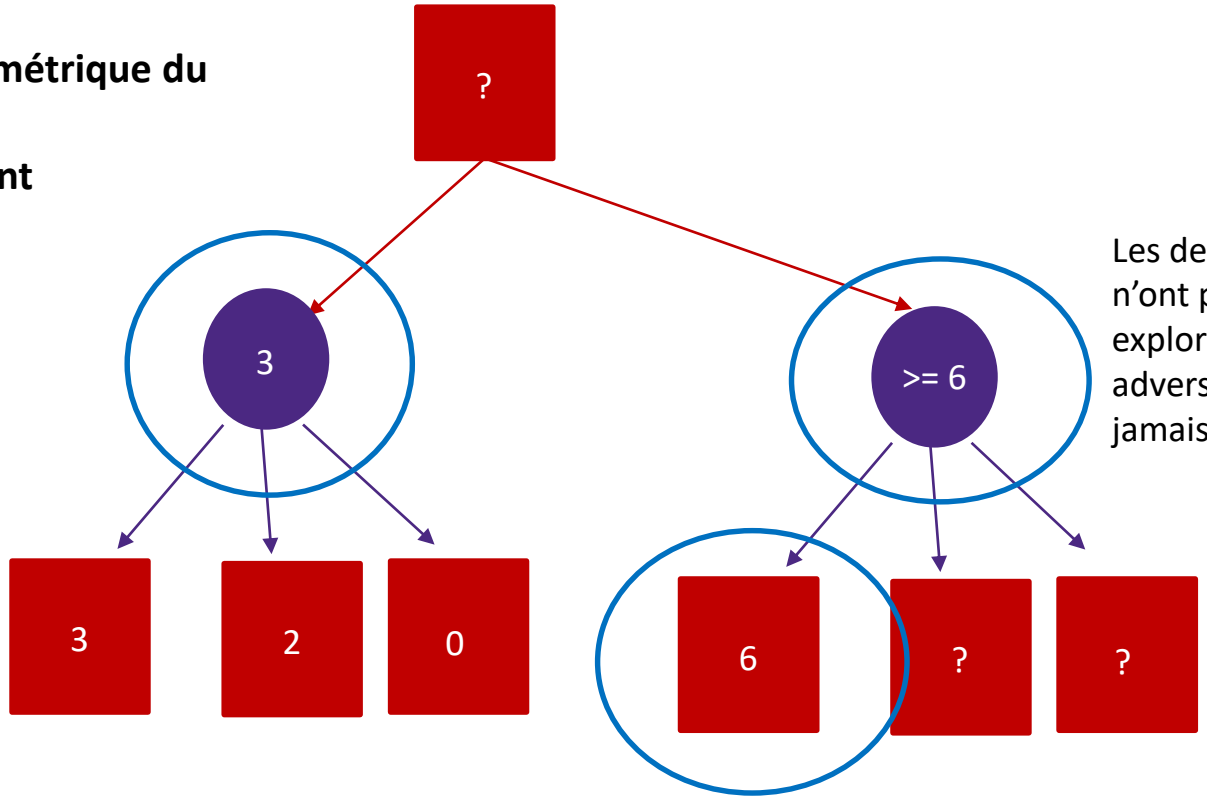
ALGORITHME MINIMAX

SCALIAN

L'élagage alpha-beta (alpha-beta pruning) : deux case de figures sont élagués

Cas 2 : le symétrique du

cas précédent



Les deux derniers fils n'ont pas besoin d'être explorés, comme mon adversaire ne choisira jamais ce nœud !

CONCLUSION

SCALIAN

Nous avons vus les algorithmes de parcours de graphes les plus utilisés en RL: parcours hauteur, sa variante A*, parcours en profondeur, algorithme minmax.

Pour être performant, ils demandent l'utilisation d'heuristiques bien choisis.

- **Connaissance métier**

- **Les heuristiques sont des algorithmes pour lesquels on sait estimer si leur réponse est bonne, sans savoir donner la réponse optimale qu'ils auraient pu donner.**
- *Cette formulation de problèmes ne vous dit rien ?*
- *Parmi les algorithmes les plus récents, on retrouve des occurrences où ces heuristiques sont appris par renforcement (par exemple Alphago au moyen de deep learning).*