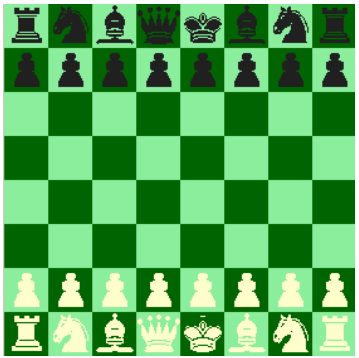
Jeux entre deux adversaires

- Noms des joueurs : Max vs. Min
  - Max est le premier à jouer (notre joueur)
  - Min est son adversaire

- On va interpréter le résultat d'une partie comme la **distribution d'une récompense au joueur Max**
  - peut voir cette récompense comme le résultat d'un pari
  - Max souhaite maximiser sa récompense
  - Min souhaite minimiser la récompense de Max (Min va recevoir l'opposé de cette récompense)

Min



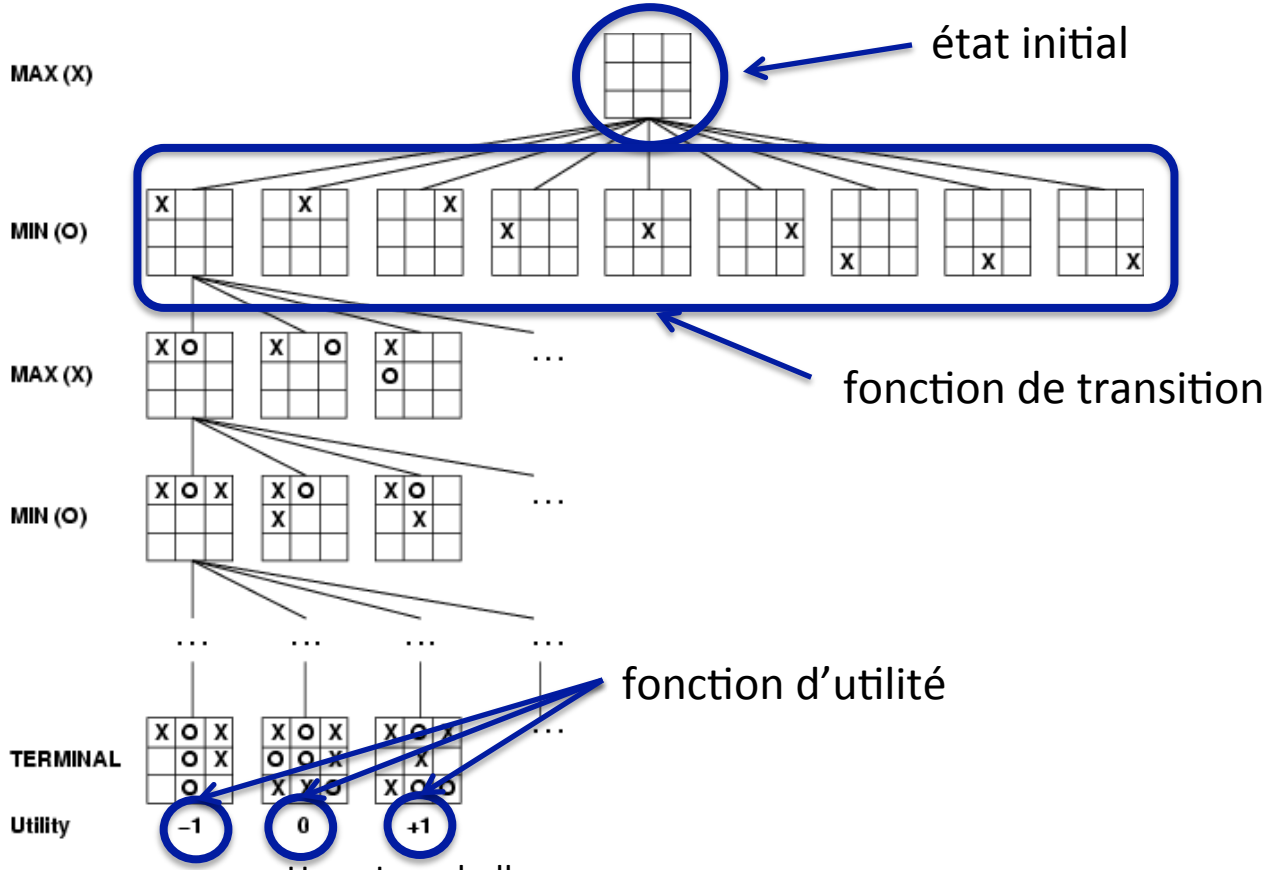
Max

Arbre de recherche

- Comme pour les problèmes que A* peut résoudre, on commence par déterminer la structure de notre espace de recherche

- Un problème de jeu peut être vu comme un problème de recherche dans un arbre :
  - Un **nœud (état) initial :** configuration initiale du jeu
  - Une **fonction de transition** :
    - » retournant un **ensemble de paires (action, nœud successeur)**
      - – action possible (légale)
      - – nœud (état) résultant de l'exécution de cette action
  - Un **test de terminaison**
    - » indique si le jeu est terminé
  - Une **fonction d'utilité** pour les états finaux
    (c'est la récompense reçue par le joueur Max)

Arbre de recherche tic-tac-toe



Algorithme minimax

- **Idée**: à chaque tour, on suppose que l'action la plus profitable pour le joueur Max ou le joueur Min est prise, c.-à-d. la plus grande **valeur minimax**

$$\textsc{valeur-minimax}(n) = \begin{cases} \textsc{utilité}(n) & \text{Si } n \text{ est un nœud terminal} \\ \max_{n' \text{ successeur de } n} \textsc{valeur-minimax}(n') & \text{Si } n \text{ est un nœud Max} \\ \min_{n' \text{ successeur de } n} \textsc{valeur-minimax}(n') & \text{Si } n \text{ est un nœud Min} \end{cases}$$

- Ces équations donnent un **programme récursif** pour calculer les valeurs pour tous les nœuds dans l'arbre de recherche

Algorithme minimax

Algorithme MINIMAX(*noeudInitial*)

1. retourne l'action choisie par tourMax(*noeudInitial*)

Algorithme TOUR-MAX(*n*)

1. si *n* correspond à une fin de partie, alors retourner UTILITÉ(*n*)
2. *u* = -∞, *a* = void
3. pour chaque paire (*a'*,*n'*) donnée par TRANSITION(*n*)
    4. si l'utilité de TOUR-MIN(*n'*) > *u* alors affecter *a* = *a'*, *u* = utilité de TOUR-MIN(*n'*)
5. retourne l'utilité *u* et l'action *a* // *l'action ne sert qu'à la fin (la racine)*

Algorithme TOUR-MIN(*n*)

1. si *n* correspond à une fin de partie, alors retourner UTILITÉ(*n*)
2. *u* = ∞, *a* = void
3. pour chaque paire (*a'*,*n'*) donnée par TRANSITION(*n*)
    4. si l'utilité de TOUR-MAX(*n'*) < *u* alors affecter *a* = *a'*, *u* = utilité de TOUR-MAX(*n'*)
5 retourne l'utilité *u* et l'action *a*

Propriétés de minimax

- Complet (retourne une solution) ?
  - Oui (si l'arbre est fini)
- Optimal (retourne la meilleure solution) ?
  - Oui **(contre un adversaire qui joue de façon optimale)**
- Complexité en temps ?
  - $O(b^m)$:
    - » b : le nombre maximum de coups (actions) légaux à chaque étape
    - » m : nombre maximum de coups dans un jeu (profondeur maximale de l'arbre)
- Complexité en espace mémoire ?
  - O(bm), puisque **recherche en profondeur**
- Pour le jeu d'échec : b ≈ 35 et m ≈100 pour une partie « raisonnable »
  - il n'est pas réaliste d'espérer trouver une solution exacte en un temps raisonnable
- Existe-t-il des chemins dans l'arbre qui sont explorés inutilement ?