Programmation d'une Intelligence Artificielle : arbres ou apprentissage automatique

Arbres : Arbres Couvrants et Algorithme MinMax.

Paul Mangold

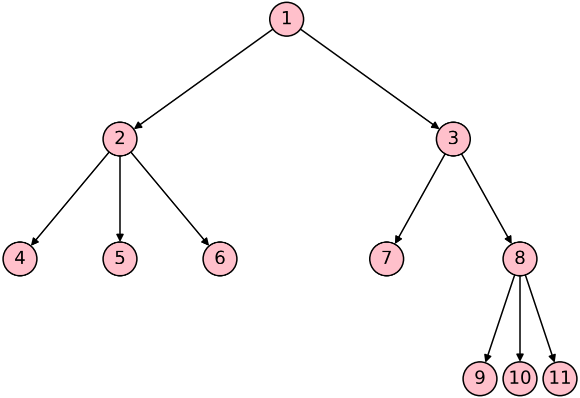
L3 MIASHS

5 Février 2021

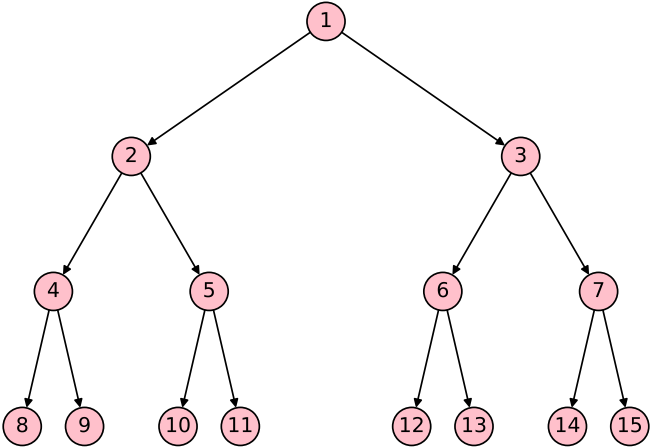
Arbres : définitions et parcours

Un **arbre** est un graphe orienté, acyclique et connexe, avec :

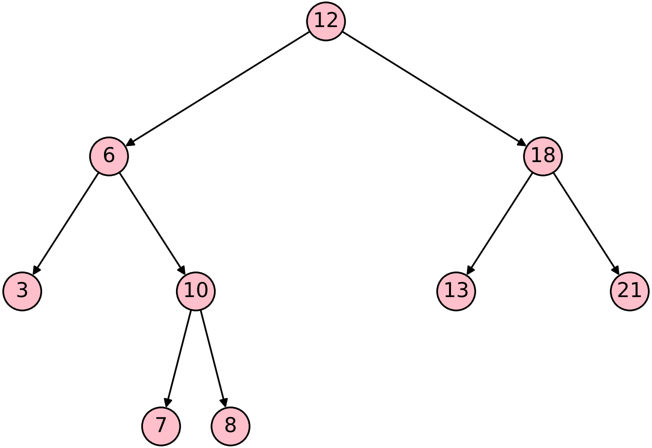
- des **feuilles** : sommets de degré 1 (sans fils) ;
- des **sommets internes** : sommets de degré > 1.



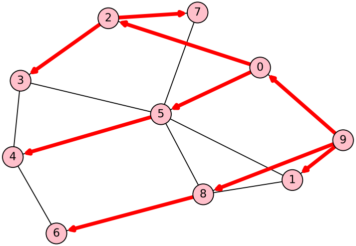
Arbre binaire (complet) : chaque sommet a au plus deux fils.
On parlera souvent de *fils gauche* et *fils droit*.



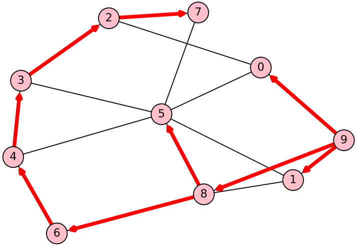
Arbre de Recherche : les valeurs à gauche d'un sommet sont plus petites que celle du sommet, et celles à droite plus grandes.



Le parcours d'un graphe définit un arbre :



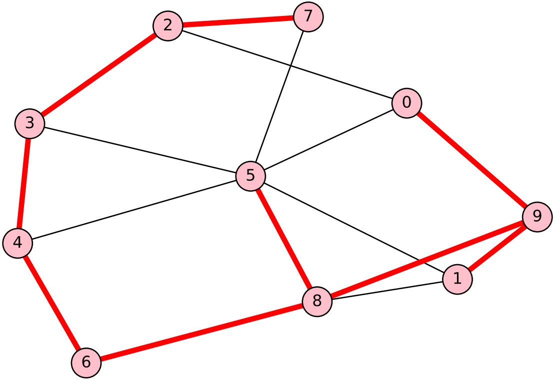
(a) Largeur



(b) Profondeur

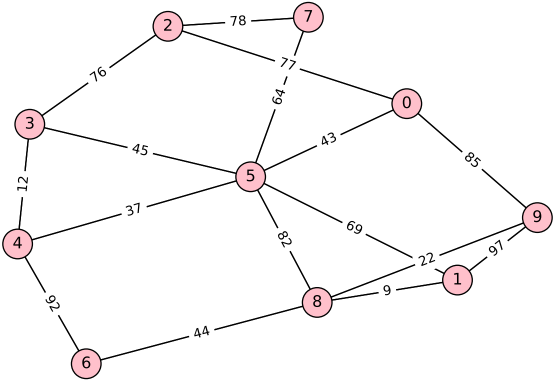
Arbres Couvrants

Un **arbre couvrant** est un arbre qui passe par tous les sommets d'un graphe :



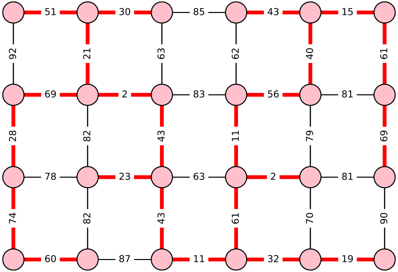
On peut toujours trouver un arbre couvrant (il suffit d'enlever les arêtes en trop jusqu'à avoir un arbre).

Et dans un graphe pondéré ?



On peut chercher un arbre couvrant **de poids minimal**.
(ie. la somme des poids des arêtes dans l'arbre est minimale.)

Historiquement : en 1926, le scientifique Tchèque Otakar Borůvka se demande comment développer un réseau de distribution d'électricité efficace à Moravia.

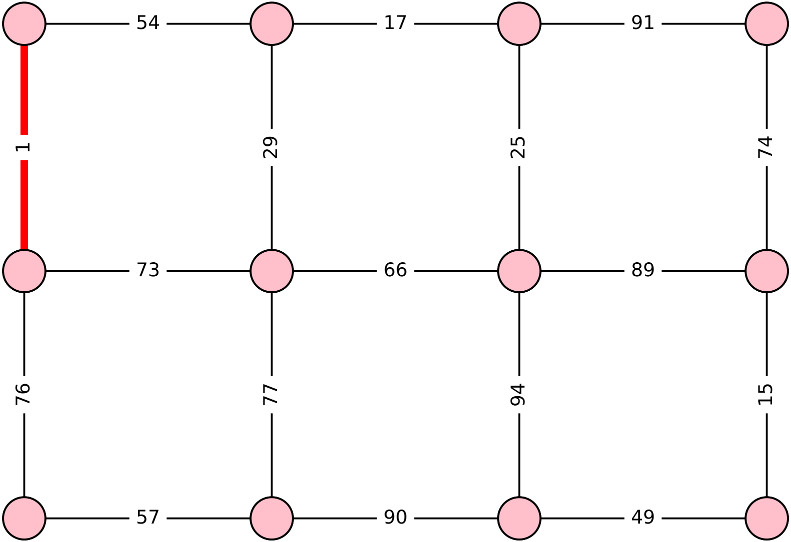


On peut aussi s'en servir pour créer des labyrinthes : DM.

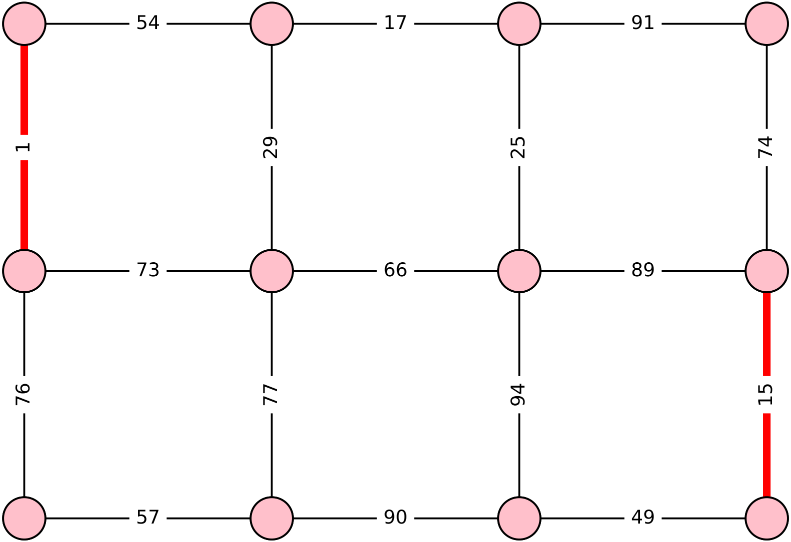
L'algorithme de Kruskal permet de trouver un arbre couvrant minimal :

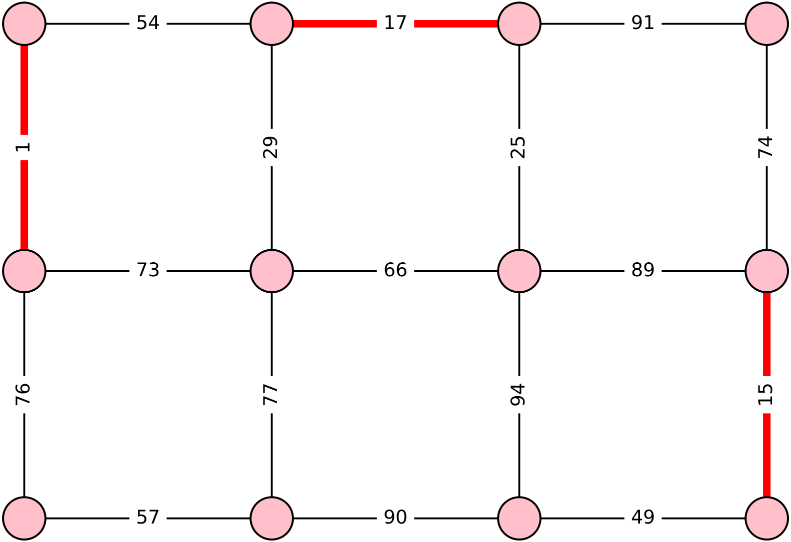
- Au début, il n'y a aucune arête dans l'arbre.
- Itérer sur les arêtes (u, v) par poids croissants :
  - si (u, v) ne sont pas encore reliés, les relier ;
  - sinon, continuer.

Complexité ? Tout réside dans le tri des poids des arêtes !

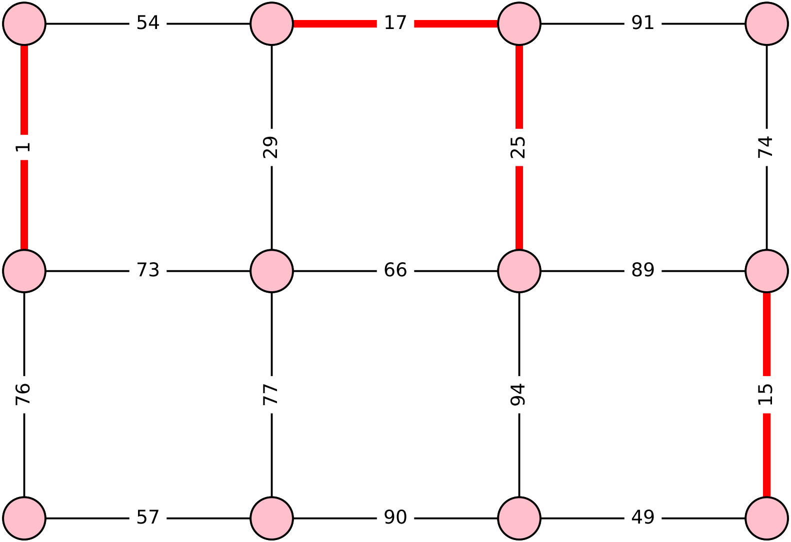

54
17
91
1
29
25
74
73
66
89
76
77
94
15
57
90
49

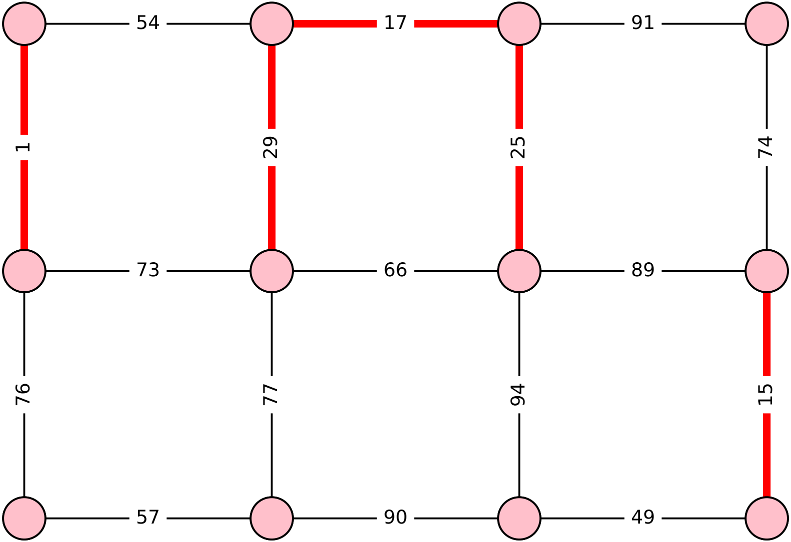
Recherche d'un Arbre Couvrant : Algorithme de Kruskal

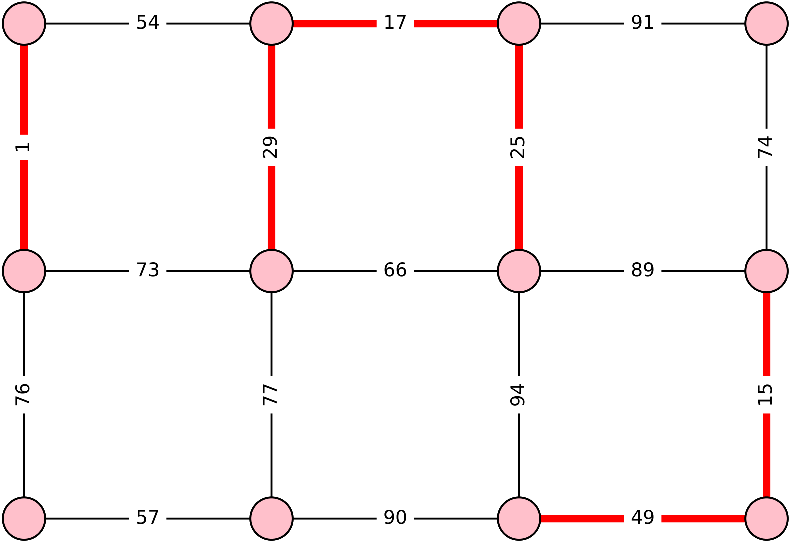


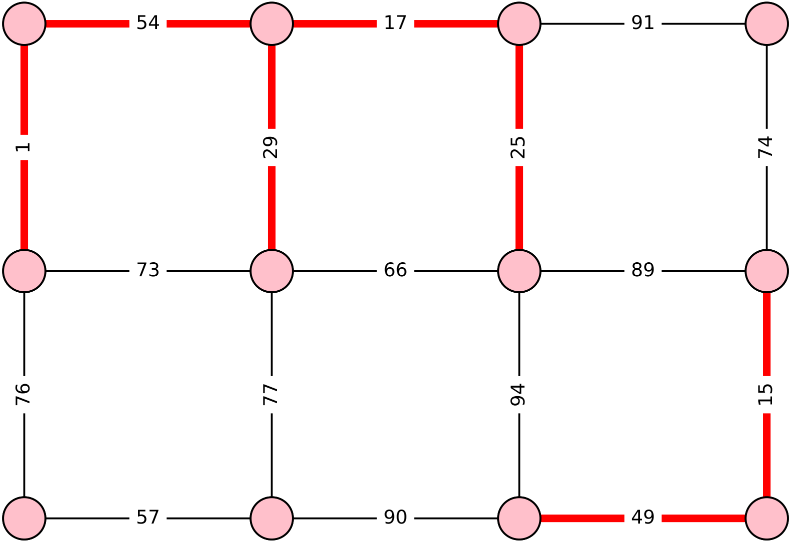

54
17
91
1
29
25
74
73
66
89
76
77
94
15
57
90
49

Recherche d'un Arbre Couvrant : Algorithme de Kruskal

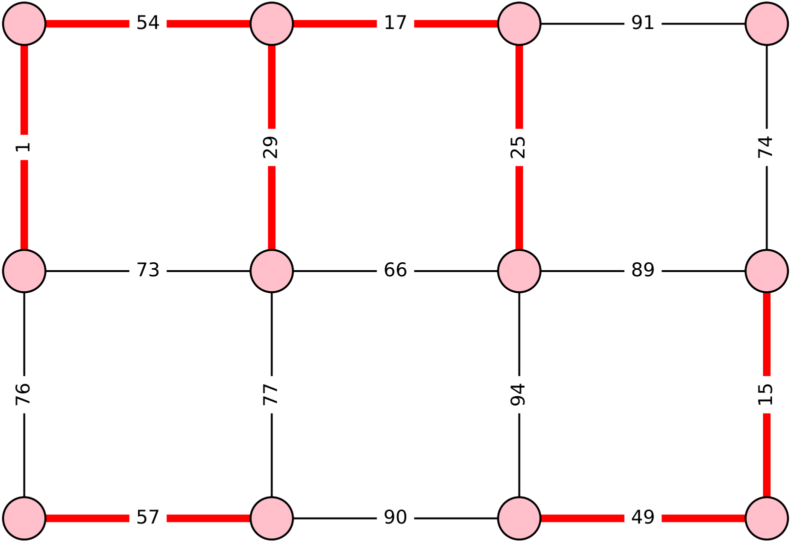



54
17
91
1
29
25
74
73
66
89
76
77
94
15
57
90
49


54
17
91
1
29
25
74
73
66
89
76
77
94
15
57
90
49



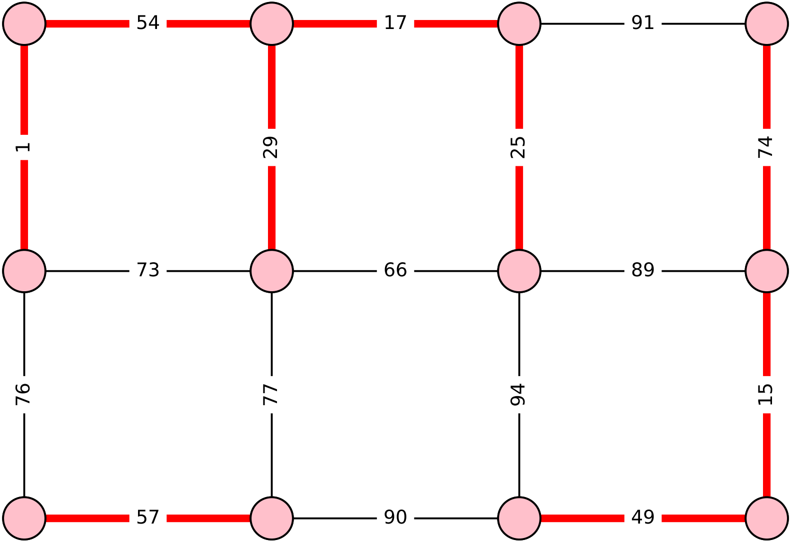
54
17
91
1
29
25
74
73
66
89
76
77
94
15
57
90
49

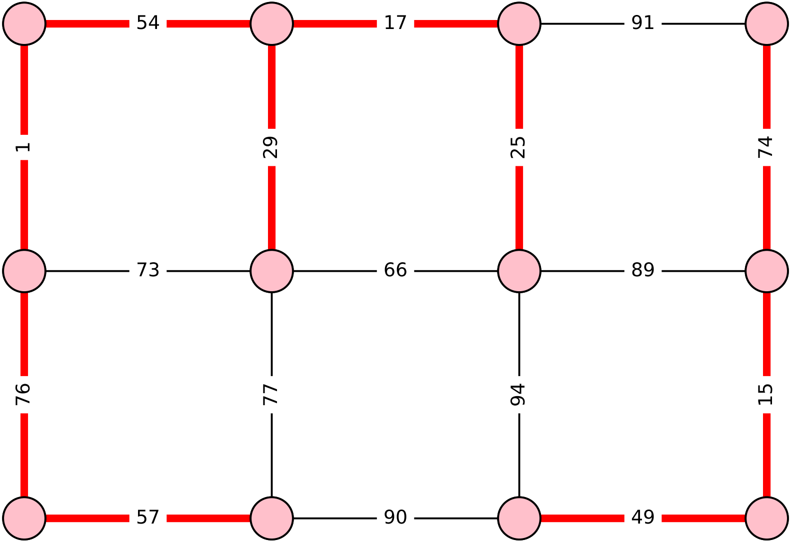


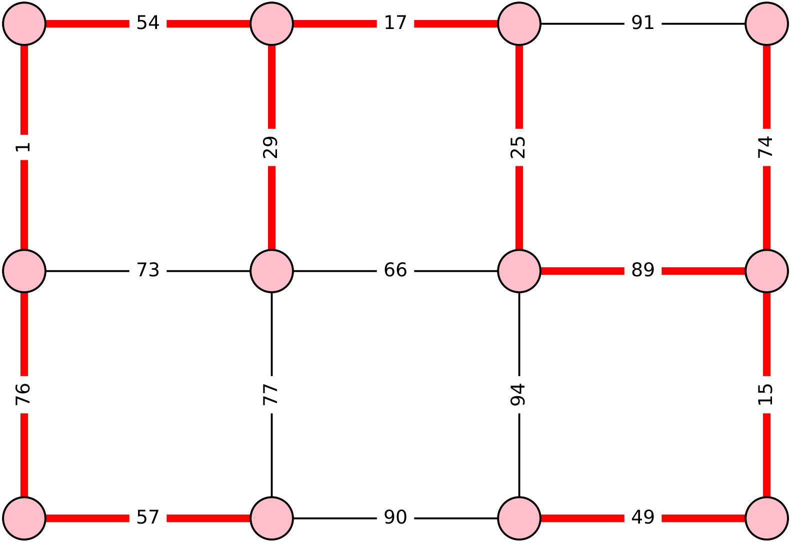
54
17
91
1
29
25
74
73
66
89
76
77
94
15
57
90
49

Arbres Couvrants

Recherche d'un Arbre Couvrant : Algorithme de Kruskal




54
17
91
1
29
25
74
73
66
89
76
77
94
15
57
90
49



54
17
91
1
29
25
74
73
66
89
76
77
94
15
57
90
49

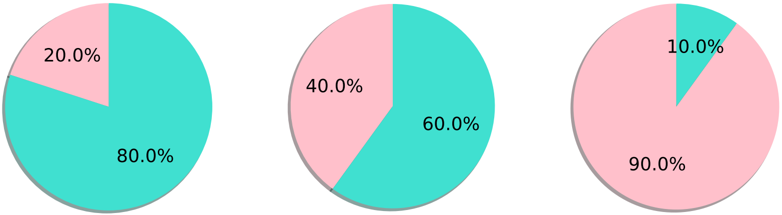
Cet algorithme permet de générer des labyrinthes.

$\rightarrow$ voir le DM : implémentation de l'algorithme de Kruskal et application à la génération de labyrinthes.

(Ce DM est facultatif, mais il se veut court et permet de comprendre des notions du cours. La note sera comptée en bonus. En outre, il est très recommandé de le faire !)

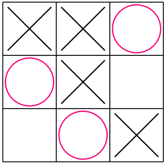
Arbres dans la Théorie des Jeux

Un **jeu à somme nulle** est un jeu où le gain d'un joueur entraîne la perte de l'autre.
Couper un gâteau est un jeu à somme nulle :

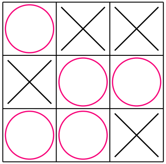


Certains disent que la politique est un jeu à somme nulle.

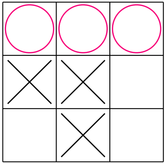
Le morpion est un jeu à somme nulle.
On représente le résultat pour chaque joueur par 1, -1 ou 0.



(a) 1/-1



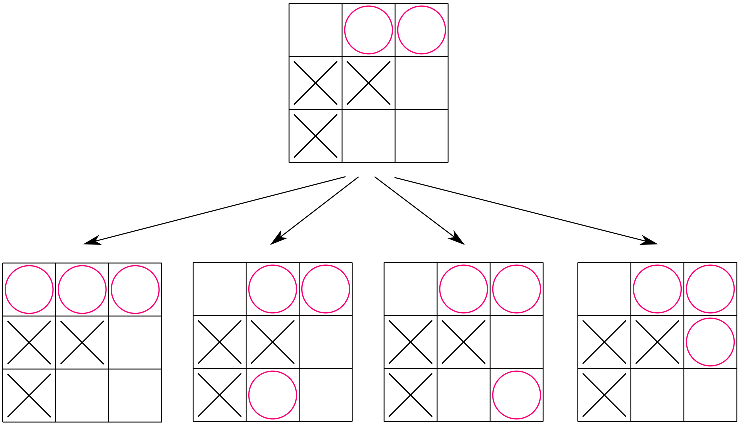
(b) 0/0



(c) -1/1

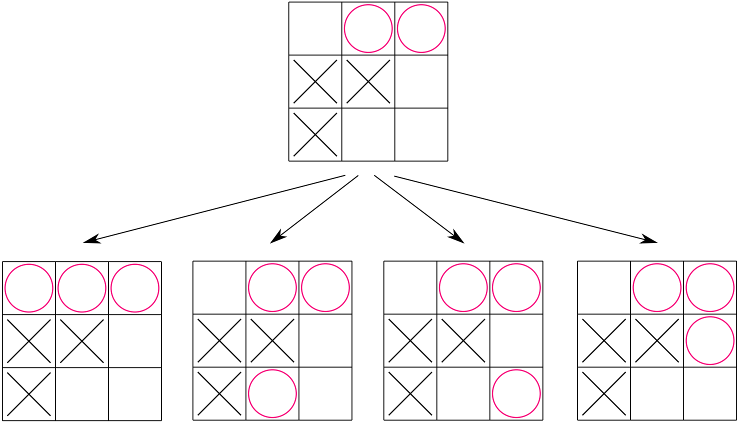
La somme des deux fait bien toujours 0 !

Les coups possibles peuvent être représentés par un arbre :



Chaque arête de l'arbre représente un coup, chaque sommet une position.

Pour construire une IA pour ce jeu... il nous faut un moyen de choisir quel coup jouer !



Il nous faut une **fonction d'évaluation**.
Une bonne façon de faire est de regarder ce qu'il se passe après chacun des coups possibles... en fait, on **parcourt l'arbre du jeu** !

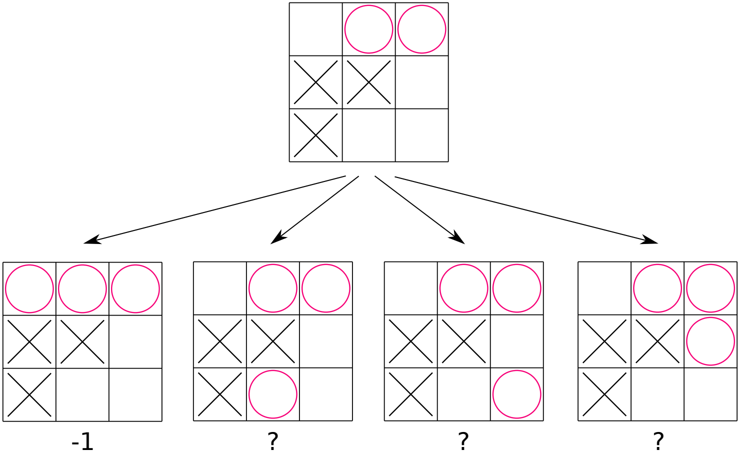
On évalue une position avec la fonction suivante :

$\textsc{Evaluation}(\textsc{position})$:

- si c'est une position finale : renvoyer le score ;
- sinon, pour chaque arête de l'arbre $(\textsc{position}, \textsc{enfant})$, calculer $\textsc{Evaluation}(\textsc{enfant})$, et agréger les résultats.

C'est un **parcours en profondeur** !

Parfois on a la réponse directement, parfois il faut creuser un peu plus...



Cela permet de construire l'algorithme MinMax (ou Minimax) :

$\textsc{MinMax}(\textsc{position}, \textsc{joueur})$:

- Si dans $\textsc{position}$ la partie est finie :
  - si le joueur 1 gagne, renvoyer `1` ;
  - si le joueur -1 gagne, renvoyer `-1` ;
  - si le match est nul, renvoyer `0`.
- Sinon :
  - si c'est au joueur 1 de jouer :
    définir `v =` $-\infty$
    pour chaque position $\textsc{pos}$ accessible en un coup :
    `v = Max(v,` $\textsc{MinMax}(\textsc{pos}, \textsc{-joueur})$`),`
  - si c'est au joueur -1 de jouer :
    définir `v =` $+\infty$
    pour chaque position $\textsc{pos}$ accessible en un coup :
    `v = Min(v,` $\textsc{MinMax}(\textsc{pos}, \textsc{-joueur})$`).`

$\rightarrow$ En fait, on regarde ce qui arrive si les deux joueurs jouent les meilleurs coups à chaque fois.

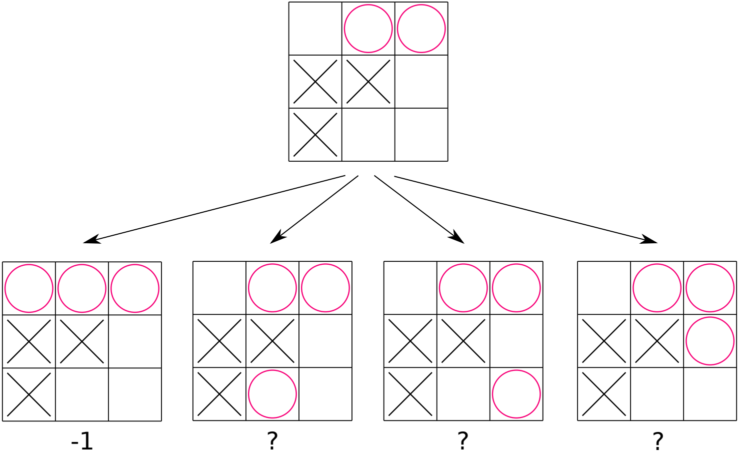
L'algorithme MinMax permet d'évaluer la position, mais il demande quand même de parcourir l'arbre tout entier pour évaluer une position.

C'est encore jouable pour le morpion, mais cela devient vite impossible.
Par exemple, aux échecs, il y a beaucoup trop de coups possibles pour qu'une telle stratégie soit efficace.

Mais on n'a pas besoin de parcourir tous les sommets !

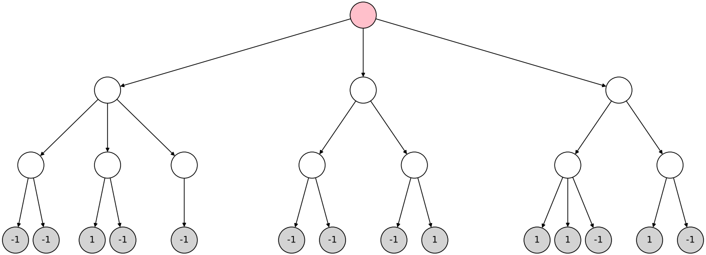
C'est le principe de l'élagage AlphaBeta : on évite de parcourir les coups qui n'apportent pas d'information.

Reprenons l'exemple précédent :

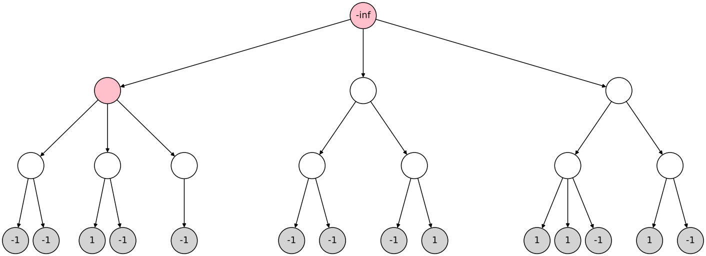


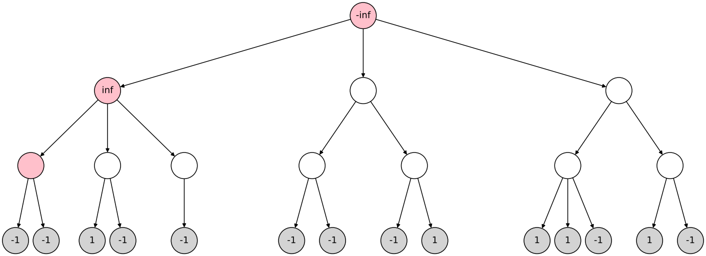
$\textsc{AlphaBeta}(\textsc{position},\ \textsc{joueur},\ \alpha,\ \beta)$:

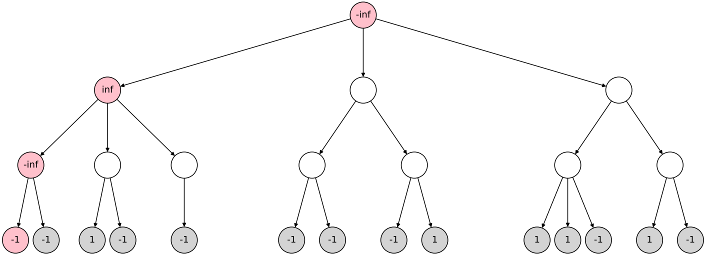
- Si dans $\textsc{position}$ la partie est finie :
  - si le joueur 1 gagne, renvoyer `1` ;
  - si le joueur -1 gagne, renvoyer `−1` ;
  - si le match est nul, renvoyer `0`.
- Sinon,
  - si c'est au joueur 1 de jouer :
    définir `v =` $-\infty$
    pour chaque position $\textsc{pos}$ accessible en un coup :
    `v = Max(v,` $\textsc{AlphaBeta}(\textsc{pos},\ \text{-}\textsc{joueur},\ \alpha,\ \beta)$`)`
    α `= Max(v,` α`)`
    si $\alpha \geq \beta$, renvoyer *v*,
  - si c'est au joueur -1 de jouer :
    définir `v =` $+\infty$
    pour chaque position $\textsc{pos}$ accessible en un coup :
    `v = Min(v,` $\textsc{AlphaBeta}(\textsc{pos},\ \text{-}\textsc{joueur},\ \alpha,\ \beta)$`)`
    β `= Min(v,` β`)`
    si $\alpha \geq \beta$, renvoyer *v*.

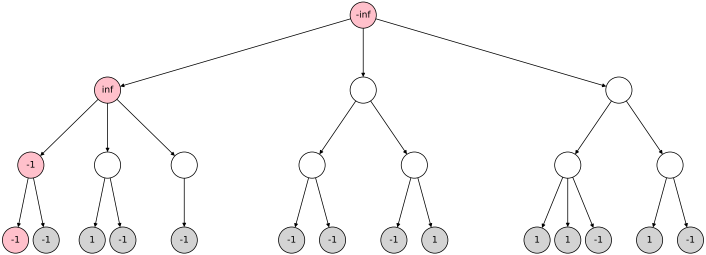


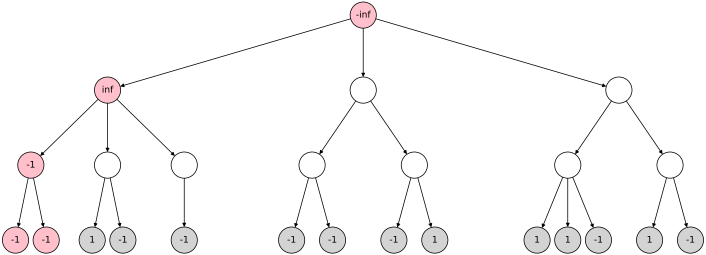
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

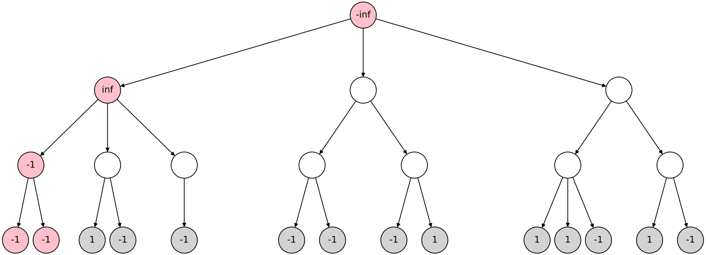

-inf
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

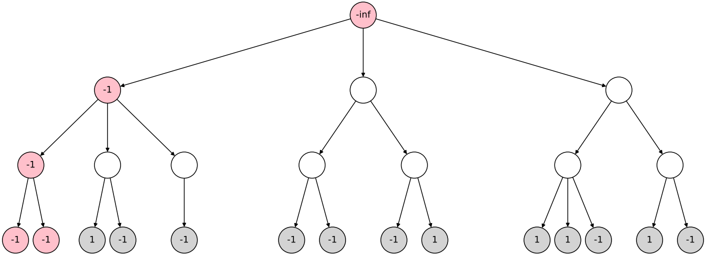

-inf
inf
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

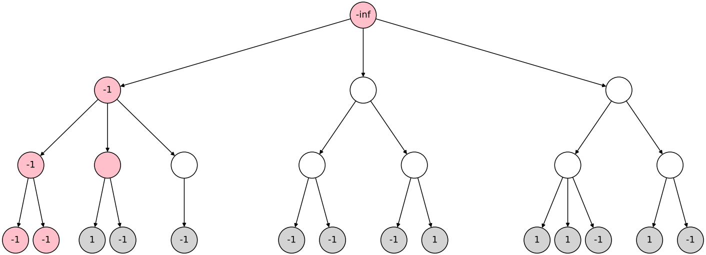

-inf
inf
-inf
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1


-inf
inf
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

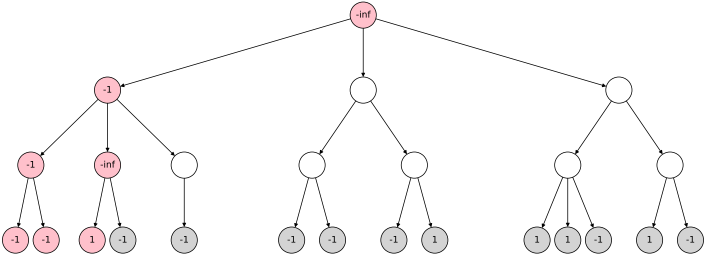

-inf
inf
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

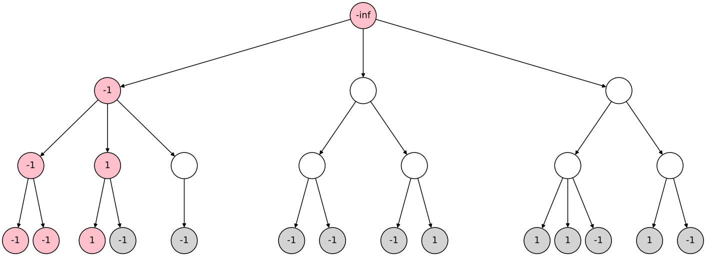

-inf
inf
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

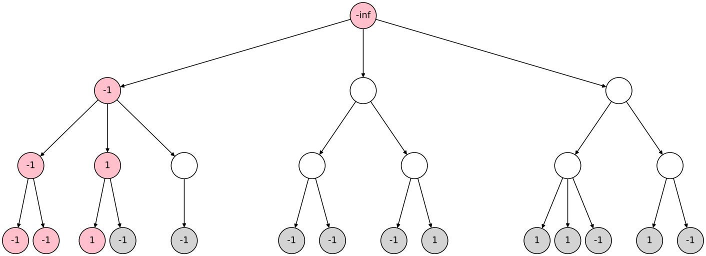

-inf
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

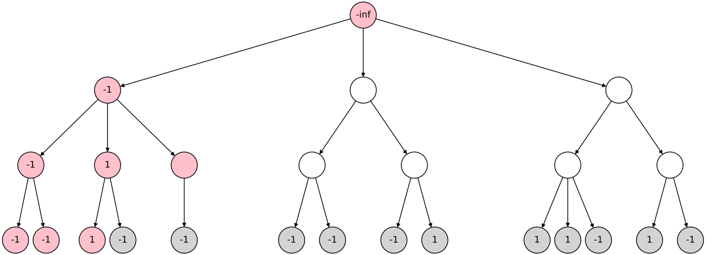


-inf
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

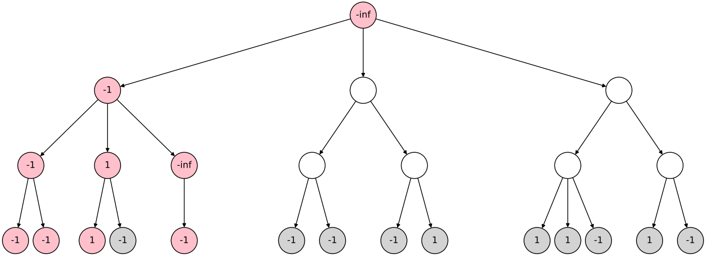

-inf
-1
-1
-inf
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

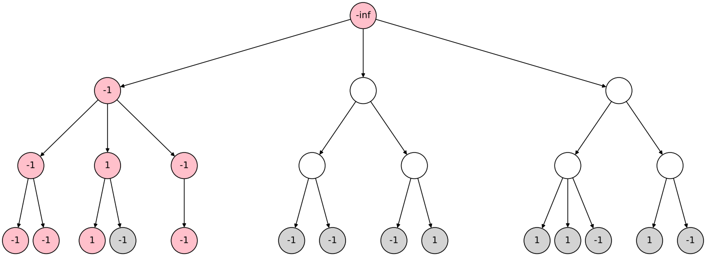

-inf
-1
1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1


-inf
-1
1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

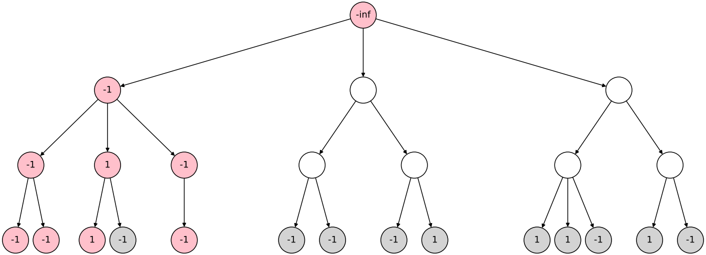


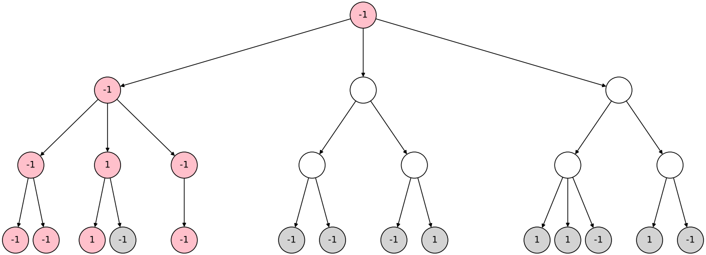
-inf
-1
-1
1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

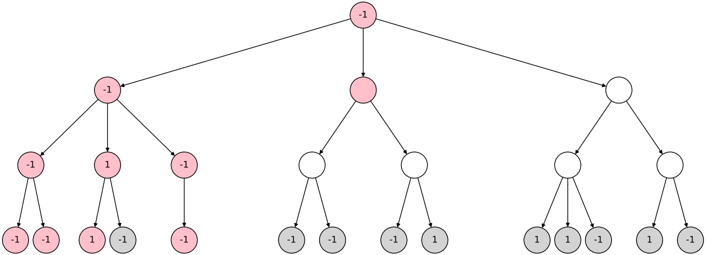

-inf
-1
-1
1
-inf
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

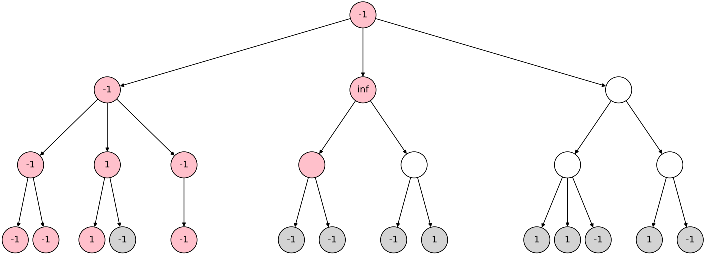


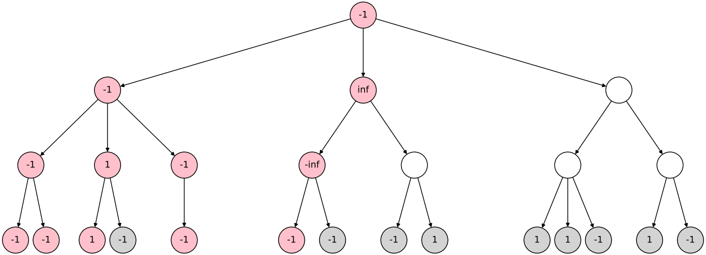
-inf
-1
-1
1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

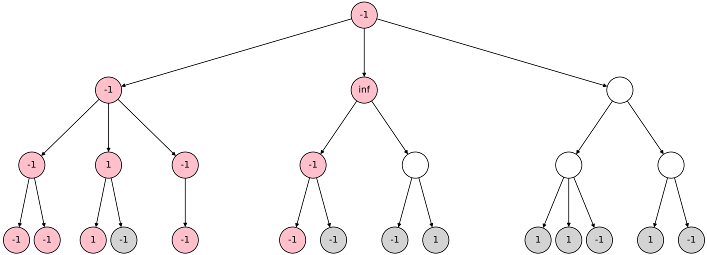

-inf
-1
-1
1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

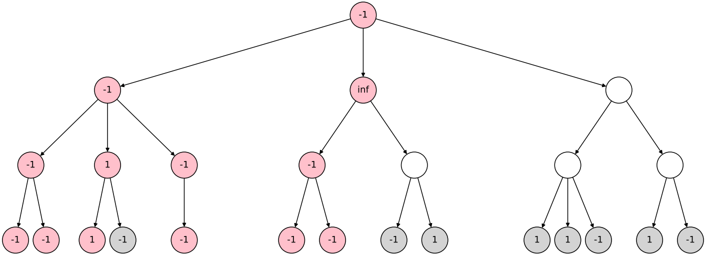

-1
-1
-1
1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

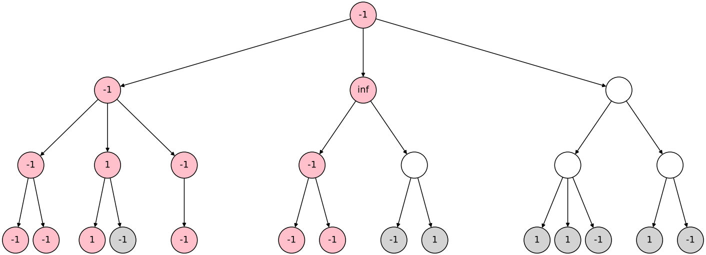

-1
-1
-1
1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1


-1
-1
inf
-1
1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

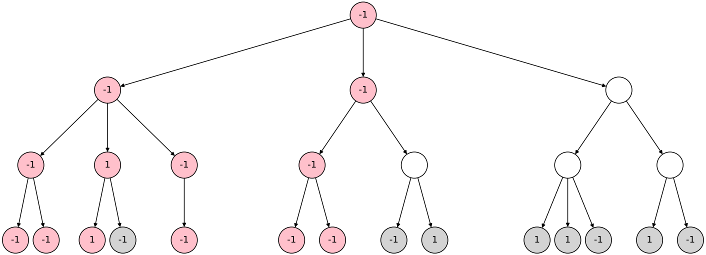

-1
-1
inf
-1
1
-1
-inf
-1
-1
1
-1
-1
-1
-1
1
1
1
-1
1
-1

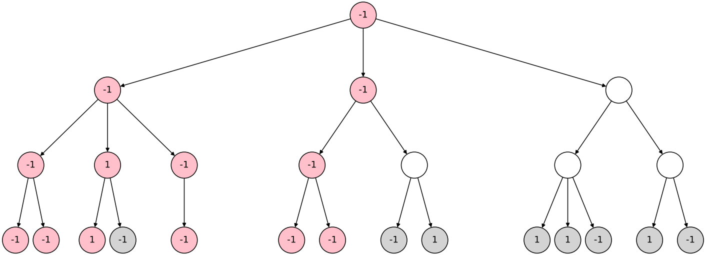

-1
-1
inf
-1
1
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

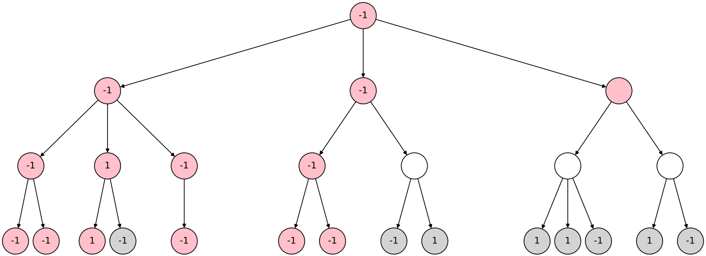

-1
-1
inf
-1
1
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

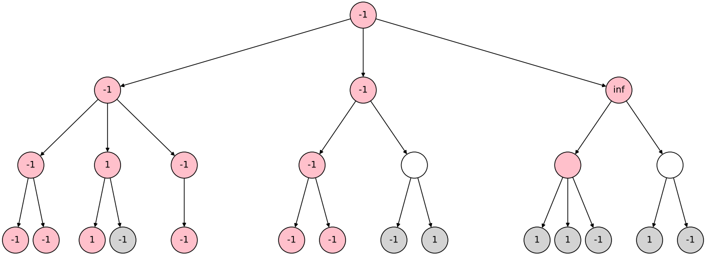


-1
-1
inf
-1
1
-1
-1
-1
-1
1
-1
-1
-1
-1
1
1
1
-1
1
-1


-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

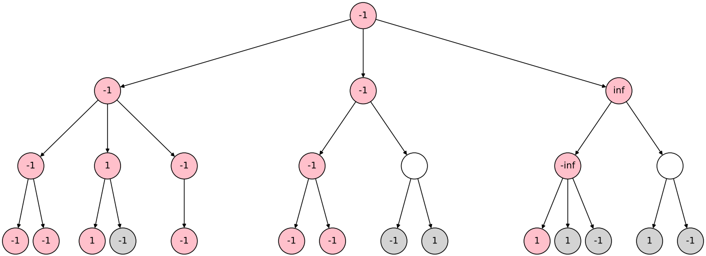

-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

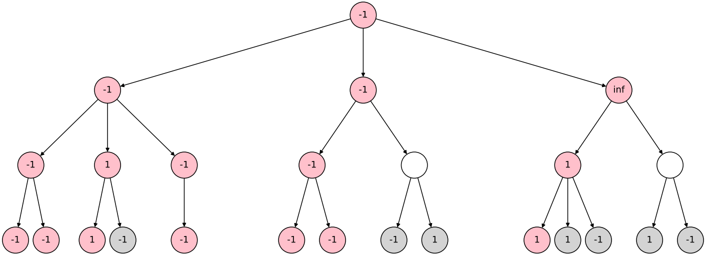

-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1



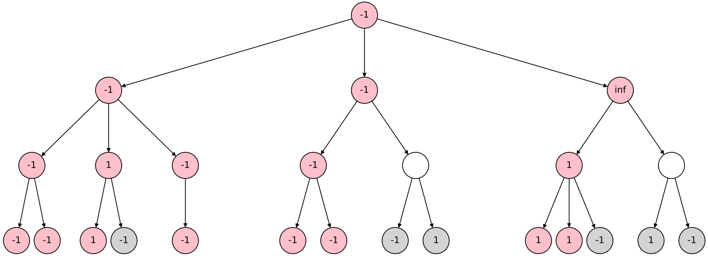
-1
-1
-1
inf
-1
1
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

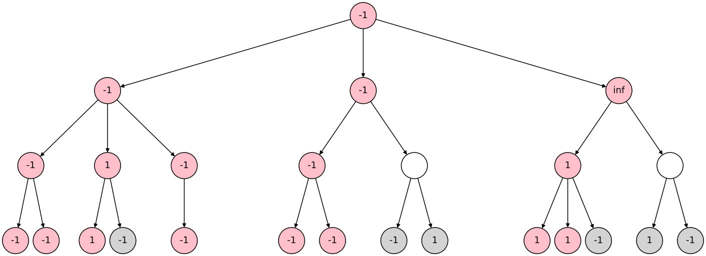
Élagage de l'Arbre du Jeu

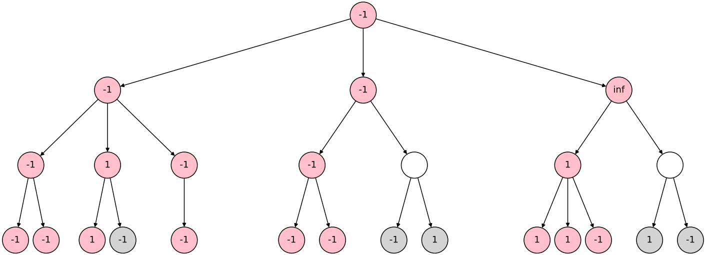


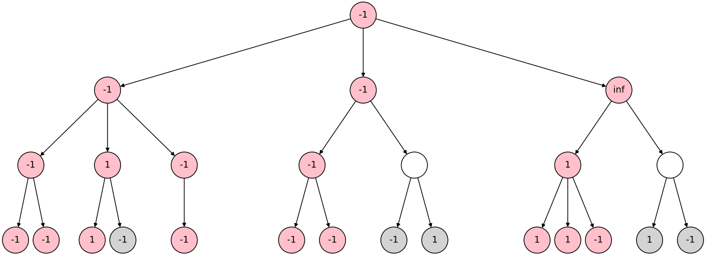


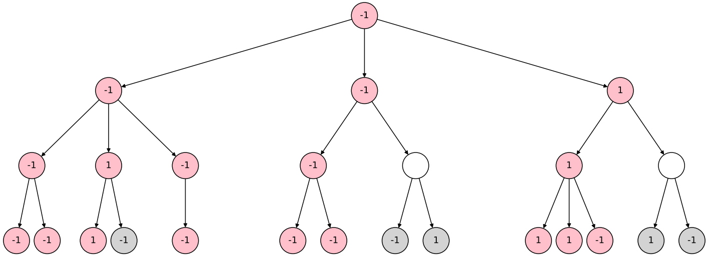
-1
-1
-1
inf
-1
1
-1
-1
1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1


-1
-1
-1
inf
-1
1
-1
-1
1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

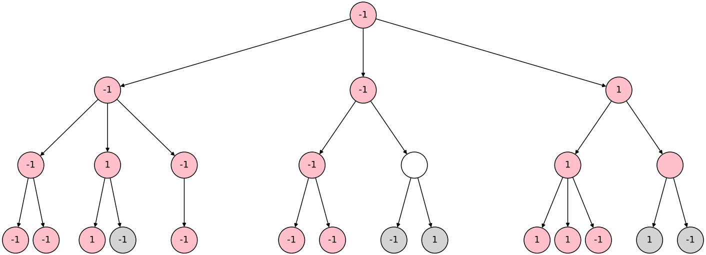

-1
-1
-1
inf
-1
1
-1
-1
1
-1
-1
1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

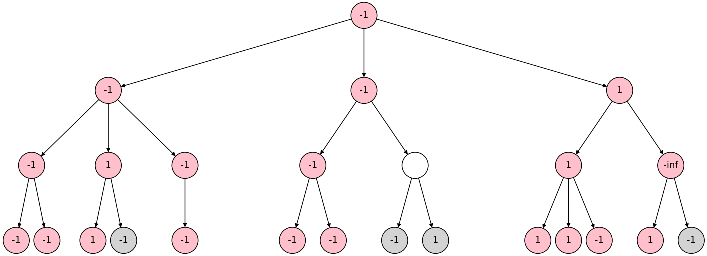

-1
-1
-1
inf
-1
1
-1
-1
1
-1
-1
1
-1
1
-1
-1
-1
-1
1
1
1
-1
1
-1


-1
-1
-1
inf
-1
1
-1
-1
1
-1
-1
1
-1
1
1
-1
-1
-1
-1
1
1
1
-1
1
-1

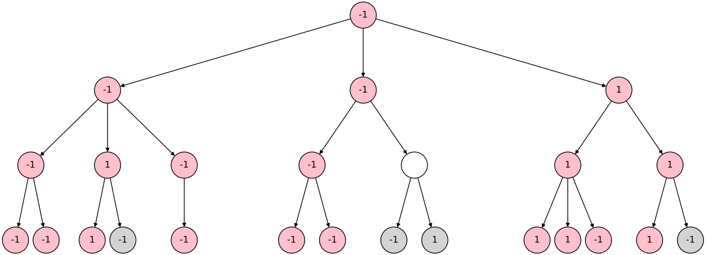


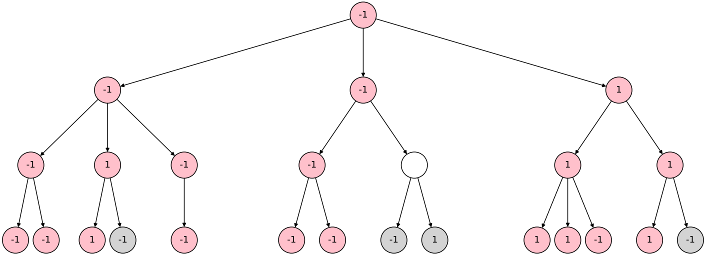
-1
-1
-1
1
-1
1
-1
-1
1
-1
-1
1
-1
-1
-1
1
1
1
-1
-1
1
1
-1
1
-1


-1
-1
-1
1
-1
1
-1
-1
-1
1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

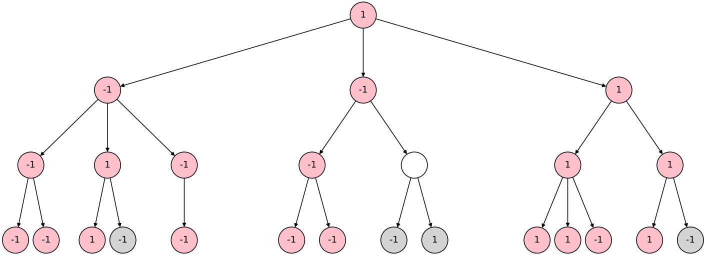


-1
-1
-1
1
-1
1
-1
-1
-1
1
-inf
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1


-1
-1
-1
1
-1
1
-1
-1
1
1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1



-1
-1
-1
1
-1
1
-1
-1
1
1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1


1
-1
-1
1
-1
1
-1
-1
1
1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
1
-1

TP

TP

Dans le TP on va :

- regarder à quoi ressemble l'arbre du jeu de morpion ;
- implémenter l'algorithme MinMax et voir ses limites ;
- implémenter l'élagage AlphaBeta ;
- construire une IA qui joue (bien !) au morpion.