CHAPITRE II

Algorithmes de recherche et résolution des problèmes

Résolution des problèmes

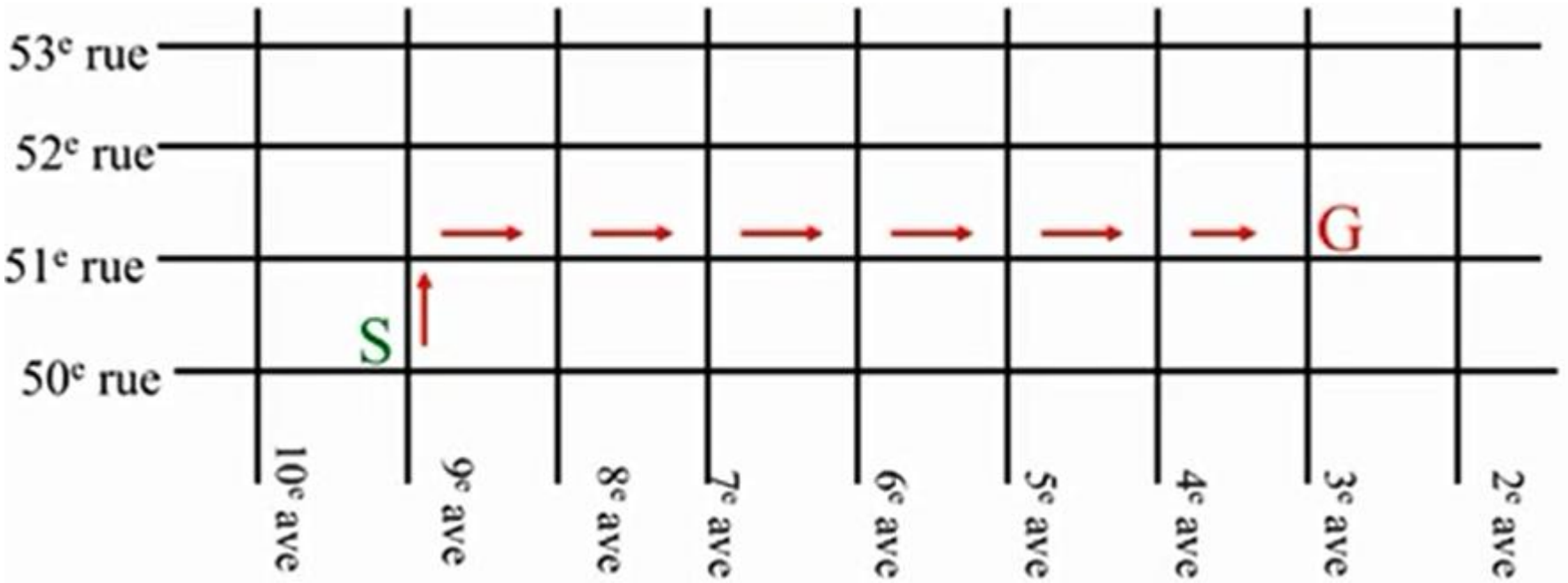
- Étapes intuitives par un humain
  1. modéliser la situation actuelle
  2. énumérer les solutions possibles
  3. évaluer la valeur des solutions
  4. retenir la meilleure option possible satisfaisant le but

- Mais comment parcourir efficacement la liste des solutions?

- La résolution de plusieurs de problèmes peut être faite par **une recherche dans un graphe**
  - chaque noeud correspond à un état de l'environnement
  - chaque chemin à travers un graphe représente alors une suite d'actions prises par l'agent
  - pour résoudre notre problème, suffit de chercher le chemin qui satisfait le mieux notre mesure de performance

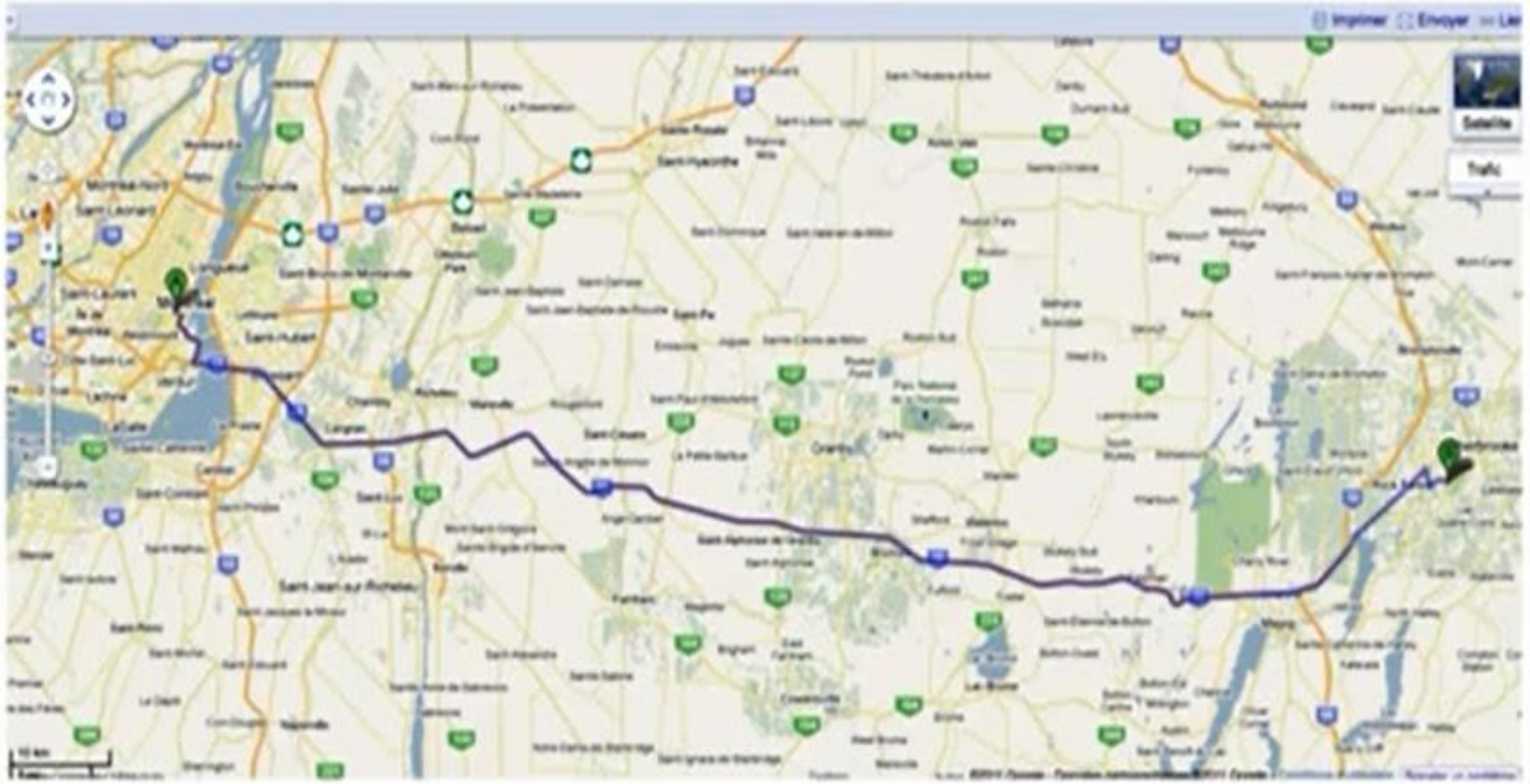
Exemple : trouver chemin dans ville

Trouver un chemin de la 9e ave & 50e rue à la 3e ave et 51e rue

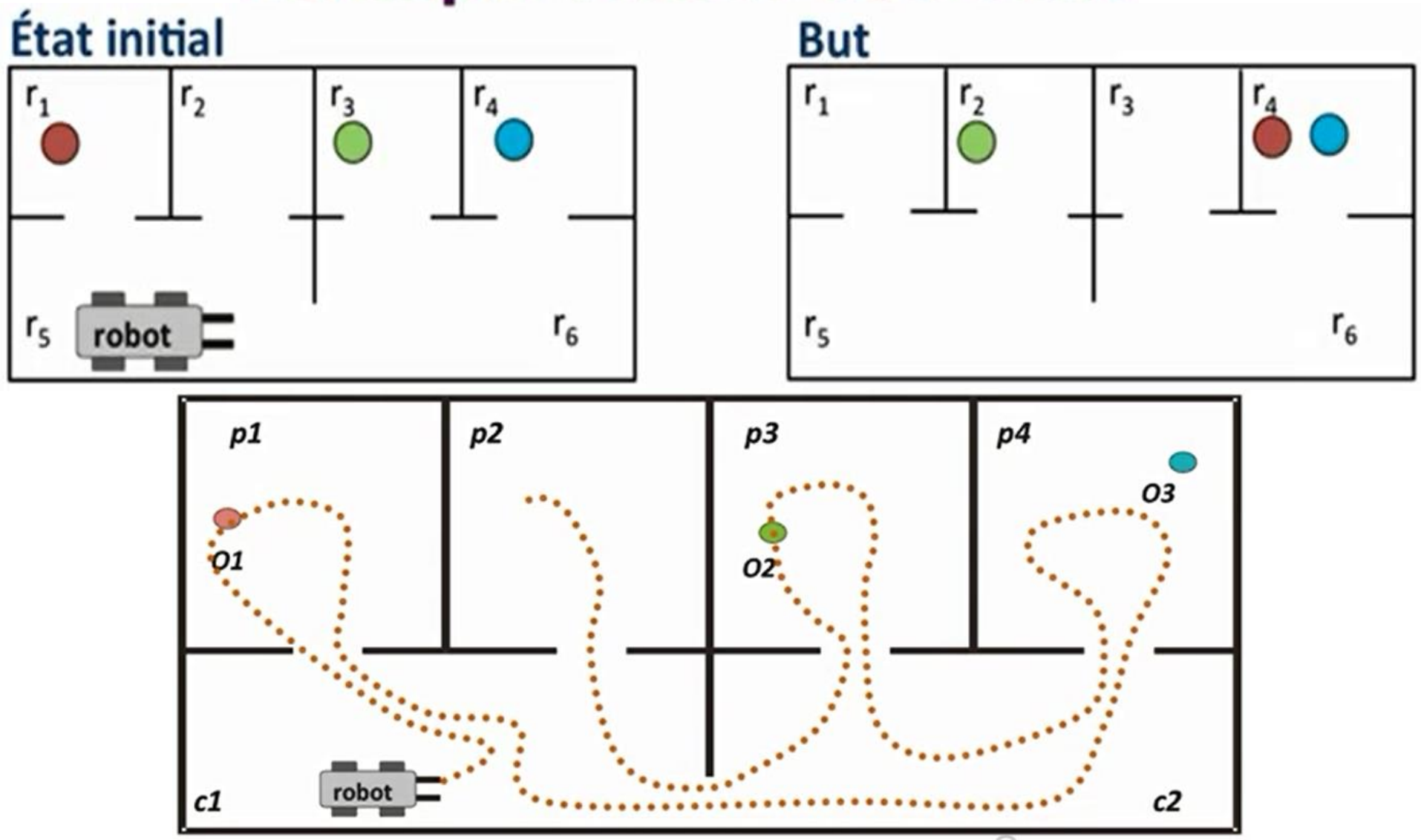


Résolutions des problèmes

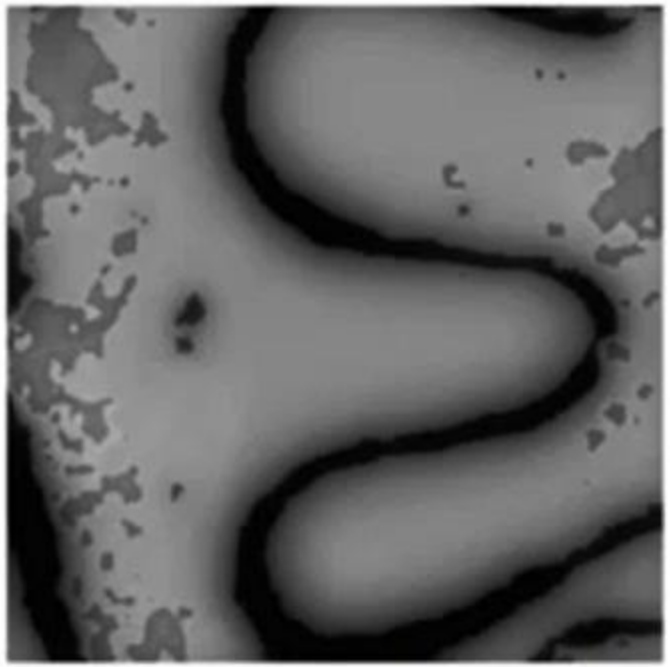
Exemple : Google Maps



Exemple : livrer des colis

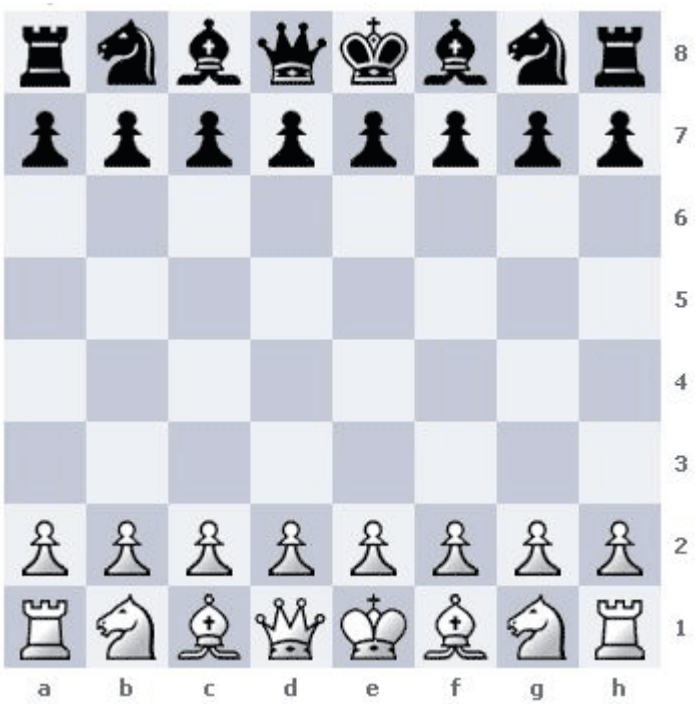


Exemple : navigation d'un robot

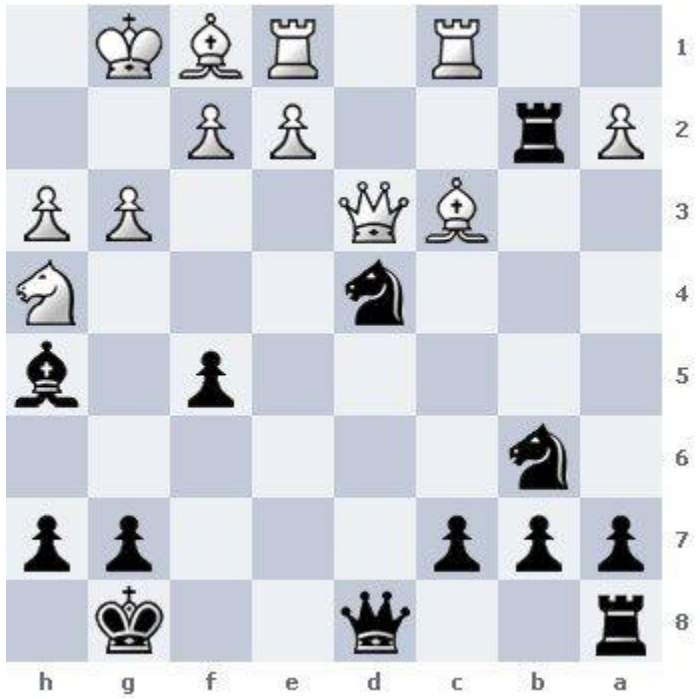


Exemple : Jeu d'échec

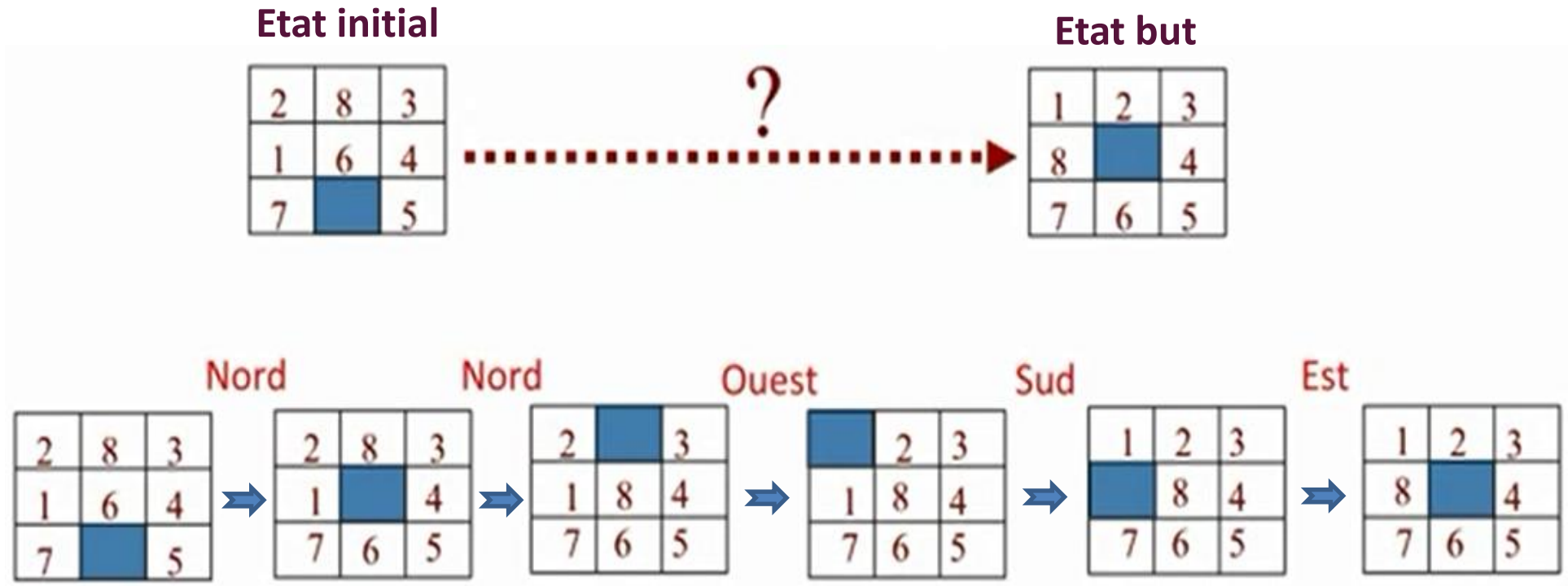
Etat initial



Etat but



Exemple : N-Puzzle (Jeu de Taquin)



Problème de recherche dans un graphe

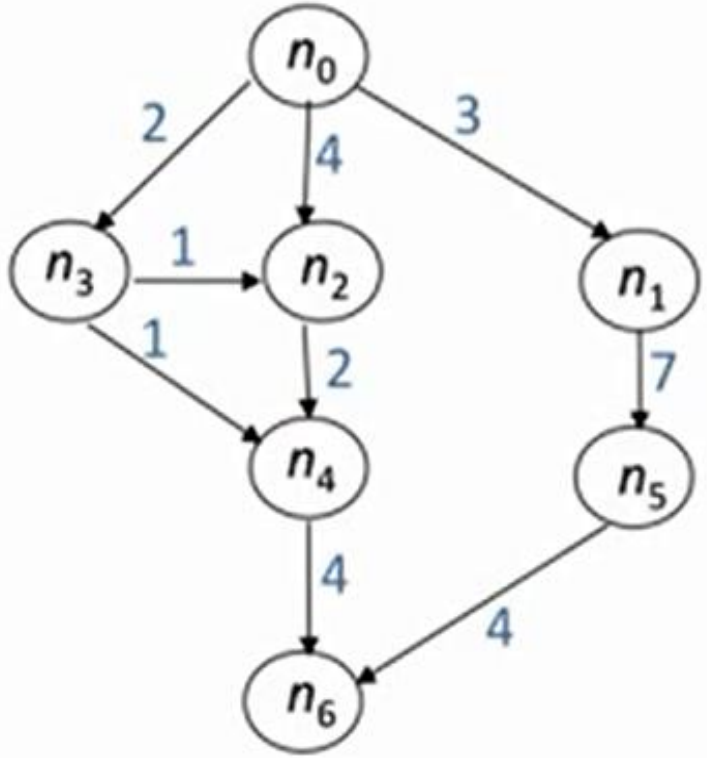
- Algorithme de recherche dans un graphe
  - Entrées :
    - » un nœud initial
    - » une fonction *goal*(*n*) qui retourne *true* si le but est atteint
    - » une fonction de transition *transitions*(*n*) qui retourne les nœuds successeurs de *n*
    - » une fonction *c*(*n*,*n'*) strictement positive, qui retourne le coût de passer de *n* à *n'* (permet de considérer le cas avec coûts variables)
  - Sortie :
    - » un chemin dans un graphe (séquence nœuds / arrêtes)
  - Le **coût d'un chemin** est la **somme des coûts des arrêtes** dans le graphe
  - Il peut y avoir plusieurs nœuds qui satisfont le but

- Enjeux :
  - trouver un chemin solution, ou
  - trouver un chemin optimal, ou
  - trouver rapidement un chemin (optimalité pas importante)

Exemple : trouver chemin entre deux villes

- Villes : nœuds
- Chemins entre deux villes : arrêtes

- Ville de départ : nœud (état) initial n_0
- Routes entre les villes : *transitions*$(n_0) = (n_3, n_2, n_1)$
- Distances entre les villes : $c(n_0,n_2) = 4$
- Ville de destination : *goal*(n) : vrai si $n=n_6$
  (où n_6 est le nœud de la ville de destination)



Algorithme de recherche : Heuristique

Meilleur d'abord : Best-first search

1. Démarrer la recherche avec la liste contenant l'**état initial** du problème.

2. Si la liste n'est pas vide alors :
   - Choisir un état **n** de mesure **minimale** à traiter.
   - Si **n** est un **état final** alors retourner succès
   - Sinon, rajouter tous les successeurs de **n** à la liste d'états à traiter par **ordre croissant** selon la mesure d'utilité.
   Recommencer au point 2.

3. Sinon retourner échec.

Algorithme de recherche : Heuristique

Meilleur d'abord : Best-first search

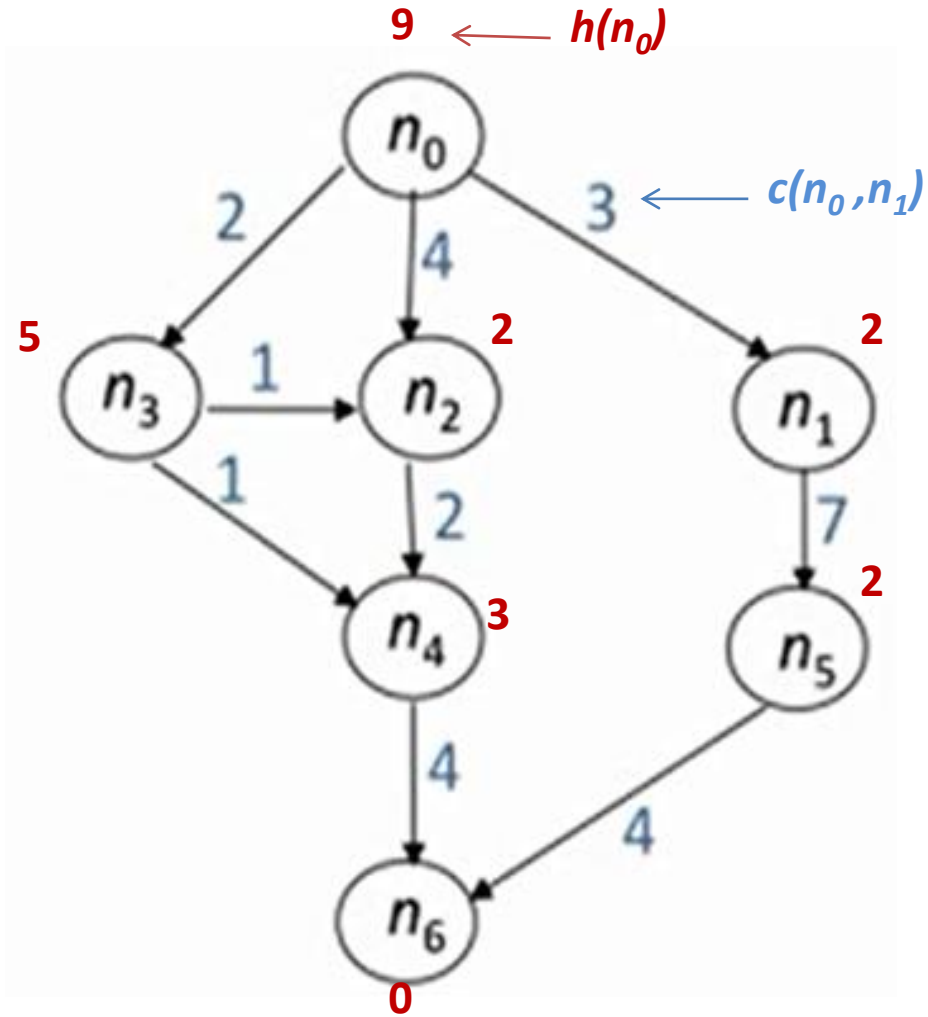
Cas particulier 1 : Recherche gloutonne (greedy best-first search)

- La mesure d'utilité est donnée par une fonction d'estimation ***h***.
- Pour chaque état ***n***, ***h(n)*** représente l'estimation du coût de ***n*** vers un état final.

Par exemple, dans le problème du chemin le plus court entre deux villes on peut prendre ***h(n)* = distance directe** entre ***n*** et la **ville destination**.

- La recherche gloutonne choisira l'état qui semble le plus proche d'un état final selon la fonction d'estimation.

Cas particulier 1 : Recherche gloutonne (greedy best-first search**)**



Algorithme de recherche : Heuristique

Meilleur d'abord : Best-first search

Cas particulier 2 : Algorithme A*

- On évite d'explorer les chemins qui sont déjà chers.
- La mesure d'utilité est donnée par une fonction d'évaluation ***f***.
- Pour chaque état ***n*** : $f(n) = g(n) + h(n)$, où :

$g(n)$ est le coût jusqu'à présent pour atteindre *n*

$h(n)$ est le coût estimé pour aller de *n* vers un état final.

$f(n)$ est le coût total estimé pour aller d'un état initial vers un état final en passant par ***n***.

Algorithme de recherche : Heuristique

Algorithme A*

- Une heuristique ***h(n)*** est une fonction d'estimation du coût restant entre un nœud ***n*** d'un graphe et le nœud ***but***.

Exemples de fonctions d'heuristique h :

- Chemin entre deux villes :

- Distance à vol d'oiseau entre la ville n et la ville de destination.

- Jeu de taquin :

- Nombre de nombres mal placés

Algorithme de recherche : A*

1. Déclarer deux nœuds ***n***, ***ns***
2. Déclarer deux listes ***Open*** et ***Closed*** (vides au départ)
3. Ajouter le nœud ***Etat initial*** dans ***Open***.
4. Si ***Open*** est vide Alors sortir de la boucle avec un ***échec***.
5. ***n=nœud*** à la tête de ***Open***
6. Enlever ***n*** de ***Open*** et l'ajouter dans ***Closed***.
7. Si ***n=but*** alors sortir de la boucle et ***retourner le chemin***.
8. Sinon : pour chaque successeur ***ns*** de ***n*** :

   - Initialiser la valeur ***g(ns) = g(n) + c(n,ns)***
   - Mettre le parent de ***ns*** à ***n***
   - Si ***Open*** ou ***Closed*** contient un nœud ***ns'=ns*** avec ***f(ns)*** $\leq$ ***f(ns')***

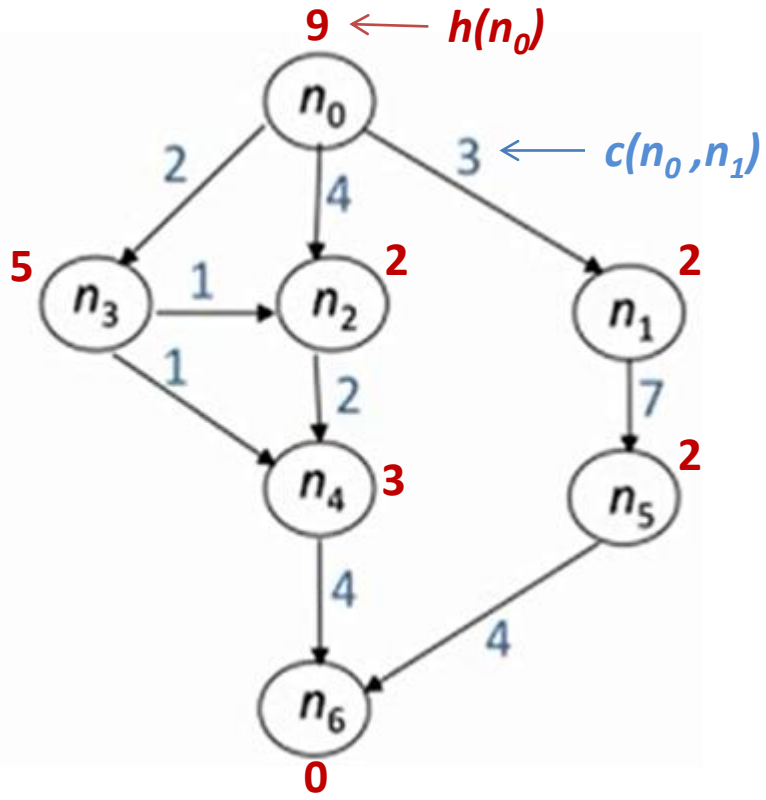
     Alors enlever ***ns'*** de ***Open*** ou ***Closed*** et insérer ***ns*** dans ***Open***

     (en respectant l'ordre croissant de ***f***)

     Sinon : Insérer ***ns*** dans ***Open*** (en respectant l'ordre croissant de ***f***)
   - Aller à 4.

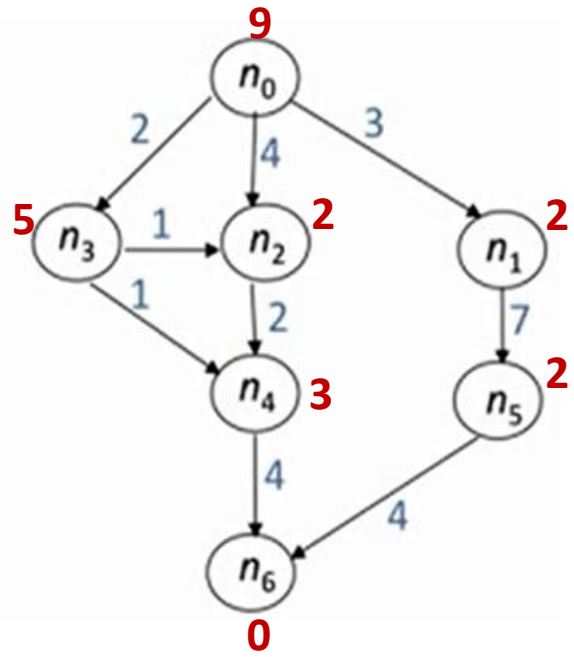
Algorithme A* : Exemple de recherche d'un chemin entre deux villes

n_0 : ville de départ (nœud initial)
n_6 : ville de destination (nœud but)
h : Distance à vol d'oiseau entre une ville et la ville de destination(heuristique)
c : Distance réelle entre deux villes



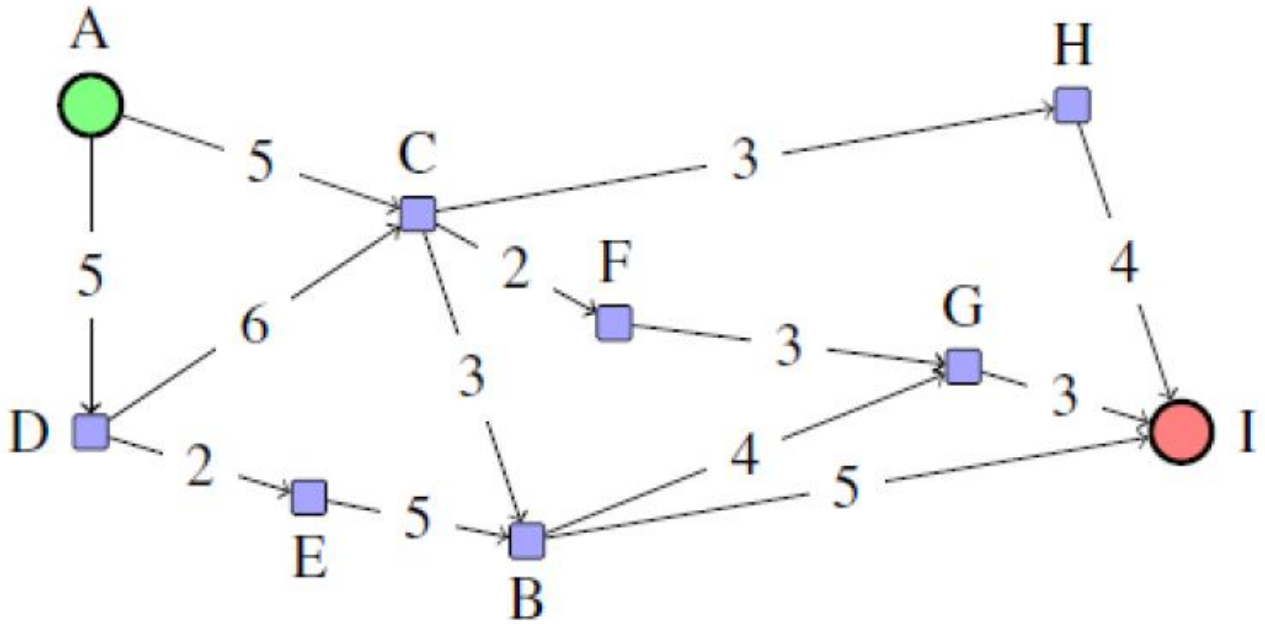
Algorithme A* : Chemin entre deux villes (Déroulement)

Contenu de *open* à chaque itération (état, f, parent) :	Contenu de *closed* à chaque itération :
1. (n_0, 9, void)	1. Vide
2. (n_1,5,n_0), (n_2,6,n_0), (n_3,7,n_0)	2. (n_0, 9, void)
3. (n_2,6,n_0), (n_3,7,n_0), (n_5,12,n_1)	3. (n_0, 9, void), (n_1,5,n_0)
4. (n_3,7,n_0), (n_4,9,n_2), (n_5,12,n_1)	4. (n_0, 9, void), (n_1,5,n_0), (n_2,6,n_0)
5. (n_2,5,n_3), (n_4,6,n_3), (n_5,12,n_1)	5. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0)
6. (n_4,6,n_3), (n_5,12,n_1)	6. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3)
7. (n_6,7,n_4), (n_5,12,n_1)	7. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3)
8. Solution : n_0,n_3,n_4,n_6	8. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3), (n_6,7,n_4)



Exercice

On considère la carte suivante. L'objectif est de trouver le chemin optimal entre **A** et **I**. On donne également deux heuristiques *h1* et *h2* :



Nœud	A	B	C	D	E	F	G	H	I
h1	10	5	5	10	10	3	3	3	0
h2	10	2	8	11	6	2	1	5	0

Appliquez un algorithme glouton en utilisant *h2* puis un algorithme A* en utilisant *h1*
(Donnez la suite des nœuds développés (l'état de la liste OPEN à chaque itération), déduire alors le chemin traversé dans les deux cas ?)

Motivations

- **Rappel sur les avantages de A* :**
  - En entrée, nous avons une fonction (goal(n)) identifiant le nœud but
  - La solution est un chemin (optimal) et non seulement un état final
  - Les nœuds visités sont stockés pour éviter de les revisiter
- Inconvénients : l'espace d'états est trop grand pour enregistrer tous les nœuds visités

- **Caractéristiques d'une recherche locale :**
  - Définition d'une fonction objective à optimiser (ex. une fonction but qui identifie un nœud final)
  - La solution recherchée est juste un nœud optimal (ou proche) et non pas le chemin qui mène au but
  - Inutile d'enregistrer tous les états visités

Exemple : N-Reines (N-Queens)

- **Problème :**
  - Placer N reines sur un échiquier de taille N×N de sorte que deux reines ne s'attaquent pas mutuellement

(Ne jamais positionner deux reines sur la même diagonale, ligne ou colonne)

Fonction objective
Minimiser le nombre de reines qui s'attaquent

Principe d'une recherche locale

Une recherche locale garde juste certains nœuds visités en mémoire :

- La méthode Hill-Climbing : un cas simple qui garde juste un nœud (courant) en mémoire et l'améliore itérativement jusqu'à converger à une solution.
- Les algorithmes génétiques : un cas plus élaboré qui garde un ensemble de nœuds (population) et l'évolue jusqu'à trouver une solution

Objectif d'une recherche locale

En général, il y a une fonction objective à optimiser (minimiser ou maximiser)

- Hill-Climbing : la fonction objective permet de trouver le nœud visité suivant.

- Algorithme génétique : la fonction objective ou fonction de fitness (d'adaptation) intervient dans le calcul de l'ensemble des nœuds successeurs de l'ensemble courant.

✓ La recherche locale ne garantie pas une solution optimale mais elle a la capacité de trouver une solution acceptable rapidement.

Méthode Hill-Climbing

- En entrée :
  - Nœud initial
  - Fonction objective (notée **F(n)**)à optimiser
  - Fonction générant des nœuds successeurs (voisins)

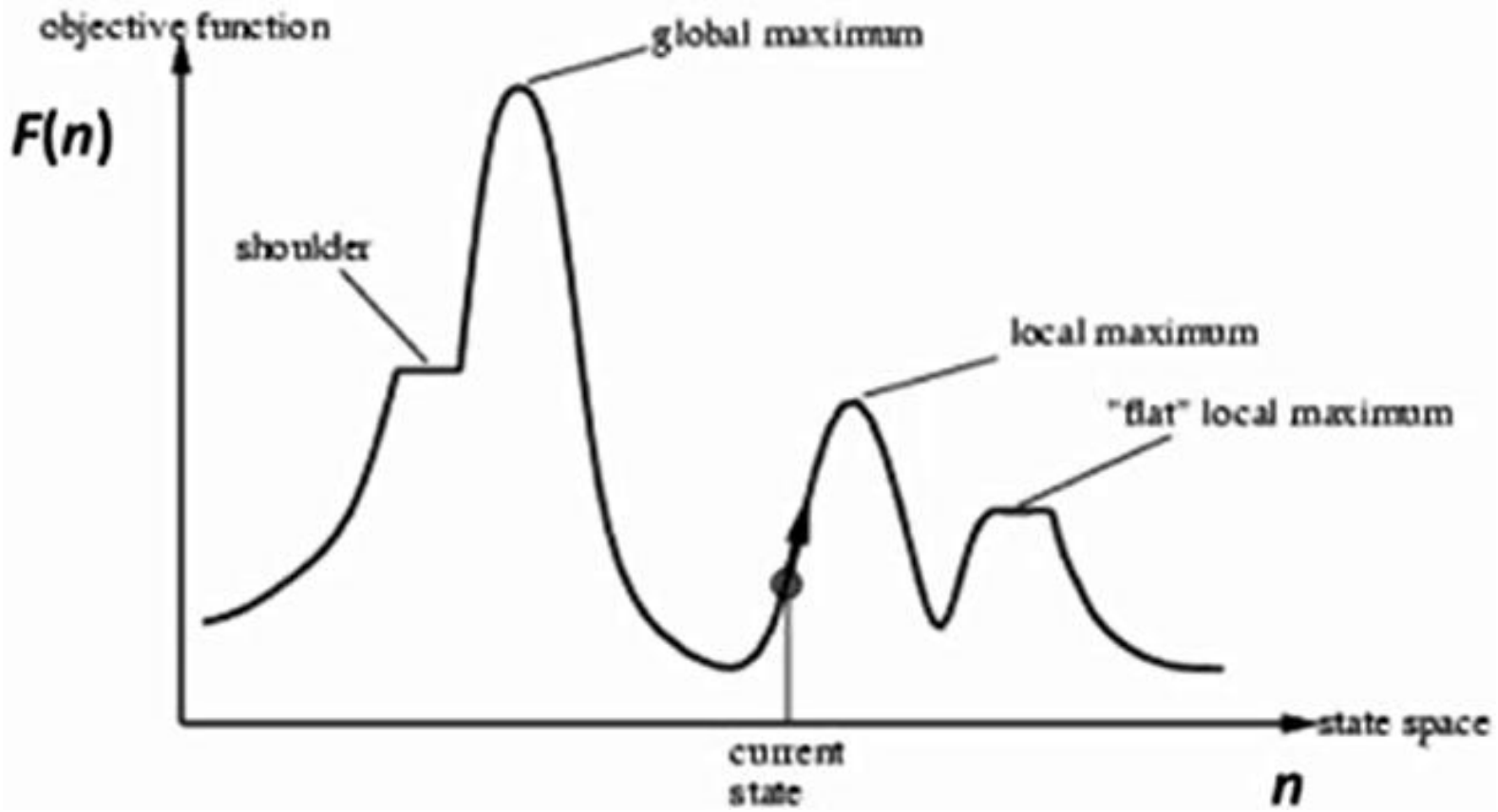
- Procédure :
  - Le nœud courant est initialisé au nœud initial
  - Itérativement, le nœud courant est comparé à ses successeurs immédiats (voisins) :
    - Le meilleur voisin **n'** ayant la plus grande valeur **F(n')** tel que **F(n')>F(n)** devient le nœud courant.
    - Si un tel voisin n'existe pas, on s'arrête et on retourne le nœud courant comme solution.

Algorithme Hill-Climbing

Algorithme HILL-CLIMBING(*noeudInitial*) // *cette variante maximise*

1. déclarer deux nœuds : *n*, *n'*
2. *n* = *noeudInitial*
3. tant que (1) // *la condition de sortie (exit) est déterminée dans la boucle*
   4. *n'* = noeud successeur de *n* ayant la plus grande valeur *F(n')*
   5. si $F(n') \leq F(n)$ // *si on minimisait, le test serait* $F(n') \geq F(n)$
      6. retourner *n* // *on n'arrive pas à améliorer p/r à F(n)*
   7. Sinon *n* = *n'* *(Aller à 3)*

Algorithme Hill-Climbing : illustration



Objet : chercher à arriver au sommet d'une colline en plein brouillard

Hill-Climbing : exemple de simulation

Soit la fonction objective suivante, définie pour les entiers de 1 à 16

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$F(n)$ =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

- Quelle valeur de *n* trouverait la méthode *hill-climbing* si la valeur initiale de *n* était 6 et que les successeurs (voisins) utilisés étaient *n-1* (seulement si *n>1*) et *n+1* (seulement si *n<16*)?

Hill-Climbing : exemple de simulation

Exécution :
Nœud initial : n = 6

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution : Suite des valeurs parcourues

6 → 7

Hill-Climbing : exemple de simulation

Exécution :
Nœud initial : n = 6

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution : Suite des valeurs parcourues

6 → 7 → 8

Hill-Climbing : exemple de simulation

Exécution :
Nœud initial : n = 6

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution : Suite des valeurs parcourues

6 → 7 → 8 → 9

Hill-Climbing : exemple de simulation

Exécution :
Nœud initial : n = 6

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution : Suite des valeurs parcourues

6 → 7 → 8 → 9 → 10

Hill-Climbing : exemple de simulation

Exécution :
Nœud initial : n = 6

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution : Suite des valeurs parcourues

6 → 7 → 8 → 9 → 10 → 11

Hill-Climbing : exemple de simulation

Exécution :
Nœud initial : n = 6

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution : Suite des valeurs parcourues

6 → 7 → 8 → 9 → 10 → 11 → 12

Hill-Climbing : exemple de simulation

Exécution :
Nœud initial : n = 6

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution : Suite des valeurs parcourues

6 → 7 → 8 → 9 → 10 → 11 → 12
Hill-Climbing s'arrête et retourne n = 12

Hill-Climbing : 8-reines

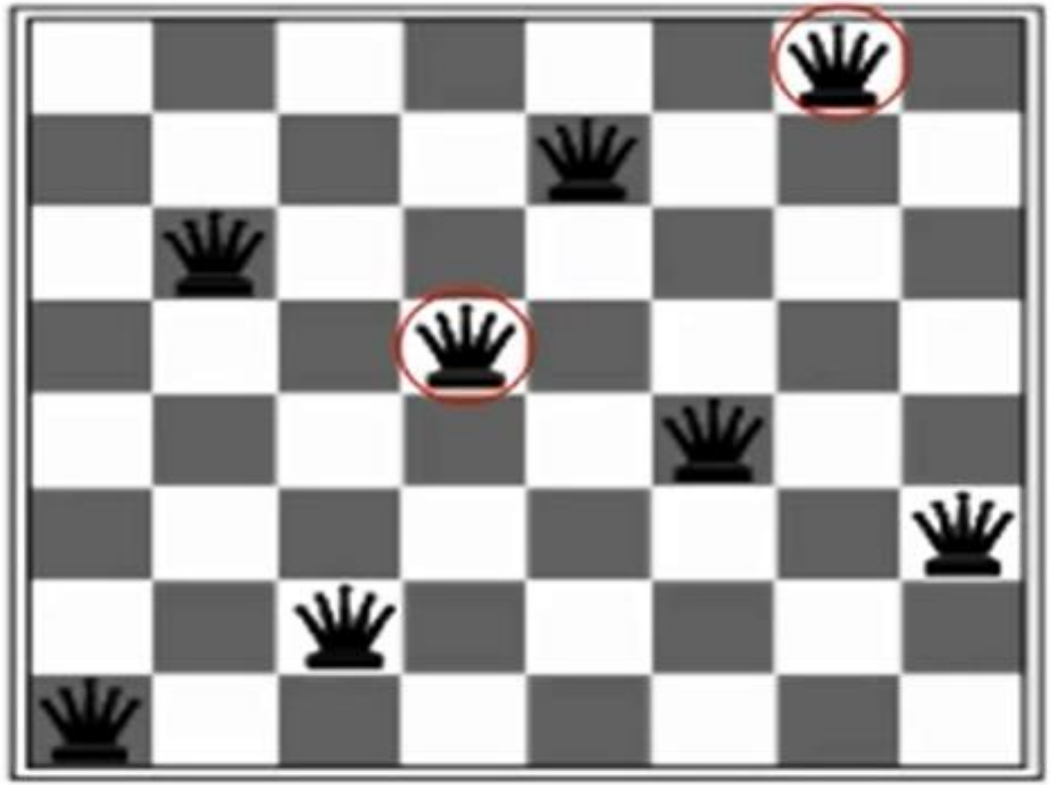
- n : configuration de l'échiquier avec N reines
- $F(n)$: nombre de paires de reines qui s'attaquent mutuellement directement ou indirectement dans la configuration n
- On veut le **minimiser**

- $F(n)$ pour l'état (nœud) affiché : 17
- Encadré : les meilleurs successeurs, si on bouge une reine dans sa colonne

$\boldsymbol{n}$=[5 6 7 4 5 6 7 6]

Hill-Climbing : 8-reines

- Un exemple de minimum local avec $F(n)=1$



Hill-Climbing : jeu de Taquin

Exercice 1

F(n): nombre de chiffres mal placés

3	1	2
4	5	8
6		7

Etat initial

	1	2
3	4	5
6	7	8

Etat but

Hill-Climbing : jeu de Taquin

Exercice 2

4	1	2
3		5
6	7	8

Etat initial

	1	2
3	4	5
6	7	8

Etat but

Méthode Simulated annealing

- Amélioration de Hill-Climbing (minimiser le risque d'être bloqué dans un optimal local)
  - Chercher **un moins bon voisin immédiat** du nœud courant (avec certaine probabilité) au lieu de chercher un meilleur voisin immédiat,
  - La probabilité de prendre un moins bon voisin est plus élevée au début ensuite elle diminue graduellement,
- Le nombre d'itérations et la diminution des probabilités sont définis à l'aide d'un schéma (schedule) de températures en ordre décroissant.
  - Exemple: Schéma de 100 itérations [2^{-0}, 2^{-1}, 2^{-2},.., 2^{-99}]

Méthode Simulated annealing

Algorithme SIMULATED-ANNEALING(*noeudInitial, schema*) // *cette variante maximise*

1. déclarer deux nœuds : n, n'
2. déclarer : $t, T, \Delta E,$
3. n = *noeudInitial*
4. pour $t = 1$ ... taille(*schema*)
   5. T = *schema*[t]
   6. n' = successeur de n choisi au hasard
   7. $\Delta E = F(n') - F(n)$ // *si on minimisait,* $\Delta E = F(n) - F(n')$
   8. si $\Delta E > 0$ alors assigner $n = n'$ // *amélioration p/r à n*
   9. sinon assigner $n = n'$ seulement avec probabilité de $e^{\Delta E / T}$
6. retourner n

plus T est petit,
plus $e^{\Delta E / T}$ est petite

Recherche locale

Amélioration de Simulated annealing

- Simulated annealing minimise le risque d'être piégé dans des optima locaux **mais** il n'élimine pas le risque d'osciller indéfiniment en revenant à un nœud antérieurement visité.

- Solution1: Algorithme **Tabu search**
  - Enregistrer les ***k*** derniers nœuds visités (ensemble taboue)

- Solution 2 : Algorithme **Beam search**
  - Au lieu de maintenir un seul nœud solution, on pourrait maintenir ***k*** nœuds différents (faisceau):
    - Commencer avec ***k*** nœuds choisi aléatoirement
    - A chaque itération, générer tous les successeurs des k nœuds choisis
    - Choisir les k meilleurs nœuds parmi les nœuds générés et recommencer

Algorithmes génétiques

Origine

- Inspiré du processus de l'évolution naturelle des espèces :
- L'intelligence humaine est le résultat d'un processus d'évolution sur des millions d'années :
  - Théorie de l'évolution de Darwin
  - Théorie de la sélection naturelle de Weismann
  - Concepts de génétiques de Mendel
- La simulation de l'évolution n'a pas besoin de durer des millions d'années sur un ordinateur

Algorithmes génétiques

Principe

- On commence avec un ensemble de ***k*** nœuds choisis aléatoirement : cet ensemble est appelé ***population.***
- Un successeur est généré en combinant deux parents.
- Un nœud est représenté par une chaine (mot) sur un alphabet : c'est le code génétique d'un nœud.
- La fonction objective est appelée ***fitness function*** (fonction d'adaptation)
- La prochaine génération est produite par :

(1)Sélection → (2)Croisement → (3)Mutation

Algorithmes génétiques

Représentation

- On représente l'espace des solutions d'un problème à résoudre par une population (ensemble de ***chromosomes***).
  - Un chromosome est une chaine de caractères (***gènes***) de taille fixe. Par exemple : **101101001**
- Une population génère des enfants par un ensemble de procédures simples qui manipulent des chromosomes :
  - *Croisement des parents*
  - *Mutation d'un enfant généré*
- Les enfants sont conservés en fonction de leur adaptation (fitness) déterminée par une fonction d'adaptation ***F(n)***

Algorithmes génétiques

Pseudocode

Algorithme ALGORITHME-GÉNÉTIQUE(k, *nb_iterations*) // *cette variante maximise*

1. *population* = ensemble $\{n_1, n_2, ..., n_k\}$ généré aléatoirement de k chromosomes
2. pour $t = 1$ ... *nb_iterations*
3. *nouvelle_population* = {}
4. pour $i = 1$ ... k
5. n = chromosome pris dans *population* avec probabilité qui augmente selon $F(n)$
6. n' = chromosome différent pris dans *population* – { n } de la même façon
7. n^* = résultat du croisement entre n et n'
8. avec petite probabilité, appliquer une mutation à n^*
9. ajouter n^* à *nouvelle_population*
10. *population* = *nouvelle_population*
11. retourner n dans *population* avec valeur de $F(n)$ la plus élevée

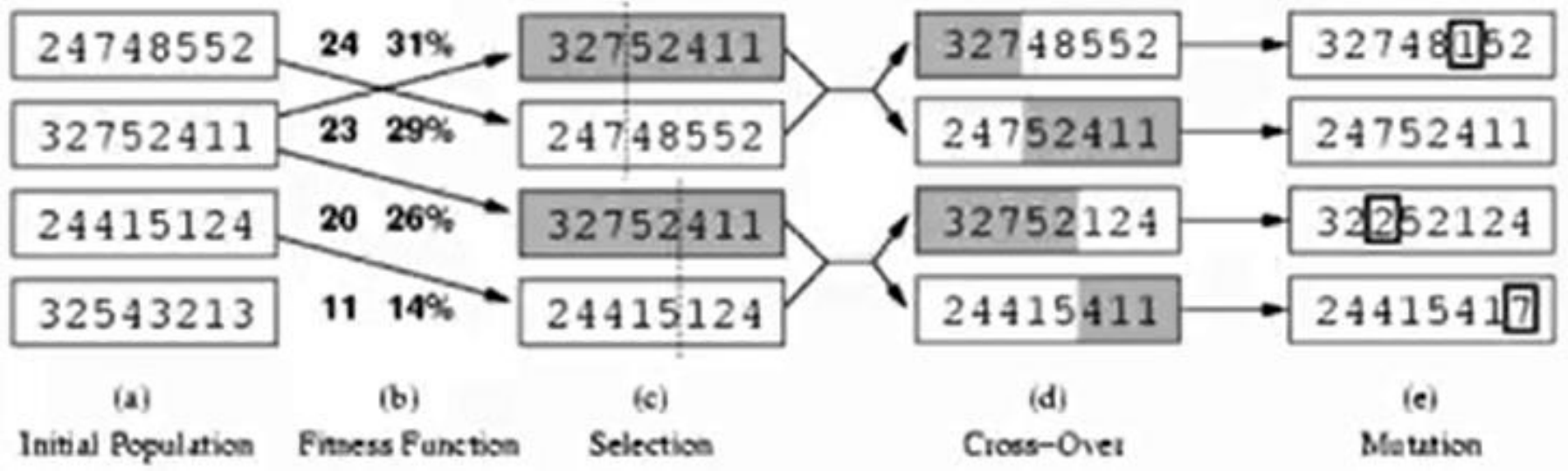
Algorithme génétique

Exemple d'un croisement : 8-reines

67247588 + 75251448 = 67251448

Algorithme génétique

8-reines



- Fonction d'adaptation : nombre de reines qui ne s'attaquent pas (min=0, max=28).
- Probabilité de sélection du premier chromosome (proportionnelle à l'adaptation) :
  - 24/(24+23+20+11) = 31%
  - 23/(24+23+20+11) = 29%
  - 20/(24+23+20+11) = 26%
  - 11/(24+23+20+11) = 14%

Vers les jeux avec adversité

- Est-il possible d'utiliser A* dans les jeux à deux adversaires ?
  - ✓ On pourrait définir un état pour le jeu (Echecs: position de toutes les pièces)
  - ✓ L'état but est la configuration de l'échiquier tel qu'un joueur gagne
  - × Quelle serait la fonction de transition ?

- Oui mais pas directement (Passer par des états finaux intermédiaires)
  - Environnement multi-agent (Le joueur adverse peut modifier l'environnement)

Types de jeux

- Jeu Coopératif
  - Les joueurs veulent atteindre le même but

- Jeu avec adversité
  - Les joueurs sont en compétition
  - Un gain pour les uns est une perte pour les autres (ou match nul)
  - Cas particulier : Jeu à somme nulle (Zero-sum game)
    - Exemples: Echecs, Tic-Tac-Toe,..

On suppose : - Jeux à deux adversaires qui jouent à tour de rôle
- Jeux à somme nulle
- Environnement déterministe (sans hasard)et complètement observable

Algorithme MiniMax

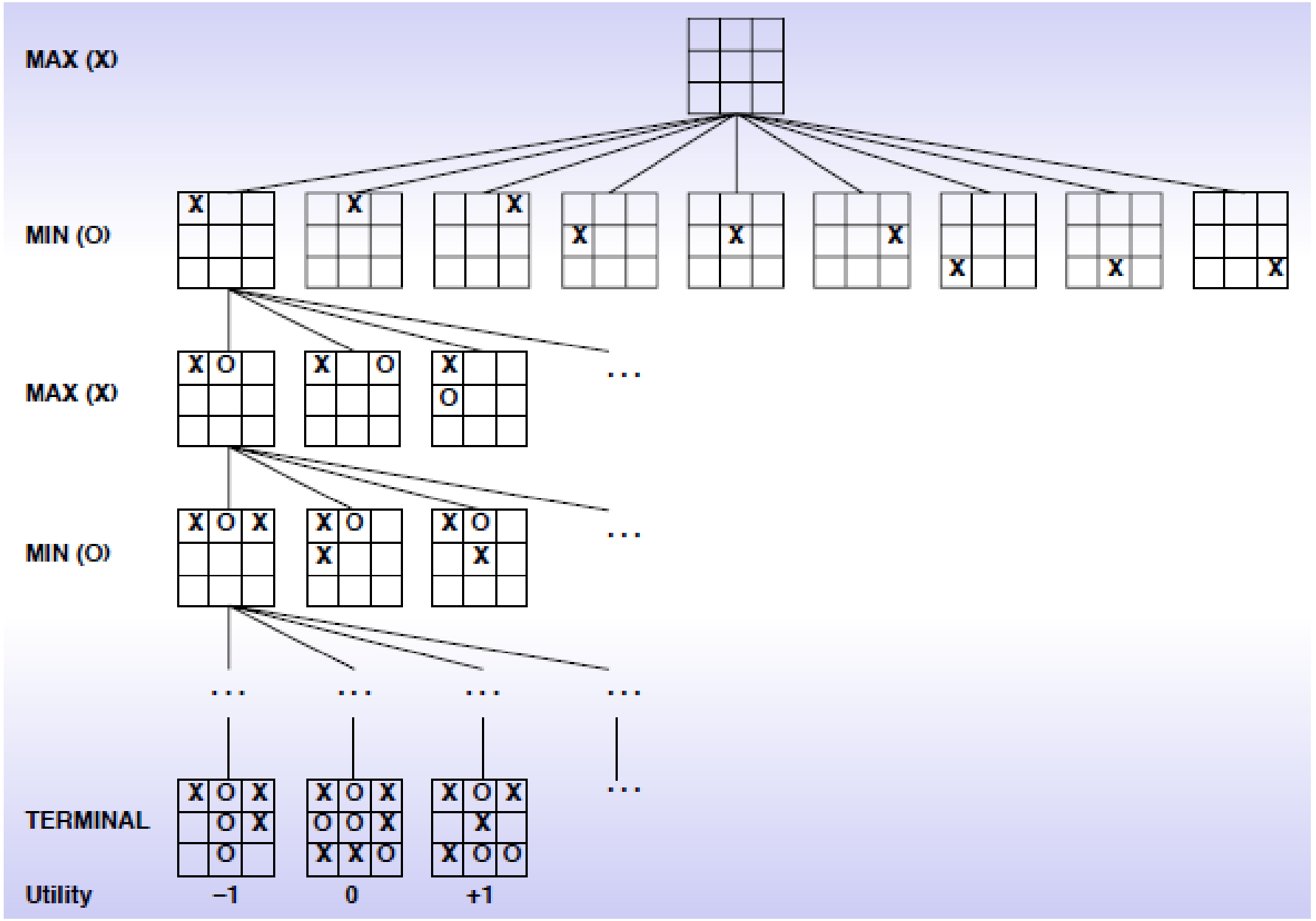
- Deux joueurs : Max vs Min
- Max est le premier à jouer
- On considère le résultat d'une partie comme une récompense distribuée au joueur Max
  - Max cherche à maximiser la récompense
  - Min cherche à minimiser la récompense de Max

Algorithme MiniMax

- Le problème de jeu est vu comme un problème de recherche dans un arbre :
  - Un **nœud initial** (configuration initiale du jeu)
  - Une **fonction de transition** (retourne des paires **(action, nœud successeur)**)
  - Un **test de terminaison** (indique si le jeu est terminé)
  - Une **fonction d'utilité** pour les états finaux (Récompense reçue par Max)

Jeux à deux adversaires

Arbre de recherche Tic-Tac-Toe



Algorithme MiniMax

- On suppose que l'action la plus profitable pour le joueur Max ou Min est prise (Obtenir la **plus grande valeur MiniMax**)

$$\text{VALEUR-MINIMAX}(n) = \begin{cases} \text{UTILITÉ}(n) & \text{Si } n \text{ est un nœud terminal} \\ \max_{n' \text{ successeur de } n} \text{VALEUR-MINIMAX}(n') & \text{Si } n \text{ est un nœud Max} \\ \min_{n' \text{ successeur de } n} \text{VALEUR-MINIMAX}(n') & \text{Si } n \text{ est un nœud Min} \end{cases}$$

- Le calcul des valeurs minimax pour tous les nœuds de l'arbre de recherche est fait en utilisant un programme récursif,

Algorithme MiniMax

Algorithme MiniMax(nœud initial)

- Retourne l'action choisie par TOUR-MAX(nœud initial)

TOUR-MAX(n){

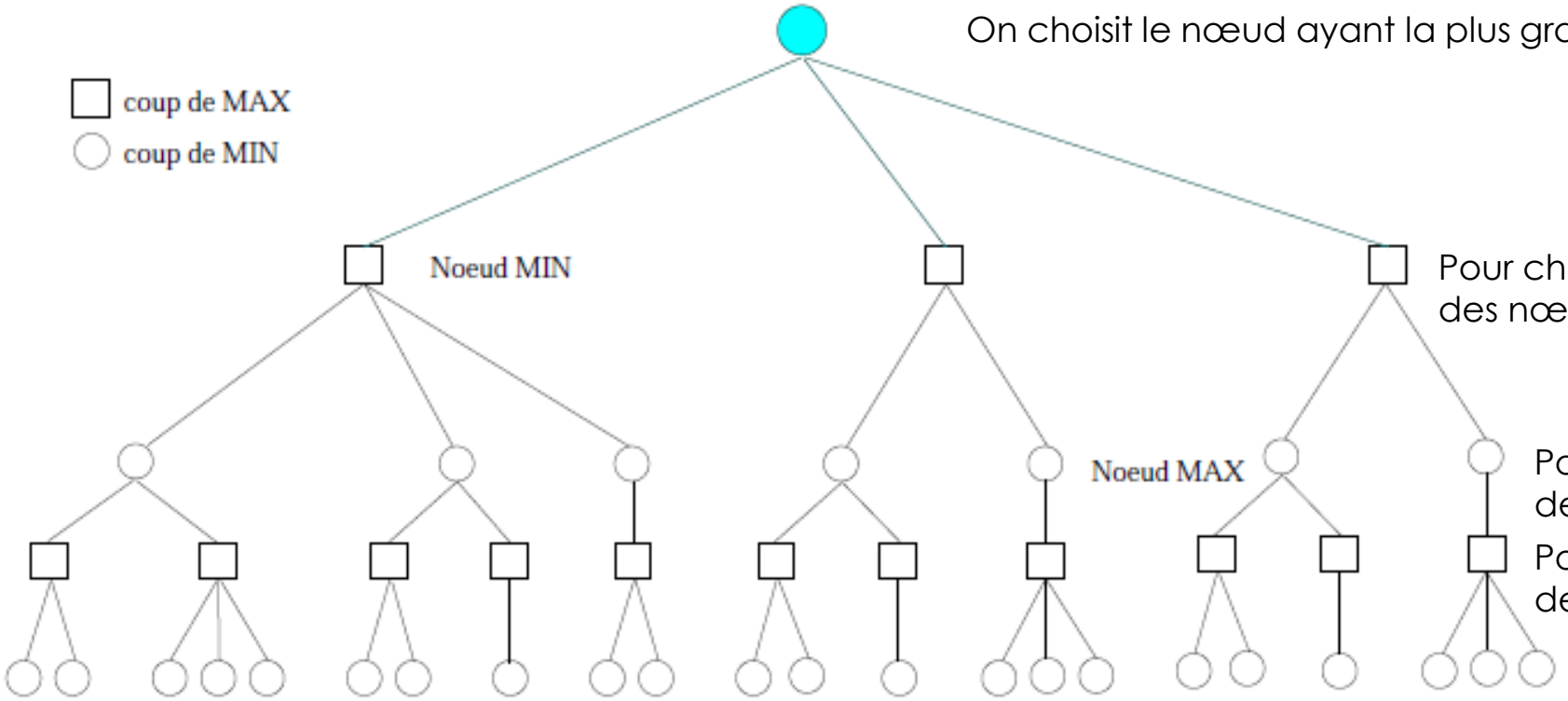
1. Si n correspond à une fin de partie Alors retourner la valeur d'utilité UTILITY(n)
2. U=-∞ , a=void
3. Pour chaque paire(a', n') donnée par TRANSITION(n)
   - Si l'utilité de TOUR-MIN(n') > u Alors affecter a=a' , u=utilité de TOUR-MIN(n')
     Sinon Retourner l'utilité u et l'action a }

TOUR-MIN(n){

1. Si n correspond à une fin de partie Alors retourner la valeur d'utilité UTILITY(n)
2. U=+∞ , a=void
3. Pour chaque paire(a', n') donnée par TRANSITION(n)
   - Si l'utilité de TOUR-MAX(n') < u Alors affecter a=a' , u=utilité de TOUR-MAX(n')
     Sinon Retourner l'utilité u et l'action a}

Jeux à deux adversaires

Algorithme MiniMax / Arbre de jeu



On choisit le nœud ayant la plus grande valeur

Pour chacun de ces nœuds on donne le min des nœuds fils

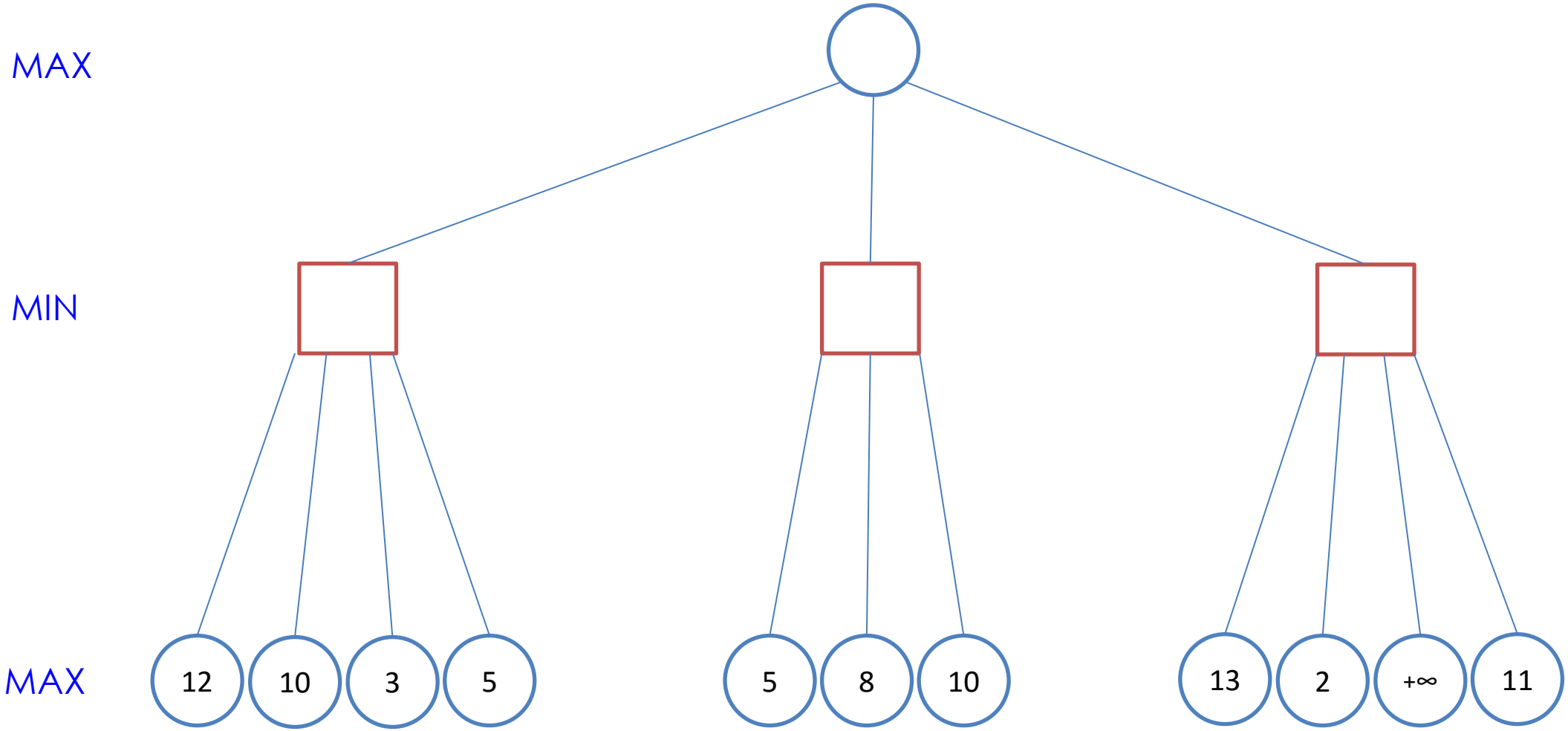
Pour chacun de ces nœuds on donne le max des nœuds fils

Pour chacun de ces nœuds on donne le min des nœuds fils

Pour chacun de ces nœuds on calcule l'utilité U

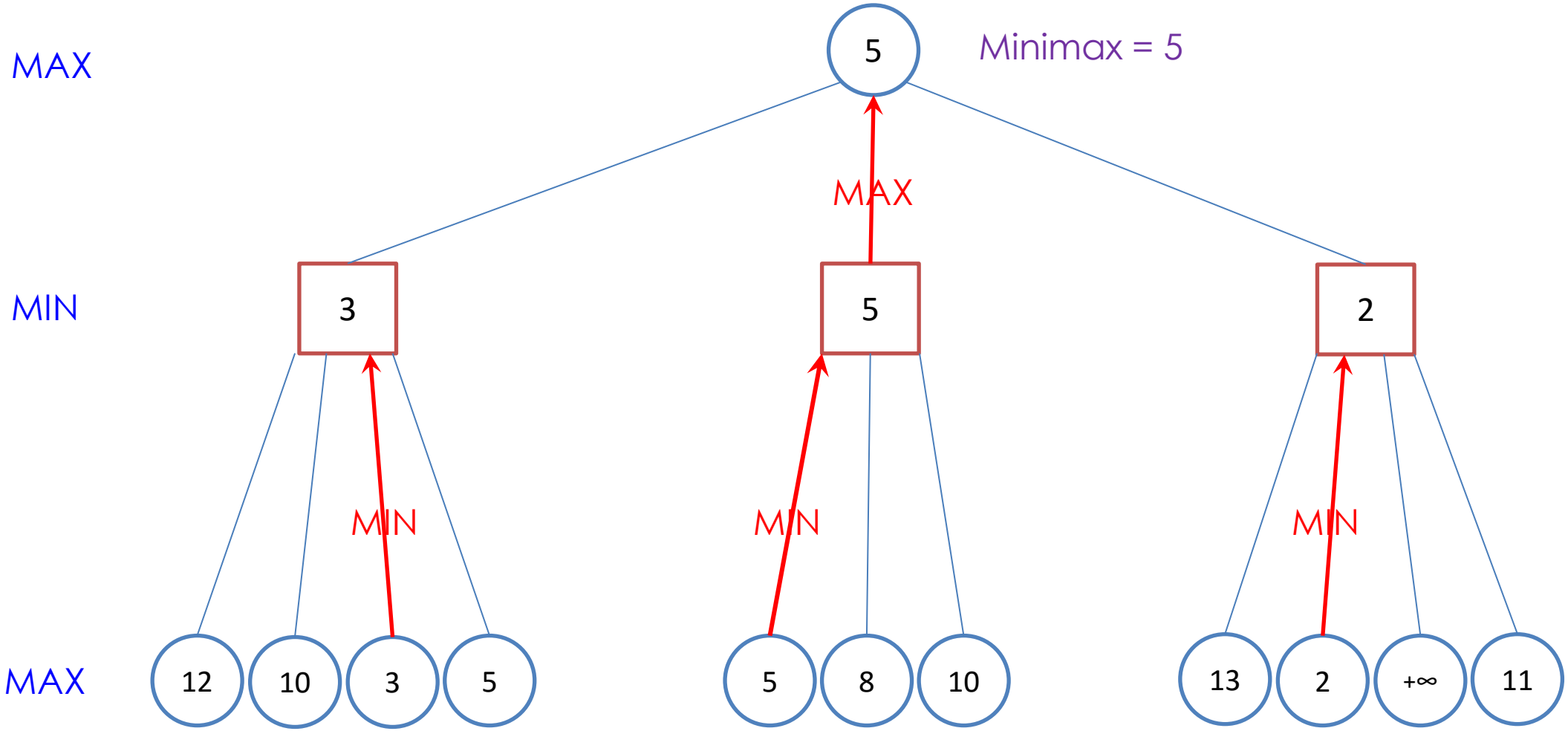
Jeux à deux adversaires

Algorithme MiniMax / Déroulement



Jeux à deux adversaires

Algorithme MiniMax / Déroulement



Jeux à deux adversaires

Algorithme MiniMax / Complexité

- Complexité en temps = $O(b^m)$
- b : nombre maximum de choix par coup (choix ou actions) à chaque étape (facteur de branchement)
- m : nombre maximum de coups dans un jeu (nombre de niveaux dans une recherche en profondeur)

- Complexité en mémoire = O(bm)

Jeu d'échecs :

Nombre de choix par coup : 35 (b ≈ 35)

Nombre moyen de coups par joueur : 50 (m ≈ 100)

Jeux à deux adversaires

Coupure Alpha-Béta

- MINIMAX : Evaluation des positions après génération de l'arbre (Développer toutes les feuilles à une profondeur limite)

- Idée : Evaluation aux feuilles et propagation aux ancêtres lorsque l'arbre est généré (Coupure ou Elagage Alpha-Béta (Alpha-Béta Pruning)):
  - Une feuille est évaluée dès qu'elle est produite.
  - identifier les chemins (dans l'arbre) qui sont explorés inutilement (savoir si une feuille est inintéressante)

Jeux à deux adversaires

Coupure Alpha-Béta

Algorithme Coupure-Alpha-Beta(nœud initial)

- Retourne l'action choisie par TOUR-MAX(nœud initial, -∞ , +∞)

TOUR-MAX(n,α, β) {

1. Si n correspond à une fin de partie Alors retourner la valeur d'utilité UTILITY(n)
2. U=-∞ , a=void
3. Pour chaque paire(a', n') donnée par TRANSITION(n)
   - Si l'utilité de TOUR-MIN(n', α, β) > U Alors affecter a=a' , U=utilité de TOUR-MIN(n' , α, β)

     Si U ≥ β Retourner l'utilité U et l'action a

     Sinon α = Max (α , U)
4. Retourner l'utilité U et l'action a,

   }

TOUR-MIN(n,α, β) {

1. Si n correspond à une fin de partie Alors retourner la valeur d'utilité UTILITY(n)
2. U=+∞ , a=void
3. Pour chaque paire(a', n') donnée par TRANSITION(n)
   - Si l'utilité de TOUR-MAX(n', α, β) < U Alors affecter a=a' , U=utilité de TOUR-MAX(n' , α, β)

     Si U ≤ α Retourner l'utilité U et l'action a

     Sinon β = Min (β , U)
4. Retourner l'utilité U et l'action a,

   }

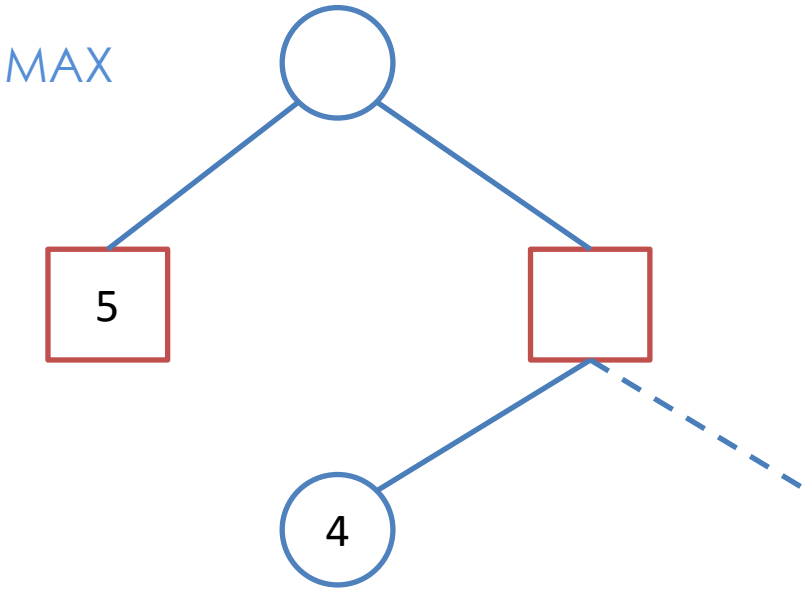
Jeux à deux adversaires

Coupure Alpha-Béta

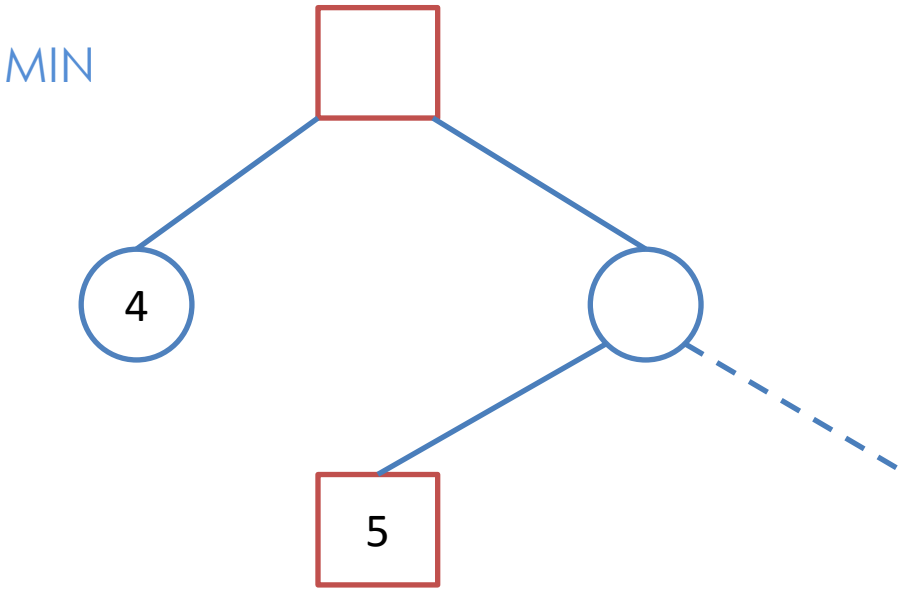
Principe de la coupure:

- On ajoute les paramètres α et β (initialement $-\infty$ et $+\infty$)
- Les nœuds coupés (élagués) sont ceux tel que $u(n) \in [\alpha, \beta]$ et $\alpha \geq \beta$
- Les nœuds non coupés sont ceux tel que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \boldsymbol{ou} \\ [-\infty, \boldsymbol{b}] \quad \boldsymbol{avec\ b} \neq +\infty & \boldsymbol{ou} \\ [\boldsymbol{a}, +\infty] \quad \boldsymbol{avec\ a} \neq -\infty \end{cases}$$



Coupure α



Coupure β

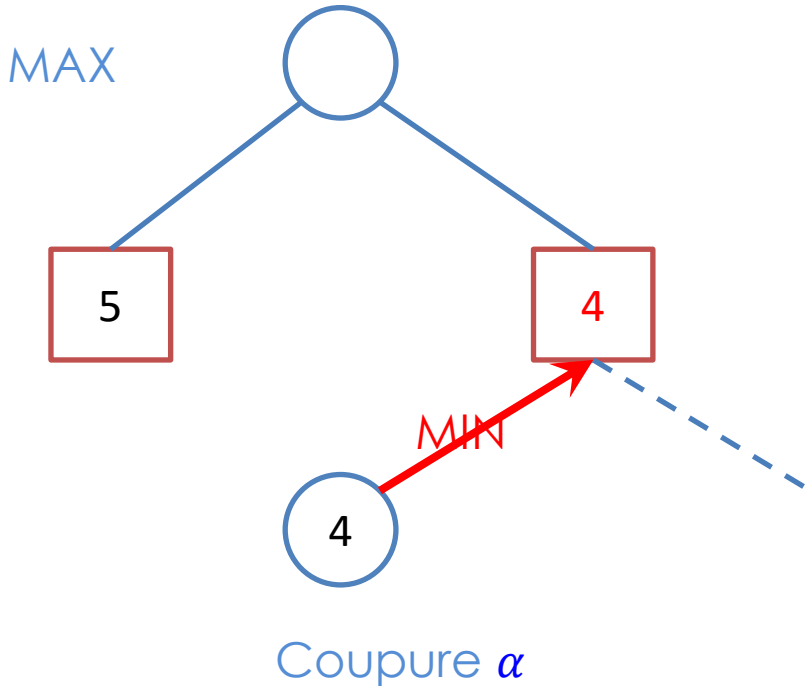
Jeux à deux adversaires

Coupure Alpha-Béta

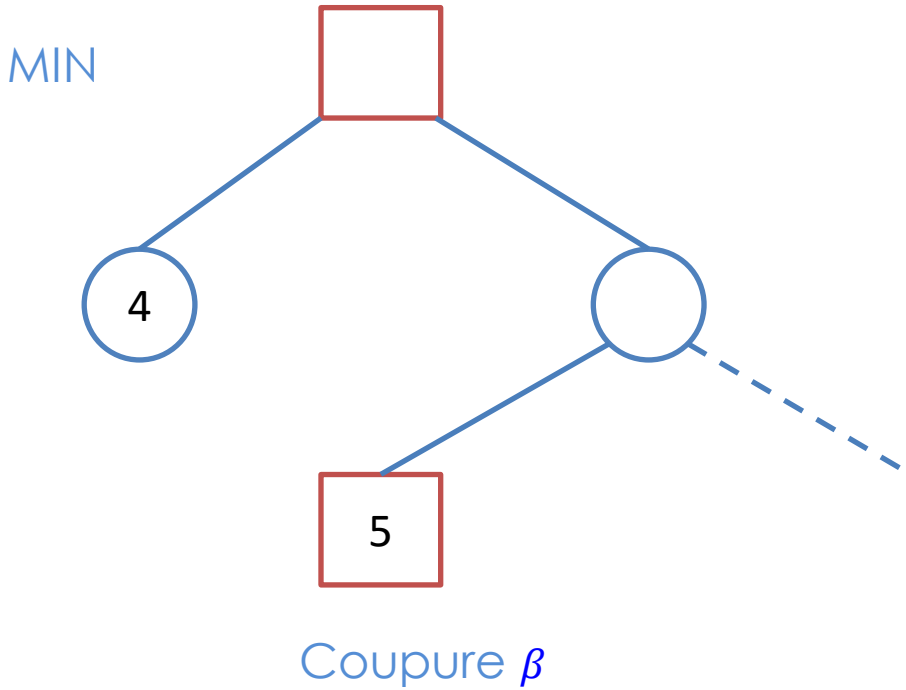
Principe de la coupure:

- On ajoute les paramètres α *et* β (initialement $-\infty$ *et* $+\infty$)
- Les nœuds coupés (élagués) sont ceux tel que $u(n) \in [\alpha, \beta]$ *et* $\alpha \geq \beta$
- Les nœuds non coupés sont ceux tel que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \textbf{ou} \\ [-\infty, b] \quad \textbf{avec } b \neq +\infty & \textbf{ou} \\ [a, +\infty] \quad \textbf{avec } a \neq -\infty \end{cases}$$



Coupure α



Coupure β

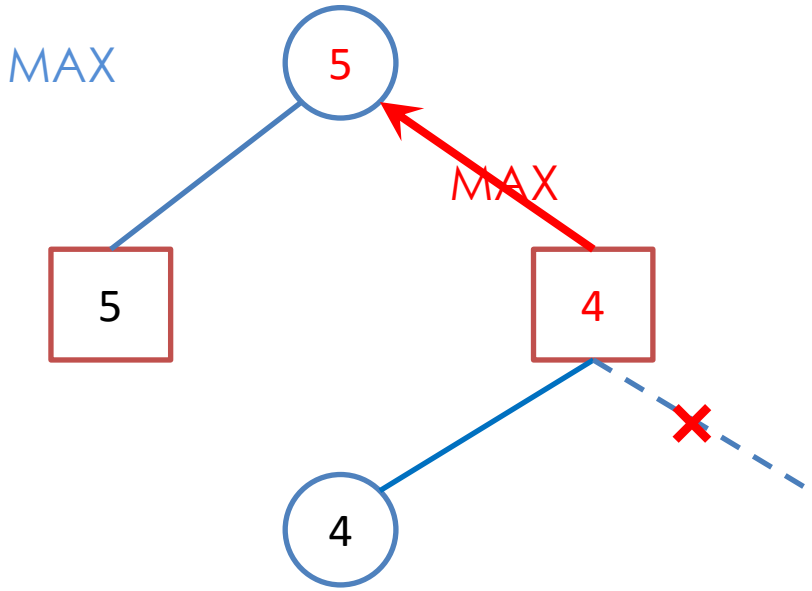
Jeux à deux adversaires

Coupure Alpha-Béta

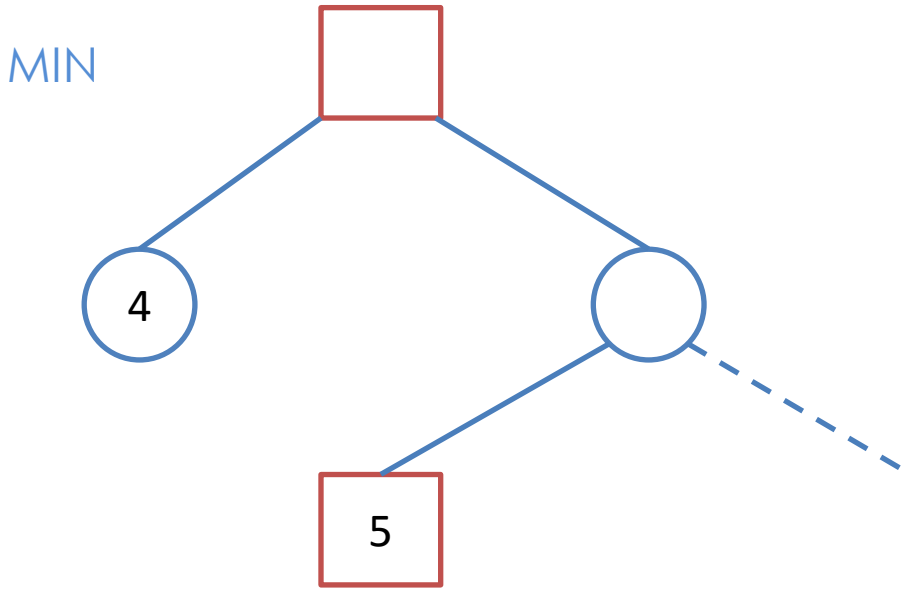
Principe de la coupure:

- On ajoute les paramètres α *et* β (initialement $-\infty$ *et* $+\infty$)
- Les nœuds coupés (élagués) sont ceux tel que $u(n) \in [\alpha, \beta]$ *et* $\alpha \geq \beta$
- Les nœuds non coupés sont ceux tel que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \textbf{ou} \\ [-\infty, b] \quad \textbf{avec } b \neq +\infty & \textbf{ou} \\ [a, +\infty] \quad \textbf{avec } a \neq -\infty \end{cases}$$



Coupure α



Coupure β

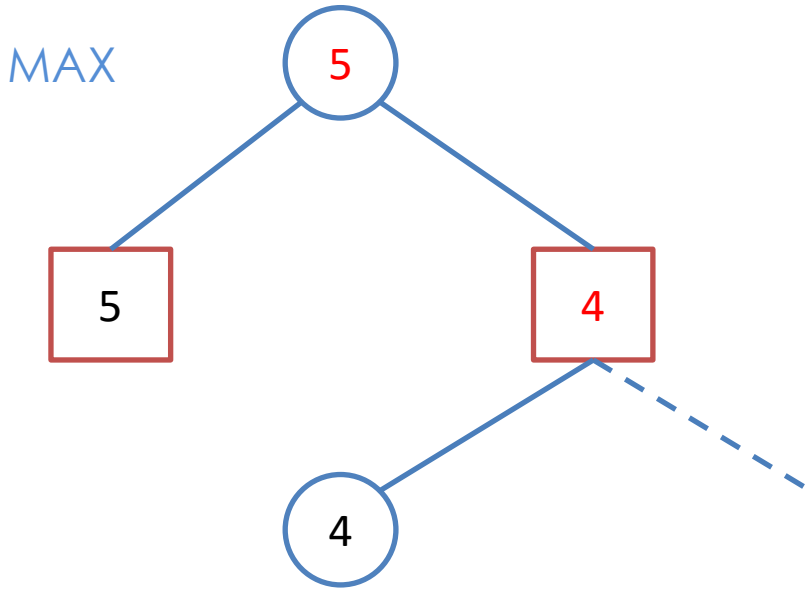
Jeux à deux adversaires

Coupure Alpha-Béta

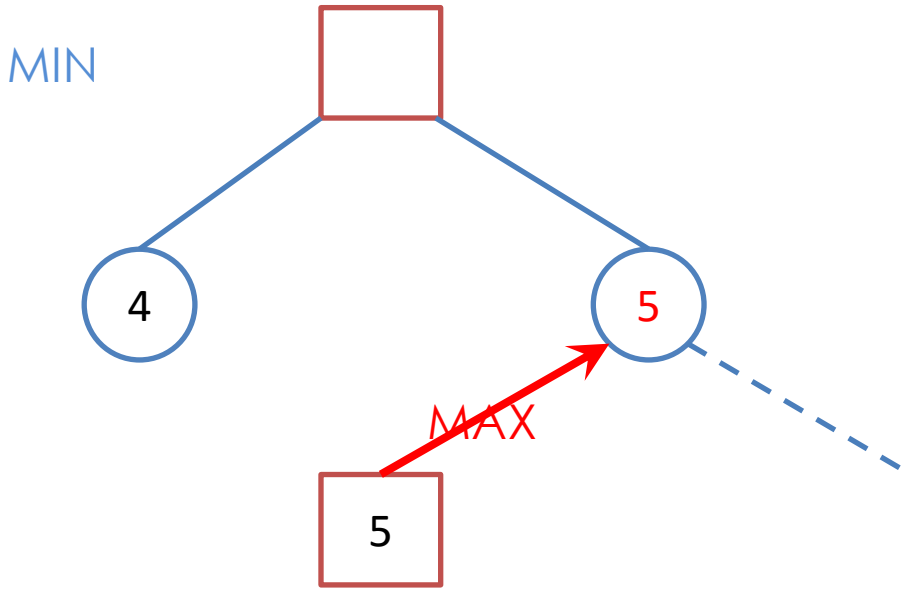
Principe de la coupure:

- On ajoute les paramètres α *et* β (initialement $-\infty$ *et* $+\infty$)
- Les nœuds coupés (élagués) sont ceux tel que $u(n) \in [\alpha, \beta]$ *et* $\alpha \geq \beta$
- Les nœuds non coupés sont ceux tel que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \textbf{ou} \\ [-\infty, b] \quad \textbf{avec } b \neq +\infty & \textbf{ou} \\ [a, +\infty] \quad \textbf{avec } a \neq -\infty \end{cases}$$



Coupure α



Coupure β

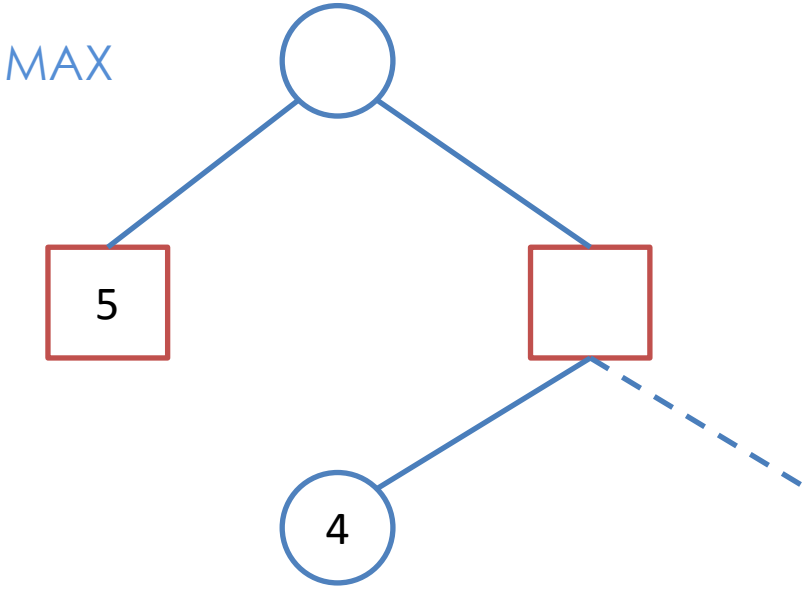
Jeux à deux adversaires

Coupure Alpha-Béta

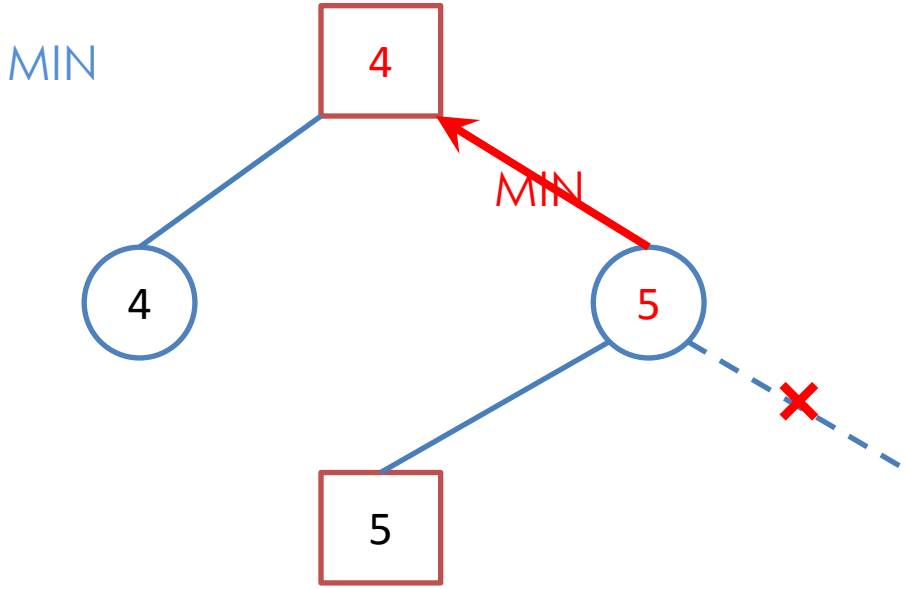
Principe de la coupure:

- On ajoute les paramètres α *et* β (initialement $-\infty$ *et* $+\infty$)
- Les nœuds coupés (élagués) sont ceux tel que $u(n) \in [\alpha, \beta]$ *et* $\alpha \geq \beta$
- Les nœuds non coupés sont ceux tel que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \textbf{ou} \\ [-\infty, b] \quad \textbf{avec } b \neq +\infty & \textbf{ou} \\ [a, +\infty] \quad \textbf{avec } a \neq -\infty \end{cases}$$



Coupure α

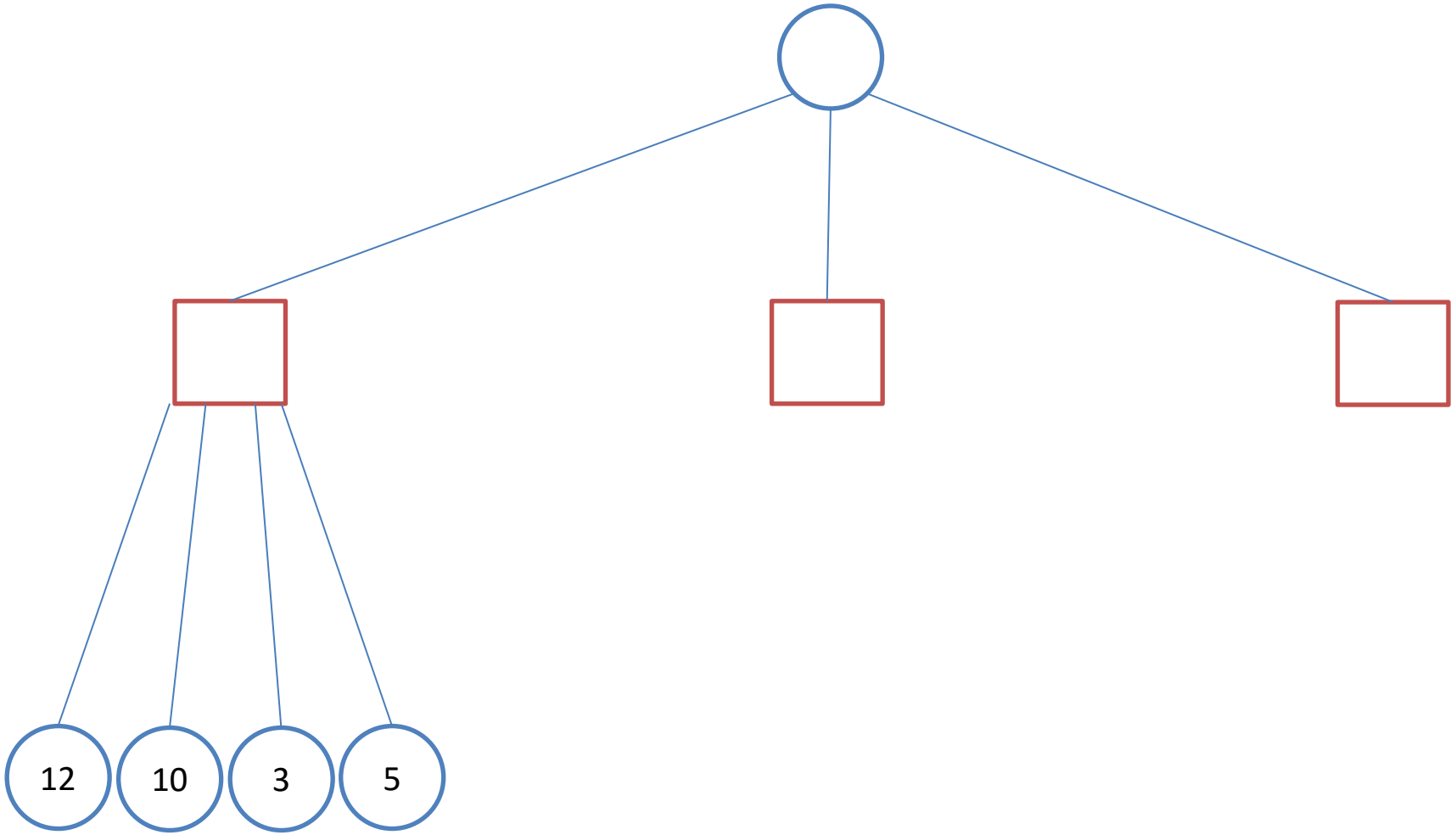


Coupure β

Jeux à deux adversaires

Coupure Alpha-Béta

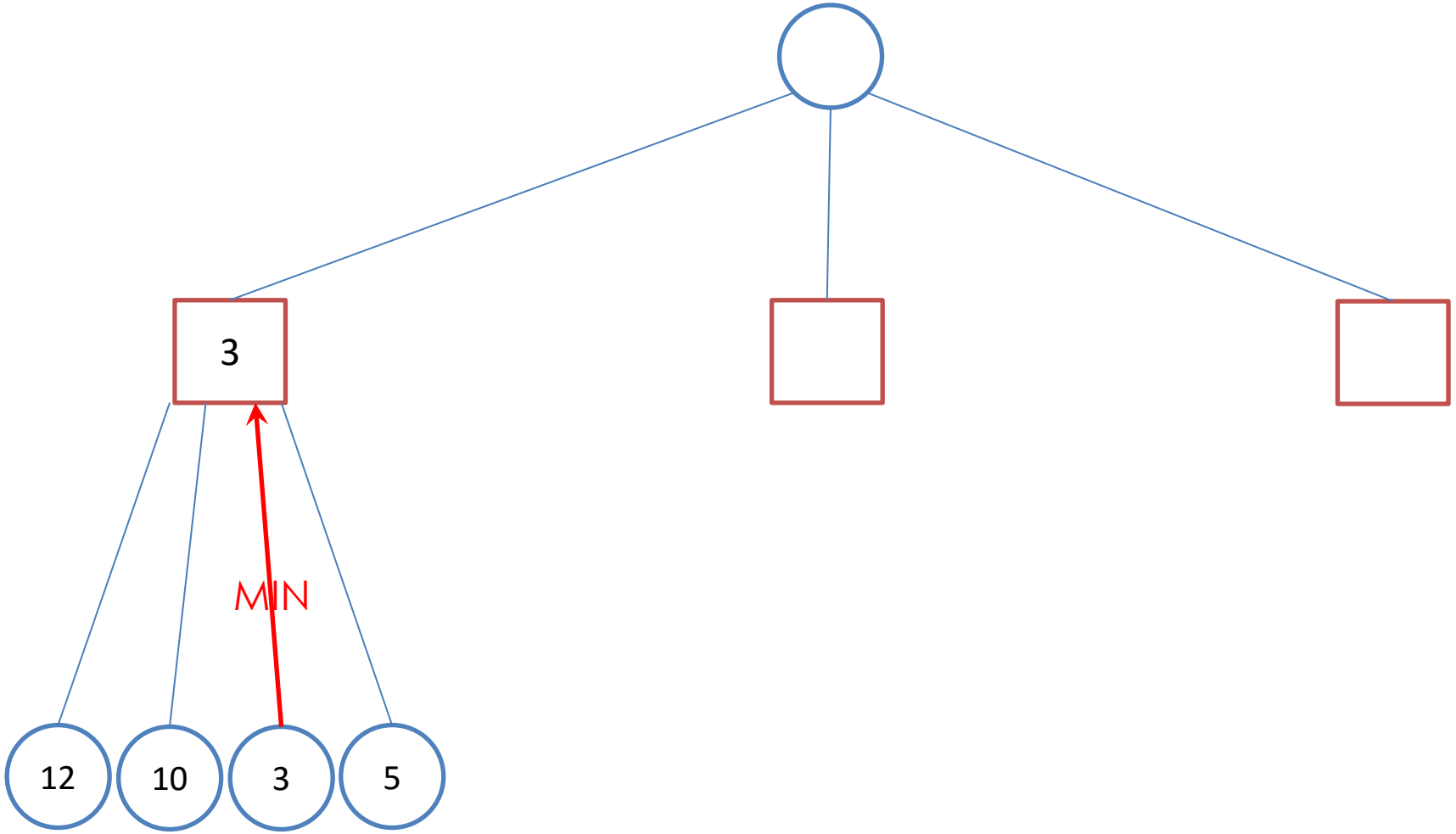
$[\alpha, \beta] = -\infty, +\infty$



Jeux à deux adversaires

Coupure Alpha-Béta

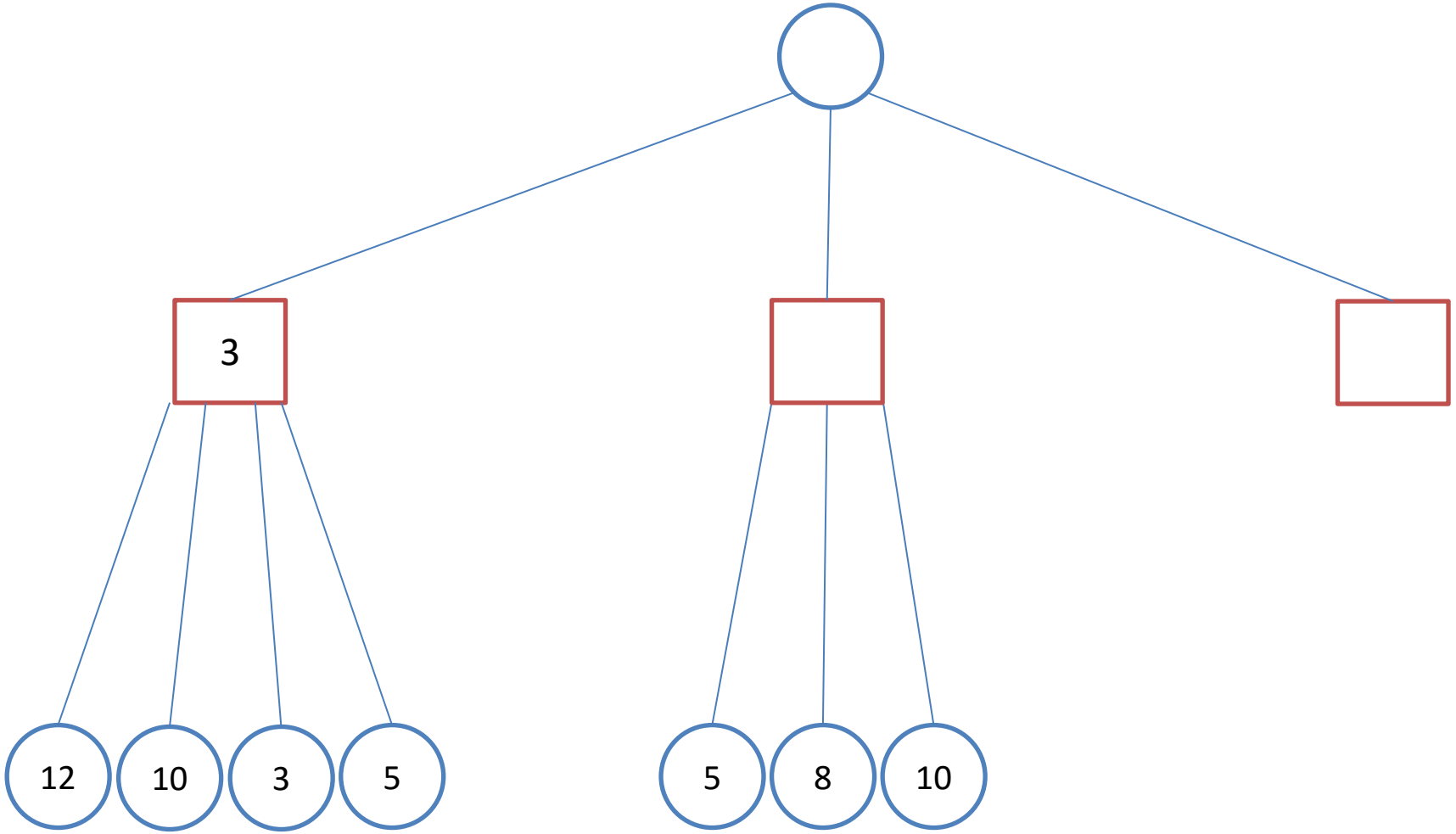
$[\alpha, \beta] = [3, +\infty]$



Jeux à deux adversaires

Coupure Alpha-Béta

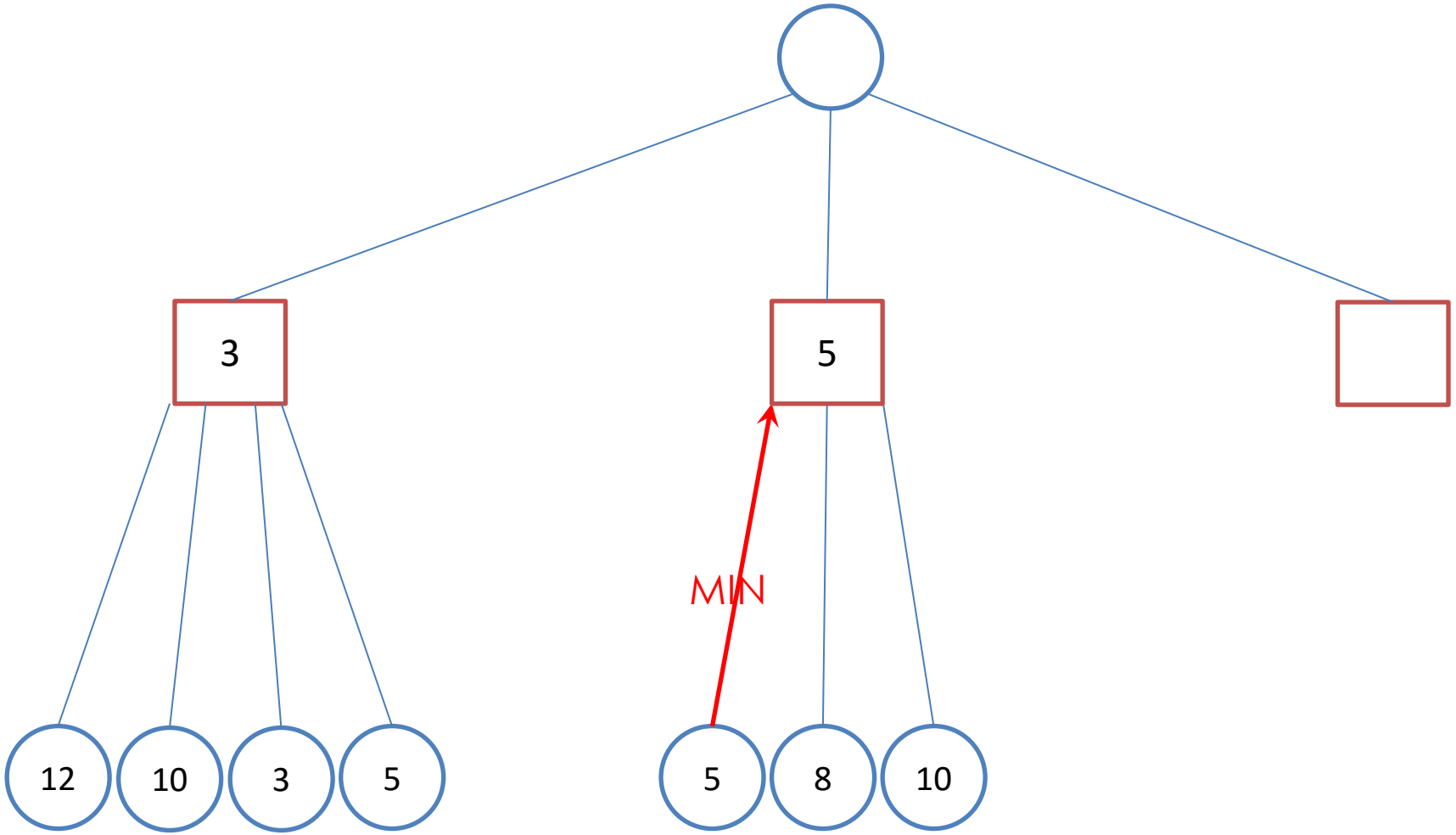
$[\alpha, \beta] = [3, +\infty]$



Jeux à deux adversaires

Coupure Alpha-Béta

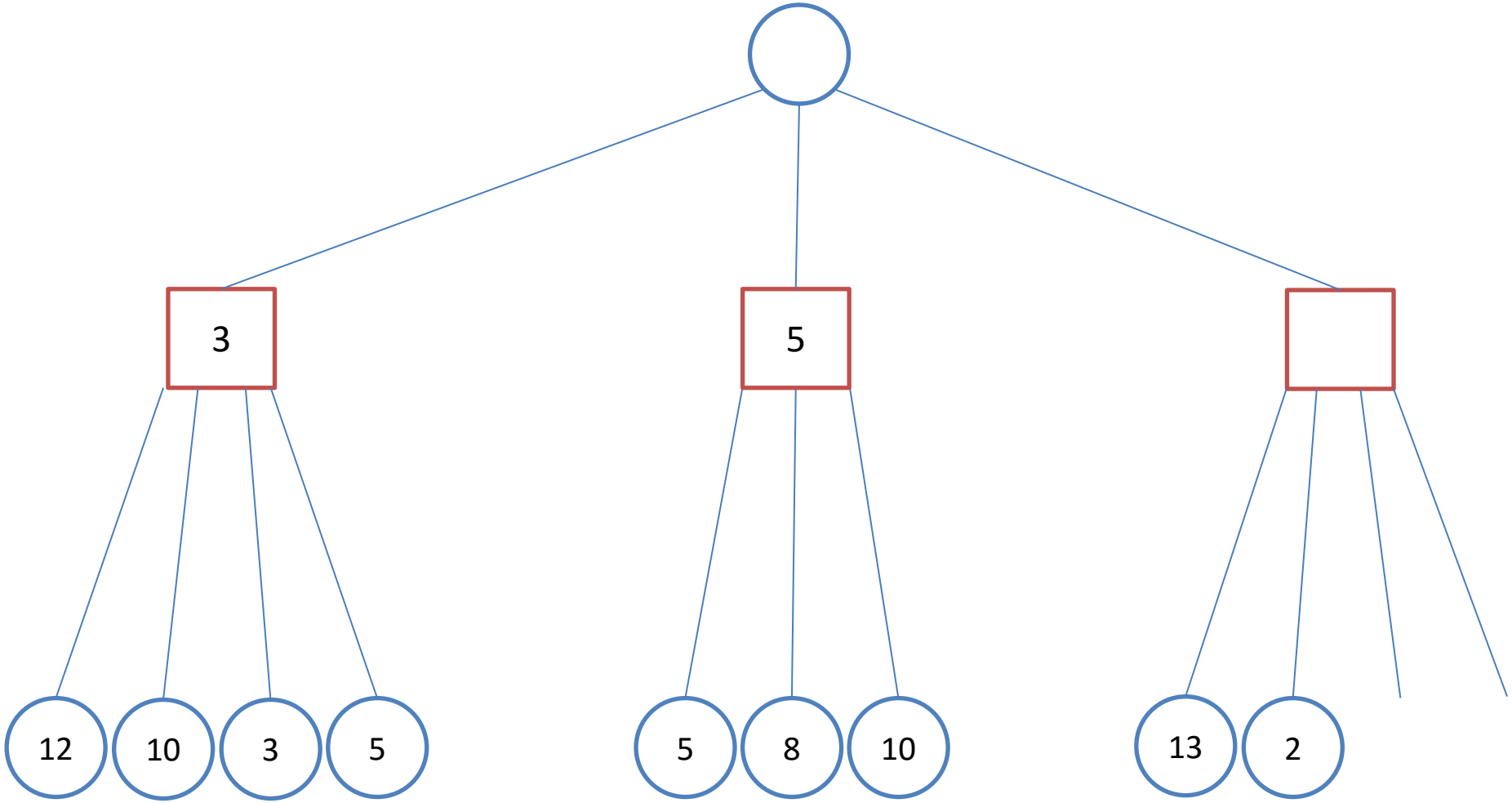
$[\alpha, \beta] = 5, +\infty$



Jeux à deux adversaires

Coupure Alpha-Béta

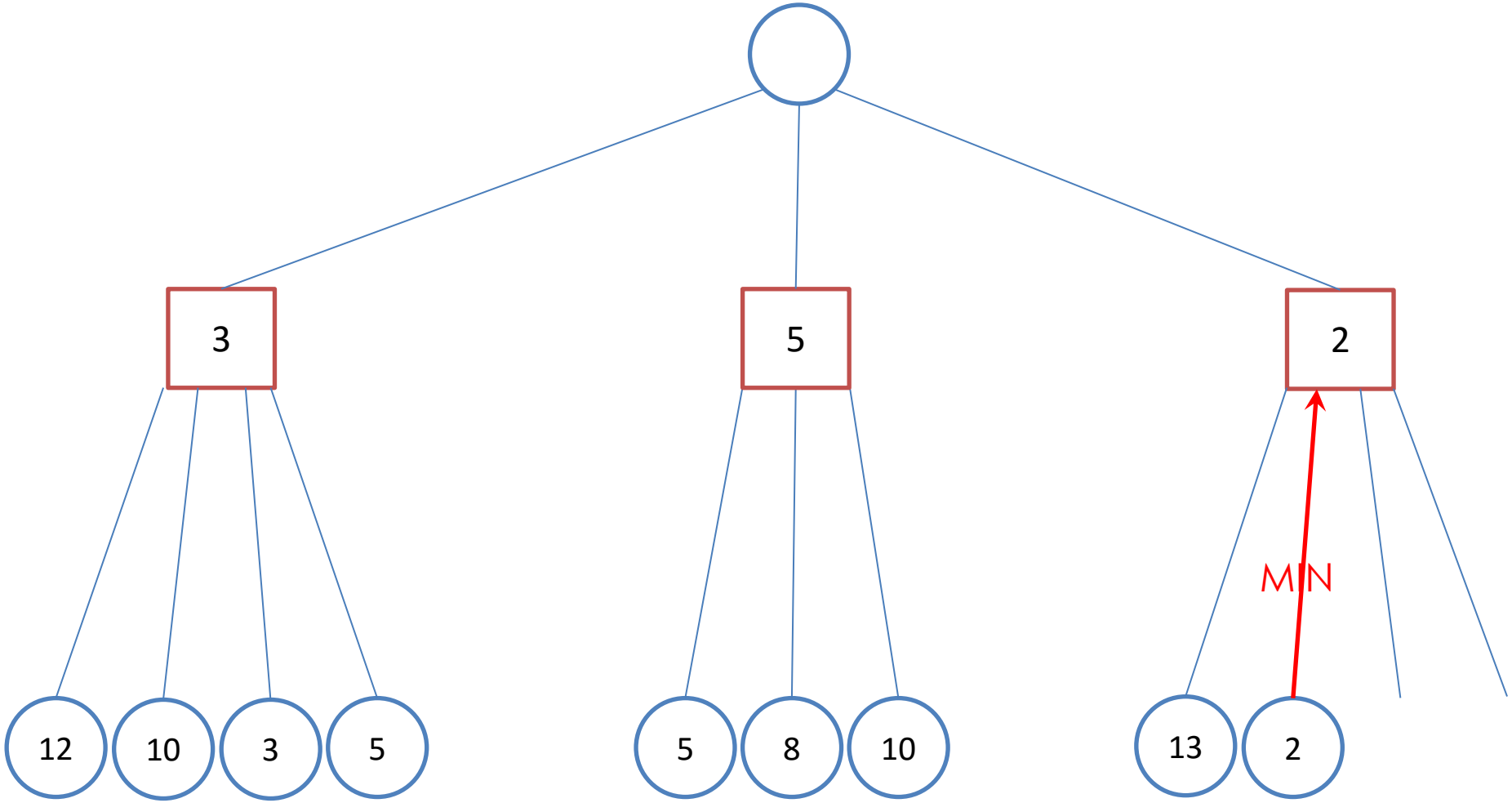
$[\alpha, \beta] = 5, +\infty$



Jeux à deux adversaires

Coupure Alpha-Béta

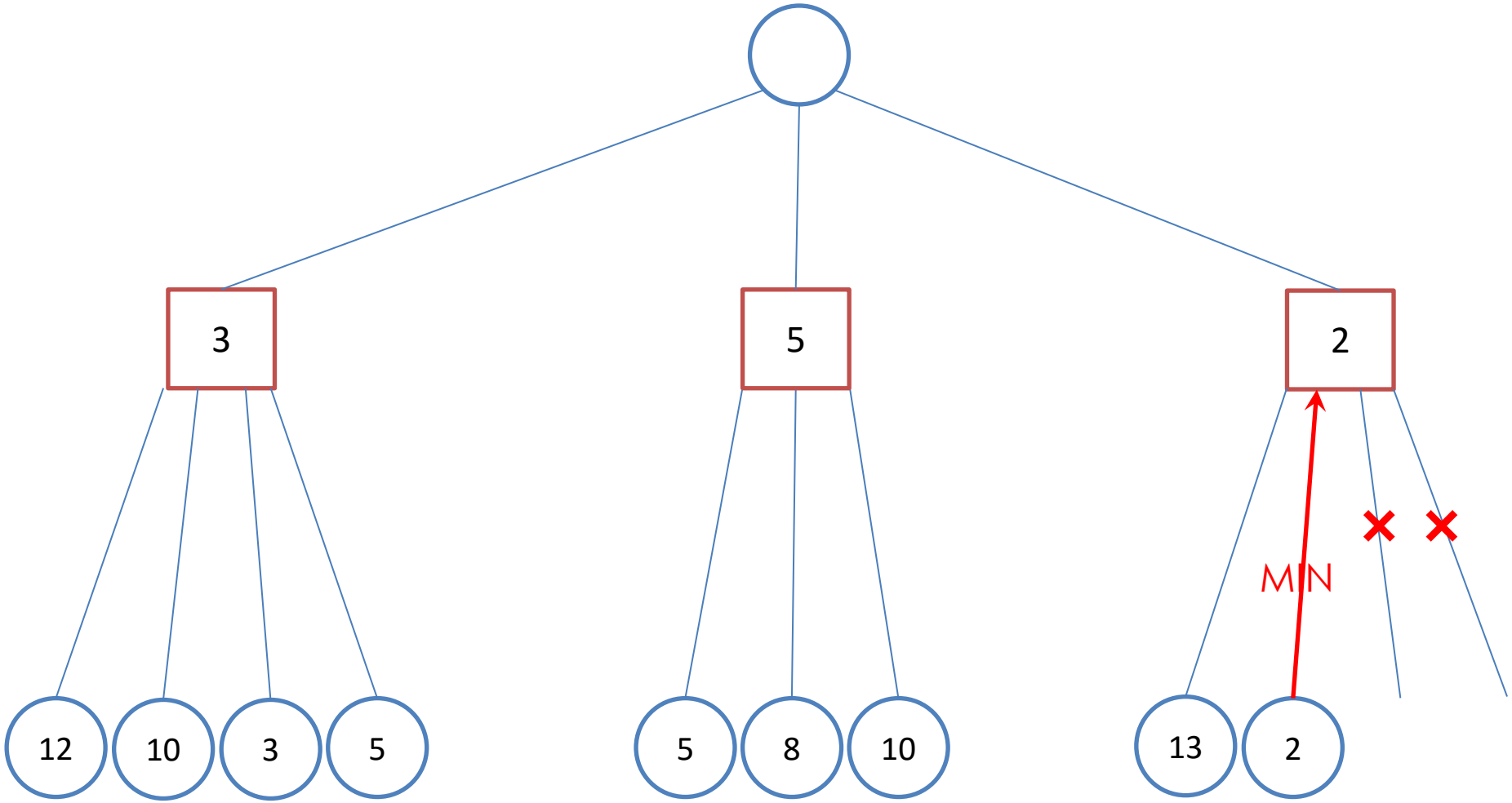
$[\alpha, \beta] = 5, +\infty$



Jeux à deux adversaires

Coupure Alpha-Béta

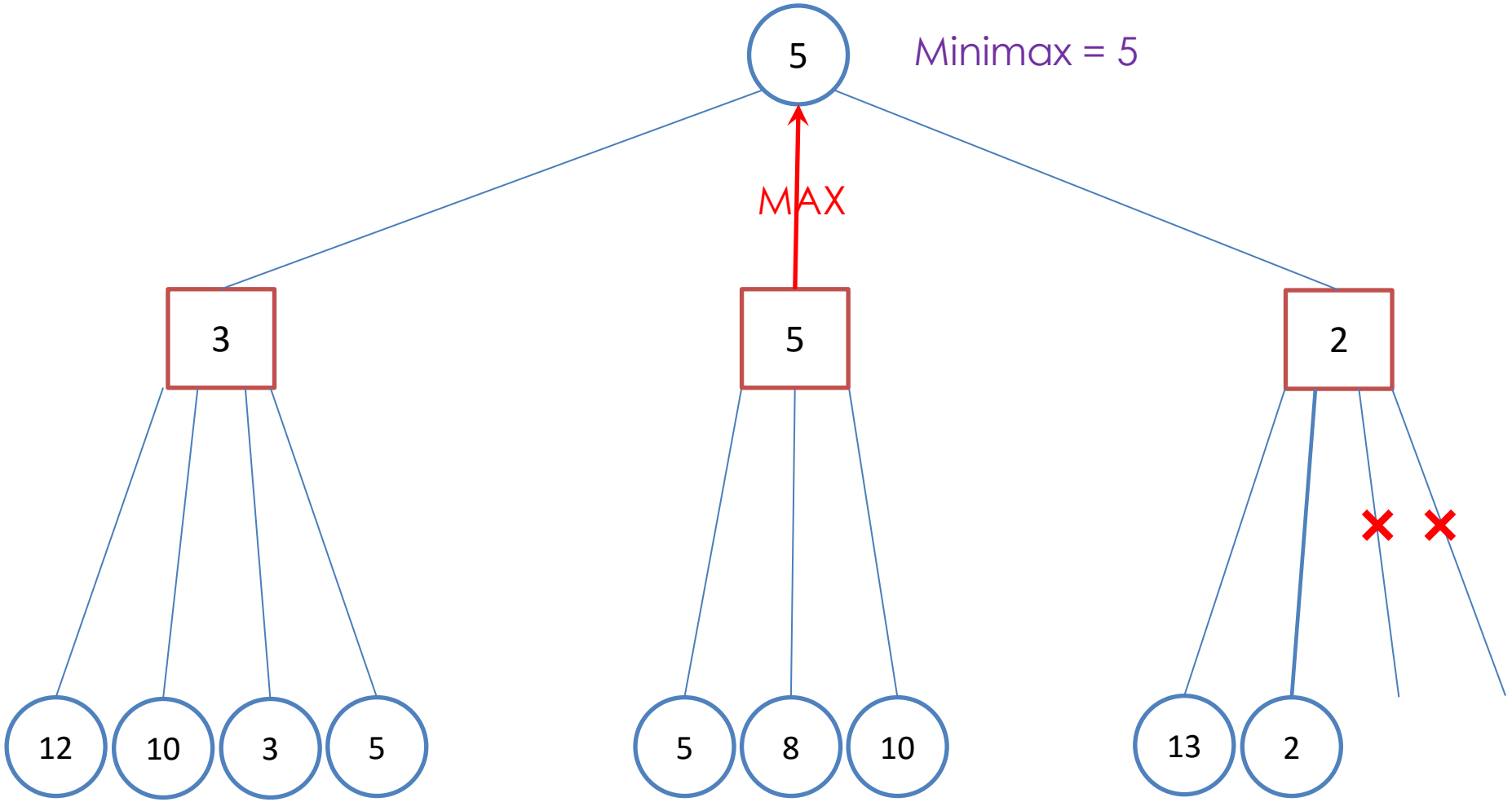
$[\alpha, \beta] = 5, +\infty$



Jeux à deux adversaires

Coupure Alpha-Béta

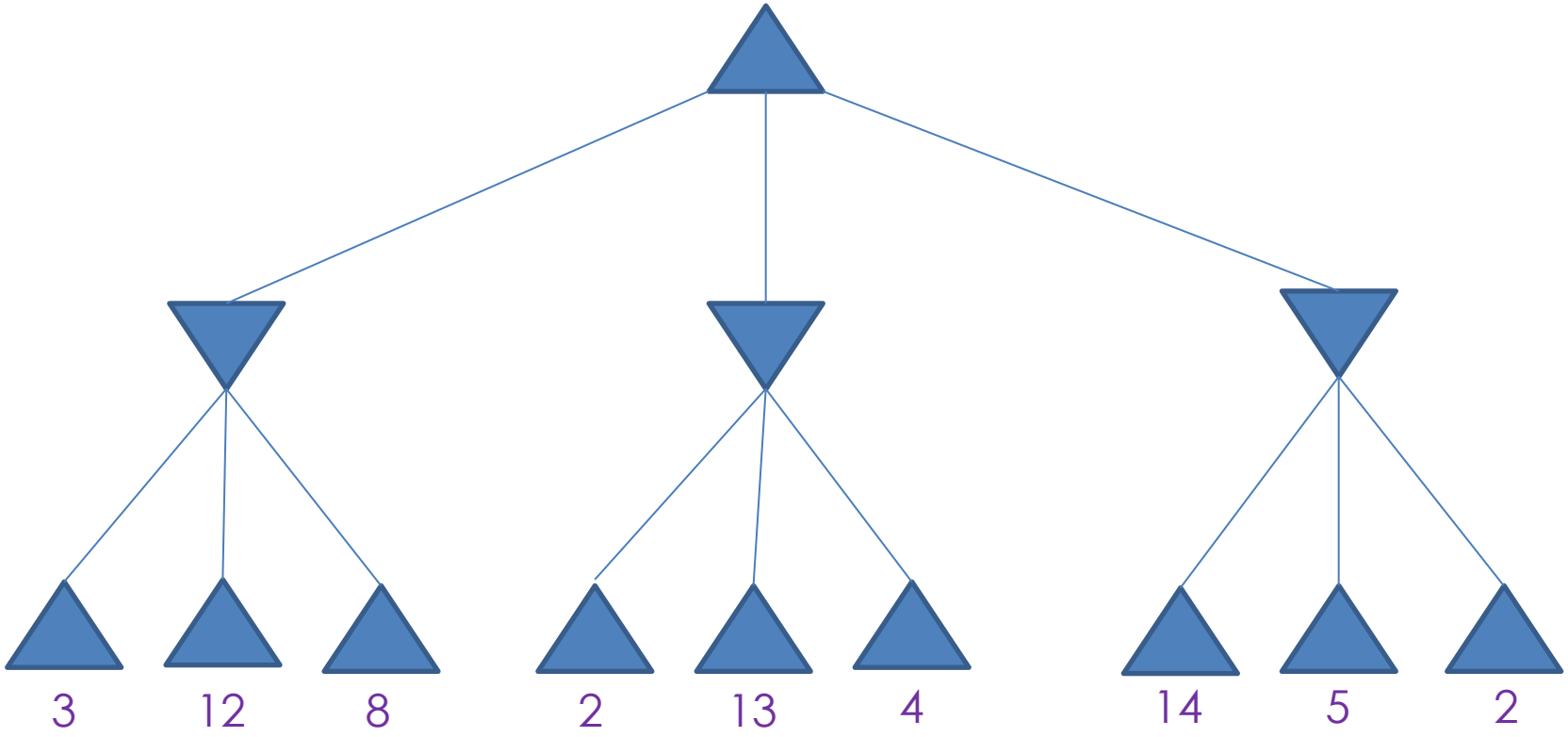
$[\alpha, \beta] = 5, +\infty$



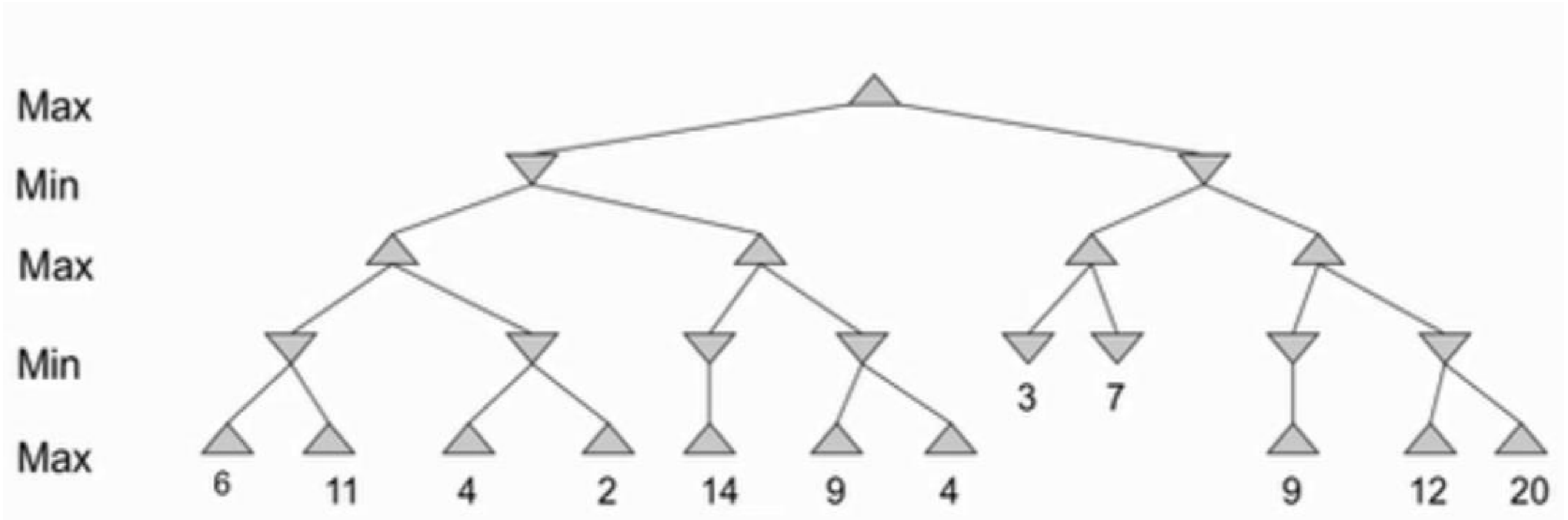
Jeux à deux adversaires

MinMax

Coupure Alpha-Béta



MinMax ? Coupure Alpha-Béta ?



Problèmes de Satisfaction des contraintes (CSP)

Définition

La programmation par contrainte (CSP : Contraint Satisfaction Problems) est à l'interface de l'Intelligence Artificielle et de la Recherche Opérationnelle. Elle s'intéresse aux problèmes définis en termes de contraintes de temps, d'espace, ... ou plus généralement de ressources :

- **Applications :**

•**les problèmes de planification et ordonnancement** : planifier une production, gérer un trafic ferroviaire, ...

•**les problèmes d'affectation de ressources** : établir un emploi du temps, allouer de l'espace mémoire, du temps cpu par un système d'exploitation, affecter du personnel à des tâches, des entrepôts à des marchandises, ...

•**les problèmes d'optimisation**: problèmes des routages de réseaux de télécommunication, ...

Problèmes de Satisfaction des contraintes (CSP)

Définition

La résolution d'un problème CSP peut être vue comme un cas particulier de la recherche heuristique :

- La structure interne des nœuds a une représentation particulière :
  - Un nœud est un ensemble de **variables** avec des **valeurs** correspondantes
  - Les transitions entre nœuds tiennent comptent de **contraintes** sur les **valeurs** possibles des **variables**
- On utilise des **heuristiques générales** plutôt qu'une heuristique spécifique à une application :
  - Dans un problème CSP on élimine la difficulté de définir une heuristique ***h*** pour notre application

Problèmes de Satisfaction des contraintes (CSP)

Exemple

- **Allocation de ressources**
  - EXP : Elaboration d'emploi du temps
  - Variables : les différentes plages horaires de tous les locaux (classes, amphis,…)
  - Les contraintes : un seul cours est assigné au même local à un instant donné, aucun groupe n'a deux cours en même temps,…

Problèmes de Satisfaction des contraintes (CSP)

Définition formelle

- Un ensemble fini de variables $V = \{X_1,\ldots,X_N\}$
  - Chaque variable X_i a un domaine D_i de valeurs possibles
- Un ensemble fini de contraintes : $C = \{C_1,\ldots,C_M\}$
- Un état d'un problème CSP est défini par une assignation de valeurs $\{X_i=v_i, X_j=v_j, \ldots\}$
  - Une assignation qui ne viole aucune contrainte est dite **compatible (consistante)**
  - Une assignation est dite **complète** si elle assigne des valeurs à toutes les variables
  - Une solution est une assignation **complète et compatible**

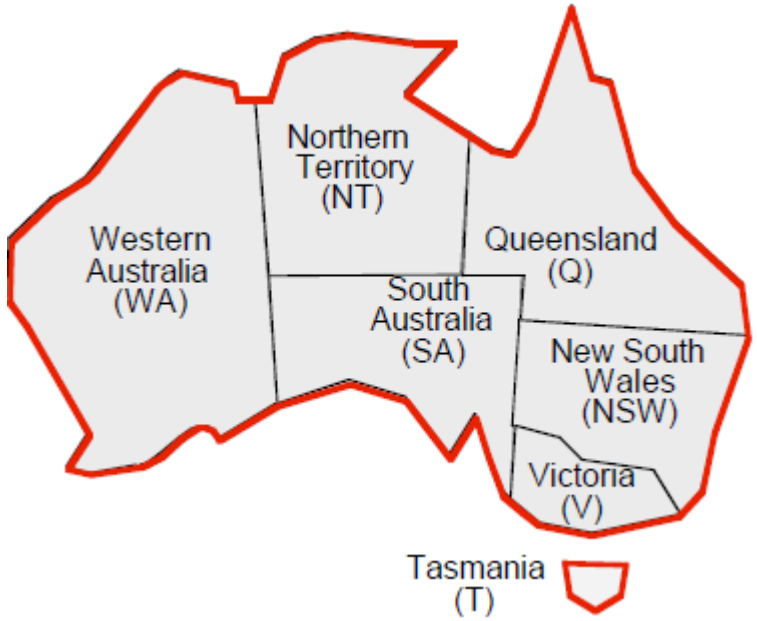
Problèmes de Satisfaction des contraintes (CSP)

Exemple 1

- Soit le problème CSP suivant :
  - V = {X1,X2,X3}
  - D1 = D2 = D3 = {1,2,3}
  - Contrainte : X1+X2 = X3
- Trois solutions possibles (assignations complètes et compatibles) :
  - {X1=1, X2=1, X3=2}
  - {X1=1, X2=2, X3=3}
  - {X1=2, X2=1, X3=3}

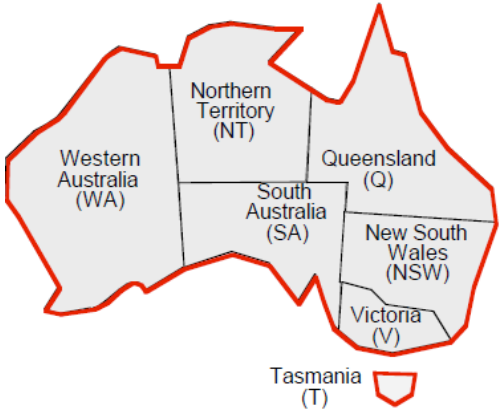
Exemple 2 : Colorier une carte

- Soit la carte de l'Australie à colorier :
  - On doit utiliser seulement trois couleurs (Rouge,Vert,Bleu) de sorte que deux états frontaliers n'aient jamais les mêmes couleurs



Problèmes de Satisfaction des contraintes (CSP)

Exemple 2 : Colorier une carte

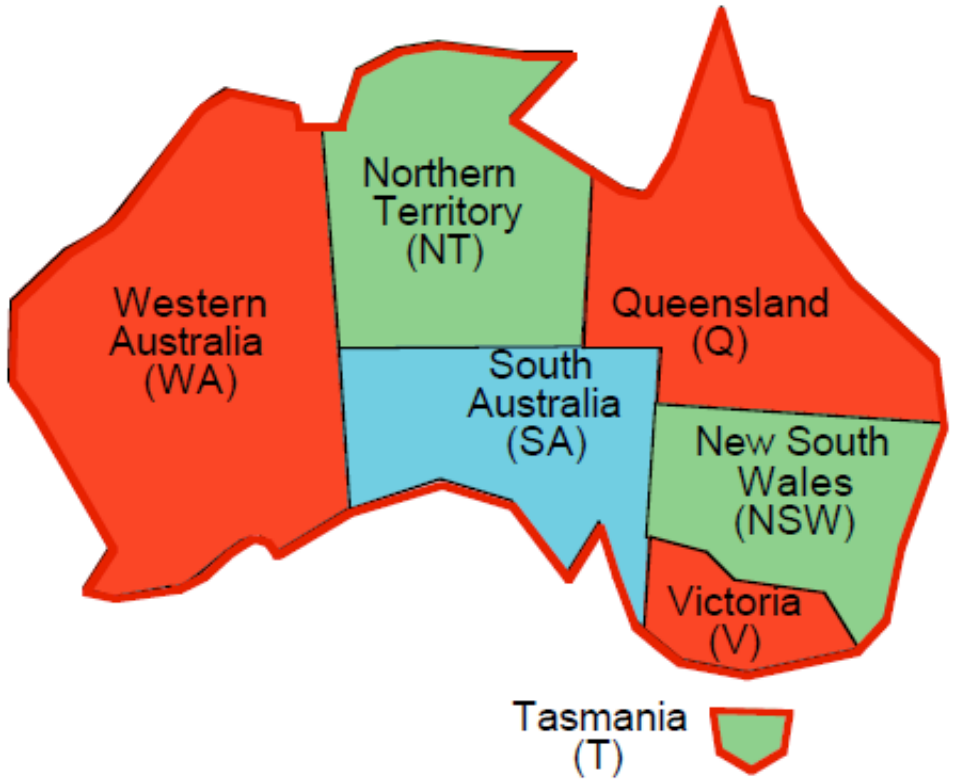


- **Formulation en CSP :**
  - Les variables sont les états de l'Australie :
    - V = {WA, NT, Q, NSW, V, SA, T}
  - Le domaine de chaque variable est l'ensemble des trois couleurs :
    - D = {**R**, **V**, **B**}

  - Les contraintes : Les états frontaliers doivent avoir des couleurs différentes :
    - WA ≠ NT, WA ≠ SA, NT ≠ Q, …

Exemple 2 : Colorier une carte

- **Solution complète et compatible :**
  - WA = R, NT = V, SA = B, Q = R, NSW = V, V = R, T = V



Problèmes de Satisfaction des contraintes (CSP)

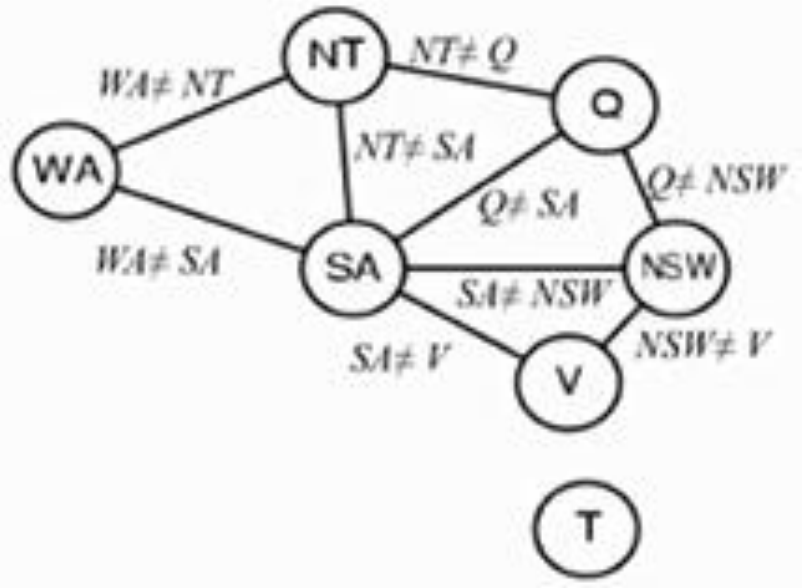
Types de contraintes

Une contrainte est caractérisée par son arité (nombre de variables qu'elle implique) :

- **Unaire:** Les contraintes ne concernent qu'une variable.
- **Binaire:** Les contraintes concernent deux variables.
- **Multiple** : Les contraintes concernent 3 variables ou plus.

Pour les problèmes avec des contraintes binaires, on peut visualiser le problème CSP par un graphe de contraintes :

- Les nœuds sont les variables (nœud = variable)
- Les arcs sont les contraintes entre deux variables



Problèmes de Satisfaction des contraintes (CSP)

Recherche en profondeur pour CSP

- **Paramètres de recherche :**
  - Un état est une assignation
  - Etat initial : assignation vide {}
  - Fonction de transition : assigne une valeur à une variable non encore assignée
  - Fonction but : Retourne Vrai si l'assignation est complète et compatible

- L'algorithme est général et s'applique à tous les problèmes CSP
- La solution doit être complète (elle apparait à une profondeur N)

Problèmes de Satisfaction des contraintes (CSP)

Recherche en profondeur pour CSP

- **Limitations :**
  - Niveau 1 de l'arbre : N*D branches (Chaque variable peut prendre D valeurs)
  - Niveau 2 : (N-1)*D branches pour chaque nœud (ainsi de suite jusqu'au niveau N)
  - Résultat : $N!*D^N$ nœuds pour seulement D^N assignations complètes
- Solution 1 : Considérer une seule variable à assigner à chaque niveau
- Solution 2 : Reculer (backtrack) lorsqu'aucune nouvelle assignation compatible est possible (inutile de continuer à assigner des variables s'il y a déjà violation de contraintes)

Solution 1 + Solution 2 = Algorithme Backtracking search

Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

Algorithme Backtracking-search(csp)

Retourner **Backtrack**({},csp)

Backtrack(assignation,csp){

1. Si assignation est complète, retourner assignation
2. Sinon X=**Var-Non-Assignée**(assignation,csp)
3. Pour chaque v dans **valeurs-Ordonnées**(X,assignation,csp)
    1. Si **compatible** ((X=v), assignation, csp)
        1. Ajouter (X=v) à assignation
        2. csp* = csp mais où **Domaine**(X, csp) est {v}
        3. csp*, ok = **Inférence**(csp*)
        4. Si ok = vrai
            1. Résultat = **Backtrack**(assignation, csp*)
            2. Si résultat ≠ faux, retourner Résultat
        5. Sinon Enlever (X=v) de assignation
4. Retourner faux

}

Variables, domaines, contraintes

Assignation de variables

Choix de prochaine variable

Ordre des valeurs à essayer

Tente de simplifier le problème CSP

S'il détecte un conflit ok = faux

Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

- Les variables du problème sont instanciées avec les valeurs des domaines, dans un ordre prédéfini, jusqu'à ce que l'un de ces choix ne satisfasse pas une contrainte. Dans ce cas, on doit remettre en cause la dernière instanciation réalisée. Une nouvelle valeur est essayée pour la dernière variable instanciée, que l'on appelle la variable courante.

- Si toutes les valeurs du domaine de cette variable on été testées sans succès, on doit procéder à un backtrack : on choisit une autre valeur pour la variable précédant immédiatement la variable courante. On répète ce processus jusqu'à obtenir une solution, c'est à dire une instanciation de toutes les variables.

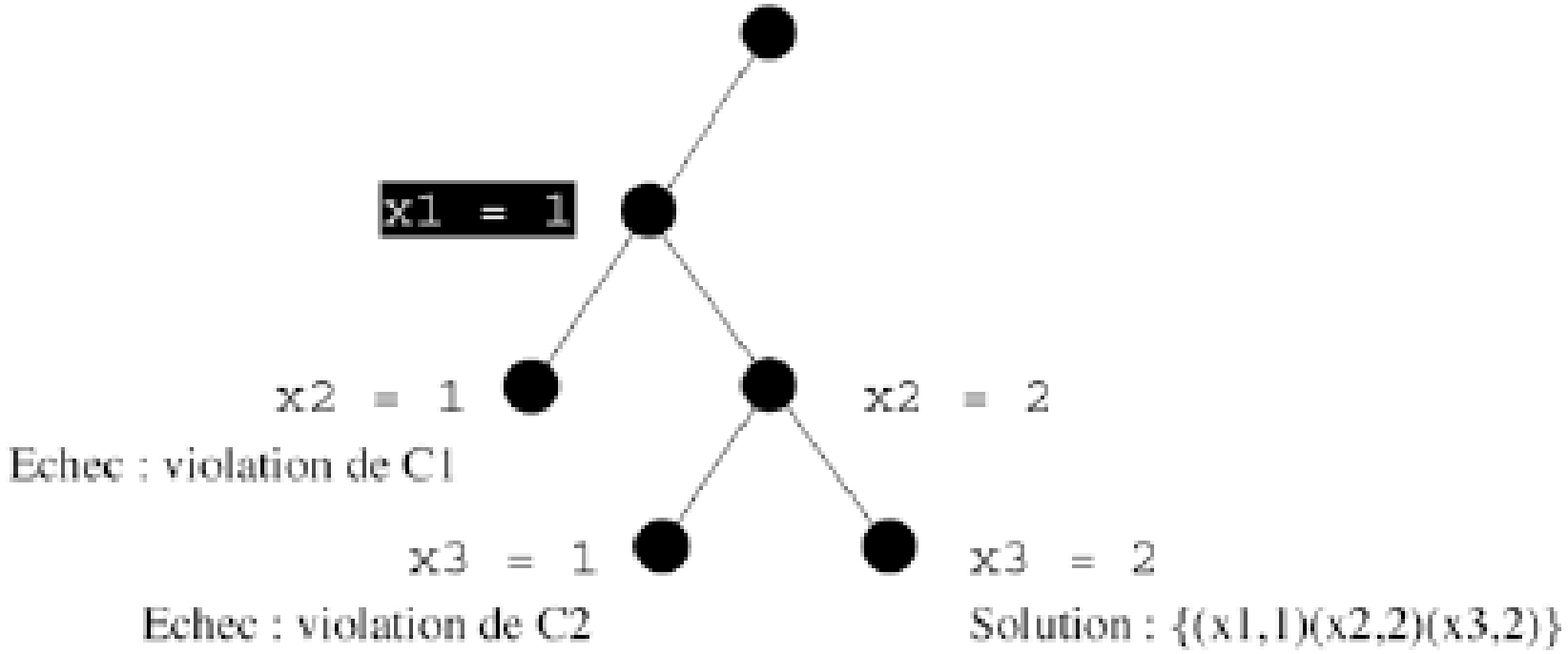
- Si on a parcouru tout l'arbre de recherche sans la trouver, alors on a prouvé que le problème n'a pas de solution.

Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

- **Illustration 1 :**

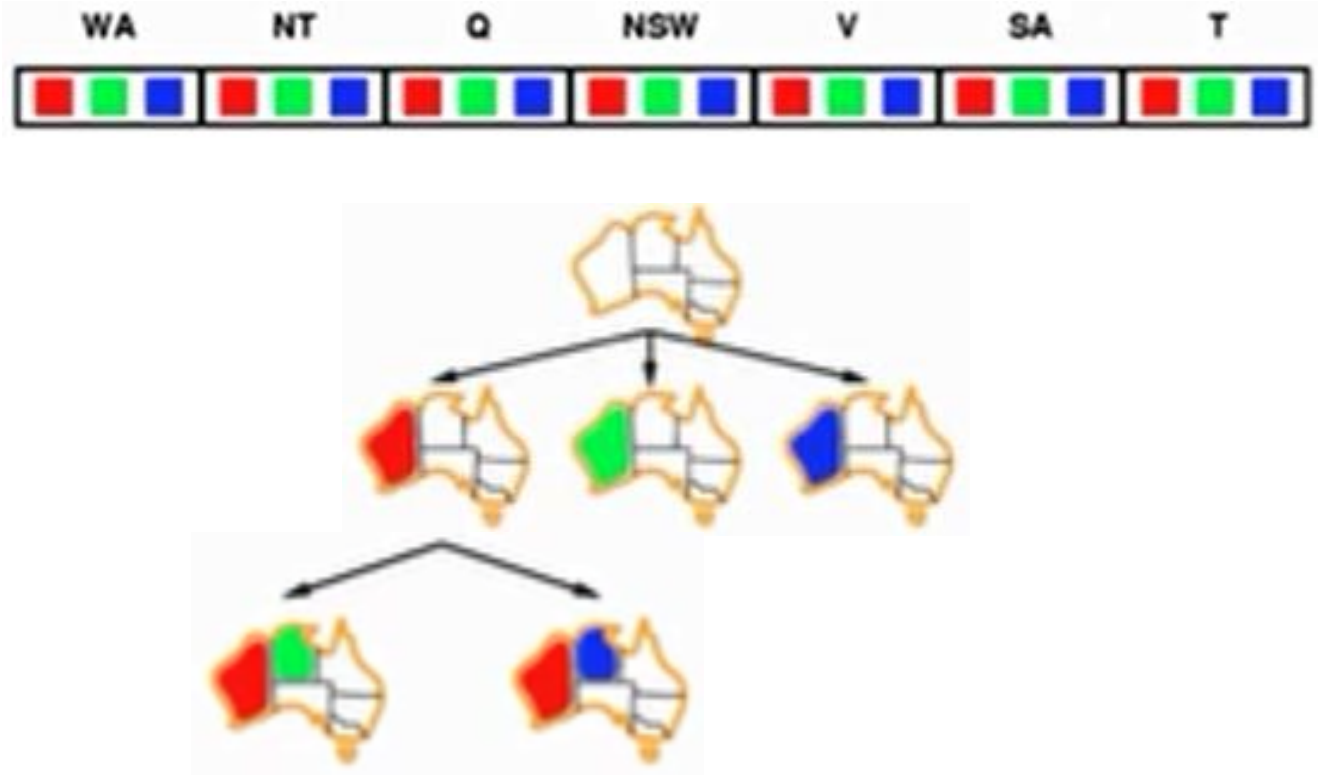
– Soient x1, x2 et x3 trois variables,
– Soient D(x1) = D(x2) = D(x3) = {1, 2, 3} leurs domaines respectifs.
– On pose les contraintes suivantes: C1 = [x1< x2] et C2 = [x2 = x3].
– On suppose que l'on instancie les variables dans l'ordre croissant des indices, en choisissant la plus petite valeur d'abord. x1 = 1



Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

- **Illustration 2 :**



Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

Exercice :

- Soit le problème CSP suivant :
  - V = {X1,X2,X3}
  - D1 = D2 = D3 = {1,2,3}
  - Contrainte : X1+X2 = X3

Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

- **Améliorations :**
  - Filtrage et propagation
  - Utilisation des heuristiques générales
    - Choix de la prochaine variable (**Var-Non-Assignée**)
    - Choix de la prochaine valeurs à assigner (**valeurs-Ordonnées**)
    - Détecter les assignations conflictuelles et réduction des domaines (**Inférence**)

Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

- **Filtrage et propagation**
  - supprimer les valeurs des domaines des variables impliquées dans une contrainte (éviter de parcourir des branches de l'arbre qui ne peuvent aboutir à une solution)

- **Exemple**

– *soient* x1, x2 et x3 *variables,*

– *soient* D(x1) = D(x2) = D(x3) = {1, 2, 3} *leurs domaines respectifs,*

– *soient les contraintes C1* = [x1 < x2] et C2 = [x2 = x3]

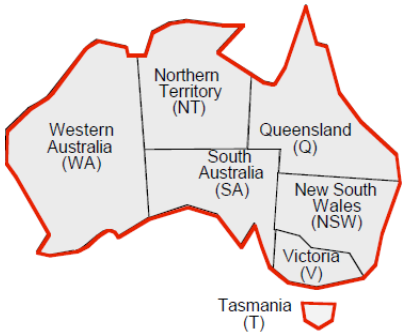
Filtrage : *La valeur* 3 *peut être supprimée de* D(x1)*, car il n'existe aucune valeur du domaine de* x2 *telle que C1 soit satisfaite si on instancie* x1 *avec* 3 *(et idem pour la variable* x2 *: la valeur* 1 *peut être supprimée).*

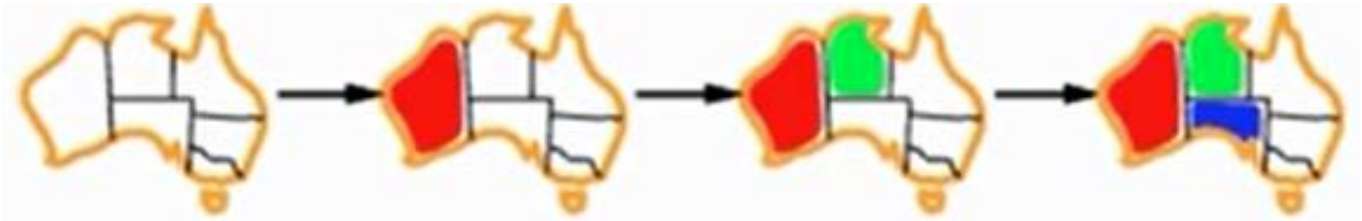
Propagation : *Le filtrage de la valeur* 1 *de* D(x2) *relatif à C*1 *peut être propagée sur* D(x3)*: si la valeur* 1 *n'appartient plus à* D(x2)*, alors la valeur* 1 *peut également être supprimée de* D(x3)*, car il n'existe plus de solution de C*2 *telle que* x3 *soit instanciée avec* 1.

Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

- **Heuristique 1 : Choix de l'ordre d'assignation des variables**
  - **Minimum Remaining Value (MRV) heuristic**
    - A chaque étape, choisir la variable avec le moins de valeurs compatibles restantes

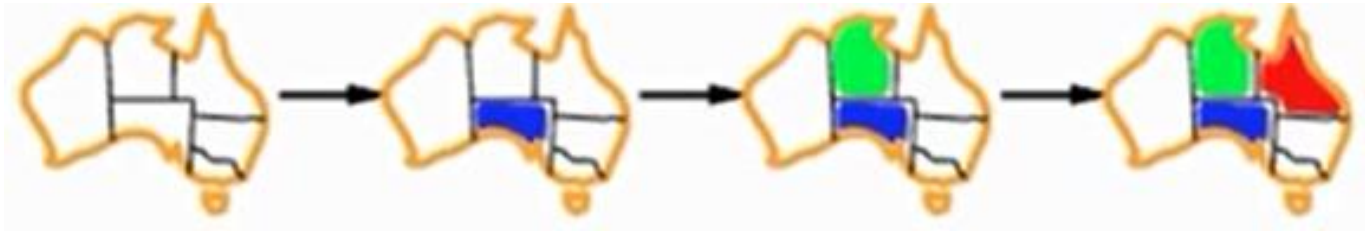


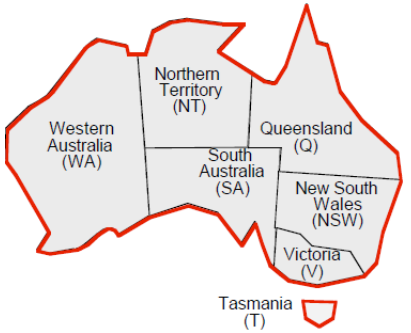


Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

- **Heuristique 1 : Choix de l'ordre d'assignation des variables**
  - **Degree heuristic**
    - Choisir la variable ayant le plus de contraintes impliquant des variables non encore assignées (si l'heuristique précédente donne le même nombre de valeurs compatibles)



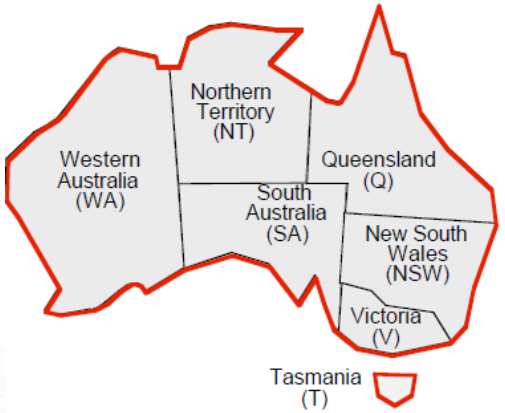


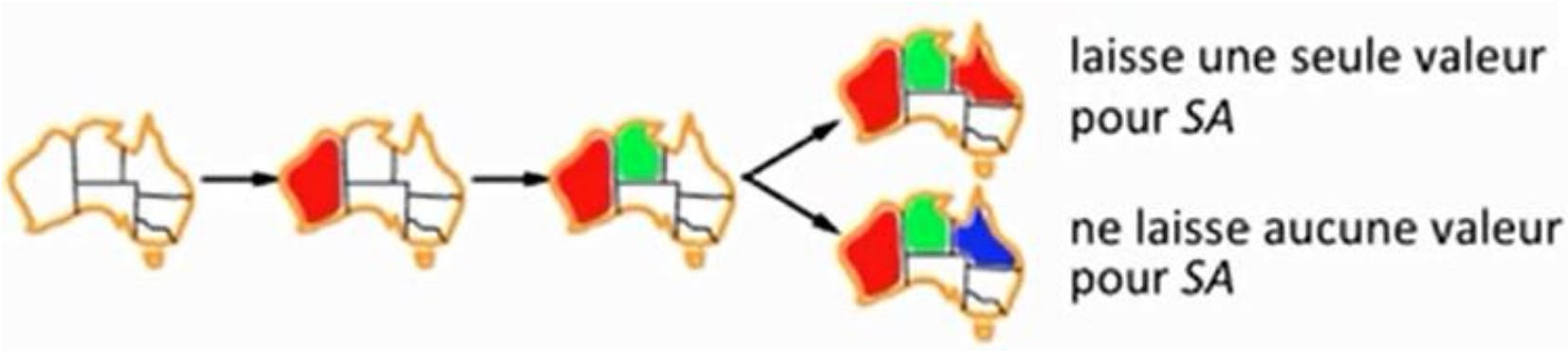
Problèmes de Satisfaction des contraintes (CSP)

Algorithme Backtracking search

- **Heuristique 2 : Choix de la prochaine valeur à assigner**
  - **Least constraining value**

Choisir une valeur qui invalide le moins de valeurs possibles pour les variables non encore assignées (Choisir la valeur qui va enlever le moins de choix pour les variables voisines)





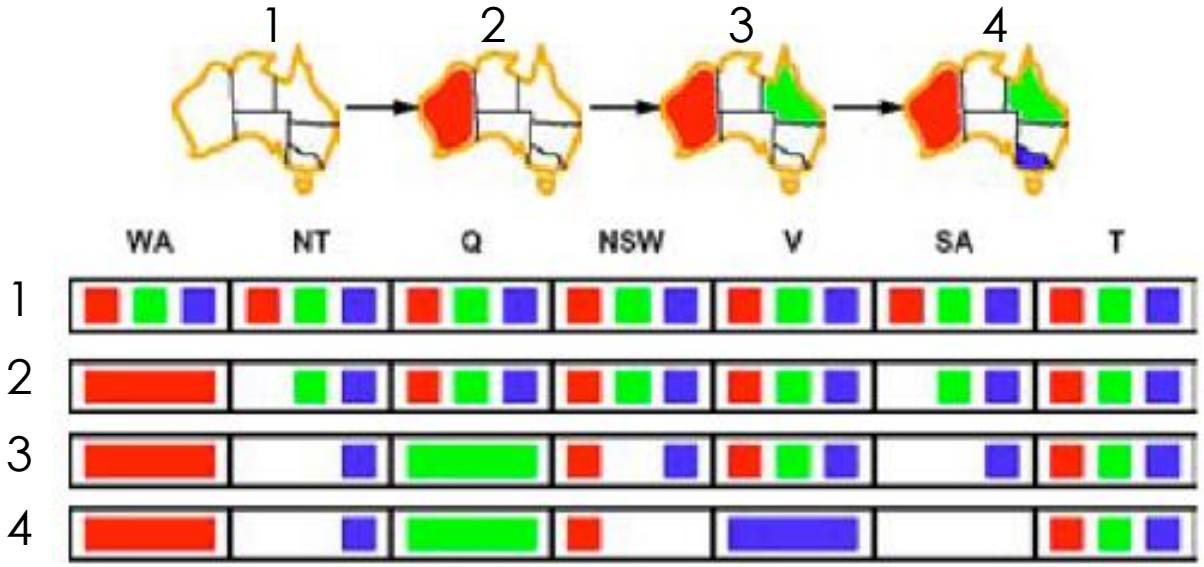
Problèmes de Satisfaction des contraintes (CSP)

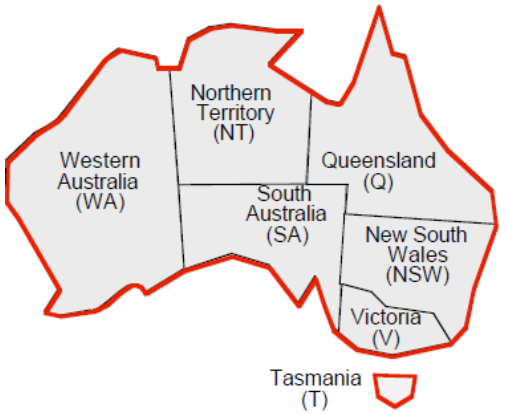
Algorithme Forward checking

- **Heuristique 3 : Détecter les assignations conflictuelles**

Forward checking (vérification anticipative) :

- Avant d'affecter une valeur *v* à une variable, vérifie que *v* est compatible avec les variables *suivantes*, c-à-d qu'il existe au moins une valeur pour chaque variable suivante qui soit consistante avec *v*. (contrairement au *backtrack*, qui vérifie que la valeur de la variable courante est compatible avec les valeurs affectées aux variables *précédentes*).





Problèmes de Satisfaction des contraintes (CSP)

Algorithme Forward checking

Algorithme Forward-Checking(X,csp)

Pour chaque X_k dans voisins(X,csp)

Changé, csp = Réviser(X_k,X,csp)

Si Changé et Domaine(X_k,csp) est vide Alors retourner(void,faux)

Sinon retourner(csp,vrai)

Réviser(X_i,X_j,csp){ *//réduire le domaine de X_i en fonction de de celui de X_j*

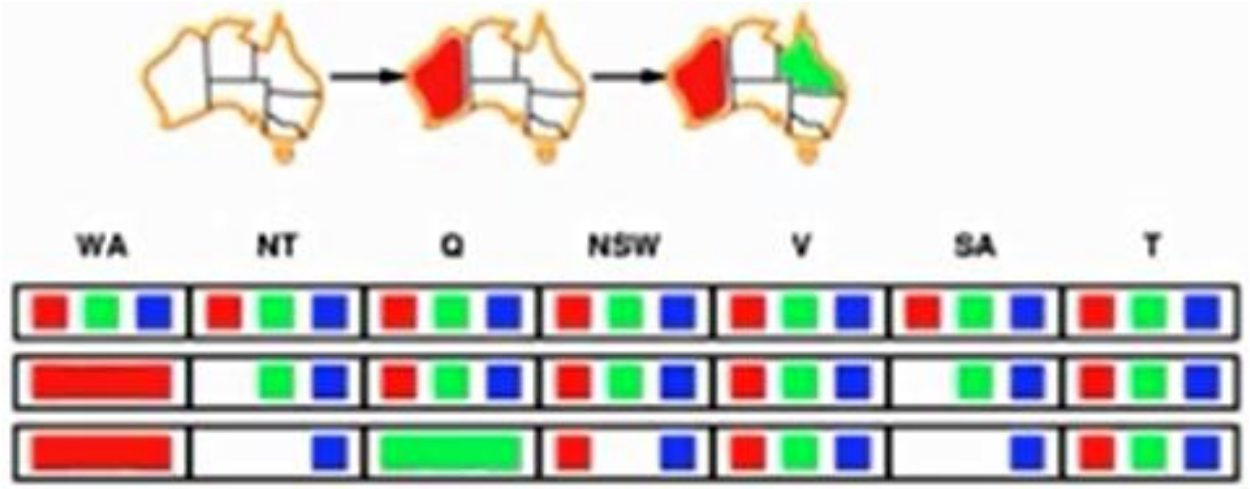
1. Changé = faux
2. Pour chaque x dans Domaine(X_i,csp)
    1. Si aucun y dans Domaine(X_j,csp) satisfait contrainte entre X_i et X_j
        1. Enlever x de Domaine(X_i,csp) *// changer le csp*
        2. Changé = vrai
        3. Retourner(Changé,csp)

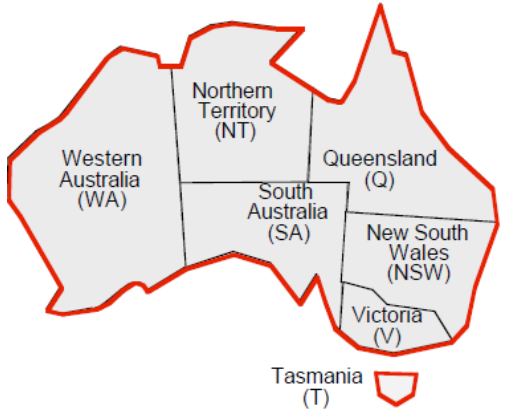
}

Problèmes de Satisfaction des contraintes (CSP)

Algorithme AC-3

- ***Forward checking*** propage l'information des variables assignées vers les variables non assignées, mais il ne détecte pas les conflits locaux entre ces variables :



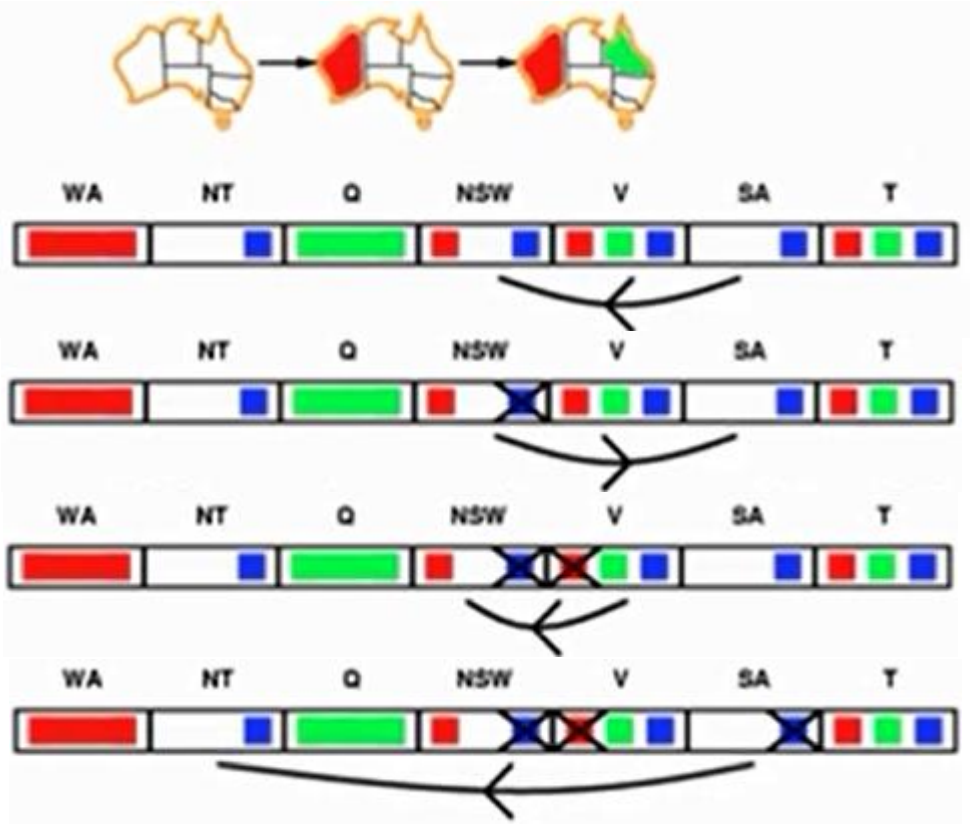


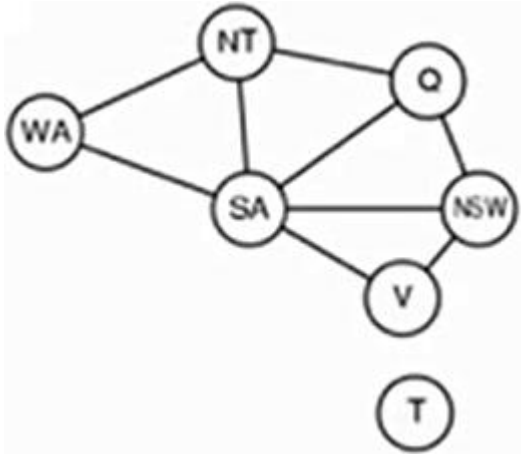
- NT et SA ne peuvent pas être bleus ensemble !
- **AC-3** (Arc consistency) permet de vérifier les contraintes localement par propagation des contraintes

Problèmes de Satisfaction des contraintes (CSP)

Algorithme AC-3

- AC-3 :
  - Vérifier la compatibilité entre les arcs (compatibilité des contraintes entre deux variables)
  - L'arc X → Y est compatible SSI : pour chaque valeur x de X il existe au moins une valeur permise y de Y





Si une variable perd une valeur, ses voisins doivent êtres revérifiées

Problèmes de Satisfaction des contraintes (CSP)

Recherche locale

- Principe de la recherche locale
  - Le chemin à la solution est sans importance (e.g : hill-climbing)
  - On peut travailler avec des états qui sont des assignations complètes (compatibles ou non)
  - Incv : peut tomber dans des optima locaux

  - Algorithme min-conflicts
    - Fonction objective : minimiser le nombre de conflits
    - Ressemble à hill-climbing mais qui utilise la stochasticité

Problèmes de Satisfaction des contraintes (CSP)

Algorithme min-conflicts

Algorithme Min-conflicts(csp, nb_itérations)

Assignation = une assignation aléatoire complète (probablement pas compatible) de csp

Pour i = 1 à nb_itérations

1. Si assignation est compatible Alors retourner assignation
2. Sinon X = variable choisie aléatoirement dans Variable(csp)
3. v = valeur dans Domaine(X,csp) satisfaisant le plus de contraintes de X
4. Assigner (X = v) dans assignation

Retourner faux

- Min-conflicts : -peut résoudre un problème de 1.000.000 reines en 50 étapes
  -planifier les observation du Télescope hubble dans 10 min au lieu de 3 semaines
- Raison du sccès : il existe plusieurs solutions possibles (éparpillées) dans l'espace des états