Approche algorithmique de l'IA décisionnelle

Cours « Découverte de l'intelligence artificielle »

Nicolas Delestre et Laurent Vercouter

INSA ROUEN NORMANDIE

CC BY NC SA

Plan

1. Avant propos
2. Introduction
3. Problème de recherche de chemin et d'optimalité
   - Force brute
   - Dijkstra
   - A*
4. Jeux à deux joueurs
   - Exploration avec limitation en profondeur
     - MinMax
     - Élagage Alpha-Beta
   - Exploration partielle mais complète
5. Conclusion

INSA ROUEN NORMANDIE

CC BY NC SA

- Ce cours a été rédigé en LaTeX sous emacs avec le mode Copilot
- ChatGPT est donc co-rédacteur de ce document (proposition de formulation des phrases)

INSA ROUEN NORMANDIE

CC BY NC SA

La récursivité 1 / 2

Qu'est ce qu'un algorithme récursif ?

- Un algorithme qui s'appelle lui-même pour résoudre un problème
- L'espace de recherche est réduit à chaque appel récursif → convergence vers un cas de base, non récursif

Factorielle

$$\begin{cases} 0! = 1! = 1 \\ n! = n(n-1)! \end{cases}$$

```
fonction fact (n : Naturel) : Naturel
debut
    si n=0 ou n=1 alors
        retourner 1
    sinon
        retourner n*fact(n-1)
    finsi
fin
```

BY NC SA

La récursivité 2 / 2

Difficulté

- Être à la fois concepteur **et** utilisateur de l'algorithme

Bonne pratique

- Identifier le(s) cas non récursifs
- Lors de l'appel récursif, ne pas essayer de comprendre « comment ce marche », considérer que cet appel fonctionne et retourne bien le bon résultat

INSA ROUEN NORMANDIE

BY NC SA

Principe de l' approche algorithmique 1 / 5

Objectif

- Trouver une séquence d'actions qui permettent de résoudre un problème
- Trouver une bonne action (la meilleure ?) à effectuer

Principe

- On modélise le problème comme un espace d'états E tel qu'il existe des actions permettant de passer d'un état à un autre
- On définit :
  - un état initial e_{init}
  - un ou plusieurs états objectifs, finaux $\{e_{final_1}, e_{final_2}, \dots\}$, solution(s) du problème
  - une action a_i qui, lorsqu'elle existe, permet de passer de e_i à e_{i+1} : $\exists e_i, e_{i+1} \in E \times E, app(a_i, e_i) = e_{i+1}$

CC BY NC SA

Principe de l' approche algorithmique 2 / 5

Graphe, définition

Un graphe G est composé de deux ensembles S et A :

- S est un ensemble fini d'éléments, appelés sommets (ou aussi nœuds)
- A est un ensemble fini d'éléments, appelés arcs

Concept d'orientation

- graphe non orienté : $(i, j) = (j, i)$
- graphe orienté : $(i, j) \neq (j, i)$

INSA ROUEN NORMANDIE

BY NC SA

Principe de l' approche algorithmique 3 / 5

Redéfinition du problème

- On modélise le problème comme un graphe d'états E (sommets du graphe sont les états) tel qu'il existe une action a permettant de passer d'un état à un autre (les actions sont les arêtes du graphe) :

$$\forall e, e' \in E^2, \exists a \in Actions, app(e, a) = e' \Leftrightarrow (e, e') \in A$$

État explicite ou implicite

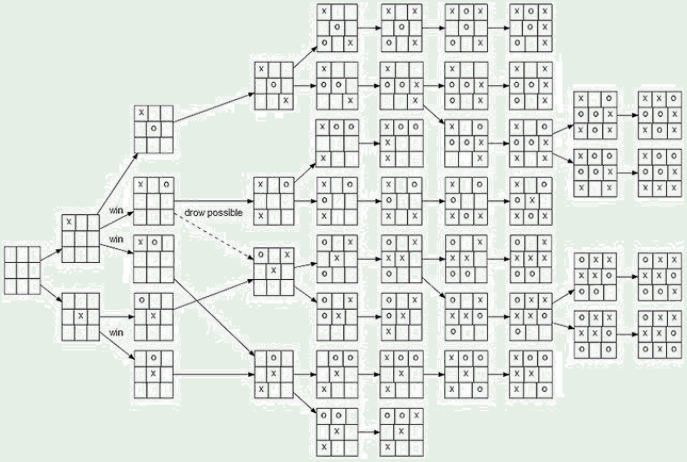
- Explicite : on a accès à tout le graphe d'états
- Implicite : on a une fonction qui génère les états accessibles depuis un état donné, et on « construit » les états à la volet

INSA ROUEN NORMANDIE

CC BY NC SA

Principe de l' approche algorithmique 4 / 5

Exemple : extrait du graphe d'états du Tic-Tac-Toe



https://www.researchgate.net/publication/290786914_Exploring_and_Expanding_the_World_of_Artificial_Intelligence

BY NC SA

Principe de l' approche algorithmique 5 / 5

Deux types de problèmes

- Problème de recherche de chemin, d'optimalité : trouver la séquence d'actions a_i pour atteindre le ou les états $\{Etat_{final_1}, Etat_{final_2}, \dots\}$
- Problème de jeu à deux joueurs : choisir la première action a_1 à effectuer pour essayer d'atteindre par la suite un état final gagnant

INSA ROUEN NORMANDIE

CC BY NC SA

Recherche de chemin : trois cas, trois méthodes 1 / 2

Force brute

- Exploration exhaustive de l'espace des états.
  - Explore systématiquement chaque possibilité jusqu'à ce que la solution soit trouvée (exploration en largeur ou en profondeur)
  - À choisir lorsque l'espace de recherche est « petit »

Dijkstra

- Exploration guidée par un coût réel
  - Trouve le chemin le plus court dans un graphe avec des coûts d'arête non négatifs. Il garantit l'optimalité
  - Lorsque le graphe est petit à modéré, et qu'une solution optimale est absolument requise

INSA ROUEN NORMANDIE

CC BY NC SA

Recherche de chemin : trois cas, trois méthodes 2 / 2

A*

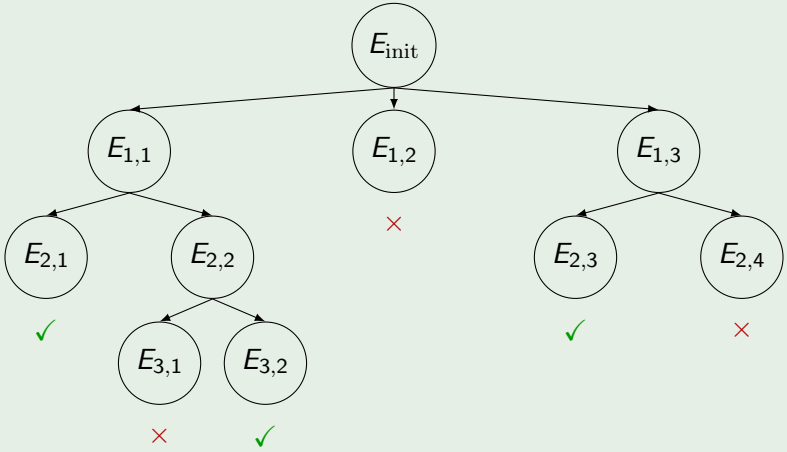
- Exploration guidée par un cout réel et une heuristique
  - Utilise une fonction heuristique pour estimer le coût restant jusqu'à l'objectif, combinée au coût réel déjà encouru
  - Lorsque l'on cherche un compromis entre optimalité et performance, surtout dans les grands espaces de recherche

INSA ROUEN NORMANDIE

BY NC SA

Force brute

Principe



BY NC SA

Principe de l'algorithme

Parcours exhaustif en profondeur : *backtracking*

- On **résout** le problème depuis e_{init}
- Si l'état courant est un état final, alors on a trouvé une solution
- Si l'état courant n'est pas l'état final et n'a plus d'action, alors pas de solution courante
- Sinon on essaye les actions possibles, en **résolvant** le problème après application de chaque action

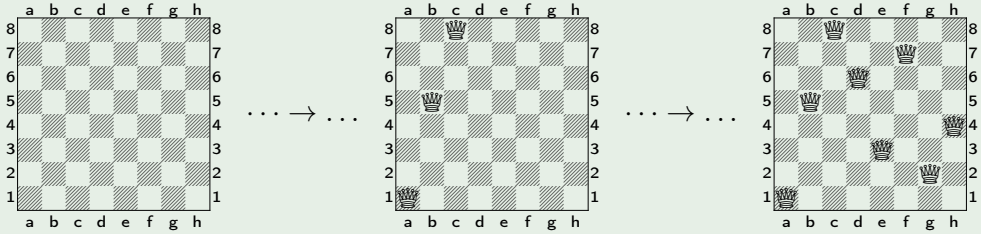
Remarques

- Le *backtracking* permet de faire un parcours exhaustive en profondeur
- Il est possible d'adapter l'algorithme pour faire un parcours en largeur (remplacement de la récursivité par l'utilisation d'une file d'états à traiter)

CC BY NC SA

Exemple : Problème des n reines 1 / 5

- Comment placer n reines (dames) sur un échiquier de $n \times n$ cases sans qu'aucune ne puisse se prendre ?
- e_{init} est l'échiquier vide, l'action a_i est posée une dame sur une colonne i vide, un état final est un échiquier avec n dames tel qu'aucune ne peut prendre une autre dame



- Nombre d'états à tester au maximum $8^8 = 16777216$

CC BY NC SA

Exemple : Problème des *n* reines 2 / 5

On suppose posséder les fonctions et procédures suivantes

- **fonction** echiquierVide **() :** Echiquier
- **fonction** caseVide **(**e : Echiquier, colonne : 1..8, ligne : 1..8**) : Booleen**
- **fonction** aucunesReinesPeuventSePrendre **(**e : Echiquier**) : Booleen**
- **fonction** colonneVide **(**e : Echiquier**) :** 0..8 # 0 si pas de colonne vide
- **procédure** placerReine **(E/S** e : Echiquier, **E** colonne : 1..8, ligne : 1..8**)**
  ⌊**précondition(s)** caseVide(e,col,lig)
- **procédure** retirerReine **(E/S** e : Echiquier, **E** colonne : 1..8, ligne : 1..8**)**
  ⌊**précondition(s)** non caseVide(e,col,lig)

INSA ROUEN NORMANDIE

cc BY NC SA

Exemple : Problème des *n* reines 3 / 5

Résolution par force brute

```
procédure resoudreReines (E e : Echiquier, E/S solutions : Tableau[1..MAX] d'Echiquier, nbSolutions : Naturel)
    Déclaration   col : 0..8
                  lig : 1..8
debut
    col ← colonneVide(e)
    si col=0 alors
        si aucunesReinesPeuventSePrendre(e) alors
            nbSolutions ← nbSolutions+1
            solutions[nbSolutions] ← e
        finsi
    sinon
        pour lig ←1 à 8 faire
            placerReine(e, col, lig)
            resoudreReines(e, solutions, nbSolutions)
            retirerReine(e, col, lig)
        finpour
    finsi
fin
```

INSA ROUEN NORMANDIE

CC BY NC SA

Exemple : Problème des *n* reines 4 / 5

Appel initial

e : Echiquier, solutions : **Tableau[**1..MAX**] d'**Echiquier, nbSolutions : **Naturel**

...

e ← echiquierVide()

nbSolutions ← 0

resoudreReines(e, solutions, nbSolutions)

Dans la pratique...

```
$ time python probleme_des_n_reines.py
Nb solutions: 92

real 9m20,462s
user 9m20,076s
sys 0m0,194s
```

BY NC SA

Exemple : Problème des *n* reines 5 / 5

Résolution par force brute (optimisé)

```
procédure resoudreReines (E e : Echiquier, E/S solutions : Tableau[1..MAX] d'Echiquier, nbSolutions : Naturel)
    Déclaration   col : 0..8
                  lig : 1..8
debut
    col ← colonneVide(e)
    si col=0 alors
        nbSolutions ← nbSolutions+1
        solutions[nbSolutions] ← e
    sinon
        pour lig ←1 à 8 faire
            placerReine(e, col, lig)
            si aucunesReinesPeuventSePrendre(e) alors
                resoudreReines(e, solutions, nbSolutions)
            finsi
            retirerReine(e, col, lig)
        finpour
    finsi
fin
```

Dans la pratique...

```
$ time python probleme_des_n_reines.py
Nb solutions: 92

real 0m0,722s
user 0m0,639s
```

Constat

- Dans certains cas, les actions peuvent avoir un « coût » : $\forall a \in A, cout(a) \geq 0$
- Le problème n'est plus seulement :
  - de trouver une suite d'actions $a_1, \ldots, a_n$ permettant de passer de l'état e_{init} à l'état final e_{final}
  - mais aussi de minimiser le coût total : $solution(a_1, \ldots, a_n) = argmin_{a_1,\ldots a_n}(E_{i=1}^{n} \sum cout(a_i)$

Exemple

- État initial : je suis à Rouen
- État final : je suis à Montpellier
- Problème : trouver suite de directions à prendre pour aller de Rouen à Montpellier par le chemin le plus court

Dijkstra

INSA
BY NC SA

Principe

Arbre, définition

- Un arbre est un graphe avec un sommet de départ (racine) tel qu'il n'existe qu'un seul chemin depuis la racine vers un autre sommet de l'arbre

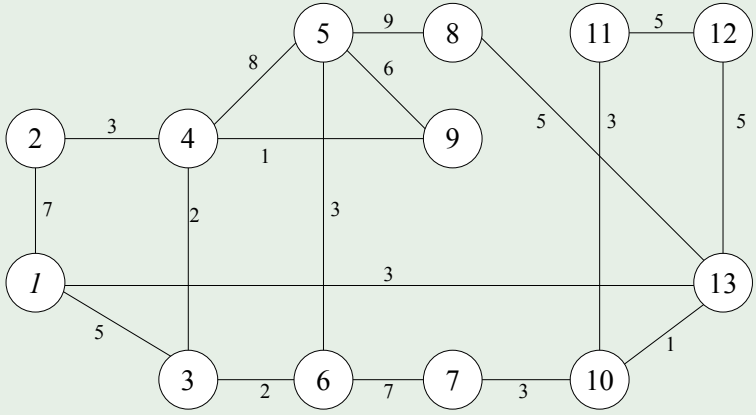
Principe de l'algorithme de Dijkstra

- Construire un arbre A qui va recouvrir le graphe d'états G tel que
  - la racine de l'arbre est e_{init}
  - à chaque étape de construction de l'arbre A on ajoute à l'arbre un état e du graphe qui n'est pas encore dans l'arbre et tel que la longueur du chemin de e_{init} à e est la plus courte
- Comment ? En associant à chaque état e de l'arbre, la longueur $l(e)$ du chemin le plus court depuis e_{init} jusqu'à e
  - $l(e_{init}) = 0$ et e_{init} est la racine de l'arbre
  - à chaque étape ajouté à l'arbre l'arête (e, e') tel que $e \in A$ et :

$$e' = \underset{\substack{e' \in G \setminus A \\ app(e,a)=e'}}{argmin} \; (l(e) + cout(a))$$

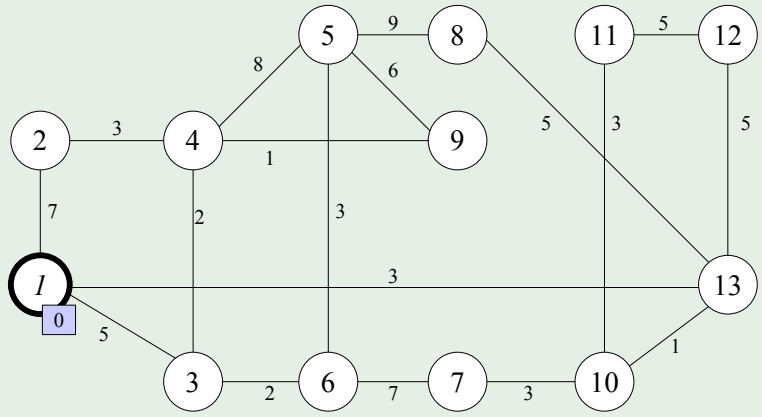
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



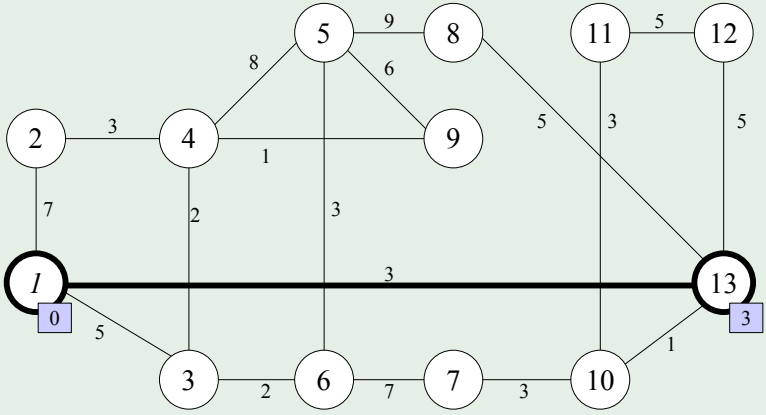
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



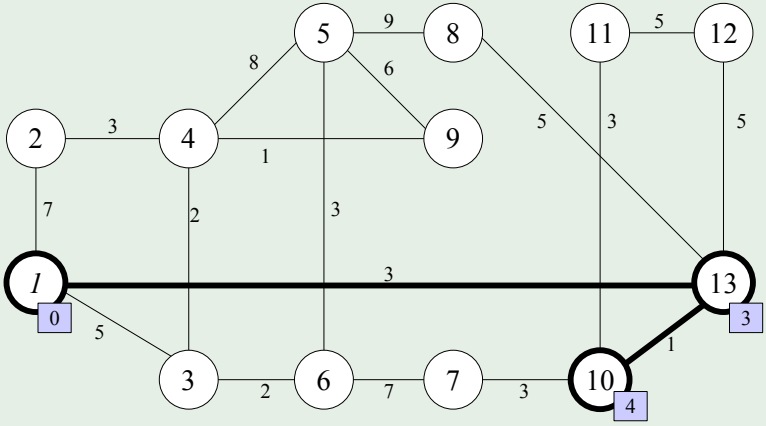
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



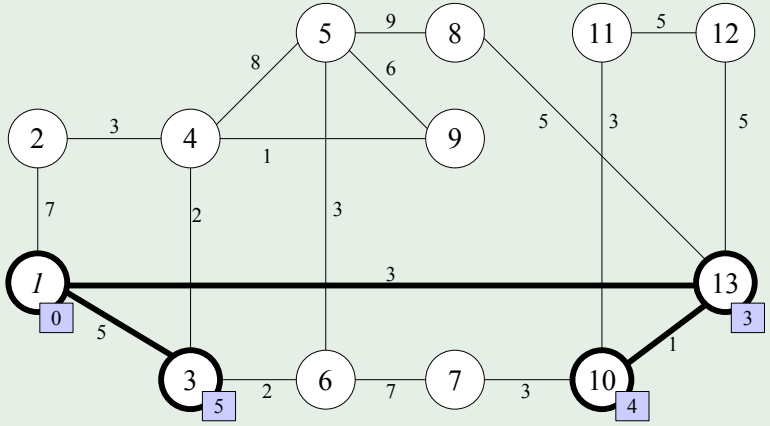
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



Dijsktra sur un exemple

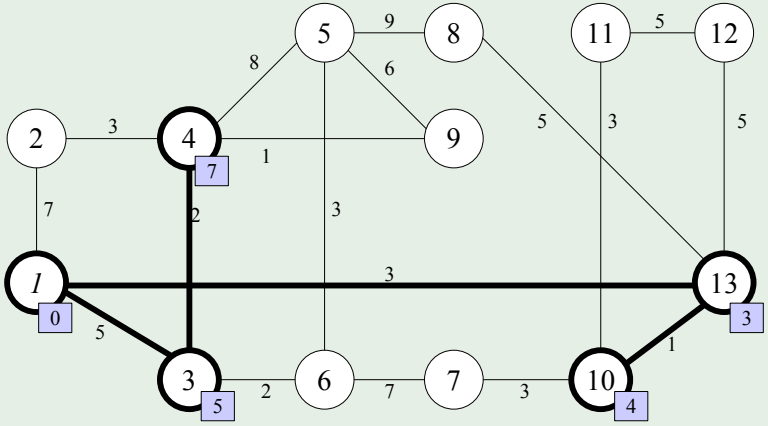
Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



BY NC SA

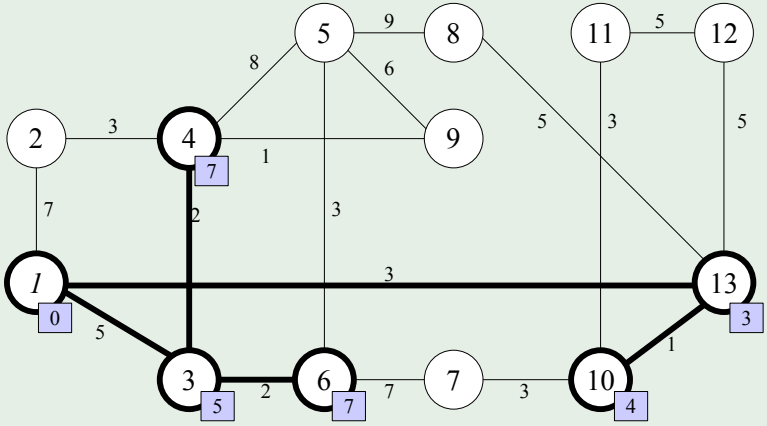
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



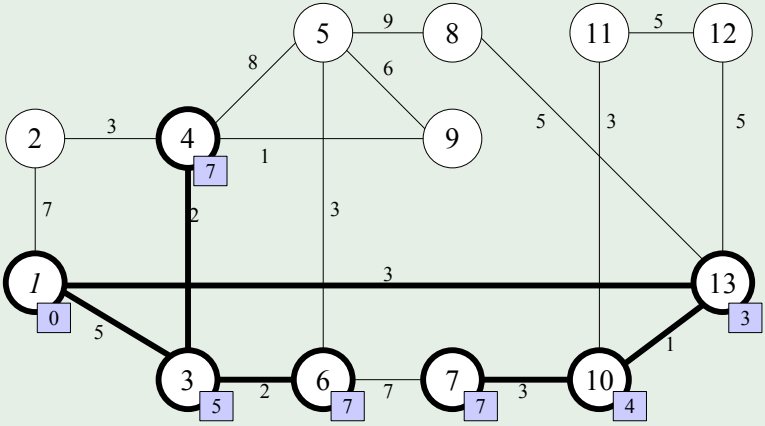
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



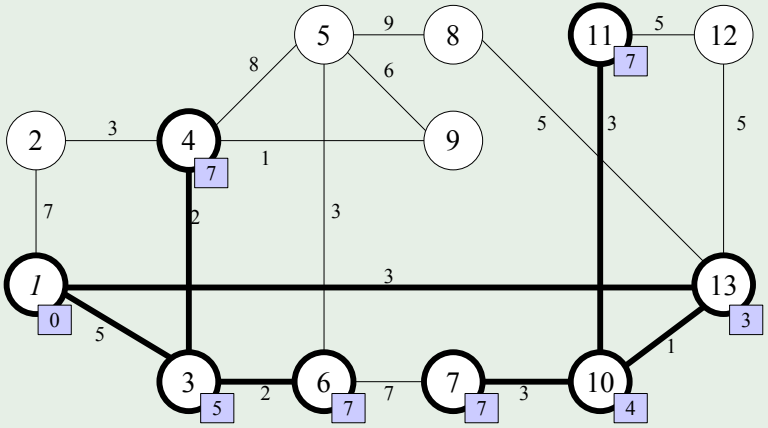
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



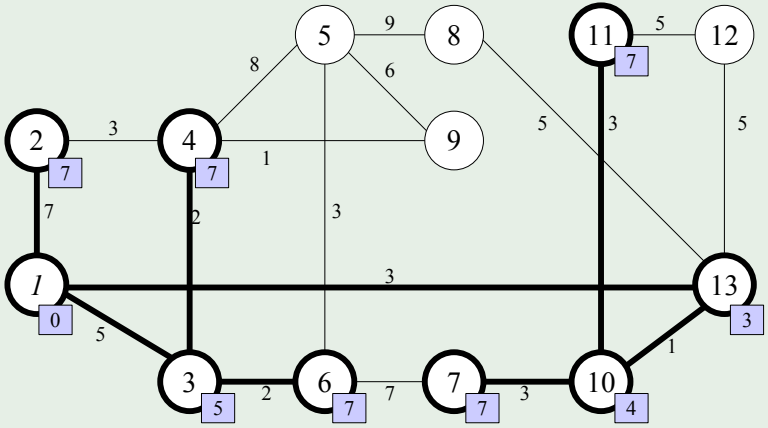
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



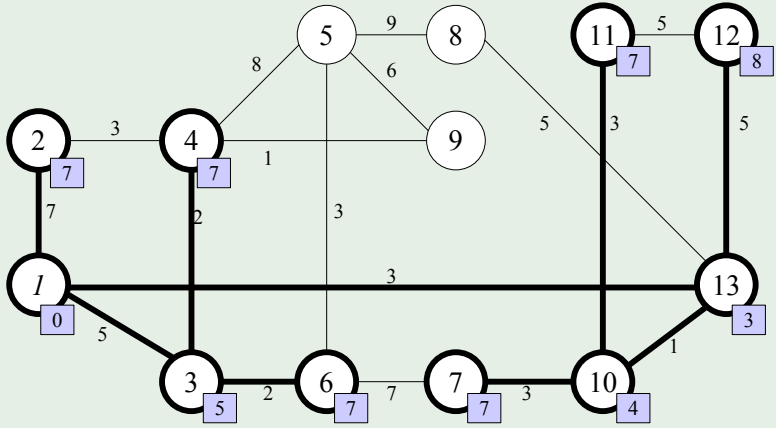
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



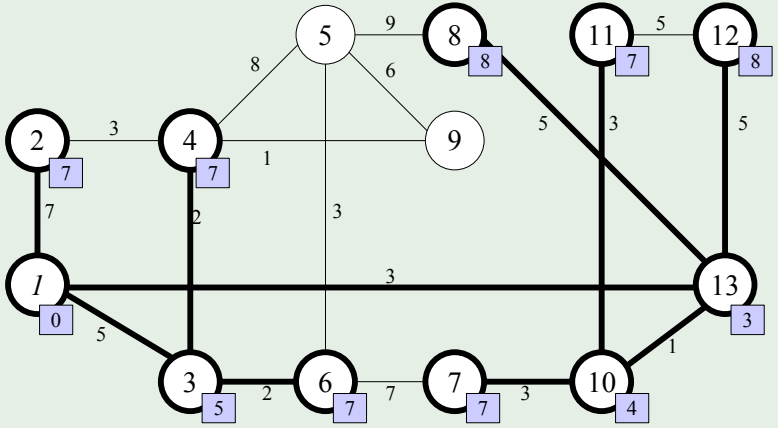
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



Dijsktra sur un exemple

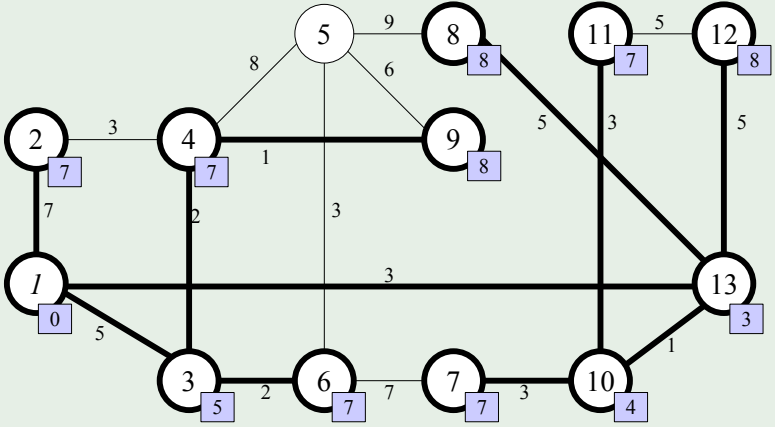
Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



BY NC SA

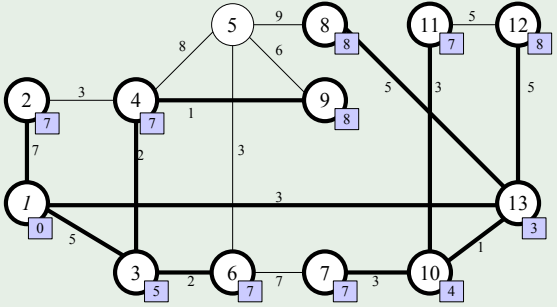
Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



Dijsktra sur un exemple

Quel est le plus court chemin pour aller du sommet 1 au sommet 9 ?



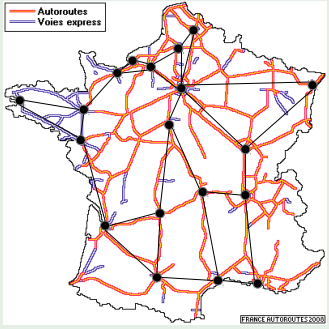
Remarque

- Si les coût de chaque arc est égal à identique (par exemple 1), alors Dijkstra revient à une recherche Force Brute en largeur

Dijsktra sur un cas réel 1 / 3

Exemple

- Quel est le plus court chemin entre Rouen et Montpellier ?

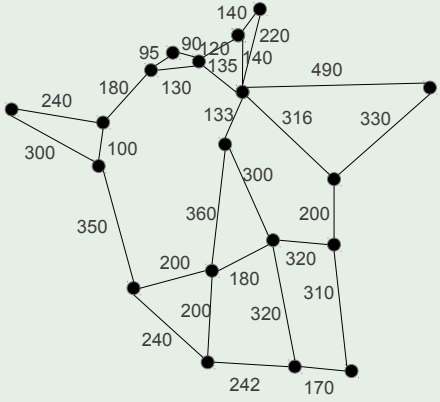


http://www.lecartographe.fr

BY NC SA

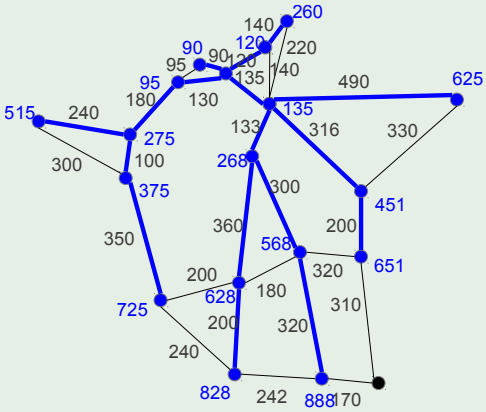
Dijsktra sur un cas réel 2 / 3

Le graphe correspondant



Dijsktra sur un cas réel 3 / 3

Utilisation de l'algorithme de Dijkstra



Principe de l'algorithme A*

Fonction heuristique, définition

- Une fonction heuristique h est une fonction qui associe à chaque sommet s un coût estimé $h(s)$ pour atteindre l'état final à partir de s

- Dans l'exemple précédent, cela pourrait être la distance à vol d'oiseau entre une ville et Montpellier

Principe de l'algorithme A*

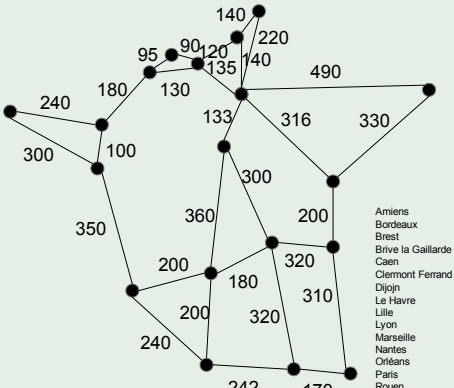
- Reprendre l'algorithme de Dijkstra mais en sélectionnant e' à ajouter à l'arbre non pas seulement en fonction du coût depuis la racine de l'arbre mais aussi en fonction de l'heuristique :

$$e' = argmin_{\substack{e' \in G \setminus A \\ e \in A \\ app(e,a)=e'}} (l(e) + cout(a) + h(e'))$$

- Dijkstra est A* avec une heuristique à 0

Plus court chemin pour aller de Rouen à Montpellier 1 / 2

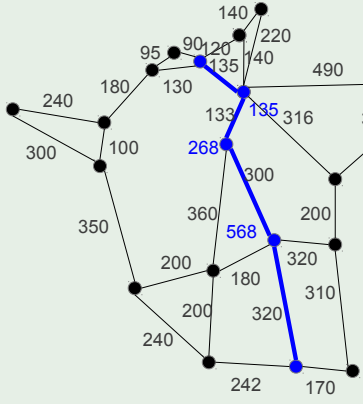
Heuristique : distance à vol d'oiseau entre une ville et Montpellier



	Montpellier
Amiens	710
Bordeaux	380
Brest	840
Brive la Gaillarde	250
Caen	700
Clermont Ferrand	250
Dijojn	420
Le Havre	715
Lille	790
Lyon	250
Marseille	130
Nantes	580
Orléans	500
Paris	600
Rouen	680
Strasbourg	630
Toulouse	195

Plus court chemin pour aller de Rouen à Montpellier 2 / 2

Utilisation de l'algorithme A*



	Montpellier
Amiens	710
Bordeaux	380
Brest	840
Brive la Gaillarde	250
Caen	700
Clermont Ferrand	250
Dijojn	420
Le Havre	715
Lille	790
Lyon	250
Marseille	130
Nantes	580
Orléans	500
Paris	600
Rouen	680
Strasbourg	630
Toulouse	195

Jeux à deux joueurs à somme nulle 1 / 2

Contexte

- L'IA est souvent utilisée comme adversaires pour des jeux de société
- Le plus souvent, ces jeux sont à deux joueurs, à somme nulle (le gain d'un joueur est la perte de l'autre) : Puissance 4, Reversi (Othello), Dames, Échecs, Go, etc.

Principe

- On est capable d'évaluer la qualité d'une position de jeu (fonction d'évaluation) : $evaluer : Etat \times Joueur \rightarrow \textbf{Entier}$
  - $evaluer(e, j) = -evaluer(e, autrejoueur(j))$
- On part du principe que tous les joueurs jouent de manière optimale (chaque joueur cherche à gagner, à maximiser son gain)

ROUEN NORMANDIE

BY NC SA

Jeux à deux joueurs à somme nulle 2 / 2

Problème

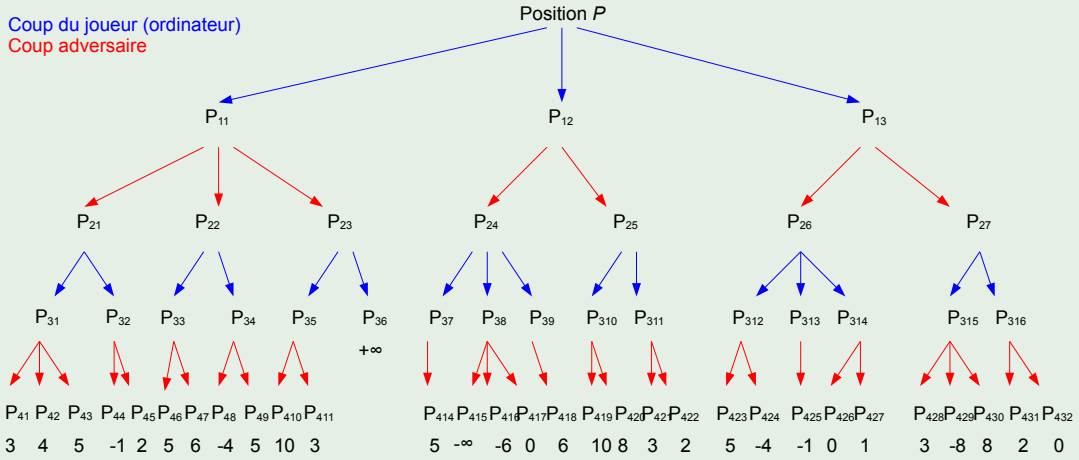
- L'espace de recherche est souvent trop grand pour permettre une recherche exhaustive
  - Puissance 4 : 4.5×10^{12} positions
  - Reversi (Othello) : 10^{28} positions
  - Dames : 10^{40} positions
  - Échecs : 10^{47} positions
  - Go : 10^{170} positions
- Le graphe d'état n'est donc pas explicite

Deux stratégies possibles pour choisir le meilleur coup

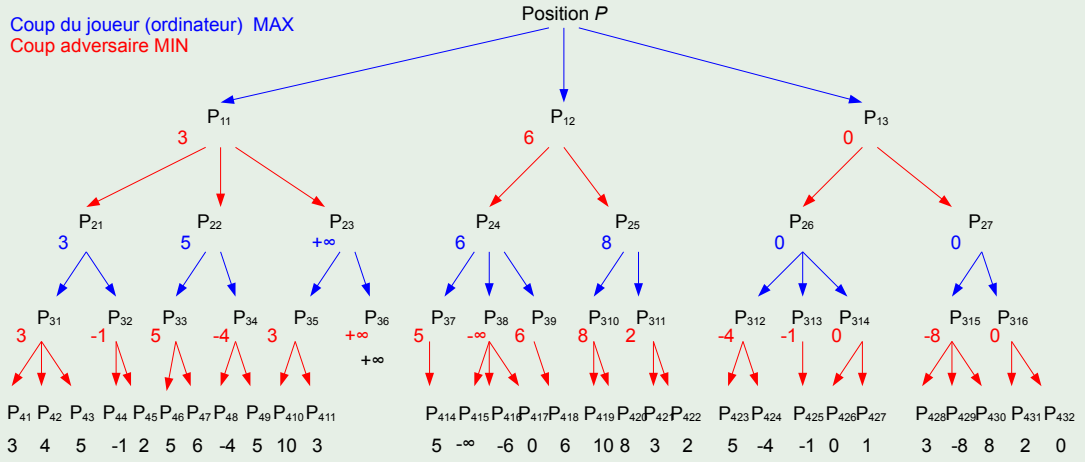
- Exploration complète mais limitée en profondeur (algorithmes MinMax, Alpha-Beta)
- Exploration partielle mais jusqu'au bout ou du moins assez profondément (algorithme Monte Carlo)

BY NC SA

Algorithme MinMax : principe

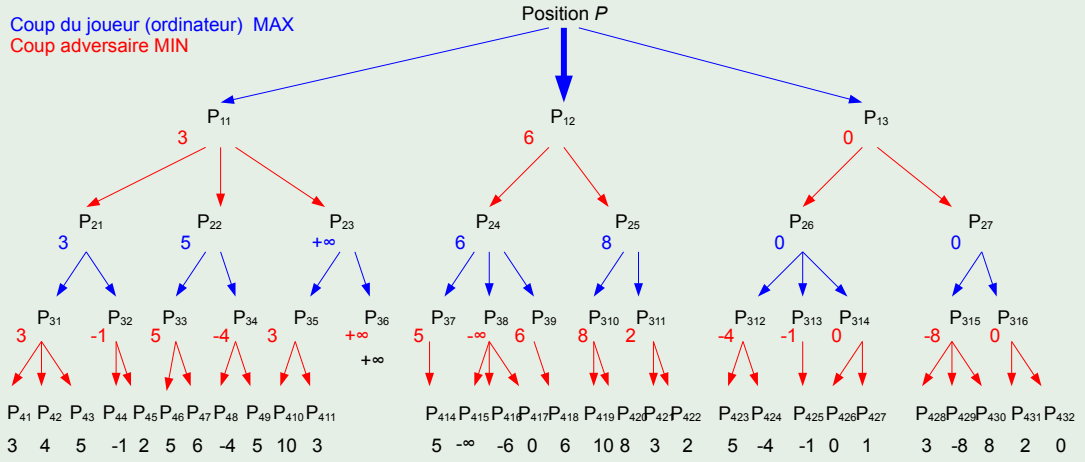


Algorithme MinMax : principe



BY NC SA

Algorithme MinMax : principe



BY NC SA

Analyse descendante 1 / 2

Version informelle et incomplète

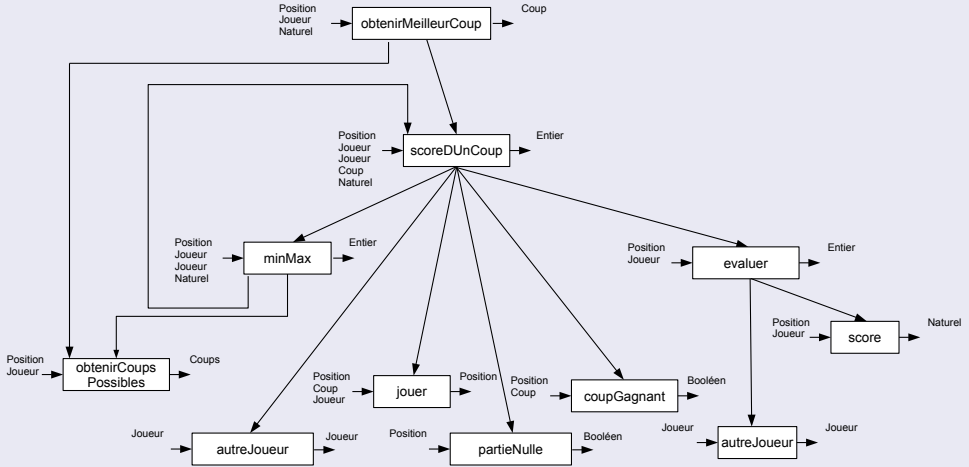
- Le sous-programme de *choix d'un coup* a besoin :
  - d'un *générateur de coups*
  - d'un *calcul de score* d'un coup
- Le sous-programme de *calcul de score* a besoin :
  - d'une *fonction d'évaluation*
  - de l'algorithme *min-max*
- Le sous-programme *min-max* a besoin :
  - d'une *générateur de coups*
  - d'un *calcul de score* d'un coup

INSA ROUEN NORMANDIE

CC BY NC SA

Analyse descendante 2 / 2

Version formelle



Algorithmes 1 / 3

Fonction objenirMeilleurCoup

```
fonction obtenirMeilleurCoup (unePosition : Position, joueur : Joueur, profondeur : Naturel) : Coup
    Déclaration  meilleurCoup : Coup, cpsPossibles : Coups
                 score, meilleurScore : Entier
                 i : Naturel
debut
    cpsPossibles ← obtenirCoupsPossibles(unePosition, joueur)
    meilleurCoup ← iemeCoup(cpsPossibles,1)
    meilleurScore ← scoreDUnCoup(unePosition, meilleurCoup, joueur, joueur, profondeur)
    pour i ←2 à nbcoups(cpsPossibles) faire
        score ← scoreDUnCoup(unePosition, iemeCoup(cpsPossibles,i), joueur, joueur, profondeur)
        si score>meilleurScore alors
            meilleurCoup ← iemeCoup(cpsPossibles,i)
            meilleurScore ← score
        finsi
    finpour
    retourner meilleurCoup
fin
```

CC BY NC SA

Algorithmes 2 / 3

Fonction scoreDUnCoup

```
fonction scoreDUnCoup (unePosition : Position, unCoup : Coup, joueurRef, joueurCourant : Pion, profondeur : Naturel) : Entier
debut
    jouer(unePosition, unCoup, joueurCourant)
    si coupGagnant(unPosition, unCoup) alors
        si joueurCourant=joueurRef alors
            retourner +∞
        sinon
            retourner −∞
        finsi
    sinon
        si partieNulle(unPosition) alors
            retourner 0
        sinon
            si profondeur=0 alors
                retourner evaluer(unePosition, joueurRef)
            sinon
                retourner minMax(unePosition, joueurRef, autreJoueur(joueurCourant), profondeur-1)
            finsi
        finsi
    finsi
fin
```


BY NC SA

Algorithmes 3 / 3

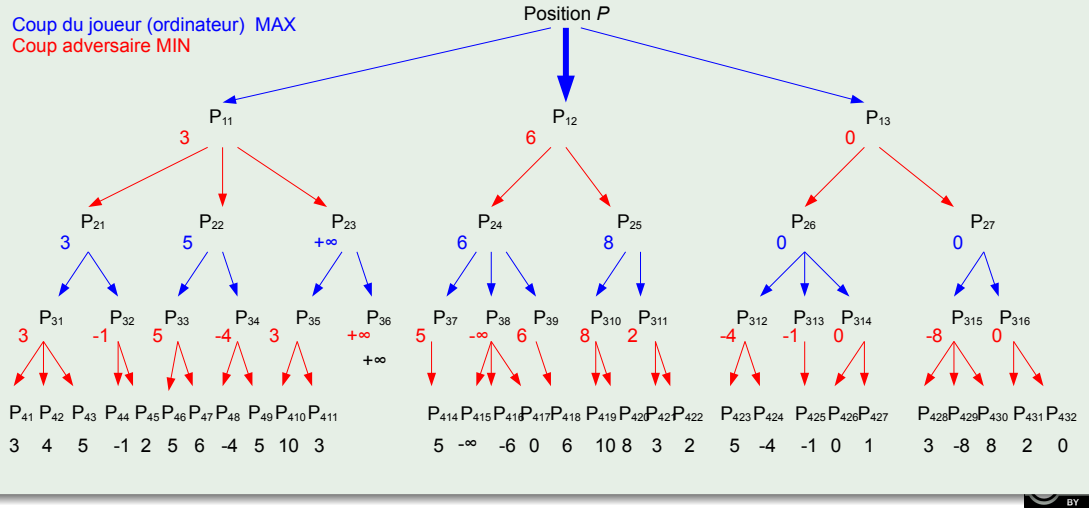
Fonction minMax

```
fonction minMax (unePosition : Position, joueurRef, joueurCourant : Joueur, profondeur : Naturel) : Entier
    Déclaration  scoreMinMax : Entier, cpsPossibles : Coups, score : Entier, i : Naturel
debut
    cpsPossibles ← obtenirCoupsPossibles(unPosition, joueurCourant)
    scoreMinMax ← scoreDUnCoup(unePosition, iemeCoup(cpsPossibles,1), joueurRef, joueurCourant,profondeur)
    pour i ←2 à nbCoups(cpsPossibles) faire
        score ← scoreDUnCoup(unePosition, iemeCoup(cpsPossibles,i), joueurRef, joueurCourant, profondeur)
        si joueurCourant=joueurRef alors
            scoreMinMax ← max(scoreMinMax,score)
        sinon
            scoreMinMax ← min(scoreMinMax,score)
        finsi
    finpour
    retourner scoreMinMax
fin
```

ROUEN NORMANDIE

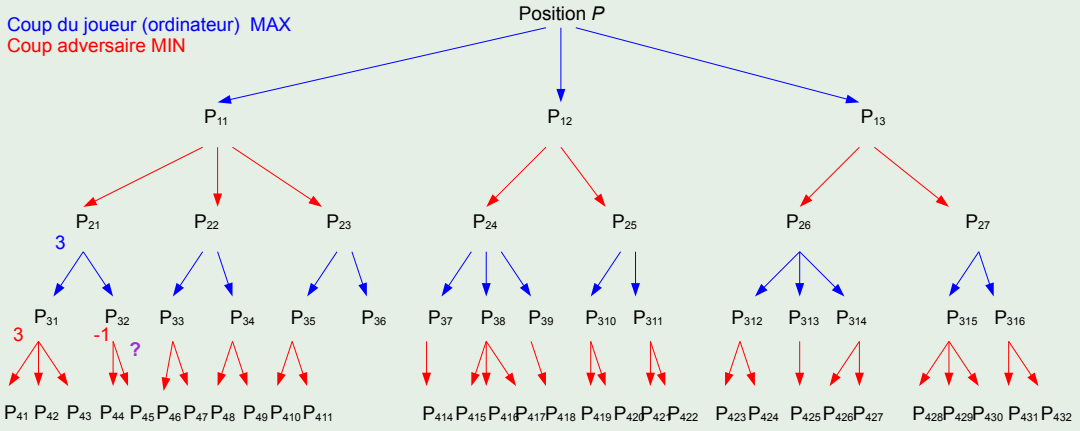
BY NC SA

Explorer toutes les branches est-il utile ?

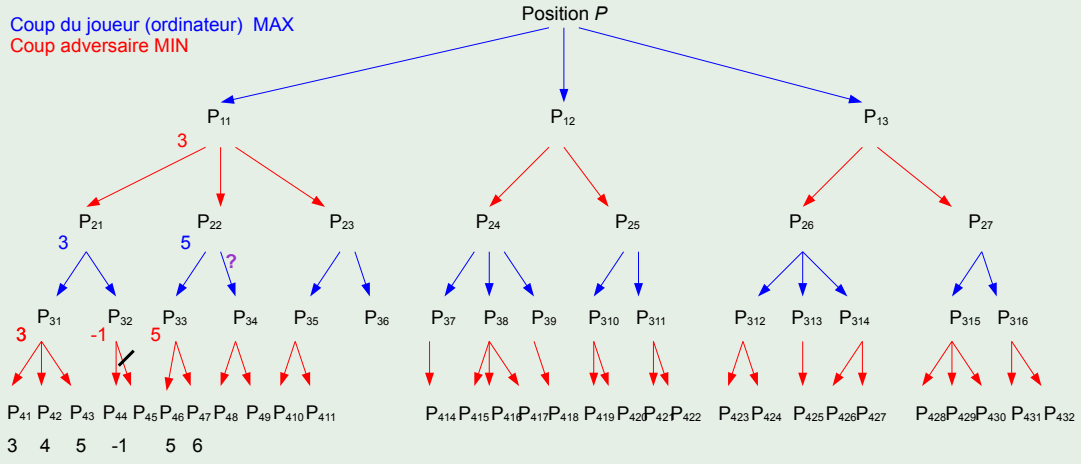


BY NC SA

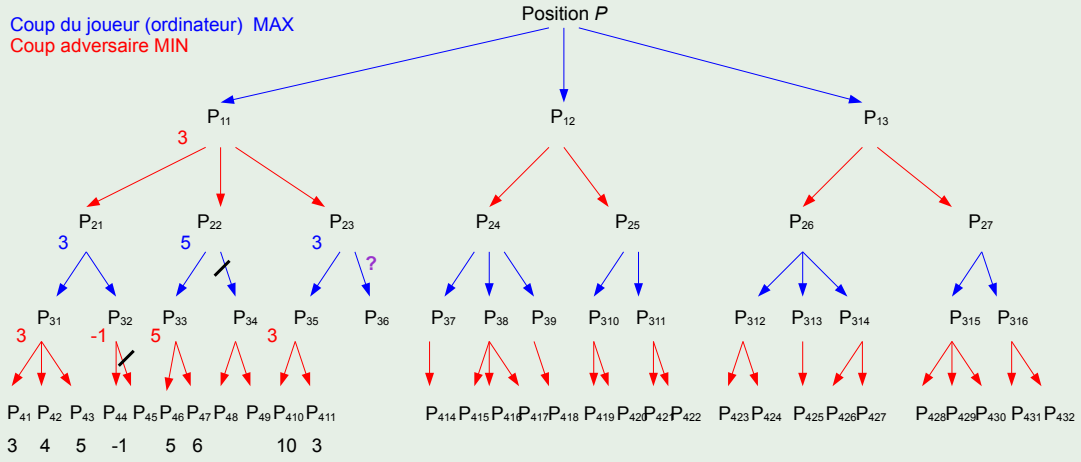
Élégage Alpha-Béta : principe



Élégage Alpha-Béta : principe

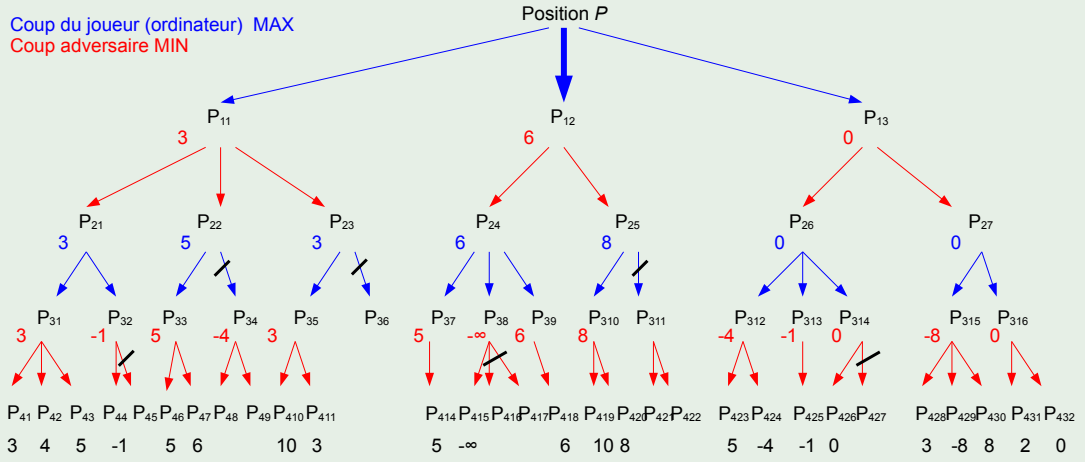


Élégage Alpha-Béta : principe



BY NC SA

Élégage Alpha-Béta : principe



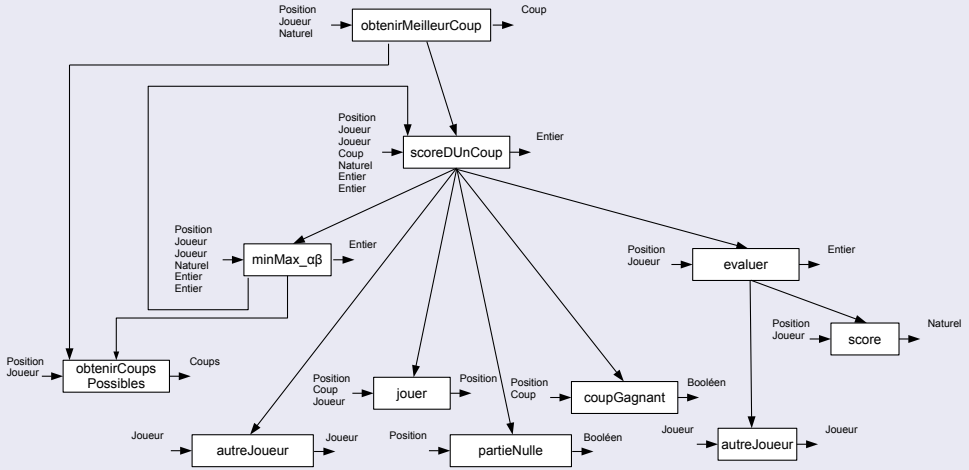
Élagage Alpha-Beta 1 / 3

Principe

- Les sous programmes `scoreDUnCoup` et `minMax_`$\alpha\beta$ prennent deux paramètres formels supplémentaires :
  - α : le meilleur score que le joueur maximisant (ordinateur) peut garantir à ce niveau ou au-dessus (le plus élevé)
  - β : le meilleur score que le joueur minimisant (adversaire) peut garantir à ce niveau ou au-dessus (le plus bas)
- Ces deux paramètres sont donc passés aux sous-programmes `scoreDUnCoup` et `minMax_`$\alpha\beta$ et ils sont mis à jour dans `minMax_`$\alpha\beta$ en fonction des résultats des appels à `scoreDUnCoup` :
  - Au premier appel à `scoreDUnCoup` (depuis `obtenirMeilleurCoup`), $\alpha = -\infty$ et $\beta = +\infty$
  - Dans le sous-programme `minMax_`$\alpha\beta$:
    - Si on est au niveau d'un min et que le score calculé (par appel à `scoreDUnCoup`) est inférieur ou égal à α, on arrête l'exploration de cette branche (le joueur maximisant ne choisira jamais cette branche), sinon on met à jour β ($\beta = min(\beta, score)$)
    - Si on est au niveau d'un max et que le score calculé (par appel à `scoreDUnCoup`) est supérieur ou égal à β, on arrête l'exploration de cette branche (le joueur minimisant ne choisira jamais cette branche), sinon on met à jour α ($\alpha = max(\alpha, score)$)

Élagage Alpha-Beta 2 / 3

Version formelle avec élagage Alpha-Beta

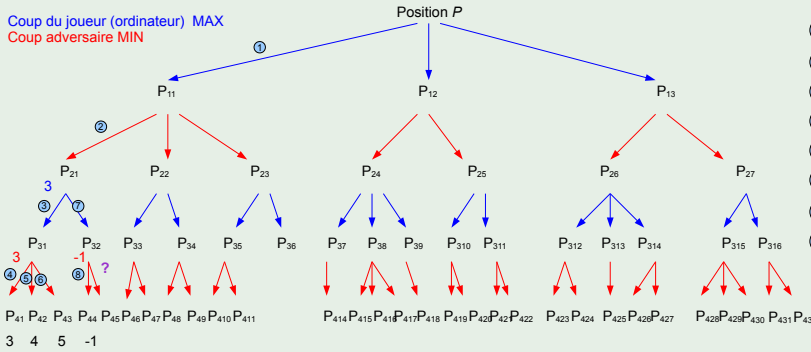


Élagage Alpha-Beta 3 / 3

Fonction minMax_$\alpha\beta$

```
fonction minMax_αβ (unePosition : Position, joueurRef, joueurCourant : Joueur, profondeur : Naturel, alpha, beta : Entier) : Entier
    Déclaration  scoreMinMaxαβ : Entier, cpsPossibles : Coups, score : Entier, i : Naturel
debut
    cpsPossibles ← obtenirCoupsPossibles(unePosition, joueurCourant)
    i ← 1
    continue ← VRAI
    si joueurCourant=joueurRef alors scoreMinMaxαβ← −∞ sinon scoreMinMaxαβ← +∞
    repeter
        score ← scoreDUnCoup(unePosition, iemeCoup(cpsPossibles,i), joueurRef, joueurCourant, profondeur, alpha, beta)
        si joueurCourant=joueurRef alors
            scoreMinMaxαβ ← max(scoreMinMaxαβ,score)
            si scoreMinMaxαβ≥beta alors
                continue ← FAUX
            finsi
            alpha ← max(alpha,scoreMinMaxαβ)
        sinon
            scoreMinMaxαβ ← min(scoreMinMaxαβ,score)
            si scoreMinMaxαβ≤alpha alors
                continue ← FAUX
            finsi
            beta ← min(beta,scoreMinMaxαβ)
        finsi
        i ← i+1
    jusqu'a ce que non continue et i>nbCoups(cpsPossibles)
    retourner scoreMinMaxαβ
fin
```

Extrait de l'arbre des appels à *minMax_αβ*



α β

1. minMax_αβ(P,o,o,c_{11},4,-∞,+∞)
2. minMax_αβ(P_{11},a,o,c_{21},3,-∞,+∞)
3. minMax_αβ(P_{21},o,o,c_{31},2,-∞,+∞)
4. minMax_αβ(P_{31},a,o,c_{41},1,-∞,+∞) → 3
5. minMax_αβ(P_{31},a,o,c_{42},1,-∞,3) → 4
6. minMax_αβ(P_{31},a,o,c_{43},1,-∞,3) → 5
7. minMax_αβ(P_{21},o,o,c_{32},2,3,+∞)
8. minMax_αβ(P_{32},a,o,c_{44},1,3,+∞) → -1

On n'évalue pas *minMax_αβ(P_{32},a,o,c_{45},1,3,+∞)*
car -1 est plus petit que α qui vaut 3

Exploration partielle mais complete : méthode de Monté Carlo 1 / 3

Principe

- Au lieu d'explorer toutes les possibilités jusqu'à une profondeur donnée, on effectue un grand nombre de parties aléatoires (simulations) depuis la position courante
- On choisit ensuite le coup qui a conduit statistiquement aux meilleurs résultats lors des simulations
- Paramètres de l'algorithme :
  - Stratégie de choix des coups durant les simulations
  - Profondeur maximale des simulations
  - Nombre de simulation à effectuer

INSA ROUEN NORMANDIE

cc BY NC SA

Exploration partielle mais complete : méthode de Monté Carlo 2 / 3

Stratégie de choix des coups durant les simulations

- Choix du coup aléatoire
  - Très simple à implémenter
  - Très bruité, résultats très variables
- Utilisation d'une heuristique
  - Variance faible, résultats plus fiables, nécessite moins de simulations
  - Difficulté d'implémentation et temps de calcul

Profondeur maximale

- Partie jusqu'à la fin
  - Résultats plus fiables
  - Temps de calcul très long, pas de fonction d'évaluation
- Partie jusqu'à une profondeur donnée
  - Temps de calcul plus court
  - Résultats moins fiables, nécessite une fonction d'évaluation

Exploration partielle mais complete : méthode de Monté Carlo 3 / 3

Nombre de simulations à effectuer

- Une limite théorique (borne de Hoeffding), nombre de simulations par coup :

$$n \geq \frac{1}{2\epsilon^2} ln(\frac{2k}{\delta})$$

  - n : nombre de simulations
  - k : nombre de coups possibles
  - ϵ : précision désirée
  - δ : niveau de confiance

- Limites pratiques :
  - Jeux simples (tic-tac-toe, petits puzzles) : 10 à 1000
  - Jeux moyens (Othello, petits jeux de plateau) : 10 à 10^5
  - Jeux complexes (Échecs, Go, etc.) : 10^5 à 10^7
- Facteurs qui réduisent les besoins : qualité du choix des coups, profondeur des simulations, réutilisation d'arbre de coups

Conclusion

Nous avons vu...

- Différentes approches pour la prise de décision en IA décisionnelle
- Approche algorithmique : recherche de chemin dans un graphe d'états
  - Force brute
  - Dijkstra
  - A*
- Jeux à deux joueurs à somme nulle
  - Exploration avec limitation en profondeur : MinMax, Alpha-Beta
  - Exploration partielle mais complète : Monte Carlo

Mais l'IA c'est aussi...

- Approche par apprentissage : apprentissage supervisé, non supervisé, par renforcement
- Approche par logique : systèmes experts, logique propositionnelle et du premier ordre, résolution par unification

BY NC SA