I4 – Découverte de l'Intelligence Artificielle

Introduction & résolution par parcours exploratoire

Laurent Vercouter
Laurent.vercouter@insa-rouen.fr

Un bref historique de l’IA...

Premières traces vers 1950

Alan Turing
1912-1954

Test de Turing : mise en interaction d’un humain avec une interface derrière laquelle agit un humain ou une machine. S’il ne sait pas distinguer entre les 2, le test est réussi

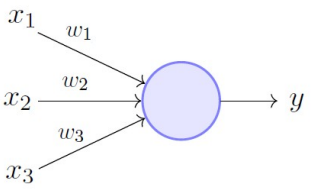
Conférence de Darmouth, 1956

Naissance du terme « Artificial Intelligence »

« la construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisante par des êtres humains car elles demandent des processus mentaux de haut niveau tels que : l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique » (Marvin Minsky)

Dans les années 50/60

- Systèmes logiques formels, logique floue
- Réseaux de neurones, perceptron



Perceptron Model (Minsky-Papert in 1969)

- Premier chat-bot : Eliza

Certains experts prédisent que « des machines seront capables, d’ici 20 ans, de faire le travail que toute personne peut faire »

1974-1980 : 1er hiver de l'IA

Les déceptions sont à la hauteur des promesses follement optimistes…

- Le modèle du perceptron est fortement critiqué et presque enterré
- Les moyens de calcul de l'époque sont trop limités
- De nombreux problèmes sont démontrés trop difficiles (NP-complets)

→ Les financements et projets se détournent de l'IA

1980 : 2ème âge d’or

- Systèmes experts, systèmes à base de règles
  - Diagnostics automatiques (Dendral, Mycin, Xcon, ...)
- Perceptron multi-couches
  - Importants progrès en reconnaissance d’écriture, en classification
- Réseaux bayésiens
  - Raisonnement causal probabiliste

1987-1993 : 2ème hiver de l’IA

Puis dans les années 1990/2000

- Algorithmes génétiques
- Apprentissage automatique
- Logique temporelle, preuves
- Représentation des connaissances
- IA distribuée, systèmes multi-agents

Depuis 2010, un 3ème age d’or

Avec des succès retentissants pour le grand public dans les jeux

1997, Deep Blue (IBM)
pour les échecs

2011, Watson (IBM)
au jeopardy

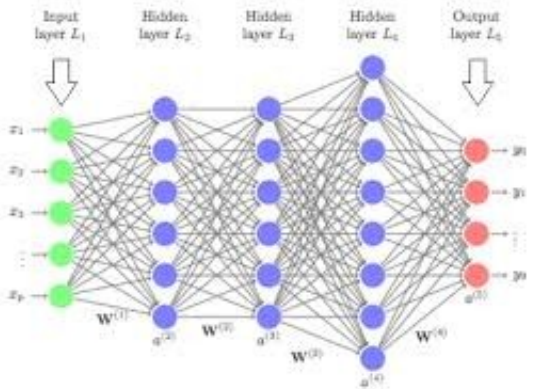
2016, Alpha Go (Google)
au Go

2017, Libratus (CMU)
au Poker

Depuis 2010, un 3ème age d’or

Progrès spectaculaires avec l’arrivée de l’apprentissage profond (deep learning) et l’explosion des moyens de calculs en

- Reconnaissance visuelle (santé, véhicule autonome, …)
- Robotique
- Traduction
- Reconnaissance de la parole, du son



Aujourd’hui nous sommes en plein dans cet âge d’or, mais de nombreux enjeux sociétaux, éthiques, juridiques, écologiques sont soulevés pour l’avenir proche.

Et maintenant l’IA générative

DeepL

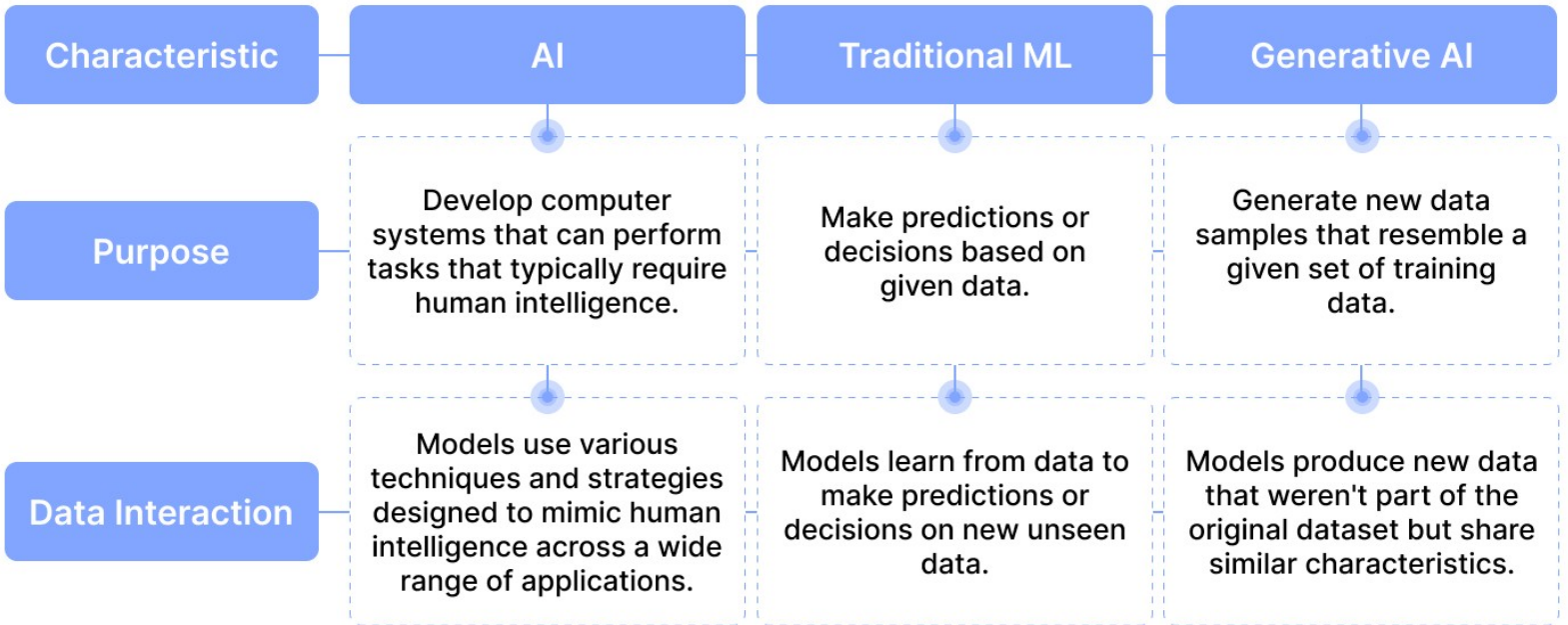
ChatGPT

Gemini

GitHub Copilot

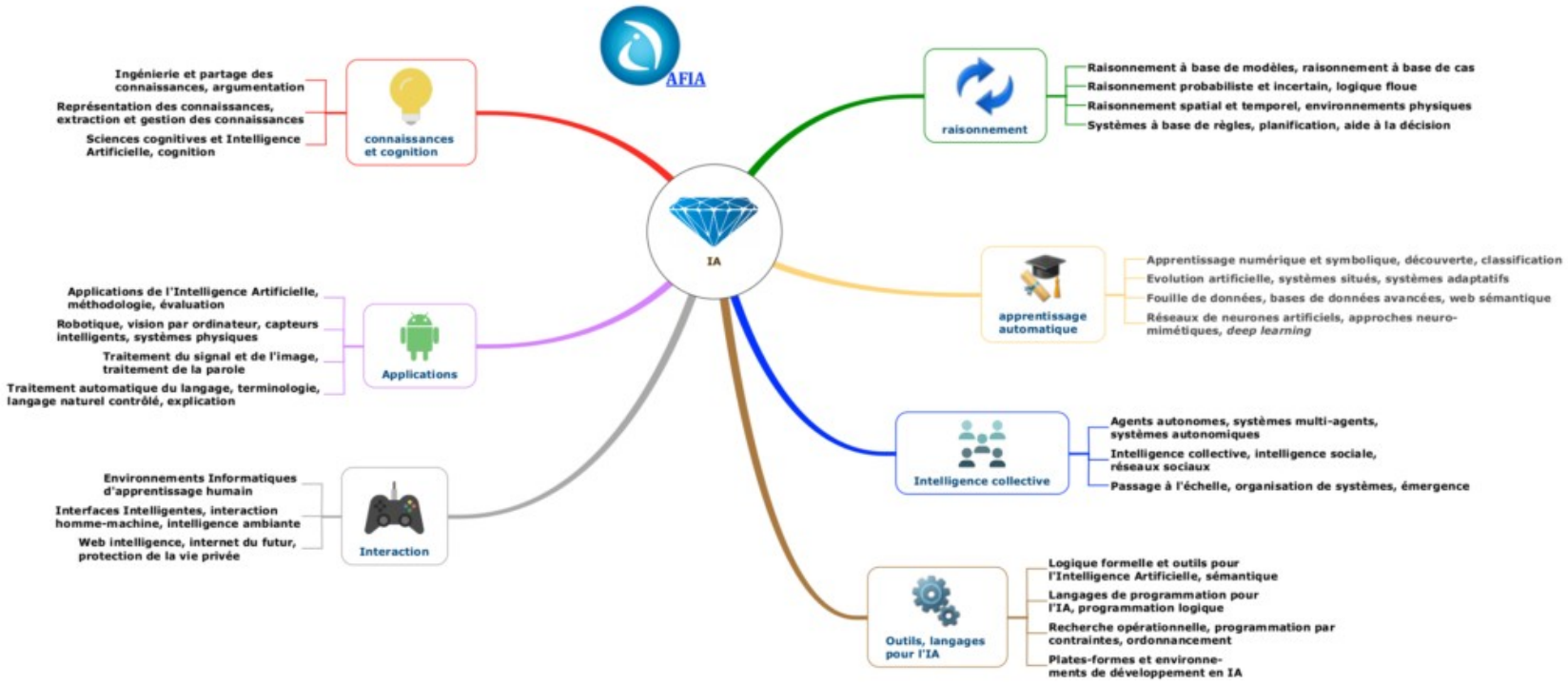
DALL·E 2

Artificial Intelligence vs. Traditional Machine Learning, Generative AI



Source : https://www.techopedia.com/definition/34633/generative-ai

L’IA aujourd’hui



Selon l’Association Française pour l’Intelligence Artificielle

Organisation du cours I4

Planning

- Python
- Résolution de problèmes par parcours exploratoire
- Apprentissage automatique
- Raisonnement logique à base de connaissances

Langages

- Python
- Prolog

Evaluation

- Examen final avec une partie théorique sur le cours et 2 exercices sur machine

Résolution de problèmes par parcours exploratoire

Résolution de problèmes

1. Définir la nature du problème

2. Représentation formelle
   - Logique
   - Graphes d’états

3. Algorithme de résolution

Résolution de problèmes

Exemple

1. Définir la nature du problème

2. Représentation formelle
   - Logique
   - Graphes d'états

3. Algorithme de résolution

1. « Quel trajet optimal pour passer au moins une fois par toutes les lignes de métro ? »[1]
   - Nature : recherche d'un chemin optimal

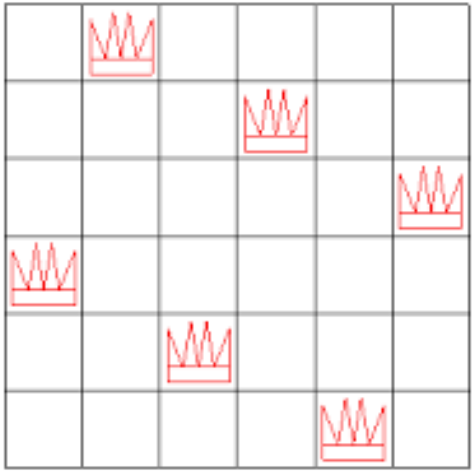
2. Représentation par un graphe
   - 1 noeud = 1 station de métro
   - 1 action = 1 déplacement d'une station vers une autre
   - Buts = 1 ensemble d'actions tq chaque noeud est visité au moins 1 fois
   - Etat = séquence d'actions réalisée depuis un état initial

3. Algorithme de parcours du graphe

[1] https://interstices.info/quel-trajet-optimal-pour-passer-au-moins-une-fois-par-toutes-les-lignes-de-metro/

Exemple : le problème des *n* reines

Comment placer n reines sur un échiquier de nxn cases sans qu'aucune ne puisse se prendre ?

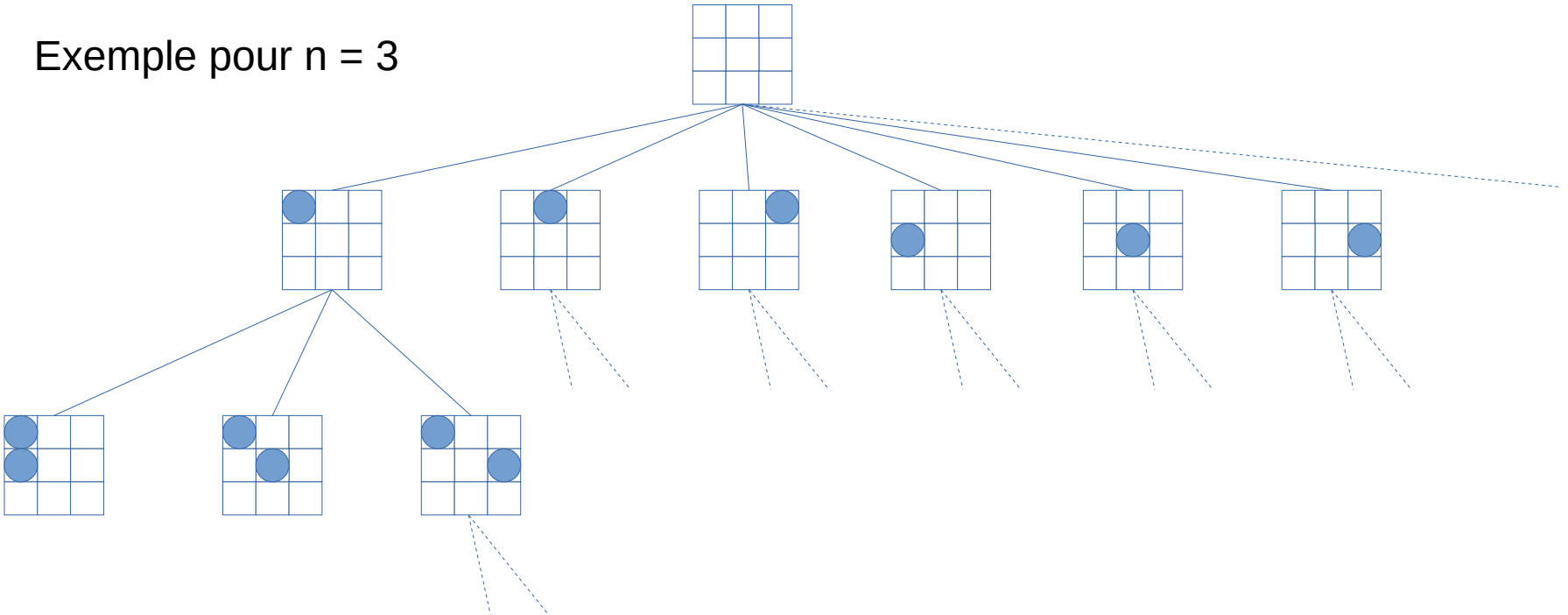


Espace d’états des *n* reines

- 1 état = 1 configuration de l’échiquier
- État initial = échiquier vide
- Action = placer une reine
- 1 algorithme de résolution possible
  - Placer la première reine, puis la suivante sur une case possible, etc

Espace d’états des *n* reines

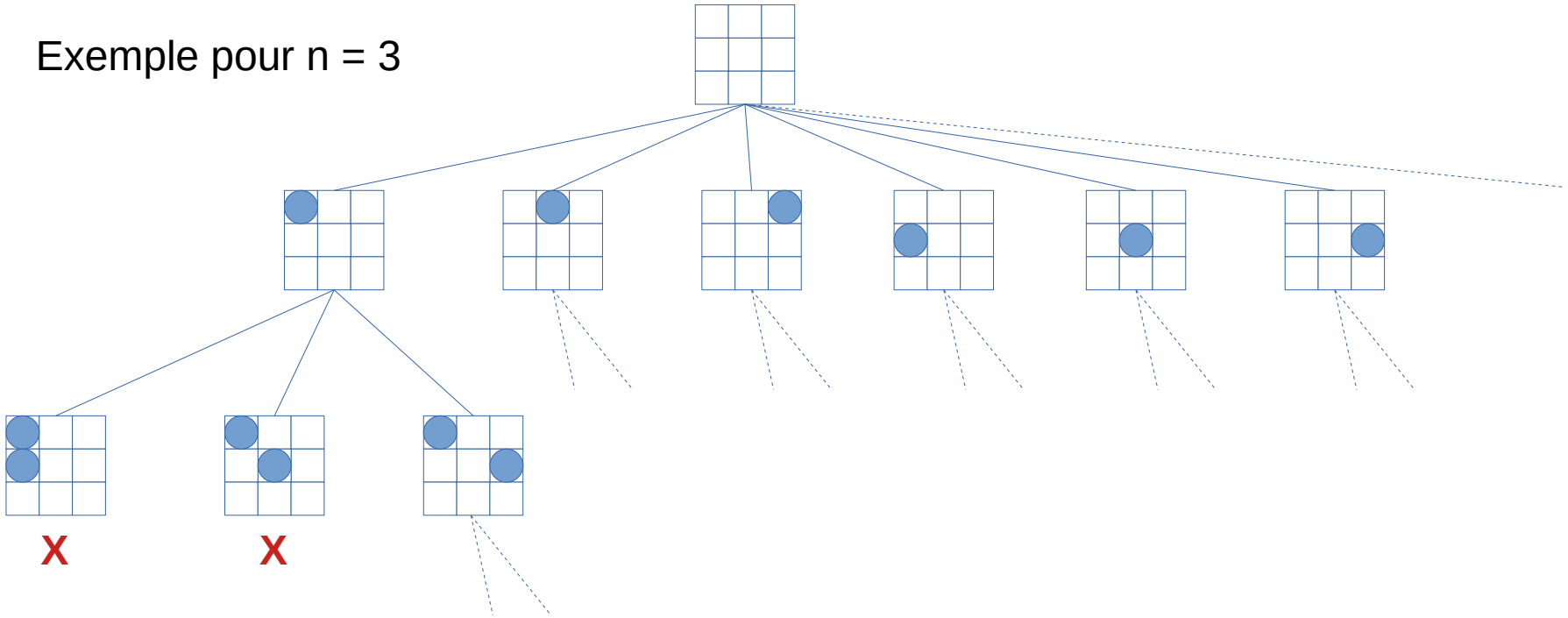
Exemple pour n = 3



Combien d’états possibles ?

Espace d’états des *n* reines

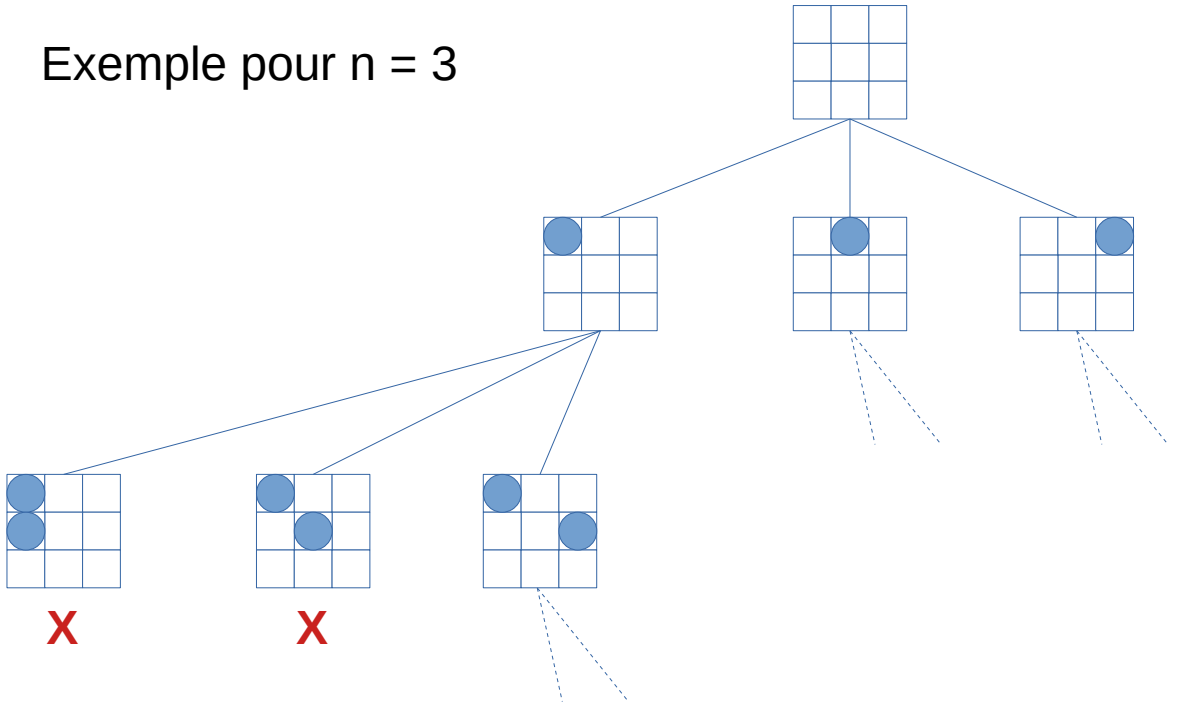
Exemple pour n = 3



Amélioration 1 : arrêter l’exploration quand l’état est un échec

Espace d’états des *n* reines

Exemple pour n = 3



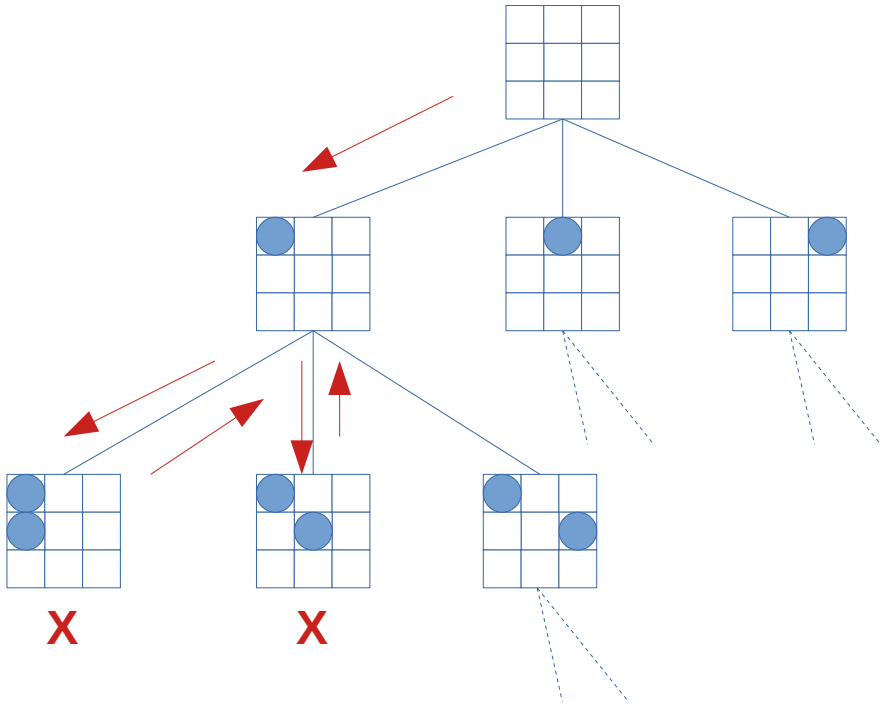
Reine de la ligne 1

Reine de la ligne 2

Reine de la ligne 3

Amélioration 2 : guider l’exploration avec une connaissance experte

Parcours de l’espace d’états



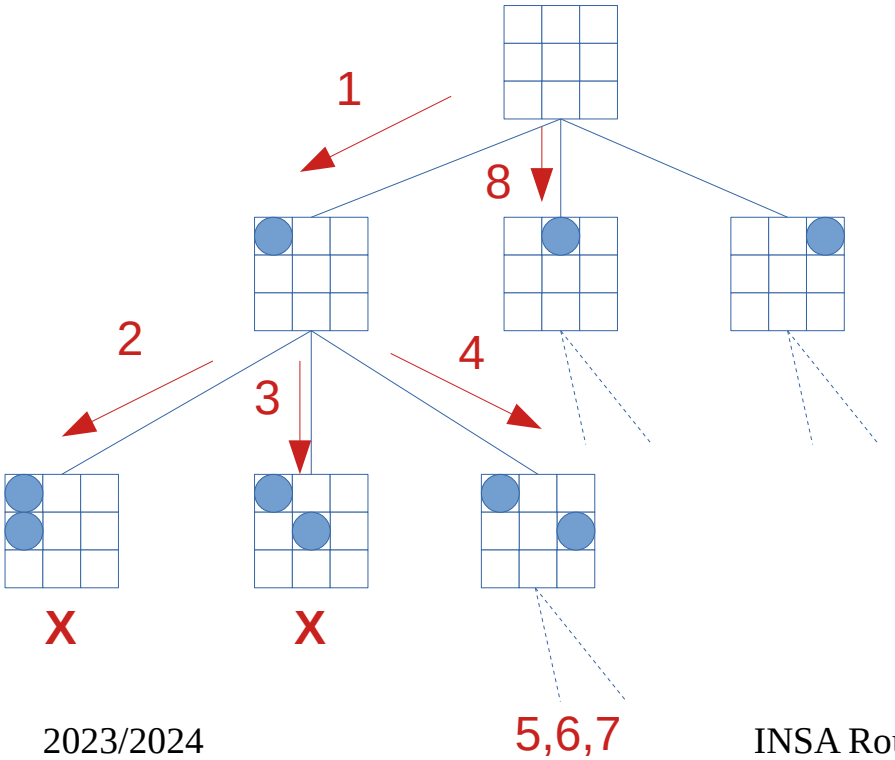
Principe du ***backtracking*** :

Il consiste en la possibilité à revenir à un état passé dans le parcours d’un graphe d‘états

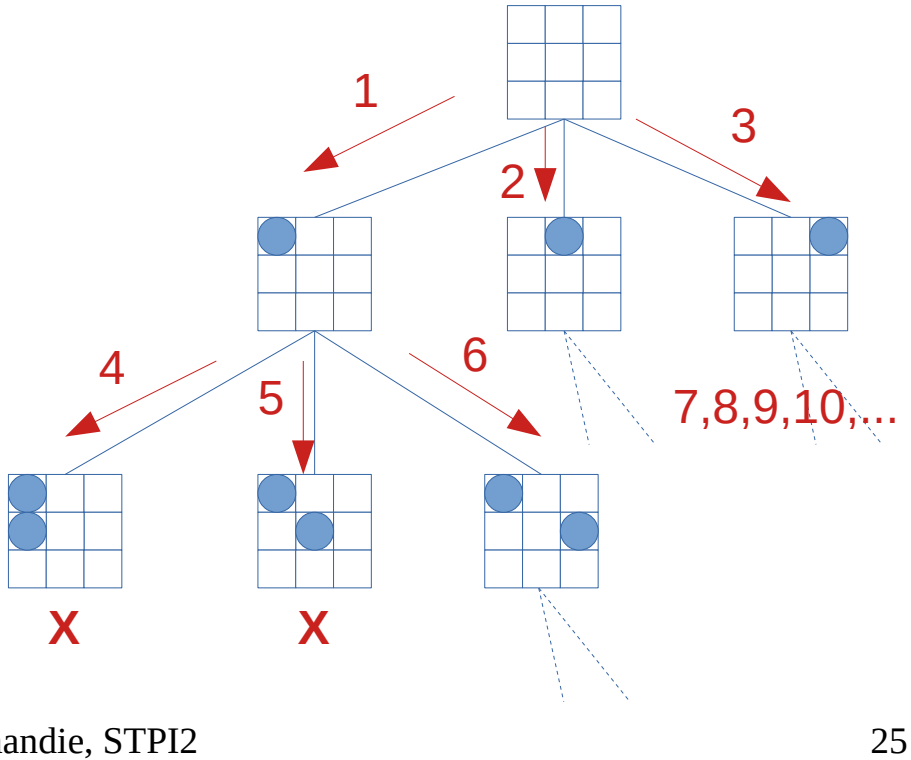
Le retour en arrière doit apporter des informations sur les états parcourus (réussite/échec, scores, ...)

Parcours de l’espace d’états

Parcours en ***profondeur***



Parcours en ***largeur***



Récursivité

Pour développer un algorithme de parcours d'espace nous devons introduire le principe de fonction récursive.

Une fonction récursive est une fonction

- qui peut s'appeler elle-même
- qui a une ou des conditions de terminaison où elle arrête de s'appeler et effectue un backtrack

Récursivité

Pour développer un algorithme de parcours d'espace nous devons introduire le principe de fonction récursive.

Une fonction récursive est une fonction

- qui peut s'appeler elle-même
- qui a une ou des conditions de terminaison où elle arrête de s'appeler et effectue un backtrack

Exemple en Python d'une fonction récursive pour calculer un factoriel :

```
def factoriel(n):
    if n <= 1:
        return 1
    fact = factoriel(n-1) * n
    return fact

print("n = ")
val = int(input())
print(val,"! = ",factoriel(val))
```

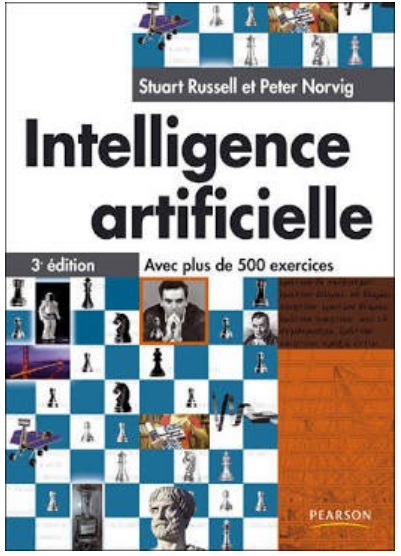
Fonction récursive de parcours en profondeur

```
Fonction explore(actions,état)
    si terminal(etat,actions) = Vrai alors
        retourner (ø,evaluation(etat))
    sinon
        meilleureAction ← actions[0]
        nouvelEtat ← transition(etat,meilleureAction)
        nouvellesActions ← actionsPossibles(etat,actions,meilleureAction)
        meilleurScore ← explore(nouvellesActions, nouvelEtat)
        pour chaque a dans actions faire
            nouvelEtat ← transition(etat,a)
            nouvellesActions ← actionsPossibles(etat,actions,a)
            score ← explore(nouvellesActions, nouvelEtat)
            si score > meilleurScore faire
                meilleurScore ← score
                meilleureAction ← a
            finsi
        finpour
        retourner (meilleureAction,meilleurScore)
    finsi
```

Types de problèmes

- Jeu à 1 joueur
- Jeu à 2 joueurs
  - Alternance d’actions avec des objectifs opposés
- Exploration complète ou limitée
  - Problème de complexité algorithmique
- Connaissances complètes ou incomplètes
  - Incertitude sur l’état courant
- Action déterministe ou indéterministe
  - Transitions probabilistes

Pour aller plus loin



MOOC de vulgarisation « avancée »
https://www.elementsofai.com/
https://www.elementsofai.fr/

Algorithmes gloutons

Méthodes heuristiques

- Une exploration aveugle ou complète peut être impossible à réaliser
- Une approche heuristique est une méthode pour orienter une recherche vers une solution non-optimale dans des problèmes d'optimisation difficiles
  - une heuristique dépend du problème à traiter
  - La qualité de l'heuristique a une influence sur la qualité de la solution et sur la difficulté de mise en oeuvre

Algorithmes gloutons (*greedy*)

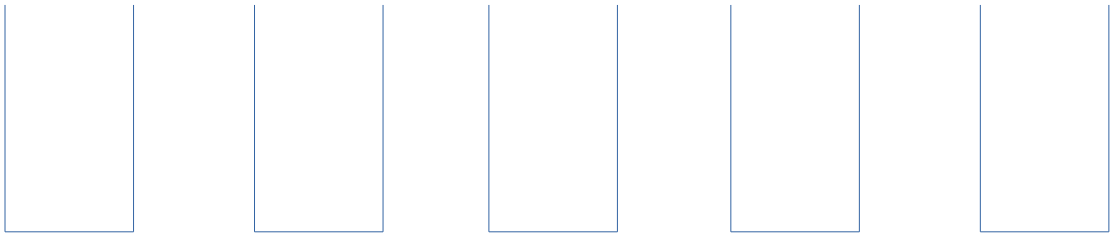
- Le principe est d'avancer par « gourmandise » vers l'état offrant la meilleure récompense
- Nécessité d'avoir une fonction d'évaluation des états
- Construction d'une solution pas à pas sans revenir en arrière
- Algorithme optimal pour certains problèmes, mais suboptimal pour d'autres

Exemple : problème d'empaquetage

- Un ensemble d'objets E = {1,…, n} de poids p_1, …, p_n
- Problème : placer les objets dans des boîtes
  - En respectant leur capacité
  - En utilisant le moins de boîtes possibles
- Méthode gloutonne : choix successifs de l'objet le plus lourd à placer dans la première boîte possible

Exemple : problème d’empaquetage

Poids 7 6 3 4 8 5 9 2

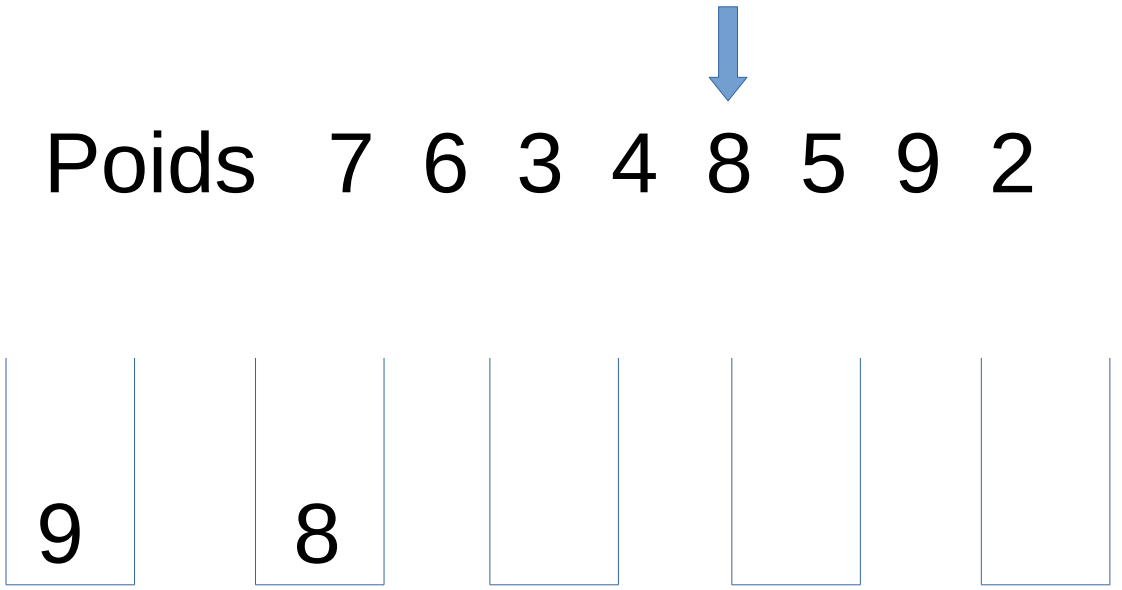


Capacité = 11

Exemple : problème d’empaquetage

Capacité = 11

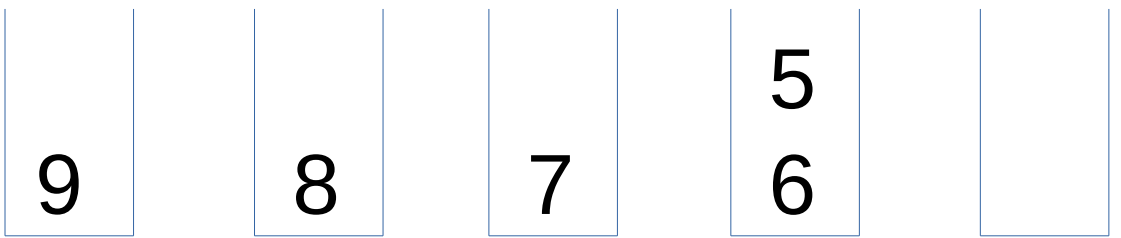
Exemple : problème d’empaquetage



Capacité = 11

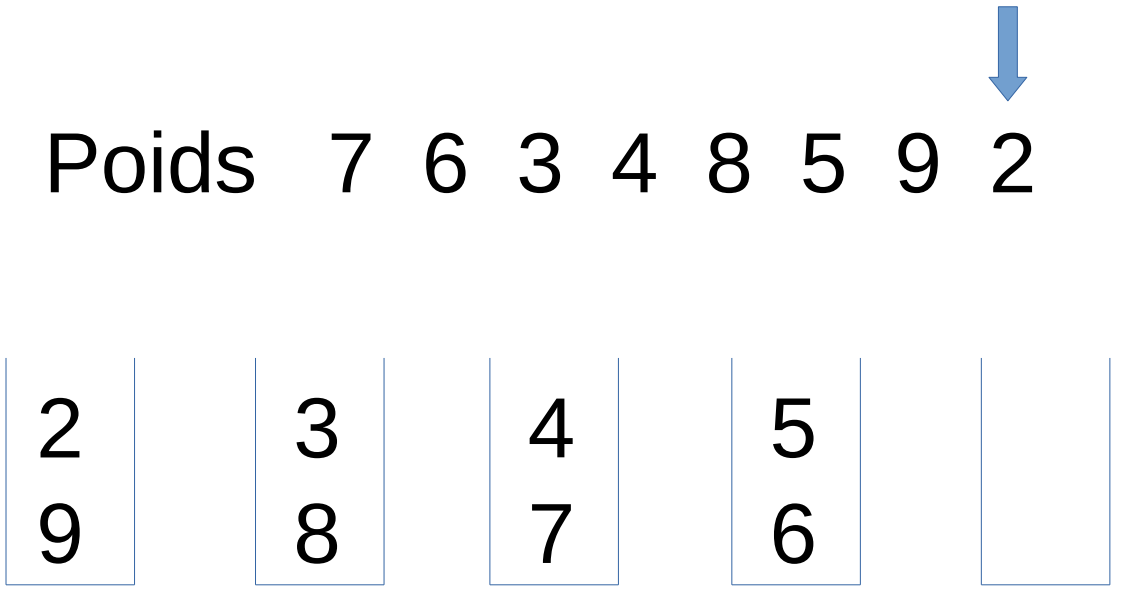
Exemple : problème d’empaquetage

Poids 7 6 3 4 8 5 9 2



Capacité = 11

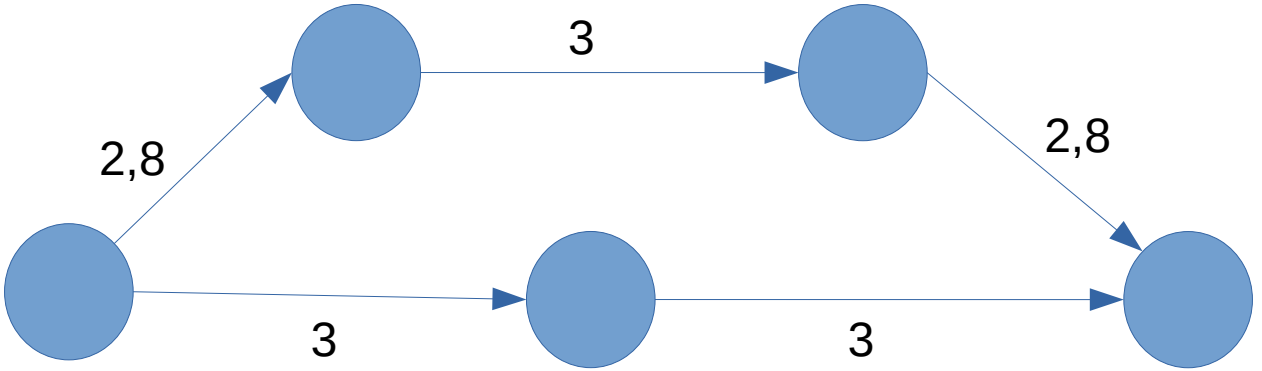
Exemple : problème d’empaquetage



Capacité = 11

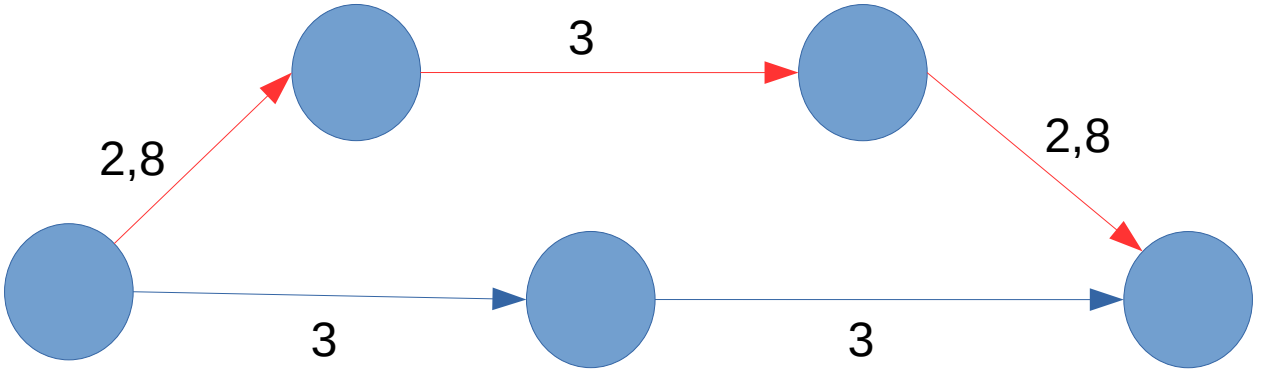
Exemple : recherche de chemin

- Objectif : trouver le chemin le plus court
- Algorithme glouton : à partir d'un point donné, prendre la transition la plus courte



Exemple : recherche de chemin

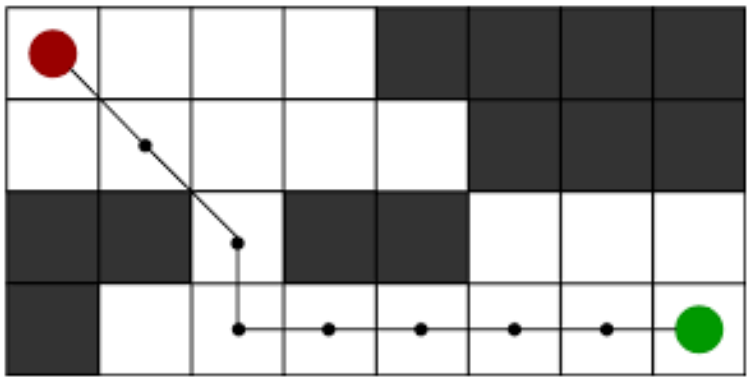
- Objectif : trouver le chemin le plus court
- Algorithme glouton : à partir d'un point donné, prendre la transition la plus courte

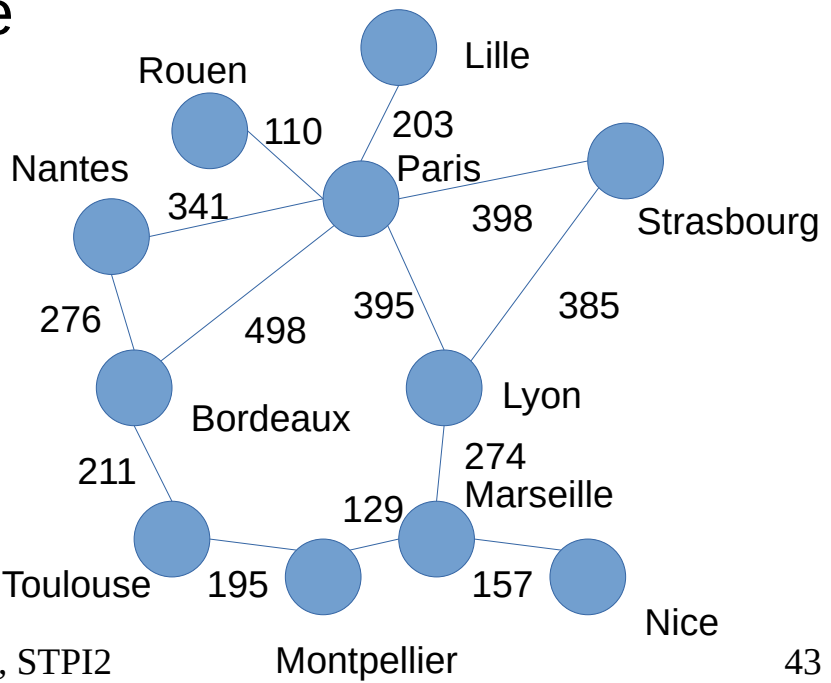


Recherche de plus court chemin

Plus court chemin

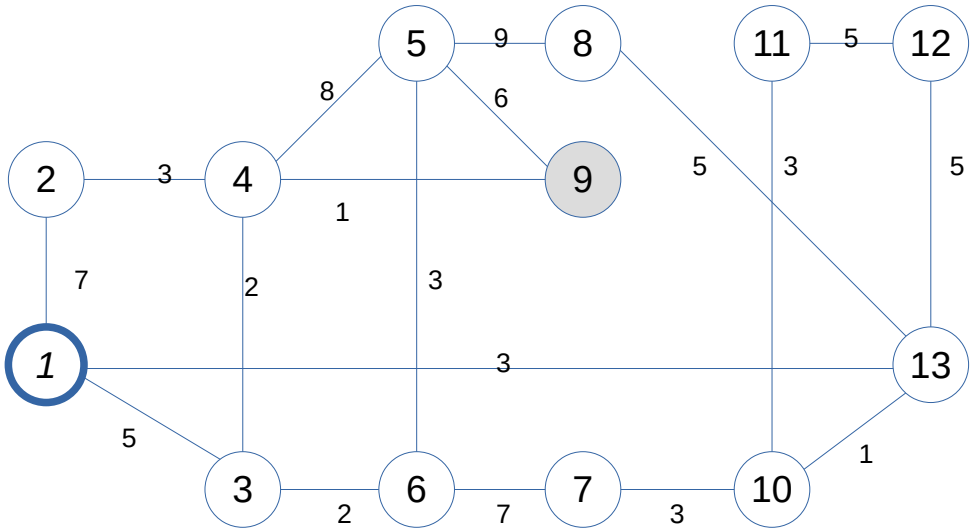
Le problème du plus court chemin consiste à trouver une suite optimale de déplacement à partir d'une localisation initiale pour rejoindre une destination finale donnée





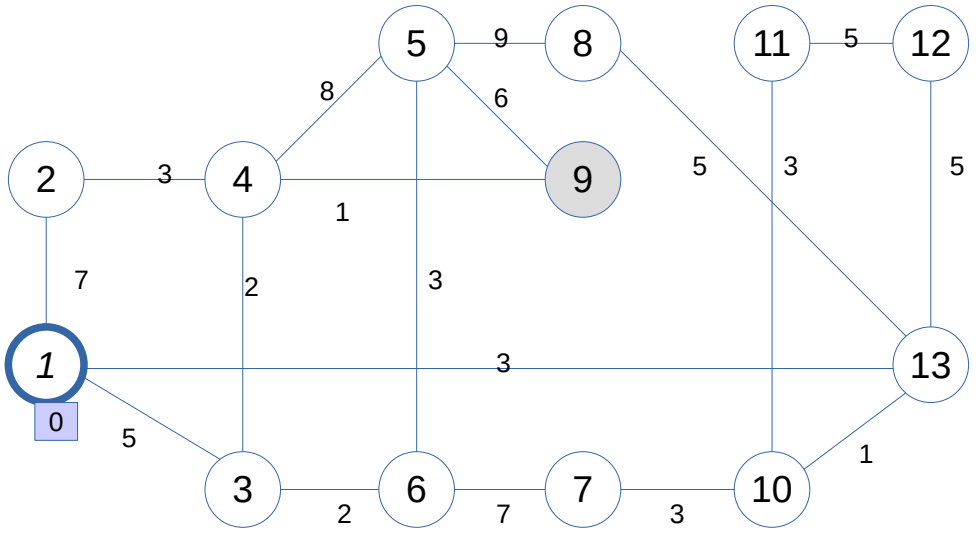
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



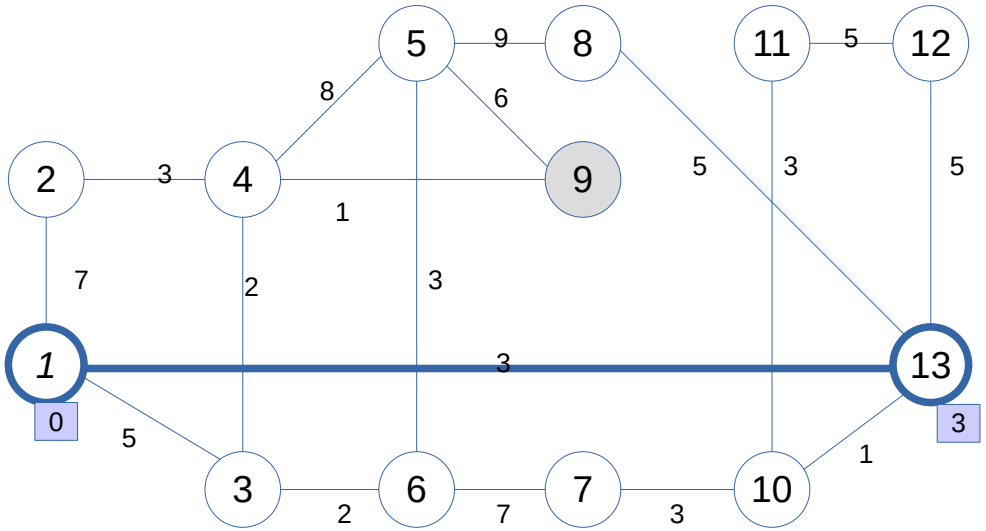
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



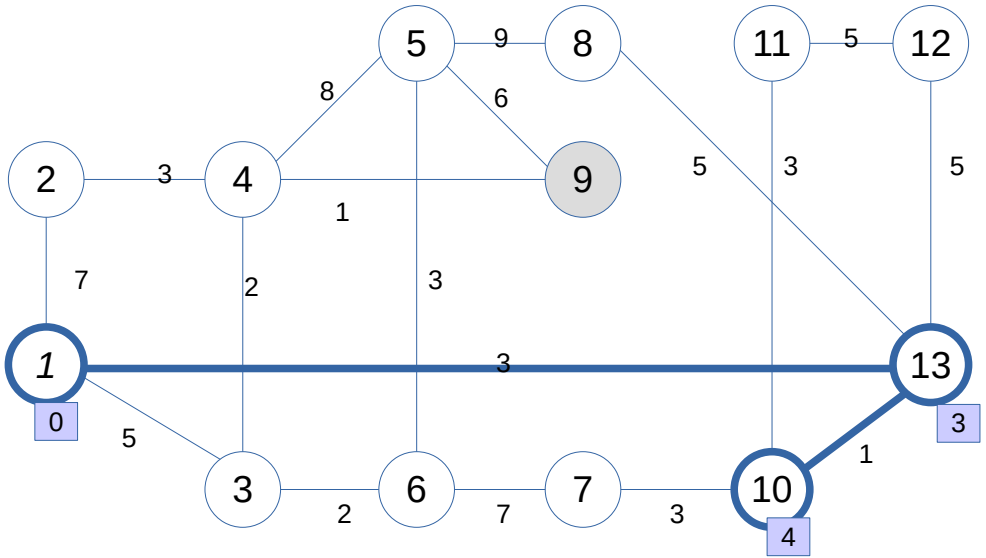
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



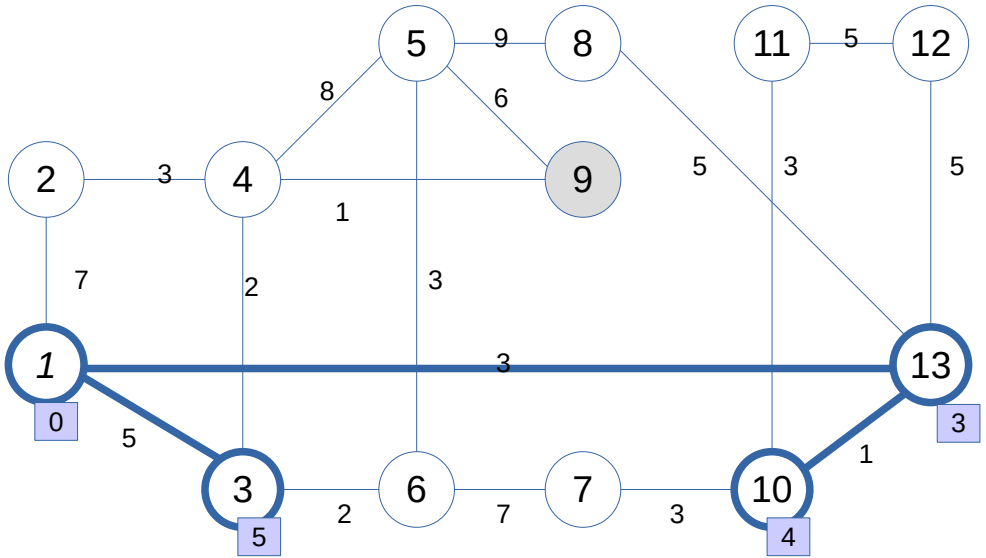
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



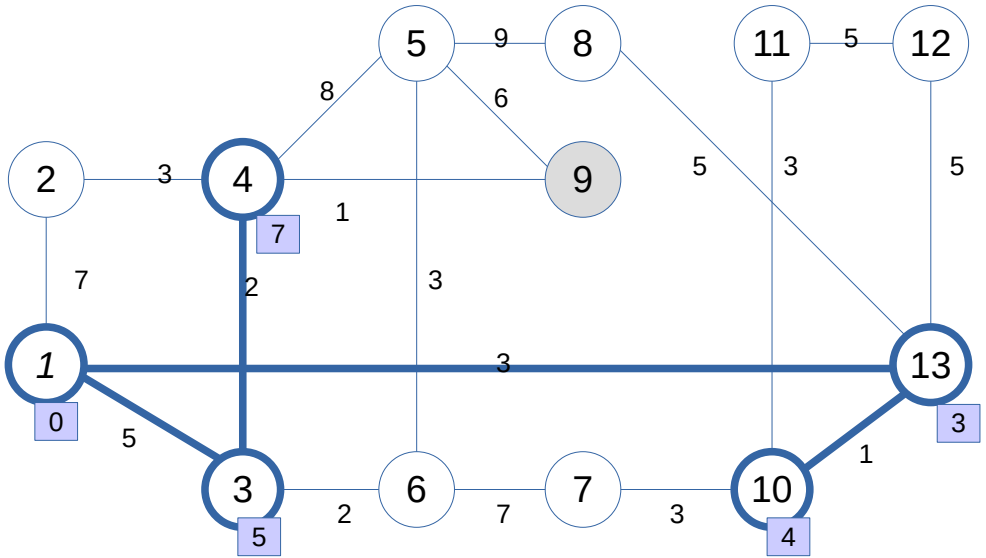
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



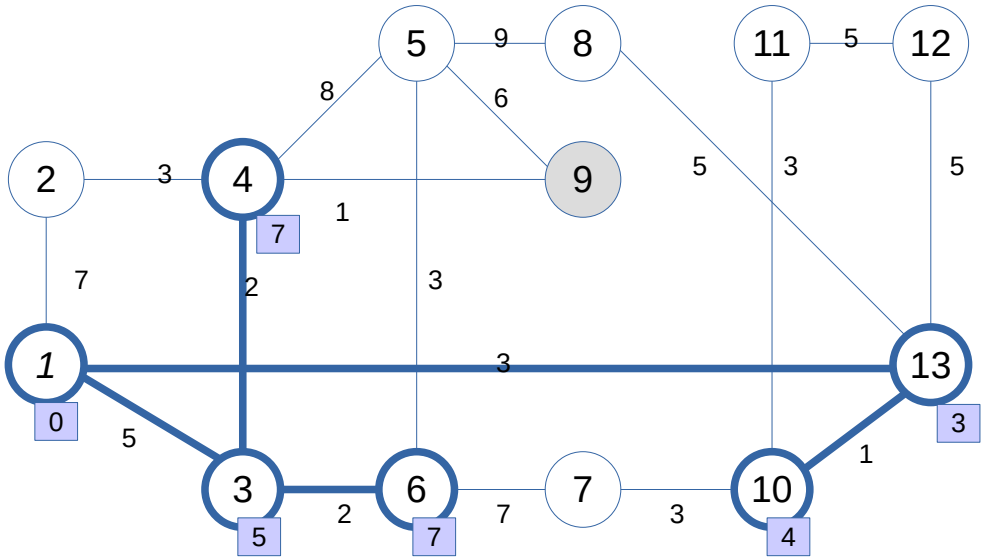
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



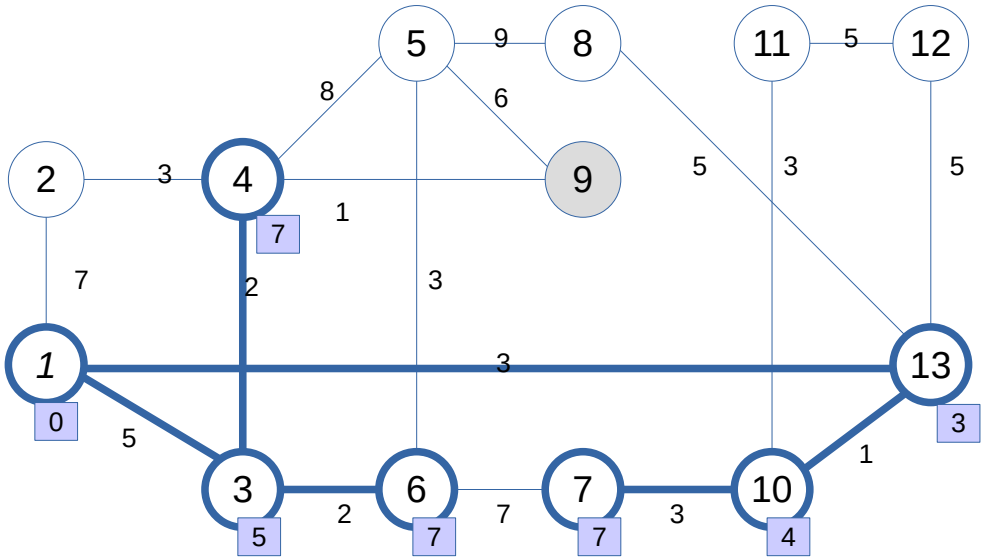
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



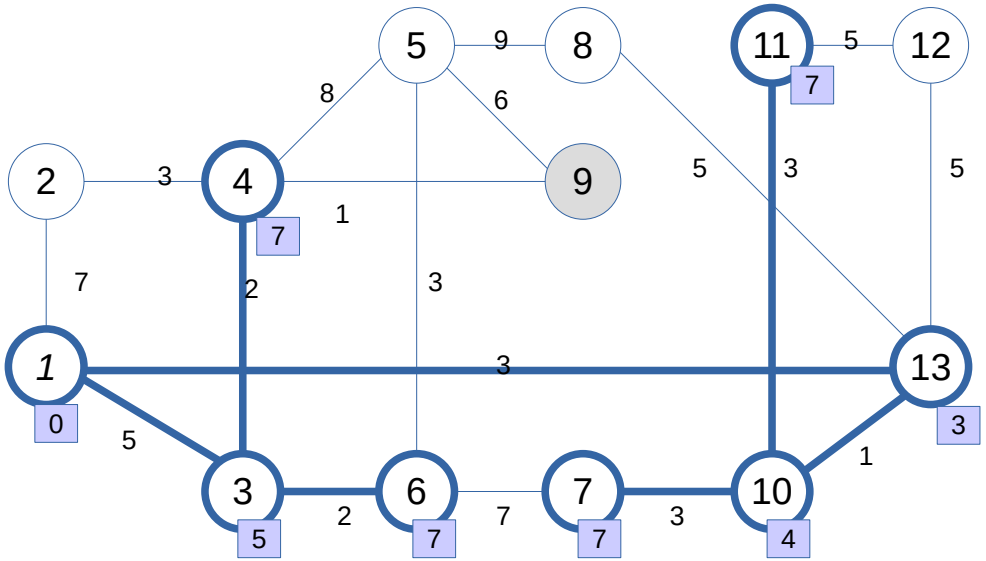
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



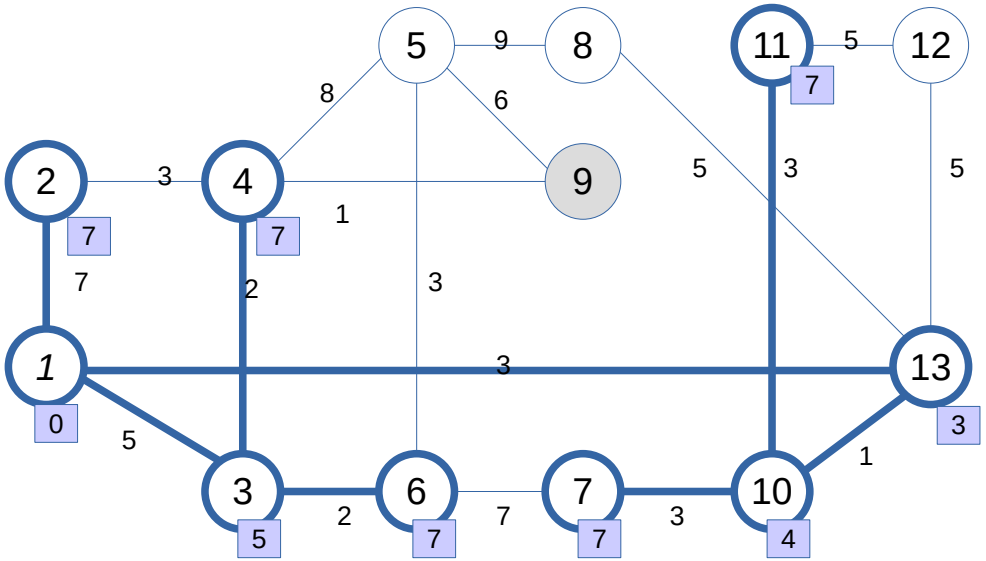
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



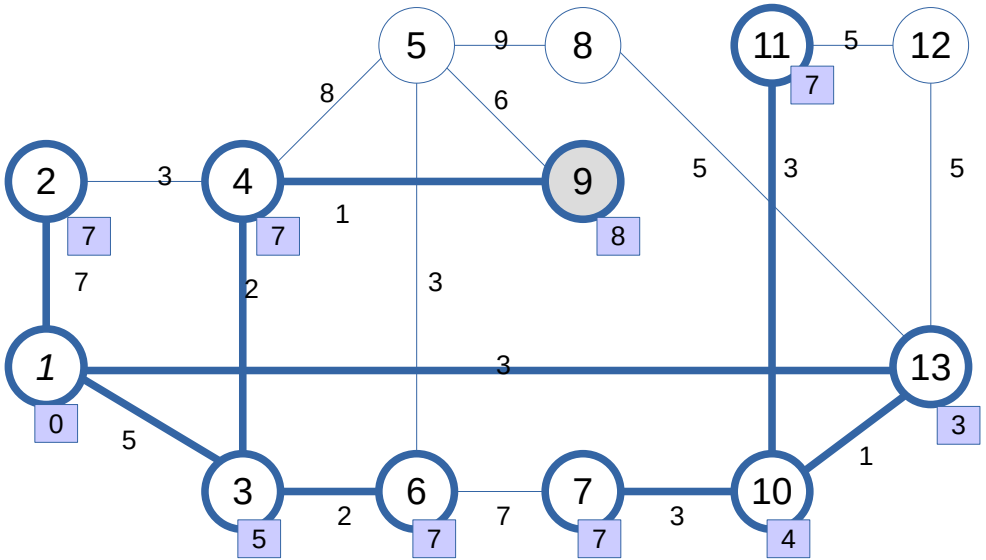
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



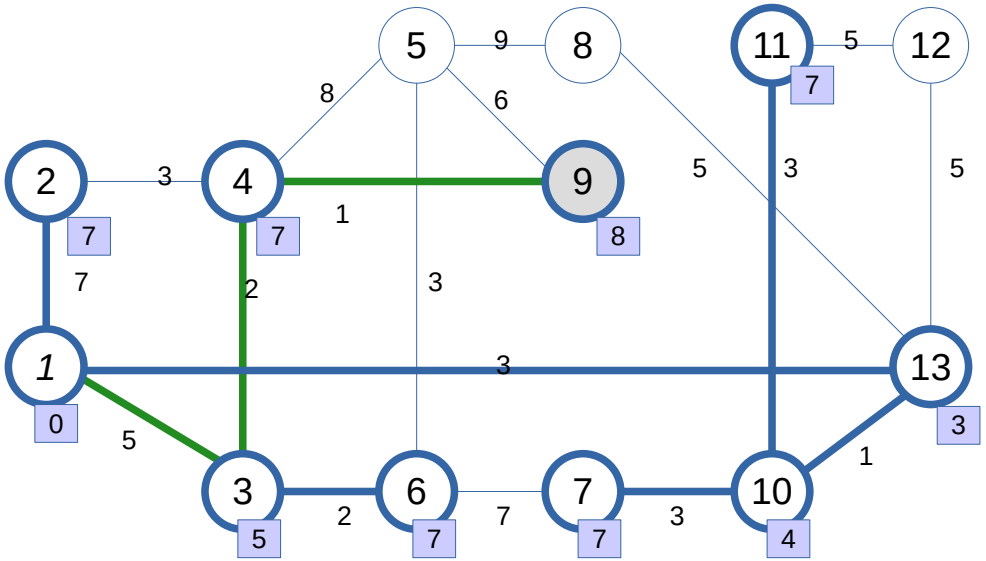
Dijkstra un algo sans heuristique

- Principe : recouvrir le graphe par la création d'un arbre minimal



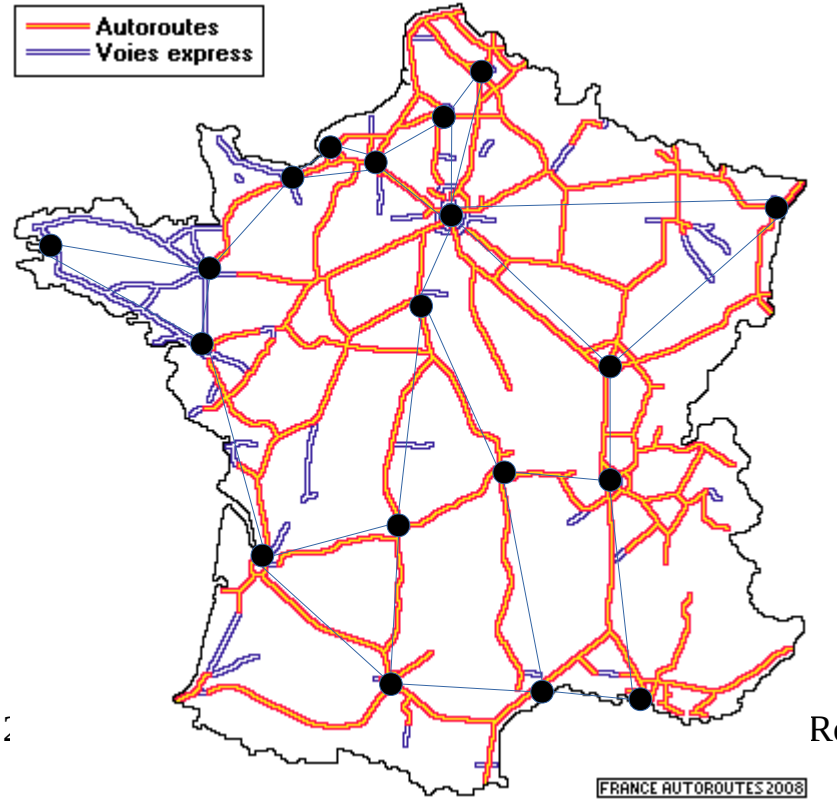
Dijkstra un algo sans heuristique

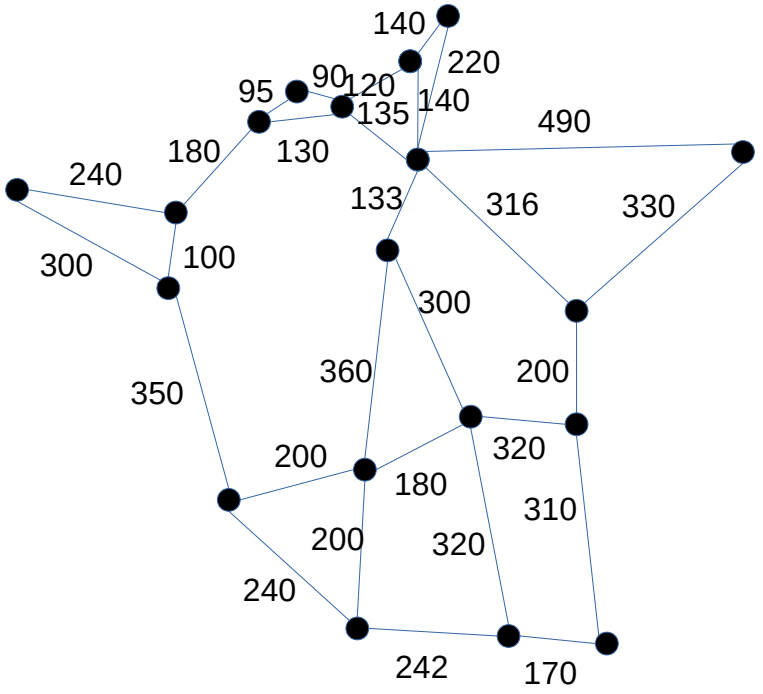
- Principe : recouvrir le graphe par la création d'un arbre minimal



Dijkstra un algo sans heuristique

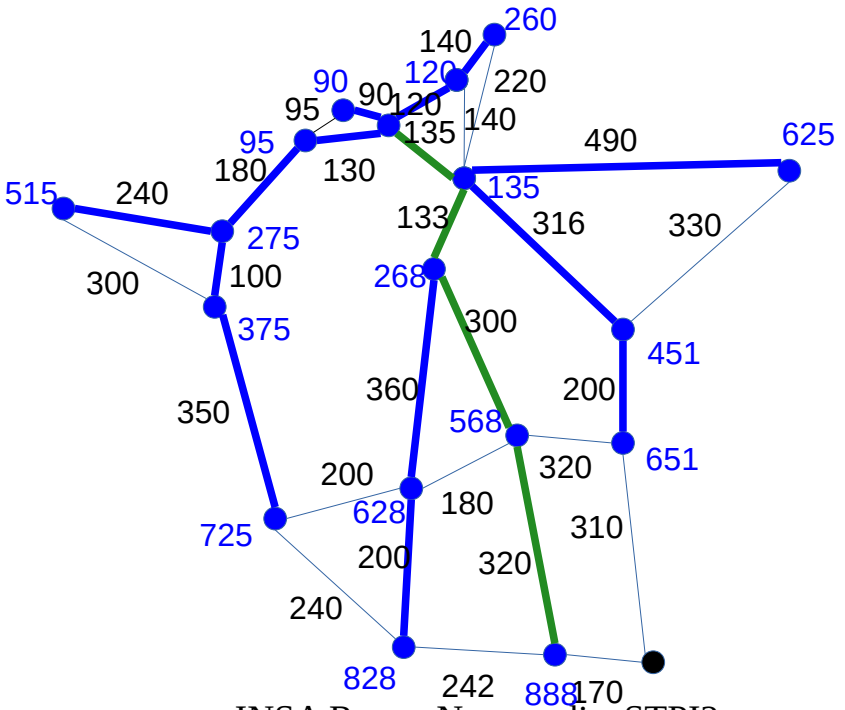
- Quelque fois pas efficace...





Dijkstra un algo sans heuristique

- Pour aller de Rouen à Montpellier...



Algorithme A*

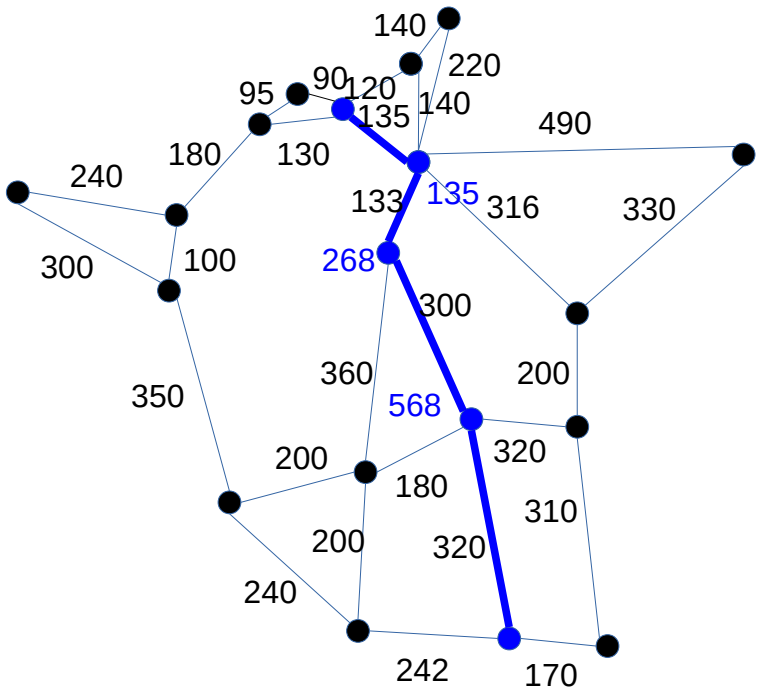
- Proposé par Nils Nilsson en 1968
- Adaptation de l'algorithme de Dijkstra qui réalise un parcours en largeur pour trouver tous les plus courts chemins à partir d'un nœud initial
- Le principe d'A* consiste à parcourir les nœuds dans un ordre défini par une heuristique
  - Exemple pour A* : distance euclidienne entre le nœud et la destination

Algorithme A*

- Soit *n* un nœud (une localisation)
  - *g*(n)* est le coût minimum entre le nœud de départ n_0 et *n*
  - *h*(n)* est le coût minimal des chemins du nœud *n* à la destination n_s
  - *f*(n) = g*(n) + h*(n)* est le coût du chemin solution optimal de n_0 à n_s en passant par *n*
- Il faut définir
  - Une heuristique *h(n)*
  - *g(n)* pour aller de n_0 à *n*

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



Un exemple en ligne

http://qiao.github.io/PathFinding.js/visual/

Jeux à 2 joueurs

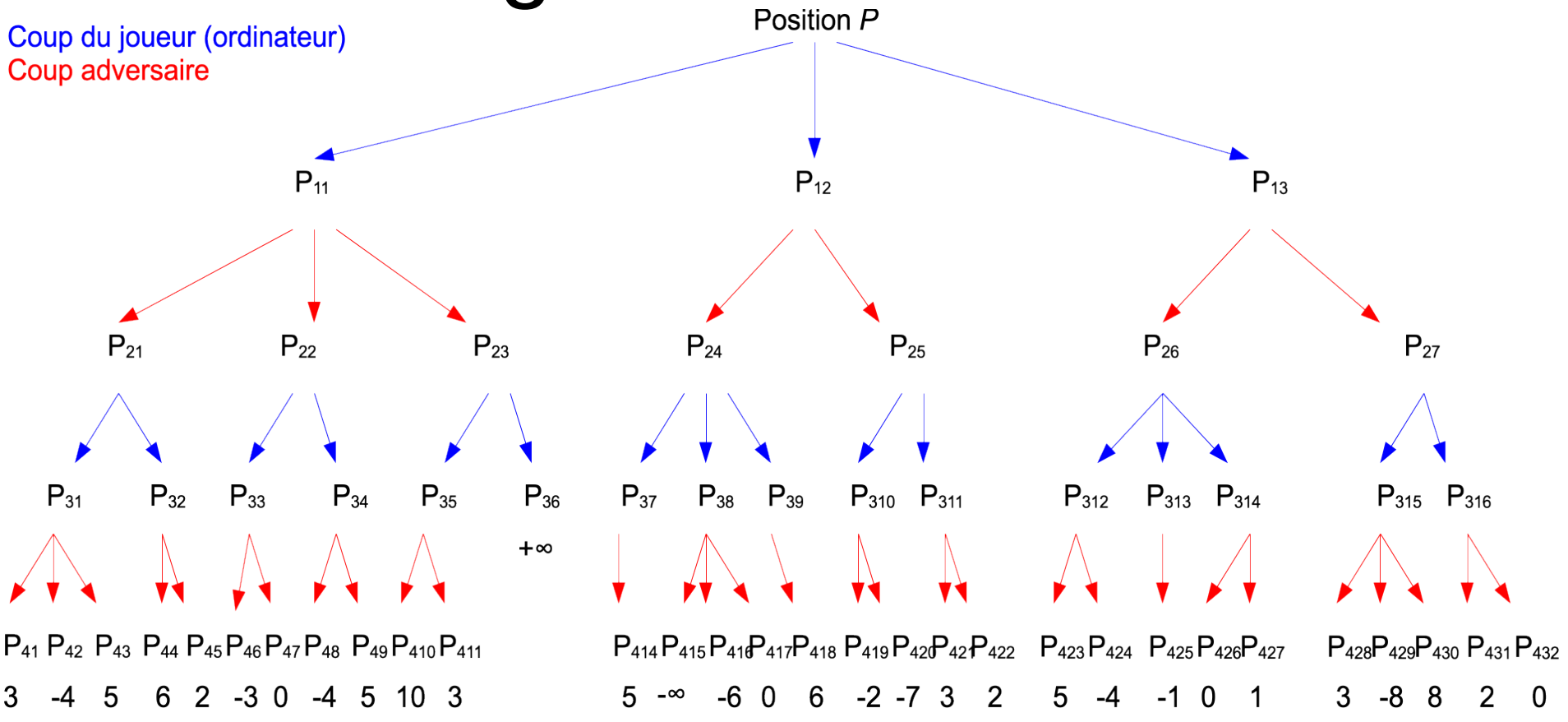
Jeu à 2 joueurs

Dans un jeu où 2 joueurs s’opposent, chacun explorera l’espace d’états vers des objectifs opposés.

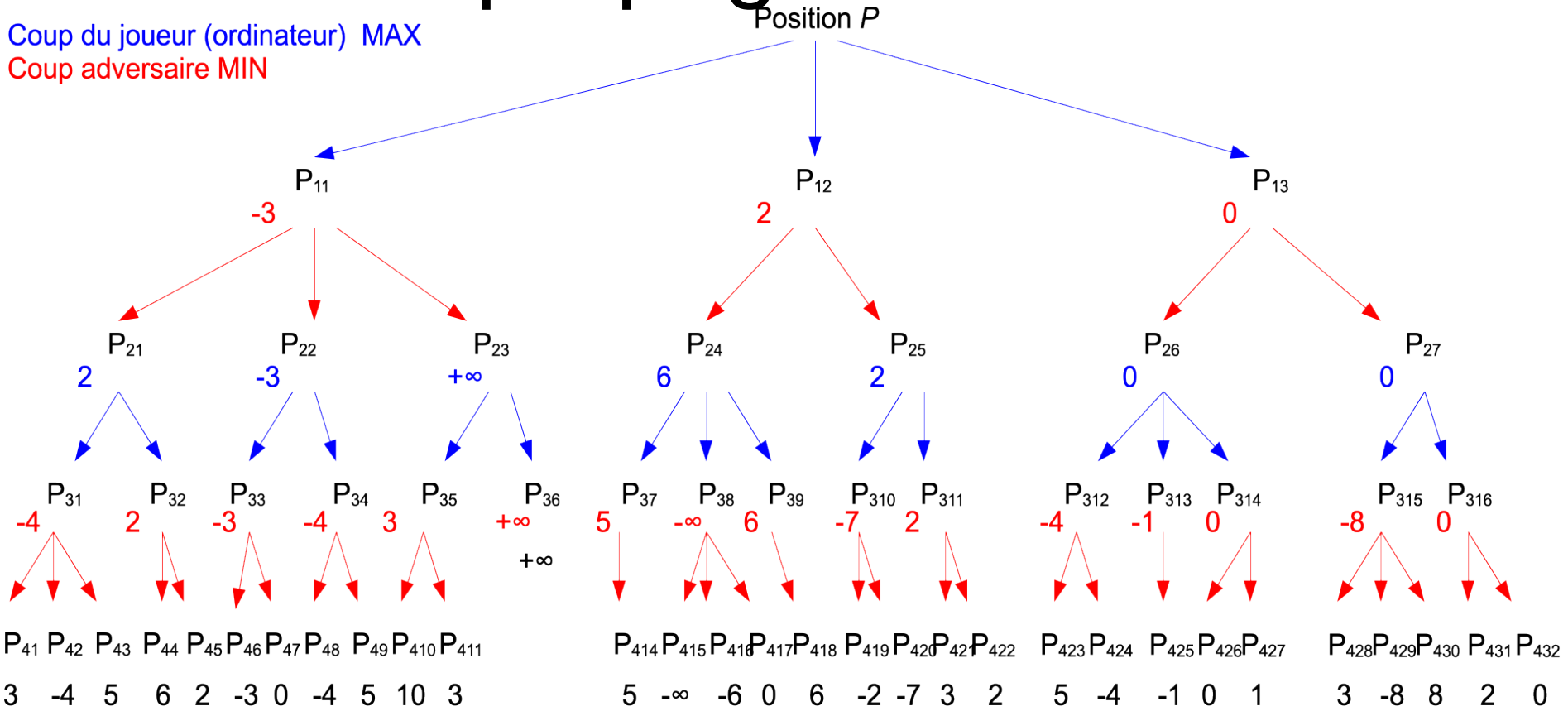
Algorithme MinMax :

- Exploration de l’espace d’états en simulant l’alternance de joueurs
- Chaque joueur cherchera à maximiser son résultat
- L’IA choisit le premier coup pour son intérêt puis simule un choix son adversaire pour le sien, puis choisit le troisième coup pour l’IA, etc.

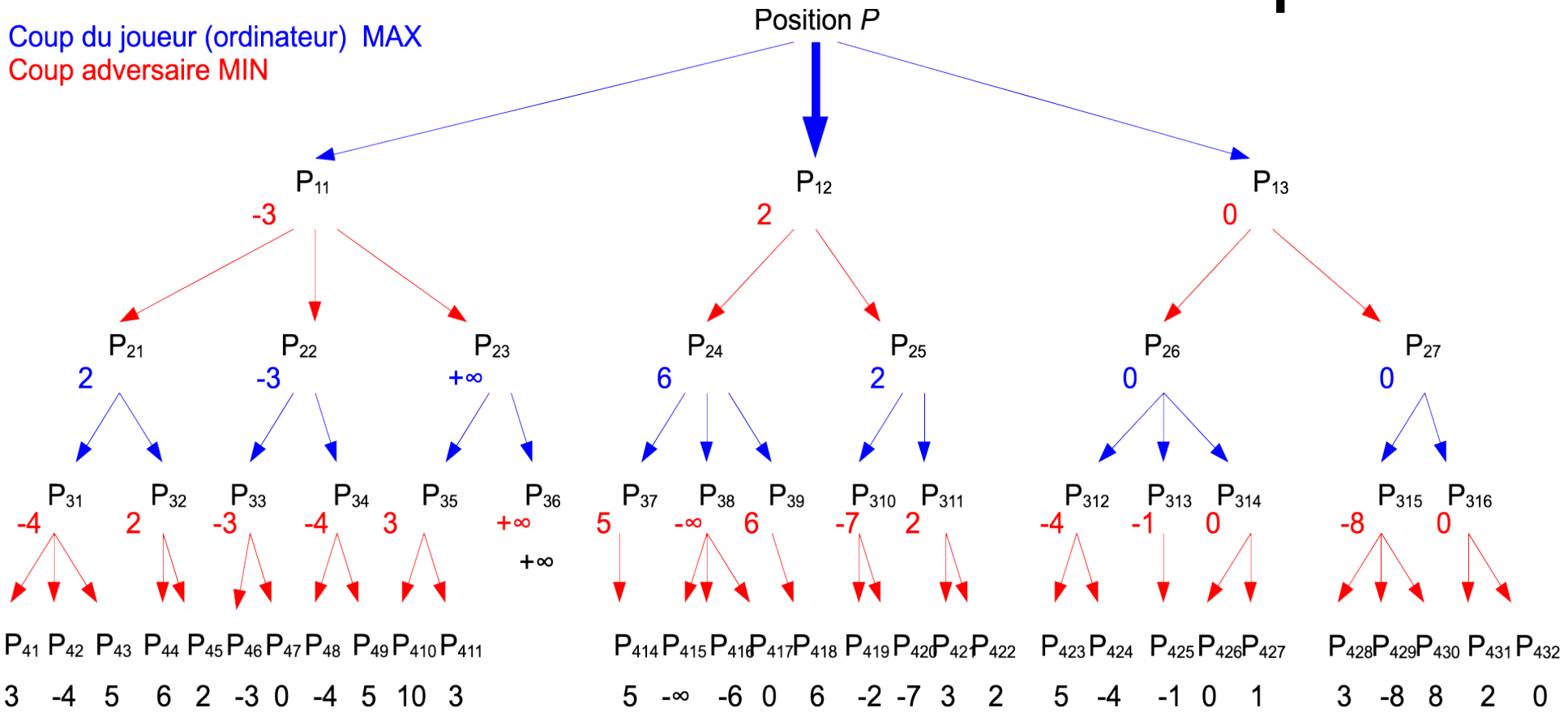
MinMax : génération et évaluation



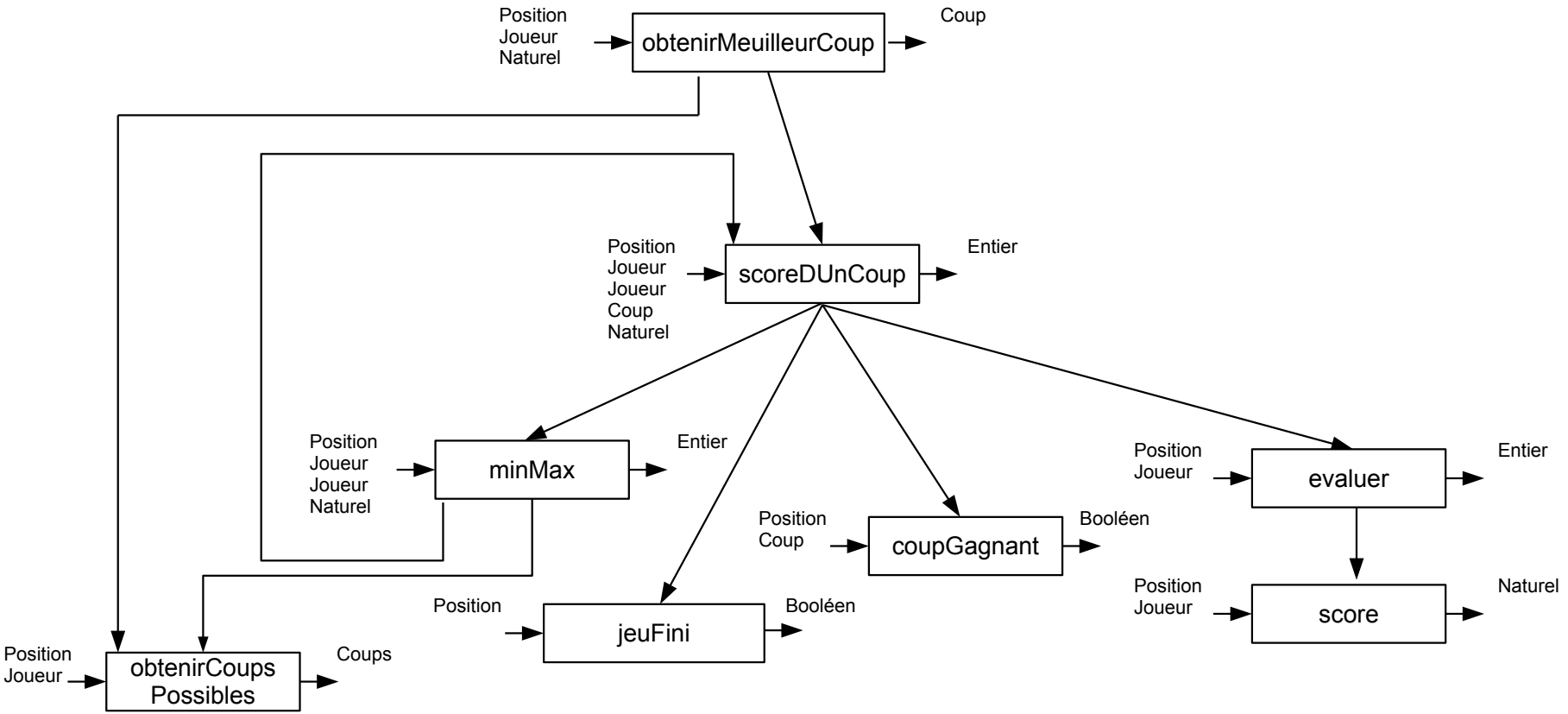
MinMax : propagation des scores



MinMax : choix d'un coup



Analyse descendante



Fonctions de coût

La fonction de coût évalue l'intérêt d'un état-feuille

- Victoire = +∞, défaite = -∞
- Calcul en fonction de connaissances expertes sur le jeu
  - Par exemple aux échecs : nb de pièces protégées, nb de prises possibles, nb de pièces restants, position des pièces, ...
- Plus récemment, estimation par apprentissage automatique