Bases de l'Intelligence Artificielle

CM2 : Résolution de problèmes

Marie Lefevre

2025-2026

Université Claude Bernard Lyon 1

De quoi va-t-on parler ?

- Qu'est-ce qu'un problème en IA ?
- Modélisation d'un problème
- Résolution d'un problème
  - Recherche dans l'espace des solutions
  - Par décomposition de problèmes
- Résolution par satisfaction de contraintes
- Zoom sur l'IA pour les jeux
- Pour aller plus loin

Exemples de problèmes

- Résoudre un casse-tête
  - Jeux
- Trouver un chemin dans un réseau
  - Routier, télécommunications, …
- Trouver un chemin dans une configuration avec obstacles
  - Jeux vidéos, robotique, …
- Trouver la succession des opérations à faire pour passer d'un état à un autre
  - Jeux, production mécanique, …
- Trouver une procédure qui fonctionne
  - Administration, entreprises, …
- Trouver une planification
  - Emploi du temps, gestion de l'espace, …

Démarche IA pour résoudre

- Principe général de l'IA :

  Considérer qu'il existe des méthodes générales permettant de résoudre n'importe quel type de problème

- L'algorithme doit donc être « neutre » sur le domaine concerné
- Les connaissances de description du problème et de sa résolution doivent être clairement séparés de l'algorithme

➢ Malgré l'ambition unificatrice visée, il existe plusieurs démarches différentes pour aborder la question

De quoi va-t-on parler ?

- Qu'est-ce qu'un problème en IA ?
- Modélisation d'un problème
- Résolution d'un problème
  - Recherche dans l'espace des solutions
  - Par décomposition de problèmes
- Résolution par satisfaction de contraintes
- Pour aller plus loin

Modéliser quoi ?

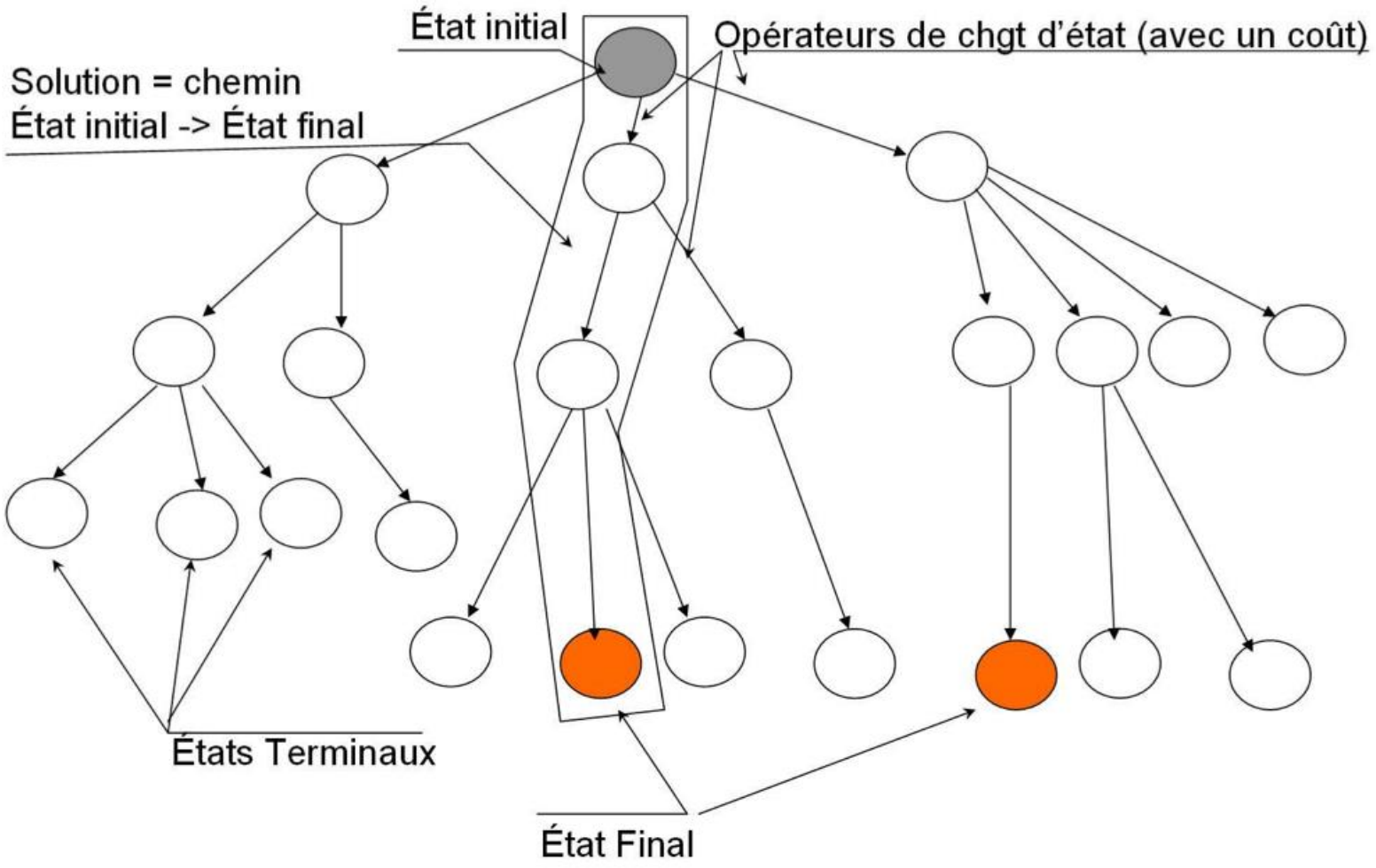
- La résolution de nombreux problèmes peut être décrite comme
  - Une séquence d'opérateurs permettant de passer
  - De l'état initial : la description de l'état du « monde » avant la résolution du problème
  - Jusqu'à l'état final : ce qui caractérisera l'état du monde quand le problème sera résolu
  - Il peut y avoir plusieurs états finaux satisfaisants et permettant de considérer le problème résolu

- Pour faire cette résolution, l'idée est de :
  - Considérer l'état initial
  - Considérer les opérateurs permettant de changer l'état du monde
  - Tenter ces opérateurs de manière systématique (ou pas…)
  - Tester si on arrive à un état final satisfaisant

Modélisation d'un problème

- Pour modéliser un problème de cette manière, il faut donc :
  - Décrire ce qu'est un état du problème
  - Décrire l'état initial
  - Définir les opérateurs permettant de passer d'un état à un autre
  - Construire l'espace des états :
    l'ensemble des états atteignables depuis l'état initial
  - Disposer d'un test permettant de savoir si on a trouvé
    un état but final
  - Construire un chemin de l'état initial à l'état final :
    une séquence d'états dans l'espace des états
  - Disposer d'une fonction de coût sur le chemin :
    cette fonction associe un coût au chemin (coût calculé comme la somme des coûts individuels des actions le long du chemin)

L'espace des états



Solution d'un problème

- Un ou plusieurs chemin(s) d'opérateurs
  conduisant de l'état initial
  aux états qui satisfont les conditions du test d'atteinte du but

- Espace des solutions
  - Ensemble des chemins possibles permettant de passer de l'état initial aux différents états finaux

- Solution optimale
  - Si la somme des coûts du chemin est minimale
  - Par forcément le plus court chemin en terme de nœuds …

Types de problèmes / solutions

- Problèmes de satisfaction de contraintes
  - On connaît l'état de départ
  - Peu importe le chemin
  - On cherche un état final respectant les contraintes
    - Exemples
      - Plan de table pour un mariage
      - Créer un emploi du temps
      - Gestion de pistes des aéroports

- Problèmes de planification
  - On connait l'état de départ
  - On connaît l'état final
  - On cherche un chemin selon certains objectifs
    - Exemples
      - Jeu du taquin, rubik's cube
      - Planification de voyage : trouver le plus court chemin, le moins couteux…

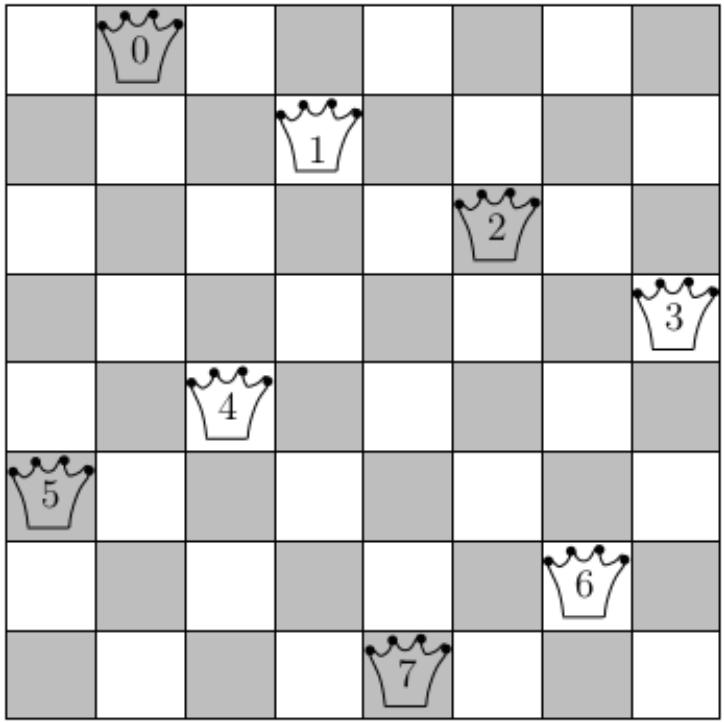
Problème des N reines

Problème de satisfaction de contraintes

Placer N reines sur un échiquier (une grille N x N)

tel qu'aucune reine attaque une autre reine

c'est-à-dire qu'il n'y a pas deux reines sur la même colonne, la même ligne, ou sur la même diagonale

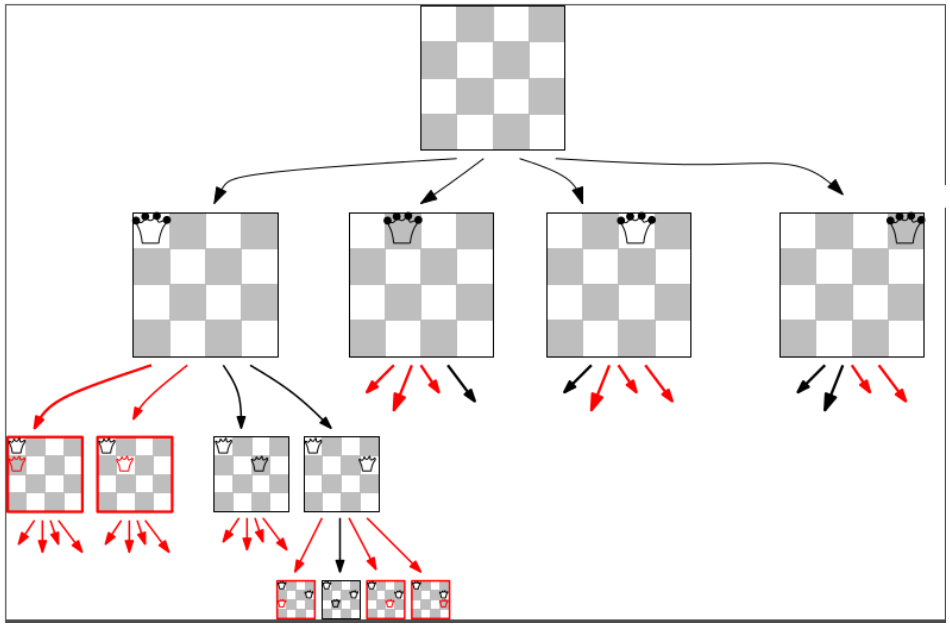


Modélisation n°1 des N reines

- Etats
  - Toute configuration de 0 à N reines sur la grille
- Etat initial
  - La grille vide
- Opérateurs
  - Ajouter une reine sur n'importe quelle case vide de la grille
  - Etat successeur : la configuration qui résulte de l'ajout d'une reine à une case spécifiée sur la configuration courante
- Test de but
  - Une configuration de N reines avec aucune reine sous attaque
- Fonction de cout
  - Ce pourrait être 0, ou un coût constant pour chaque opérateur
  - Nous ne nous intéressons pas au chemin, seulement à l'état but obtenu

Modélisation n°1 des N reines

- Espace des états pour 4 reines :



- Pour 8 reines : 3 * 10^{14} états …

Modélisation n°2 des N reines

- Comment réduire l'espace d'états ?
  - En observant qu'il est inutile de continuer de développer des configurations où il y a déjà un conflit
  - Car on ne pourra pas les résoudre en ajoutant des nouvelles reines

- Changements dans notre modélisation en incluant plus de connaissances du domaine :
  - Etat initial
    - Une grille avec x reines (0 <= x <= N)
    - Avec 1 reine par colonne dans les x colonnes les plus à gauche sans conflit
  - Opérateurs
    - Ajouter une reine dans une case vide
    - Dans la colonne vide la plus à gauche de façon à éviter un conflit

- Pour N = 8, espace des états passe de 3 * 10^{14} états à 2057 !
- Quelle est l'autre différence majeure entre les deux modélisations ?

Modélisation n°3 des N reines

- Modélisation 1 et 2 incrémentales
  - Car nous plaçons les reines une par une sur la grille
- Autre possibilité
  - Commencer avec les huit reines sur la grille (une par colonne)
  - Puis changer la position d'une reine à chaque tour
- Changements dans notre modélisation :
  - Etat initial : une grille avec une seule reine par colonne
  - Opérateurs : changer la position d'une reine dans sa colonne

➢ Dans les modélisations 1 et 2 : espace des états = arbre
  - Nous ne tombons jamais sur un état déjà visité

➢ Dans la modélisation 3 : espace des états = graphe
  - Nous pouvons remettre une reine dans sa position d'origine
  - Il faut donc faire attention
    - A ne pas explorer une deuxième fois un chemin déjà exploré
    - A ne pas tourner en boucle

Problème d'itinéraire

Problème de planification

Trouver les vols permettant de faire un circuit touristique passant par les 10 plus grandes villes des Etats Unis *, avec pour contrainte de partir de New York et d'y revenir

* 1. New-York
2. Washington, D.C.
3. Orlando
4. Las Vegas
5. Los Angeles
6. Boston
7. Miami
8. San Francisco
9. San Diego
10. Seattle

Modélisation d'itinéraire

- Etats
  - Un état est composé d'un aéroport, de la date et de l'heure actuelle
- Etat initial
  - L'aéroport d'où part le client (dans ce cas précis New York)
  - La date et l'heure de départ souhaitées
- Opérateurs
  - Les vols d'un aéroport à un autre
  - Qui partent de l'aéroport de l'état courant plus tard que l'heure actuelle en laissant un certain délai pour les visites
- Test de but
  - Les états où le client est à l'aéroport de sa destination (New York)
  - Et où le chemin comprend les 9 autres villes
- Fonction de coût
  - Dépend des préférences du client
    - 1 pour chaque opération pour minimiser le nombre de vols
    - La durée des vols pour minimiser la durée du voyage
    - Le prix des trajets pour trouver le voyage le moins cher
    - …

Problème de jeu

Cas particulier des jeux à plusieurs joueurs

Les jeux à plusieurs joueurs sont plus compliqués car le joueur (humain ou IA) ne peut pas choisir les actions des autres joueurs

Donc au lieu de chercher un chemin qui relie l'état initial à un état but (qui contiendrait des actions des adversaires que nous ne maîtrisons pas), nous cherchons une stratégie

C'est-à-dire un choix d'action pour chaque état où pourrait se trouver le joueur

Exemple classique : les échecs, les jeux vidéos…

Modélisation des échecs

- Etats
  - Une configuration de l'échiquier et un nom du joueur
- Etat initial
  - La configuration standard pour le début d'une partie d'échecs
- Opérateurs
  - Les coups qui peuvent être joués par le joueur courant dans la configuration actuelle du jeu (y compris abandonner le jeu)
  - La configuration du jeu évolue selon le coup choisi et le joueur change
- Test de but
  - Beaucoup de possibilités pour terminer une partie
  - La plus connue : échec et mat
- Fonction de cout
  - 1 pour chercher les coups qui amènent le plus vite à une victoire

De quoi va-t-on parler ?

- Qu'est-ce qu'un problème en IA ?
- Modélisation d'un problème
- Résolution d'un problème
  - Recherche dans l'espace des solutions
  - Par décomposition de problèmes
- Résolution par satisfaction de contraintes
- Zoom sur l'IA pour les jeux
- Pour aller plus loin

Résolution d'un problème

- Structure générale d'un algorithme de recherche

```
Si il n'y a plus d'états à traiter
Alors renvoyer échec
Sinon, choisir un des états à traiter (*)
      Si l'état est un état final
      Alors renvoyez la solution correspondante
      Sinon supprimer cet état de l'ensemble des états à traiter
            et le remplacer par ses états successeurs
```

- Ce qui va différencier les différents algorithmes = le choix à l'étape (*)

- Dans la vie courante, une recherche systématique est la conséquence d'un manque de connaissances
- C'est vrai également en IA !!!
  - L'algorithme n'est qu'un outil pour « attaquer » les problèmes

Méthodes de résolution

- Deux grandes familles selon la manière de construire la solution à partir d'un état de départ :

- Augmentation de solution partielle [Generate and Test]
  - On commence à résoudre le problème en utilisant un opérateur valide pour l'état initial
  - On continue à le faire à partir de l'état résultant
  - S'il n'est plus possible de faire une nouvelle opération ou si on revient sur un état déjà parcouru
    alors on revient en arrière (backtrack) pour essayer un nouvel opérateur
  - Si plus aucun choix d'opérateur n'est possible
    alors il n'y a pas de solution

- Application récursive [Divide and Solve]
  - On décompose le problème en sous-problèmes jusqu'à arriver à des sous-problèmes « triviaux »
  - On résout ces problèmes par la méthode précédente
  - On remonte l'arbre de décomposition jusqu'à obtenir la solution au problème complet

Évaluation des algorithmes

- Complexité en temps
  - Combien de temps prend l'algorithme pour trouver la solution ?
- Complexité en espace
  - Combien de mémoire est utilisée lors de la recherche d'une solution ?
- Complétude
  - Est-ce que l'algorithme trouve toujours une solution s'il y en a une ?
- Optimalité
  - Est-ce que l'algorithme renvoie toujours des solutions optimales ?

De quoi va-t-on parler ?

- Qu'est-ce qu'un problème en IA ?
- Modélisation d'un problème
- Résolution d'un problème
  - Recherche dans l'espace des solutions
  - Par décomposition de problèmes
- Résolution par satisfaction de contraintes
- Zoom sur l'IA pour les jeux
- Pour aller plus loin

Algo de recherche générique

```
Entrée : un état initial, une fonction de successeurs, un test de but,
et une fonction de coût

Sortie : un chemin contenant tous les états entre Ei et Ef
et les opérateurs permettant de passer de Ei à Ef

fonction RECHERCHE(Ei, SUCCESSEURS, TEST_BUT, COUT)

   nœuds_a_traiter <- CREER_LISTE(CRÉER_NŒUD(Ei, [], 0))

   boucle
        si VIDE?(nœuds_a_traiter)
          alors renvoyer « echec »

        nœud <- ENLEVER_PREMIER_NOEUD(nœuds_a_traiter)

        si TEST_BUT(ETAT(noeud)) = vrai
          alors  renvoyer  CHEMIN(noeud), ETAT(noeud)

        pour tout (action, etat) dans SUCCESSEURS(ETAT(noeud))

          chemin <- [operateur, CHEMIN(noeud)]
          cout_chemin <- COUT_DU_CHEMIN(noeud) + COUT(etat)
          s <- CREER_NOEUD(etat, chemin, cout_chemin)
          INSERER(s, nœuds_a_traiter)
```

Recherche non-informée

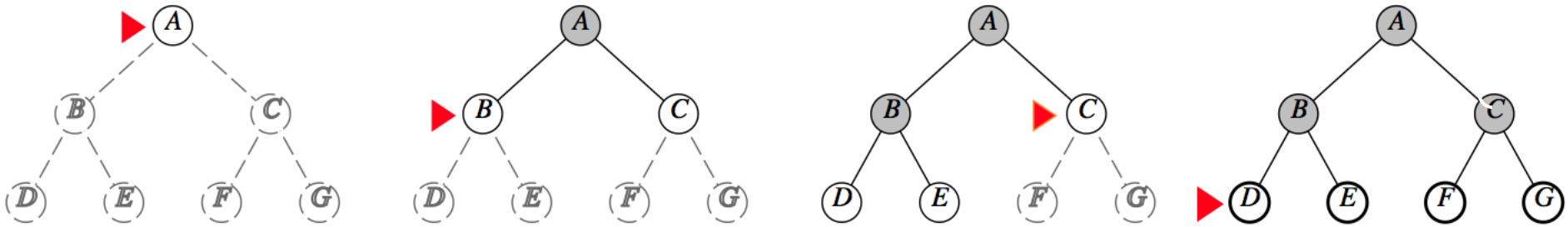
- Ces algorithmes ne disposent pas d'informations supplémentaires pour pouvoir distinguer des états prometteurs
  - Recherche exhaustive de tous les chemins possibles

- Deux possibilités
  - Parcours en largeur
  - Parcours en profondeur
    - Classique, limitée, itérée

Parcours en largeur

- Principe : tous les nœuds d'une certaine profondeur sont examinés avant les nœuds de profondeur supérieure
  - D'abord l'état initial
  - Puis tous ses successeurs directs
  - Puis les successeurs des successeurs
  - Etc.

- Pour l'implémenter : utiliser une file d'attente
  - Placer les nouveaux nœuds systématiquement à la fin de la liste de nœuds à traiter



Parcours en largeur

- Complexité en temps
  - $O(s^{p+1})$ où s est le nombre de successeurs de chaque état et p le nombre minimal d'actions pour relier l'état initial à un état but
- Complexité en espace
  - $O(s^{p+1})$
- Complétude
  - Algorithme complet à condition que le nombre de successeurs des états soit toujours fini (très souvent le cas dans les problèmes courants)
- Optimalité
  - Trouve toujours un état final de plus petite profondeur possible
  - Donc si on cherche une solution quelconque, ou une solution avec le moins d'actions possibles, le parcours en largeur est optimal
  - ➢ Le parcours en largeur est optimal quand le coût du chemin ne dépend que du nombre d'actions de ce chemin mais dans le cas général il ne l'est pas

Parcours en largeur

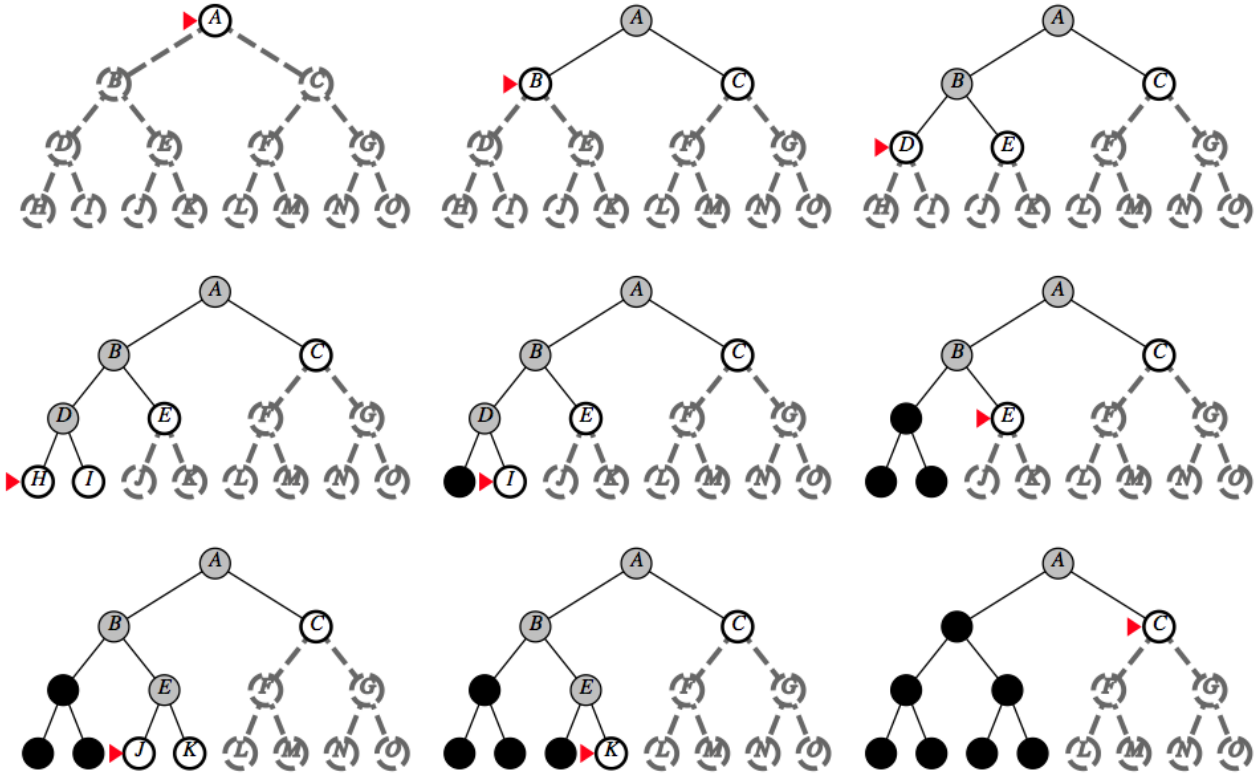
- Limite : sa complexité en temps et espace

- Exemple :
  - Pour un problème ayant 10 successeurs par état
  - Et une solution de profondeur de 12
  - Il faudra à l'algorithme 10 peta-octets de mémoire
    - Donc infaisable sur vos ordinateurs personnels
    - Et 30 aines de minutes pour trouver la solution avec un serveur de calcul…

➢ Cette haute complexité en temps et en espace restreint l'utilisation du parcours en largeur aux petits problèmes

Parcours en profondeur

- Principe : suit le chemin courant le plus longtemps possible
- Pour l'implémenter : utiliser une pile
  - Placer successeurs du nœud courant au début de la liste de nœuds à traiter



Parcours en profondeur

- Complexité en temps
  - $O(s^m)$ où s est le nombre de successeurs de chaque état et m la profondeur maximale d'actions
- Complexité en espace
  - O(s*m)
- Complétude
  - Pas complet parce que l'algo peut continuer sur un chemin infini, ignorant un état but qui se trouve sur un autre chemin
- Optimalité
  - Pas optimal : aucune garantie que le premier état but trouvé sera le bon

Parcours en profondeur

- Limite : pas complet, pas optimal, pas plus efficace en temps que le parcours en largeur
- Mais même quand s et m sont grands, la mémoire nécessaire pour le parcours en profondeur reste raisonnable

➢ Cette faible complexité en espace permet de traiter des problèmes qui ne sont pas abordables par le parcours en largeur

Parcours en profondeur amélioré

- Parcours en profondeur limitée
  - Restreint la profondeur du parcours
  - Ne trouve que certaines solutions mais se termine si le nombre de branches est fini
  - ➢ Difficulté de bien choisir la borne de profondeur

- Parcours en profondeur itérée
  - Comme on ne sait pas quelle borne de profondeur choisir, on les teste les unes après les autres
  - Donc un parcours en profondeur limitée avec une borne de 1, puis avec une borne de 2, et nous continuons à augmenter la borne jusqu'à ce que l'on trouve une solution

Etats redondants

Deux types de redondances possibles :

- Des états qui peuvent être visités deux fois sur le même chemin
  - Ex des N reines : on déplace une reine puis on la remet sur la même case
- Pour l'éviter : ne jamais ajouter à la liste de nœuds à traiter des nœuds dont les chemins contiennent plusieurs fois le même état
  - Cette modification ne change pas significativement la complexité en espace des différents algorithmes

- Plusieurs chemins différents peuvent amener au même état
- Pour l'éviter : il faut se rappeler de tous les états déjà visités et de ne garder qu'un seul chemin par état
  - Il faut donc gérer une liste des états déjà visités et y ajouter les états quand ils sont visités pour la première fois
  - Avant de générer un successeur, il faut vérifier qu'il n'est pas dans la liste
  - Cette modification ne changera pas trop la complexité en espace pour le parcours en largeur, mais très significativement pour le parcours en profondeur

Recherche informée

- Algorithme de recherche non informée
  - Recherche exhaustive de tous les chemins possibles
  - ➢ L'explosion combinatoire est vite là !
  - ➢ Inefficace voire inutilisable sur les problèmes de grande taille

- Solution : algorithme de recherche heuristique
  - Utilise des connaissances supplémentaires pour pouvoir guider la recherche

- Une heuristique
  - Un moyen d'ordonner dynamiquement la liste des successeurs selon leur « promesse de se rapprocher d'un but »
  - L'expression d'une connaissance spécifique au problème à résoudre et donc indépendante de l'algorithme de recherche

Fonction d'évaluation

- Tout algorithme de recherche heuristique dispose d'une fonction d'évaluation f
- Elle détermine l'ordre dans lequel les nœuds sont traités
  - La liste de nœuds à traiter est ordonnée avec les nœuds de plus petite valeur en tête de liste
- La fonction d'évaluation a souvent comme composante une fonction heuristique h
  - h(n) = coût estimé du chemin reliant l'état n à un état but
  - h(n) = 0 si n est un état but

Exemple

- Problème : Nous sommes à Paris, et nous voulons rejoindre Berlin en parcourant la plus petite distance possible

- Modélisons le problème
  - Etats : toutes les villes d'une carte
  - Etat initial : l'état Paris
  - Opérateur : une route entre deux villes
  - Test de but : être sur l'état Berlin

- Fonction de cout : pour mesurer à quel point nous sommes proches du but lorsque nous sommes dans la ville X, on voudrait utiliser h(X) = d(X, Berlin)
  - où d(A,B) la distance routière entre la ville A et la ville B
    i.e. le plus court chemin entre A et B
- Problème : on n'a pas accès à la fonction d, et son calcul n'est pas trivial

- C'est ici qu'intervient l'heuristique, c'est-à-dire l'approximation
  - On approxime d(X, Berlin) par h(X) = distance à vol d'oiseau entre la ville X et Berlin
    - Calculable à l'aide d'une carte routière et d'un décimètre
  - Comme les distances par la route sont toujours supérieures à la distance à vol d'oiseau, h(X) n'est qu'une approximation du coût réel
  - Si notre heuristique était parfaite, nous n'aurions pas besoin de faire une recherche ☺

Best-first Search

- Principe : utiliser la fonction heuristique comme fonction d'évaluation i.e. f(n) = h(n)

- Complexité en temps
  - $O(s^m)$ où s est le nombre de successeurs et m la profondeur maximale
- Complexité en espace
  - $O(s^m)$
- Complétude
  - Pas complet : l'algorithme peut tourner en boucle même s'il existe une solution
- Optimalité
  - Pas optimal en général

- La complexité et l'optimalité dépendent en pratique de la qualité de la fonction heuristique
- Donne la préférence aux nœuds dont les états semblent les plus proches d'un état but, mais ne prend pas en compte les coûts des chemins reliant l'état initial à ces nœuds

A star

- Principe : le coût d'un chemin passant par un nœud n est la somme du coût du chemin entre l'état initial et n
et le coût du chemin reliant n à un état but

- **f(n) = g(n) + h(n)**
  - g(n) : le coût du chemin entre l'état initial et n
  - h(n) : l'estimation de coût entre n et un état but
  - ➢ La fonction d'évaluation f donne une estimation du coût de la meilleure solution passant par le nœud n
  … et plus seulement partant de n !

- On note **f***, **g*** et **h*** les coûts optimaux

- Attention :
  - **h(n)** garde toujours la même valeur alors que **g(n)** varie en cours de recherche
  - Mais on a toujours **g(n)** ≥ **g*(n)**

A star

- L'algorithme de recherche A* est complet et optimal
  - Si il y a un nombre fini de successeurs
  - Et si la fonction h est admissible

- Qu'est-ce qu'une heuristique admissible ?
  - La valeur h(n) ne doit jamais être supérieure au coût réel du meilleur chemin entre n et un état but

- Dans notre exemple, l'heuristique prenant les distances à vol d'oiseau est un exemple d'une fonction heuristique admissible
  - La distance à vol d'oiseau (la valeur heuristique) n'est jamais supérieure à la distance par la route (le vrai coût)

A star

- Pour garantir l'optimalité et éliminer les nœuds redondants
  - Il ne faut garder que le premier chemin amenant à un état
  - Donc que la fonction heuristique soit consistante

- Qu'est-ce qu'une heuristique consistante ?
  - Pour tout nœud n et tout successeur n' obtenu à partir de l'opération o, il faut respecter l'inégalité : $h(n) \leq cout(o) + h(n')$

- Attention
  - Les fonctions heuristiques consistantes sont toujours admissibles
  - Mais les fonctions heuristiques admissibles sont très souvent, mais pas toujours, consistantes

A star

- La complexité dépend de la fonction heuristique
- En général, la complexité en temps et en espace est grande
  - ➢ Mal adapté pour les problèmes de grande taille
- Plusieurs autres algorithmes heuristiques moins gourmands en mémoire ont été proposés
  - Iterated A* Search (IDA*)
  - Recursive Best-First Search (RBFS)
  - Memory-Bounded A* (MA*)
  - Simplified Memory-Bounded A* (SMA*)

➢ Algorithme A* en TD…

Recherche locale

- Pour les problèmes de satisfaction de contraintes
  - Le chemin n'est pas important
  - Seul trouver l'état but est important

- Pour ces types de problèmes, il existe une autre stratégie
  - Générer les états successivement sans s'intéresser aux chemins jusqu'à trouver un état but
  - ➢ Idée de base pour les algorithmes de recherche locale

- La recherche locale sacrifie la complétude pour gagner du temps et de la mémoire

- L'algorithme local le plus simple = recherche locale gloutonne

Recherche locale gloutonne

- Principe :

```
Commencer avec un état choisi aléatoirement
Si l'état est un état final
Alors la recherche a réussie
Sinon  Générer les successeurs et leurs valeurs heuristiques
       S'il n'existe pas de successeur avec une meilleure
          valeur que la valeur heuristique de l'état courant
       Alors la situation ne peut être améliorée :
          la recherche a échoué
       Sinon  Choisir l'état successeur ayant la meilleure
                 valeur heuristique
              Et continuer
```

Recherche locale gloutonne

- Avantage
  - Très faible consommation mémoire : on ne garde que l'état courant en mémoire
  - Très faible consommation en temps
    - Par exemple, pour le problème des 8 reines (en moyenne)
      - Se termine après 4 étapes quand il trouve une solution
      - S'arrête en 3 étapes quand il est bloqué

- Limite
  - Son faible taux de succès : arrêt de la recherche si jamais la valeur heuristique n'est plus améliorable
    - Pour le problème des 8 reines : 14% de réussite

Recherche locale gloutonne

- Plusieurs améliorations possibles…
- Permettre à l'algorithme de visiter les successeurs ayant la même valeur que l'état courant
  - Il faut mettre une borne sur le nombre de déplacements consécutifs de ce type pour ne pas tourner en boucle
  - Pour le problème des 8 reines, avec max 100 déplacements : 94% de réussite
- Ajouter de l'aléatoire : au lieu de choisir toujours le meilleur voisin, choisir un voisin avec une probabilité relative à sa valeur heuristique
- Lors des blocages : recommencer la recherche locale gloutonne à partir d'un autre état choisi au hasard, et continuer ainsi jusqu'à obtenir un état but
  - Technique simple mais très efficace : résout le problème des 3 millions de reines en moins d'une minute sur un ordinateur de bureau !

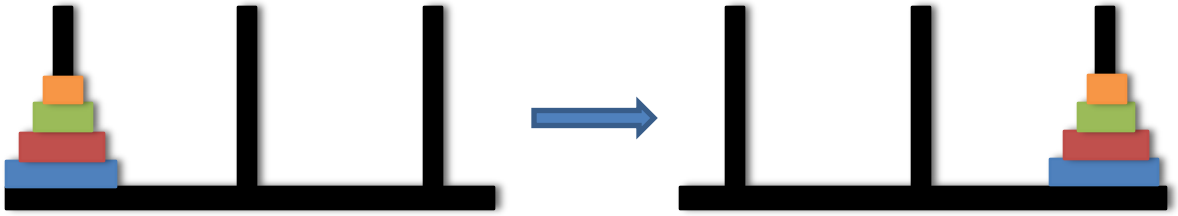
➢ Grâce à leur très faible complexité en espace, les algorithmes locaux peuvent être utilisés pour résoudre des problèmes de taille importante qui ne peuvent pas être résolus avec des algorithmes classiques

De quoi va-t-on parler ?

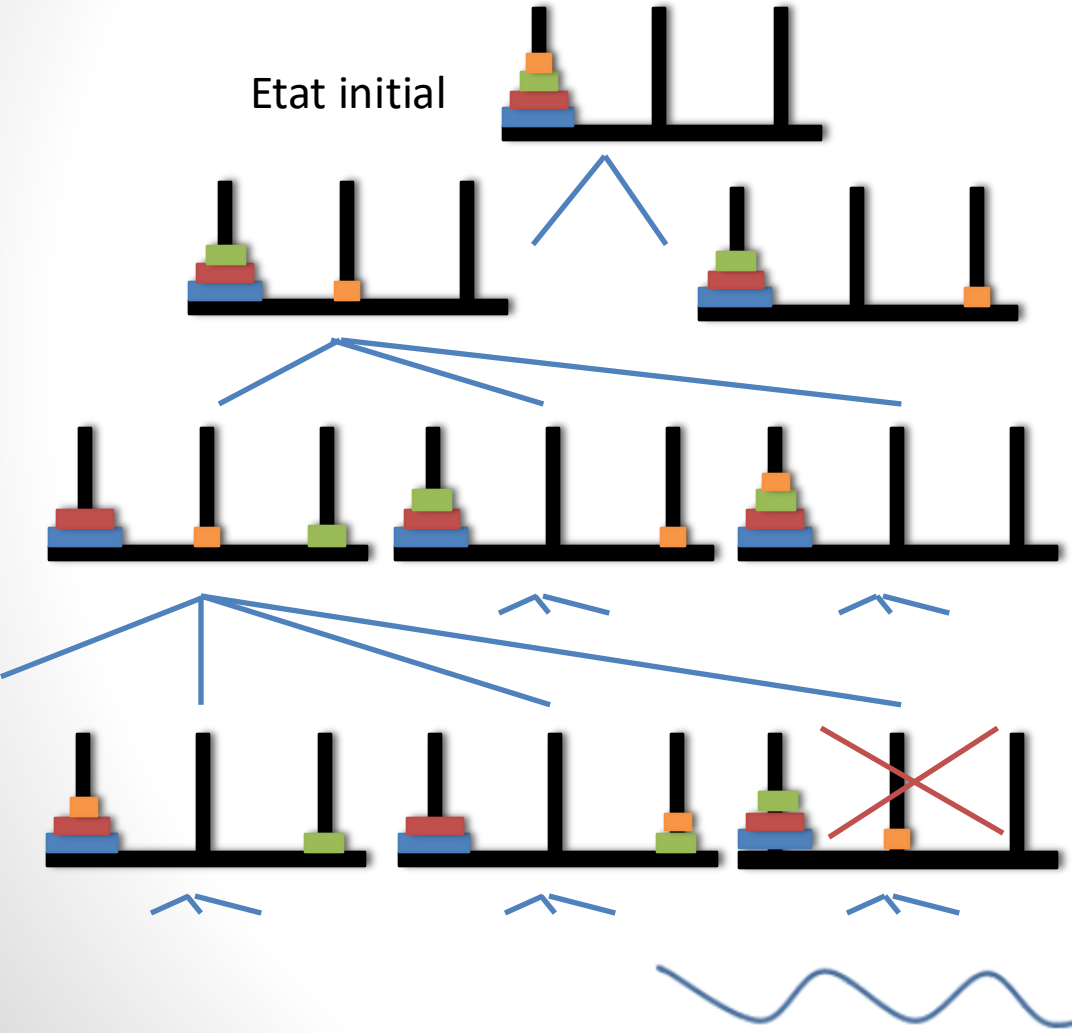
- Qu'est-ce qu'un problème en IA ?
- Modélisation d'un problème
- Résolution d'un problème
  - Recherche dans l'espace des solutions
  - Par décomposition de problèmes
- Résolution par satisfaction de contraintes
- Zoom sur l'IA pour les jeux
- Pour aller plus loin

Exemple : les tours de Hanoï

- Jeu de réflexion
  - Déplacer des disques de diamètres différents d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire
  - L'objectif est de le faire en un minimum de coups, tout en respectant les règles suivantes :
    - On ne peut déplacer plus d'un disque à la fois,
    - On ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.
  - On suppose que cette dernière règle est également respectée dans la configuration de départ.



Exemple : les tours de Hanoï avec l'approche précédente…



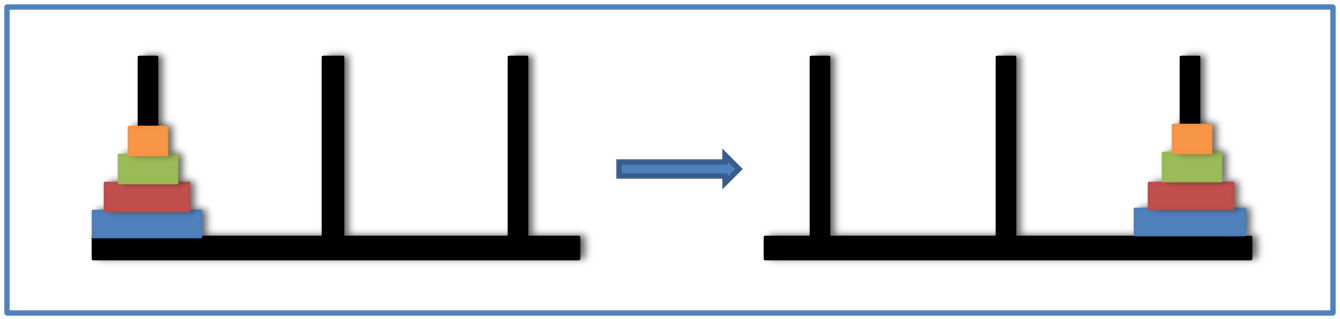
Etat :
- 3 tours avec sur chaque tour les disques
- i.e. 3 listes ordonnées contenant les disques d1, d2, d3, d4

Opérateur :
- Prendre un disque « libre » et le mettre dans sur une autre tour
- i.e. prendre le premier élément d'une liste et le mettre dans une autre liste

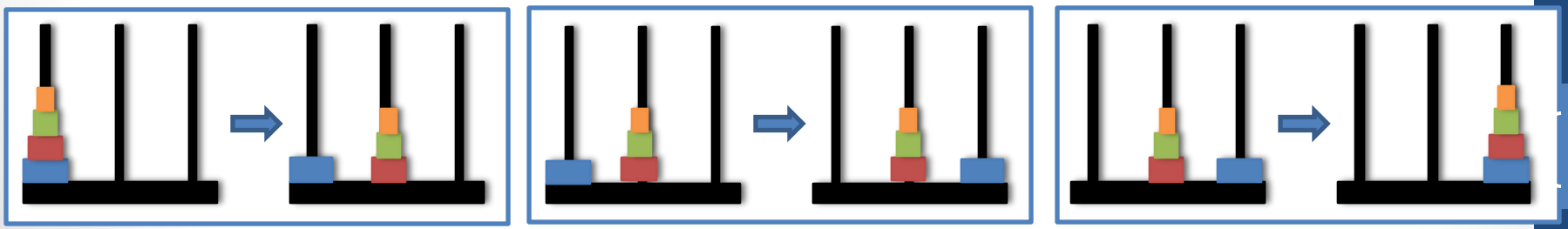
ATTENTION AUX CONTRAINTES !!!

Exemple : les tours de Hanoï en décomposant le problème

- L'état initial = le problème en entier



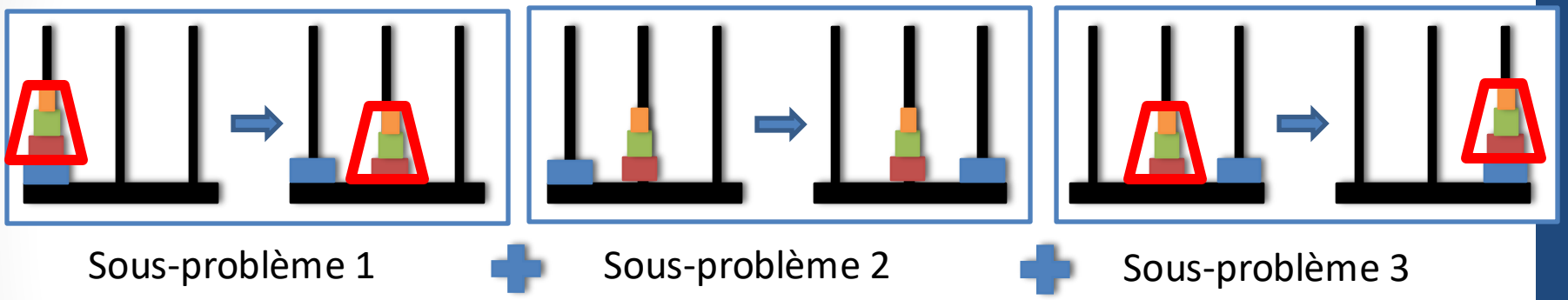
- Les opérateurs = découper le problème en sous-problèmes plus simples



Sous-problème 1 + Sous-problème 2 + Sous-problème 3

Exemple : les tours de Hanoï en décomposant le problème

- Et après ?



- Le sous-problème 2 est trivial, on sait le résoudre
- Les sous-problèmes 1 et 3 ne le sont pas
  - Mais ils sont du même type que le problème de base :
  - ➢ Déplacer n disques d'une tour de départ à une tour d'arrivée
  - … alors on recommence, on décompose pour arriver sur des problèmes triviaux

Graphe de sous-problèmes

- Dans cette stratégie de résolution de problèmes :
  - Un état est un problème (ou un sous-problème à résoudre)
  - L'état initial est le problème non décomposé
  - Les états finaux sont des sous-problèmes triviaux (solution immédiate)
  - L'opérateur est celui de décomposition de problème

- Il y a deux types de nœuds :
  - Les nœuds « OU » associés aux problèmes
  - Les nœuds « ET » associés aux règles de décomposition

Recherche dans un graphe et/ou

- Rechercher une solution = rechercher un sous-graphe contenant :
  - Le nœud initial (problème complet)
  - Pour tout nœud non terminal
    un seul connecteur (une seule règle) et les nœuds auxquels il mène

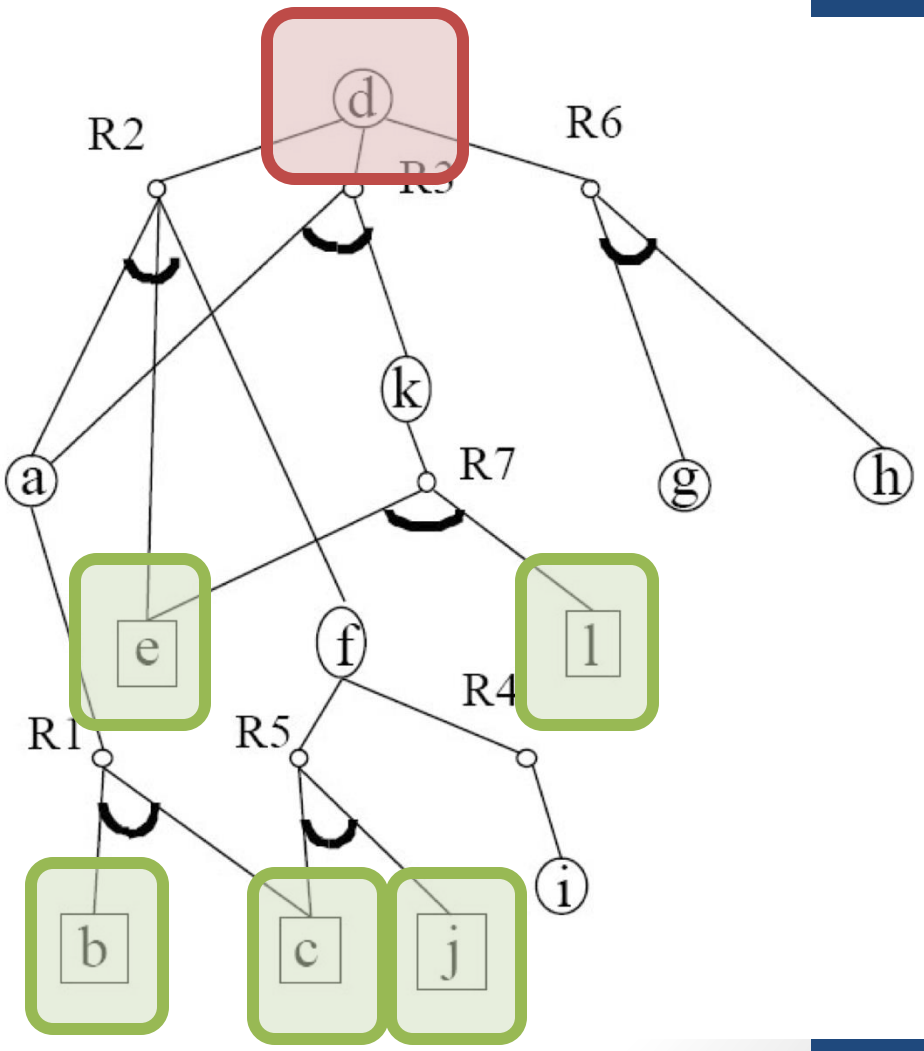
- On définit alors
  - Des états RESOLUS
    - Les états terminaux
    - Les états dont un connecteur a tous ses fils RESOLUS
  - Des états INSOLUBLES
    - Les états non terminaux sans successeur
    - Les états dont tous les connecteurs ont au moins un successeur insoluble

- Algorithme : on développe en profondeur un sous-graphe représentant une solution partielle avec retour arrière en cas d'échec

Un exemple jouet

- Problème à résoudre
  - d
- Problèmes terminaux
  - b, c, e, j, l

- Règles de décompositions
  - R1 : a → b, c
  - R2 : d → a, e, f
  - R3 : d → a, k
  - R4 : f → i
  - R5 : f → c, j
  - R6 : d → g, h
  - R7 : k → e, l



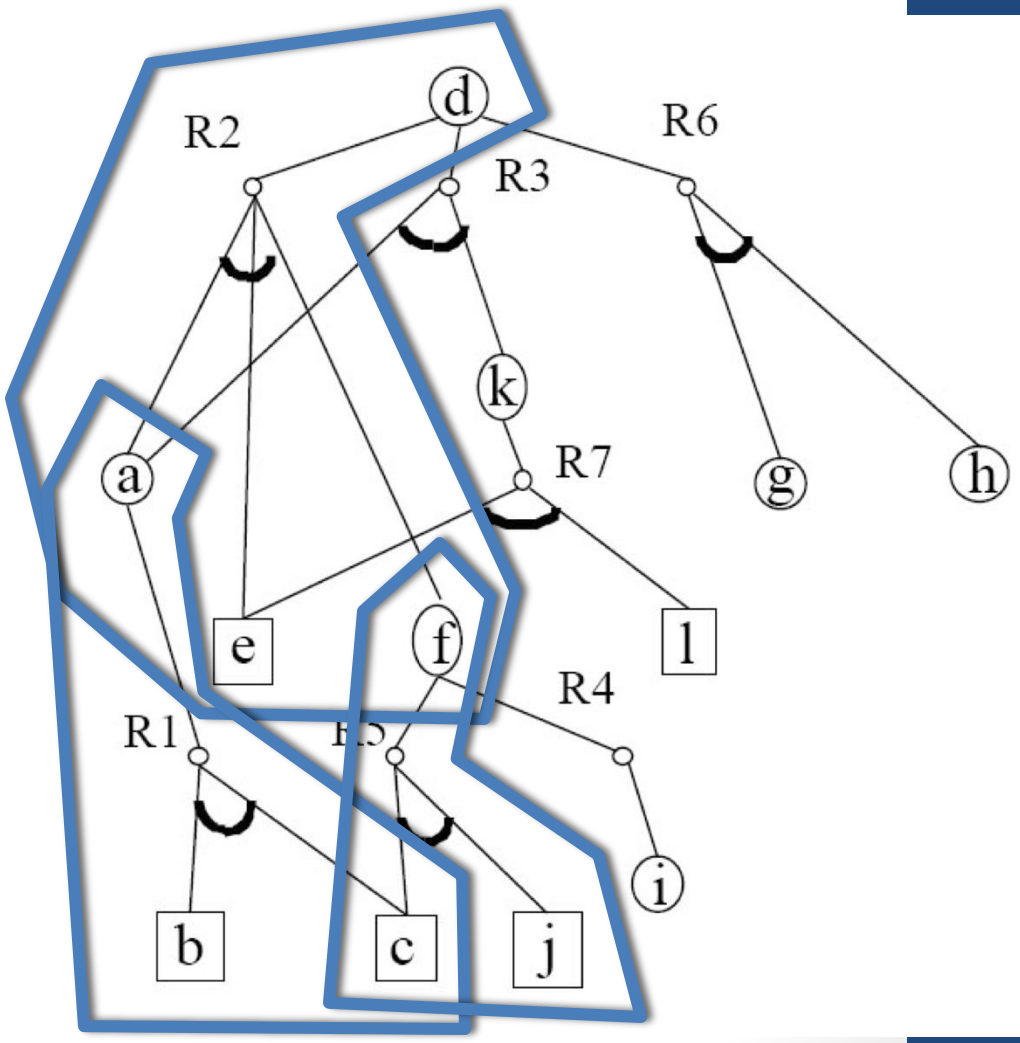
Une recherche possible

- En commençant par R2, de gauche à droite…

- Problème à résoudre : d
- Problèmes terminaux
  - b, c, e, j, l

- Etats résolus
  - e
  - b, c => a
  - j => f
  - => d

- Etats insolubles



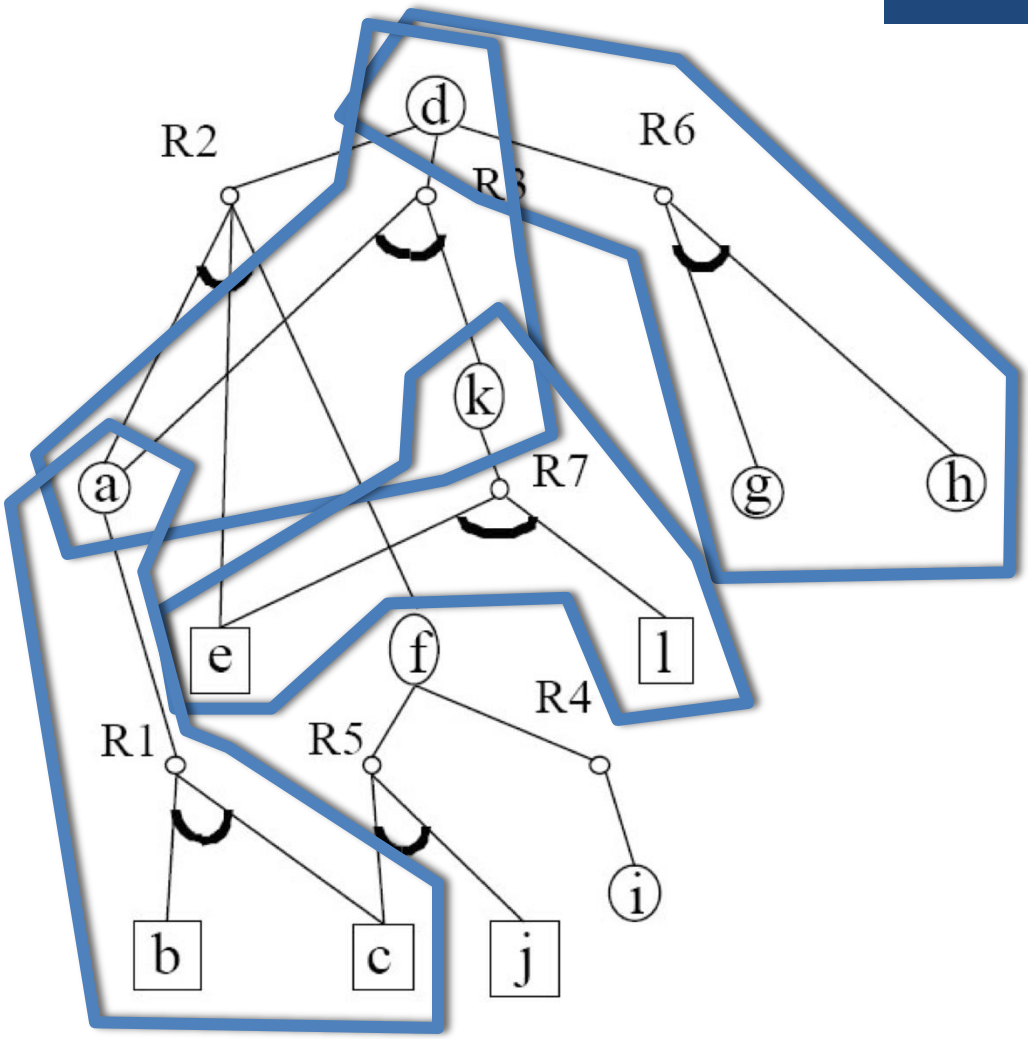
Une autre recherche possible

- En commençant par R6, de droite à gauche…

- Problème à résoudre : d
- Problèmes terminaux
  - b, c, e, j, l

- Etats résolus
  - l, e => k
  - c, b => a
  - => d

- Etats insolubles
  - h, g



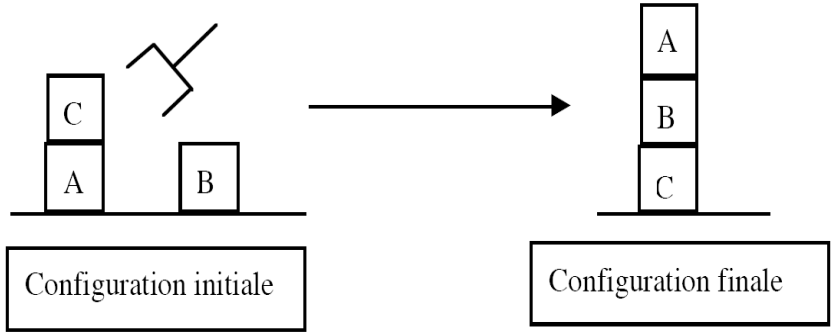
Recherche dans un graphe et/ou

- Pour améliorer l'efficacité de la recherche
  - Introduire un seuil sur le rang des nœuds pour réduire l'espace exploré
  - Exploiter une heuristique pour ordonner l'examen des connecteurs d'un nœud
  - Exploiter une heuristique pour interrompre la récursion sur des états peu prometteurs en les rendant insolubles arbitrairement
  - …

➢ Algorithme BSH (Backtrack Search dans un Hypergraphe) en TD…

Attention … pas toujours simple !

- Nouvel exemple : le monde des blocs
  - Problème de planification
  - Objectif : trouver la suite d'étapes permettant à un bras articulé muni d'une pince de manipuler des cubes se trouvant sur une table afin de les empiler dans un ordre donné



- Les contraintes :
  - Le bras articulé peut saisir un cube, le déplacer, le poser sur une table ou l'empiler sur un autre cube
  - La pince ne peut prendre qu'un cube à la fois
  - Seul un cube sur lequel rien n'est empilé peut être saisi.

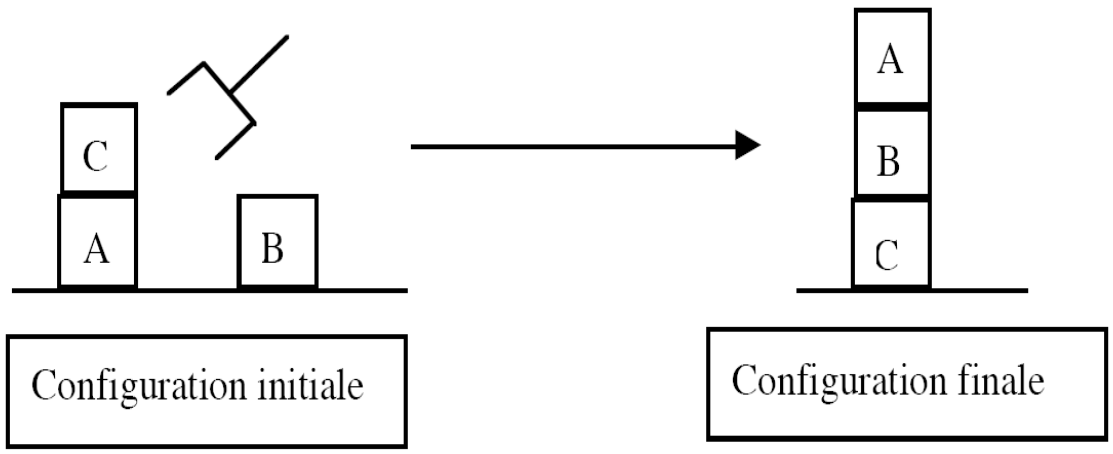
Modélisation du monde des blocs

- Par exemple en utilisant des variables d'état (des prédicats)

- Etats
  - PinceVide : vrai ou faux
  - Tenir(X) : se lit « la pince tient le cube X » avec X dans {A,B,C}
  - Sur(X1,X2) : se lit « le cube X1 est sur le cube X2 » avec X1 dans {A,B,C}, X2 dans {A,B,C} et X1 ≠ X2
  - Libre(X) : : se lit « le cube X est en sommet de pile » avec X dans {A,B,C}
  - SurTable(X) : vrai ou faux

- Etat initial
  - SurTable(A)
  - Sur(C,A)
  - Libre(C)
  - SurTable(B)
  - Libre(B)
  - PinceVide

- Test de but = état final
  - SurTable(C)
  - Sur(B,C)
  - Sur(A,B)
  - Libre(A)



Modélisation du monde des blocs

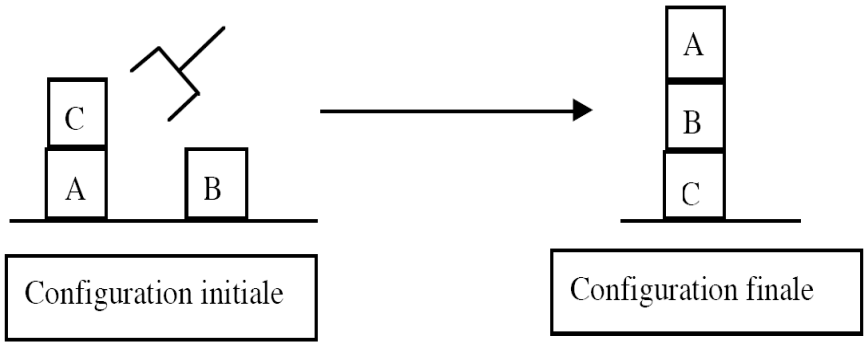
- Opérateurs
  - Empiler(Bi,Bj) : sert à mettre Bi sur Bj
    - Préconditions : Libre(Bj) et Tenir(Bi)
    - Fonction successeur :
      - Libre (Bi) devient Vrai
      - Sur(Bi,Bj) devient Vrai
      - Tenir(Bi) devient Faux et PinceVide devient Vrai
  - dépiler(Bi,Bj) : sert à enlever Bi qui est sur Bj
    - Préconditions : Libre(Bi) et Sur(Bi,Bj) et PinceVide est vrai
    - Fonction successeur :
      - Libre(Bj) devient Vrai et Libre(Bi) devient Faux
      - Tenir(Bi) devient vrai et PinceVide devient Faux
  - Prendre(Bi) : sert à prendre Bi qui se trouve sur la table
    - Préconditions : SurTable(Bi) et Libre(Bi) et PinceVide
    - Fonction successeur :
      - SurTable(Bi) devient Faux, Sommet(Bi) devient Faux, PinceVide devient Faux, Tenir(Bi) devient Vrai
  - Déposer(Bi) : sert à déposer Bi sur la table
    - Préconditions : Tenir(Bi)
    - Fonction successeur :
      - SurTable(Bi) devient Vrai, Libre(Bi) devient Vrai, PinceVide devient Vrai, Tenir(Bi) devient Faux

PAS LA SEULE MODELISATION POSSIBLE
…
On peut par exemple considérer la table comme un bloc, mais les préconditions et les fonctions successeurs se complexifient…

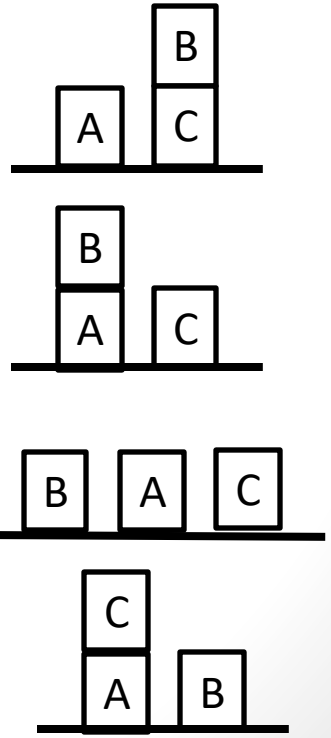
Résolution du problème

- Avec une recherche dans un espace de solutions
  - On va générer énormément d'états à tester …
- Avec une résolution par décomposition de problèmes
  - Quelle décomposition ?
  - ➢ Essayer sur des exemples…
  - ➢ Reprenons la configuration donnée et partons à l'envers pour trouver quelques sous-problèmes

Décomposition du problème



- Sous-problème 1 :
  - Pour arriver à la situation finale
    à partir d'une situation où « A est sur la table » et « B est sur C »,
    la solution est triviale.
- Sous-problème 2 :
  - Pour arriver à la situation où « A est sur la table » et « B est sur C »,
    à partir d'une situation où « B est sur A » et « C est sur la table »
    la solution est triviale.
- Sous-problème 3 :
  - Pour arriver à la situation où « B est sur A » et « C est sur la table »,
    à partir de la situation où « A, B et C sont sur la table »
    la solution est triviale.
- Sous-problème 4 :
  - Pour arriver de la situation où « A, B et C sont sur la table »,
    à partir de la situation où « C est sur A » et « B sur la table »
    la solution est triviale.



Règle de décomposition

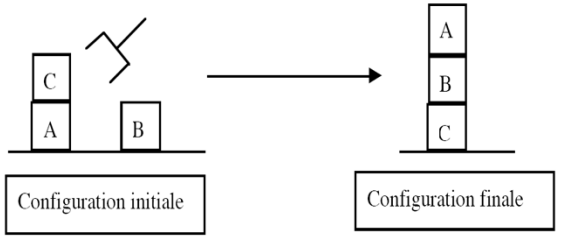
- Avec cet exemple, on voit qu'une règle de décomposition peut être de toujours essayer de mettre tous les cubes sur la table

➢ Cette décomposition n'est pas terrible… mais la simplicité marche !

➢ Pas la seule décomposition … à vous de réfléchir ;-)

➢ Mais attention aux effets de bords …

- En résolvant un sous-but, on peut « casser » un autre sous-but qui était résolu précédemment.
  - Par exemple, si je veux poser A sur B, je dois
  - (1) libérer A, (2) libérer le dessus de B puis (3) mettre A sur B…
  - mais en libérant B, je peux recouvrir A qui ne sera plus accessible pour le dernier sous-but….



De quoi va-t-on parler ?

- Qu'est-ce qu'un problème en IA ?
- Modélisation d'un problème
- Résolution d'un problème
  - Recherche dans l'espace des solutions
  - Par décomposition de problèmes
- Résolution par satisfaction de contraintes
- Zoom sur l'IA pour les jeux
- Pour aller plus loin

Représentation des problèmes

- Avantage de cette approche :
  - La représentation des connaissances est explicite
    - Elle n'est pas « cachée » dans la fonction d'évaluation de l'heuristique
  - Elle peut être appliquée à de très nombreux problèmes
- Limite :
  - Elle nécessite une bonne capacité de modélisation
  - Pour présenter le problème de manière adaptée à la méthode
- Principe :
  - Identifier les « variables » du problème = ce qui peut varier dans la description d'un état
  - A chaque variable, on associe un ensemble de valeurs possibles
  - Puis on contraint les valeurs possibles des variables par des relations que leurs valeurs doivent respecter impérativement

Représentation des problèmes

- Plus formellement, on définit un problème par :
  - $X = \{X_1 , X_2 , ..., X_n\}$ l'ensemble des variables caractérisant le problème
  - $D(X_i)$ le domaine de chaque variable X_i = l'ensemble des valeurs que X_i peut prendre théoriquement
  - $C = \{C_1 , C_2 , ..., C_k\}$ l'ensemble des contraintes
  - Chaque contrainte C_j est une relation entre certaines variables de X restreignant les valeurs que peuvent prendre simultanément ces variables
- Souvent plusieurs modélisations possibles !
- Critères de choix :
  - simplicité d'expression
  - efficacité : taille de l'espace de recherche de solutions

Exemple des 4 reines

- Une modélisation possible : on met une reine par ligne
- Les variables
  - $X = \{x_1, x_2, x_3, x_4\}$ avec x_i où i représente la colonne
- Leur domaine de valeur
  - $D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{1, 2, 3, 4\}$
- Les contraintes
  - $C = \{c_1, c_2\}$
  - Les reines doivent être sur des colonnes différentes :
  - $c_1 = \{x_i \neq x_j / i \in \{1, 2, 3, 4\}, j \in \{1, 2, 3, 4\} \text{ et } i \neq j\}$
  - Les reines doivent être sur des diagonales différentes :
  - $c2 = \{x_{i+j} \neq x_i + j \text{ et } x_{i+j} \neq x_i - j$
    $/ i \in \{1, 2, 3, 4\}, j \in \{-4, -3, -2, -1, 1, 2, 3, 4\} \text{ et } i \neq j\}$

Résolution d'un CSP

- Résolution :
  - Affectation de valeurs aux variables
    de telle sorte que toutes les contraintes soient satisfaites

- Affectation totale : affectation de toutes les variables
  - $A = \{(x_1 , 1), (x_2 , 1), (x_3 , 1), (x_4 , 1)\}$
- Affectation partielle : affectation de certaines variables
  - $A = \{(x_1 , 1), (x_3 , 2)\}$
- Affectation consistante : affectation qui ne viole aucune contrainte
  - $A = \{(x_1 , 1), (x_3 , 2)\}$
- Affectation inconsistante : affectation qui viole au moins une contrainte
  - $A = \{(x_1 , 1), (x_2 , 2)\}$

- Solution : affectation totale et consistante
  - $A = \{(x_1 , 2), (x_2 , 4), (x_3 , 1), (x_4 , 3)\}$

Generate and Test

- Principe :
  - Générer les solutions possibles théoriquement (celles conformes aux domaines des variables)
  - Puis tester que les contraintes sont satisfaites

- Sur le problème des reines
  - A = {(x_1 , 1), (x_2 , 1), (x_3 , 1), (x_4 , 1)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 1), (x_3 , 1), (x_4 , 2)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 1), (x_3 , 1), (x_4 , 3)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 1), (x_3 , 1), (x_4 , 4)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 1), (x_3 , 2), (x_4 , 1)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 1), (x_3 , 2), (x_4 , 2)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 1), (x_3 , 2), (x_4 , 3)} => inconsistante
  - A = {(x_1 , 1), (x
  - ….

La combinatoire est désastreuse
Il faut utiliser des connaissances supplémentaires pour rendre la méthode efficace

Méthode du retour arrière

- Principe :
  - Ne pas développer une solution partielle qui viole déjà les contraintes

- Dans l'exemple des reines :
  - A chaque fois que l'on place une reine, on vérifie qu'elle n'est pas attaquée
  - Sinon on révise les choix précédents

  - A = {(x_1 , 1), (x_2 , ?), (x_3 , ?), (x_4 , ?)} => consistante
  - A = {(x_1 , 1), (x_2 , 1), (x_3 , ?), (x_4 , ?)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 2), (x_3 , ?), (x_4 , ?)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 3), (x_3 , ?), (x_4 , ?)} => consistante
  - A = {(x_1 , 1), (x_2 , 3), (x_3 , 1), (x_4 , ?)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 3), (x_3 , 2), (x_4 , ?)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 3), (x_3 , 3), (x_4 , ?)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 3), (x_3 , 4), (x_4 , ?)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 4), (x_3 , ?), (x_4 , ?)} => consistante
  - A = {(x_1 , 1), (x_2 , 4), (x_3 , 1), (x_4 , ?)} => inconsistante
  - A = {(x_1 , 1), (x_2 , 3), (x_3 , 2), (x_4 , ?)} => consistante
  - ….

Filtrage

- Principe :
  - Réduire le domaine des variables à chaque affectation

- Dans l'exemple des reines :
  - A chaque fois que l'on pose une reine, on élimine du domaine toutes les positions qui sont maintenant attaquées

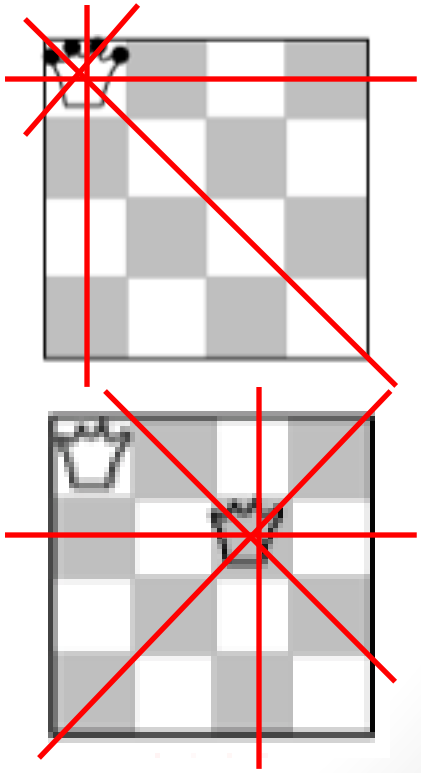
  - A = {(x_1 , 1), (x_2 , ?), (x_3 , ?), (x_4 , ?)}
  - D(x_1) = {1}, D(x_2) = {3, 4}, D(x_3) = {2, 4}, D(x_4) = {2, 3}

  - A = {(x_1 , 1), (x_2 , 3), (x_3 , ?), (x_4 , ?)}
  - D(x_1) = {1}, D(x_2) = {3}, D(x_3) = {}, D(x_4) = {2}

  - … retour arrière et on re-filtre ….

  - A = {(x_1 , 2), (x_2 , ?), (x_3 , ?), (x_4 , ?)}
  - D(x_1) = {2}, D(x_2) = {4}, D(x_3) = {1, 3}, D(x_4) = {1, 3, 4}

  - …



Utiliser une heuristique

- Heuristiques générales aux CSP :
  - Heuristique des valeurs minimum restantes (MRV)
    - Choisir une des variables ayant le plus petit domaine de valeur
  - Heuristique du degré
    - Choisir la variable qui a le plus de contraintes à respecter
  - Heuristique de la valeur la moins contraignante
    - Choisir la variable qui empêche le moins d'affectations possibles sur les variables restantes
  - …
- Heuristique spécifique au problème :
  - Par exemple pour les N reines,
    - Lorsqu'on place une reine dans le premier ou dernier tiers de l'échiquier , il faut commencer par les valeurs du milieu
    - Lorsqu'on place une reine dans le tiers du centre, il faut commencer par les variables du bord de manière alternée
    - ….

➢ Les TD seront l'occasion de mettre en œuvre ces quatre approches…

De quoi va-t-on parler ?

- Qu'est-ce qu'un problème en IA ?
- Modélisation d'un problème
- Résolution d'un problème
  - Recherche dans l'espace des solutions
  - Par décomposition de problèmes
- Résolution par satisfaction de contraintes
- Zoom sur l'IA pour les jeux
- Pour aller plus loin

L'IA et les jeux

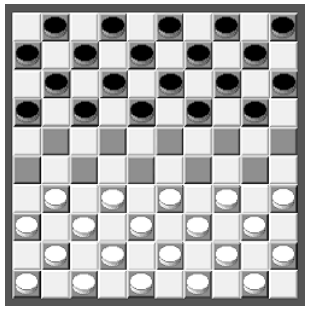
- Nombreux types de jeux avec propriétés très différentes
- En IA, classe de jeux finis déterministes à 2 joueurs, avec tours alternés, à information complète, et à somme nulle

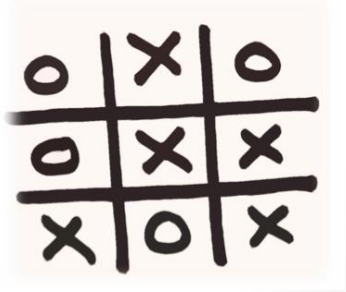
- Un jeu est **fini** si le jeu termine toujours
  - Dans beaucoup de jeux, e.g. les échecs, les règles du jeu imposent des conditions pour rendre le jeu fini
- Un jeu est **déterministe** si le déroulement du jeu est entièrement déterminé par les choix des joueurs
  - Des jeux utilisant les dés (e.g. le Monopoly, le backgammon) ou des cartes distribuées au hasard (e.g. le poker,) sont non-déterministes
- Dans les jeux **à information complète**, les joueurs connaissent parfaitement la configuration actuelle du jeu
  - Ce n'est pas le cas, par exemple, avec la plupart des jeux de cartes (e.g. le poker)
- Un jeu est dit à **somme nulle** si la valeur gagnée par l'un des joueurs est la valeur perdue par l'autre.
  - Le poker est un jeu à somme nulle parce que le montant qui est gagné par un joueur est égal à la somme des montants perdus par les autres joueurs
  - Les jeux où il y a toujours un gagnant et un perdant sont des jeux à somme nulle

Exemples de jeux de cette classe

- Les échecs, le jeu de dames, le morpion…

- Le morpion
  - Il est fini parce qu'une partie dure au plus 9 tours
  - Il est déterministe parce que le hasard n'intervient pas
  - Il est à information complète parce que les deux joueurs connaissent l'état de la grille
  - Il est à somme nulle parce que lorsqu'un joueur gagne, c'est l'autre qui perd





Modélisation de ces jeux

- Etats
  - Toute configuration du jeu associé au nom du joueur à jouer
- Etat initial
  - Une des configuration du jeu associé au nom du joueur à jouer
- Opérateurs
  - Condition d'application : définit quelles actions sont possibles pour un joueur dans une configuration donnée
  - Etat successeur : la configuration qui résulte
- Test de but
  - Les configurations terminales du jeu
- **Fonction d'utilité pour associer une valeur à chaque configuration terminale**
  - Souvent, les valeurs sont +1, -1, et 0 qui correspondent à un gain pour le premier joueur, une perte pour le premier joueur, et un match nul
  - Comme jeux à somme nulle, besoin que d'une seule valeur par configuration : la valeur attribuée au 2ème joueur est exactement le négatif de la valeur reçue par le 1er joueur

Objectif de la résolution du pb

- L'objectif du jeu pour un joueur : maximiser son gain.

- Plus spécifiquement, le premier joueur veut que la valeur d'utilité de la configuration terminale du jeu soit la plus grande possible,
- Et inversement, le deuxième joueur souhaite que cette valeur soit la plus petite possible

- Afin de pourvoir se rappeler de leurs objectifs respectifs, il est courant d'appeler les deux joueurs par *max* et *min*.

Modélisation du morpion

- Etats
  - Une grille de 3*3 avec dans chaque case {X, O ou vide} + le nom du joueur courant
- Etat initial
  - La grille vide et un nom de joueur à jouer, X par exemple
- Opérateur
  - Remplir une case $C_{i,j}$ avec un caractère X ou O
  - Cond. d'application : quand c'est le tour de joueur X, il peut mettre un X dans un des cases vides de la grille ; même chose pour O
  - Etat successeur : la grille précédente avec un caractère en plus dans la case choisie
- Test de but
  - Une configuration est terminale s'il y a trois X (ou trois O) sur une ligne, une colonne, ou une diagonale, ou si la grille est complète.
- Fonction d'utilité
  - +1 à toute configuration terminale contenant trois X dans une ligne, colonne, ou diagonale (c'est X qui gagne)
  - -1 à toute configuration terminale avec trois O dans une même ligne, colonne, ou diagonale (c'est O qui gagne)
  - 0 pour les autres configurations terminales (match nul)

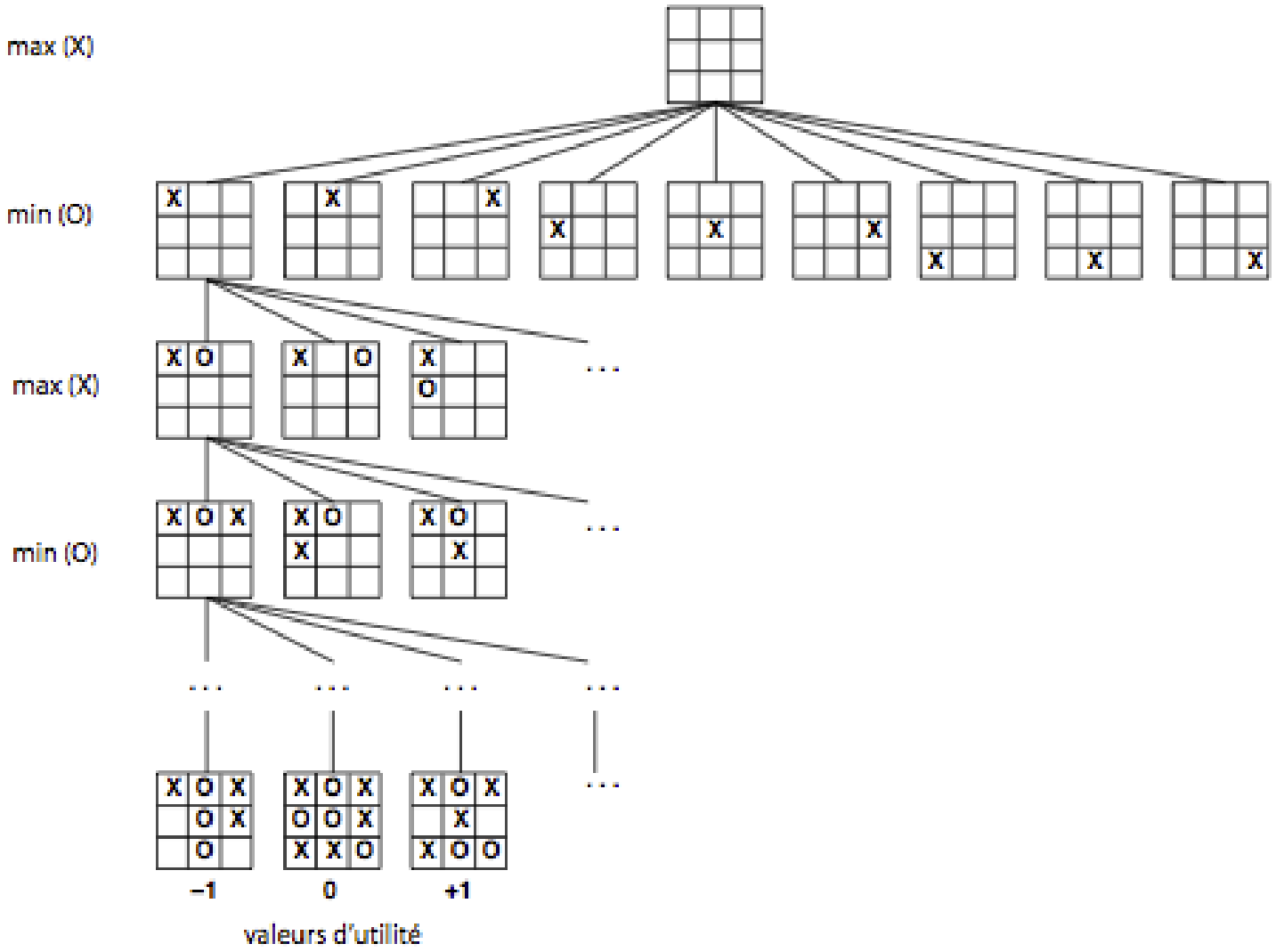
L'arbre de jeu

- L'arbre de jeu
  - Les nœuds sont les configurations (les états)
  - Les arcs les coups possibles
  - Les feuilles de l'arbre sont des configurations terminales
    - étiquetées par leurs valeurs d'utilité

- Un nœud est appelé **nœud max** si c'est à *max* de jouer
- Un nœud est appelé **nœud min** si c'est à *min* de jouer

- Dans le cas du morpion
  - La racine de l'arbre = configuration initiale où la grille est vide
  - Les fils de la racine = 9 coups possibles pour le joueur max
  - Chacun de ces nœuds possède 8 fils = les différentes façons pour min de répondre
  - Et ainsi suite

L'arbre de jeu pour le morpion



Génération de stratégies

- Pour les problèmes de recherche
  - Nous cherchons une solution
  - i.e. une suite d'actions reliant l'état initial à l'état but

- Pour les jeux, une simple suite de coups n'est pas suffisante
  - Parce que nous ne savons pas quels seront les coups de l'adversaire !!!

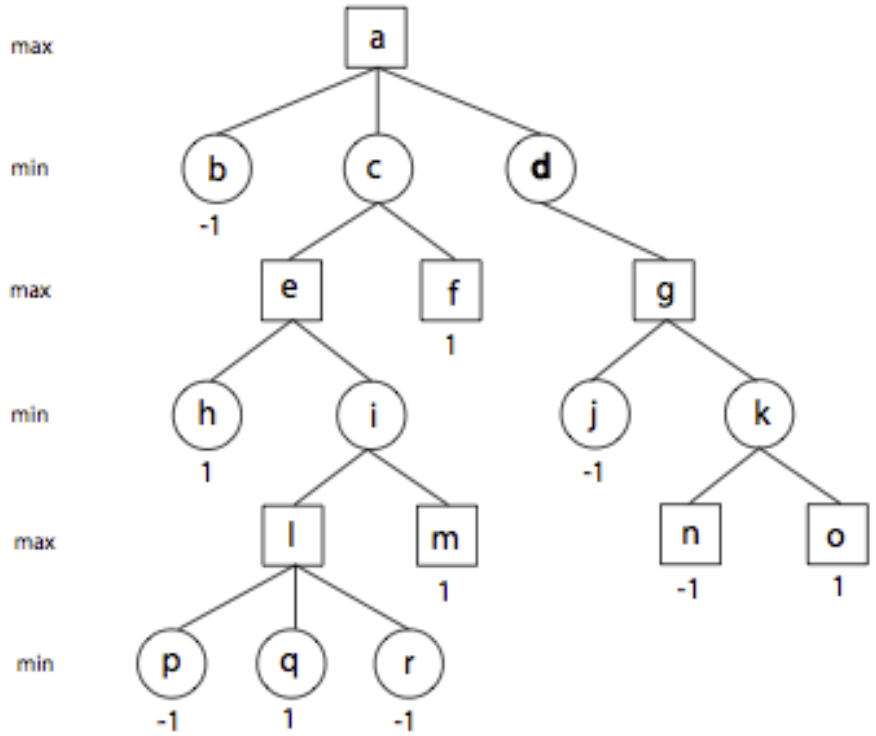
- Il faut donc trouver une stratégie qui définit comment le joueur doit jouer dans toutes les évolutions possibles du jeu
  - Une stratégie consiste en un premier coup,
  - puis un choix de coup pour chaque réponse possible de l'adversaire,
  - puis un choix de coup pour les réponses à notre dernier coup,
  - et ainsi suite.

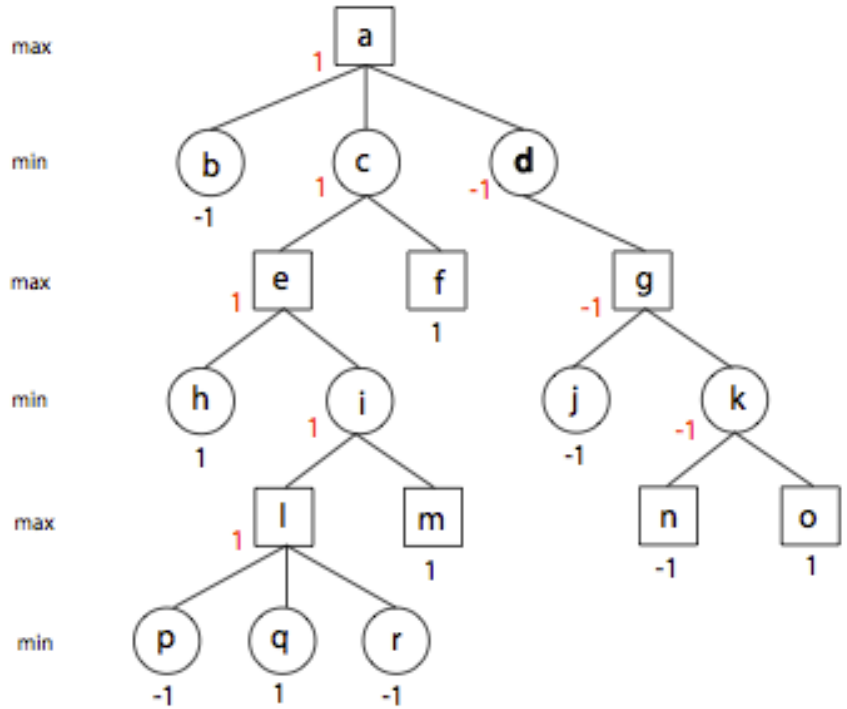
- On dit qu'une **stratégie est optimale** s'il n'existe pas d'autre stratégie qui donnerait un meilleur résultat contre un adversaire idéal.

Algorithme MinMax

- Pour trouver une stratégie optimale, il faut examiner les valeurs minimax des nœuds de l'arbre de jeux
- La valeur minimax d'un nœud = la valeur d'utilité qui sera atteinte si chaque joueur joue de façon optimale à partir de la configuration de ce nœud
- Pour calculer ces valeurs :
  - Commencer avec les configurations terminales, dont les valeurs minimax sont leurs valeurs d'utilité
  - Puis évaluer chaque nœud non-terminal en utilisant des valeurs minimax de ces fils :
    - La valeur minimax d'un nœud max sera le maximum des valeurs minimax de ces fils, parce que le joueur max veut maximiser la valeur de la configuration terminale.
    - La valeur minimax d'un nœud min sera le minimum des valeurs minimax de ces fils car le joueur min cherche à obtenir une valeur minimale pour la configuration terminale.

Exemple des valeurs *minimax*

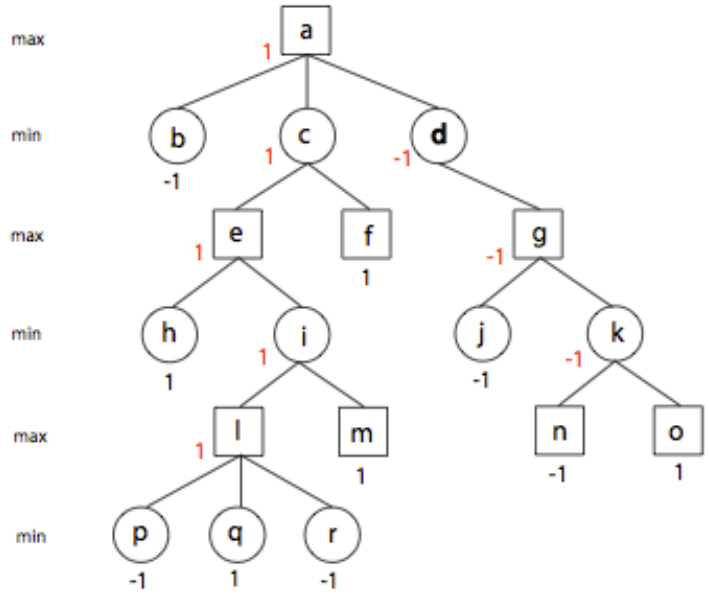




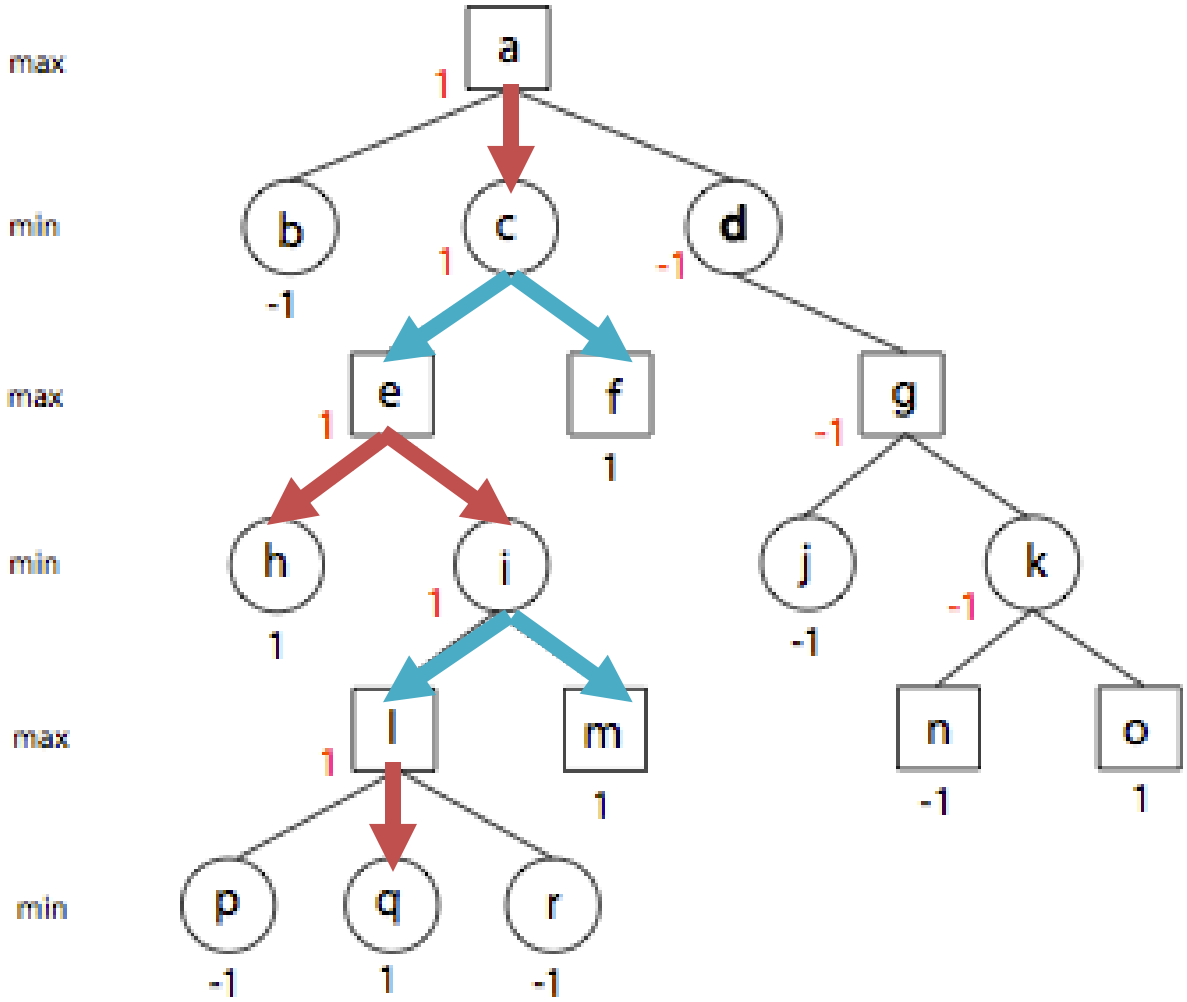
Algorithme MinMax

- Comment déterminer la stratégie optimale à partir des valeurs minimax ?
  - Si nous sommes le joueur max et que c'est à nous de jouer
  - Alors choisir un coup qui amène à un nœud avec une valeur minimax maximale.
  - Inversement, pour le joueur min, le meilleur coup est celui qui a la plus petite valeur minimax.

- Dans l'exemple
  - La valeur de a est 1
  - Donc le joueur max peut gagner s'il suit la stratégie optimale
  - Peu importe les coups de l'autre joueur !



Stratégie donnée par MinMax



Calcul des valeurs *minimax*

- Parcours en profondeur ou en parcours en largeur
  - Préférable en profondeur pour réduire la complexité spatiale

- Pour un arbre avec un facteur de branchement de b et une profondeur p
  - Complexité spatiale de O(p*b) : il y a au plus p x b nœuds en mémoire en même temps
  - Complexité temporelle de $O(b^p)$: besoin de visiter tous les nœuds de l'arbre

```
fonction MINIMAX(n)
   si n est un noeud terminal alors
      RETOURNER la valeur d'utilité de n
   sinon
      si n est un nœud de type min alors
         v = +∞
         pour tout fils f de n faire
            v = MIN(MINIMAX(f), v)
         RETOURNER v
      sinon
         v = −∞
         pour tout fils f de n faire
            v = MAX(MINIMAX(f), v)
          fin pour
         RETOURNER v
      fin si
   fin si
fin
```

Les algorithmes MinMax

- Peut être étendu à plus de 2 joueurs
  - Il faut un vecteur d'utilités pour prendre en compte les utilités des différents joueurs

- Peut être étendu aux jeux stochastiques
  - Il faut ajouter des nœuds supplémentaires aux arbres de jeu pour prendre en compte l'aspect stochastique
  - e.g. des noeuds pour représenter les lancers de dés
  - et en utilisant les probabilités des différentes configurations résultantes dans le calcul des valeurs minimax.
  - ➢ Algorithme *expectiminimax*

➢ Concrètement : le temps de calcul rend MinMax inutilisable en pratique pour pouvoir prendre les décisions en temps réel

Optimisation des MinMax

- Limiter la profondeur d'exploration
  - Algorithme Min-Max avec profondeur bornée
- Effectuer des coupes dans l'arbre de jeu
  - Coupes Alpha et Bêta
- Ne pas lancer une exploration dès le début de la partie
  - Calculer les premiers coups à l'avance, avec une profondeur de recherche élevée (= ouvertures apprises par cœur par les joueurs humains)
- Se constituer une bibliothèque de fins de partie
  - Ces calculs peuvent être faits à l'avance et les résultats stockés
- Faire varier la profondeur de recherche selon le moment / l'état de la partie
  - Aller plus profond
    - lorsqu'il n'y a pas beaucoup de coups possibles et que l'on peut donc explorer plus profond dans le temps imparti
    - lorsque les situations évaluées aux feuilles semblent instables et qu'il faut poursuivre un peu pour déterminer qui a vraiment l'avantage
    - uniquement pour le coup qui a été choisi et dans le but de confirmer ce choix pour combattre l'effet d'horizon (défaut connu des algorithmes de type Min-Max)
    - en fin de partie

De quoi va-t-on parler ?

- Qu'est-ce qu'un problème en IA ?
- Modélisation d'un problème
- Résolution d'un problème
  - Recherche dans l'espace des solutions
  - Par décomposition de problèmes
- Résolution par satisfaction de contraintes
- Pour aller plus loin

Pour aller plus loin

- Sur la résolution de problèmes en général
  - Artificial Intelligence : A Modern Approach, Stuart Russell & Peter Norvig
- Sur l'IA pour les jeux
  - https://fabien-torre.fr/Enseignement/Cours/Intelligence-Artificielle/jeux.php
- Sources pour construire ce cours
  - Les cours d'Alain Mille et de Meghyn Bienvenu