

# Intelligence Artificielle Résolution des Problèmes (Recherche)

# Les grandes questions de l'IA

- Acquérir des connaissances, Représenter
- Algorithmes généraux de résolution de problème
- Intelligence artificielle « collective »
- Formaliser, mécaniser les types de raisonnement
- Evaluer des situations, décider, planifier
- Reasonner sur le temps et l'espace
- Résumer, apprendre, découvrir
- Langue et IA
- Indexation et IA
- Réalité virtuelle et IA

# Multiples facettes de l'IA

- **Facette des mathématiques**

- formalisation du raisonnement mathématique (logique)
- Contribution à de nouveaux champs (logiques modales  $\sqsubset$  logique possibiliste)

- **Facette informatique**

- ***Nouveau « paradigmes » de programmation***
  - Programmation fonctionnelle
  - Programmation logique
  - Programmation objet
  - Programmation Agent
- ***Nouvelles façons de voir les systèmes d'information***
  - Gestion de la connaissance
  - Indexation WEB (semantic web)
  - Description des documents numériques
- ***Applications nombreuses***
  - Aide à la décision
  - Aide au diagnostic
  - Aide à la planification
  - Aide au traitement automatique de la langue
  - Aide à la conception
  - ...
- ***Omniprésence dans les Systèmes de Traitement de l'Information et de la Communication (STIC)***

## *Résolution des Problèmes ( Recherche)*

## ***LES STRATÉGIES DE RÉOLUTION DE PROBLÈME***

- Explorer attentivement et méthodiquement la situation
- Aller chercher toutes les informations pertinentes
- Noter, comparer, classer et retenir les informations essentielles
- Explorer les différentes pistes de solution et vérifier leur validité
- Évaluer et comparer les différentes solutions valides afin d'assurer le meilleur choix
- Faire le point pour savoir où l'on est, ce qu'on a appris et les voies encore à explorer

## ***Les avantages de bien définir un problème avant de chercher à le résoudre\_***

- Avoir une idée précise de ce que l'on cherche
- Distinguer plus facilement l'information pertinente de celle qui ne l'est pas
- Réduire le sentiment de désorientation face à un problème nouveau
- Apporter des solutions durables aux problèmes

## Critères d'évaluation

*Il y a de nombreux **types de critères** différents selon la nature du problème, la qualité et la quantité des alternatives de solution et l'importance de la décision à prendre. La détermination des critères d'évaluation des idées de solution est donc un processus très spécifique à la situation. Cependant, un certain nombre de catégories de base reviennent souvent :*

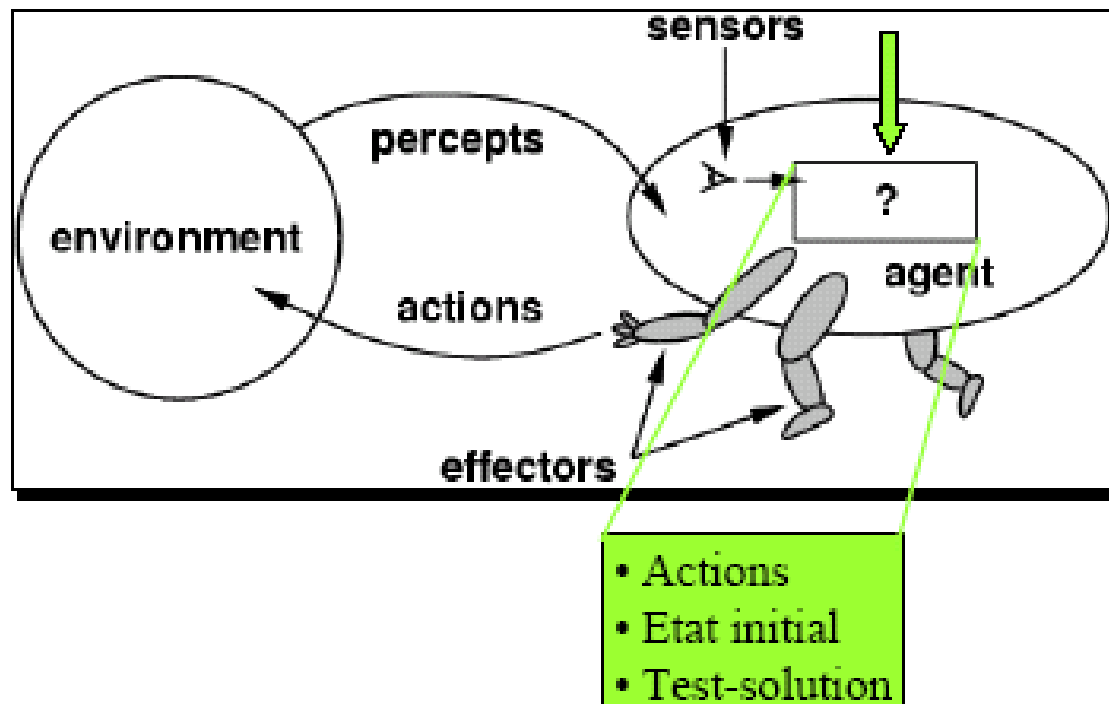
- **Les coûts** : coûts financiers, économiques, énergétiques, etc. en relation avec les disponibilités de même ordre comme le budget, l'expertise, l'effort qu'on est prêt à fournir, etc. Combien la solution coûte-t-elle? Est-on en mesure de se l'offrir?
- **Le temps** : Quelle est l'urgence? Combien de temps ça prendra pour rendre la solution opérationnelle? La solution fait-elle gagner du temps?
- **La faisabilité** : La solution est-elle applicable concrètement dans le contexte donné?
- **L'acceptabilité** : La solution plaira-t-elle aux personnes impliquées? Sera-t-elle utilisée? Rencontrera-t-elle de la résistance?
- **L'utilité** : la solution apporte-t-elle quelque chose de plus? Quel est le bénéfice?

## Problèmes & Algorithmes de recherche (généralités)

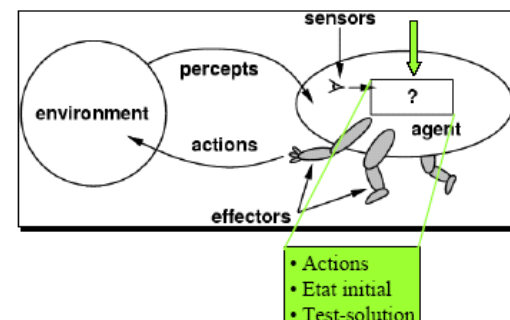
- Agents de résolution de problèmes
- Types de problèmes
- Formulation de problèmes
- Exemples de problèmes
- Algorithmes de recherche  
(aveugles/heuristiques/optimistes)



## Agent de résolution de problèmes



# Agents de résolution de problèmes



- Ce sont des agents basés sur des objectifs.
- La solution d'un problème s'obtient sous la forme d'une séquence d'actions qui mène aux états-solutions souhaités.
- Pour cela il faut:
  - exprimer **l'objectif** comme un sous-ensemble des états possibles du monde(Environnement),
  - spécifier clairement quel est **l'espace d'états** du problème: c-à-d quels sont les aspects du monde importants pour le problème à résoudre,
  - spécifier clairement toutes les **actions possibles**.
- Une fois que **l'espace d'états**, **l'ensemble d'actions**, **les états-solutions** ont été identifiés et spécifiés, alors la séquence d'actions menant de **l'état initial** à **l'état-objectif** peut être obtenue par une **méthode de recherche**.

## Introduction aux algorithmes de recherche

- Les algorithmes de recherche constituent l'une des approches les plus puissantes pour la résolution de problèmes en IA.
- Les algorithmes de recherche sont un mécanisme général de résolution de problèmes qui:
  - se déroule dans un espace appelé *espace d'états*
  - explore systématiquement toutes les *alternatives*
  - trouve la *séquence d'étapes* menant à la solution
- "Problem Space Hypothesis"  
(Allen Newell, SOAR: An Architecture for General Intelligence)
  - toute recherche orientée vers un but se déroule dans *l'espace du problème* (*espace d'états*)
  - toute recherche dans l'espace du problème constitue un *modèle général de comportement intelligent*.

# Définir un problème de recherche

Certains problèmes peuvent être définis par:

- **Espace d'états**
  - chaque état est une représentation abstraite de l'environnement
  - l'espace d'états est discret
- **État initial**
  - habituellement l'état courant
  - parfois un ou plusieurs états hypothétiques
- **Fonction "successeur" (Opérateurs)**
  - fonction : [ état  $\rightarrow$  sous-ensemble d'états ]
  - une représentation abstraite des actions possibles
- **Test-solution**
  - habituellement une condition à satisfaire
  - parfois la description explicite d'un état
- **Coût du chemin**
  - fonction : [ chemin  $\rightarrow$  nombre positif ]
  - habituellement: coût du chemin = somme des coûts de ses étapes

## Définir un espace d'états

- Le monde réel est excessivement complexe  
⇒ l'espace d'états doit être une *abstraction* de la réalité
- *état* (abstrait) = ensemble d'états réels
- *opérateur* (abstrait) = combinaison d'actions réelles
- *solution* (abstraite) = ensemble de chemins-solutions dans le monde réel

## Agent simple de résolution de problèmes

```
function Simple-Problem-Solving-Agent (p) returns an action
{
  inputs : p a percept
  static : seq , //an action sequence, initially empty
  state , //some description of the current world state
  goal , //a goal, initially null
  problem , //a problem formulation
  state ← Update-State (state, p)
  if seq is empty then
    goal ← Formulate-Goal (state)
    problem ← Formulate-Problem (state, goal)
    seq ← Search (problem)

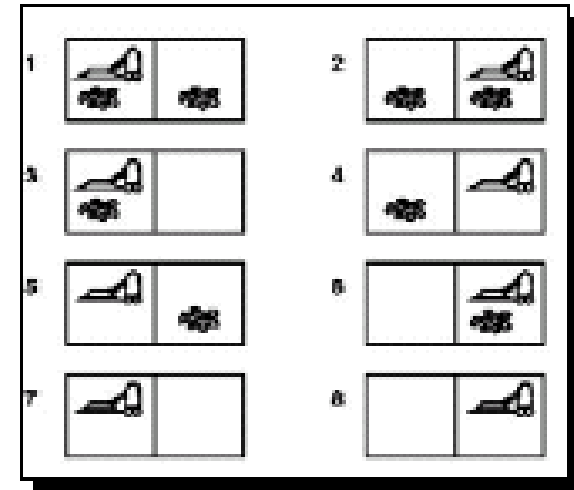
  action ← Recommendation (seq, state)
  seq ← Remainder (seq, state)
return action
}
```

Note:

- Une méthode "offline" de résolution de problèmes, la solution est exécutée "les yeux fermés".
- Les méthodes "online" nécessitent souvent d'agir sans une connaissance complète.

### ***Exemple :l'aspirateur***

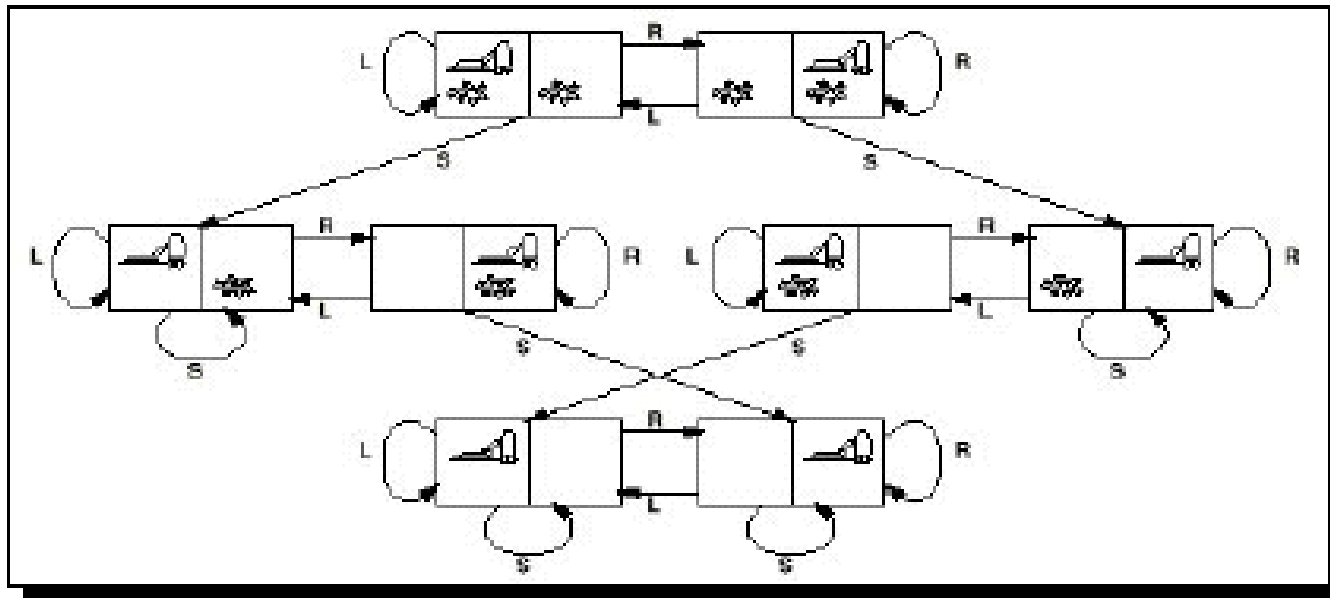
- Le monde consiste en 2 positions
  - pièce gauche, pièce droite
- chaque pièce peut être poussiéreuse
- l'agent (aspirateur) est dans l'une des 2 pièces
- il y a au total 8 états possibles
- 3 actions ( ou 4) possibles:
  - aller à gauche, aller à droite, aspirer, ne rien faire



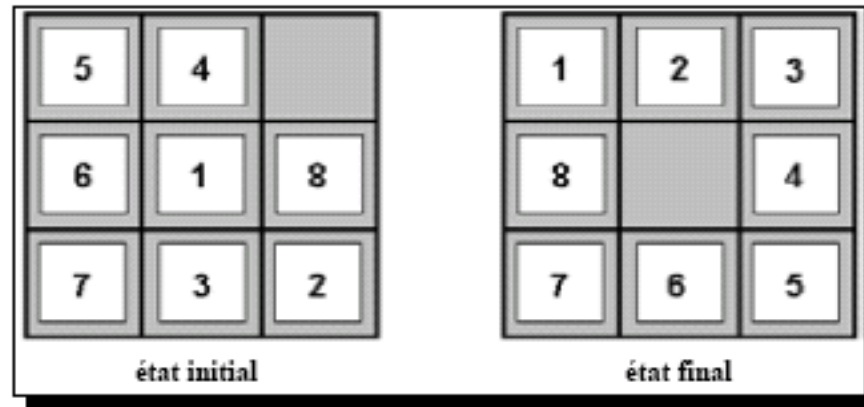
### ***Exemple : l'aspirateur***

- **Etat initial**: L'aspirateur est dans l'une des 2 pièces
- **formulation du but**:
  - éliminer toute la poussière
- **formulation du problème**:
  - **états**: les 8 états possibles du problème
  - **actions**: se déplacer à gauche, à droite, aspirer
- **solution**:
  - être dans l'état 7 ou 8





- **états**: entiers indiquant la position du robot et de la poussière (ignorer la quantité de poussière)
- **opérateurs**: aller à gauche (L), aller à droite (R), aspirer (S), ne rien faire (NoOp)
- **test**: plus de poussière nulle part
- **fonction-coût**: chaque action coûte 1 unité, (0 pour NoOp)

*Exemple: puzzle-8*

- **états:** numéros des positions des plaquettes (ignorer les positions intermédiaires)
- **opérateurs:** déplacer la case vide à gauche (L), à droite (R), en haut (U), en bas (D)
- **test:** état courant = état final
- **fonction-coût:** chaque déplacement de la case vide vaut 1, coût total = nombre total de déplacements de la case vide

**remarque:** solution optimale pour puzzle-n est NP-difficile

dimension de l'espace d'états ( machine à 10 millions états/sec)

puzzle-8    181,440     $\square$  0.18 sec

puzzle-15  $\square$   $.65 \times 10^{12}$      $\square$  6 jours

puzzle-24  $\square$   $.5 \times 10^{25}$      $\square$  12 milliards d'années

## *Exemple : arithmétique cryptée*

FORTY	Solution: 29786	F=2, O=9, R=7, etc
+ TEN	850	
+ TEN	850	
-----	-----	
SIXTY	31486	

- **état initial:** problème d'arithmétique avec des chiffres remplacés par des lettres
- **opérateurs:** remplacer toutes les occurrences d'une lettre par un chiffre non-répété
- **test:** problème ne contient que des chiffres et représente la somme correcte
- **fonction-coût:** pas applicable ou 0

# Problèmes réels

- **Recherche de parcours**

- itinéraires automatiques, guidage routier, planification de routes aériennes, routage sur les réseaux informatiques, ...

- **Configuration de circuits VLSI**

- placement ([démon](#)) de millions d'éléments sur un chip déterminant pour le fonctionnement efficace du circuit

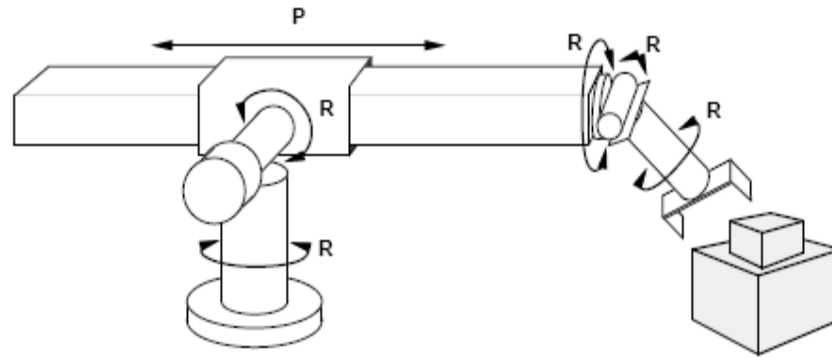
- **Robotique**

- assemblage automatique, navigation autonome, ...

- **Planification et ordonnancement**

- horaires, organisation de tâches, allocation de ressources, ...

## *Exemple: assemblage automatique*



- **états:** coordonnées des articulations du robot et des pièces à assembler
- **opérateurs:** mouvements continus des articulations du bras robotique
- **test:** assemblage terminé, robot en position de repos
- **fonction-coût:** temps d'exécution

## Typologie des problèmes de recherche?

- Tous les problèmes qui peuvent être décrits par un ensemble fini d'états, un ensemble fini d'actions, un sous-ensemble d'états initiaux et finaux, une relation "successeur" définie sur l'ensemble des états et des actions dans l'ensemble des états, et une fonction de coût positive.
- Principalement les problèmes dont la solution s'exprime en termes de chemin dans des graphes finis.

## Formulation d'un problème de recherche

- Environnement réel → **Abstraction**
  - **Validité:**
    - La solution est-elle exécutable?
    - Est-ce que l'espace d'états contient la solution?
  - **Utilité:**
    - Est-ce que le problème abstrait est plus facile (à résoudre) que le problème réel?
- Sans **abstraction** l'agent de résolution serait submergé par les informations du monde réel.

## Paramètres importants

- **Nombre d'états** dans l'espace d'états
- **Espace mémoire** nécessaire pour stocker un état
- **Temps d'exécution** de la fonction "successeur"



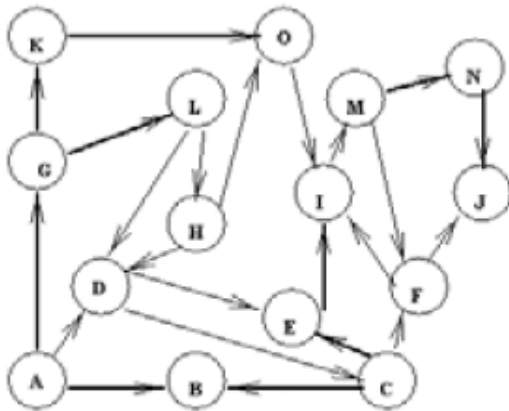
# *Algorithmes de recherche (aveugles)*

- Représentation (Espace d'Etats et Opérateurs)
- Critères d'évaluation des méthodes de recherche
- Méthodes de recherche "aveugles"
  - recherche en largeur
  - recherche en coût uniforme
  - recherche en profondeur
  - recherche en profondeur limitée
  - recherche par approfondissement itératif
  - recherche bi-directionnelle
- Comparaisons des méthodes aveugles

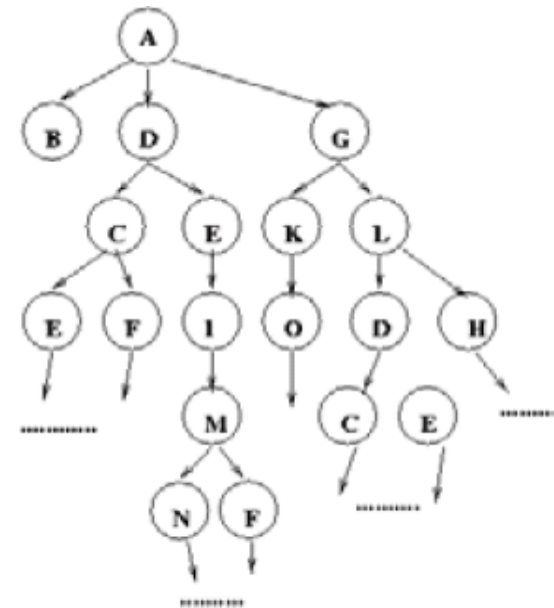
# Représentation: Graphes, Arbres

Nœuds: Etats, Arcs: Actions

Graphe en général



arbre utilisé



## *Méthodes: Critères d'évaluation*

Les différentes méthodes de recherche sont évaluées selon les critères suivants:

- **Complétude:** est-ce que la méthode garantit de trouver une solution si elle existe?
- **Optimalité:** est-ce que la méthode trouve la meilleure solution s'il en existe plusieurs?
- **Complexité en temps:** combien de noeuds faut-il produire pour trouver la solution?
- **Complexité en espace:** nombre maximum de noeuds à conserver en mémoire pour trouver la solution?
- Les complexités en temps et en espace sont mesurées en fonction de:
  - ***b*** = facteur de branchement de l'arbre de recherche (= nombre maximum de successeurs pour un état)
  - ***d*** = profondeur à laquelle se trouve le (meilleur) noeud-solution
  - ***m*** = profondeur maximum de l'espace de recherche (peut être  $\infty$ )

## Concepts de base pour la recherche

En résumé un algorithme de recherche repose sur les éléments suivants:

- Arbre de recherche
- Recherche d'un nœud ( état + pointeurs chemin)
- Expansion (développement) d'un nœud ( recherche des successeurs )
- Stratégie de recherche ( décision sur les nœuds à développer ):  
à chaque étape du processus de recherche,  
déterminer quel est le noeud à "étendre" (développer)

# Stratégies aveugles et stratégies heuristiques

- **Stratégies aveugles**

n'exploitent aucune information contenue dans un noeud donné

- **Stratégies heuristiques**

exploitent certaines informations pour déterminer si un noeud est « plus prometteur » qu'un autre

## Algorithme général de recherche

```
function General-Search (problème, Ordo) returns solution, ou echec
{
    liste_noeuds ← Make-Queue ( Make-Node (Etat-Initial [ problème ]))
loop do
    if liste_noeuds est vide then return echec
    noeud ← Remove-Front (liste_noeuds)
    Make_solution(solution, noeud);
    if Goal-Test [ problème ] (appliqué à Etat (noeud)) succès then Fin Loop
    liste_noeuds ← Ordo (liste_noeuds , Expand (noeud , Operateurs [problème]))
end

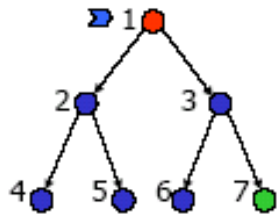
return solution
}
```

Les méthodes de recherche diffèrent les unes des autres selon l'ordre dans lequel les noeuds sont développés (càd les successeurs sont produits et mis dans la file d'attente), rôle des fonctions Expand et Ordo

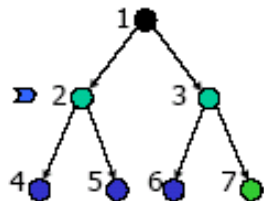
## Algorithme de recherche en largeur (RBF)

- Stratégie: étendre le noeud le moins profond
- Implémentation:

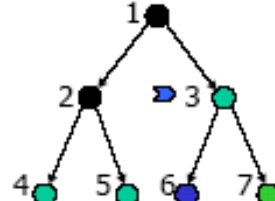
**Ordo** = insertion des successeurs à la fin de la file d'attente



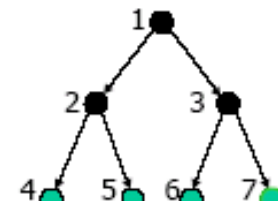
File attente = (1)



File attente = (2, 3)



File attente = (3,4,5)



File attente = (4, 5, 6, 7)



## Algorithme de recherche en largeur

- **Complétude** Oui (si  $b$  est fini)

- **Complexité** en temps

$$1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = (b^{d+1} - 1)/(b - 1) = \mathbf{O}(b^{d+1}) \text{ (exponentiel en } d\text{)}$$

- **Complexité** en espace  $\mathbf{O}(b^{d+1})$  (il faut garder tous les noeuds en mémoire)

- **Optimalité** Oui (si coût unitaire par étape ) en général pas optimal

## Algorithme de recherche en largeur

Complexités en temps et en place pour la recherche en largeur dans les conditions suivantes:

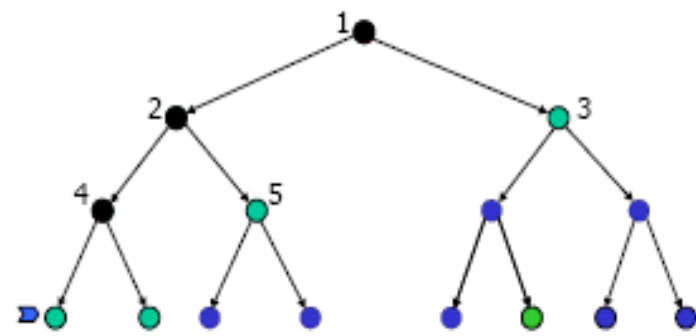
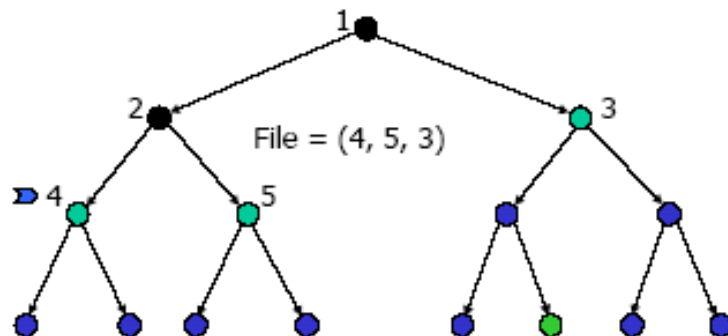
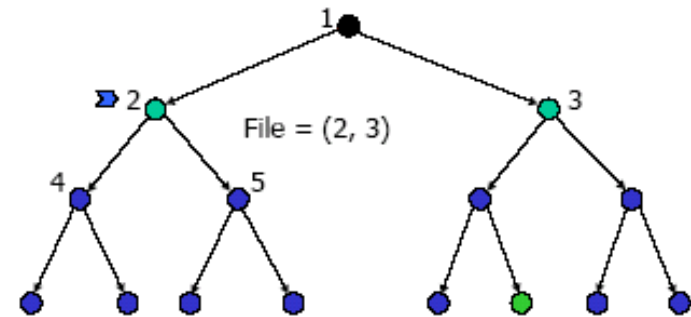
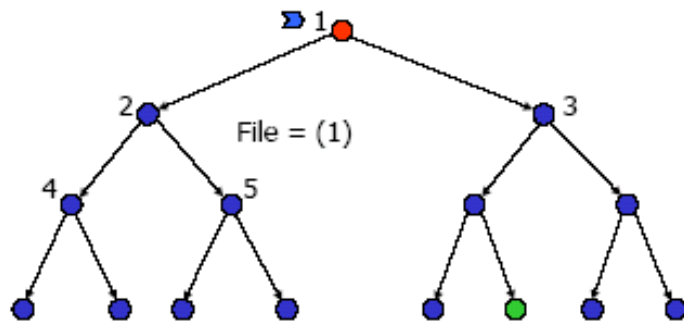
- facteur de branchement  $b = 10$
- production de 1000 noeuds/seconde
- espace-mémoire de 100 octets/noeud

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	$10^6$	18 minutes	111 megabytes
8	$10^8$	31 hours	11 gigabytes
10	$10^{10}$	128 days	1 terabyte
12	$10^{12}$	35 years	111 terabytes
14	$10^{14}$	3500 years	11,111 terabytes

## Algorithme de recherche en profondeur(RDF)

- Stratégie: étend le noeud le plus profond
- Implémentation:

**Ordo** = insertion des successeurs en tête de la file d'attente



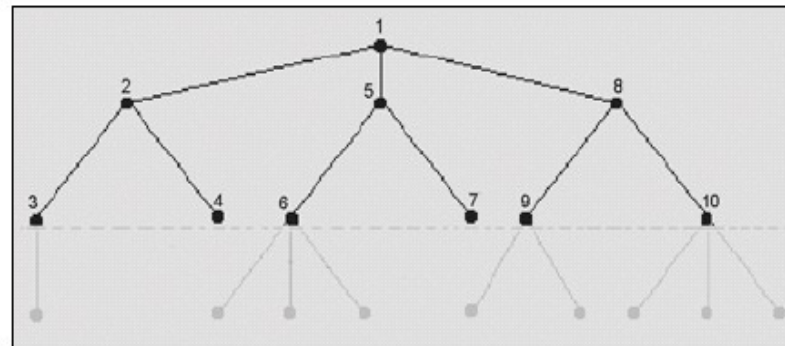
# Algorithme de recherche en profondeur

- **Complétude** Non échoue dans les espaces infinis ou avec cycle  
⇒ complet dans les espaces finis acycliques
- **Complexité en temps**  
 $1 + b + b^2 + \dots + b^m = O(b^m)$  = terrible si  $m$  est beaucoup plus grand que  $d$
- **Complexité en espace**  $O(b * m)$  ou  $O(m)$  linéaire!
- **Optimalité** Non
- **discussion:** besoins modestes en espace
  - pour  $b = 10$ ,  $d = 12$  et 100 octets/noeud:
  - recherche en profondeur a besoin de **12 Koctets**
  - recherche en largeur a besoin de **111 Tera-octets**
  - \*  **$10^{10}$  !!!**

# Algorithme de recherche en profondeur limitée

- algorithme de recherche en profondeur avec une limite de profondeur d'exploration  $L$
- Implémentation
  - les noeuds de profondeur  $L$  n'ont pas de successeurs

exemple avec  $L = 2$



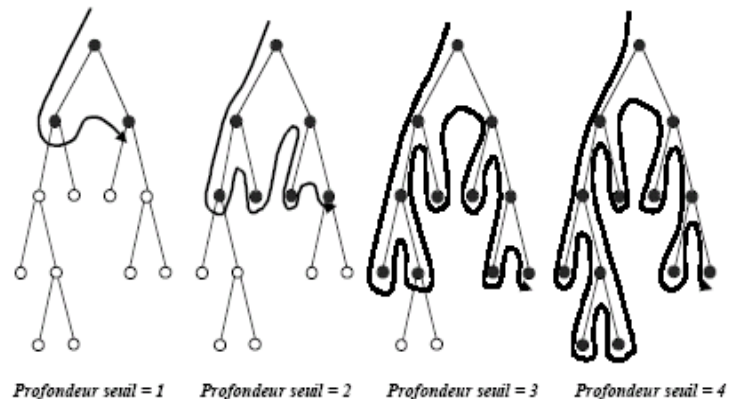
- **Complétude** Oui si  $L \geq d$
- **Complexité en temps**  $O(b^L)$
- **Complexité en espace**  $O(b * L)$
- **Optimalité** Non (oui si coût unitaire)
- **3 possibilités:**
  - solution
  - échec
  - absence de solution dans les limites de la recherche

# Recherche par approfondissement itératif (IDS)

- Le problème avec la recherche en profondeur limitée est de fixer la bonne valeur de  $L$
- approfondissement itératif = répéter pour toutes les valeurs possibles de  $L = 0, 1, 2, \dots$
- combine les avantages de la recherche en largeur et en profondeur
  - optimal et complet comme la recherche en largeur
  - économe en espace comme la recherche en profondeur
- c'est l'algorithme de choix si l'espace de recherche est grand et si la profondeur de la solution est inconnue

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
  inputs: problem, a problem

  for depth  $\leftarrow$  0 to  $\infty$  do
    if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
  end
  return failure
```



## Recherche par approfondissement itératif (IDS)

- **Complétude** Oui il considère systématiquement tous les chemins de longueur 1, 2, 3, ...
- **Complexité en temps**  $(d+1)b^0 + db + (d-1)b^2 + \dots + b^d = O(b^d)$
- **Complexité en espace**  $O(b * d)$
- **Optimalité** Oui si coût unitaire par étape

approfondissement itératif = largeur  $\oplus$  profondeur

- Peut paraître du gaspillage car beaucoup de noeuds sont étendus de multiples fois !
- Mais la plupart des nouveaux noeuds étant au niveau le plus bas, ce n'est pas important d'étendre plusieurs fois les noeuds des niveaux supérieurs.  
( On ajoute à chaque algorithme une liste d'états (noeuds) déjà développés).

Comparaison numérique pour  $b = 10$  et  $d = 5$ , solution tout à droite de l'arbre

complexité en temps de la recherche en largeur:

$$1 + b + b^2 + \dots + b^{d-2} + b^{d-1} + b^d + b^{d+1}-b$$

$$1 + 10 + 100 + 1000 + 10\,000 + 100\,000 + 999\,990 = 1\,111\,101 \text{ noeuds}$$

complexité en temps de la recherche par approfondissement successif:

$$(d+1)1 + (d)b + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

$$6 + 50 + 400 + 3000 + 20\,000 + 100\,000 = 132\,456 \text{ noeuds}$$

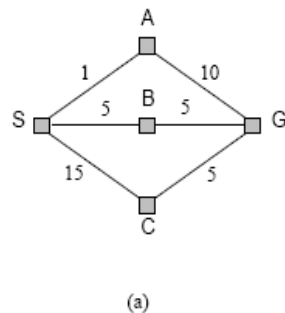
## Algorithme de recherche en coût uniforme

- La recherche en largeur trouve le noeud-solution le moins profond
- la recherche en coût uniforme étend systématiquement le noeud de coût le plus faible (mesuré par la fonction de coût  $g(n)$ )
- On suppose que chaque opérateur entre un nœud  $n'$  et son successeur  $n$  est muni d'un coût  $c(n',n)$  (valeur sur l'arc:  $n' \rightarrow n$ ). Le coût  $g(n)$  est la somme des  $c$  sur un chemin depuis le nœud initial.
- le coût d'un chemin ne doit jamais décroître
  - c-à-d qu'il ne doit pas y avoir de coûts partiels négatifs
  - $\Rightarrow g(\text{successeur}(n)) \geq g(n)$



## Algorithme de recherche en coût uniforme

S ●  
n



(b)

- (a) espace d'états avec coût pour chaque transition
- (b) progression de la recherche, chaque noeud est étiqueté avec le coût du chemin  $g(n)$ .

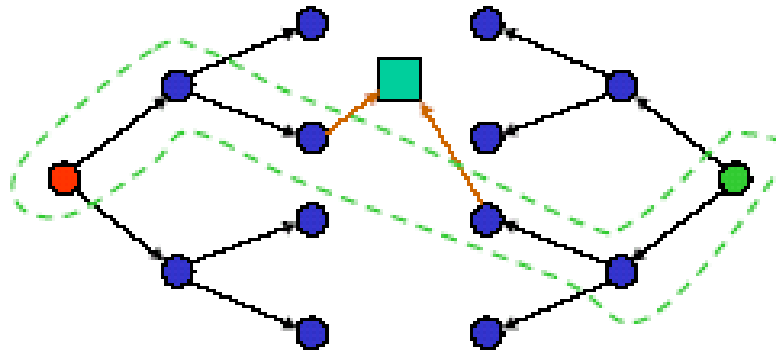
La solution sélectionnée sera celle avec un coût  $g = 10$

- Complétude Oui
- Complexité en temps  $O(b^d)$
- Complexité en espace  $O(b^d)$
- Optimalité Oui

Equivalent à Largeur si les coûts =1

# Algorithme de recherche bidirectionnelle

- Rechercher en largeur de l'état-solution à partir de l'état initial
- Recherche en largeur de l'état initial à partir de l'état-solution
- Arrêter la recherche lorsque les 2 processus se rencontrent



- Complétude Oui
- Complexité en temps  $O(b^{d/2}) \ll O(b^d)$
- Complexité en espace  $O(b^{d/2}) \ll O(b^d)$
- Optimalité Oui
- **nécessite:**
  - des états-solutions explicitement définis,
  - des opérateurs dont on connaît la fonction inverse  
(capable de générer l'état prédécesseur d'un état donné)

# Comparaison des algorithmes de recherche aveugles

- Comparaison de 6 algorithmes selon les 4 critères d'évaluation retenus

avec:  $b$  = facteur de branchement

$d$  = profondeur de la solution

$m$  = profondeur maximum de l'arbre de recherche


$l$  = limite de la profondeur de recherche

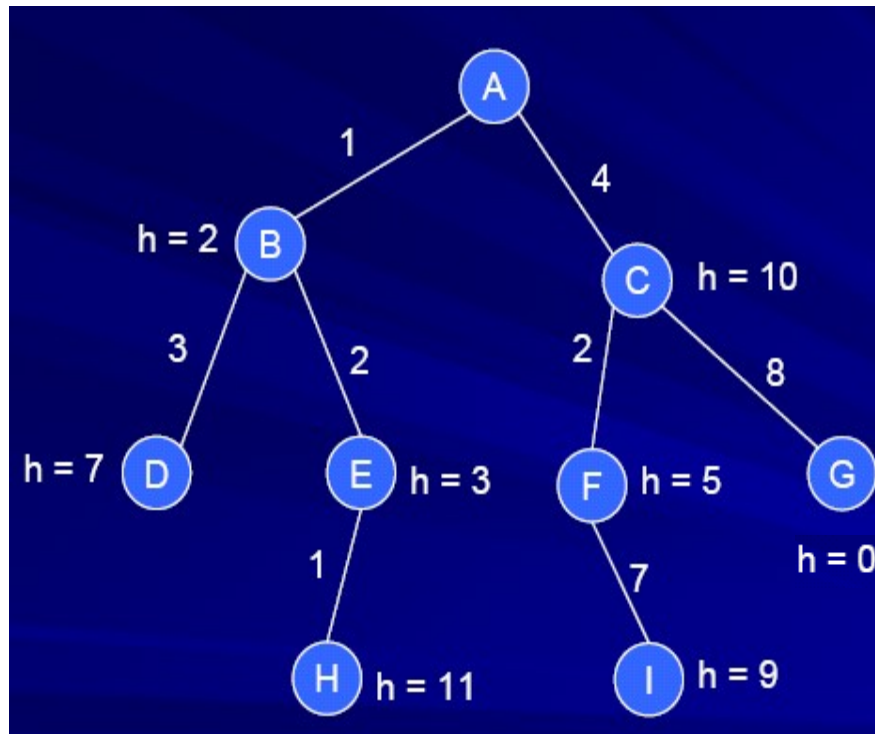
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	$b^d$	$b^d$	$b^m$	$b^l$	$b^d$	$b^{d/2}$
Space	$b^d$	$b^d$	$bm$	$bl$	$bd$	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

# Quelle recherche? quand?

- **RDF**: Préférée pour chercher un espace de recherche structuré en un arbre fini avec des noeuds finaux dans les feuilles de l'arbre.
- **RBF**: Préférée lorsque le “branching factor” est petit, les opérateurs ont une application coûteuse et les noeuds finaux sont attendus à une profondeur raisonnable.
- **COUT UNIFORME** : Lorsqu'on dispose d'une fonction coût.
- **RID**: Préférée pour chercher un espace de recherche structure en un arbre fini et si la profondeur de l'arbre est bien plus grande que la profondeur d'au moins un noeud final.

# Recherche dans les Graphes

- 2 méthodes:
  - **Méthode 1**: Faire la même chose que dans un arbre, mais garder une liste des noeuds déjà visités qui ne doivent pas être re-visités.  On peut s'y perdre!!!
  - **Méthode 2**: Retracer le graphe en un arbre en suivant tous les chemins possibles jusqu'à ce qu'ils ne puissent plus être allongés sans créer de boucle.

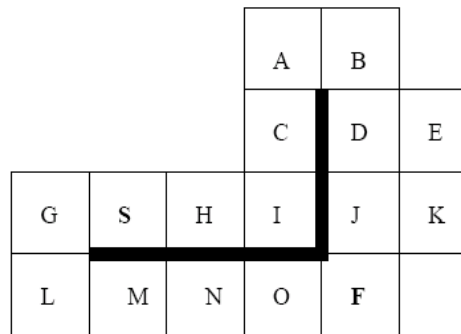
**Exercice 1**

- Largeur d'abord
- Coût uniforme
- Profondeur d'abord
- Profondeur limitée
- Profondeur itérative

Dans quel ordre les noeuds sont développés pour chacun des algorithmes?

## Exercice 2

Considérez le plan suivant où les successeurs de chaque case sont les cases adjacentes dans les directions Nord, Sud, Est et Ouest sauf à la limite du plan ou s'il y a une barrière (ligne épaisse). Par exemple  $\text{successeurs}(J) = \{K, F, D\}$ . On suppose que chaque opérateur ([Nord](#), [Sud](#), [Est](#), [Ouest](#)) à coût = 1.



Le problème est de trouver un chemin de S vers F. On suppose que les successeurs sont engendrés dans l'ordre Est, Sud, Ouest et Nord.

Donnez l'ordre des noeuds développés pour les 4 méthodes de recherche suivante:

1. Recherche en [largeur d'abord](#), [Coût Uniforme](#) [profondeur d'abord](#), et [Profondeur d'approfondissement itératif](#).

On suppose qu'on détecte des cycles, c.-à-d. on ne développe pas un noeud déjà développé avant.

## *Algorithmes de recherche heuristiques*



## Méthodes de recherche heuristiques

- Les algorithmes de recherche aveugle n'exploitent aucune information concernant la structure de l'arbre de recherche ou la présence potentielle de noeuds-solution pour optimiser la recherche.
- Recherche "rustique" à travers l'espace jusqu'à trouver une solution.
- La plupart des problèmes réels sont susceptibles de provoquer une explosion combinatoire du nombre d'états possibles.
- Un algorithme de recherche heuristique utilise l'information disponible pour rendre le processus de recherche plus efficace.
- Une information heuristique est une règle ou une méthode qui améliore, presque toujours, le processus de recherche.

## Fonction heuristique

- Une fonction heuristique (représentation)

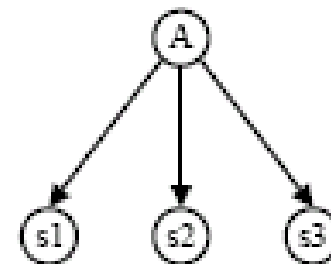
$h : E(\text{états}) \rightarrow \text{Réels}$

fait correspondre à un état  $s \in E$  (espace d'états) un nombre  $h(s) \in R$  qui est (généralement) une estimation du rapport coût/bénéfice qu'il y a à étendre le chemin courant en passant par  $s$ .

- Contrainte:  $h(\text{solution}) = 0$
- Le noeud A a 3 successeurs pour lesquels:

$$h(s1) = 0.8 \quad h(s2) = 2.0 \quad h(s3) = 1.6$$

- la poursuite de la recherche par  $s1$  est heuristiquement la meilleure



# Meilleur d'abord (Best First)

- Combinaison entre recherche en profondeur et en largeur
- en profondeur
  - avantage: solution trouvée sans avoir besoin de calculer tous les noeuds
- en largeur
  - avantage: ne risque pas de rester pris dans un "cul-de-sac"
- l'algorithme "Best-First search" permet d'explorer les noeuds dans l'ordre (croissant/décroissant) de leurs valeurs heuristiques

**Idée:** étendre le noeud le plus prometteur selon sa valeur heuristique

**Ordo** = insérer les successeurs en ordre (dé)croissant de leur valeur heuristique

- Cas particuliers de "best-first search"
  - greedy search (recherche gourmande ou gloutonne)
  - $A^*$

## ➤ Recherche gourmande (Greedy search)

- Stratégie la plus simple de « best-first search »
- fonction heuristique  $h(n)$  = estimation du coût du noeud  $n$  au but
- **greedy search** = minimiser le coût estimé pour atteindre le but
- le noeud qui semble être le plus proche du but sera étendu en priorité
- fonctions heuristiques classiques
  - distance à vol d'oiseau
  - distance "Manhattan": déplacements limités aux directions verticales et horizontales

**Complétude:** Non, elle peut être prise dans des cycles. Oui, si l'espace de recherche est fini avec vérification des états répétés.

**Complexité de temps:**  $O(b^m)$ , mais une bonne fonction heuristique peut améliorer grandement la situation.

**Complexité d'espace:**  $O(b^m)$ , elle retient tous les noeuds en mémoire.

**Optimale:** non, elle s'arrête à la première solution trouvée.

## ➤ Algorithme A\*

- **Greedy search** minimise le coût estimé  $h(n)$  du noeud  $n$  au but réduisant ainsi considérablement le coût de la recherche, mais il n'est pas optimal et pas complet
- l'algorithme de recherche en coût uniforme minimise le coût  $g(n)$  depuis l'état initial au noeud  $n$ , il est optimal et complet, mais pas très **Efficace**
- **idée:** combiner les deux algorithmes et minimiser le coût total  $f(n)$  du chemin passant par le noeud  $n$

$$f(n) = g(n) + h(n)$$

C'est l'algorithme A\*

## Exemple: 8\_puzzle

$g(n)$  = profondeur du noeud  $n$  dans l'arbre de recherche

$h1(n)$  = nombre de carrés mal placés dans ce nœud (sans le carré vide)

$h2(n)$  = somme des distances "manhattan" de chaque carré par rapport à sa destination finale  
[sans tenir compte des obstacles]

$g(n)$  est important puisque l'on cherche le chemin le plus court!

-  $f1(n) = g(n) + h1(n)$

-  $f2(n) = g(n) + h2(n) \equiv$  Meilleure estimation (vis a vis du nombre d'étapes avant le but)

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

$f1(n) = g(n) + h1(n) = 0 + 4 = 4$

$f2(n) = g(n) + h2(n) = 0 + 1 + 1 + 1 + 2 = 5$

Une solution optimale prend 5 mouvements

**$f2$  est en effet une meilleure estimation que  $f1$**

Dans la vie réelle on utilise aussi l'heuristique:

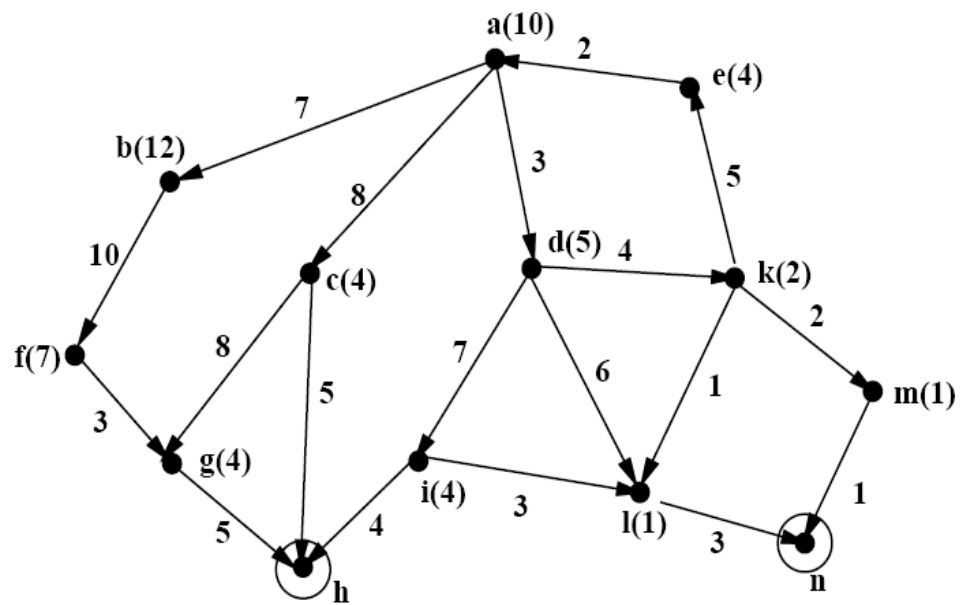
Exemple: Au supermarché, on choisit la queue la moins longue ou alors on choisit la queue dans laquelle les clients ont le plus petit nombre d'objets dans leur panier.

Avez-vous d'autres exemples?

## L'Algorithme A\*

1.  $Q \leftarrow$  noeud initial
2.  $C \leftarrow$  vide
3. **répéter**
4. **si**  $Q$  est vide, **retour** avec échec
5.  $n \leftarrow$  premier élément de  $Q$ ,  
     $Q \leftarrow$  reste( $Q$ )
6. **si**  $n$  est un noeud but,  
    **retour** avec succès (chemin)
7. **si**  $n \notin C$  ou a un coût inférieur (selon  $f$ ) **alors**
8. ajouter  $n$  à  $C$  (avec repointage)
9.  $S \leftarrow$  succ( $n$ )
10.  $Q \leftarrow$  **merge**( $Q, S$ ) ( concaténer  $Q$  et  $S$  )
11.  $Q \leftarrow$  **Ordo**( $Q, f$ )  
    ( $Q$  triée par ordre (dé)croissant de  $f(n) = g(n) + h(n)$ )
12. **fin**si
13. **fin**

# Exemple d'une Recherche Heuristique



L'Algorithme A\*

1.Q ← noeud initial

2.C ← vide

3.répéter

4. si Q est vide, retour avec échec

5. n ← premier élément de Q,  
Q ← reste(Q)

6. si n est un noeud but,  
retour avec succès (chemin)

7. si n ≡ C ou a un coût inférieur (selon f) alors

8. ajouter n à C (avec repointage)

9. S ← succ(n)

10. Q ← merge(Q,S) ( concaténer Q et S )

11. Q ← Ordo(Q,f)  
(Q triée par ordre (dé)croissant de f(n) = g(n) + h(n))

12. finsi

13.fin

Etat initial : a            Etat final : h ou n

Ordre de l'exploration ( l'heuristique est indiquée entre parenthèses sur le graphe):

- Q=(a(10))⇒C={a(10)}a
- Q=(d(8),c(12),b(19))⇒C={a(10),d(8)}a≡d
- Q=(k(9),l(10),c(12),i(14),b(19)) ⇒C={a(10),d(8),k(9)}a≡d≡k
- Q=(l(9),m(10),c(12),i(14),e(16),b(19)) ⇒C={a(10),d(8),k(9),l(9)}a≡d≡k≡l
- Q=(m(10),n(11),c(12),i(14),e(16),b(19)) ⇒C={a(10),d(8),k(9),l(9),m(10)}a≡d≡k≡(m,l)
- Q=(n(10),c(12),i(14),e(16),b(19)) ⇒C={a(10),d(8),k(9),l(9),m(10),n(10)}a≡d≡k≡m≡n
- solution!



### Admissibilité

- A\* utilise une heuristique **admissible**, c'est-à-dire  $h(n) \leq h^*(n)$ , où  $h^*(n)$  est le véritable coût pour se rendre de  $n$  au but.
- A\* Demande aussi que  $h(n) \geq 0$  et que  $h(G) = 0$  pour tous les buts  $G$ .

### Dominance

Si  $h_2(n) \geq h_1(n)$  pour tout  $n$  (les deux étant admissibles), alors  $h_2$  domine  $h_1$  et par conséquent  $h_2$  est meilleure que  $h_1$ .

Il est toujours préférable de choisir l'heuristique dominante, car elle va développer moins de noeuds

- **théorème:** si A\* utilise une fonction heuristique admissible, c-à-d qui ne surestime jamais le coût réel:

$\forall n \quad 0 \leq h(n) \leq h^*(n)$  avec  $h^*(n)$  = coût réel de  $n$  au but,  
alors **A\* est optimal**

une fonction heuristique admissible est toujours optimiste!

- **exemple:**  $vol\ d'oiseau(n)$  ne surestime jamais la distance réelle.

## Propriétés de A\*

**Complétude:** oui, à moins qu'il y est une infinité de noeuds avec  $f \leq f(but)$ .

**Complexité de temps:** exponentielle selon la longueur de la solution.

**Complexité en espace:** elle garde tous les noeuds en mémoire: exponentielle selon la longueur de la solution.

**Optimale:** Oui Habituellement, on manque d'espace longtemps avant de manquer de temps.

## Exemple: 8\_puzzle

- **Question:** Est-ce que la fonction  $h_1(n)$  du puzzle a 8 pièces est admissible?

- **Réponse:**

Oui, car  $h_1(n)$  est une sous-estimation de  $h^*(n)$  puisque les carrés qui ne sont pas à leur destination finale ont besoin d'au moins un mouvement, et probablement plus.

- **Question:** Est-ce que la fonction  $h_2(n)$  du puzzle a 8 pièces est admissible?

- **Réponse:**

Oui, car  $h_2(n)$  est une sous-estimation de  $h^*(n)$  puisque les carrés qui se trouvent à une distance "manhattan"  $d$  de leur destination finale ont besoin d'au moins  $d$  mouvements, et possiblement plus.

## ***Variantes de A\****

- les problèmes réels sont très complexes
- l'espace de recherche devient très grand
- même les méthodes de recherche heuristiques deviennent inefficaces
- A\* connaît alors des problèmes de place-mémoire
- algorithmes de recherche "économes" en place-mémoire?

2 variantes de A\*:

– IDA\* = A\* avec approfondissement itératif

qui est un algorithme A\* qui effectue des approfondissements successifs en rapport avec des valeurs limites pour  $f(n)$  ; IDA\* est complète et optimale et sa complexité en espace mémoire est  $O(b \times d)$  de l'ordre

– SMA\* = A\* avec gestion de mémoire

qui est un algorithme A\* qui effectue la gestion de sa propre mémoire disponible: élimine les nœuds ayant les valeurs de  $f(n)$  plus élevées quand la file ordonnée est pleine. L'algorithme SMA\* est complet si l'espace mémoire disponible est suffisant pour contenir le chemin: état initial – état solution.

L'algorithme SMA\* retourne toujours la meilleure solution qui peut être obtenue avec l'espace mémoire alloué.

## Qualité des fonctions heuristiques

- Facteur de branchement effectif

soit  $N$  = nombre total d'états produits pour obtenir la solution

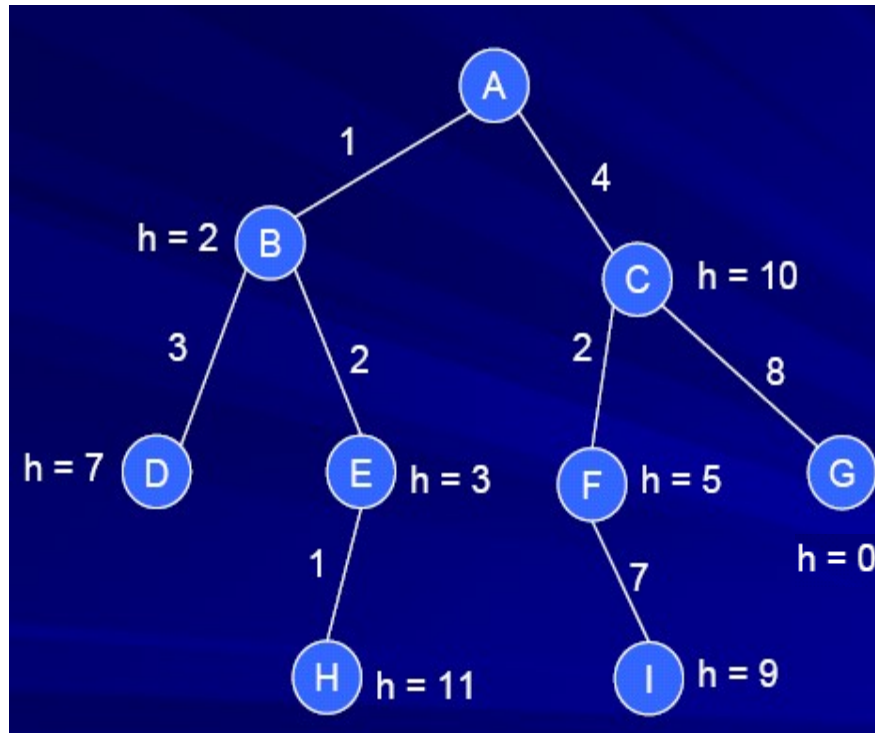
soit  $d$  = profondeur à laquelle la solution a été trouvée

alors  $b^*$  est le facteur de branchement d'un arbre fictif parfaitement équilibré tel que

$$N = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

exemple: si  $d = 5$  et  $N = 52 \rightarrow b^* = 1.91$

- une bonne fonction heuristique aura une valeur de  $b^*$  proche de 1

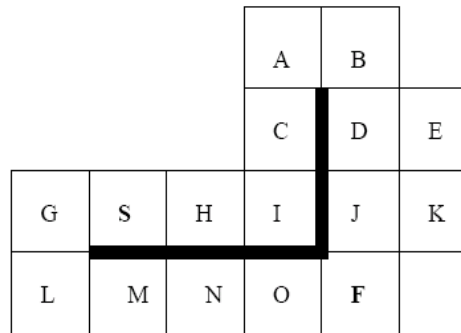
**Exercice 1 (suite)**

- Meilleur d'abord gourmande
- $A^*$

Dans quel ordre les noeuds sont développés pour chacun des algorithmes?

## Exercice 2

Considérez le plan suivant où les successeurs de chaque case sont les cases adjacentes dans les directions Nord, Sud, Est et Ouest sauf à la limite du plan ou s'il y a une barrière (ligne épaisse). Par exemple  $\text{successeurs}(J) = \{D, K, F\}$ . On suppose que chaque opérateur (N,S,E,O) à coût 1.

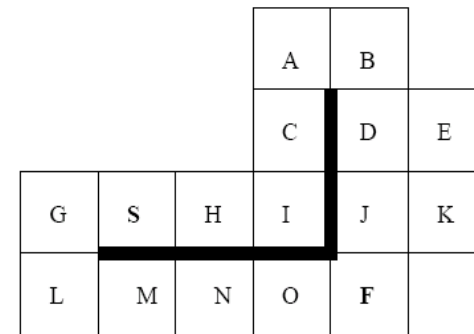


Le problème est de trouver un chemin de S vers F. On suppose que les successeurs sont engendrés dans l'ordre Est, Sud, Ouest et Nord.

Donnez l'ordre des noeuds développés pour les 4 méthodes de recherche suivante:

1. Recherche en [largeur d'abord](#), [Coût Uniforme profondeur d'abord](#), et [Profondeur itérative](#).

On suppose qu'on détecte des cycles, c.-à-d. on ne développe pas un noeud déjà développé avant.



## Exercice 2 (suite)

2. Recherche gourmande. On utilise l'heuristique  $h(\text{case}) = \text{distance de Manhattan de la case avec la case F}$  en supposant qu'il n'y a pas de barrière. Par exemple  $h(I) = 2$  et  $h(S) = 4$ . Si deux noeuds ont la même valeur, c'est d'abord celui qui est le premier dans l'alphabet qui est développé.

3. Recherche A\* avec même heuristique que pour la recherche gourmande.  
Si deux noeuds ont la même valeur, c'est d'abord celui qui est le premier dans l'alphabet qui est développé.

Questions diverses (Justifiez vos réponses):

Est-ce que  $h$  est admissible ?

Est-ce que  $h_2(n) = \min(2; h(n))$  est admissible ?

Est-ce que  $h_3(n) = \max(2; h(n))$  est admissible ?



## *Algorithmes d'optimisation*

# Algorithmes d'amélioration itérative

Dans plusieurs problèmes d'optimisation, le chemin n'est pas important, MAIS l'état **but** (qui est la solution).

Avec ce type d'algorithme, l'espace de recherche est l'ensemble des configurations et l'objectif est soit:

- de trouver la solution optimale
- ou de trouver une configuration qui satisfait les contraintes

On part donc avec une configuration complète et, par amélioration itérative, on essaie d'améliorer la qualité de la solution.

## Hill Climbing (ou descente du gradient)

- on se déplace constamment dans la direction de la valeur de pente la plus forte (croissante ou décroissante)
- aucun arbre de recherche n'est généré
- si il y a plus d'un état-successeur possible on choisit au hasard

**C'est comme escalader le mont Everest en plein brouillard en souffrant d'amnésie!**



## Hill Climbing (Algorithme)

**function** Hill-Climbing ( *problem* ) **returns** a solution state

**inputs** : *problem* ; a problem

**local variables** : *current* ; a node

*next* ; a node

*current*  $\leftarrow$  Make-Node ( Initial-State [ *problem* ] )

**loop do**

*next*  $\leftarrow$  a highest-valued successor of *current*

**if** Value [*next*] < Value [*current*] **then return** *current*

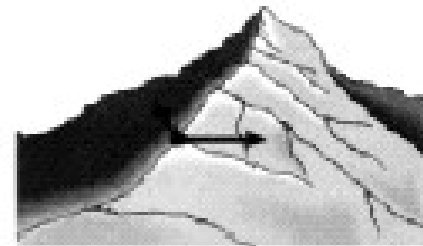
*current*  $\leftarrow$  *next*

**end**

# Hill Climbing

Cette méthode a 3 inconvénients:

- a) **maximum local** présence d'un maximum local par opposition à global
- b) **Plateau** région de l'espace de recherche où la fonction d'évaluation est plate, exigeant alors une marche aléatoire
- c) **Arrête** présence de pentes raides, la recherche progresse de côté et pas vers le sommet



## Amélioration : Recuit simulé

- Alternative au "hill climbing" aléatoire pour échapper à un extremum local est d'autoriser quelques "mauvais" déplacements, mais en *en réduisant graduellement la taille et la fréquence*
- **recuit** = processus physique de chauffage suivi d'un lent refroidissement pour obtenir une structure cristalline plus solide
- **recuit simulé** = processus qui baisse lentement sa "température " jusqu'à ce que le système se fige et que plus aucun changement ne soit observé

Autres.....

## Recuit simulé (Algorithme)

**function** Simulated-Annealing ( *problem*, *schedule* ) **returns** a solution state

**inputs** : *problem* ; a problem

*schedule* ; a mapping from time to "temperature"

**local variables** : *current* ; a node

*next* ; a node

*T* ; a "temperature" controlling the probability of downward steps

*current*  $\leftarrow$  Make-Node ( Initial-State [ *problem* ] )

**for** *t*  $\leftarrow$  1 **to**  $\infty$  **do**

*T*  $\leftarrow$  *schedule* [ *t* ]

**if** *T* = 0 **then return** *current*

*next*  $\leftarrow$  a randomly selected successor of *current*

$\Delta E \leftarrow$  Value [ *next* ] - Value [ *current* ]

**if**  $\Delta E > 0$  **then** *current*  $\leftarrow$  *next*

**Else** *current*  $\leftarrow$  *next* only with probability  $e^{\Delta E/T}$

## Propriétés du recuit simulé

- A température fixe  $T$ , la distribution de probabilité des états est celle de la distribution de Boltzman

$$p(x) = \alpha \cdot e^{-\frac{E(x)}{kT}}$$

- si  $T$  décroît suffisamment lentement on est assuré d'atteindre le meilleur état final
- développé par Metropolis et al., 1953, pour modéliser des processus physiques
- très utilisé en conception de circuits VLSI, planification et horaire de lignes aériennes, etc.



# Algorithmes des jeux

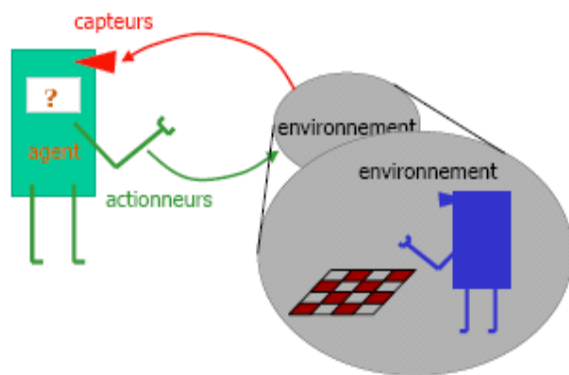
- Pourquoi étudier les jeux?
- Algorithme *MiniMax*
- *MiniMax* avec élagage  $\alpha$ - $\beta$



*AIMA chapitre 6*

- Domaine favori de l'IA
  - tâche bien structurée
  - défi intellectuel
  - abstraction
  - mesure de performance
  - ne nécessite pas (obligatoirement) de grandes quantités d'informations
- tous les jeux ne sont pas adaptés à une étude par l'IA, on se limitera aux jeux parfaits de plateau à 2 joueurs

- 2 joueurs: MAX et MIN jouant à tour de rôle (MAX joue en premier)
- Espace d'états
- État initial
- Fonction « successeur » (règles de jeu)
- Test de fin de partie
- Fonction « score » indique si un état terminal est un état de gain (pour MAX), de perte ou de nul
- Connaissance parfaite des états, pas d'incertitude dans l'effet des fonctions « successeur »



Agents « joueurs »

Caractéristiques:

- Présence d'un adversaire
  - introduit un élément d'incertitude
  - tous les mouvements ne sont pas contrôlés par l'ordinateur
- Les programmes de jeu doivent faire face à l'imprévu
- Complexité: les jeux intéressants sont trop complexes pour envisager une solution exhaustive
  - exemple: les échecs ont un facteur de branchement de  $\sim 35$
- Le but à chaque pas est de choisir le prochain (meilleur) coup à jouer
- Optimalité: les coups doivent être évalués selon leur coût

## Éléments d'un système informatique de jeu

Composants de base d'un "moteur" de jeu:

- Générateur de mouvements
  - génère les mouvements valides à partir de l'état courant
- Test de terminaison
  - décide si l'état courant est un gain, une perte, un nul ou rien de particulier
- Fonction d'évaluation (fonction de gain ou d'utilité)
  - évalue la qualité d'un état donné indépendamment des coups passés ou futurs
- Stratégie de contrôle
  - fournit les éléments nécessaires pour choisir entre différentes options celle qui semble la plus prometteuse

## TERMINOLOGIE:

**Arbre de Jeu:** arbre qui consiste en des noeuds qui représentent les options des joueurs dans un jeu à deux joueurs. Les noeuds de cet arbre alternent entre les options des deux joueurs.

**Tours/Couches:** Les différents niveaux dans un arbre de jeux s'appellent les tours ou les couches.

**Noeuds Finaux dans un Arbre de Jeu:** Les noeuds terminaux indiquent la fin du jeu, c'est à dire une configuration gagnante ou perdante (+1 ou -1) pour le joueur dont on a pris le point de vue. S'il y a **match nul**, la valeur est de 0.

**Maximiser:** C'est le joueur dont le but est d'atteindre un noeud final de valeur +1.

**Minimiser:** C'est le joueur dont le but est d'atteindre un noeud final de valeur -1.

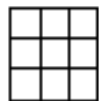
**Match nul:** Un joueur doit préférer une victoire plutôt qu'un match nul, et un match nul plutôt qu'une défaite.

## Fonction d'évaluation pour un état du jeu

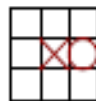
- $e(s) = +\infty$  si  $s$  est une situation de gain pour Max
- $e(s) = -\infty$  si  $s$  est une situation de gain pour Min
- $e(s)$  = une mesure du caractère "favorable" de  $s$  pour Max
  - > 0 si  $s$  est considéré favorable pour Max
  - < 0 si non

- exemple: Tic-Tac-Toe (Morpion)

$e(s) = (\text{\# lignes/cols/diags ouvertes pour Max}) - (\text{\# lignes/cols/diags ouvertes pour Min})$



$$8-8 = 0$$



$$6-4 = 2$$



$$3-3 = 0$$

*Il faut disposer d'une fonction d'évaluation statique capable de mesurer la qualité d'une configuration par rapport à un joueur (gén.Max)*

*– car il n'est pas possible de produire tout l'arbre de recherche jusqu'à la fin du jeu, càd au moment où la décision gain/perte/nul est claire.*

**Principe:** maximiser la valeur d'utilité pour Max avec l'hypothèse que Min joue parfaitement pour la minimiser,

- étendre l'arbre de jeu
- calculer la valeur de la fonction de gain pour chaque noeud terminal
- propager ces valeurs aux noeuds non-terminaux:
  - la valeur minimum (adversaire) aux noeuds MIN
  - la valeur maximum (joueur) aux noeuds MAX

### Formulation récursive de MiniMax

- Fonction MiniMax (noeud, profondeur)
  - si profondeur = 0, retourner évaluation(noeud)
  - si c'est à MAX de jouer à ce noeud, retourner max MiniMax(n, profondeur - 1)
  - si c'est à MIN de jouer à ce noeud, retourner min MiniMax(n, profondeur - 1)
- ceci est calculé de manière "en profondeur" par l'utilisation d'une pile des appels récursifs.

# Algorithme MiniMax

**function** Minimax-Decision ( *game* ) **returns** an operator

**for each** *op* **in** Operators [ *game* ] **do**

    Value [ *op* ]  $\leftarrow$  Minimax-Value ( Apply ( *op* , *game* ), *game* )

**end**

**return** the *op* with the highest Value [ *op* ]

**function** Minimax-Value ( *state*, *game* ) **returns** a utility value

**if** Terminal-Test [ *game* ]( *state* ) **then**

**return** Utility [ *game* ]( *state* )

**else if** max is to move in *state* **then**

**return** the highest Minimax-Value of Successors ( *state* )

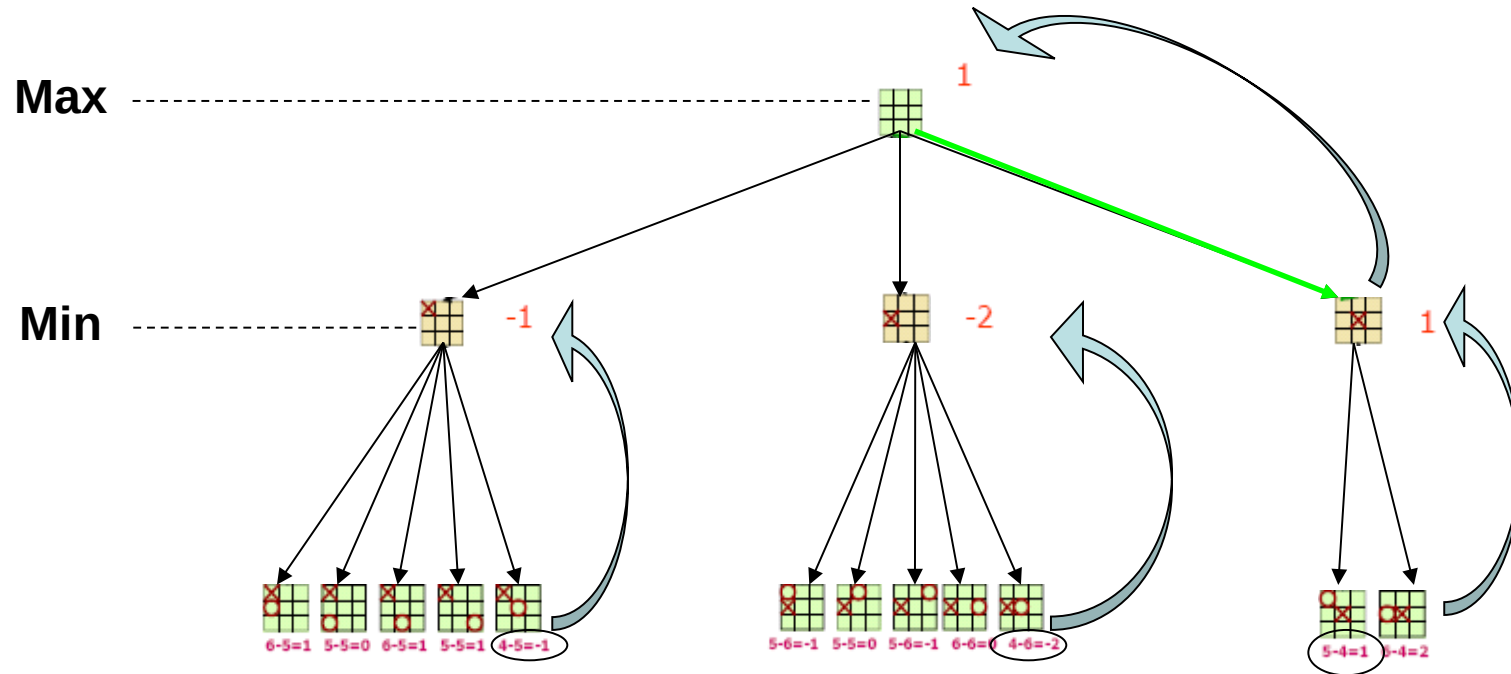
**else**

**return** the lowest Minimax-Value of Successors ( *state* )



## Exemple: Tic-Tac-Toe

$e(s) = (\# \text{ lignes/cols/diags ouvertes pour Max}) - (\# \text{ lignes/cols/diags ouvertes pour Min})$



## Réduction de l'espace

- Complet Oui si l'arbre de jeu est fini
- Optimal Oui si l'adversaire est aussi optimal
- Complexité en temps  $O(b^m)$  où  $b$  est le facteur de branchement et  $m$  est l'horizon de recherche
- Complexité en place  $O(bm)$  (car exploration en profondeur)

pour les échecs: avec  $b = 35$  et  $m = 100$

l'algorithme *MiniMax* est totalement impraticable !!!

### 2 solutions:

– *MiniMax* avec profondeur limitée

– élagage  $\alpha$ - $\beta$

### MiniMax avec profondeur limitée

1. Étendre l'arbre de jeu à partir de l'état courant (où c'est à Max de jouer) jusqu'à une profondeur  $h$  (Horizon de la procédure),
2. Calculer la fonction d'évaluation pour chacune des feuilles de l'arbre,
3. Rétro-propager les valeurs des noeuds-feuilles vers la racine de l'arbre de la manière suivante:
  1. Un noeud Max reçoit la valeur maximum de l'évaluation de ses successeurs,
  2. Un noeud Min reçoit la valeur minimum de l'évaluation de ses successeurs,
4. Choisir le mouvement vers le noeud Min qui possède la valeur rétropropagée la plus élevée.

- jeu d'échecs

si  $bm = 106$  et  $b = 35 \rightarrow m = 4$  niveaux de prévision

– prévision de 4 niveaux:: joueur novice

– prévision de 8 niveaux:: joueur niveau "maître" et bon programme PC

– prévision de 12 niveaux: G. Kasparov et Deep-Blue

## Réduction de l'espace

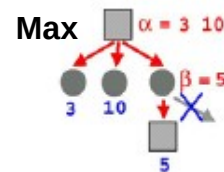
### Élagage $\alpha$ - $\beta$

#### Principe

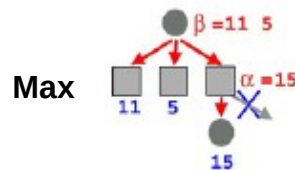
- étendre l'arbre de jeu jusqu'à une profondeur  $h$  par recherche en profondeur
- ne plus générer les successeurs d'un noeud dès qu'il est évident que ce noeud ne sera pas choisi (compte tenu des noeuds déjà examinés)
- chaque noeud Max garde la trace d'une  $\alpha$ -valeur = valeur de son meilleur successeur trouvé jusqu'ici
- chaque noeud Min garde la trace d'une  $\beta$ -valeur = valeur de son plus mauvais successeur trouvé jusqu'ici
- valeurs initiales:  $\alpha = -\infty$   $\beta = +\infty$

#### 2 règles

1. Interrompre la recherche d'un noeud Max si son  $\alpha$ -valeur  $\geq \beta$ -valeur de son noeud-parent



2. Interrompre la recherche d'un noeud Min si sa  $\beta$ -valeur  $\leq \alpha$ -valeur de son noeud-parent



Algorithme  $\alpha - \beta$ 

**function** Max-Value ( *state*, *game*,  $\alpha$  ,  $\beta$  ) **returns** the minimax value of *state*

**inputs** : *state* , current state in game

*game* , game description

$\alpha$ , the best score for max along the path to *state*

$\beta$ , the best score for min along the path to *state*

**if** Cutoff-Test ( *state* ) **then return** Eval ( *state* )

**for each** *s* **in** Successors ( *state* ) **do**

$\alpha \leftarrow \text{Max} ( \alpha, \text{Min-Value} ( s, \text{game}, \alpha , \beta ) )$

**if**  $\alpha \geq \beta$  **then return**  $\beta$

**end**  $\alpha$

**return**

**function** Min-Value ( *state*, *game*,  $\alpha$  ,  $\beta$  ) **returns** the minimax value of *state*

**if** Cutoff-Test ( *state* ) **then return** Eval ( *state* )

**for each** *s* **in** Successors ( *state* ) **do**

$\beta \leftarrow \text{Min} ( \beta, \text{Max-Value} ( s, \text{game}, \alpha , \beta ) )$

**if**  $\beta \leq \alpha$  **then return**  $\alpha$

**end**

**return**

## Algorithme $\alpha - \beta$ (version française)

```
Fonction alphabeta_decision(état) Renvoie action  
  v <- max_valeur(état, -INF, +INF)  
  Renvoyer l'action de successeur(état) dont le minimax est v.
```

---

```
Fonction max_valeur(état, alpha, beta) Renvoie minimax  
  Si état_terminal(état) Alors Renvoyer utilité(état)  
  v <- -INF  
  Pour chaque état s dans successeur(état)  
    v <- Max(v, min_valeur(s, alpha, beta))  
    Si v >= beta  
      Alors Renvoyer v  
      Sinon alpha <- Max(alpha, v)  
  Renvoyer v
```

---

```
Fonction min_valeur(état, alpha, beta) Renvoie minimax  
  Si état_terminal(état) alors renvoyer utilité(état)  
  v <- +INF  
  Pour chaque état s dans successeur(état)  
    v <- Min(v, max_valeur(s, alpha, beta))  
    Si v <= alpha  
      Alors Renvoyer v  
      Sinon beta <- Min(beta, v)  
  Renvoyer v
```

## Propriétés de $\alpha - \beta$

- L'élagage n'affecte pas le résultat final,
- l'efficacité de l'élagage  $\alpha - \beta$  est fonction de l'ordre d'apparition des noeuds successeurs,

### complexité en temps

- meilleur des cas:  $O(b^{m/2})$ 
  - permet de doubler la profondeur de recherche pour atteindre une prédiction sur 8 niveaux et ainsi jouer "dignement" aux échecs
- pire des cas: identique à MiniMax
- cas moyen:  $O((b/\log b)^m)$  [Knuth&Moore 75]

## Variantes de Alpha-Beta

- Approfondissements itératifs
- Étendre les noeuds singuliers (ceux qui présentent un intérêt particulier) en priorité

# Applications

- 1952: Samuel développe le 1er programme qui apprend sa propre fonction d'évaluation au cours du temps.
  - 1992: Chinook (J. Schaeffer) gagne le championnat de USA. Ce programme utilise l'algorithme d'élagage  $\alpha - \beta$ .
  - 1994: Chinook met fin à 40 ans de règne du champion du monde Marion Tinsley. Il utilisait une base de données de fin de parties définissant le jeu parfait pour toutes positions incluant 8 pièces ou moins, soit au total 443'748'401'247 positions.
- 
- 1997: "Deep Blue" bat G. Kasparov. Il effectue une recherche sur 14 niveaux et explore plus de 1 milliard de combinaisons par coup à raison de plus de 200 millions de positions par seconde, utilise une fonction d'évaluation très sophistiquée et des méthodes non divulguées pour étendre certaines recherches à plus de 40 niveaux. Force des programmes de jeu d'échecs mesurée selon l'échelle de la "US Chess Federation". Gary Kasparov est classé à 2600.



Nom: Marion Tinsley  
Profession: professeur de mathématiques  
Hobby: le jeu de dames  
Record: en plus de 42 ans n'a perdu que 3 (!) parties

Nom: Chinook  
Record: premier ordinateur ayant gagné un championnat du monde face à un humain!



- Septembre 2002: match entre V. Kramnik et Deep Fritz:  
<http://www.brainsinbahrain.com/>



- Décembre 2002: match entre G. Kasparov (champion de son monde) et Deep Junior (champion de son monde).

Kasparov	<u>Nom</u>	Deep Blue
1 m 83	Taille	2 m 20
77 kg	Poids	900 kg
34 ans	Age	4 ans
50 milliards neurones	"Ordinateur"	512 processeurs
2 pos/sec	Vitesse	200,000,000 pos/sec
Extensive	Connaissance	Primitive
Electrique/chimique	Source d'énergie	Electrique

## Autres types de jeux

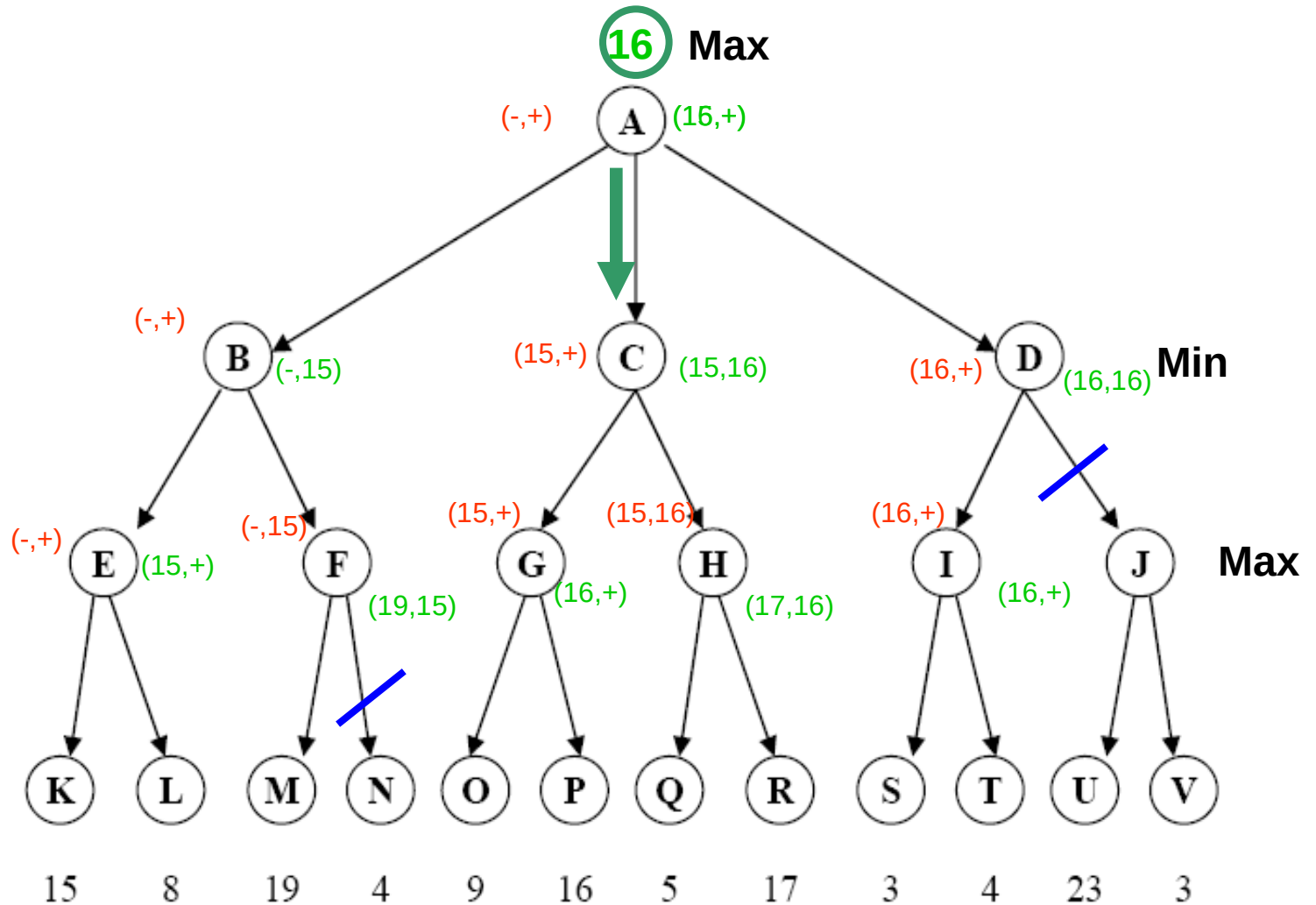
- Jeux à plusieurs joueurs ( $> 2$ ), avec ou sans alliance
- Jeux avec intervention du hasard dans la fonction « successeur » (ex lancer un dé)
- États partiellement connus (ex jeux de cartes)



# Exercice : jeux

1- MiniMax

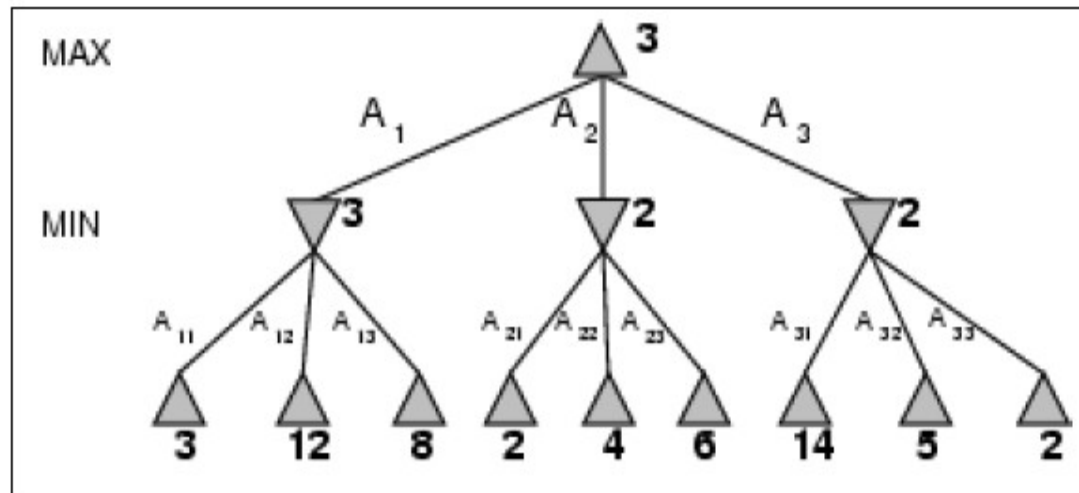
2- Alpha-Beta



- A : [-INF,+INF]
  - B : [-INF,+INF]
    - E : [-INF,+INF]
      - K : 15
    - E : [15,+INF]
      - L : 8
    - E : [15,+INF]
  - B : [-INF,15]
    - F : [-INF,15]
      - M : 19
    - F : v=19
  - B : [-INF,15]
- A : [15,+INF]
  - C : [15,+INF]
    - G : [15,+INF]
      - O : 9
    - G : [15,+INF]
      - P : 16
    - G : [16,+INF]

- C : [15,16]
  - H : [15,16]
    - Q : 5
  - H : v=5
    - R : 17
  - H : v=17
- C : [15,16]
- A : [16,+INF]
  - D : [16,+INF]
    - I : [16,+INF]
      - S : 3
    - I : v=3
      - T : 4
    - I : v=4
  - D : v=4
- A : [16,+INF] , v=16

## Exercise



On indique les valeurs  $[\alpha, \beta]$  pour chaque nœud.

1) **racine** :  $[-\text{INF} ; +\text{INF}]$

a. **a<sub>1</sub>** :  $[-\text{INF} ; +\text{INF}]$

i. **a<sub>11</sub>** : 3

b. **a<sub>1</sub>** :  $v \leftarrow 3$  (car  $3 < +\text{INF}$ ) ;  $\beta \leftarrow 3$  (car  $3 < +\text{INF}$ )  $[-\text{INF} ; 3]$

i. **a<sub>12</sub>** : 12

c. **a<sub>1</sub>** :  $[-\text{INF} ; 3]$

i. **a<sub>13</sub>** : 8

d. **a<sub>1</sub>** :  $[-\text{INF} ; 3]$

2) **racine** :  $v \leftarrow 3$  (car  $3 > -\text{INF}$ ) ;  $\alpha \leftarrow 3$  (car  $3 > -\text{INF}$ )  $[3 ; +\text{INF}]$

a. **a<sub>2</sub>** :  $[3 ; +\text{INF}]$

i. **a<sub>21</sub>** : 2

b. **a<sub>2</sub>** :  $v \leftarrow 2$  (car  $2 < +\text{INF}$ ) ; et  $v < \alpha$  donc on remonte

3) **racine** :  $3 > 2$  donc  $v$  ne change pas

a. **a<sub>3</sub>** :  $[3 ; +\text{INF}]$

i. **a<sub>31</sub>** : 14

b. **a<sub>3</sub>** :  $v \leftarrow 14$  (car  $14 < +\text{INF}$ ) ;  $\beta \leftarrow 14$  (car  $14 < +\text{INF}$ )  $[3 ; 14]$

i. **a<sub>32</sub>** : 5

c. **a<sub>3</sub>** :  $v \leftarrow 5$  (car  $5 < 14$ ) ;  $\beta \leftarrow 5$  (car  $5 < 14$ )  $[3 ; 5]$

i. **a<sub>32</sub>** : 2

d. **a<sub>3</sub>** :  $v \leftarrow 2$  (car  $2 < 5$ ) ; et  $v < \alpha$  donc on remonte

4) **racine** :  $3 > 2$  donc  $v$  ne change pas

5) **racine** :  $v = 3$

# 7.5 Extensions

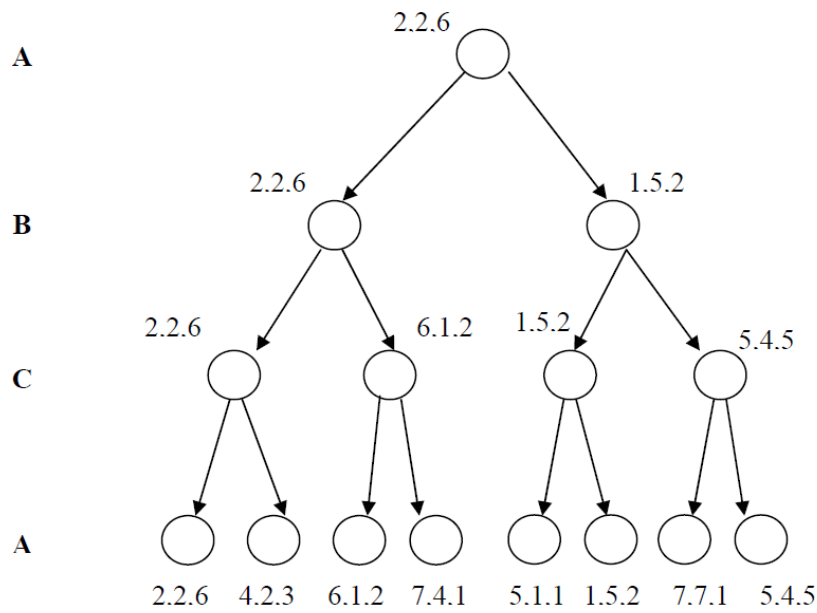
## 7.5.1 Plus de deux joueurs

On peut traiter un jeu à plus de deux joueurs avec un algorithme classique de type minimax à condition d'effectuer quelques modifications.

- au lieu de manipuler une valeur  $v$ , on va manipuler un **vecteur de valeurs**  $v_1, \dots, v_n$  (une par joueur). Remarque : à deux joueurs on ne représentait qu'une valeur car le jeu étant à somme nulle, l'une dépendait de l'autre. Maintenant, ce n'est plus le cas.

- A chaque noeud, on fait remonter le vecteur de valeurs le plus favorable au joueur qui doit jouer à ce moment là. Donc, le vecteur où ce joueur a un  $v_i$  maximal.

*Exemple : jeu à 3 joueurs*



## Algorithmes génétiques

*Quel est le dispositif de résolution de problèmes le plus puissant de l'univers ?*

**L' électricité** : qui permet de faire fonctionner les ordinateurs, les ampoules, etc.

*ONE.*

**L'amour** : qui est la solution à tous les maux

*Aphrodite.*

**Le cerveau humain** : qui inventa les précédents

*Descartes.*

**Le mécanisme de l' évolution** : qui créa le cerveau humain

*Darwin.*

## Un peu d'histoire :

Au siècle dernier, Charles Darwin observa les phénomènes naturels et fit les constatations suivantes :

- l'évolution n'agit pas directement sur les êtres vivants ; elle opère en réalité sur les chromosomes contenus dans leur ADN.
- l'évolution a deux composantes : **la sélection et la reproduction.**
  - **la sélection** garantit une reproduction plus fréquente des chromosomes les plus forts.
  - **la reproduction** est la phase durant laquelle s'effectue l'évolution.

## Un peu d'histoire :

Dans les années 60s, John H. Holland expliqua comment **ajouter de l'intelligence dans un programme informatique avec les croisements (échangeant le matériel génétique) et la mutation (source de la diversité génétique).**

Il formalisa ensuite les principes fondamentaux des algorithmes génétiques :

- **la capacité de représentations** élémentaires, comme les chaînes de bits, à coder des structures complexes.
- **le pouvoir de transformations** élémentaires à améliorer de telles structures.

Et récemment, David E. Goldberg ajouta à la théorie des algorithmes génétiques les idées suivantes :

- un individu est lié à un environnement par son code d'ADN.
- une solution est liée à un problème par son indice de qualité (adaptation).



## Principe:

Les algorithmes génétiques (AGs) sont des **algorithmes d'optimisation stochastique** fondés sur les mécanismes de la sélection naturelle et de la **génétique**: (**Appliquer à une population des transformations afin de produire une population mieux adaptée**).

## Applications :

- Les applications des AG sont multiples : optimisation de fonctions numériques difficiles (discontinues...), traitement d'image (alignement de photos satellites, reconnaissance de suspects...), optimisation d'emplois du temps, optimisation de design, contrôle de systèmes industriels, apprentissage des réseaux de neurones, etc.
- Les AG peuvent être utilisés pour contrôler un système évoluant dans le temps (chaîne de production, centrale nucléaire...) car la population peut s'adapter à des conditions changeantes. Ils peuvent aussi servir à déterminer la configuration d'énergie minimale d'une molécule.
- Les AG sont également utilisés pour optimiser des réseaux (câbles, fibres optiques, mais aussi eau, gaz...), des antennes... Ils peuvent être utilisés pour trouver les paramètres d'un modèle petit-signal à partir des mesures expérimentales.
- Très utilisés en Economie: Prévision, Décision, Apprentissage,...

# Terminologie:

Selon Lerman et Ngouenet (1995) un algorithme génétique est défini par:

**Individu/chromosome/séquence:** une solution potentielle du problème;

**Population:** un ensemble de chromosomes ou de points de l'espace de recherche;

**Environnement:** l'espace de recherche;

**Fonction de fitness (adaptation):** la fonction - positive - à optimiser.

# Démarche:

- **But poursuivi :** approcher le maximum (par exemple) d'une fonction.
- **Base:**
  - Les points du domaine de définition forment une population d'individus.
  - La valeur de la fonction en un point mesure l'adaptation (*fitness*) de l'individu (sous-entendu, aux conditions du milieu ambiant - pour ne pas dire, du marché...).
  - Les points où le maximum est atteint sont les individus les mieux adaptés (*fittest*).
- **Hypothèse évolutionniste (Darwinisme élémentaire) :**
  - Les individus les mieux adaptés sont produits par l'évolution de la population (*survival of the fittest*)
  - Cette évolution est produite par **croisements** et **mutations**.

**ALGOTITHME (PRINCIPE):**

- initialiser le temps
- créer une population initiale
- évaluer l'adaptation de chaque individu

**tant que** (il n'y a pas de solution satisfaisante) **et** (le temps est inférieur au temps limite)

**faire :**

- incrémenter le temps
- sélectionner les parents
- calculer les gènes des nouveau-nés par recombinaison des parents
- faire subir des mutations aléatoires à la population
- évaluer l'adaptation de chaque individu
- sélectionner les survivants

- Concevoir une représentation.
- Comment initialiser une population.
- Faire correspondre un **phénotype** à un **génotype** (**CODAGE**) :
  - génotype** = le matériel génétique propre à un individu,
  - phénotype** = les caractères apparents d'un individu.
- Evaluer un individu par le biais d'une fonction de fitness.
- Définir les opérateurs de mutation, de recombinaison.
- Décider comment gérer la population.
- Définir le processus de sélection des parents.
- Décider des règles de remplacement des individus.
- Définir les conditions d'arrêt de l'algorithme.

Pré-requis

Le Codage

Trois principaux type de codage: **Binaire**, **Gray** et **Réel**.  
*Parallèle avec la biologie (codage binaire): on parle de **génotype** en ce qui concerne la **représentation binaire** d'un individu, et de **phénotype** pour ce qui est de **sa valeur réelle correspondante** dans l'espace de recherche.*

Codage des variables

La première étape est de définir et de coder convenablement le problème. A chaque variable d'optimisation  $x_i$  (à chaque paramètre du dispositif), nous faisons correspondre un **gène**.  
Un **chromosome** un ensemble de gènes. Chaque dispositif est représenté par un **individu** doté d'un génotype constitué d'un ou plusieurs chromosomes. Une **population** un ensemble de  $N$  individus que nous allons faire évoluer.

Sur le plan informatique, on utilise généralement un codage **binaire**. C'est-à-dire qu'un gène est un entier long (32 bits). Un chromosome est un tableau de gènes (figure 2 et figure 3). Un individu est un tableau de chromosomes. La population est un tableau d'individus. Notons qu'on pourrait aussi utiliser d'autres formes de codage (réel, codage de Gray...).

On aboutit à une structure présentant cinq niveaux d'organisation (figure 1), d'où résulte le comportement complexe des AG :

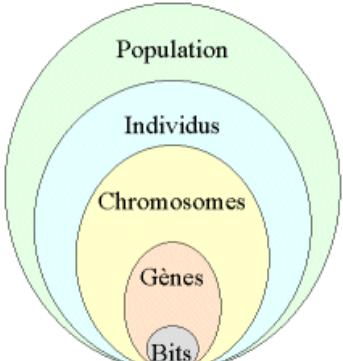


Figure 1

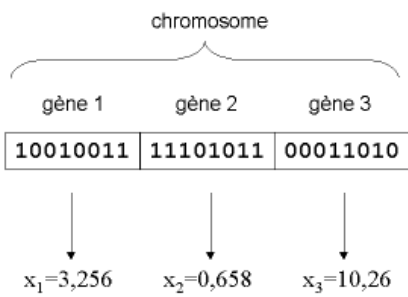


Figure 2

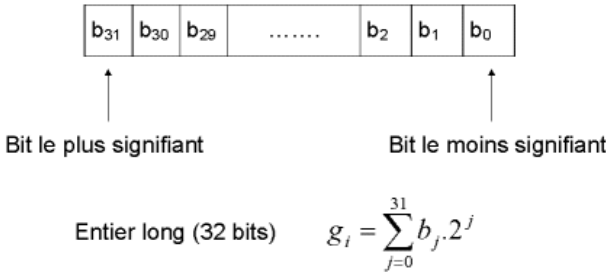


Figure 3

Un des avantages du codage binaire est que l'on peut ainsi facilement coder toutes sortes d'objets : des réels, des entiers, des valeurs booléennes, des chaînes de caractères... Cela nécessite simplement l'usage de fonctions de codage et décodage pour passer d'une représentation à l'autre.

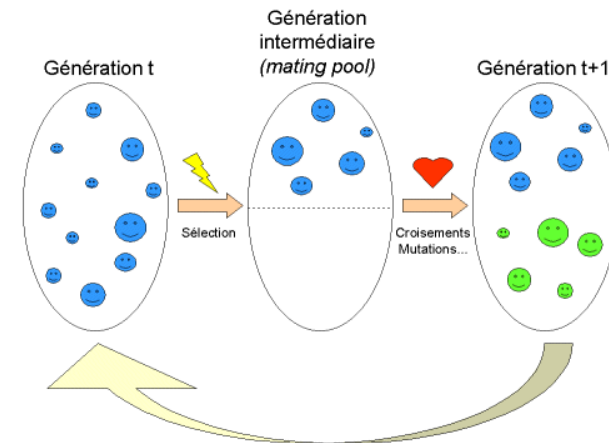
# OPERATEURS

## Sélection:

**But :** Permettre aux individus d'une population de survivre, de se reproduire ou de mourir.

**Règle général :** La probabilité de survie d'un individu sera directement liée à son efficacité relative au sein de la population.

Représentation schématique du fonctionnement de l'AG



Il y a plusieurs méthodes de sélection, citons quelques-unes :

- **Roulette de casino** : C'est la sélection naturelle la plus employée pour l'AG binaire. Chaque individu occupe un secteur de roulette dont l'angle est proportionnel à son indice de qualité (fitness). Un individu est considéré comme bon aura un indice de qualité élevé, un large secteur de roulette et alors il aura plus de chance d'être sélectionné.
- **N/2 –élitisme** : Les individus sont triés selon leur fonction d'adaptation, seul la moitié supérieure de la population correspondant aux meilleurs composants est sélectionnée. Il est important de maintenir une diversité de gènes pour les utiliser dans la population suivante et avoir des populations nouvelles quand on les combine.
- **Par tournoi** : Choisir aléatoirement deux individus et comparer leur fonction d'adaptation (combattre). On accepte le plus apte pour accéder à la génération intermédiaire, et on répète cette opération jusqu'à remplir la génération intermédiaire (N/2 composants).

# OPERATEURS

## Croisement :

Le phénomène de croisement est une propriété naturelle de l'ADN. On pratique le croisement dans les AG d'une manière analogique.

### a - croisement binaire :

**a-1 croisement en un point** : on choisit au hasard un point de croisement, pour chaque couple (fig. 1). Notons que le croisement s'effectue directement au niveau binaire, et non pas au niveau des gènes. Un chromosome peut donc être coupé au milieu d'un gène.

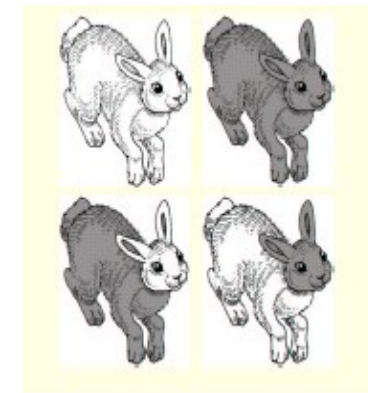
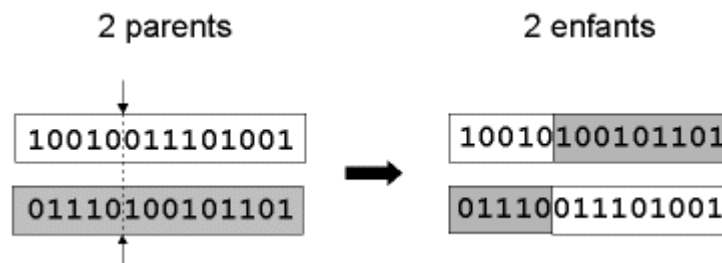


Fig. 1 : représentation schématique du croisement en 1 point. Les chromosomes sont bien sûr généralement beaucoup plus longs.

**a-2 croisement en deux points** : On choisit au hasard deux points de croisement (Fig. 2). Par la suite, nous avons utilisé cet opérateur car il est généralement considéré comme plus efficace que le précédent [Beasley, 1993b]. Néanmoins nous n'avons pas constaté de différence notable dans la convergence de l'algorithme.

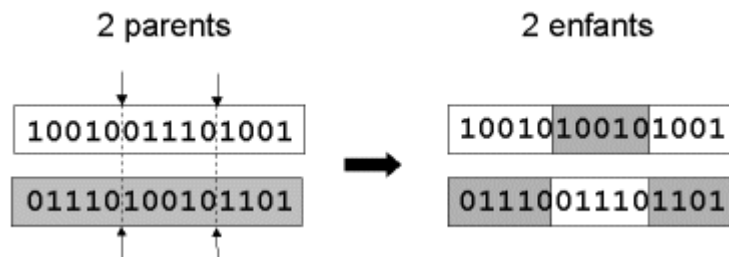


Fig. 2 : représentation schématique du croisement en 2 points.

Notons que d'autres formes de croisement existent, du croisement en  $k$  points jusqu'au cas limite du croisement uniforme.

## **b - Croisement réel :**

Le croisement réel ne se différencie du croisement binaire que par la nature des éléments qu'il altère : ce ne sont plus des bits qui sont échangés à droite du point de croisement, mais des variables réelles.

## **c - Croisement arithmétique :**

Le croisement arithmétique est propre à la représentation réelle. Il s'applique à une paire de chromosomes et se résume à une moyenne pondérée des variables des deux parents.

Soient  $[a_i, b_i, c_i]$  et  $[a_j, b_j, c_j]$  deux parents, et  $p$  un poids appartenant à l'intervalle  $[0, 1]$ , alors les enfants sont  $[p a_i + (1-p) a_j, p b_i + (1-p) b_j, p c_i + (1-p) c_j] \dots$

Si nous considérons que  $p$  est un pourcentage, et que  $i$  et  $j$  sont nos deux parents, alors l'enfant  $i$  est constitué à  $p\%$  du parent  $i$  et à  $(100-p)\%$  du parent  $j$ , et réciproquement pour l'enfant  $j$ .

# OPERATEURS

## Mutation :

Nous définissons une *mutation* comme étant l'inversion d'un bit dans un chromosome (Fig. 3 ). Cela revient à modifier aléatoirement la valeur d'un paramètre du dispositif. Les mutations jouent le rôle de bruit et empêchent l'évolution de se figer. Elles permettent d'assurer une recherche aussi bien globale que locale, selon le poids et le nombre des bits mutés. De plus, elles garantissent mathématiquement que l'optimum global peut être atteint.

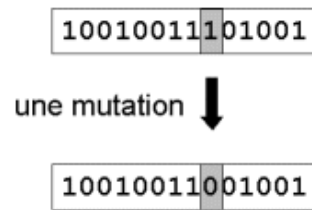


Fig. 3 : représentation schématique d'une mutation dans un chromosome.

Comment réaliser un opérateur mutation ?

De nombreuses méthodes existent. Souvent la probabilité de mutation  $p_m$  par bit et par génération est fixée entre 0,001 et 0,01. On peut prendre également  $p_m = 1/s$  où  $s$  est le nombre de bits composant un chromosome. Il est possible d'associer une probabilité différente de chaque gène. Et ces probabilités peuvent être fixes ou évoluées dans le temps.

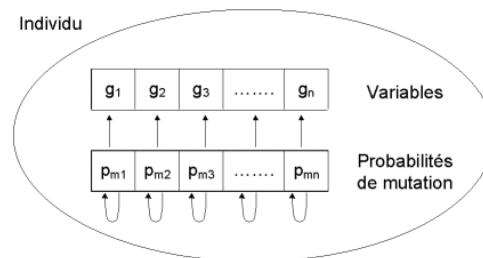


Fig. 4 : principe de l'auto-adaptation. A chaque variable est associée sa propre probabilité de mutation, qui est elle-même soumise au processus d'évolution. L'individu possède donc un second chromosome codant ces probabilités.



## Mutation :

### a- Mutation binaire :

La mutation binaire s'applique à un seul chromosome. Un bit du chromosome est tiré au hasard. Sa valeur est alors inversée. Il existe une variante où plusieurs bits peuvent muter au sein d'un même chromosome. Un test sous le taux de mutation est effectué non plus pour le chromosome mais pour chacun de ses bits : en cas de succès, un nouveau bit tiré au hasard remplace l'ancien.

### b- mutation réelle :

La mutation réelle ne se différencie de la mutation binaire que par la nature de l'élément qu'elle altère : ce n'est plus un bit qui est inversé, mais une variable réelle qui est de nouveau tirée au hasard sur son intervalle de définition.

**EXEMPLE:**

Trouver le maximum de la fonction  $f(x)=x$  sur l'intervalle  $[0,31]$  où  $x$  est un entier.

**1 : Codage de la fonction**

**Utilisation de la valeur binaire de  $x$  .**

ex:  $x = 2 \rightarrow \{1, 0\}$  , de même  $x = 31 \rightarrow \{1, 1, 1, 1, 1\}$  .

chromosome = chaîne binaire, allèle = "bit" particulier.

La fonction d'évaluation (fitness) est codée sur 5 bits (32 valeurs possibles de  $x$  ).

## 2 : Tirage et évaluation de la population initiale

**Taille de la population = 4 .**

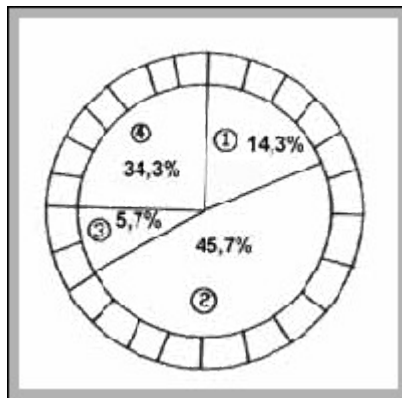
On tire de façon aléatoire 4 chromosomes sachant qu'un chromosome est composé de 5 bits et chaque bit dispose d'une probabilité  $\frac{1}{2}$  d'avoir une valeur 0 ou 1 .

Numéro	Chaîne	Fitness	% du total
1	00101	5	14.3
2	10000	16	45.7
3	00010	2	5.7
4	00110	12	34.3
Total		35	100

### 3 : Sélection

**Création d'une nouvelle population par un processus de sélection.**

Exemple : Roulette wheel (Goldberg [1989])



On tourne la roue 4 fois.

**Nouvelle Population**

Numéro	Chaîne
1	10000
2	01100
3	00101
4	10000

## 4 : Reproduction

### Phase 1 : Le Croisement

Les parents sont sélectionnés aléatoirement. Un lieu de croisement (*locus*) est tiré aléatoirement dans la chaîne. Une probabilité  $p_c$  est donnée que le croisement s'opère à ce lieu.

On suppose que les chromosomes 1 et 3, puis 2 et 4 sont appariés

$l=3$	$l=2$
100 00	01 100
001 01	10 000
10001	01000
00100	10100

## 4 : Reproduction

### Phase 2 : La mutation

La mutation est la modification aléatoire occasionnelle (de faible probabilité) de la valeur d'un bit (inversion d'un bit)  
 Ex:  $p_m = 0.05$  .

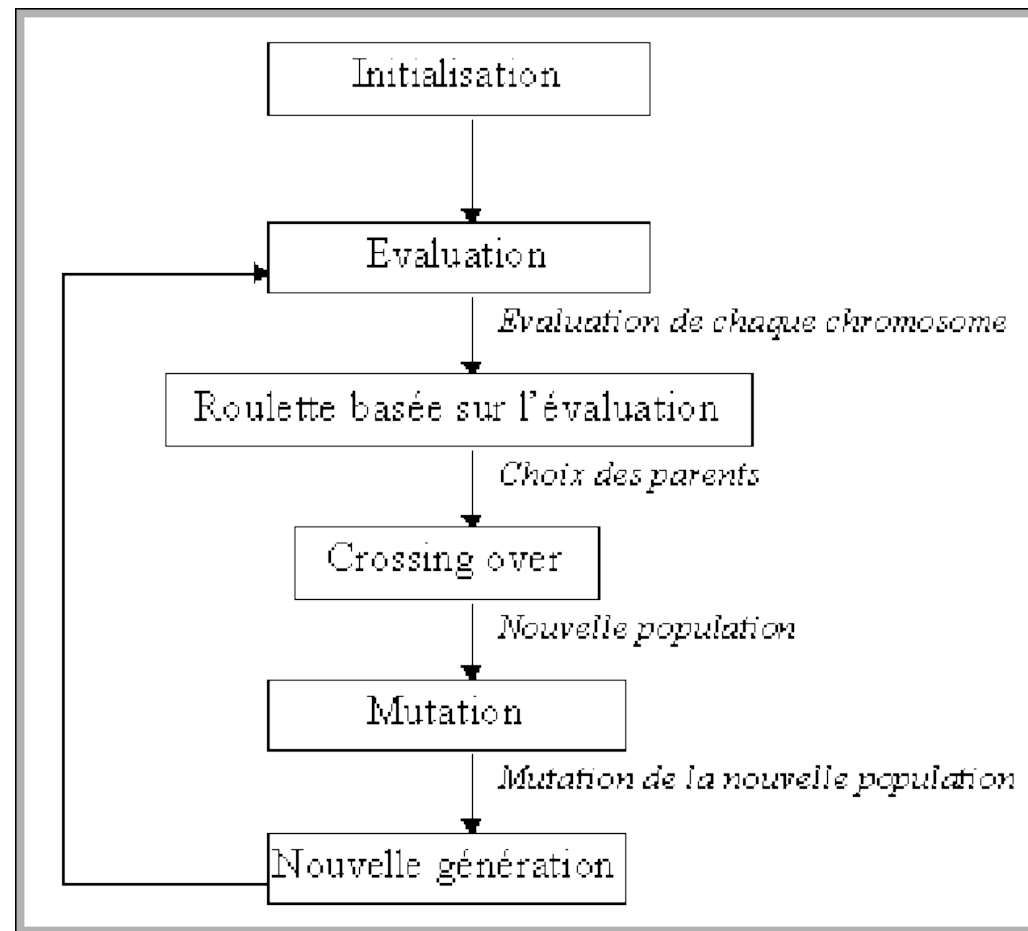
On tire pour chaque bit un chiffre aléatoire entre 0 et 1 et si ce chiffre est inférieur à  $p_m$  alors la mutation s'opère.

Anc. Chr.	Tirage aléat.	Nveau Bit	Nveau Chr.
	15 25 36 <b>04</b> 12		10011
10001	26 89 13 48 59	1	
00100	32 45 87 22 65	-	00100
01000	47 <b>01</b> 85 62 35	-	01000 11100
10100		1	

## 5 : Retour à la phase d'évaluation

Numéro	chaîne	Fitness	% du total
1	10011	19	32.2
2	00100	4	6.8
3	01000	8	13.5
4	11100	28	47.5
Total		59	100

# En Résumé





## Les problèmes de l'algorithme génétique :

### 1- le codage :

L'un des problèmes les plus importants est le codage des données :

c'est difficile de trouver un bon codage adaptée à la structure du problème.

L'application de la fonction de décodage lors l'évaluation de la fitness est coûteuse en temps de calcul.

Les opérateurs de croisement et mutation ne tiennent aucun compte de la structure du problème.

### 2- La dominance :

La domination d'un chromosome provoque la disparition rapide de certains éléments de la population, mais la méthode des AG présente plusieurs solutions contrairement à d'autres algorithmes comme recuit simulé qui présente une seule solution, alors il faut éliminer les chromosomes et laisser le meilleur (le mieux adapté).

### 3- Le croisement :

L'un des problèmes des AG c'est la position de codage, par exemple : une propriété importante est codée dans une chaîne de 8 bits par le 1er et le 8th bit, dans ce cas tout croisement peut se faire facilement mais si le codage est fait entre le 1er et le 2nd bit le croisement sera presque insensible.

C'est de ça qu'on vient l'idée de réorganiser les bits dans un chromosome de façon à créer des blocs stables.