Université de Batna2
Département d'informatique

Chapitre 2

Méthodes de Résolution de Problèmes

Introduction

Les algorithmes de recherche constituent l'une des approches les plus puissantes pour la résolution des problèmes en intelligence artificielle. Ce sont les mécanismes de résolution les plus généraux qui se déroulent dans un espace appelé espace d'états. Cet espace de recherche permet de recenser les états d'un système donné et de trouver parmi ces états une ou plusieurs solutions. Le passage d'un état à un autre se fait par l'application d'une action donnée. Ainsi, le problème de recherche de l'état solution dans l'espace nous ramènera à développer un arbre de recherche et à définir une stratégie de recherche sur cet arbre. Le but de la recherche dans un tel arbre serait la diminution du temps de recherche en trouvant une stratégie qui converge rapidement vers la solution.

La connaissance de quelques caractéristiques est très importante, elles peuvent avoir un impact immense sur la pertinence des diverses techniques possibles pour résoudre le problème.

Modélisation d'un problème

Pour modéliser un problème, il faut donc :

- Décrire qu'est ce qu'un état du problème
- Décrire l'état initial
- Définir les opérateurs permettant de passer d'un état à un autre: fonction successeur
- 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'état initial et la fonction successeur définissent implicitement l'espace des états du problème autrement dit l'ensemble de tous les états accessibles à partir de l'état initial.

Problème de recherche : Espace d'états

Les problèmes que l'on cherche à résoudre consistent à faire évoluer un système, d'un état dit initial à l'un des états dits terminaux ou buts, au moyen d'un ensemble de règles.

Un espace d'états est un graphe de résolution de problème (GRP) dont les sommets sont les états possibles du problème. Il y a un arc u = (i, j) dans le graphe si une règle permet de passer de l'état i à l'état j. On associe une fonction coût C(u) à cet arc. On doit distinguer le sommet initial et les sommets terminaux.

Formellement, un graphe d'états ou espace de recherche est un quintuplet (S, A, S_0, G, C) :

– **S** est l'ensemble des sommets du graphe (ou états du problème),

– **A : S →S** l'ensemble des arcs défini par un ensemble d'opérateurs

– **$S_0 \in S$** le sommet de départ (état initial du problème),

– **$G \subset S$** l'ensemble des sommets buts,

– **C : A→** $\mathbb{R}$, où C(i, j) est le coût de l'application de la règle permettant de passer de l'état i à l'état j.

Pour un même problème, on peut en général associer plusieurs graphes d'états.

Processus de recherche

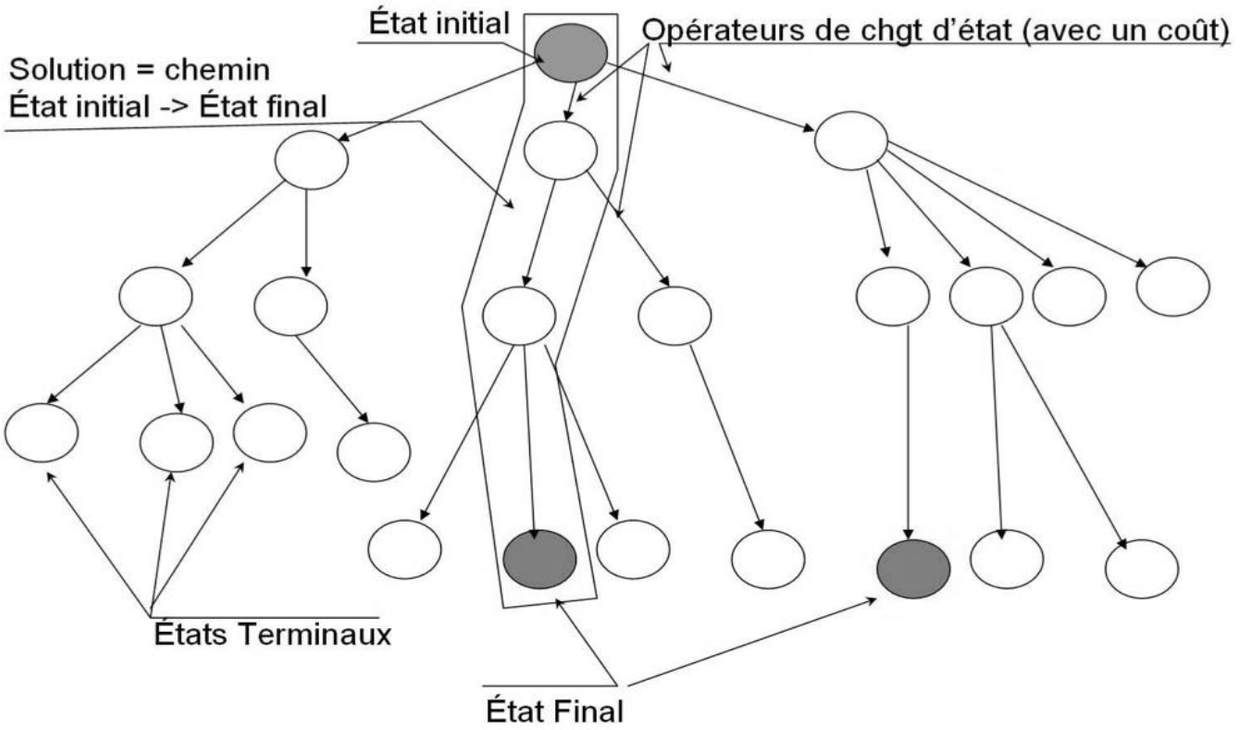
- Vérifier l'état actuel,
- Exécuter les actions permises pour passer à l'état suivant,
- Vérifier si le nouveau état est bien l'état but

→ si ce n'est pas le cas :

  – le nouveau état devient l'état courant

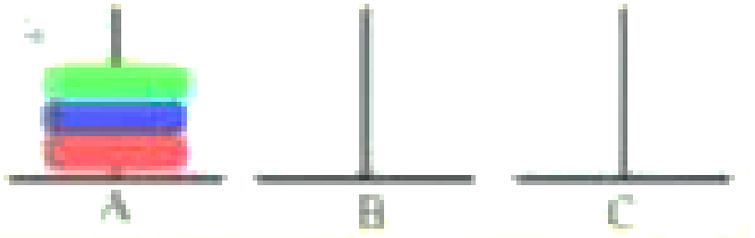
  – répéter la procédure jusqu'atteindre l'état but ou bien l'espace de recherche est complètement exploré

Une solution est un chemin qui correspond à une séquence d'états en passant de l'état initial jusqu'à l'état final.

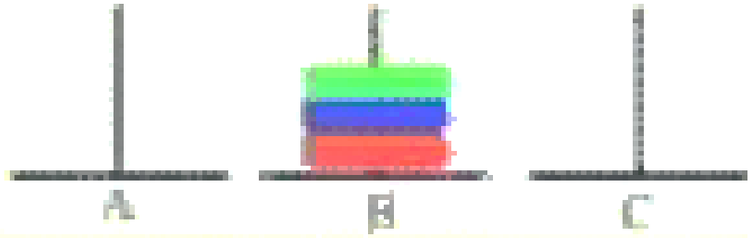


Problème des tours et de disques

- Considérons le **problème** des 3 tours et 3 disques de couleurs vert, bleu et rouge. **Opérateurs:** Déplacement du dernier disque empilé d'une tour vers n'importe quel autre tour (position top). **Objectif**: Atteindre la configuration finale

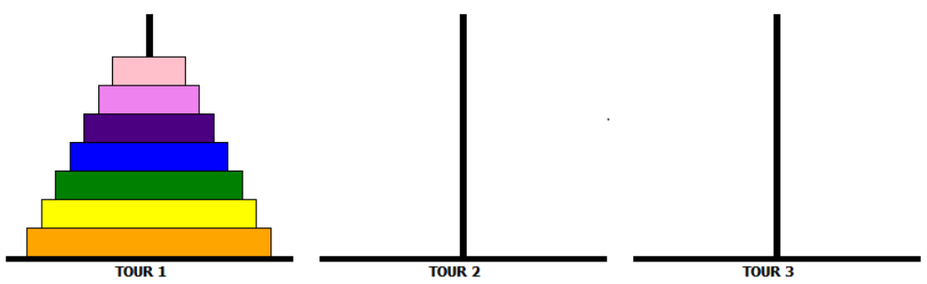


Etat initial

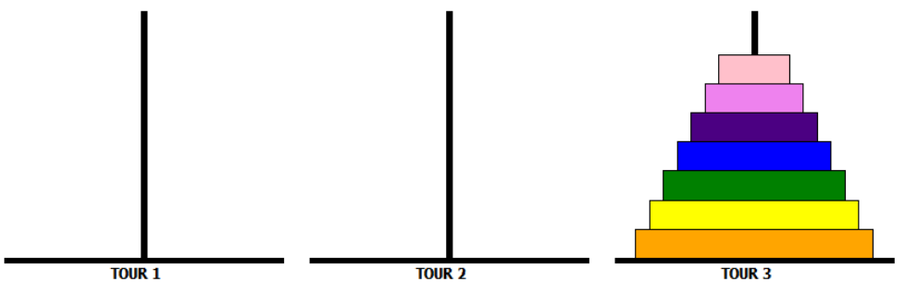


Etat final (but)

- Le problème de la **tour de Hanoi** consiste en trois tours et une pile de disques rangés du plus grand au plus petit. Les disques sont initialement empilés à gauche. Il faut réussir à déplacer cette pile entièrement sur la tour de droite. Pour cela, il faut respecter les règles suivantes :
  - ne déplacer qu'un seul disque à la fois,
  - un disque ne peut pas être posé sur un disque plus petit.



Etat initial



Etat final (but)

Problème des 8 reines

Placer 8 reines sur l'échiquier de sorte qu'aucune Reine ne soit attaquée par une autre Reine.
Comment nous allons formuler l'espace de recherche pour ce problème?

→ Il existe différentes façons de formuler ce problème

Formulation 1

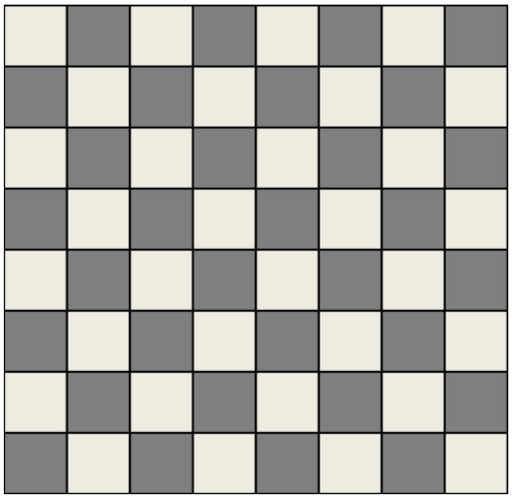
- Etats : Un arrangement quelconque des 8 reines sur l'échiquier,
- Etat initial : 0 reine sur l'échiquier,
- Fonction successeur : ajouter une reine dans n'importe quelle case,
- Test But : 8 reines sur l'échiquier, aucune d'entre elles n'est attaquée.

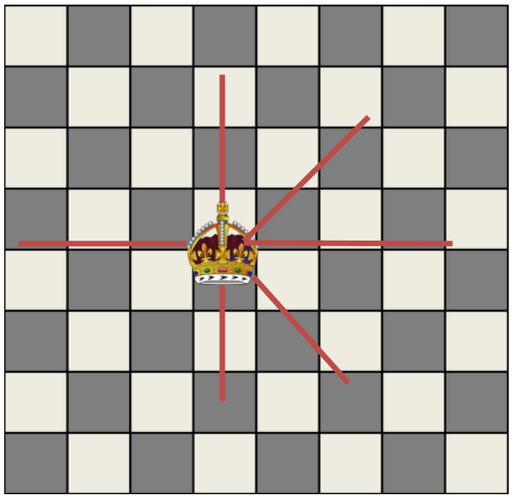
Formulation 2

- Etats : Un arrangement quelconque des 8 reines sur l'échiquier,
- Etat initial : Toutes les reines sont positionnées sur la colonne 1,
- Fonction successeur : changer la position d'une quelconque reine,
- Test But : 8 reines sur l'échiquier, aucune d'entre elles n'est attaquée.

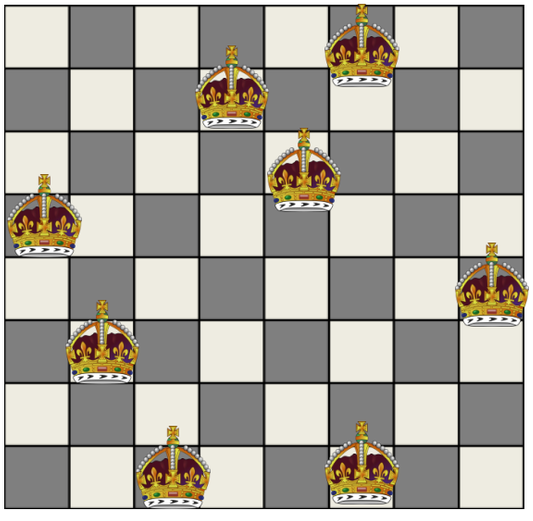
Problème des 8 reines

Placer les 8 reines sur l'échiquier de telle façon que 2 reines ne peuvent se trouver sur la même ligne, ni la même colonne et ni la même diagonale

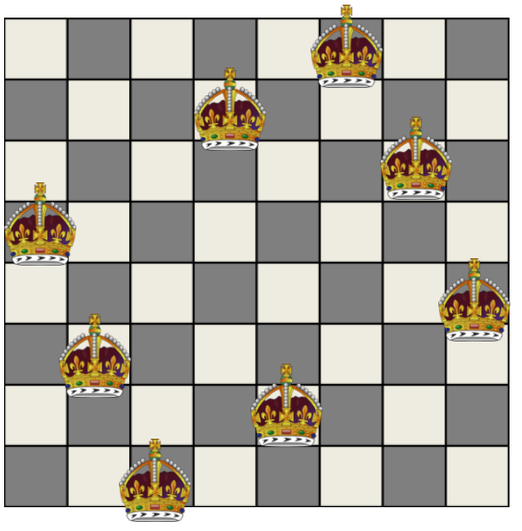




Un exemple de 2 arrangements possibles



Non valide

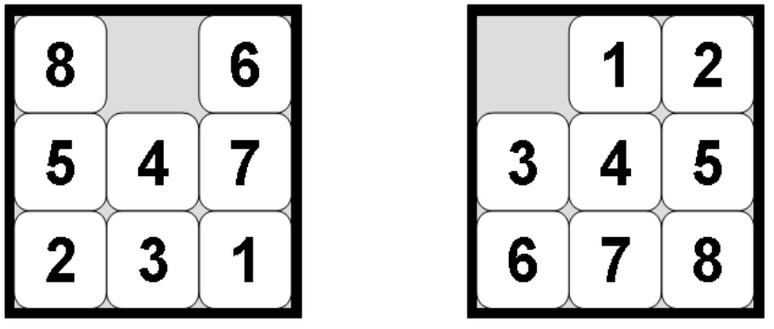


Solution

Autres problèmes classiques de l'IA

- **Problème du voyageur du commerce** (Travelling Salesman Problem: TSP) : Etant donné une liste de villes, et des distances entre toutes les paires de villes, le problème consiste à déterminer un plus court chemin qui visite chaque ville une et une seule fois et qui termine dans la ville de départ.

- **8-Puzzle:** Le problème consiste à partir d'une situation initiale, à déterminer les différents mouvements de la case grise qui ne peut être permutée qu'avec une case qui lui est adjacente. On cherche donc à obtenir la situation finale ou but.



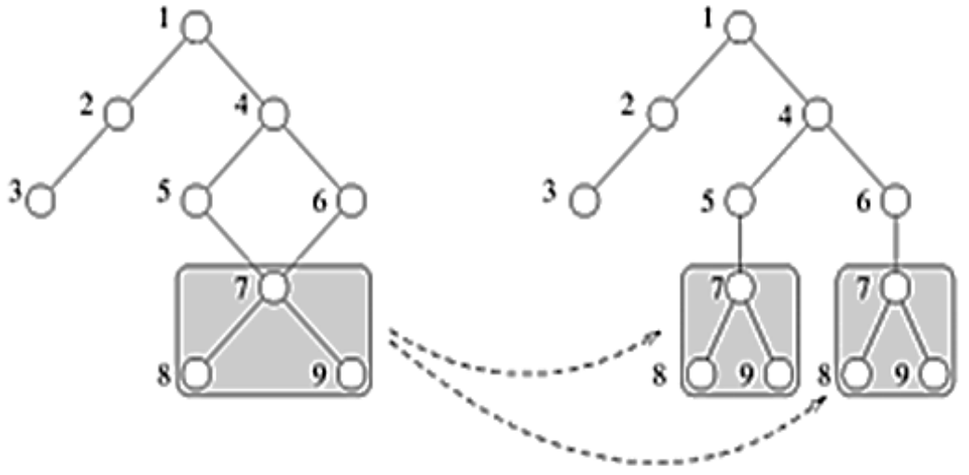
Configuration initiale & finale

→ **Pas d'algorithme permettant de trouver une solution exacte et rapide dans tous les cas.**

Graphe vs Arbre de Recherche

Différentes stratégies permettent d'explorer l'espace de recherche en se basant sur la structure d'un arbre

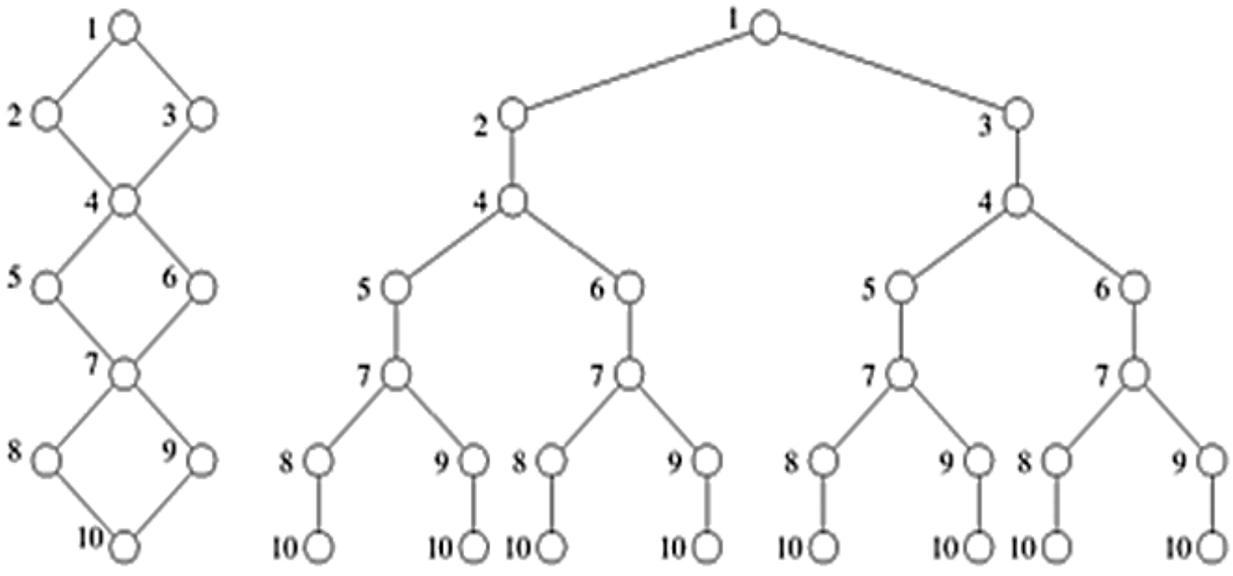
→ Trouver la solution du problème : un chemin (optimal ou quelconque)



(a)

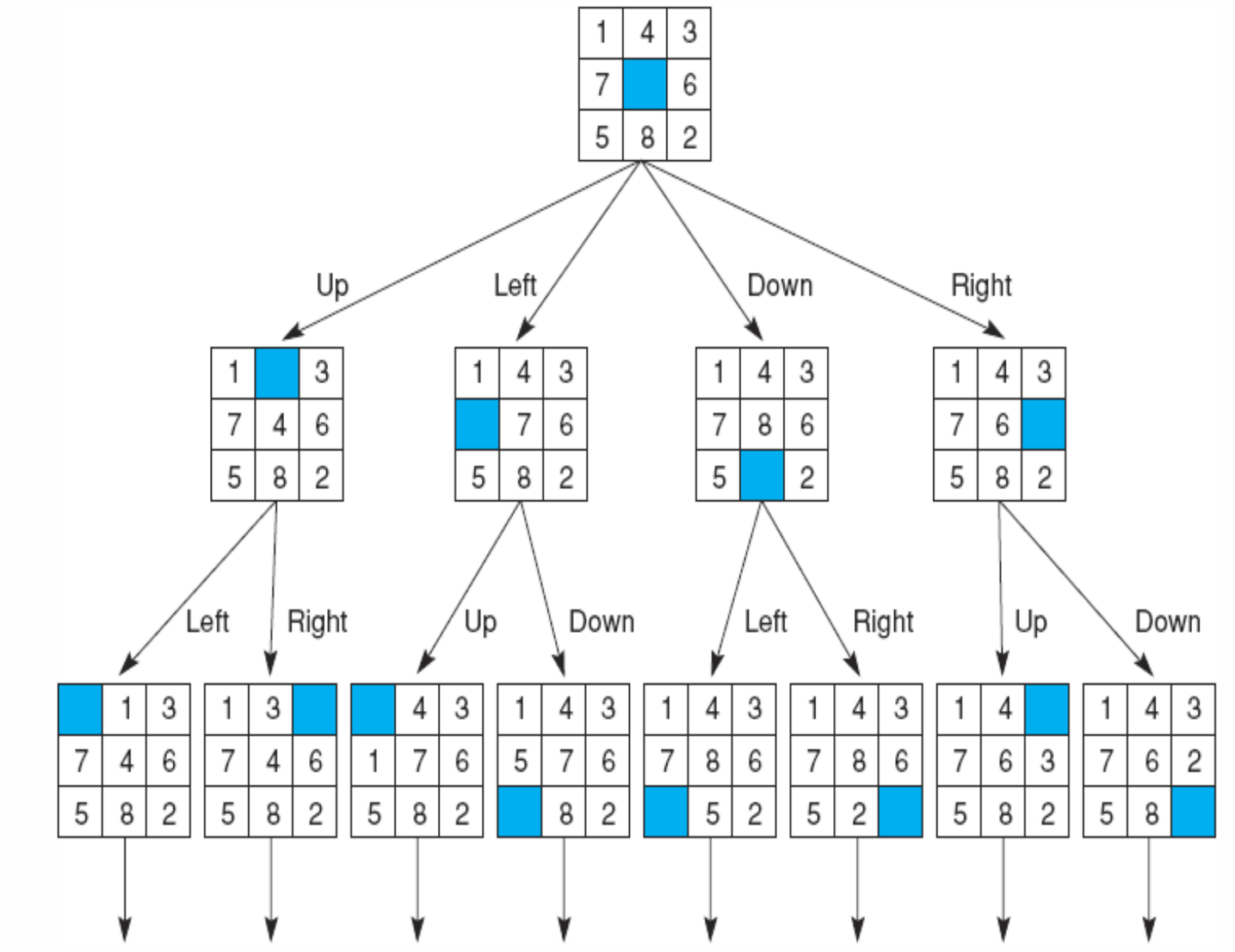
Lister tous les chemins possibles, et éliminer les cycles

→ Un arbre de recherche



(b)

Arbre de recherche : Problème du 8-puzzle



Algorithme de recherche

Idée générale

1. Démarrer la recherche avec la liste contenant l'état initial du problème.
2. Si la liste n'est pas vide alors :
   a) Choisir (à l'aide d'une stratégie) un état *e* à traiter.
   b) Si *e* est un état final alors retourner recherche positive
   c) Sinon, rajouter tous les successeurs de *e* à la liste d'états à traiter et recommencer au point 2.
3. Sinon retourner recherche négative.

Comment éviter les redondances des états

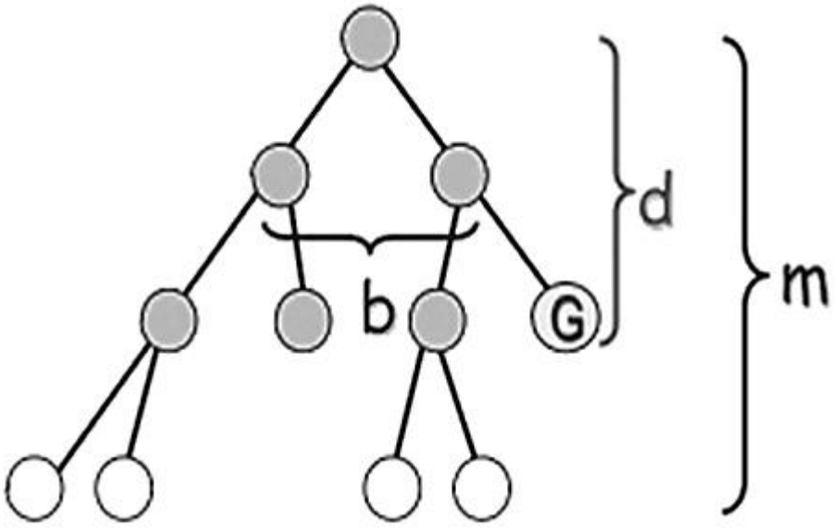
Dans l'ordre croissant de l'efficacité et le temps de calcul:

- Ne pas retourner à l'état *e* duquel on vient: développer une fonction qui peut sauter les successeurs qui ont le même nœud parent que *e*,
- Ne pas créer un chemin avec cycle: développer une fonction qui peut sauter les successeurs qui sont dans la liste des nœuds exécutés,
- Ne pas générer un état qui a été déjà développé: garder en mémoire tous les nœuds qui ont été développés à moins que le cout du nœud développé soit inférieur au nœud courant.

Evaluation des stratégies de recherche

Les algorithmes de recherche sont généralement évalués selon quatre critères:

- **Complétude**: si la stratégie garantit de trouver la solution si elle existe,
- **Optimalité:** une fois la solution trouvée, est ce qu'elle garantit d'avoir un cout minimal,
- **Complexité en temps**: temps calculé en nombre de nœuds développés pour trouver la solution,
- **Complexité en espace**: l'espace utilisé par l'algorithme est mesuré en terme de la taille maximale de la liste des nœuds.



G : le but

b : Le facteur de branchement

m : La profondeur de l'arbre

d : la profondeur jusqu'au premier but

Classes d’algorithmes de recherche

On distingue deux classes principales d'algorithmes de recherche :

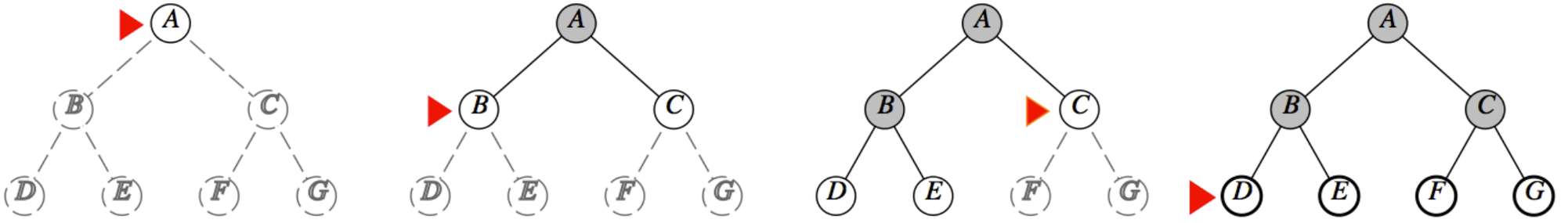
1. **Algorithmes non-informés ou "aveugles"**: qui réalisent une recherche exhaustive (essayer toutes les solutions possibles), sans utiliser aucune information supplémentaire sur la structure de l’espace d’états pour optimiser la recherche.
2. **Algorithmes informés ou heuristiques**: qui utilisent des sources d’informations supplémentaires. Ces algorithmes parviennent ainsi à de meilleures performances.

RECHERCHE NON INFORMÉE : MÉTHODES AVEUGLES

Parcours en Largeur d'abord

La stratégie la plus naturelle est l'exploration du graphe G en largeur à partir de l'état initial S_0 (racine de l'arbre de recherche). En effet, le principe de cette méthode consiste à examiner tous les nœuds d'une certaine profondeur avant d'examiner les nœuds d'une profondeur supérieure. Tout d'abord on commence par l'état initial puis tous ses successeurs directs puis les successeurs des successeurs et ainsi de suite.

- Développement de tous les nœuds au niveau i avant de développer les nœuds au niveau i + 1.
- Les nouveaux successeurs vont à la fin.



Les nœuds visités sont : **A – B – C – D – E – F – G**

L'avantage de cette méthode est sa simplicité de conception et de programmation. Son inconvénient en est le coût, au niveau de la complexité en temps mais aussi en espace.

Algorithme Parcours en Largeur

L'implémentation de cet algorithme nécessite l'utilisation d'une *file d'attente.* Cette structure de données permet de placer les nouveaux nœuds à la fin de la liste des nœuds à développer. La stratégie utilisée pour gérer la file est *FIFO* (First In First Out). Les opérations d'ajout et de suppression d'un élément *s* de la file sont : Enfiler(file, s) et défiler(file, s).

Il est important de marquer les états déjà visités au fur et à mesure de la recherche afin d'éviter les calculs redondants. La trace est gardée dans une liste initialement vide. L'algorithme s'arrête dès que l'on rencontre l'état But.

```
BFS (graphe G, sommet s) /* Breadth First Search
Debut
    f = CreerFile();
    Marquer(s);
    Enfiler(f, s);
    Tantque NON FileVide(f) Faire
        x = Défiler(f);
        Afficher(x)
        Tantque ExisteSucc(x) Faire
            z = SuccSuivant(x);
            Si NonMarqué(z) Alors
                Marquer(z);
                Enfiler(f, z);
            Finsi
        Fintantque
    Fintantque
Fin
```

Exemple : Recherche en Largeur

OPEN: liste des noeuds générés, mais non encore explorés ou bien explorés, mais non encore développés

CLOSED: liste des noeuds explorés et développés

X : noeud courant

Succ X : Liste des noeuds successeurs du noeud X

- A vers B et C
- B vers D et E
- C vers F et G
- D vers I et J
- I vers K et L

→ Etat initial: A

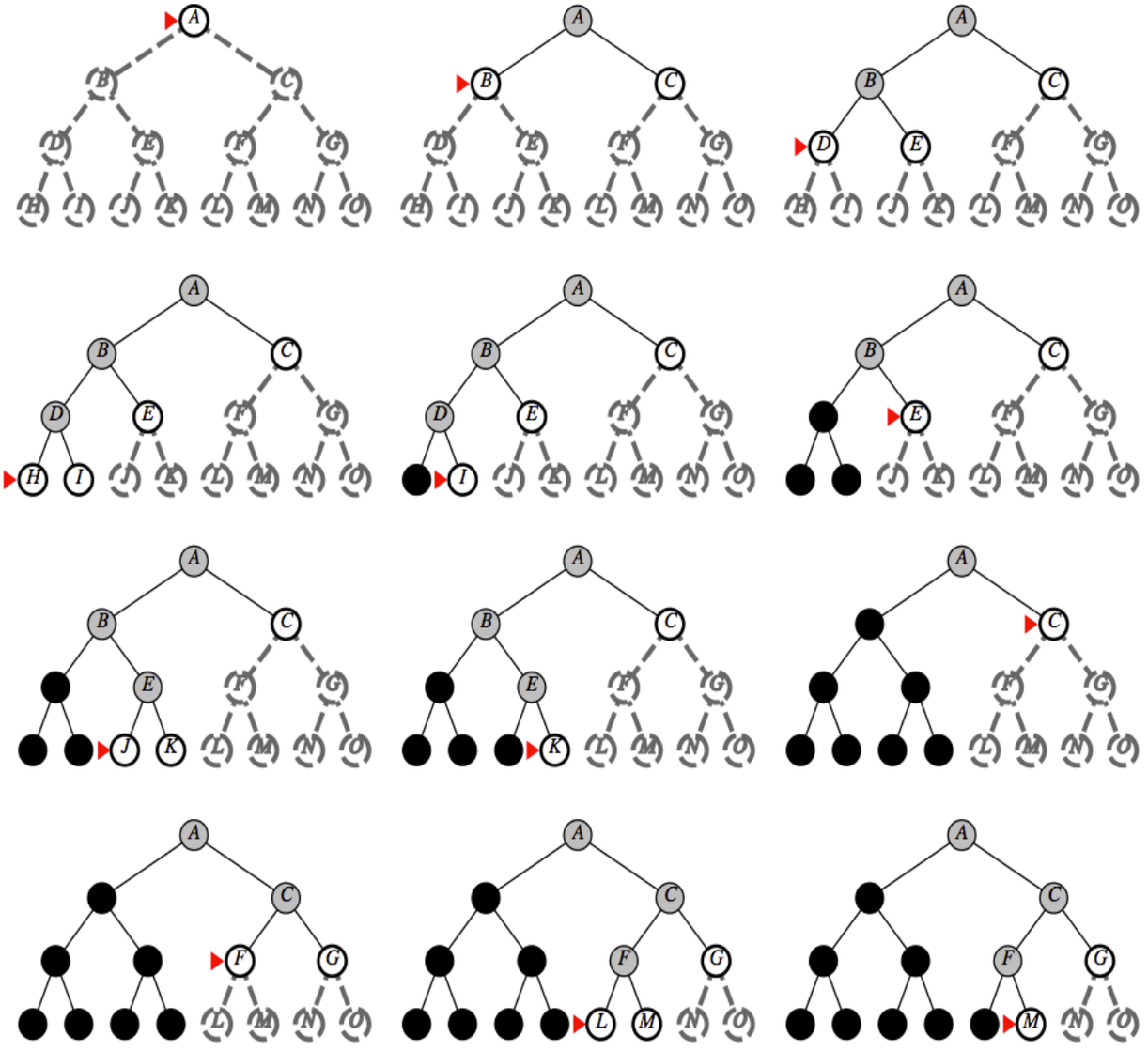
→ Etats finaux: E , J

N	OPEN	CLOSED	X	SUCC X	Etat
1	A				Echec
2			A	B, C	Echec
3	B, C	A			Echec
4	C	A	B	D, E	Echec
5	C, D, E	A, B			Echec
6	D, E	A, B	C	F, G	Echec
7	D, E, F, G	A, B, C			Echec
8	E, F, G	A, B, C	D	I, J	Echec
9	E, F, G, I, J	A, B, C, D			Echec
10	F, G, I, J	A, B, C, D	E	aucun	Succès

Parcours en profondeur d'abord

Le parcours en profondeur signifie le développement total d'une branche entière avant de parcourir le reste de l'arbre. La descente se fait jusqu'à un but ou une feuille. Ce développement peut ramener à trois situations différentes :

- La solution est trouvée et dans ce cas le développement de la branche s'arrête avec une possibilité de "backtraking" si d'autres solutions sont encore sollicitées.
- La solution n'est pas trouvée et un état d'échec est détecté (c'est un état qui n'engendre pas d'autres états) et dans ce cas le "backtraking " est appliqué pour poursuivre la recherche.
- Une branche infinie est à explorer et dans ce cas, un test d'arrêt sur une profondeur maximale doit être intégré dans l'algorithme.



Algorithme du parcours en profondeur

L'implémentation de cette méthode de recherche nécessite l'utilisation d'une *Pile.* Cette structure de données permet de placer les nouveaux nœuds au début de la liste des nœuds à développer. La stratégie utilisée pour gérer la pile est *LIFO* (Last In First Out). Les opérations d'ajout et de suppression d'un élément *s* de la file sont : Empiler(pile,s) et Dépiler(pile,s).

DFS2 (graphe G, sommet s) */ recursive
Debut
    Marquer(s);
    Pour chaque z ϵ Succ (s) **Faire**
        Si NonMarqué(z) **Alors**
            DFS(G, z);
        Finsi
    Fin-pour
Fin

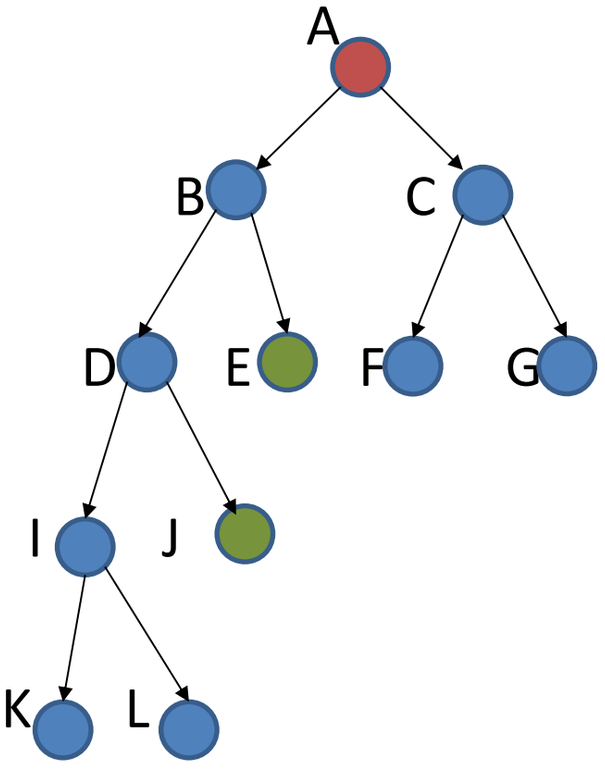
L'algorithme ci-dessous est une implémenetation itérative

DFS1 (graphe G, sommet s) /* Depth First Search
Debut
    p = CreerPile();
    Marquer(s);
    Empiler(p, s);
    Tantque NON PileVide(p) **Faire**
        x = Dépiler(p);
        Afficher(x)
        Pour chaque z ϵ Succ (x) **Faire**
            Si NonMarqué(z) **Alors**
                Marquer(z);
                Empiler(p, z);
            Finsi
        Fintantque
    Fintantque
Fin

Exemple : Parcours en Profondeur

→ Etat initial: A

→ Etats finaux: E , J

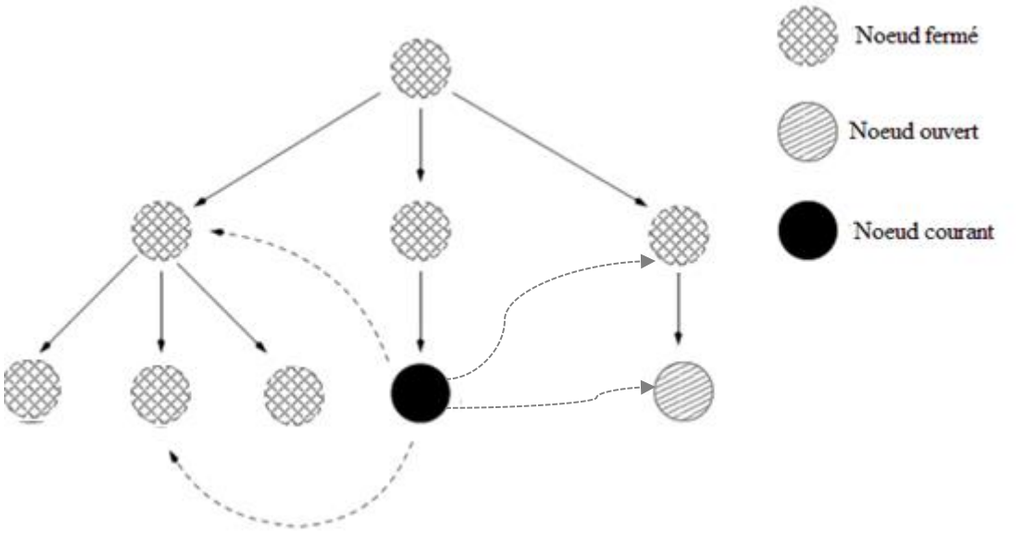


It	OPEN	CLOSED	X	SUCC X	Etat
1	A				Echec
2			A	B, C	Echec
3	B, C	A			Echec
4	C	A	B	D, E	Echec
5	D, E, C	A, B			Echec
6	E, C	A, B	D	I, J	Echec
7	I, J, E, C	A, B, D			Echec
8	J, E, C	A, B, D	I	K, L	Echec
9	K, L, J, E, C	A, B, D,I			Echec
10	L, J, E, C	A, B, D, I	K	aucun	Echec
11	J, E, C	A, B, D, I	L	aucun	Echec
12	E, C	A, B, D, I	J		Succès

Traitement des états répétés

➢ Parcours en Largeur d'abord (BFS)

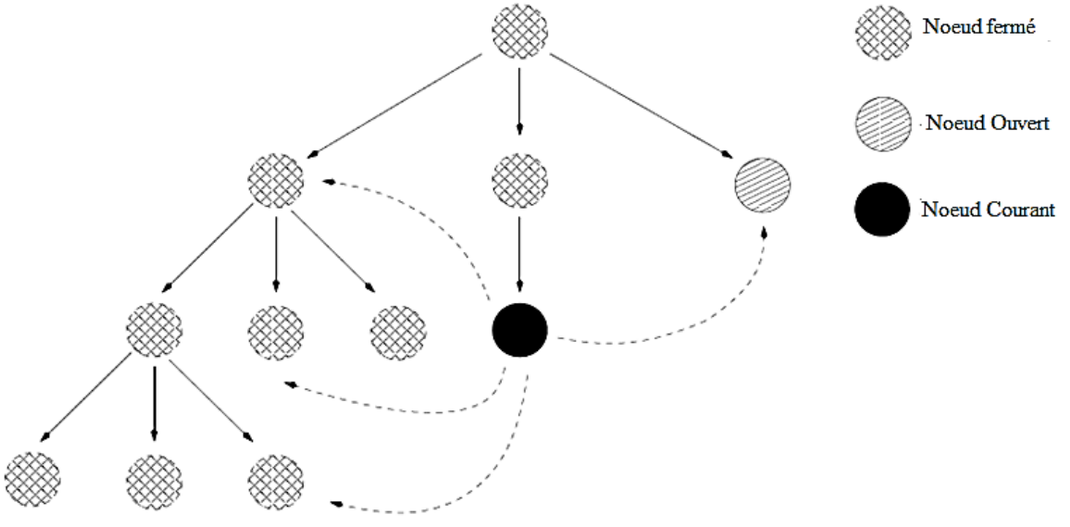
Si l'état répété appartient à la liste des nœuds fermés(closed) ou ouverts(open), le chemin actuel a une profondeur égale ou supérieure à l'état répété et dans ce cas il peut être ignoré.



➢ Parcours en Profondeur d'abord (DFS)

Si l'état répété appartient à la liste des nœuds fermés, le chemin actuel est conservé si sa profondeur est inférieure à l'état répété.

Si l'état répété est dans la liste des nœuds ouverts, le chemin actuel a toujours plus de profondeur que l'état répété et dans ce cas il peut être ignoré.



Satisfaction des critères d'évaluation

Recherche en Profondeur D'abord

Complétude:

- Oui → si on évite les états répétés ou si l'espace de recherche est fini
- Non → si la profondeur est infini, s'il y a des cycles

Complexité en temps: l'algorithme explore tous les nœuds

→ $1 + b + b^2 + \ldots + b^m = O(b^m)$: exponentielle en d

Complexité en espace: Garde tous les nœuds du chemin à un instant donné en mémoire,

→ A chaque profondeur $p < m$, on a (b-1) nœuds

→ A la profondeur m, on a b nœuds

→ Au total, (m-1)*(b-1) + b = O(b*m) : linéaire en m

La longueur du chemin est au plus m et chacun des nœuds sur le chemin aura au plus b − 1 successeurs à considérer.

Optimalité: Non

Recherche en Largeur D'abord

Complétude:

Oui → si b est fini,

Complexité en temps: l'algorithme explore tous les nœuds → $1 + b + b^2 + \ldots + b^d = O(b^d)$: exponentielle en d

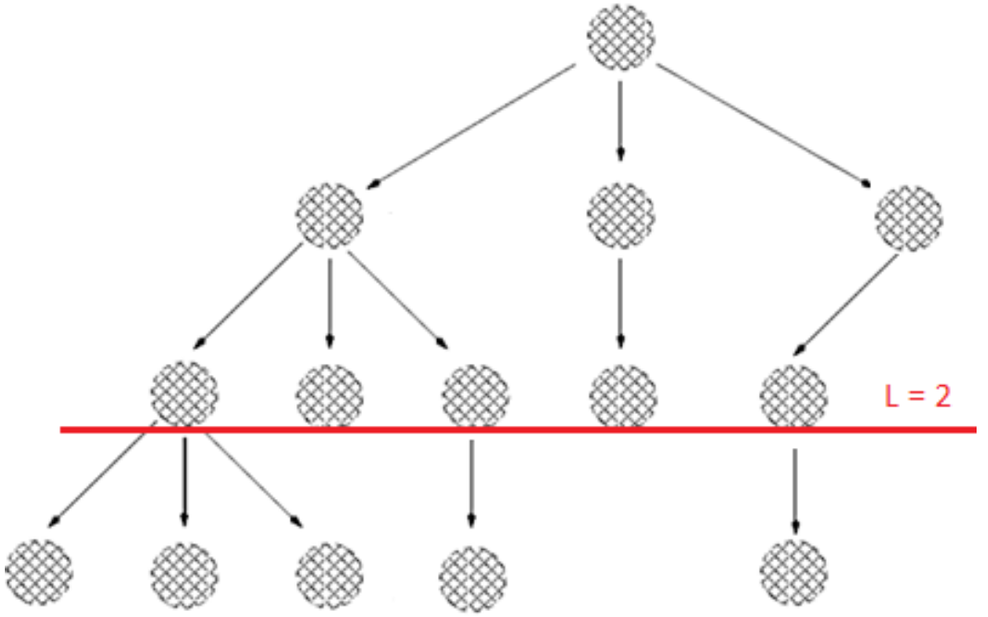
Complexité en espace: Garde tous les nœuds en mémoire → $O(b^d)$

Optimalité: Oui si les couts sont identiques

Recherche en profondeur limitée

Principe

Elle se base sur la recherche en profondeur d'abord en imposant une limite de profondeur l, ce qui signifie que les nœuds à la profondeur l n'ont pas de successeurs. La recherche est coupée à une certaine profondeur bien précise l.



Algorithme

DFSlimite(Graphe, d, limit)
L ← liste contenant l'état initial,
Boucle
  Si L est vide return 'echec'
  nœud ← Premier(L)
  Si nœud = but return chemin(Graphe, d)
  Sinon
    Si d =limit alors couper(Graphe, d)
    sinon ajouter les nœuds générés
          au début de la liste L,
  FSi
FinBoucle

Recherche en profondeur limitée itérative

- L'algorithme consiste à appliquer, d'une manière itérative, la recherche en profondeur d'abord ayant une profondeur maximale qui augmente à chaque itération. Initialement, la profondeur maximale est de 1.
- Le comportement de cette recherche est similaire à BFS, mais avec une complexité spatiale moins importante. Seul le chemin réel est conservé en mémoire; les nœuds sont régénérés à chaque itération.
- Les problèmes de DFS liés aux branches infinies sont évités.
- Pour garantir que l'algorithme se termine, dans le cas où il n'y a pas de solution, une profondeur d'exploration maximale générale peut être définie.

Principe :

- Appliquer DFS au niveau 0 : traitement du nœud initial ayant 0 successeurs,
- Si aucune solution n'est trouvée alors passer au niveau 1 et ainsi de suite jusqu'à obtention de la solution.

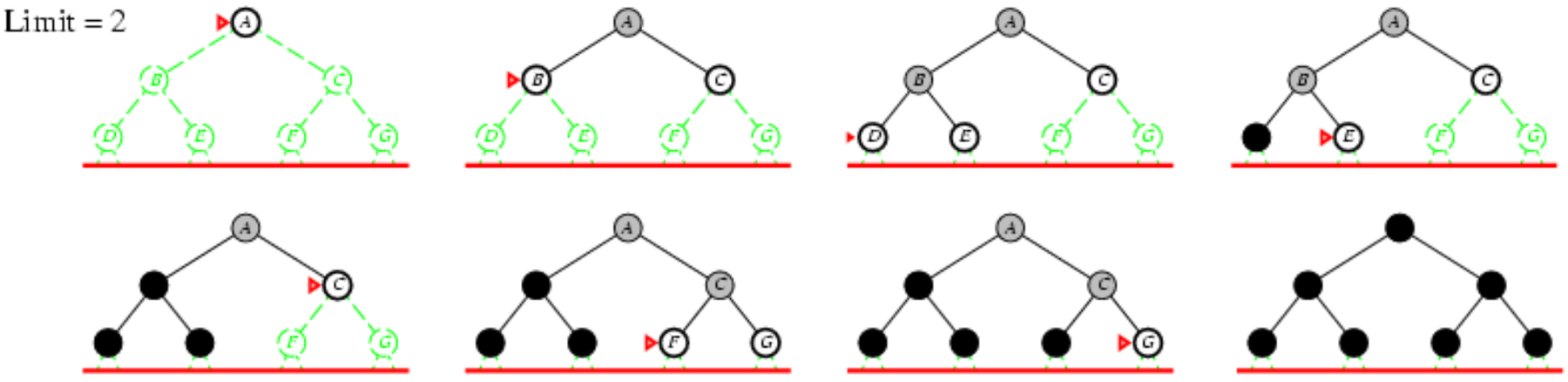
Recherche en profondeur Itérative L=0

Limit = 0

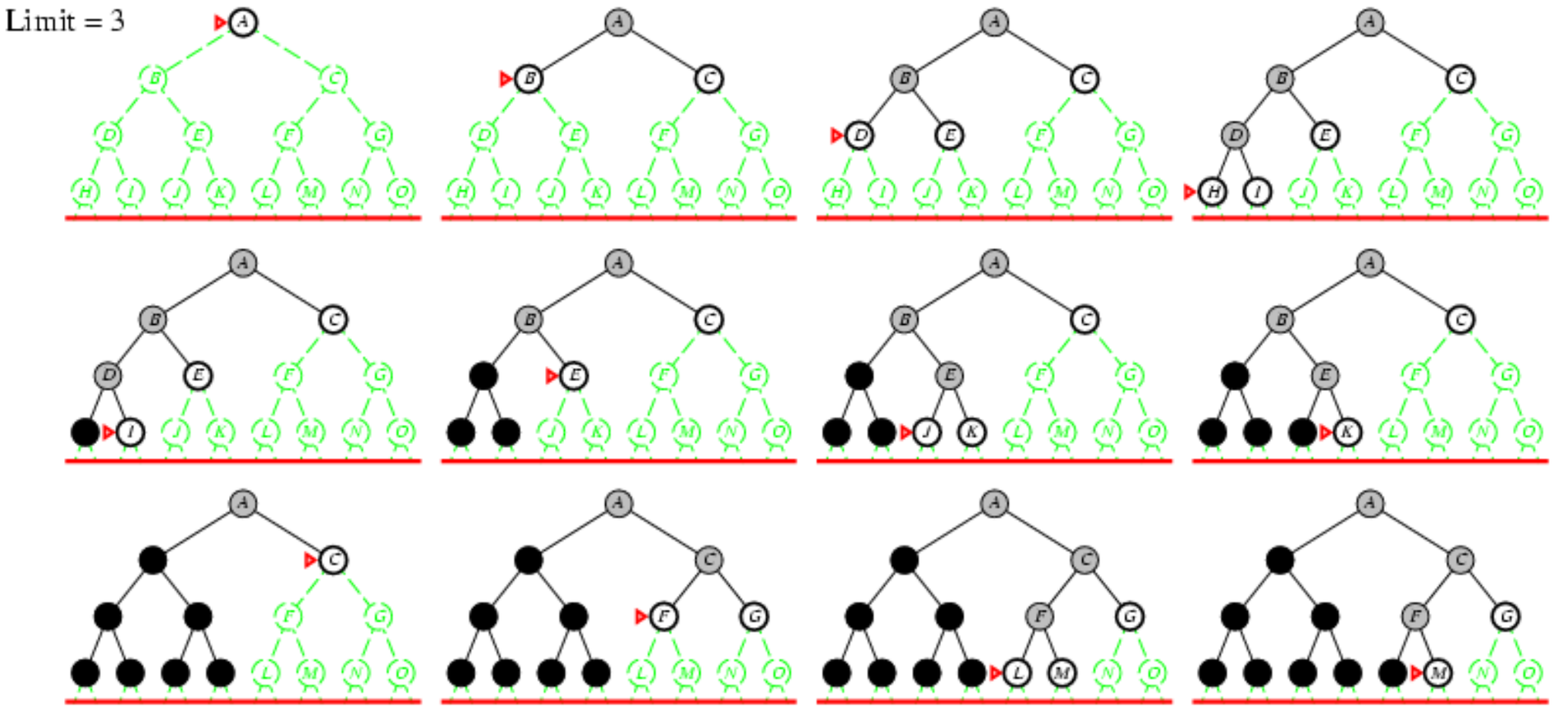
Recherche en profondeur Itérative L=1

Limit = 1

Recherche en profondeur Itérative L=2



Recherche en profondeur Itérative L=3



Satisfaction des critères d'évaluation

Recherche en profondeur itérative

Complétude:

- Oui → si on évite les états répétés ou si l'espace de recherche est fini
- Non → si la profondeur est infini, s'il y a des cycles

Complexité en temps: l'algorithme explore tous les nœuds

→ $(d+1) + d.b + (d-1).b^2 + \ldots + 1.b^d = O(b^d)$: exponentielle en d

Complexité en espace: Nombre de nœuds conservés

→ O(b.d) : linéaire en d

Optimalité: Non mais si les cout sont identiques on peut parler d'optimalité

Recherche en profondeur Limitée

Complétude:

→ Oui : Si L > d,

Complexité en temps: $O(b^L)$ exponentielle en L

Complexité en espace : O(b.L) linéaire

Optimalité: Non

→ échec ou absence de solution dans les limites de la recherche

Recherche Bidirectionnelle

Principe: Cette technique consiste à lancer deux agents pour explorer l'arbre de recherche. Le premier part de l'état initial et se dirige vers l'état but (*forward search*) tandis que le second part l'état but et se dirige vers l'état initial (*backward search*). Si au cours du processus de recherche les deux agents se rencontrent , ou atteignent un des états accessible soit par l'état initial ou l'état final, on dit que le chemin solution est trouvé.

Stratégie de recherche: deux files d'attente

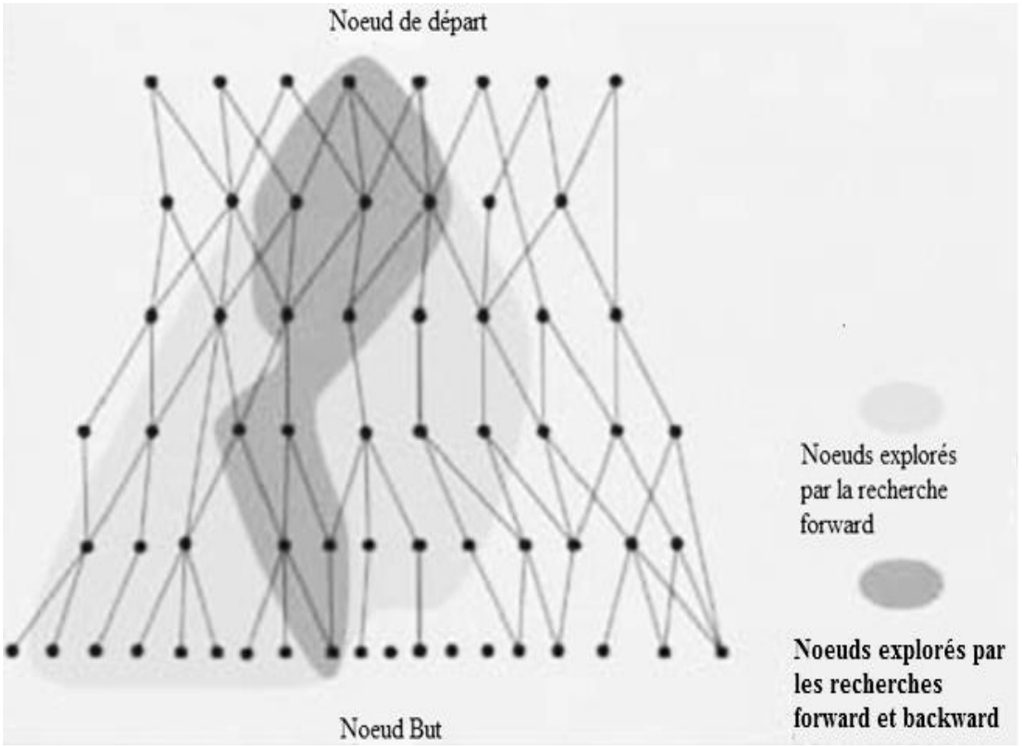
Complétude : Oui

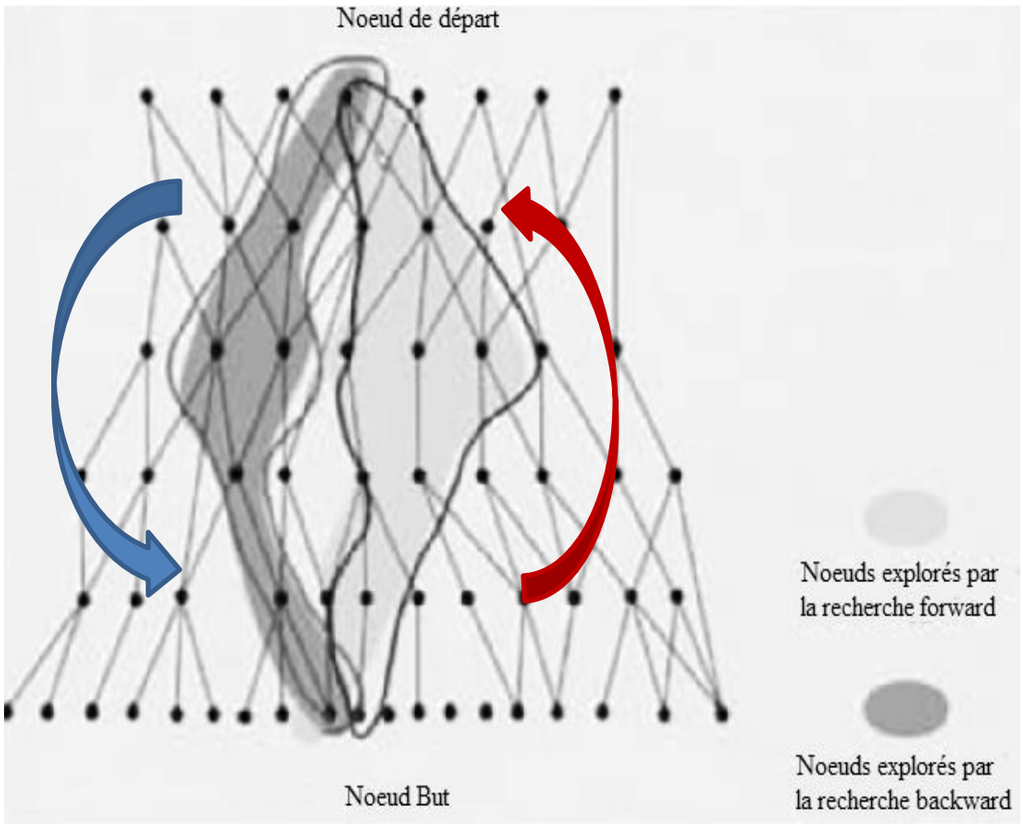
Complexité en temps: $O(b^{d/2}) << O(b^d)$

Complexité en espace: $O(b^{d/2}) << O(b^d)$

Optimalité: Oui (si les couts sont identiques et dans les directions, BFS est utilisée)

Nécessite: des états solutions explicitement définies, des opérateurs dont on connaît la fonction inverse (capable de générer l'état prédécesseur d'un état donné)





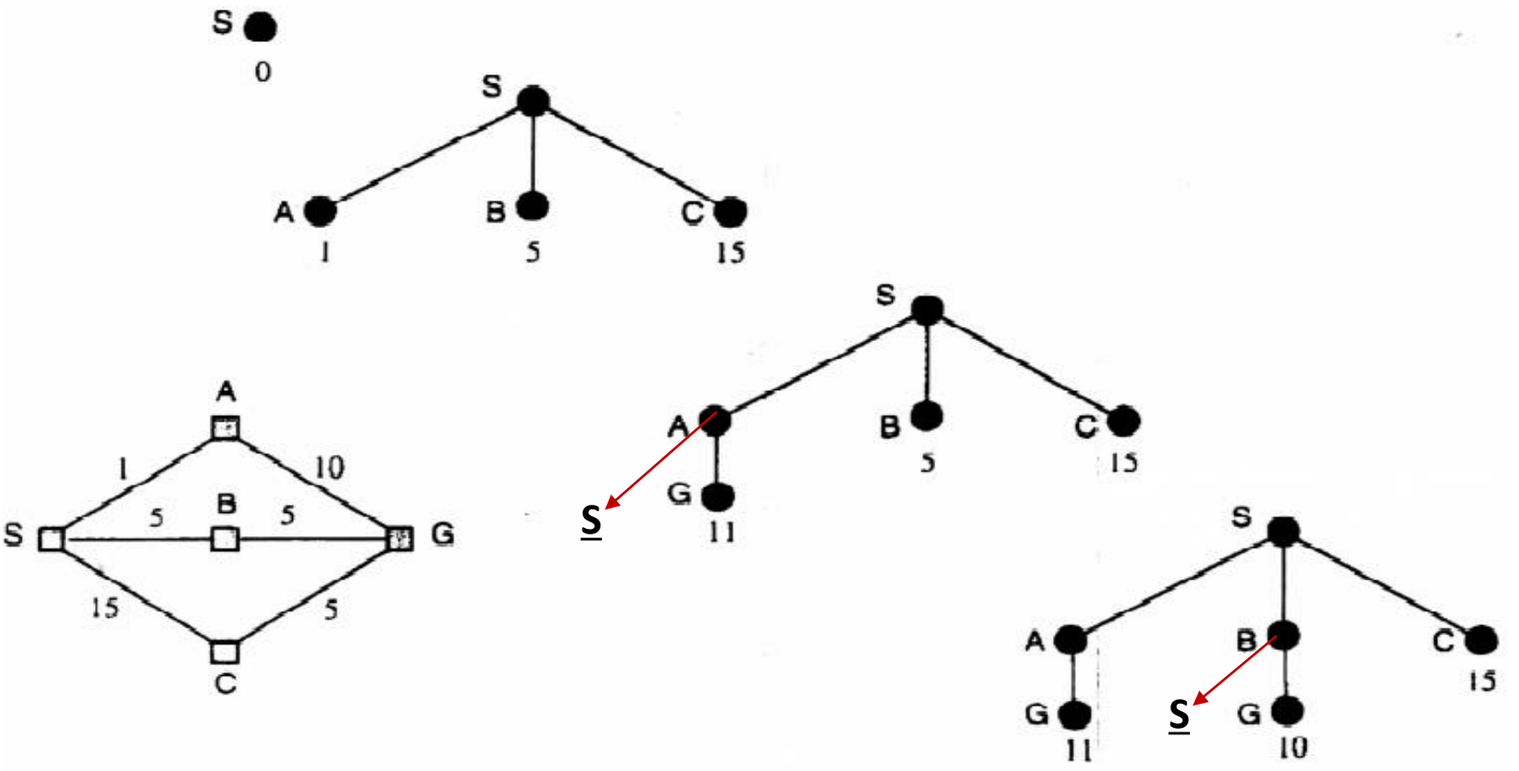
Recherche à coût uniforme (UCS)

Principe : Cette technique est une variante de la méthode de recherche en largeur (BFS). Au lieu de générer les nœuds niveau par niveau, les nœuds seront générés selon la valeur du coût associée.

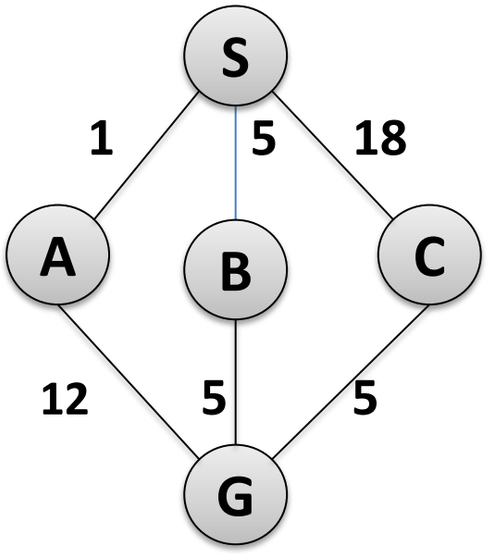
Algorithme :

- Enfiler les nœuds selon l'ordre croissant des coûts,
- Soit g(n): le coût du chemin partant de l'état initial jusqu'à l'état courant n,
- Développer le nœud de coût minimal,

Dans l'espace de recherche ci-dessous, chaque nœud est étiqueté par la fonction coût définie par g. Le nœud But (G) est déterminé avec un coût de 10.



Exemple



It	OPEN	CLOSED	X	Succ X	Etat
1	S		S		Echec
2		S		A, B, C	
3	A(1), B(5), C(18)	S			Echec
4	B(5), C(18)	S	A	G	Echec
5	B(5),G(13), C(18)	S, A			
6	G(13), C(18)	S, A	B	G	Echec
7	G(10), C(18)	S, A, B			
8	C(18)	S, A, B	G		Succès

A chaque mise à jour , la liste OPEN doit être triée par ordre croissant de la fonction g.

Ligne 7: ∀ d, C(18)+ d >= G(10)

Ligne 8: Arrêt ,

Chemin obtenu de coût minimal est : S - B - G.

ALGORITHMES DE RECHERCHE INFORMÉE : RECHERCHE HEURISTIQUE

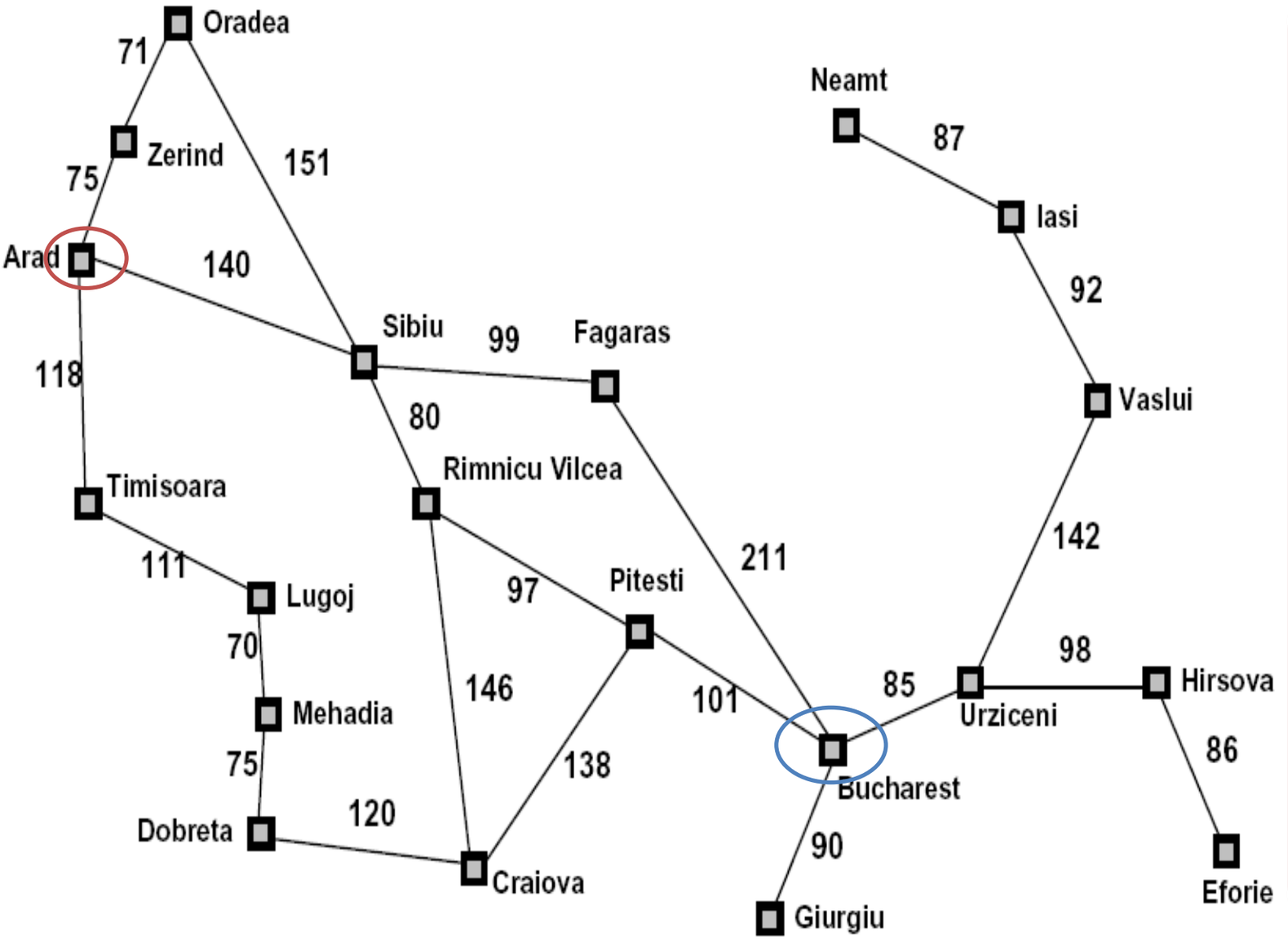
Introduction

- L’analyse de complexité des algorithmes de recherche DFS et BFS nécessitent un espace mémoire important pour générer l’espace d’état complet. Les heuristiques permettent de réduire cet espace de recherche,
- En général, il n'y a aucun moyen de contourner ce problème. Cependant, en pratique, de bonnes heuristiques permettent d’indiquer quelle partie explorée dans un arbre de recherche,
- Les heuristiques sont utilisées donc pour guider la recherche du chemin solution en ordonnant dynamiquement la liste des successeurs selon leur promesse de se rapprocher du but.
- Les algorithmes de recherche informés sont considérés comme des problèmes d'optimisation (plutôt que de simples problèmes de recherche tels que les algorithmes de recherche non informés). Chaque action est alors associé à un coût et leur intérêt principal se focalise sur le chemin solution qui minimise le coût global.

Propriétés des heuristiques

- Une heuristique est une connaissance spécifique au problème à résoudre, elle est indépendante de l'algorithme de recherche et non généralisable,
- Une heuristique est une règle d'estimation, une stratégie, une astuce, une simplification ou une autre règle permettant les choix non déterministes,
- Une heuristique permet de détecter grâce à une fonction d'évaluation le nœud qui semble potentiellement meilleur que les autres et par la suite ordonner la liste de ses successeurs,
- Une recherche exhaustive n'est pas réalisable pour des problèmes complexes (échecs,…). La notion de complexité conduit à la notion d'heuristiques,
- A la différence des algorithmes aveugles, les heuristiques sont tirées de l'expérience, d'une abstraction ou d'un apprentissage plutôt que d'une analyse scientifique.

Exemple: Fonction Heuristique et Carte de Roumanie



H(x)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Recherche Meilleur d'abord (Best First Search)

- Cette méthode est une combinaison entre la recherche en profondeur et en largeur
  - En profondeur : la solution est trouvée sans avoir besoin de développer tous les nœuds
  - En largeur : pas de risque d'être bloqué
- L'algorithme Recherche Meilleur d'abord permet d'explorer les nœuds dans l'ordre croisant/décroissant de leurs valeurs heuristiques,
- Il étend le nœud le plus prometteur selon sa valeur heuristique.
- Il exploite les caractéristiques de chaque état, afin d'estimer à quel point ce nœud est intéressant,
- Une fonction d'évaluation f est utilisée au cours de la recherche et associée à chaque nœud n: f(n),
- En général, la plus petite valeur de f(n) est choisie.
- Deux cas d'étude :
  - **Recherche meilleur d'abord glouton**
  - **Recherche A***

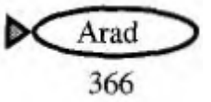
Méthode de recherche gloutonne (Greedy Best First)

- Greedy best-first Search développe le nœud le plus proche du but, il est susceptible de conduire à une solution rapidement.
- Cette recherche évalue les nœuds en utilisant uniquement la fonction d'évaluation f(n)= h(n). Elle correspond à l'estimation du coût à partir du nœud courant jusqu'au nœud but,
- Dans l'exemple de la carte de Roumanie, $h_{SLD}(n)$ = distance en ligne droite de n à Bucharest (Straight Line Distance).

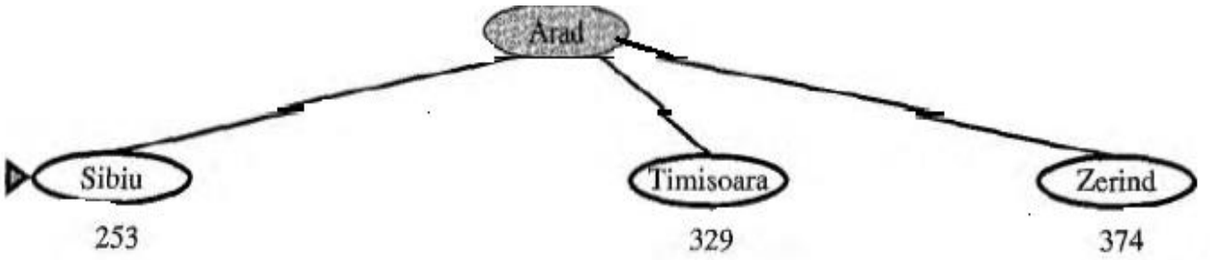
- **Critères d'évaluation**
  - **Complétude:** Non, risque de boucle infinie
  - **Complexité en temps** : $O(b^m)$, exponentielle en m cependant une bonne heuristique peut fournir une amélioration
  - **Complexité en espace**: tous les nœuds sont gardés en mémoire $O(b^m)$
  - **Optimalité:** Non

Arbre de recherche (Greedy Best First)

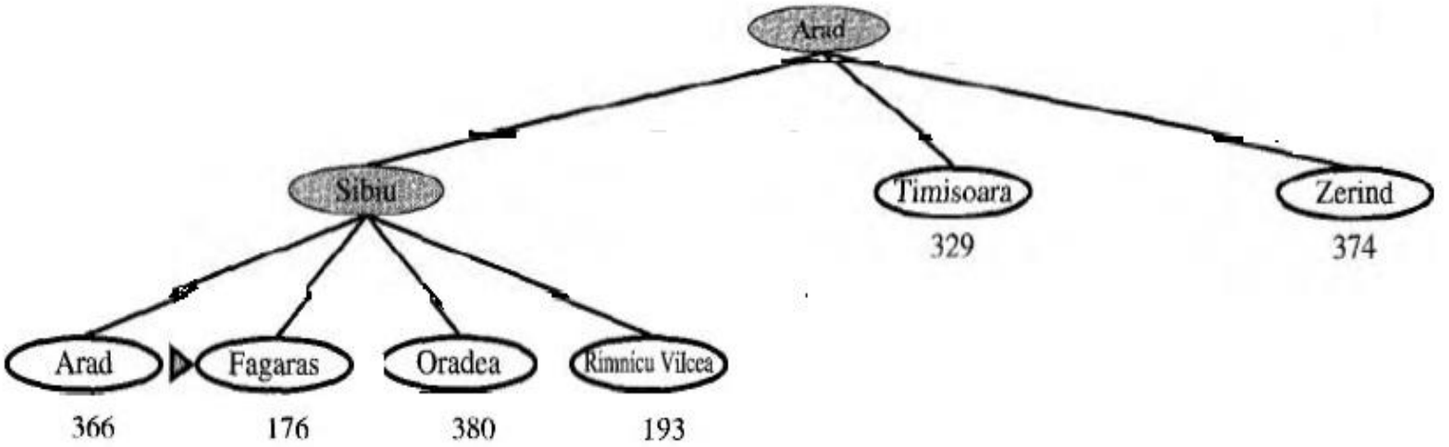
- Etat initial



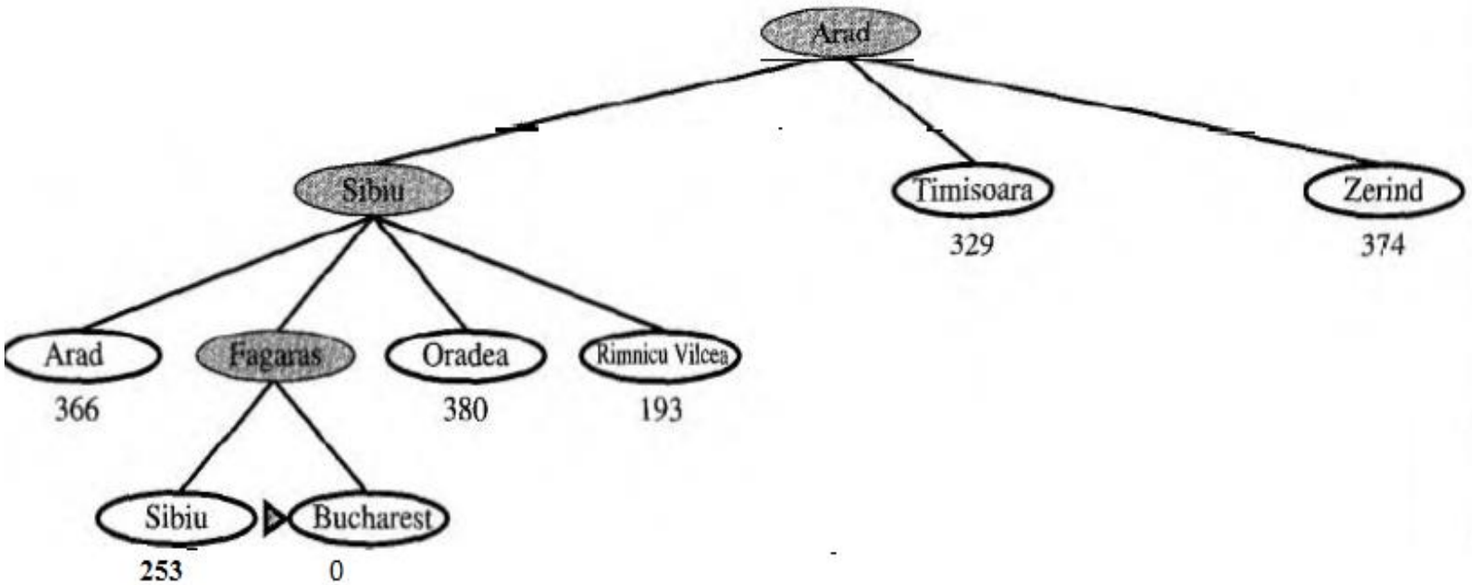
- Expansion de Arad



- Expansion de Sibiu



- Expansion de Fagaras



Algorithme Best First Search

```
Bestfs (entrée : init, sortie : etat)
Open = [init], Closed= [ ], etat = echec;
tantque (open <> [ ]) et ( etat <> succes)
Début
  X ← Suppeltgauche(Open)
  Si X = But alors etat = succes
  Sinon
    Succ ← generer(X)
    pour chaque Fils ϵ Succ faire
      case
        1. Fils n'est ni dans Open ni dans Closed:
           debut
             Fils ← valeur_heuristique,
             open ← open + Fils,
           fin;
        2. Fils est déja dans Open :
           Si Fils a atteint le plus court chemin alors
             Fils (open) ←plus court chemin
        3. Fils est déja dans Closed :
           Si Fils est arrivé avec chemin plus court alors
             Closed ← Closed – Fils,
             Open ← Open + Fils
           fin;
      fincase
    finpour
    Closed ← X; Reordonner la liste Open selon heuristique ( la meilleure
  la plus à gauche )
  Finsi
Fintantque;  retourner etat;
```

Recherche A star

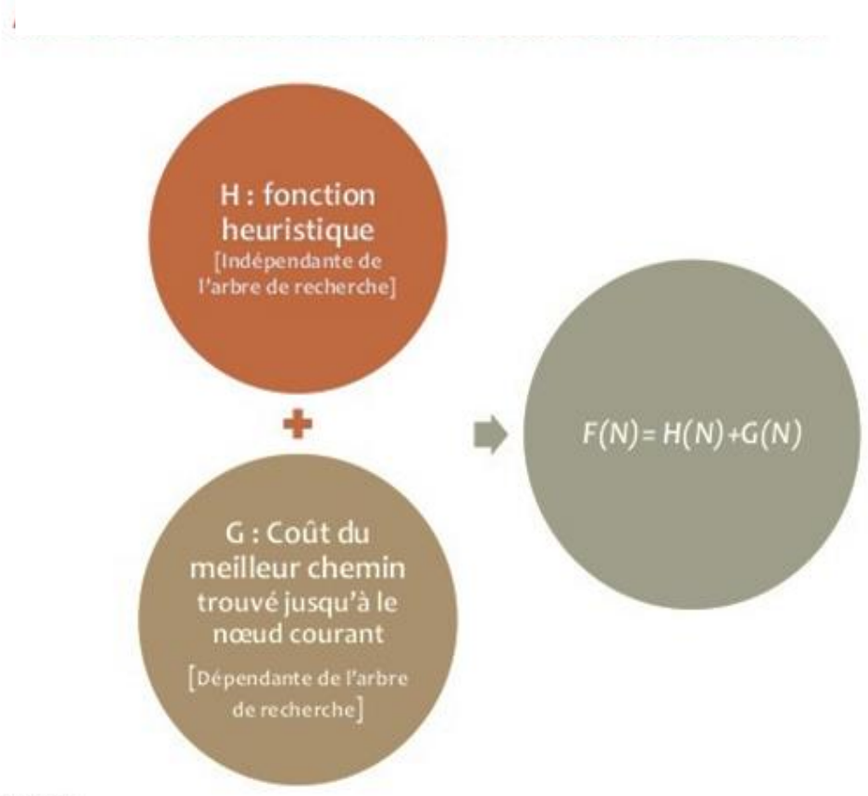
- L'algorithme A* est plus performant que n'importe quel autre algorithme puisqu'il réduit l'ensemble des sommets à explorer,
- Les algorithmes de recherche qui garantissent de trouver le chemin optimal sont appelés algorithmes admissibles, la fonction d'évaluation utilisée est donnée comme suit:

f(n) = g(n) + h(n) où :

g(n) : coût estimé de la distance du nœud initial vers un nœud donné n,

h(n) : fonction heuristique qui estime la distance du nœud n vers le nœud but. C'est le coût estimé du chemin le moins cher du nœud n à un nœud de but.

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



Principe:

La liste des nœuds sera ordonnée selon la valeur minimale de la fonction d'évaluation f,

Si on arrive à un nœud avec un chemin ayant un coût c et après d'autres exécutions de l'algorithme on arrive à redécouvrir le même nœud ayant un coût < c, on doit éliminer le premier chemin et ne considérer que le second.

Heuristique : Admissibilité

- h*(n): le véritable coût pour atteindre l'objectif à partir du nœud n,
- h(n): coût estimé du chemin le moins cher entre n et le nœud but,
- h(n) estimation de h*(n),
- h est une heuristique admissible si: $\forall n \; h(n) \leq h^*(n)$ (h sous estime h*) où h*(n) est le coût du nœud n au nœud but
- Solution optimale: la « meilleure » solution mesurée relativement à h,
- Si $h_2(n) \geq h_1(n)$ (les deux admissibles), alors h_2 domine h_1 et par conséquent h_2 est meilleure pour la recherche que h_1,
- h_2 est préférable à h_1 si elle conduit à explorer moins d'alternative que h_1 (h_2 est plus proche à h* que h_1)

➢ Exemples heuristique amissibles : Jeu du 8-Puzzle

- h**1**(n) = le nombre de pièces mal placées (par rapport à l'état final),
- h**2**(n) = la distance de Manhattan globale (la distance de chaque pièce en nombre de place de sa position finale).

h**1**(S) = 8;

h**2**(S) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2

où S est l'état initial

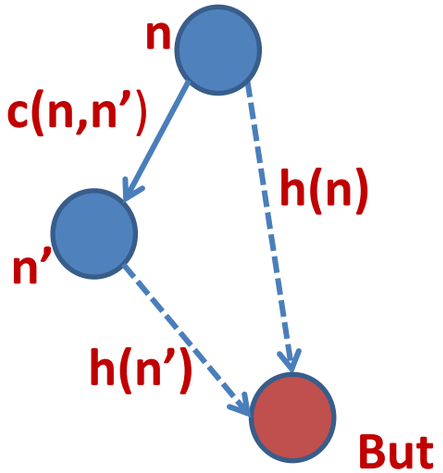
7	**2**	**4**
5		6
8	3	1

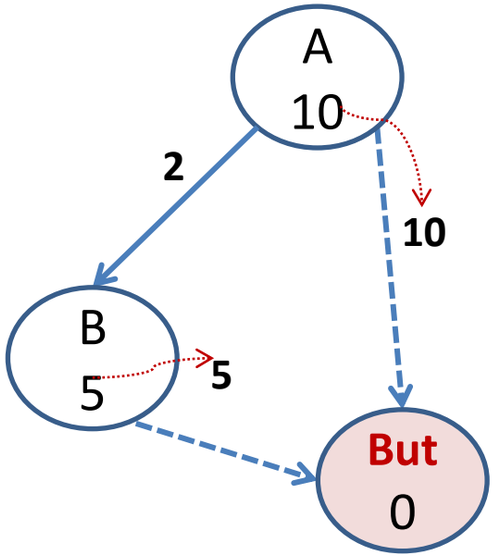
Etat initial

	1	**2**
3	4	5
6	7	8

Etat final

Propriétés de A*





h(A) = 10; coût(A,B) + h(B) = 2 + 5 = 7
⇒ h(A) > coût (A,B) + h(B) ⇒ h est inconsistance

- **A* génère une solution optimale si h(n) est une heuristique admissible**
- h(n) est **admissible** si elle ne surestime jamais le cout réel d'atteindre le noeud destination ou But (G):

$$\forall\, n\ h(n) \leq \text{Coût}(n, G)$$

- **A* génère une solution optimale si h(n) est une heuristique consistante,**
- h(n) est **consistante** si pour chaque noeud n et pour chaque noeud successeur n' de n, on a :

$$\forall\, n, n' \quad h(n) \leq \text{Coût}(n, n') + h(n') \text{ ou bien}$$

$$\forall\, n, \forall\, n' \ \ |h(n) - h(n')| \leq \text{Coût}(n, n')$$

où **Coût** (n, n') représente la distance reélle entre les noeuds n et n',

- **Si h(n) est consistante alors h(n) est admissible**

Recherche A*

Critères d'évaluation

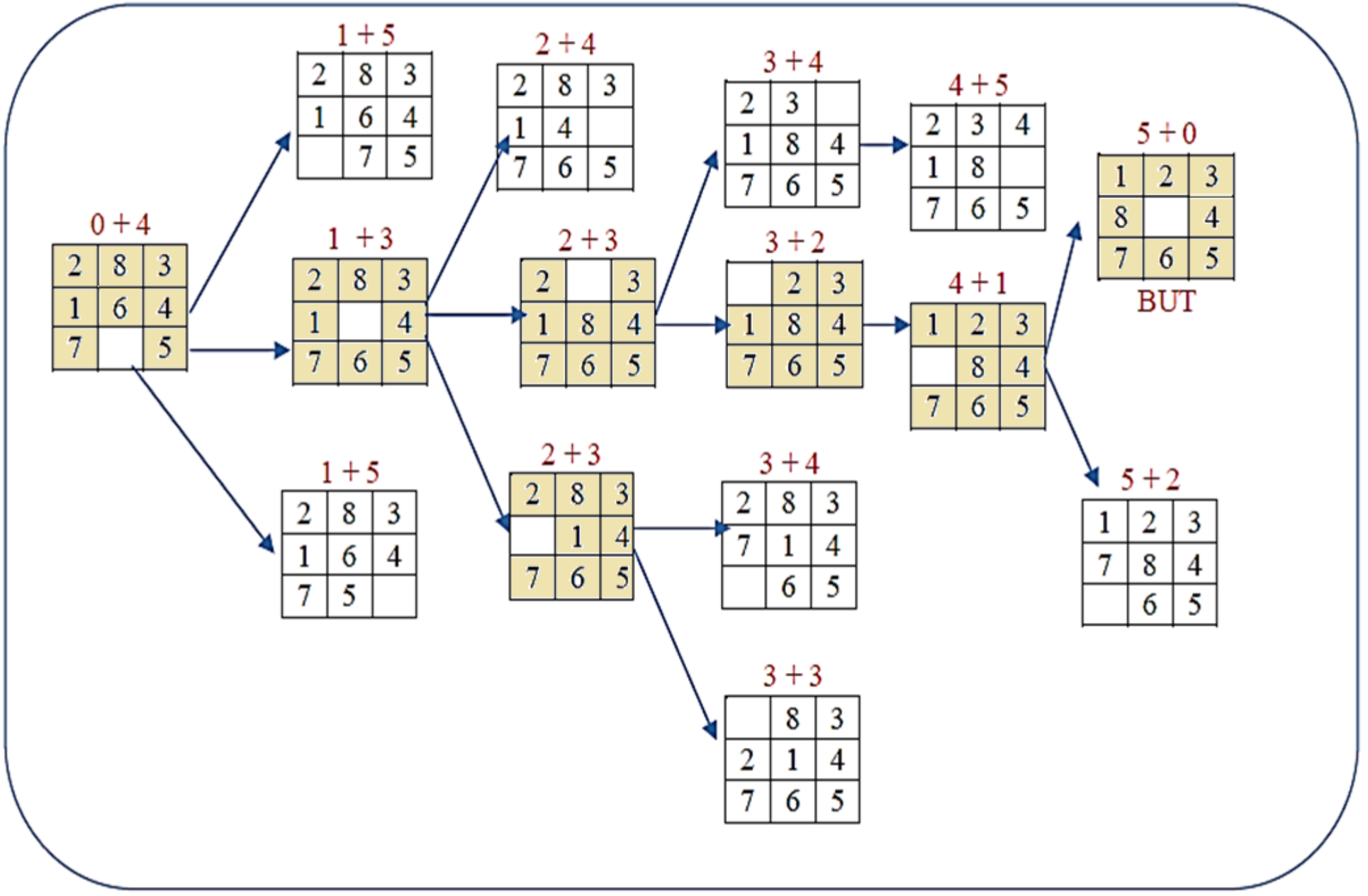
- **Complétude :** Oui, sauf s'il y a une infinité de nœuds
- **Complexité en temps :** exponentielle selon la longueur de la solution
- **Complexité en espace :** exponentielle (garde tous les nœuds en mémoire)
- **Optimalité :** Oui

Algorithme

X0 ← nœud_initial
Open ← X0 ; Closed ← Ø ; g(X0) ← 0, X ← X0;
Tant que [Open ≠ Ø] **faire**
  Open ← Open - X
  Closed ← Closed + X
  si X = But retourner la solution, stop,
  Pour chaque Y ∈ Succ(X) **faire**
    Si [Y ∉ (Open ∪ Closed) ou g(Y) > g(X) + coût(X, Y)] **alors**
      g(Y) ← g(X) + coût(X, Y)
      f(Y) ← g(Y) + h(Y)
      parent(Y) ← X
      Open ← Open + Y , dans l'ordre croissant de f
    Fin
  Fin
  Si Open ≠ Ø Alors X ← tête(Open)
Fin
Si Open = Ø **Alors** le problème n'admet pas de solution
Sinon retourner la solution

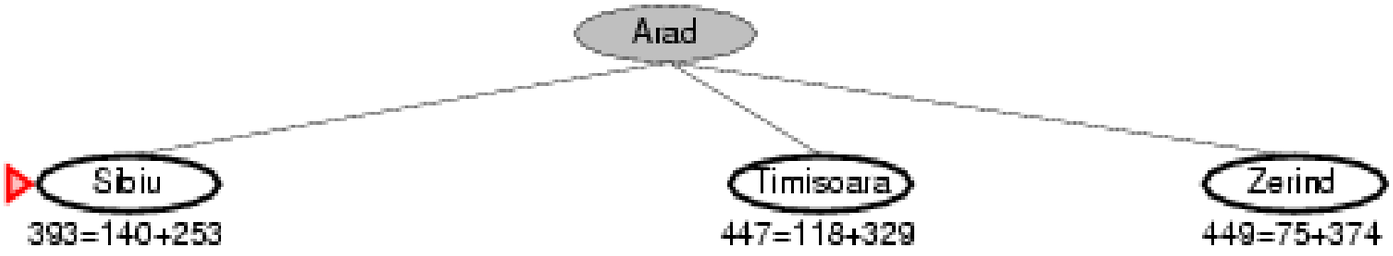
Exemple A*: 8-Puzzle

Chaque état est étiqueté par sa fonction d'évaluation f(n) = g(n) + h(n). Par exemple f(état_initial)=0+4, où g(état_initial) = 0 et h(état_initial) = 4 (représente le nombre de cases mal placées par rapport à l'état final. La fonction coût g est égale à 1 pour chaque action.

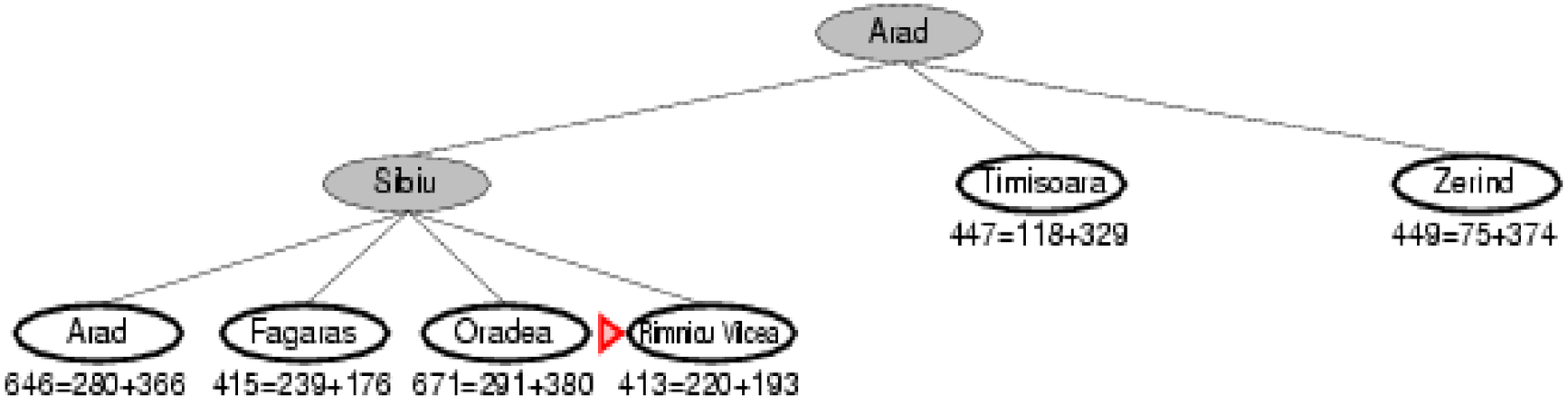


Exemple A* : Carte de Roumanie

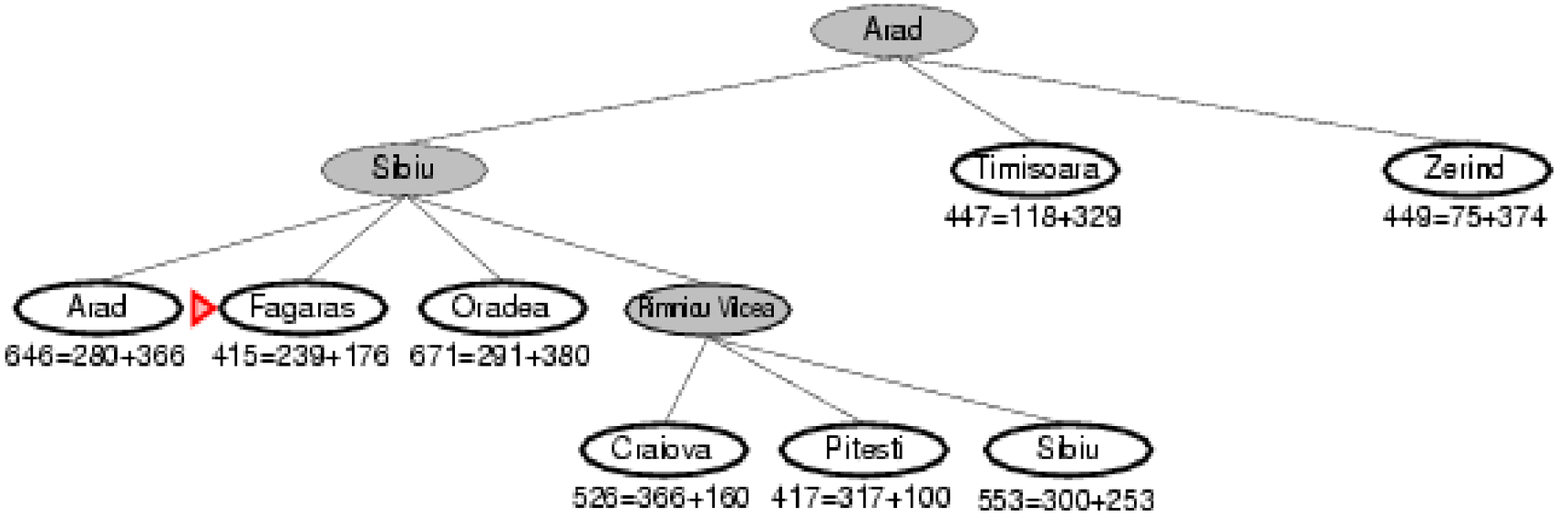
Exemple A* : Carte de Roumanie



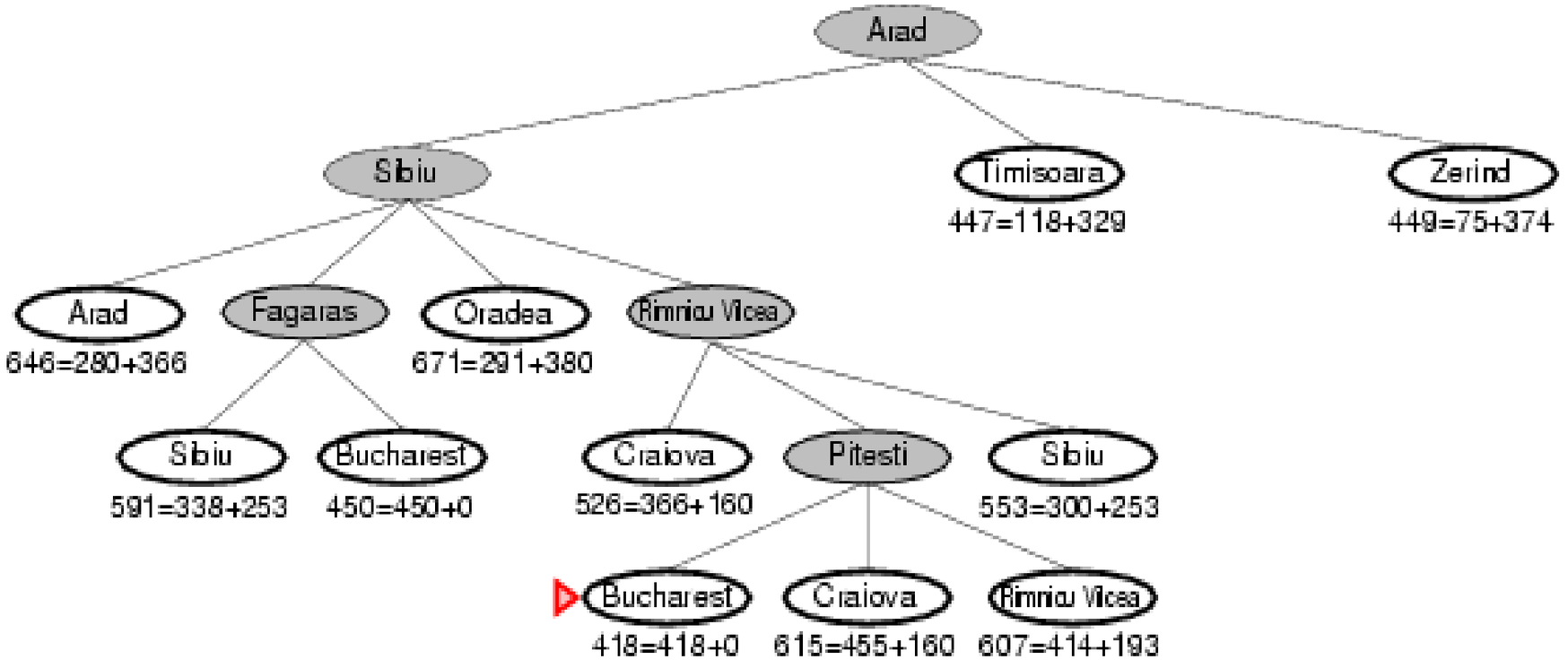
Exemple A* : Carte de Roumanie



Exemple A* : Carte de Roumanie



Exemple A* : Carte de Roumanie



Théorie des jeux

ALGORITHME MINIMAX ET ELAGAGE ALPHA/BETA

Introduction

MinMax est un algorithme, et plus généralement une stratégie, qui provient du domaine de la « Théorie des jeux » et a été identifié par Von Neumann il y a plus de 60 ans. Son amélioration la plus connue est l'algorithme «MinMax avec élagage AlphaBeta » qui est encore aujourd'hui à la base, dans des versions quelque peu améliorées, des programmes de jeu les plus évolués. **MiniMax,** et de son amélioration principale l'élagage **AlphaBeta,** permettant la sélection d'un coup dans les jeux dits « jeux à deux joueurs à somme nulle et information complète », famille dans laquelle se trouvent la plupart des jeux de réflexion (Othello, échecs, morpion, puissance 4, Go, etc) :

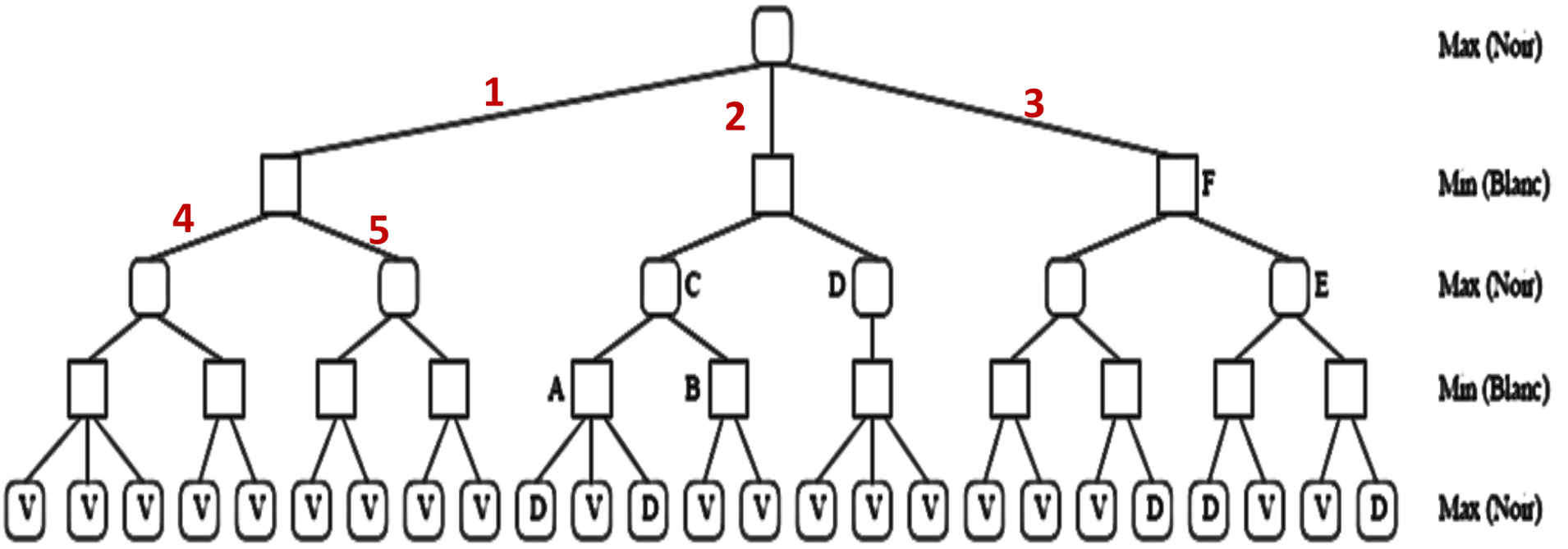
- Somme nulle : les gains d'un joueur sont exactement l'opposé des gains de l'autre joueur;
- Information complète : lors de sa prise de décision (i.e. du choix d'un coup à jouer dans le cas qui nous intéresse), chaque joueur connaît précisément
  - ses possibilités d'action ;
  - les possibilités d'action de son adversaire;
  - les gains résultants de ces actions.

Du fait de la présence de deux joueurs ayant des objectifs antagonistes, la recherche dans ces arbres ne peut se faire en utilisant des algorithmes de recherche de type **A***

Contrairement aux systèmes de résolution de problèmes, le planificateur ne dispose pas de la maitrise complète de l'enchainement des opérateurs, puisque des décisions extérieures à lui sont prises **par l'adversaire.** Il est impossible, dans la plupart des jeux de générer l'ensemble de tout l'espace d'état Un arbre de jeu pour les dames, plus difficiles que le Morpion, moins difficile que l'échec, est estimé à **1040** nœuds.

Exemple

Un arbre du jeu représente le déplacement de deux adversaires



L'arbre de recherche **MinMax** (de la Figure 1 ci-dessus) développé jusqu'à la fin de la partie**.** **V** ou **D** indique si la position terminale est une victoire de Noir ou bien une défaite. Un nœud Max correspond au cas où le joueur en question (ici, Noir) essaie de maximiser son gain alors qu'un nœud Min correspond au cas inverse.

L'arbre de cette figure indique que le **Noir** est gagnant de cette partie car il a la possibilité de jouer deux coups qui lui assureront de gagner quelles que soient les répliques du **Blanc**. Après une brève analyse de l'arbre de jeu, nous constatons que la troisième branche est une défaite (F) par contre les branches à gauche indiquent une victoire (V) assuré du Noir.

Suite de l'exemple

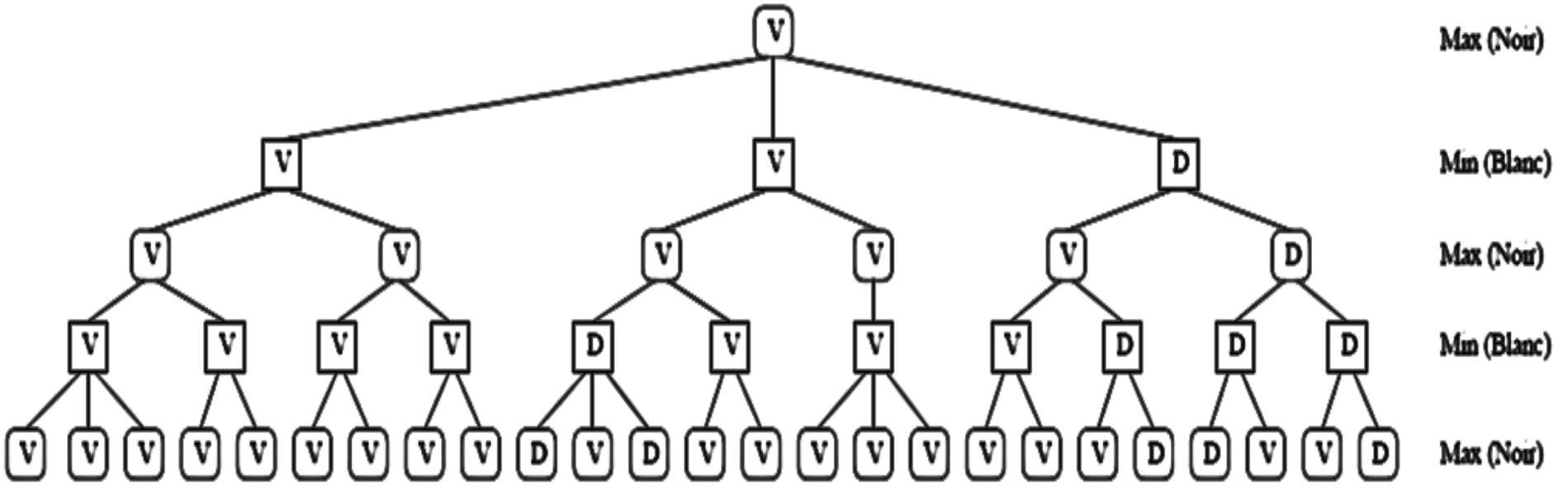


Figure 2 – Détail de l'algorithme MinMax sur l'arbre de la Figure 1.

La figure 2 illustre le réétiquetage des nœuds internes correspondant au raisonnement utilisé pour déterminer si la position est gagnante ou non. Chaque nœud Max choisit le maximum de ses fils et inversement pour chaque nœud Min.

Stratégie de recherche

L'analyse d'une situation ne soit entreprise qu'après développement du jeu sur plusieurs niveaux de coups et de réponses

► Si le développement se termine à une profondeur raisonnable, on peut comparer les situations terminales et déduire une option pour le coup suivant

► On doit limiter l'exploration à une profondeur maximale de résolution (Il est évident qu'une recherche dans un arbre à une profondeur plus grande permet généralement de choisir un coup de meilleur qualité que le coup retourné par un *MiniMax* utilisant une profondeur plus faible

► À cette profondeur on attribue aux (pseudo) feuilles une valeur correspondante à l'estimation de la position à travers une fonction d'évaluation (heuristique): Une fonction heuristique permet d'estimer s'il y a gain, perte ou match nul

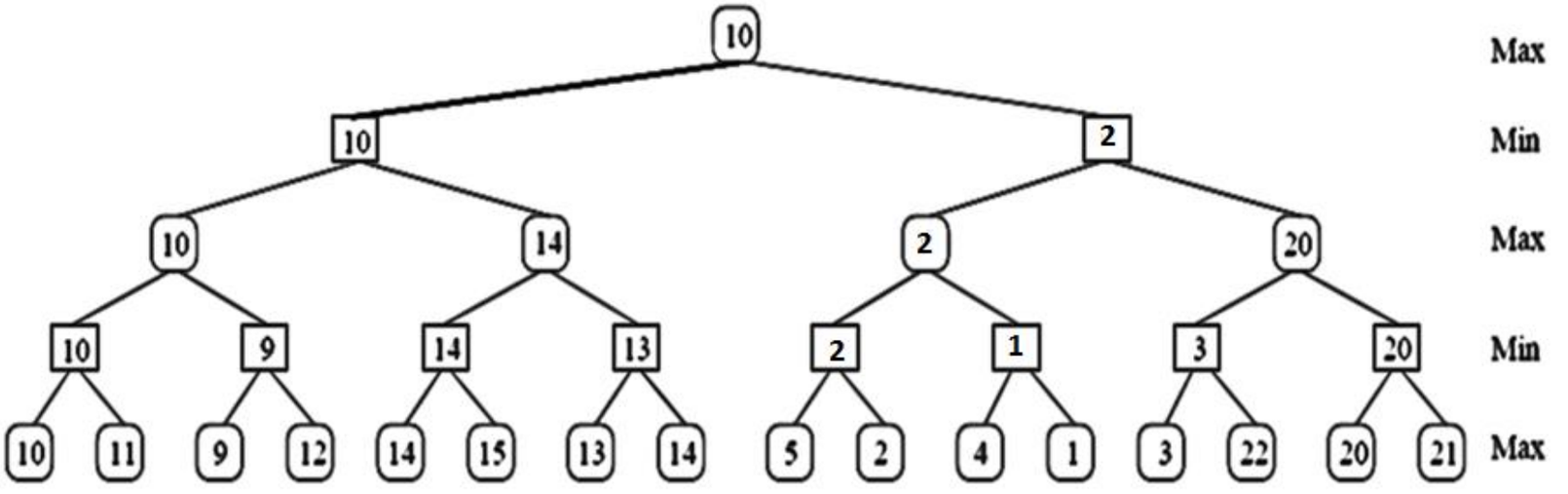


Figure 3 – Application de l'algorithme MinMax en utilisant les notes obtenues par une fonction d'évaluation.

Méthodologie de l'algorithme

1. Etendre l'arbre de jeu uniformément de l'état courant (où c'est à MAX de jouer) jusqu'à une profondeur d.
2. Calculer la fonction d'évaluation pour chacun des nœuds terminaux de l'arbre.
3. Remonter les valeurs des feuilles vers la racine de l'arbre de sorte que:
   - Un nœud MAX reçoive la valeur maximum des évaluations de ses successeurs.
   - Un nœud MIN reçoive la valeur minimum des évaluations de ses successeurs.
4. Sélectionner le coup menant à un nœud MIN ayant la valeur remontée.

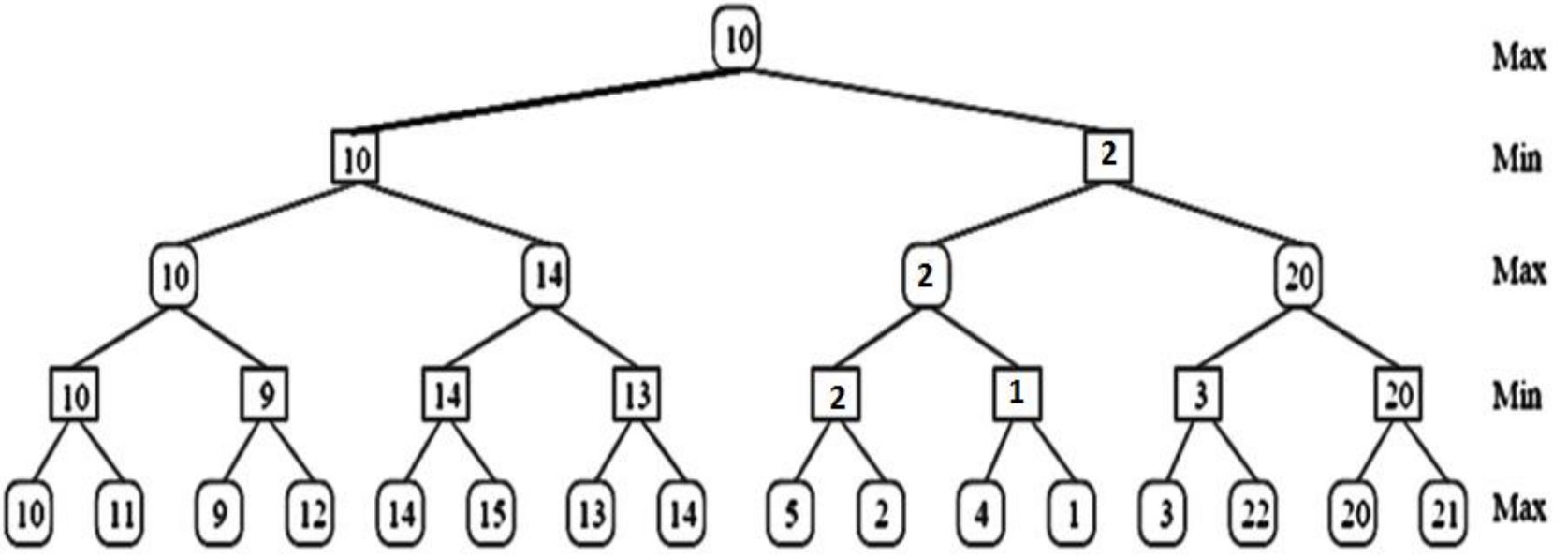
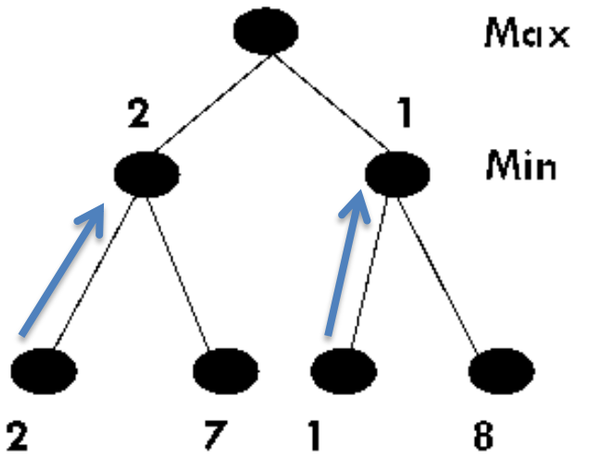
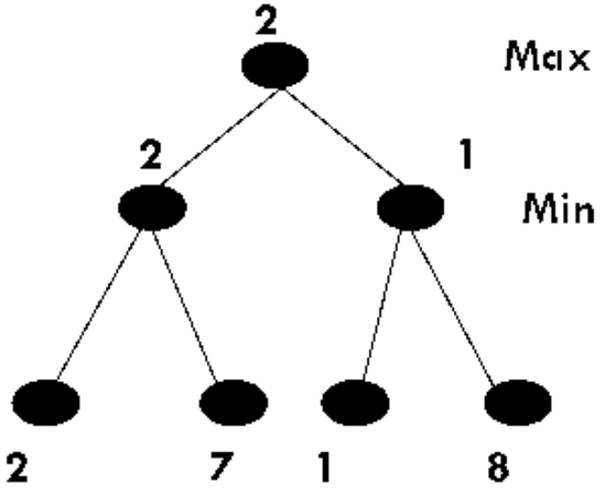


Figure 3 – Application de l'algorithme MinMax en utilisant les notes obtenues par une fonction d'évaluation.

Algorithme MiniMax





Fonction *MiniMax*(node, depth, playerMax)

Si depth = 0 **ou** node est nœud terminal **alors**
    return heuristic(node)

Si playerMax = True **alors**
    bestValue= -∞ (-infini)

    Pour chaque succ(node) **faire**
        score = *MiniMax*(succ(node), depth-1, False)
        bestValue= max(bestValue, score)

    Finpour

    return bestValue
Sinon
    Si playerMax = False **alors**
        bestValue= ∞

        Pour chaque succ(node) **faire**
            score = *MiniMax*(succ(node), depth-1, True)
            bestValue= min(bestValue, score)

        Finpour

        return bestValue

    FinSi
FinSi

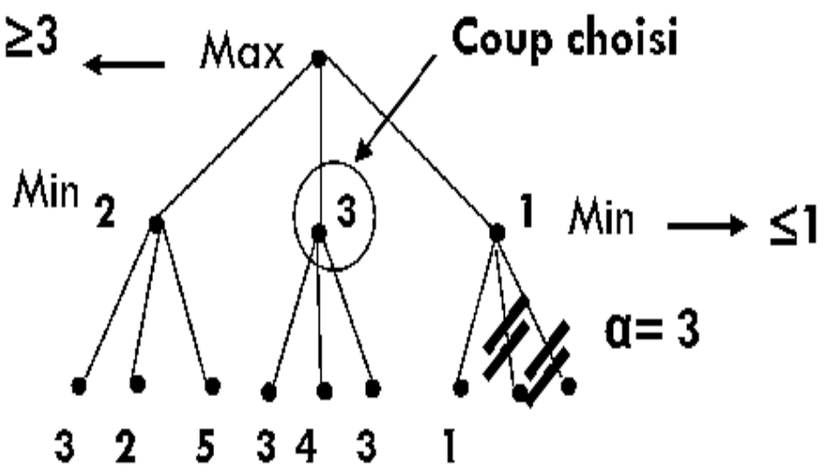
Elagage Alpha/Beta : Principe

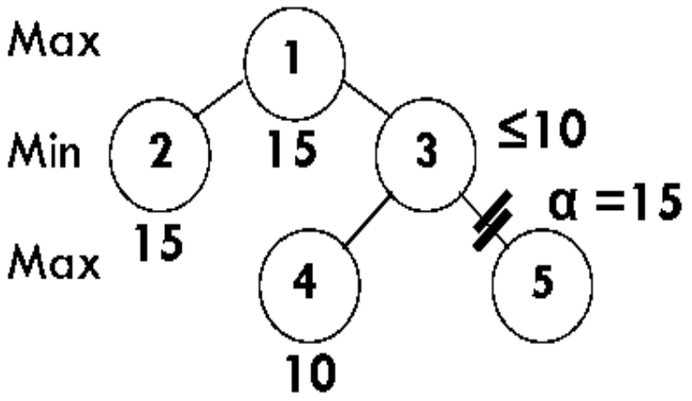
Problème de MiniMax : les informations ne circulent que dans un seul sens : des feuilles vers la racine. Il est ainsi nécessaire d'avoir développé chaque feuille de l'arbre de recherche pour pouvoir propager les informations sur les scores des feuilles vers la racine.

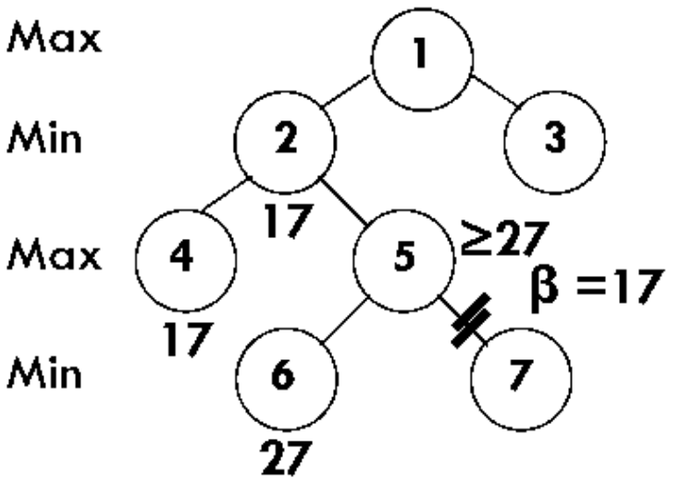
Principe α/β : Cette technique consiste à éviter la génération de feuilles et de parties de l'arbre qui sont inutiles :

- les deux variables α et β contiennent respectivement à chaque moment du développement de l'arbre *la valeur minimale* et *la valeur maximale* que le joueur peut espérer obtenir pour le coup à jouer étant donné la position où il se trouve. α est la borne inférieure attribuée au nœud max alors que β est la borne supérieure attribuée au nœud min.
- Certains développements de l'arbre sont arrêtés car ils indiquent qu'un des joueurs a l'opportunité de faire des coups qui violent le fait que α est obligatoirement la note la plus basse que le joueur Max sait pouvoir obtenir ou que β est la valeur maximale que le joueur Min autorisera Max à obtenir.
- L'α-β détermine la valeur MiniMax de la racine de l'arbre de jeu en traversant l'arbre dans un ordre prédéterminé (de gauche à droite) sautant tous les nœuds qui ne peuvent plus influencer la valeur MiniMax de la racine.

Algorithme Alpha/Beta







Fonction *AlphaBeta*(node, depth, α, β, playerMax)

Si depth = 0 **ou** node est nœud terminal **alors**
    return heuristic (node)

Si playerMax = True **alors**

    Pour chaque succ(node) **faire**
        score = *AlphaBeta* (succ(node), depth-1, α, β, False)
        α = max(α, score)
        Si α ≥ β **break**

    Finpour

    return α

Sinon

    Si playerMax = False **alors**

        Pour chaque succ(node) **faire**
            score = *AlphaBeta* (succ(node), depth-1, α, β, True)
            β= min(β, score)
            Si α ≥ β **break**
        Finpour

        return β

    FinSi

FinSi

Exemple1 : Elagage $\alpha\beta$

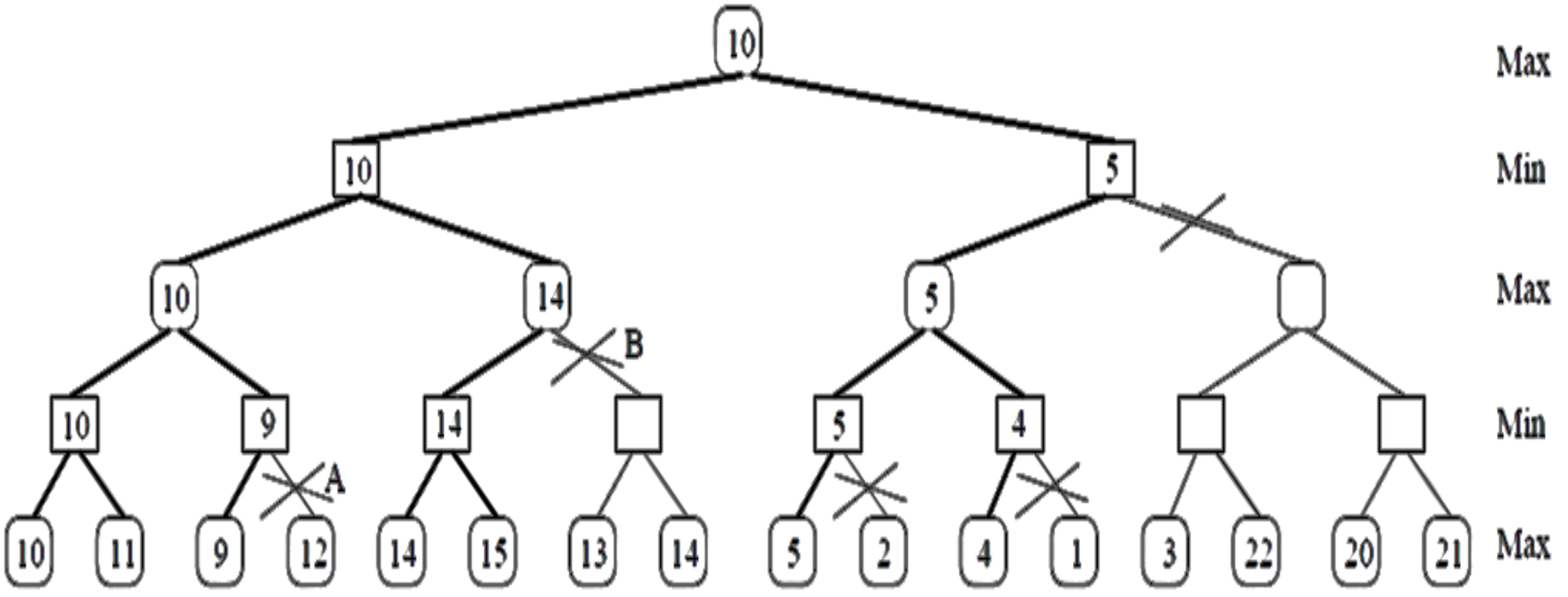
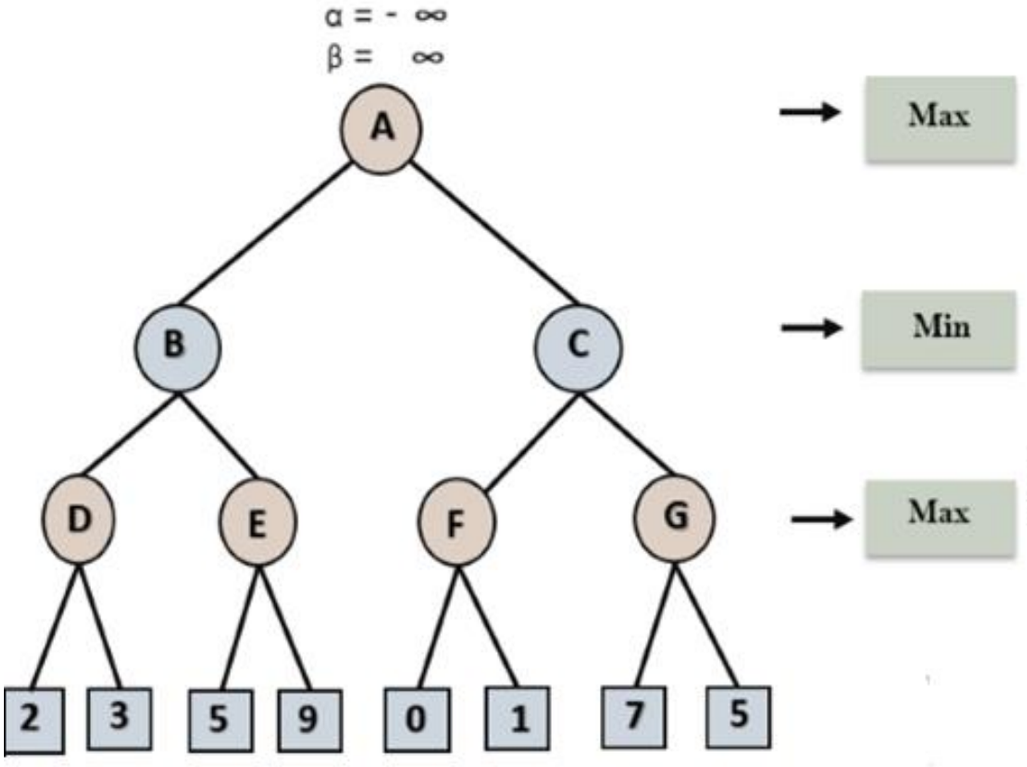


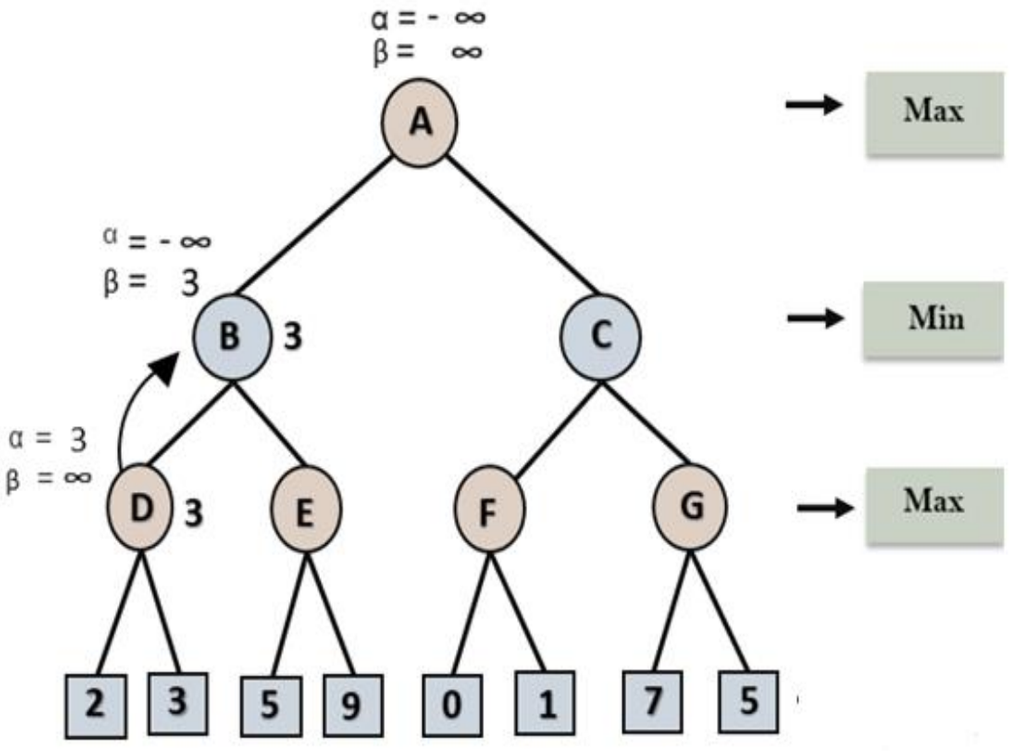
Figure 4 – Algorithme AlphaBeta : les branches développées sont en trait gras ; les autres parties de l'arbre sont celles développées par MinMax mais pas par AlphaBeta.

Exemple2 : Elagage αβ

1. Initialisation des valeurs α et β au niveau du nœud A:



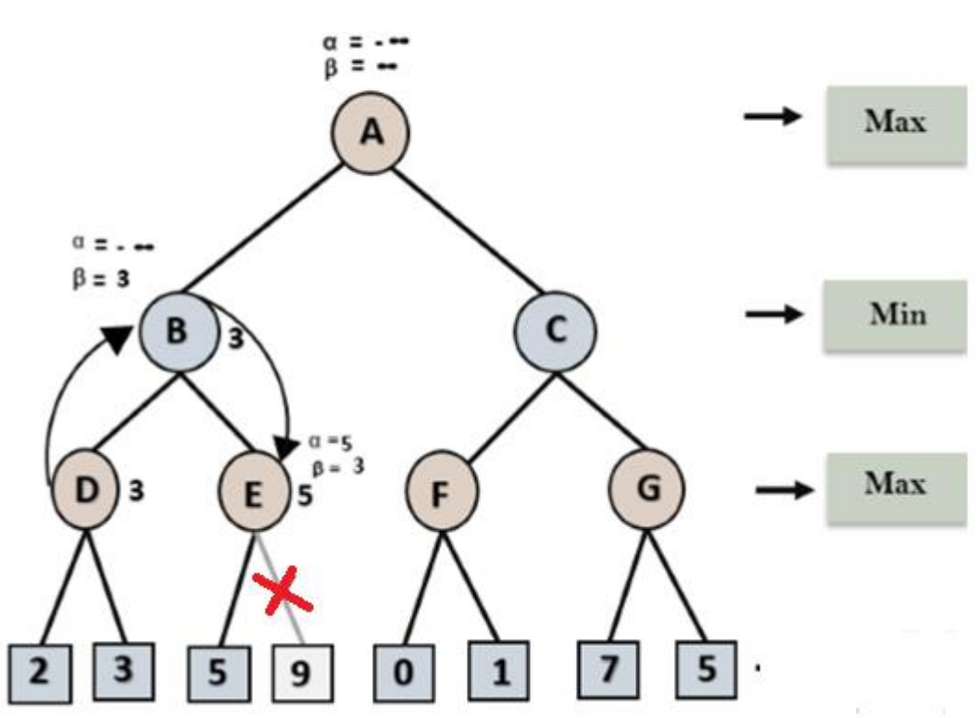
2. Calcul des valeurs α et β au niveau des nœuds D puis B:

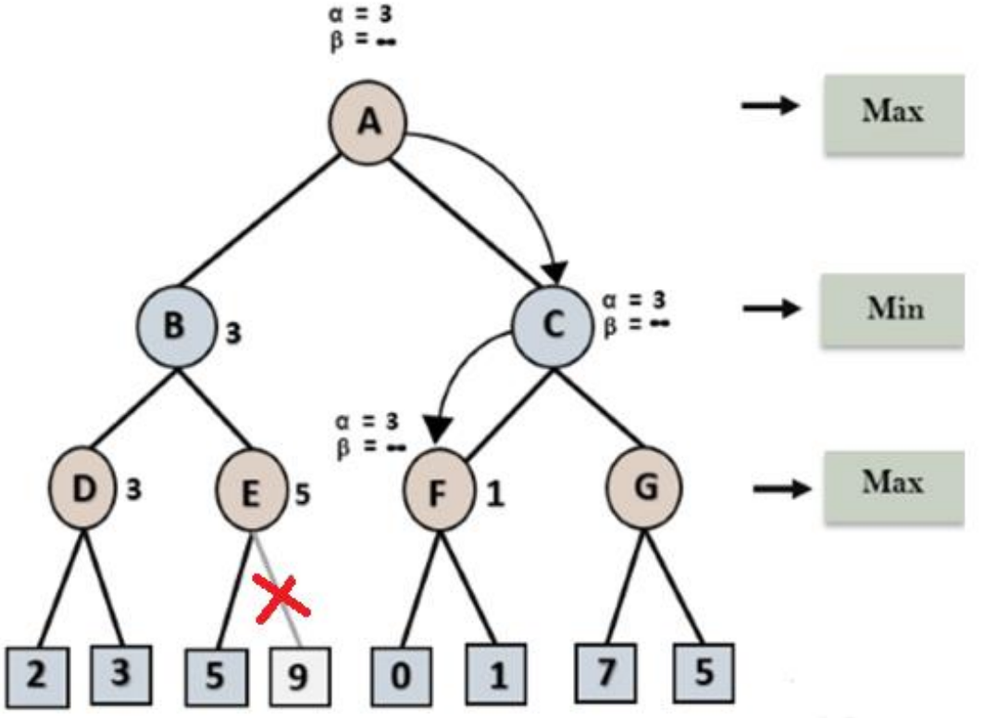


Exemple2 : Elagage αβ

3. Après calcul des valeurs α et β au niveau du nœud E, nous remarquons que $\alpha \geq \beta$, d'où coupure de type β.

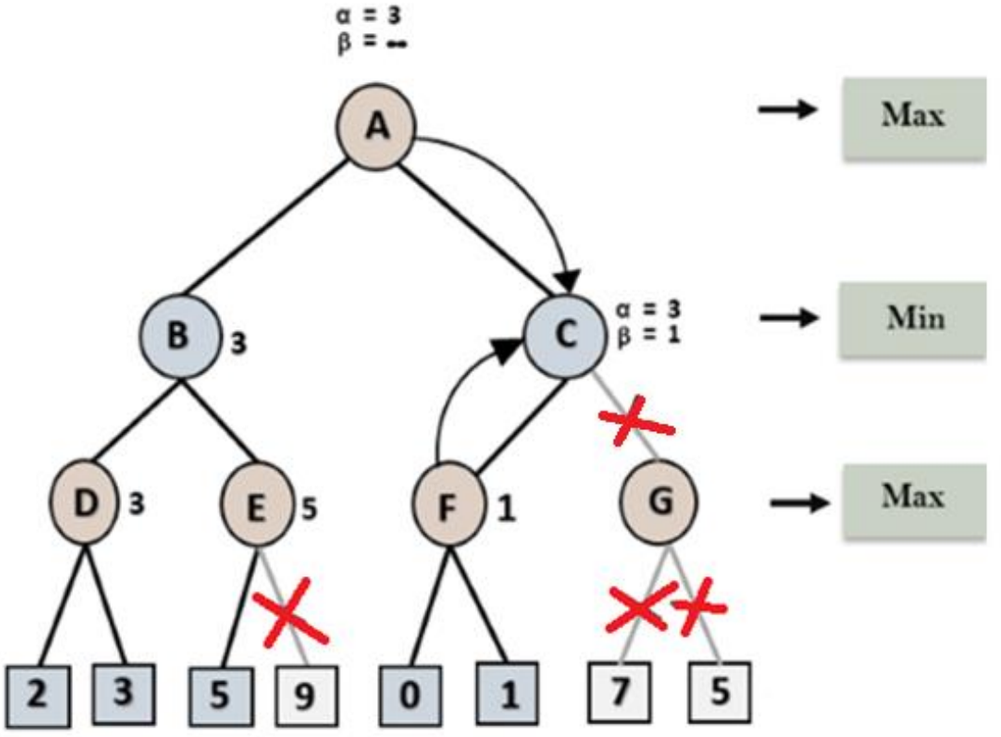
4. calcul des valeurs α et β au niveau du nœud E. La valeur maximale trouvée à cet instant est 3, d'où α = 3 au niveau des nœuds C et F.



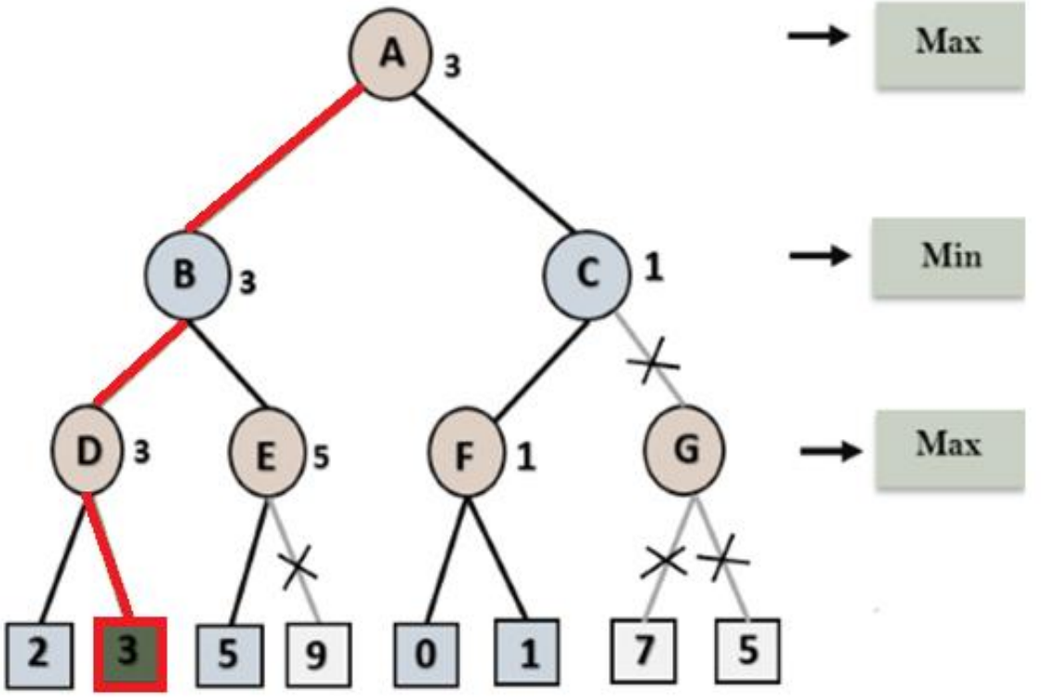


Exemple2 : Elagage $\alpha\beta$

5. Après calcul du maximum au niveau du nœud F, cette valeur est passée au nœud C d'où $\alpha \geq \beta$ donc coupure de type α.



6. Finalement, le coup choisi est celui qui mène vers la branche B avec un minimum de calcul.



Recherche Locale

ALGORITHME GÉNÉTIQUE

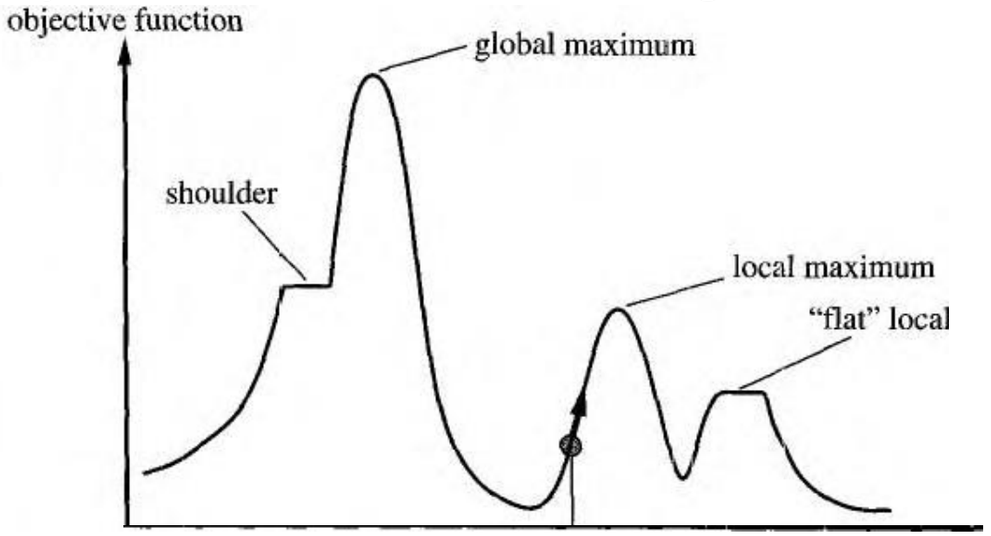
Introduction

Les algorithmes de recherche locale fonctionnent en utilisant seulement l'état actuel (plutôt que plusieurs chemins) et ils se déplacent généralement uniquement vers les voisins de cet état. En règle générale, les chemins suivis par la recherche ne sont pas conservés. Les algorithmes de recherche locale présentent deux avantages :

- ils utilisent très peu mémoire
- ils peuvent souvent trouver des solutions raisonnables dans des espaces d'états larges ou infinis (continus) pour lesquels les algorithmes classiques ne conviennent pas,

En plus de trouver des buts, les algorithmes de recherche locale sont utiles pour résoudre des problèmes d'optimisation purs, dans lesquels le but est de trouver le meilleur état selon une fonction objective. De nombreux problèmes d'optimisation ne correspondent pas au mode de recherche classique (informée et non informée).

Un algorithme de recherche locale complet trouve toujours un but s'il en existe et un algorithme optimal trouve toujours un minimum / maximum global.



Algorithme génétique

En 1975, Holland a décrit, dans son livre «Adaptation in Natural and Systems », comment appliquer les principes de l'évolution naturelle à des problèmes d'optimisation et construit les premiers algorithmes génétiques. L'objectif de Holland est de concevoir des systèmes artificiels ayant des propriétés similaires à celles du processus d'adaptation naturelle.

Maintenant les algorithmes génétiques (AG) se présentent comme un puissant outil de résolution de problèmes de recherche et d'optimisation. Parmi ses applications , l'emploi du temps, ordonnancement d'atelier de production, les jeux, etc.

Ces algorithmes dépendent de deux facteurs principaux, la fonction fitness et la fonction aléatoire, afin d'explorer la fonction objective à optimiser.

Dans les AG, nous avons une population de solutions possibles (générée d'une manière aléatoire) au problème donné. Ces solutions subissent une recombinaison et une mutation produisant de nouveaux enfants. Ce processus se répète sur différentes générations. A chaque individu (ou solution candidate) est attribué une valeur d'adaptation appelée « fitness » (généralement basée sur la fonction objective) et les meilleurs individus auront une plus grande chance de produire de nouveaux individus ayant des caractéristiques plus intéressantes.

De cette façon, les meilleurs individus ou solutions continuent à évoluer au fil des générations, jusqu'à atteindre un critère d'arrêt.

Domaines d’applications

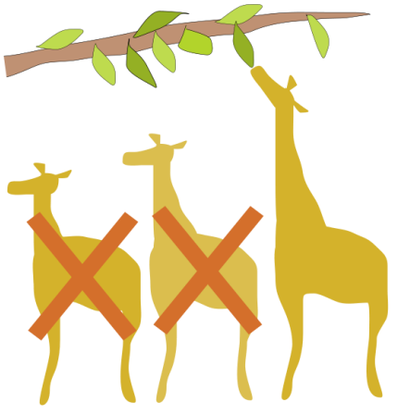
Les AG ont été utilisés pour résoudre un grand nombre de problèmes dans différents domaines :

- **Optimisation -** Les algorithmes génétiques sont le plus couramment utilisés dans les problèmes d'optimisation dans lesquels nous devons maximiser ou minimiser une valeur de fonction objective donnée sous un ensemble donné de contraintes.
- **Économie -** Les AG sont également utilisées pour caractériser divers modèles économiques comme le modèle de la toile d'araignée, la résolution de l'équilibre de la théorie des jeux, la tarification des actifs, etc.
- **Réseaux de neurones -** Les AG sont également utilisés pour entraîner les réseaux de neurones, en particulier les réseaux de neurones récurrents.
- **Parallélisation -** Les AG ont également de très bonnes capacités de parallélisme, se révèlent être des moyens très efficaces pour résoudre certains problèmes, et constituent également un bon domaine de recherche.
- **Traitement d'image -** Les AG sont utilisés pour diverses tâches de traitement d'image numérique,
- **Problèmes de routage des véhicules -** Une flotte de véhicules de capacité prédéfinie, basée dans un dépôt, doit assurer des tournées entre plusieurs clients (ou villes) ayant demandé chacun une certaine quantité de marchandises.

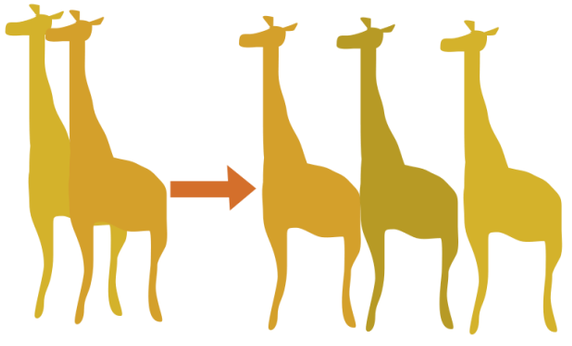
Principes de l'évolution

Les algorithmes génétiques utilisent la théorie de Darwin sur l'évolution des espèces. Elle repose sur trois principes.

Principe de variation: La variation fait référence aux différences indiquées par l'individu d'une espèce et aussi par les descendants des mêmes parents.



Principe d'adaptation: Les individus les plus adaptés à leur environnement atteignent plus facilement l'âge adulte. Ceux ayant une meilleure capacité de survie pourront donc se reproduire d'avantage.

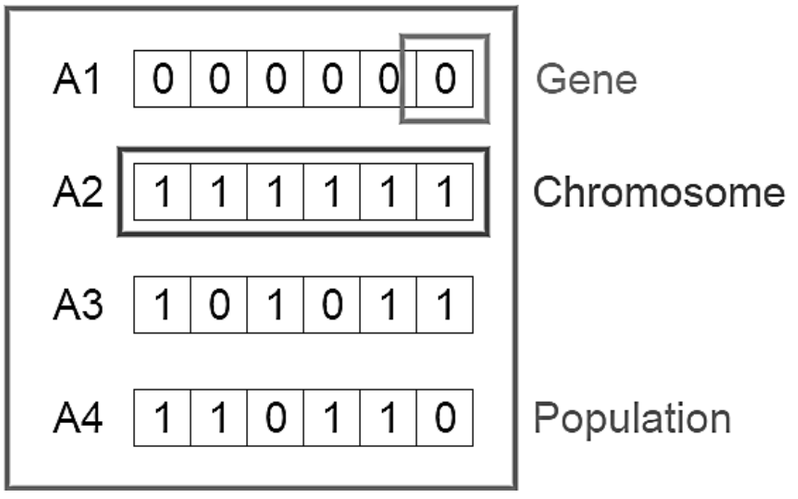


Principe d'hérédité: Les caractéristiques des individus sont transmises à leur descendance. Ce mécanisme permettra de faire évoluer l'espèce pour partager les caractéristiques favorables à sa survie.

Algorithme génétique : Terminologie & Représentation

La terminologie de base nécessaire pour comprendre les AGs sont:

- **Population** - C'est un sous-ensemble de solutions possibles (codées en chaines de caractères et générées aléatoirement) au problème donné.
- **Chromosomes** - Un chromosome (individu) est l'une de ces solutions au problème donné.
- **Gène** - Un gène est une position d'élément d'un chromosome.



- **Fitness :** est une fonction d'évaluation qui mesure la performance d'une solution donnée. Dans certains cas, la fonction de fitness et la fonction objective peuvent être identiques, tandis que dans d'autres, elles peuvent être différentes et dépend du problème.
- **Codage Binaire :** Il s'agit de l'une des représentations les plus simples et les plus largement utilisées dans les AGs. Dans ce type de représentation, le génotype se compose de chaine de bits.
- **Codage entier ou réel :** L'espace de solutions ne se limite pas à une représentation binaire, dans certains problèmes la représentation entière ou réelle est préférable.

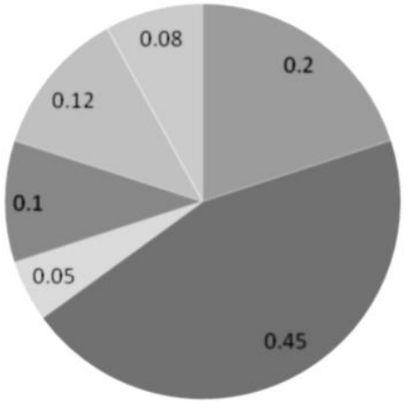
1	4	3	7	1	2

0,2	0,1	0,7	0,4	0,2	0,1

Algorithme génétique: Opérateurs

Les opérateurs génétiques font évoluer les populations de solutions de manière progressive. Ils se résument à:

1. **Sélection :** les individus les plus adaptés sont choisis pour la reproduction tandis que les moins adaptés meurent avant la reproduction, ce qui améliore globalement l'adaptation. Il existe plusieurs techniques:
   - **Probabilité de sélection proportionnelle à l'adaptation (Roulette Wheel Method):** pour chaque individu, la probabilité d'être sélectionné est proportionnelle à son adaptation au problème. Afin de sélectionner un individu, on utilise le principe de la roue de la fortune. chaque individu est représenté par une portion proportionnelle à son adaptation (fitness). La roue tourne N fois, où N est le nombre d'individus dans la population. À chaque tour, l'individu sous le marqueur de la roue est sélectionné pour faire partie du groupe de parents de la prochaine génération.

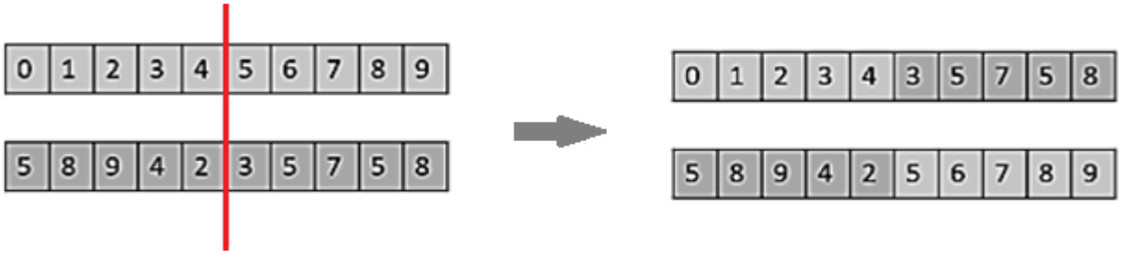


Individu	1	2	3	4	5	6
Fitness	0.2	0.45	0.05	0.1	0.12	0.08

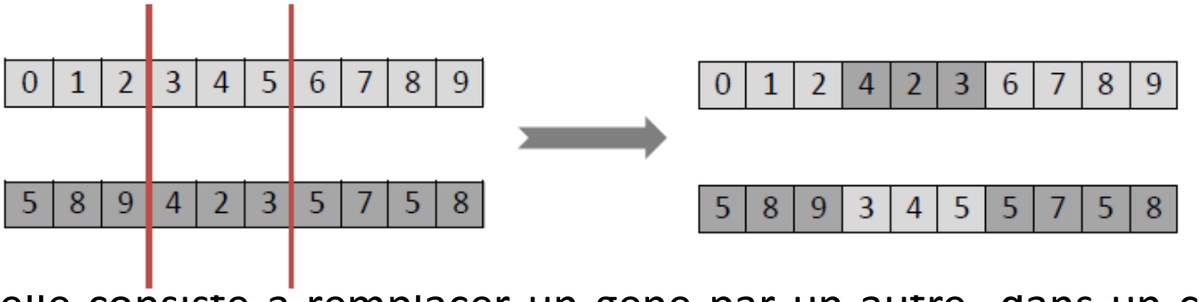
   - **Sélection par tournoi :** « K » individus de la population sont choisis au hasard et les meilleurs d'entre eux sont sélectionnés pour devenir parents. Le même processus est répété pour sélectionner le parent suivant.
   - **Sélection par rang :** elle est utilisée lorsque les individus de la population ont des valeurs de fitness très proches. Un rang est associé à chaque individu de la population selon sa valeur de fitness. Les individus les mieux classés sont préférés et ils sont choisis pour devenir parents.

Algorithme génétique: Opérateurs

2. **Croisement :** c'est une combinaison de deux chromosomes (individus) qui échangent des parties de leurs codes, pour donner de nouveaux chromosomes. Cette opération peut être simple ou multiple. Le point de sélection est choisi aléatoirement. Une probabilité de croisement est également introduite afin de donner la possibilité à un individu d'appliquer ou non le processus de croisement.
   - Croisement simple (crossover one point)

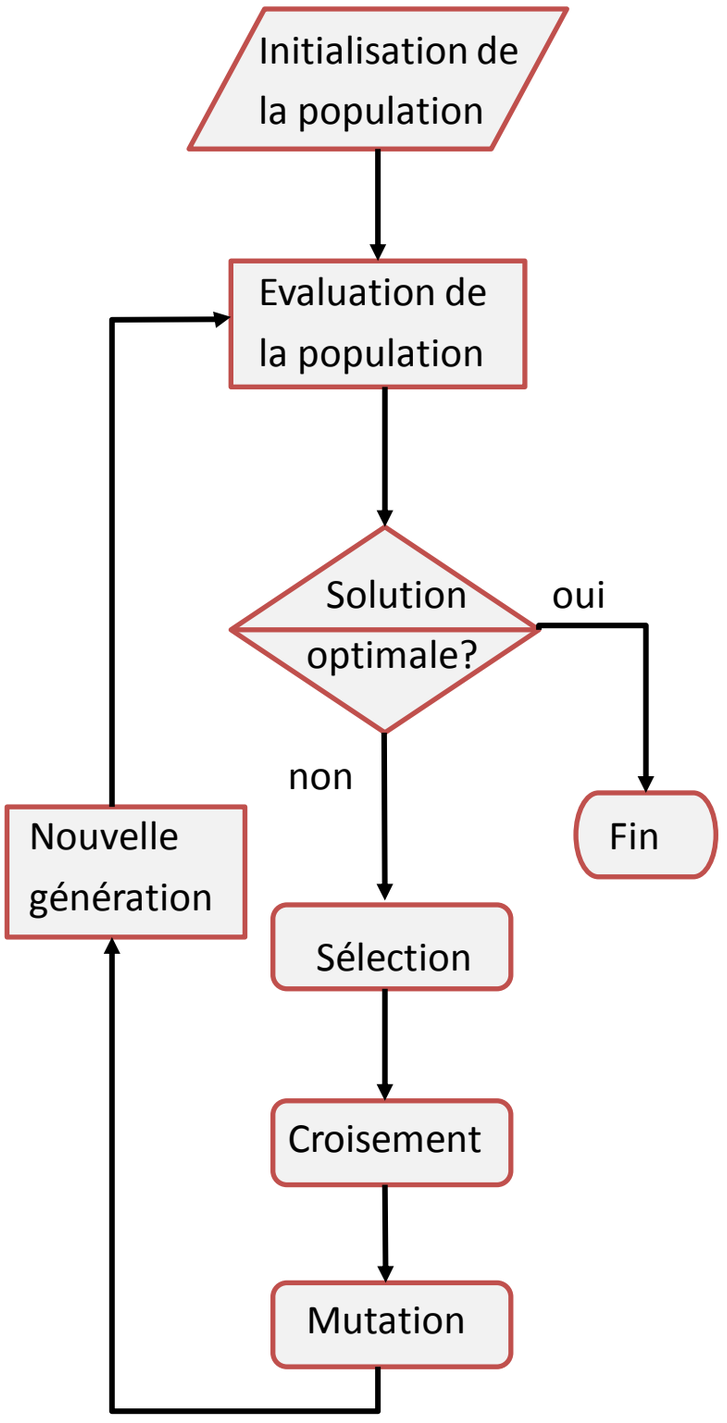


   - Croisement à 2 points (crossover two points)



3. **Mutation :** elle consiste à remplacer un gene par un autre dans un chromosome. Un taux de mutation est défini lors des changements de population qui est généralement compris entre 0,001 et 0,01 (Ce facteur est la probabilité qu'une mutation soit effectuée sur un individu). La mutation sert à éviter la convergence vers un extremum local.

Algorithme génétique : Pseudo-code



Fonction AlgoGenetique(Population: Pop, Fitness: F)

n : *nombre d'individus dans pop*

probc, probm : *probabilités de croisement et de mutation*

Début

Tantque *Non Convergence* faire

NouvPop ← ∅

Pour *i* = 1 jusqu'à *n* faire

par_1 ← Selection(*Pop*, F)

par_2 ← Selection(*Pop* - par_1 , F)

enf ← Croisement(par_1, par_2, *probc*)

prob ← GenerAlea( )

Si *prob* ≤ *probm* alors *enf* ← Mutation(*enf*)

NouvPop ← *NouvPop* + *enf*

FinPour

Pop ← *NouvPop*

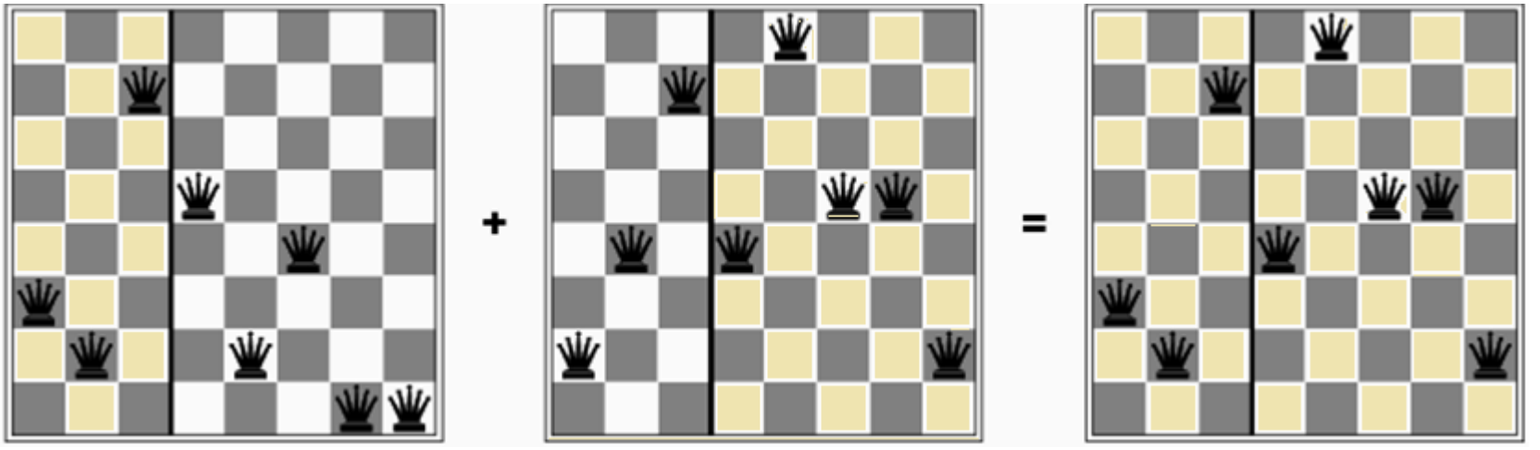
FinTQ

retourner (*meilleur individu*)

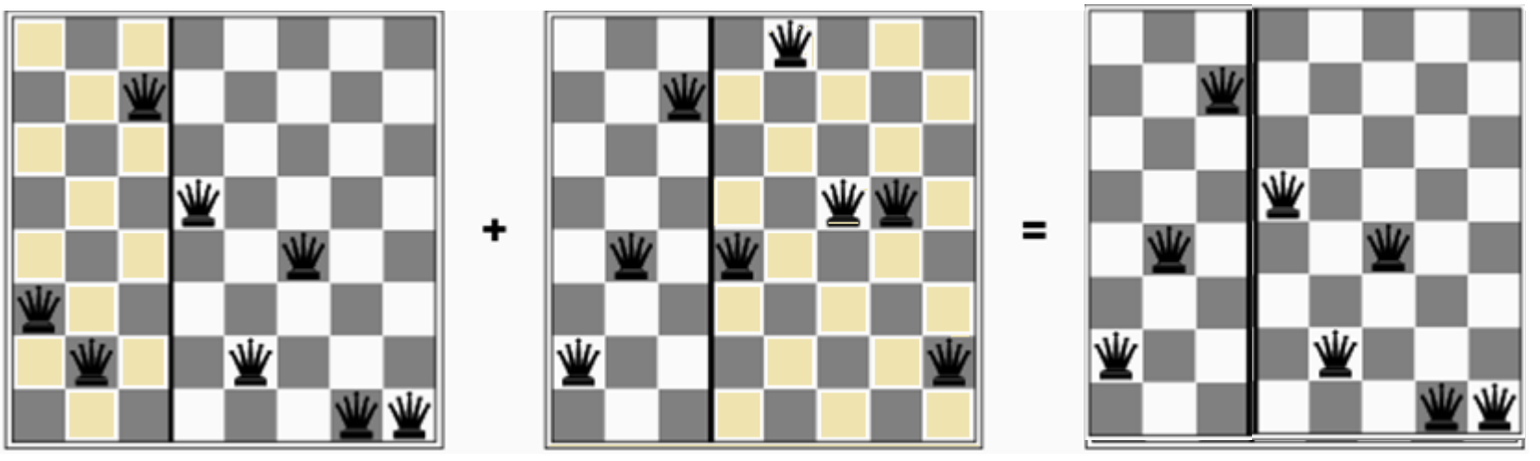
Fin

Exemple des 8-reines

- Dans cet exemple, le croisement du parent1 et du parent2 a donné naissance à 2 nouveaux enfants (chromosomes),
- Un parent est représenté par une chaine de caractères de type entier allant de 1 jusqu'à 8. Cet individu définit les positions des 8 reines.

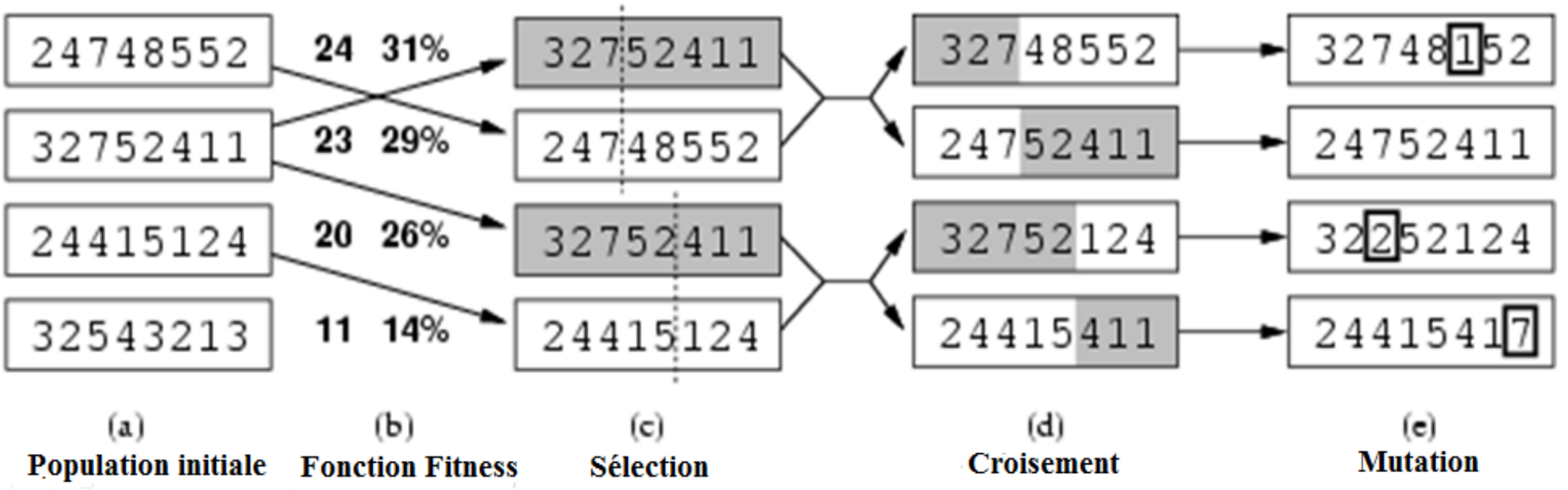


6 7 2 | 4 7 5 8 8 + 7 5 2 | 5 1 4 4 7 = 6 7 2 | 5 1 4 4 7



6 7 2 | 4 7 5 8 8 + 7 5 2 | 5 1 4 4 7 = 7 5 2 | 4 7 5 8 8

Exemple des 8-reines



- **Fonction de Fitness** : nombre de paires de reines qui ne s'attaquent pas (min = 0, max = 8 × 7/2 = 28
- **Probabilité de sélection** est donnée par: $P_i = {}^{F_i}/_{\sum_i F_i}$ où F_i désigne la fonction fitness associée à chaque individu. Cette probabilité est proportionnelle à l'adaptation :
  - P_1 = 24/(24+23+20+11) = 31% ,
  - P_2 = 23/(24+23+20+11) = 29% ,
  - P_3 = 20/(24+23+20+11) = 26%
  - P_4 = 11/(24+23+20+11) = 14%

Dans cet exemple, on remarque parmi l'ensemble de la population un seul individu est sélectionné 2 fois tandis qu'un autre a été écarté vu son faible pourcentage . Pour chaque paire d'individus sélectionnée, pour reproduction, un point de croisement est choisi au hasard parmi les positions de la chaîne de valeurs. Plusieurs autre choix de sélection seraient valides.

Dans le problème des 8-reines, le processus de mutation correspond à choisir une reine au hasard et à la déplacer sur une case aléatoire dans sa colonne.

Exemple : Problème du voyageur du commerce

Ce problème (Travelling Salesman Problem) concerne les trajets d'un voyageur de commerce en passant par plusieurs villes. Pour cela, il doit visiter chaque ville une seule fois et revenir à la ville de départ. Le but de l'algorithme est d'optimiser le trajet de façon que celui-ci soit le plus court possible.

L'implémentation de l'algorithme génétique considère les villes comme des gènes et le score de fitness est défini comme la longueur du chemin décrit par le gène (les villes visitées).

Soient 5 villes notées {0, 1, 2, 3, 4}. Initialement, le voyageur est dans la ville 0 et il doit trouver le chemin le plus court pour traverser toutes les villes et retourner à la ville 0. Un chromosome représentant le chemin choisi est donné par :

0	1	4	2	3	0

Ce chromosome subit une mutation. Pendant ce processus, la position de deux villes dans le chromosome est échangée pour former une nouvelle configuration, à l'exception de la première et de la dernière cellule, car elles représentent le point de départ et d'arrivée.

Le Chromosome initial avait un chemin de longueur «L». Supposons que le chemin entre la ville 1 et la ville 4 n'existait pas. Après mutation, le nouvel enfant obtenu a un chemin de longueur égale «L'», où L' < L, solution plus optimisée que l'hypothèse originale. C'est ainsi que l'algorithme génétique optimise les solutions relatives aux problèmes difficiles.

Limites des algorithmes génétiques

- **Temps de calcul long :** Les algorithmes génétiques requiert de nombreuses itérations, ainsi que l'utilisation excessive de la fonction d'évaluation,
- **Choix des paramètres :** la taille de la population ou le taux de mutation sont parfois difficiles à déterminer,
- **Choix de la fonction d'évaluation :** il est primordial de la définir avec soin pour prendre en compte tous les paramètres du problème. Sa complexité doit être optimale,
- **Incertitude des résultats :** il est impossible d'être assuré que la solution obtenue est la meilleure malgré le nombre important de générations. On est juste sûr de s'approcher de la solution optimale,
- **Risque des optimaux locaux :** ce problème est le résultat d'une convergence prématurée qui écarte les autres solutions potentielles.