University Mohamed Boudiaf of M'sila
Faculty of Mathematics and Informatics

Introduction à l'Intelligence Artificielle

Chapitre IV: Stratégies de Recherche

Dr. SAID KADRI

Associate Professor

Department of Computer Science, Faculty of Mathematics and Informatics, University Mohamed Boudiaf of M'sila

E-mail: kadri.said28@gmail.com

Website: https://kadrisaid28.wixsite.com/sgadri

2017 - 2018

Préface

- L’un des composants principaux d’un système intelligent est le moteur d’inférence (plus la BC).
- Un moteur d’inférence est caractérisé par :

1. **Un cycle de base qui comprend 04 phases:**
   - La phase de sélection (rassembler, trier les faits et les règles de la BC ayant un rôle spécifique)
   - La phase de filtrage : déterminer quelles règles peuvent être applicables parmi celles de la BC
   - La phase de résolution de conflit : choisir une règle à appliquer selon une stratégie de choix
   - La phase d’exécution : appliquer la règle choisie

2. **Une méthode de chaînage :**
   - Chaînage en avant
   - Chaînage en arrière
   - Chaînage mixte

3. **Une stratégie de recherche**
   - Une stratégie aveugle
   - Une stratégie informée (heuristique)

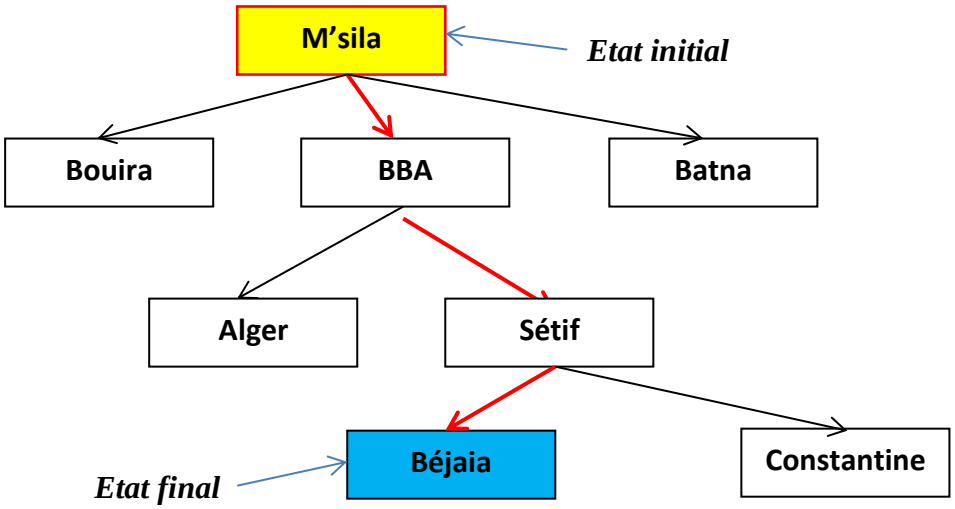
Formulation d'un problème en espace d'états

1. ***Etat:*** une abstraction du monde réel (une configuration possible).
   - Etre dans une ville.
   - Dans une pièce.
   - Tenir un objet (robot).
   - Position d'un objet (robot).
   - Interrupteur/switcher (ouvert/fermé ...).

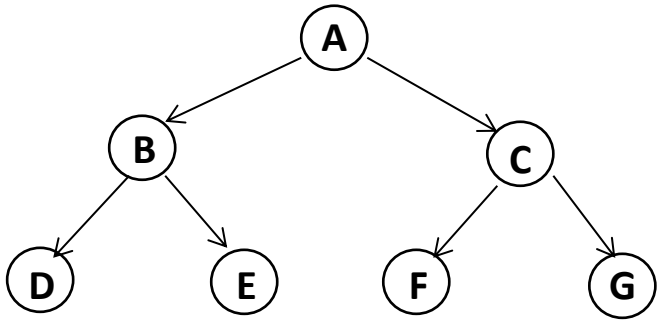
2. ***Actions, transformation d'état, passage entre états:*** abstraction des actions réelles plus ou moins complexes.
   - Prendre ou poser un objet.
   - Avancer des pas.
   - Se déplacer d'un point A vers un autre point B
   - Grimper, ...

- Un problème de recherche est défini par :

1. ***Une situation de départ*** (état intial e_0)
2. ***Un but à atteindre*** (état final e_f)
3. ***Un espace de recherche :*** constitué de l'ensemble des états possibles.
4. ***Une solution :*** est une séquence d'actions reliant l'état initial à un état but.



- Un espace de recherche peut être représenté par un arbre, tel que :
  - Chaque nœud représente un état de cet espace.
  - Deux nœuds sont reliés par un arc, s'il existe une transition entre les deux états.
  - S'il existe un arc du nœud B vers le nœud D, alors B est appelé parent de D, et D est l'enfant (le fils) de B.
  - Le degré de descendance d'un nœud est le nombre d'arcs sortant de ce nœud.



Représentation d'un espace de recherche par un arbre

- Le nœud A est le parent du nœud C
- G est le fils de C
- Degré de descendance de C est 2

Exemples de problèmes de recherche

1. ***Recherche de parcours***
   Itinéraires automatiques, guidage routier, planification de routes aériennes, routage sur les réseaux informatiques, ...

2. ***Robotique***
   Assemblage automatique, navigation autonome, ...

3. ***Planification et ordonnancement***
   Horaires, organisation de tâches, allocation de ressources, ...

Idée de base d'une stratégie de Recherche

Exploration de l'espace d'états en générant les états successeurs des états déjà explorés (appelé aussi "expansion" d'états). On s'arrête lorsqu'on a choisi de développer un nœud qui représente un état final.

```
function Tree-Search (problem, strategy) returns a solution, or failure

initialize the search tree using the initial state of problem

loop do
  if there are no candidates for expansion then return failure
     choose a leaf node for expansion according to strategy
  if the node contains a goal state then return the corresponding solution
  else expand the node and add the resulting nodes to the search tree
end
```

Donc, un algorithme de recherche repose sur les concepts suivants:

- Arbre de recherche.
- Recherche d'un nœud.
- Expansion d'un nœud.
- Stratégie de recherche : à chaque étape du processus de recherche, déterminer quel est le nœud à "étendre".

Gestion des nœuds de l'arbre de recherche

- La gestion des nœuds de l'arbre de recherche repose sur l'utilisation d'une file d'attente.
- La file contient les nœuds de recherche non encore "étendus".
- La file utilise les deux procédures suivantes :

1. ***INSERER (nœud, file)*** : insérer un nouveau nœud dans la file d'attente.
2. EXTRAIRE (file) : extraire le nœud qui occupe la première position dans la file d'attente (stratégie FIFO).

- L'ordre dans lequel les nœuds sont arrangés dans la file d'attente définit la stratégie de recherche.

Deux grandes familles (classes) de stratégies de recherche :

1. Stratégies de recherche aveugles

Caractérisées par :

- N'utilisent aucune information relative à la localisation du but dans l'espace de recherche ou le nœud à étendre.
- Vérifient pour chaque état exploré s'il s'agit du but cherché ou non.

Parmi les stratégies de recherche aveugle, on trouve :

1. Recherche en largeur d'abord
2. Recherche en coût uniforme
3. Recherche en profondeur
4. Recherche en profondeur limitée
5. Recherche en profondeur itérative (RPI).
6. Recherche bidirectionnelle

2. Stratégie de recherche informée (heuristiques)

- Largement utilisées pour les problèmes dont l'espace d'états est caractérisé par une explosion combinatoire.
- Basées sur des règles permettant d'évaluer la probabilité qu'un chemin menant au but soit meilleur par rapport aux autres.
- Ces règles permettent d'estimer les bénéfices des différents chemins avant de les parcourir ➔ Améliorer le processus de recherche.

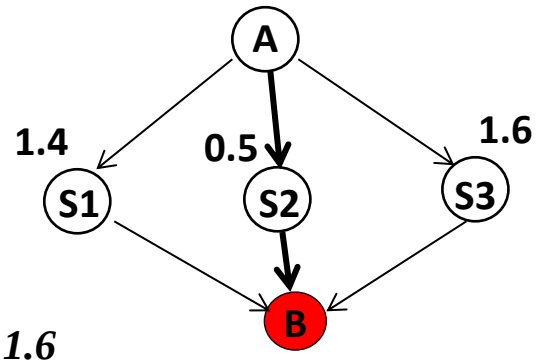
- Une stratégie de recherche heuristique utilise une fonction appelée fonction heuristique ***h***.

$$h : E \longrightarrow R$$

Avec :

- E : l'espace d'états.
- R : l'ensemble des réels.
- h(n) : estime généralement le rapport (coût/bénéfice) du chemin choisi de l'état courant au but cherché, c.à.d choisir le nœud ***n*** qui a la valeur ***h(n)*** [Minimum/Maximum] selon l'heuristique choisie (le nœud le plus promoteur dans la recherche).

Exemple:



- ***h(S1)=1.4 ; h(S2)=0.5 ; h(S3)= 1.6***

- On constate que la poursuite de recherche selon S2 Est heuristiquement meilleure que S1, S3.

- Sous cette famille de stratégies, plusieurs algorithmes peuvent être envisagés :

1. Best First Search BFS (le meilleur d'abord)
2. Greedy search (la recherche gourmande/Gloutonne).
3. Hill climbing.
4. Recuit simulé.
5. A* et variantes : IDA*, SMA*

Critère d'évaluation d'une méthode de recherche

1. ***Complétude :*** est-ce que la méthode garantit de trouver une solution si elle existe ?
2. ***Optimalité :*** est-ce que la méthode trouve la meilleure solution s'il en existe plusieurs ?
3. ***Complexité en temps :*** combien de nœuds faut-il produire pour trouver la solution ?
4. ***Complexité en espace :*** nombre maximum de nœuds à conserver en mémoire pour trouver la solution ?

Remarque :

Les complexités en temps et en espace sont mesurées en fonction des paramètres suivants :

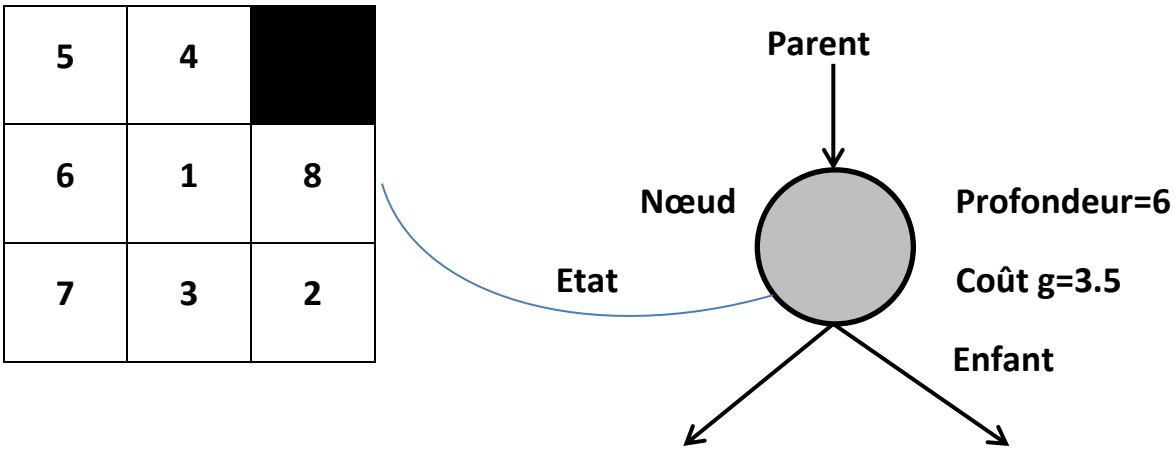
- ✓ b : facteur de branchement de l'arbre de recherche (nombre maximum de successeurs pour un état)
- ✓ d : profondeur à laquelle se trouve le (meilleur) nœud-solution
- ✓ m = profondeur maximum de l'espace de recherche (peut être ∞)

États vs. Nœuds

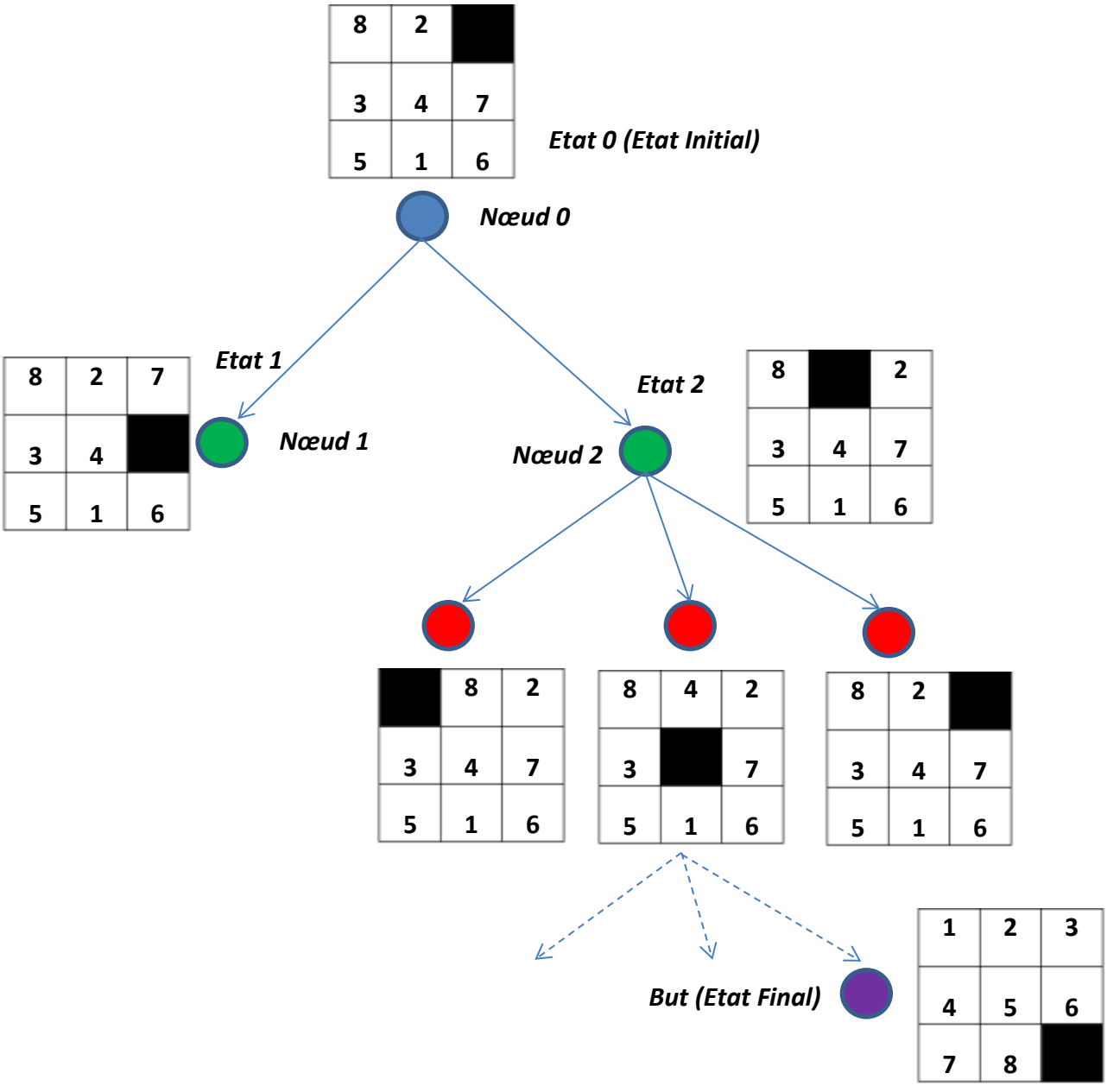
- Un état est une représentation d'une configuration réelle du problème
- Un nœud est un élément de base de l'arbre de recherche ou une structure constituée des champs suivants :
  - Etat parent.
  - Etat enfants.
  - Profondeur
  - Coût de chemin g(x)

N.B: Ces propriétés n'existent pas pour les états.

Etat (PUZZLE-8)



Exemple complet (PUZZLE-8 /Taquin)



Remarques :

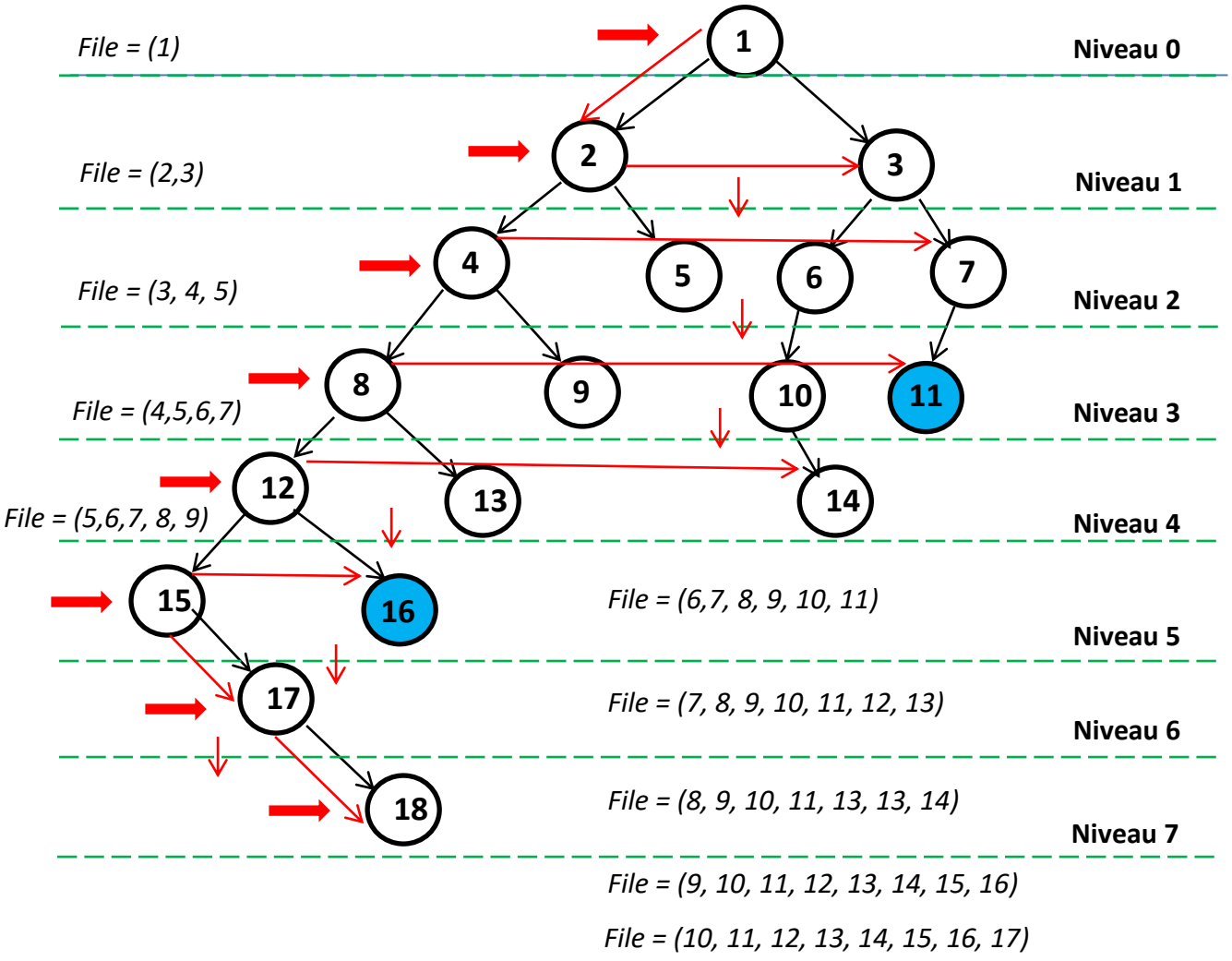
- L'arbre de recherche peut être infini, alors que l'espace des états est fini.
- Des problèmes formulés comme des problèmes de recherche sont NP-difficiles, ex : puzzle-(n^2-1). On ne peut pas espérer les résoudre en moins qu'un temps exponentiel dans le pire des cas.

Stratégies de recherche aveugles

1. Recherche en largeur d'abord

- Appelée aussi recherche par niveau.
- On parcourt l'arbre de recherche à partir de la racine, de gauche vers la droite, de haut vers le bas (niveau par niveau)
- Etendre le nœud le moins profond.

Exemple : Soit l'arbre qui représente l'espace de recherche d'un problème



Evolution de la file d'attente (suite)

File = (12, 13, 14, 15, 16, 17, 18) ➔ File = (13, 14, 15, 16, 17, 18) ➔ File = (14, 15, 16, 17, 18)
File = (11, 12, 13, 14, 15, 16, 17, 18)
⇨ *File = (15, 16, 17, 18) ➔ File = (16, 17, 18) ➔ File = (17, 18) ➔ File = (18) ➔ File = ()*

Séquencement du parcours

L'ordre de parcours des nœuds : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, **16**, 17, 18

Performance de l'algorithme

1. Complétude : Oui (si b est fini).
2. Complexité en temps

$$1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = (b^{d+1} - 1)/(b - 1)$$

$$= O(b^{d+1}) \text{ (exponentiel en d)}$$

3. Complexité en espace $O(b^{d+1})$ (il faut garder tous les nœuds en mémoire)
4. Optimalité : Oui (si coût unitaire pour chaque pas est 1), mais en général pas optimal.

Inconvénients

- Gourmand en espace mémoire.
- Gourmand en temps.

Exemple :

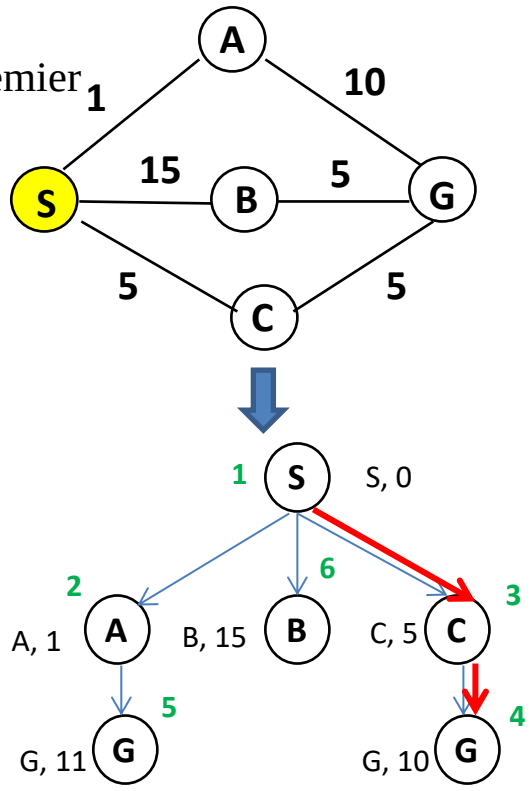
- B (facteur de branchement) = 10 ; d (profondeur de l'arbre) = 8
- Effectuer 10000 Nœuds/Seconde
- 1000 octets/nœud en mémoire

profondeur	Nœuds $O(b^{d+1})$	Temps $O(b^{d+1})$	Mémoire
8	10^9	$10^9/10^4$ s = 31 h	$10^9 x 10^3 = 10^{12}$=1 To
12	10^{13}	10^9 s = 35 ans	10^{16}octets=10 Peta.O

2. Recherche en coût uniforme

Elle a le même principe que la recherche en largeur. Sauf ici que l'insertion du nœud suivant dans la file se fait dans l'ordre du coût g(x) associé à chaque transition. C.à.d. prendre le nœud de coût le plus faible.

1. Insérer l'état initial ayant un coût nul.
   {(S, 0)} dans la file et construisant le premier Nœud de l'arbre
2. {(A, 1), (C, 5), (B, 15)} toujours selon l'ordre du coût.
3. {(C, 5), (G, 11), (B, 15)}
4. {(G, 10), (G, 11), (B, 15)}
5. {(G, 11), (B, 15)}
6. {(B, 15)}
7. { }



N.B: Ordre de construction des nœuds en vert

Propriétés de l'algorithme :

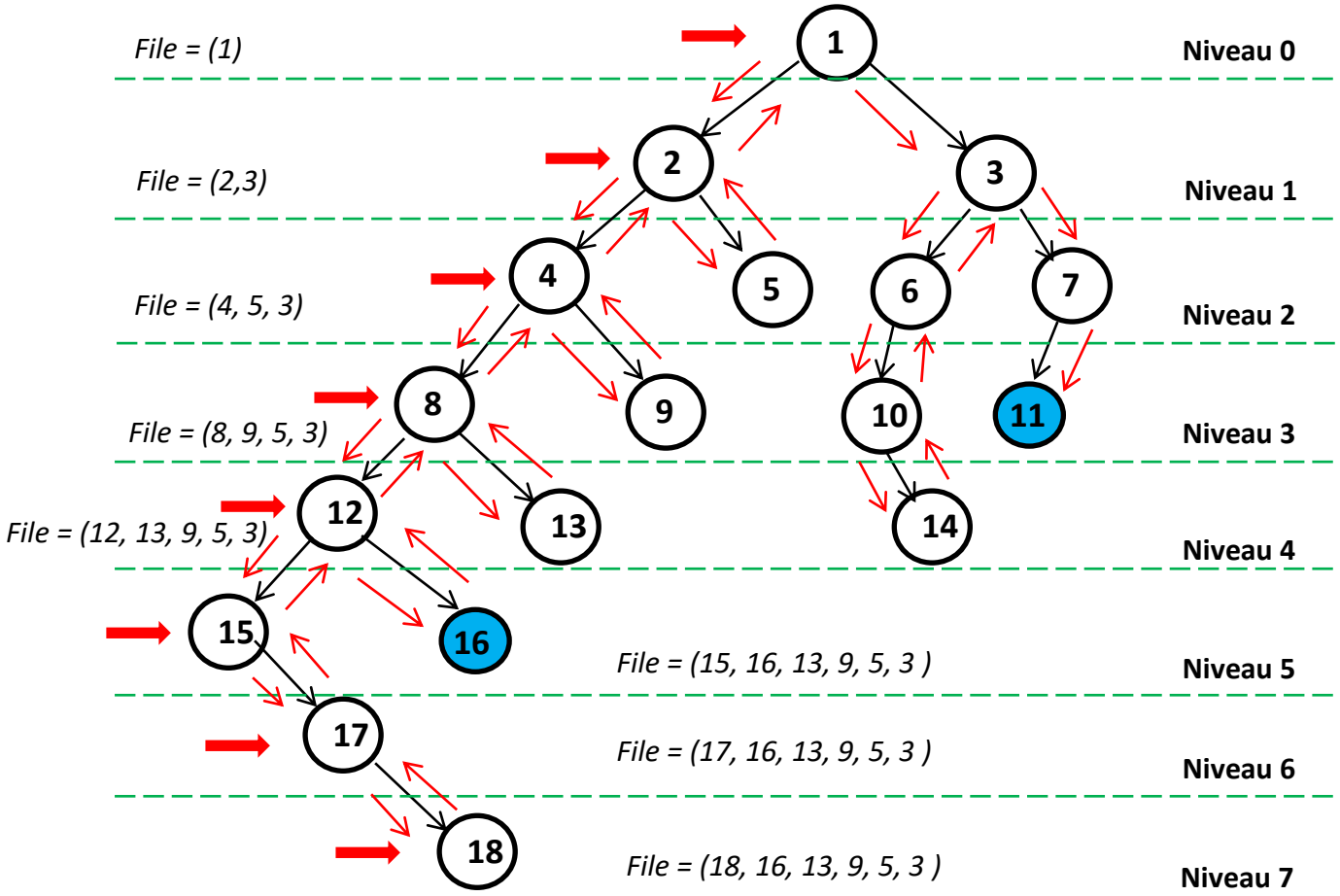
- Equivalent au précédent (la largeur d'abord) si le coût est toujours le même.
- Complétude : Oui
- Complexité en temps : O (b^d)
- Complexité en espace : O (b^d)
- Optimalité : Oui

3. Recherche en profondeur d'abord

Principe: étendre chaque fois le nœud le plus profond.

Et pour la file, on insère toujours les nouveaux nœuds au début, et on enlève le nœud accédé.

Exemple :



File = (15, 16, 13, 9, 5, 3)

File = (17, 16, 13, 9, 5, 3)

File = (18, 16, 13, 9, 5, 3)

File = (16, 13, 9, 5, 3)

File = (13, 9, 5, 3)

File = (9, 5, 3)

File = (5, 3)

File = (3)

File = (6, 7)

File = (10, 7)

File = (14, 7,) ➔ *File= (7)* ➔ *File(11) = File()*

Séquencement du parcours

L'ordre de parcours des nœuds : 1, 2, 4, 8, 12, 15, 17, 18, **16,** 13, 9, 5, 3, 6, 10, 14, 7, **11**

Performance de l'algorithme

1. Non complet dans les espaces d'états infinis ou avec boucle
   - Il est possible d'ajouter un test pour détecter les répétitions
2. Complexité en temps : O (b^m)
   - Très mauvais si *m* est beaucoup plus grand que *b*
3. Complexité en espace : O (b*m)
   - Linéaire !
4. Non optimale

Avantages

- Moins couteux en espace mémoire.

Inconvénients

1. Si le but se trouve à droite la recherche sera longue.
2. S'il existe une branche infinie située avant une branche contenant un but, la recherche rentre dans une boucle infinie.

Comparaison :

Pour b = 10, d = 12 et 100 octets/noeud:

- Recherche en profondeur a besoin de **12 Koctets (**besoins modestes en espace)
- Recherche en largeur a besoin de **111 Téraoctets.**

4. Recherche en profondeur limitée

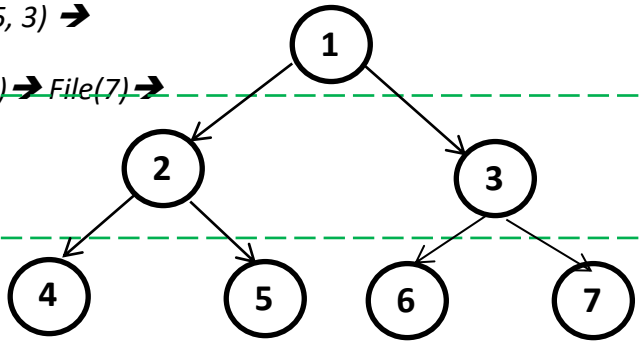
Principe : le même principe que l'algorithme de recherche en profondeur d'abord, mais avec une limite *L* sur la profondeur d'exploration => (1). Découper l'arbre en tranche de *L* niveaux.
(2). La recherche dans chaque tranche est une recherche en profondeur.

Exemple pour l = 2

File=(1) ➔ *File(2,3)* ➔ *File(4, 5, 3)* ➔

File(5, 3) ➔ *File(3)* ➔ *File(6, 7)* ➔ *File(7)* ➔

File()



Propriétés

- Complétude : Oui (si L < d)
- Complexité en temps : O (b^L)
- Complexité en espace : O (b * L)
- Optimalité : Non

Avantages

1. Eviter le problème d'une branche infinie en limitant une profondeur maximale.
2. Moins couteux en espace mémoire.

5. Recherche en profondeur itérative

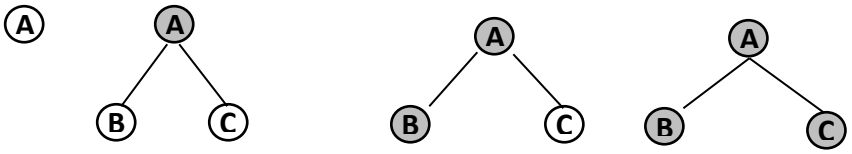
Principe:

- Profondeur limitée, mais en essayant toutes les profondeurs : 0, 1, 2, 3, etc.
- Evite le problème de trouver une limite pour la recherche profondeur limitée.
- Combine les avantages de largeur d'abord (complète et optimale), mais a la complexité en espace de profondeur d'abord

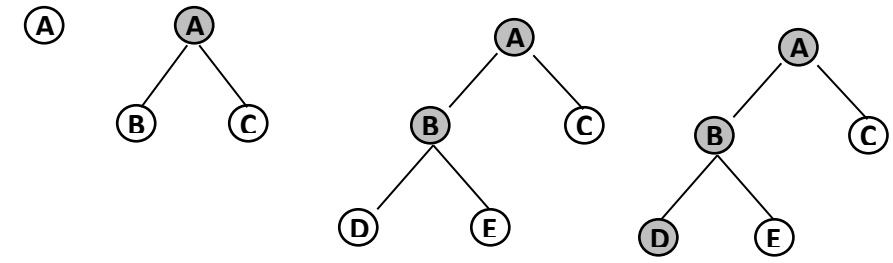
Exemple :

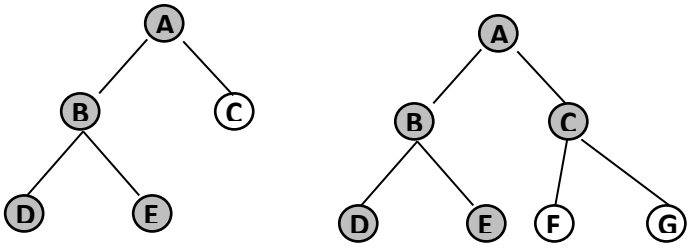
- Pour L = 0

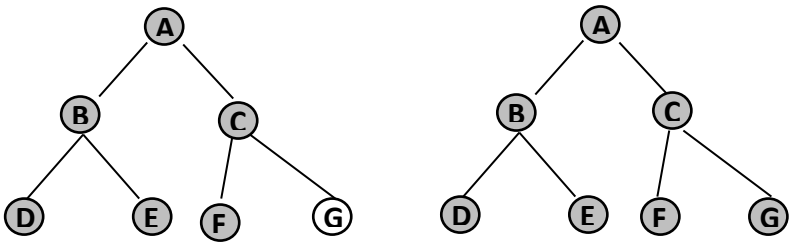
- Pour L = 1



- Pour L = 2







Propriétés

- Peut paraître du gaspillage car beaucoup de nœuds sont étendus de multiples fois.
- Mais la plupart des nouveaux nœuds étant au niveau le plus bas, ce n'est pas important d'étendre plusieurs fois les nœuds des niveaux supérieurs.
- Complet
- Complexité en temps :
  $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots\ldots + b^d = O(b^d)$
- Complexité en espace : O(b*d)
- Optimale : oui (si le coût de chaque action est de 1). Peut être modifiée pour une stratégie de coût uniforme.

Résumé des algorithmes aveugles

Critères	Largeur D'abord	Coût Uniforme	Profondeur D'abord	Profondeur Limitée	Profondeur Itérative
Complétude	Oui	Oui	Non	Oui si L < d	Oui
Temps	$O(b^{d+1})$	$O(b^d)$	$O(b^m)$	$O(b^L)$	$O(b^d)$
Espace	$O(b^{d+1})$	$O(b^d)$	O(b*m)	O(b*L)	O(b*d)
Optimalité (coût d'une action = 1)	Oui	Oui	Non	Non	Oui
Optimalité (cas général)	Non	Non	Non	Non	Non

Stratégies de recherche informée (heuristique)

1. La recherche du meilleur d'abord BFS

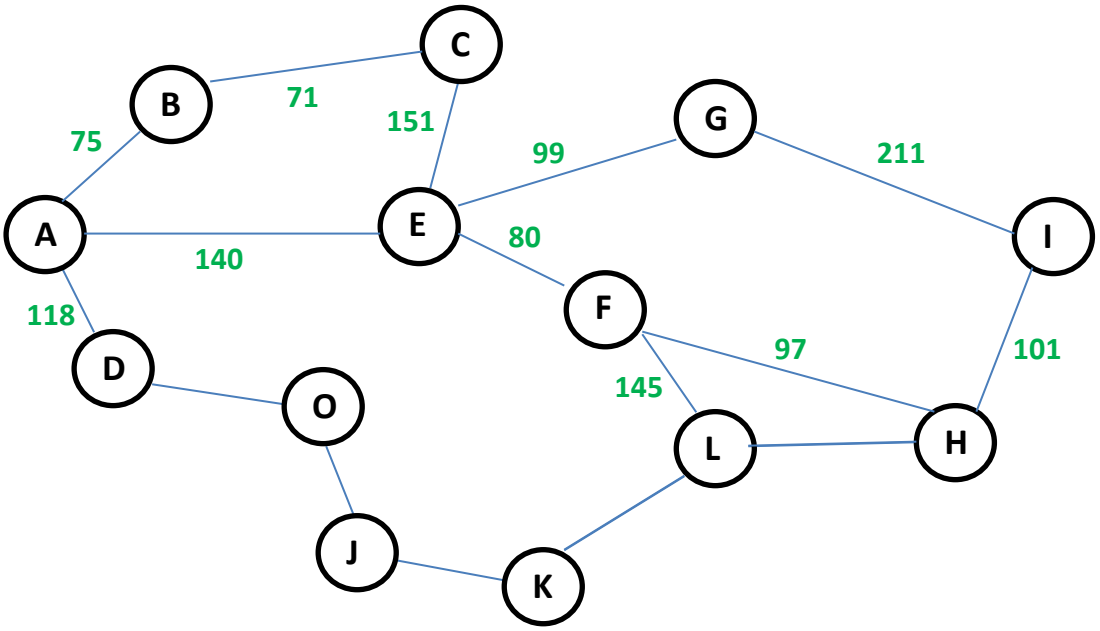
Principe : cet algorithme utilise une fonction qui mesure l'utilité d'un nœud (une fonction heuristique) ➔ on prend toujours le nœud qui a la meilleure valeur d'utilité h(n).

Notons aussi que :

- La fonction h(n) est généralement une estimation d'un rapport/coût/bénéfice.
- Pour la gestion de la file d'attente, on insère le nœud le plus utile en premier.

Exemple :

Soit l'exemple suivant avec :

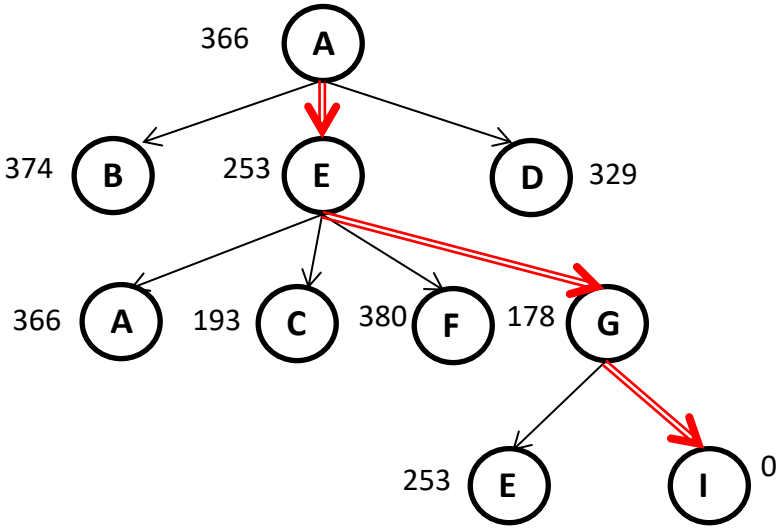


Nœud	A	B	C	D	E	F	G	H	I	J	K	L
h(n)	366	374	193	329	253	380	178	98	0	500	602	615

- Les nœuds représentent des villes.
- Les valeurs sur les arcs représentent des distances entre ces villes.
- Les valeurs enregistrées dans le tableau indiquent les valeurs de la fonction heuristique *h*.
- La ville *A* est l'état initial et la ville *I* est l'état final.
- A chaque fois on passe vers le nœud qui a la valeur heuristique minimale *h(n)*

Problème : ***comment passer de la ville A (l'état initial) vers l'état I (l'état final) ?***

L'arbre de recherche de la solution



Alors la solution est : {A, E, G, I}

Cas spéciaux de l'algorithme recherche du meilleur d'abord

a) La recherche gloutonne (Greedy-Search)

- La variante la plus simple de l'algorithme BFS.
- Utilise une fonction heuristique *h(n)* qui estime le coût total minimal pour passer d'un état à l'autre.
  (ex : *h(n)* : la distance directe minimale entre la ville *n* et la ville destination).
- L'algorithme gloutonne privilège le nœud qui parait être plus proche de l'état final selon la fonction choisie.

Propriétés :

- Incomplet (peut aller dans des boucles)
- Complet si on ajoute test d'états répétés (les négliger).
- Complexité en temps : $O(b^m)$ (bonne heuristique peut améliorer beaucoup).
- Complexité en espace : $O(b^m)$.
- Non optimal.

Inconvénients :

- Couteux en temps.
- Couteux en espace mémoire (garder tous les nœuds en mémoire).

*b) L'algorithme A**

Principe :

- Eviter les chemins de coût élevé.
- Utilise une fonction heuristique composée $f(n) = g(n) + h(n)$
  Avec :

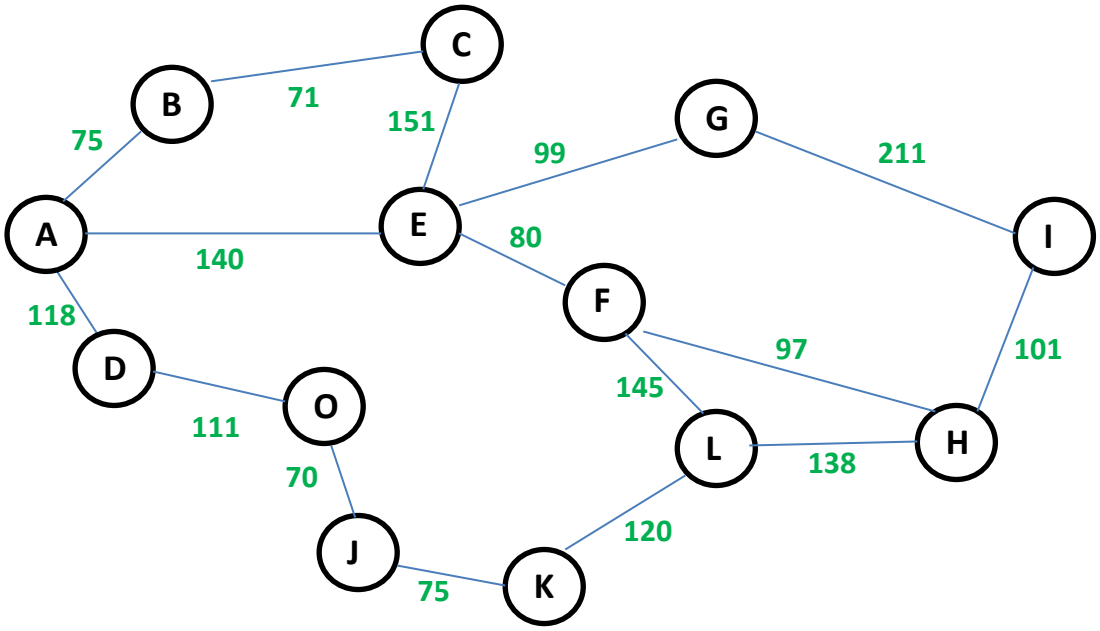
- *g(n)* : le coût estimé jusqu'à présent (à partir de l'état initial jusqu'au nœud courant n. c.à.d. coût de la partie déjà explorée)
- *h(n)* : coût estimé pour aller du nœud courant *n* vers un état final. (coût de la partie non déjà explorée)
- *F(n)* : le coût total

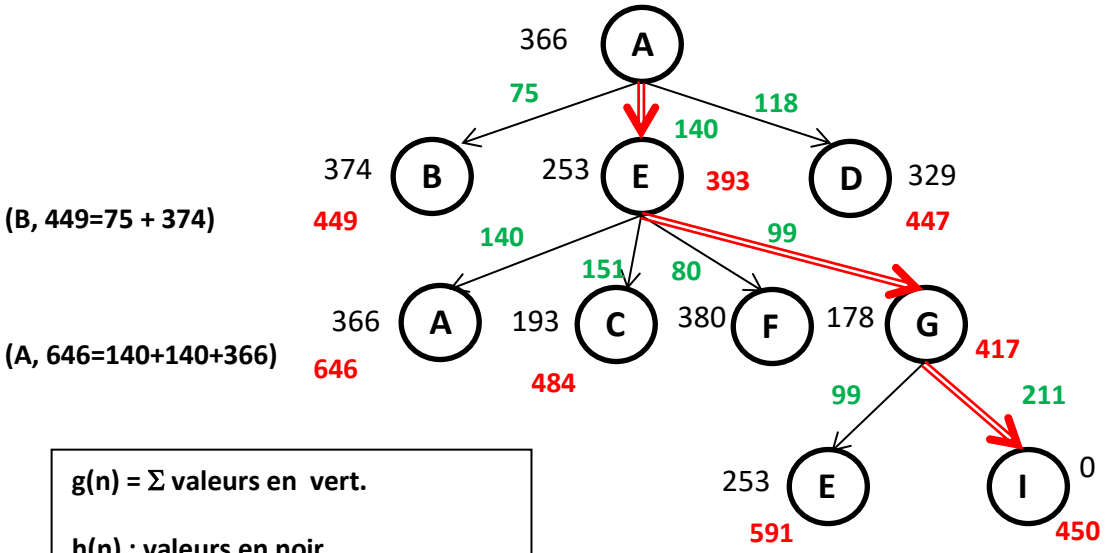
Remarque : A* est toujours optimal

Propriétés :

- Complet.
- Complexité en temps : exponentiel.
- Complexité en espace : tous les nœuds sont chargés.
- L'algorithme A^* est optimal.

Exemple 1 : Atteindre la ville I à partir de la ville A





g(n) = Σ valeurs en vert.

h(n) : valeurs en noir.

F(n) = g(n) + h(n)

Comment choisir entre deux heuristiques admissibles ?

Prenons l'exemple du PUZZLE-8

Etat initial S

5	4	
6	1	8
7	3	2

Etat final F

1	2	3
8		4
7	6	5

Deux heuristiques admissibles h_1, h_2 :

- $h_1(n)$: nombre de pièces mal placées
- $h_2(n)$: Σ distance de chaque pièce à sa position finale

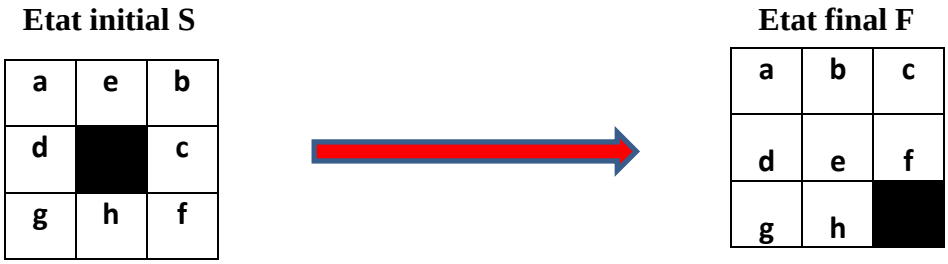
Dans l'exemple précédent on a:

$h_1(S) = 7$;
$h_2(S) = d(1) + d(2) + d(3) + d(4) + d(5) + d(6) + d(7) + d(8)$
$= 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18$;

Remarques:

- On dit que *h2* domine *h1* si seulement si $h2(n) \geq h1(n)$ pour tout état *n*.
- Une heuristique dominante peut réduire considérablement l'espace de recherche.

Exemple complet



1. Opérateurs possibles : déplacer le vide à gauche G, à droite D, en haut H, en bas B.
2. On ne doit pas régénérer la même configuration (état).
3. On utilise une fonction heuristique composée f.
   Avec $f(n) = g(n) + h(n)$
   $g(n)$: la longueur du chemin de l'état initial e_0 à l'état courant e_n
   $h(n)$: le nombre de carrées mal placées par rapport à la configuration finale.