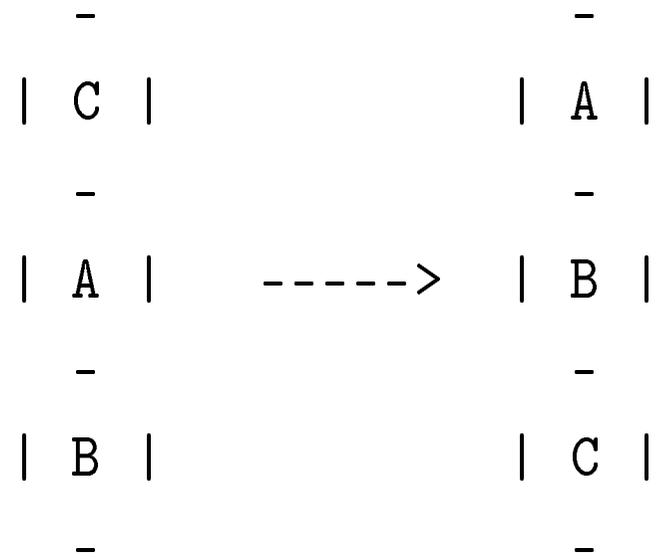

Algorithmes de recherche

Résolution de problèmes par recherche

- On représente un problème par un **espace d'états (arbre/graphe)**.
- Chaque état est une **configuration possible** du problème.
- Résoudre le problème consiste à trouver un **chemin** dans le graphe.
 - Parcours aveugles non informés : profondeur, largeur.
 - Parcours informés.

Exemple : la tour de cubes



Contraintes

- On veut réarranger la pile de cubes, en ne déplaçant qu'un cube à la fois.
- On peut déplacer un cube uniquement s'il n'a pas un autre au dessus de lui.
- On peut poser un cube sur la table ou sur un autre cube.

Configurations et actions possibles

C						B
A	<->	A	<->	A	<->	A
B		B C		B C		C

Compléter....

Éléments du graphe

- Les **noeuds/états** du graphe sont les configurations possibles du jeu.
- Les **arcs** du graphe sont les transitions possibles.
- Il y a (au moins) un **état initial** (p.e. C A B).
- Il y a (au moins) un **état final**.
 - Explicite (p.e. A B C).
 - Implicite (décrit par une propriété).
- Trouver une solution consiste à trouver un **chemin** allant d'un état initial vers un état final (p.e. un chemin de l'état C A B vers l'état A B C).
- Coût du chemin : somme de distances, nb d'opérateurs, etc.

Éléments du graphe

- **États** : toutes les configurations possibles pour les entiers 1..8 dans le puzzle.
- **Arcs** : déplacer le blanc vers la gauche, vers la droite, vers le haut, vers le bas.
- **État initial** : toute configuration sauf celle de droite.
- **État final** : celui à droite.
- **Coût du chemin** : nb de mouvements.

Exemple : l'aspirateur

<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="border-right: 1px solid black; padding: 5px;">aspirateur</td> <td style="padding: 5px;">poussiere</td> <td style="border-right: 1px solid black; padding: 5px;"></td> </tr> <tr> <td style="border-right: 1px solid black; padding: 5px;">et</td> <td style="padding: 5px;"></td> <td style="border-right: 1px solid black; padding: 5px;"></td> </tr> <tr> <td style="border-right: 1px solid black; padding: 5px;">poussiere</td> <td style="padding: 5px;"></td> <td style="border-right: 1px solid black; padding: 5px;"></td> </tr> </table>	aspirateur	poussiere		et			poussiere			<p>-----></p>	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="border-right: 1px solid black; padding: 5px;">aspirateur</td> <td style="padding: 5px;"></td> <td style="border-right: 1px solid black; padding: 5px;"></td> </tr> </table>	aspirateur		
aspirateur	poussiere													
et														
poussiere														
aspirateur														

Éléments du graphe

- **États** : toutes les configurations possibles pour l'aspirateur dans le carré double avec ou sans poussière.
- **Arcs** : déplacer l'aspirateur à droite, déplacer l'aspirateur à gauche, aspirer.
- **État initial** : celui à gauche.
- **État final** : pas de poussière.
- **Coût du chemin** : nb d'opérateurs.

Algorithmes 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, ajouter tous les **successeurs** de **e** à la liste d'états à traiter et recommencer au point 2.
3. Sinon retourner **recherche négative**.

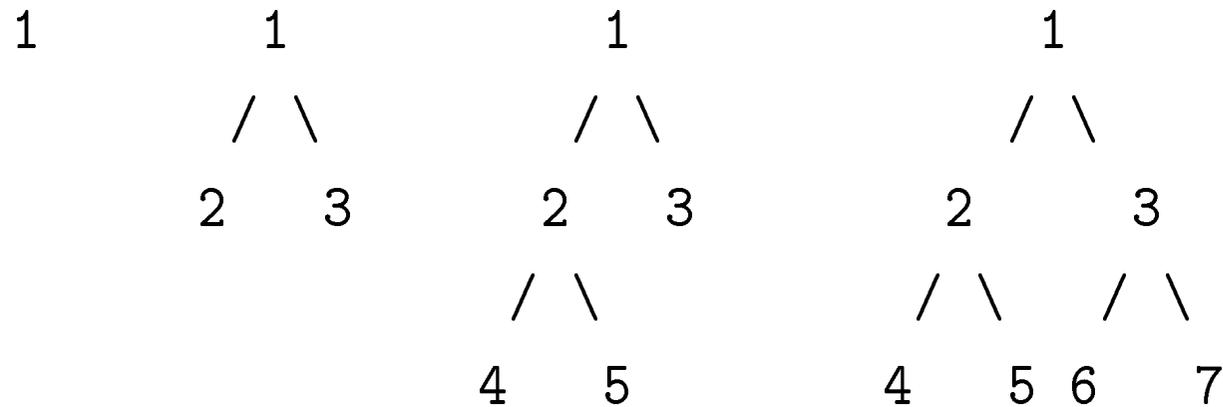
Les stratégies

- C'est un **critère** qui permet de choisir un ordre pour traiter les états du problème.
- On tiendra compte de :
 - La **complétude**
 - L'**optimalité** (selon le coût).
 - La **complexité** (en temps et en espace) mesurée par :
 - b : branchement maximal de l'arbre de recherche
 - d : profondeur de la meilleur solution
 - m : profondeur maximale de l'espace d'états

Stratégies aveugles

- En largeur d'abord
- À coût uniforme
- En profondeur d'abord
- En profondeur limitée
- En profondeur itérative

Recherche en largeur d'abord



Structure de données

On utilise une structure de liste :

1. Mettre 1 dans la liste. On obtient [1].
2. Enlever le premier élément de la liste (le 1) et rajouter ses successeurs 2, 3. On obtient [2, 3].
3. Enlever le premier élément de la liste (le 2) et rajouter ses successeurs 4, 5. On obtient [3, 4, 5].
4. Enlever le premier élément de la liste (le 3) et rajouter ses successeurs 6, 7. On obtient [4, 5, 6, 7].

Caractéristiques de la recherche en largeur d'abord

- Complète si b est fini.
- Complexité en temps : $\mathcal{O}(b^d)$
- Complexité en espace : $\mathcal{O}(b^d)$
- Optimale si coût = 1 , non optimale en générale.

Variante : recherche à coût uniforme

Problème : On veut aller de S à G en utilisant le chemin le plus court, où les distances sont données par le tableau suivant :

$$\text{dist}(S,A) = 1$$

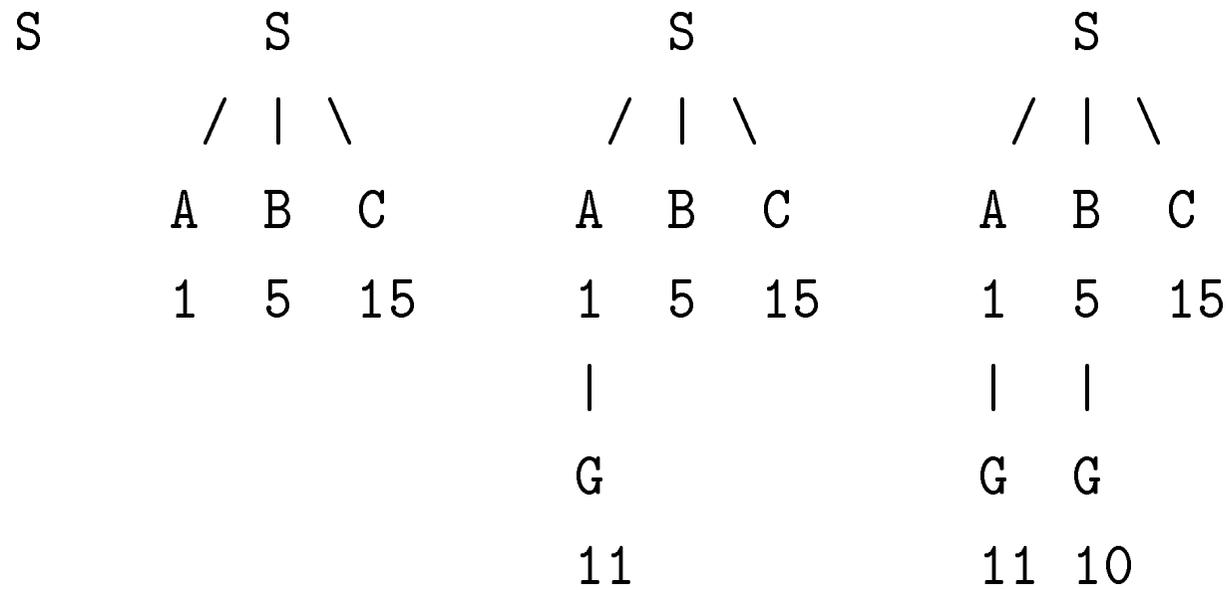
$$\text{dist}(S,B) = 5$$

$$\text{dist}(S,C) = 15$$

$$\text{dist}(A,G) = 10$$

$$\text{dist}(B,G) = 5$$

$$\text{dist}(C,G) = 5$$



Structure de données

On utilise une structure de liste triée :

1. Mettre S dans la liste avec son coût initial. On obtient $[(S, 0)]$.
2. Enlever le premier élément de la liste $(S, 0)$ et rajouter ses successeurs A, B, C à la liste d'états en respectant l'ordre croissant. Pour cela on calculera le coût total de chaque successeur : coût de S + coût de chaque arc. On obtient $[(A, 1), (B, 5), (C, 15)]$.
3. Enlever le premier élément de la liste $(A, 1)$ et rajouter son successeur G à la liste en respectant l'ordre croissant. Pour cela on calculera le coût total de G : coût de A + coût de l'arc $A \rightarrow G$. On obtient $[(B, 5), (G, 11), (C, 15)]$.

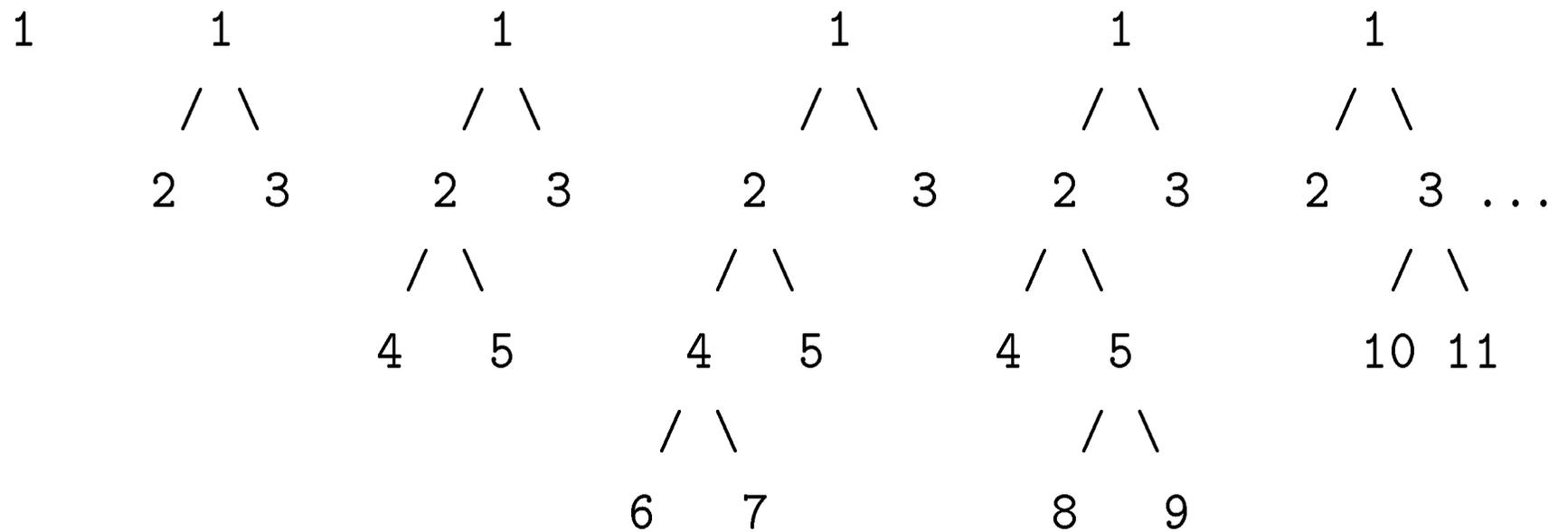
4. Enlever le premier élément de la liste $(B, 5)$ et rajouter son successeur G à la liste en respectant l'ordre croissant. Pour cela on calculera le coût total de G : coût de B + coût de l'arc $B \rightarrow G$. On obtient $[(G, 10), (G, 11), (C, 15)]$.

Observations et variantes

Le coût est associé à **chaque arc**.

Si coût de chaque arc = 1, alors recherche à coût uniforme = recherche en largeur d'abord.

Recherche en profondeur d'abord



Structure de données

On utilise une structure de pile :

1. Mettre 1 dans la pile. On obtient [1].
2. Enlever le premier élément de la pile (le 1) et rajouter ses successeurs 2, 3. On obtient [2, 3].
3. Enlever le premier élément de la pile (le 2) et rajouter ses successeurs 4, 5. On obtient [4, 5, 3].
4. Enlever le premier élément de la pile (le 4) et rajouter ses successeurs 6, 7. On obtient [6, 7, 5, 3].
5. Enlever le premier élément de la pile (le 6) qui n'a pas de successeurs. On obtient [7, 5, 3].
6. Enlever le premier élément de la pile (le 7) qui n'a pas de

- successeurs. On obtient $[5, 3]$.
7. Enlever le premier élément de la pile (le 5) et rajouter ses successeurs 8, 9. On obtient $[8, 9, 3]$.
 8. Enlever le premier élément de la pile (le 8) qui n'a pas de successeurs. On obtient $[9, 3]$.
 9. Enlever le premier élément de la pile (le 9) qui n'a pas de successeurs. On obtient $[3]$.
 10. Enlever le premier élément de la pile (le 3) et rajouter ses successeurs 10, 11. On obtient $[10, 11]$.
 11. Enlever le premier élément de la pile (le 10) et rajouter ses successeurs 12, 13. On obtient $[12, 13, 11]$.
 12. Enlever le premier élément de la pile (le 12) qui n'a pas de successeurs. On obtient $[13, 11]$.

13. Enlever le premier élément de la pile (le 13) qui n'a pas de successeurs. On obtient [11].
14. Enlever le premier élément de la pile (le 11) et rajouter ses successeurs 14, 15. On obtient [14, 15].
15. Enlever le premier élément de la pile (le 14) qui n'a pas de successeurs. On obtient [15].
16. Enlever le premier élément de la pile (le 15) qui n'a pas de successeurs. On obtient [].

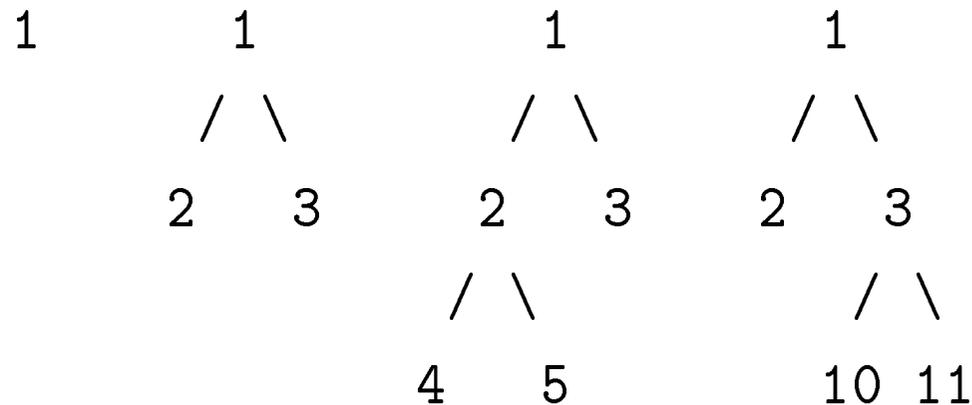
Caractéristiques de la recherche en profondeur d'abord

- Non complète si espace d'états infini.
- Complexité en temps : $\mathcal{O}(b^m)$
- Complexité en espace : $\mathcal{O}(b \times m)$
- Non optimale.

Variante : recherche en profondeur limitée

C'est une recherche en profondeur d'abord où l'on explore les états jusqu'à une profondeur limitée.

Exemple avec limite 2



On rajoute les successeurs d'un état **uniquement** si on n'a pas dépassé la limite 2.

1. Mettre 1 dans la pile avec sa profondeur. On obtient $[(1,0)]$.
2. Enlever le premier élément de la pile $(1,0)$ et rajouter ses successeurs 2, 3 si on n'a pas dépassé la limite. On obtient $[(2,1), (3,1)]$.
3. Enlever le premier élément de la pile $(2,1)$ et rajouter ses successeurs 4, 5 si on n'a pas dépassé la limite. On obtient $[(4,2), (5,2), (3,1)]$.
4. Enlever le premier élément de la pile $(4,2)$ et ne rien rajouter. On obtient $[(5,2), (3,1)]$.
5. Enlever le premier élément de la pile $(5,2)$ et ne rien rajouter. On obtient $[(3,1)]$.
6. Enlever le premier élément de la pile $(3,1)$ et rajouter ses successeurs 10, 11 si on n'a pas dépassé la limite. On obtient

$[(10, 2), (11, 2)]$.

7. Enlever le premier élément de la pile $(10, 2)$ et ne rien rajouter.
On obtient $[(11, 2)]$.
8. Enlever le premier élément de la pile $(11, 2)$ et ne rien rajouter.
On obtient $[]$.

Variante : recherche en profondeur itérative

C'est une itération de la recherche en profondeur limitée.

Pour p de 0 a infini faire

`recherche_profondeur_limitee(p)`

Caractéristiques de la recherche en profondeur itérative

- Complète.
- Complexité en temps : $1 + b + b^2 + \dots + b^d = \mathcal{O}(b^d)$.
- Complexité en espace : $\mathcal{O}(b \times d)$.
- Optimale si coût = 1.

Stratégies informées

- Meilleur d'abord
- L'algorithme A^*
- Heuristiques

Recherche meilleur en premier

- **Compromis** entre recherche en largeur et recherche en profondeur
- On associe à **chaque état** une **mesure d'utilité** unique
- On utilise cette mesure pour :
 - Choisir l'état suivant à traiter
 - Rajouter les successeurs d'un état à la liste d'états à traiter

Algorithme de recherche meilleur en premier

1. **Démarrer** la recherche avec la liste contenant l'**état initial** du problème.
2. Si la liste n'est pas vide alors :
 - (a) **Choisir** un état **e** de mesure **minimale** 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 par **ordre croissant** selon la mesure d'utilité.
Recommencer au point 2.
3. Sinon retourner **recherche négative**.

Cas spéciaux de la recherche meilleur en premier

- Recherche gloutonne
- Algorithme A^*

Recherche gloutonne

- La mesure d'utilité est donnée par une **fonction d'estimation** h .
- Pour chaque état n , $h(n)$ représente l'**estimation** du coût de n vers un état final. Par exemple, dans le pb du chemin le plus court entre deux villes on peut prendre $h_1(n) = \text{distance}$ **directe** entre n et la ville destination.
- La recherche gloutonne choisira l'état qui **semble** le plus proche d'un état final selon la fonction d'estimation.

Caractéristiques de la recherche gloutonne

- Incomplète (car boucles)
- Complexité en temps : $\mathcal{O}(b^m)$
- Complexité en espace : $\mathcal{O}(b^m)$
- Non optimale.

Algorithme A^*

- On évite d'explorer les chemins qui sont déjà chers.
- La mesure d'utilité est donnée par une **fonction d'évaluation** f .
- Pour chaque état n , $f(n) = g(n) + h(n)$, où
 - $g(n)$ est le coût jusqu'à présent pour atteindre n
 - $h(n)$ est le coût estimé pour aller de n vers un état final.
 - $f(n)$ est le coût total estimé pour aller d'un état initial vers un état final en passant par n .

Caractéristiques de l'algorithme A^*

- L'heuristique de A^* est **admissible** : pour tout état n on a

$$h(n) \leq h^*(n)$$

où $h^*(n)$ est le **vrai** coût pour aller de n vers un état final.

- Par exemple $h_1(n) \leq h^*(n)$.
- Complet.
- Complexité en temps : exponentiel.
- Complexité en espace : tous les noeuds.
- L'algorithme A^* est **optimal**.

Exemple : Roumanie avec coûts estimés

On veut aller de **Arad** à **Bucharest**.

Etape 1

(Arad, 366)

Etape 2

(Sibiu, $393=140+253$)

(Timisoara, $447=118+329$)

(Zerind, $449=75+374$)

Un exemple : Roumanie avec coûts estimés

Etape 3

(Rimnicu Vilcea, $413=80+140+193$)

(Fagaras, $417=99+140+178$)

(Timisoara, 447)

(Zerind, 449)

(Arad, $646=140+140+366$)

(Oradea, $671=151+140+380$)

Un exemple : Roumanie avec coûts estimés

Etape 4

(Pitesti, 415=97+80+140+98)

(Fagaras, 417=99+140+178)

(Timisoara, 447)

(Zerind, 449)

(Craiova, 526=146+80+140+160)

(Sibiu, 553=80+80+140+253)

(Arad, 646=140+140+366)

(Oradea, 671=151+140+380)

Un exemple : Roumanie avec coûts estimés

Etape 5

(Fagaras, $417=99+140+178$)

(Bucharest, $418=101+97+80+140+0$)

(Timisoara, 447)

(Zerind, 449)

(Craiova, $526=146+80+140+160$)

(Sibiu, $553=80+80+140+253$)

(Rimnicu Vilcea, $607=97+97+80+140+193$)

(Craiova, $615=138+97+80+140+160$)

(Arad, $646=140+140+366$)

(Oradea, $671=151+140+380$)

Un exemple : Roumanie avec coûts estimés

Etape 6

(Bucharest, $418=101+97+80+140+0$)

(Timisoara, 447)

(Zerind, 449)

(Bucharest, $450=211+99+140+0$)

(Craiova, $526=146+80+140+160$)

(Sibiu, $553=80+80+140+253$)

(Sibiu, $591=99+99+140+253$)

(Rimnicu Vilcea, $607=97+97+80+140+193$)

(Craiova, $615=138+97+80+140+160$)

(Arad, $646=140+140+366$)

(Oradea, $671=151+140+380$)

L'algorithme A^* est optimal (preuve)

Supposons qu'un but G_2 pas optimal est dans la queue.

Soit n un noeud dans la queue qui se trouve dans un chemin vers un but optimal G_1 .

$$\begin{aligned}
 f(G_2) &= g(G_2) && \text{car } h(G_2) = 0 \\
 &> g(G_1) && \text{car } G_1 \text{ est optimal} \\
 &\geq f(n) && \text{car } h \text{ admissible implique} \\
 &&& f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G_1)
 \end{aligned}$$

Comme $f(G_2) > f(n)$, l'algorithme A^* ne va jamais choisir G_2 .

Lors que la fonction d'évaluation décroît

n $g=5, h=4, f=9$

|

| 1

|

m $g=6, h=2, f=8$

Perte de l'information car :

- $f(n) = 9$ dit que le vrai coût d'un chemin qui passe par n est ≥ 9 .
- Donc le vrai coût d'un chemin qui passe par n et puis par m est aussi ≥ 9 .

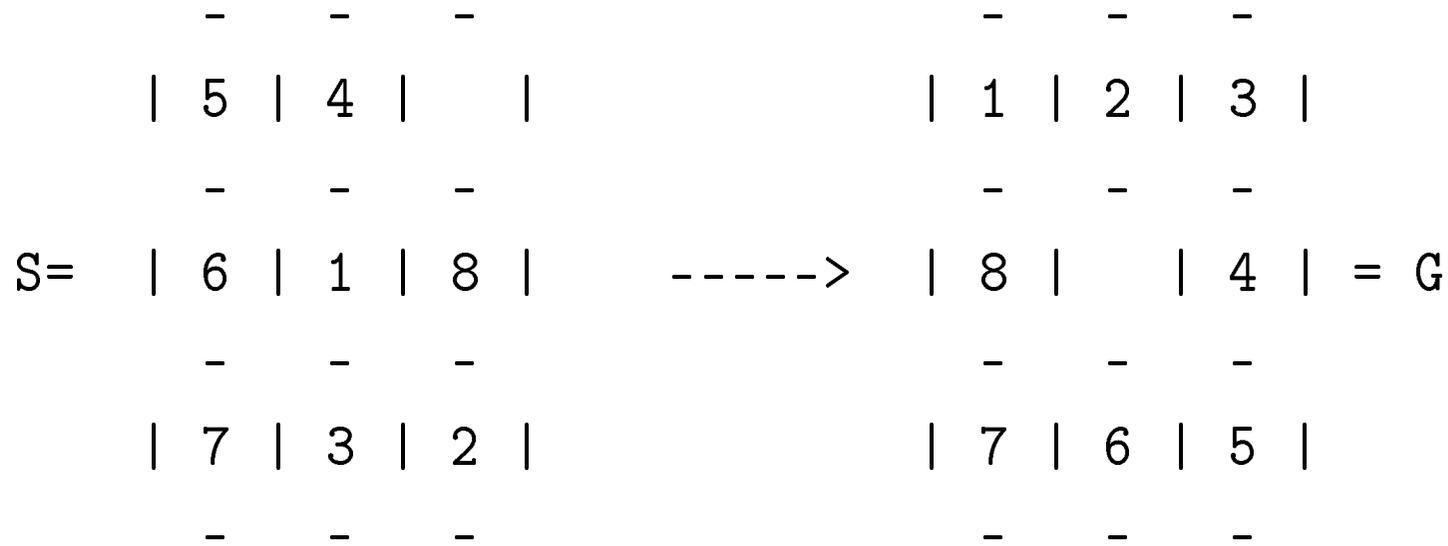
Modification de l'algorithme A^* lorsque f décroît

Si m est un successeur de n :

$$f(m) = \max(g(m) + h(m), f(n))$$

f ne décroît plus...

Comment choisir entre deux heuristiques admissibles ?



Deux heuristiques admissibles :

$h_1(n)$ = nb de pièces mal placées

$h_2(n)$ = Σ distance de chaque pièce à sa position finale

Dans l'exemple :

$$h_1(S) = 7 \text{ et } h_2(S) = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18$$

On dit que h_2 **domine** h_1 ssi $h_2(n) \geq h_1(n)$ pour tout n . Dans ce cas, on choisit l'heuristique dominante.

Une heuristique dominante peut réduire l'espace de recherche considérablement.

Algorithmes d'amélioration itérative

- À utiliser lorsque le chemin qui mène vers une solution n'est pas important.
- Idée : améliorer l'état final lui même.

Algorithmes d'amélioration itérative

- **Problème** : se retrouver bloqué dans une solution locale (et pas globale)
- **Solution** : sortir de la localité en autorisant des changements drastiques de temps en temps.