

Fondement de l'Intelligence Artificielle



Abir CHaabani

abir.chaabani@gmail.com

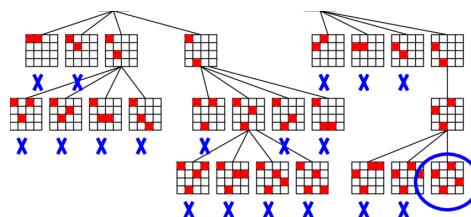
abir.chaabani@enicar.ucar.tn

Université de Carthage - ENICARTHAGE

2 GINF 2024/2025

1

Chapitre 2



Résolution des problèmes par exploration

2

Plan chapitre 2

- Introduction
- Agents de résolution de problèmes
- Problèmes bien définis
- Exemples de problèmes
- Recherche de solutions
- Stratégies d'exploration
 - => Non informées
 - => Informées (heuristiques)
- Fonctions heuristiques
- Stratégies d'exploration locales

3

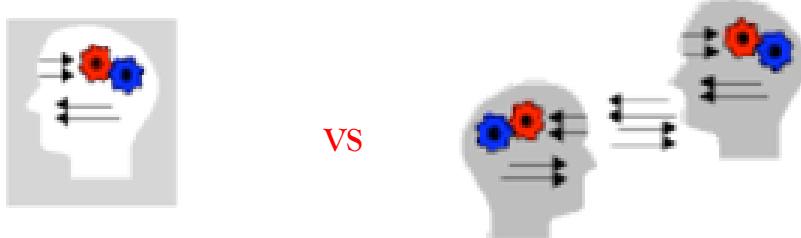
Partie 1



Agent de résolution de problème

4

Introduction



- ➔ 1985: Le domaine de l'intelligence artificielle distribuée (IAD) est apparu.
- ➔ Transfert du raisonnement

5

Agent: définition

- Un agent est tout ce qui peut être compris comme **percevant** son environnement à travers des senseurs et **agissant** sur cet environnement par l'intermédiaire d'effecteurs (Russel Norvig 95)
- Un agent est une entité qui fonctionne continuellement et de manière **autonome** dans un environnement où d'autres processus se **déroulent et d'autres agents** existent (Shoham, 1993)
- Un agent est une entité **autonome, réelle ou abstraite**, qui est capable **d'agir sur elle-même et sur son environnement**, qui, dans un univers multi-agents, peut **communiquer avec d'autres agents**, et dont **le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents** (Ferber, 1995)

6

Agent: architecture

2 types d'architectures d'agents [Russel et Norvig 95] :

Les agents à réflexes simples

Les agents conservant une trace du monde

Agents réactifs

Les agents ayant des buts

Les agents utilisant une fonction d'utilité

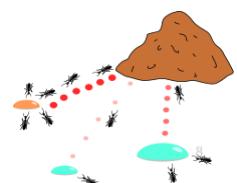
Agents cognitifs

7

Agents réactifs

- Les agents réactifs sont basés sur l'intelligence artificielle réactive. Cette école de l'intelligence artificielle est basée sur la possibilité de concevoir des comportements intelligents à partir de comportements simples
- Un agent réactif ne fait que réagir aux changements qui surviennent dans l'environnement.
- Un tel agent ne fait ni délibération ni planification, il se contente simplement d'acquérir des perceptions et de réagir à celles-ci en appliquant certaines règles prédefinies.
- Étant donné qu'il n'y a pratiquement pas de raisonnement, ces agents peuvent agir et réagir très rapidement (comme le réflexe humain)

En effet, l'intelligence est un phénomène émergent de l'interaction des entités simples



Agents cognitifs

- Cette classe des agents fait partie de l'intelligence artificielle symbolique.
- L'agent possède une représentation symbolique de **son environnement, des autres agents, des actions et des états internes de l'agent.**
- Le raisonnement de l'agent consiste à trouver l'ensemble des actions possibles pour satisfaire ses buts (modélisés comme des états internes)
- Différents attributs dans le processus de raisonnement sont pris en compte : les croyances de l'agent sur les croyances des autres agents, l'historique de l'exécution de l'agent, etc.
 - l'intégration de divers attributs peut compliquer le processus de raisonnement et la performance devient une question pertinente dans cette classe d'agents.
 - Un système multi-agent de cette catégorie est composé d'un petit nombre d'agents **hétérogènes.**



Agents cognitifs

- Un **agent cognitif** est considéré comme "intelligent" dans le sens où il présente des caractéristiques qui sont généralement associées à l'intelligence humaine, telles que
 1. **Flexibilité:** est la capacité de l'agent de changer son comportement ou sa structure afin d'atteindre ses objectifs.

Exemple: un joueur de football, peut être doté de capacité d'attaque et/ou de capacité de défense. Sachant que l'objectif du joueur est d'aider son équipe à vaincre l'adversaire, l'attaquant a pour but de marquer des buts et le défenseur a pour but d'empêcher l'adversaire de marquer un but. Alors, **un joueur attaquant flexible** c'est un attaquant qui retourne à la zone de défense pour donner l'aide à ses collègues en cas d'une attaque de l'adversaire. De même, un défenseur peut aider les attaquants en cas d'un contre attaque par exemple.

Agents cognitifs

- Un **agent cognitif** est considéré comme "intelligent" dans le sens où il présente des caractéristiques qui sont généralement associées à l'intelligence humaine, telles que
 - 1. Flexibilité:** est la capacité de l'agent de changer son comportement ou sa structure afin d'atteindre ses objectifs.

Exemple: un joueur de football, peut être doté de capacité d'attaque et/ou de capacité de défense. Sachant que l'objectif du joueur est d'aider son équipe à vaincre l'adversaire, l'attaquant a pour but de marquer des buts et le défenseur a pour but d'empêcher l'adversaire de marquer un but. Alors, **un joueur attaquant flexible** c'est un attaquant qui retourne à la zone de défense pour donner l'aide à ses collègues en cas d'une attaque de l'adversaire. De même, un défenseur peut aider les attaquants en cas d'un contre attaque par exemple.

Agents cognitifs

- 2. Perception de l'environnement :** Il est capable de percevoir son environnement de manière active (comme le ferait un humain avec ses sens).
- 3. Apprentissage A:** Il peut apprendre de ses expériences, s'adapter à de nouvelles situations et améliorer ses actions et ses décisions au fil du temps.
- 4. Raisonnement et prise de décision :** Un agent cognitif peut réfléchir, analyser des situations complexes, et prendre des décisions basées sur des critères logiques.
- 5. Adaptabilité :** Il peut s'ajuster aux changements dans son environnement sans intervention externe, ce qui lui permet de s'adapter à des situations imprévues.

Agents de résolution de problème

- Un agent **intelligent** doit maximiser sa mesure de performance en formulant un but et en essayant de le satisfaire.
- Un agent de résolution de problèmes est un système intelligent capable de trouver une solution à un problème donné en explorant un espace de recherche.



13

Formulation de problème

Exemple : Agent en vacances à Tunis et doit rejoindre Tozeur, doit formuler son but selon la situation actuelle et la mesure de performance

==> L'agent doit découvrir comment agir pour arriver à un état but : **aller à Tozeur**.

==> Il doit donc choisir un ensemble d'actions lui permettant d'atteindre son but.

==> Mais il ne peut pas considérer directement l'état but, il faut le décomposer en sous-but : c'est la **formulation de problème**.



Processus consistant à décider quelles actions et quels états considérer en vue d'un but donné :

✓ **États** : les différentes villes,

✓ **Actions** : déplacement entre les villes.

14

Exploration du problème



Processus de recherche d'une séquence d'actions qui atteint le but.

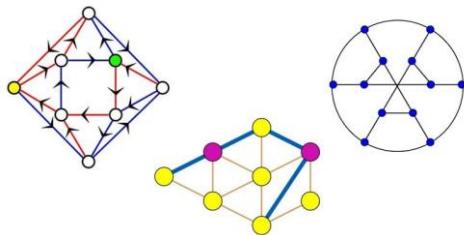
- ***Un algorithme d'exploration*** prend en entrée un problème et renvoie une solution optimale sous forme **d'une séquence d'actions**

Exécution



- Les actions recommandées sont réellement effectuées.

Partie 2



Problème bien défini

17

Problème: Définition

- **Un problème** est une collection d'informations que l'agent (décideur) utilise pour décider quelle(s) action(s) accomplir.
- Définir un problème c'est choisir une abstraction de la réalité en termes :
 - Identification **d'un état initial** (donc choix d'un langage de description d'états du problème)
 - Identification des actions possibles par définition **d'opérateurs de changement d'état** (donc définition de l'ensemble des états possibles du problème)
- Essentiellement : information = états + actions.

18

Problème: 5 composants

1. L'état du problème : une ville,
2. Les actions : passer d'un état à un autre,
3. Modèle de transition/ fonction successeur : définit l'état résultant de l'exécution d'une action à partir d'un état,
4. Test de buts : détermine si un état donné est un état but
5. Le coût du chemin : une fonction de coût associant à chaque action un nombre non-négative (le coût de l'action) , ex. coût numérique de performance de l'agent (longueur en km).

19

Espace d'états

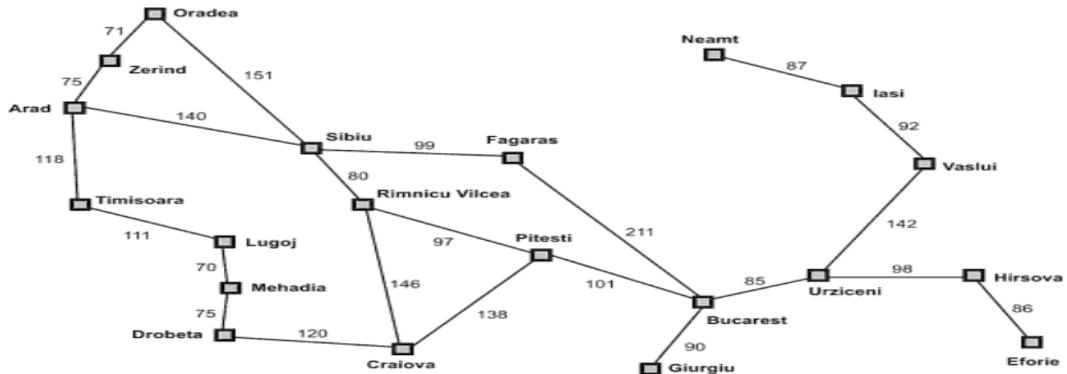
- On appelle espace de recherche (ou **espace des états**) d'un problème l'ensemble **des états atteignables depuis l'état initial** par n'importe quelle séquence d'actions **n**

Espace d'états = état initial + actions + modèle de transitions.

- Un espace de recherche peut être représenté par un graphe orienté
 1. **Les nœuds** sont les états
 - Il y a (au moins) **un état initial**
 - Il y a (au moins) **un état final**.
 2. **Les arcs** sont les actions
 3. **Un chemin** à travers un graphe représente alors **une suite d'actions** prises allant d'un état initial vers un état final

Graph de l'espace d'état du problème agent en Roumanie

Abstraction : suppression des détails : paysage, météo, radio,



- Les villes de Roumanie (nœuds du graphe).

21

- Les routes entre les villes (arêtes du graphe, avec des coûts de déplacement).

Exemples de problème

1. Problèmes de jouets (Toys problems) : illustrer ou expérimenter diverses méthodes de résolution de problèmes (recherche):

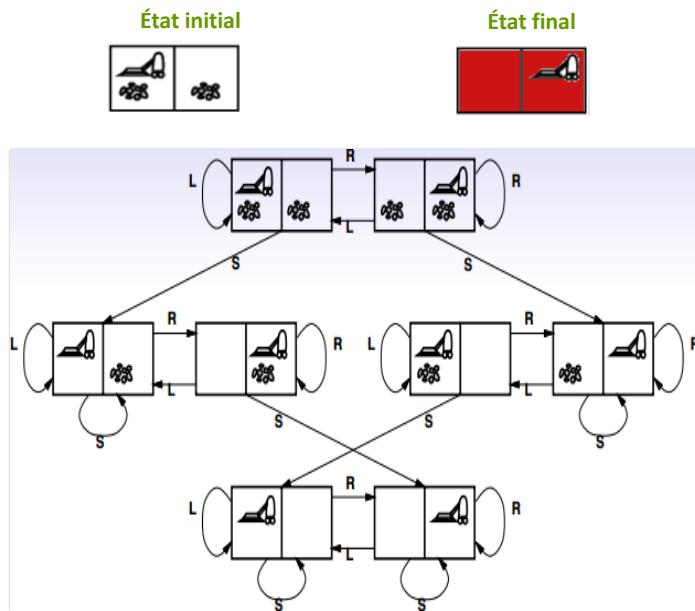
- ✓ Jeu de taquin,
- ✓ Problème des tours de Hanoi,
- ✓ Problème du chien, de la chèvre et du chou,
- ✓ L'aspirateur,
- ✓ Le problème des huit reines,
- ✓ Les mots croisés,
- ✓ Les jeux d'échecs, de dames,...

2. Problèmes du monde réel : un problème réel dont la description est compliquée, mais on peut trouver une idée de leur formalisation :

- ✓ Certains sont des extensions du problème de Roumanie (systèmes embarqués dans les voitures, voyageur de commerce),
- ✓ D'autres sont plus complexes (routage de flux vidéo, planification d'opérations militaires).

22

Exemple 1: Aspirateur



Éléments du graphe

Etats (Nœuds)

Les emplacements de l'agent et de la poussière

Actions (Arcs)

Déplacer à droite, à gauche, aspire

État initial :

Un état arbitraire

Coût

1 par déplacement

Test du but

Vérifie que le sol est propre: deux pièces doivent être propres

Exemple 2: Agent en Roumanie



Comment un agent peut-il trouver le chemin optimal entre deux villes ?

Éléments du graphe

Etats (Nœuds)

Graphe des points de la carte

Actions (Arcs)

Déplacer vers l'arrêts sortantes d'un point

État initial :

Son point de départ

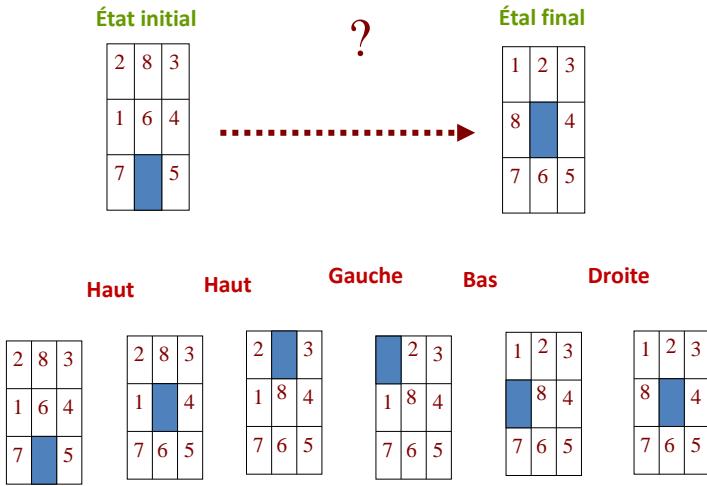
Coût

Distance, durée, difficulté, danger

Test du but

Vérifier si on a arrivé vers la destination finale

Exemple 3: Jeu de Taquin



Éléments du graphe

Etats (Nœuds)

Un état est une configuration du jeu de taquin

Actions (Arcs)

Déplacer une case à droite, gauche, en haut, en bas

État initial

Un état arbitraire

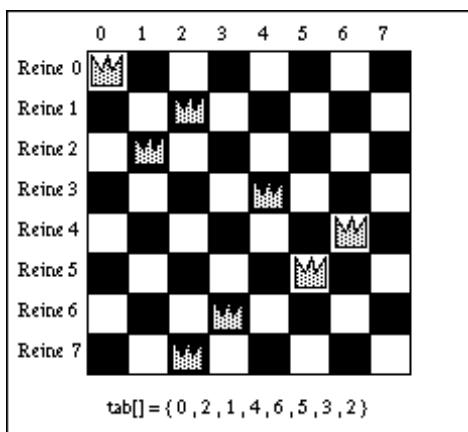
Coût

Un point par déplacement, nombre de cases mal placées, etc.,

Test du but

Vérifier l'état actuelle de la grille

Exemple 4: Huit reines



Éléments du graphe

Etats (Nœuds)

Configuration de 0 à 8 reines sur la grille sans conflit

Actions (Arcs)

Ajouter une reine à une position valide.

État initial

La grille vide

Coût

Un coût constant pour chaque action

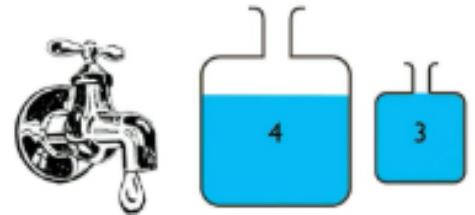
Test du but

Configuration de huit reines avec aucune reine sous attaque

Problème 1: Représentation du problème du bidon

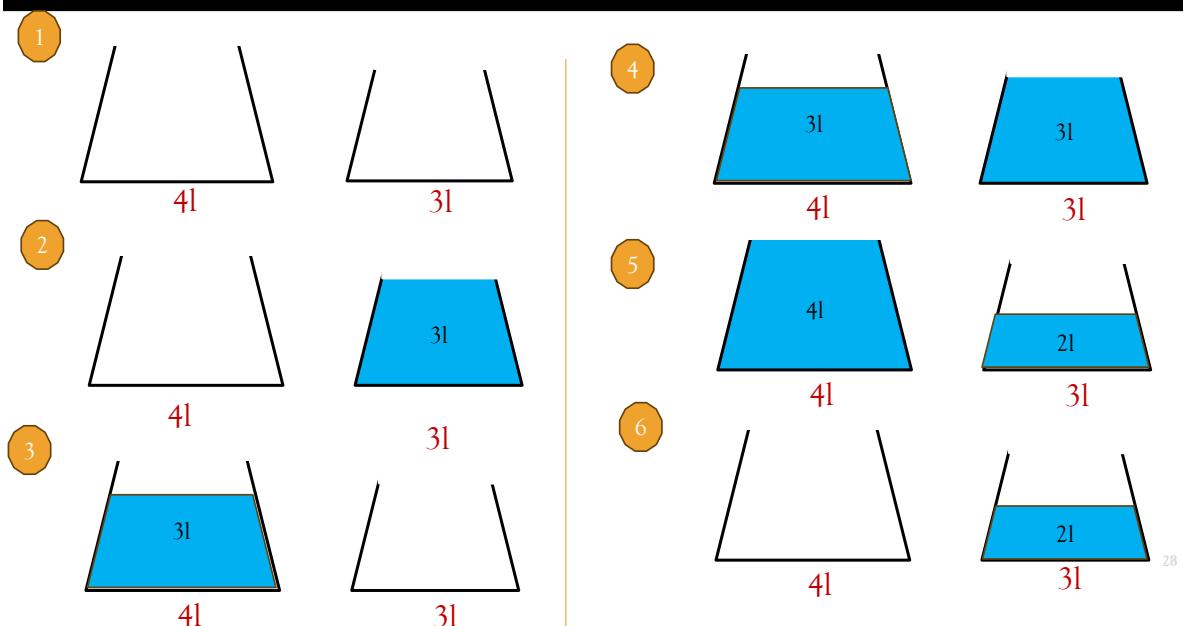
- Vous disposez de 2 bidons: un de 4 gallons et l'autre de 3 gallons, vides.
- Il n'y a aucune marque de graduation.
- Vous disposez également d'une pompe capable de remplir les bidons.

1. Comment remplir le bidon de 4 gallons avec exactement 2 gallons d'eau ?
2. Représenter l'espace d'états du problème de bidons.



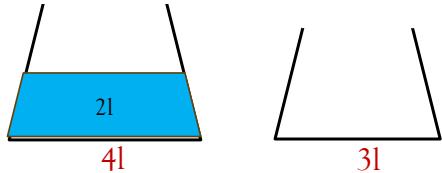
27

Application: Résolution du problème du bidon



Application: Résolution du problème du bidon

7



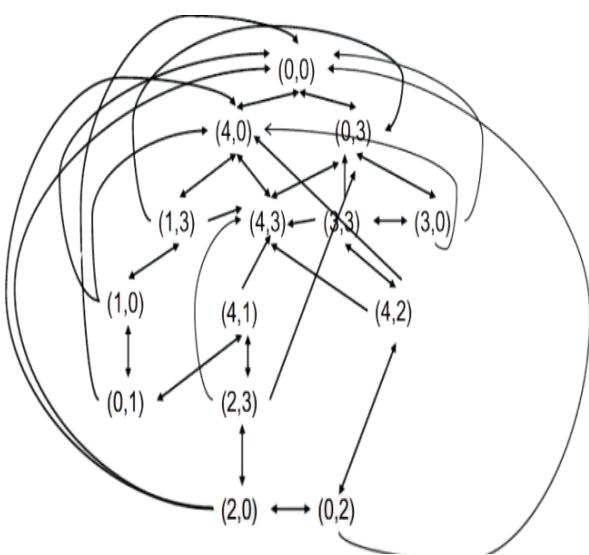
Solution: L'espace d'états du problème des bidons peut être représenté sous forme de graphe, où chaque nœud représente un état donné des bidons, et chaque arête représente une action possible.

- **Un état** peut être représenté par un couple (x,y) s où :
 - ✓ x est la quantité d'eau dans le bidon de 4 gallons (max 4).
 - ✓ y est la quantité d'eau dans le bidon de 3 gallons (max 3)
 - ✓ L'état initial est $(0,0)$ (les deux bidons sont vides).
 - ✓ L'état but est $(2, 0)$
- **Action** : Remplir un bidon/ Vider un bidon/ Transférer l'eau d'un bidon à l'autre

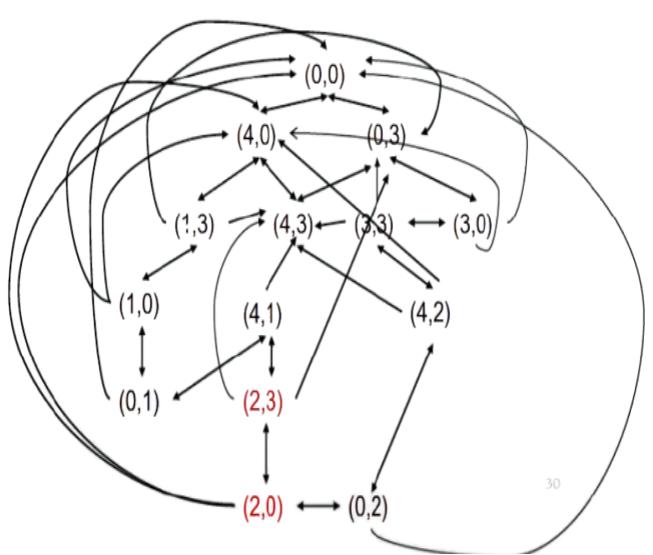
29

Application: Représentation du problème du bidon

Graphe d'état



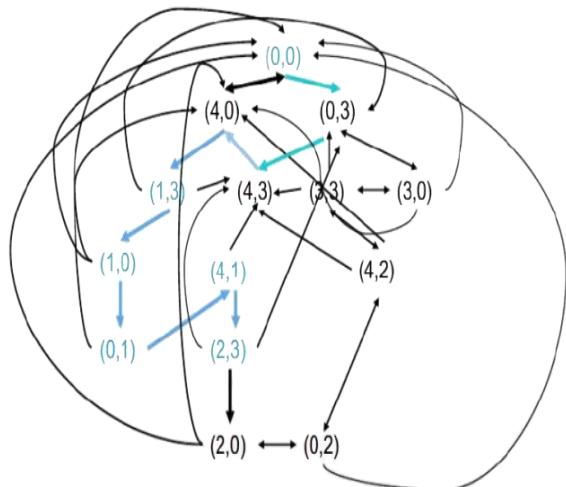
Etat buts dans espace d'états



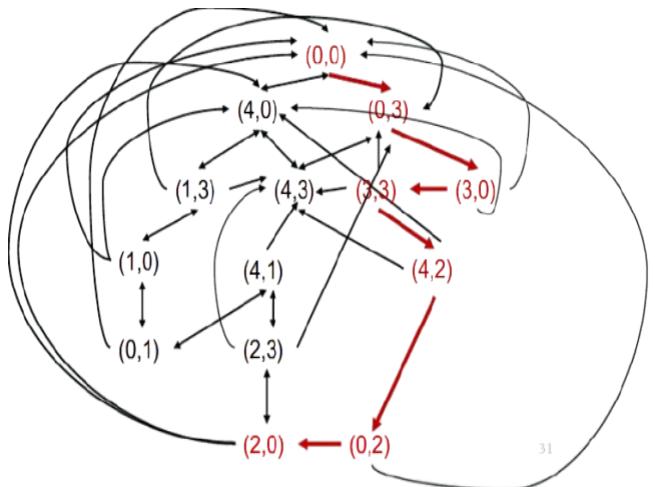
30

Application: Représentation du problème du bidon

Chemin dans espace d'état



Une solution optimale du problème de bidon



31

Problème 2: Les missionnaires et les cannibales

Trois missionnaires et trois cannibales se trouvent sur la même rive d'une rivière. Ils voudraient tous se rendre sur l'autre rive. Cependant, si le nombre de cannibales est supérieur à celui des missionnaires, alors les cannibales mangeront les missionnaires. Il faut donc que le nombre de missionnaires présents sur l'une ou l'autre des rives soit toujours supérieur à celui des cannibales. Le seul bateau disponible ne peut pas supporter le poids de plus de deux personnes.



Comment est-ce que tout le monde peut traverser la rivière sans que les missionnaires risquent d'être mangé?

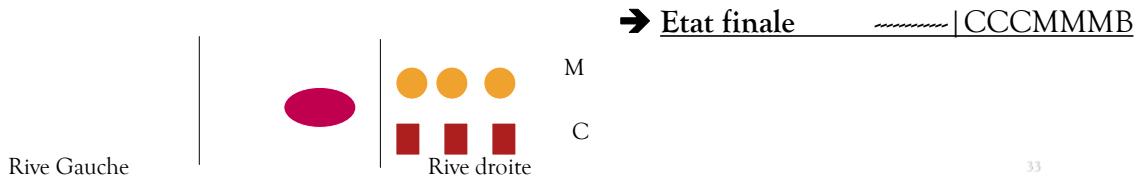
32

Problème 2: Représentation du problème

❖ Configuration initiale:



❖ Configuration finale

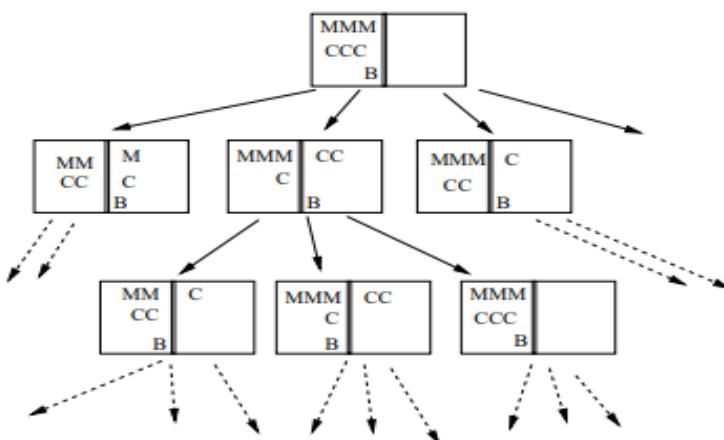


33

Problème 2: Représentation du problème

Contraintes

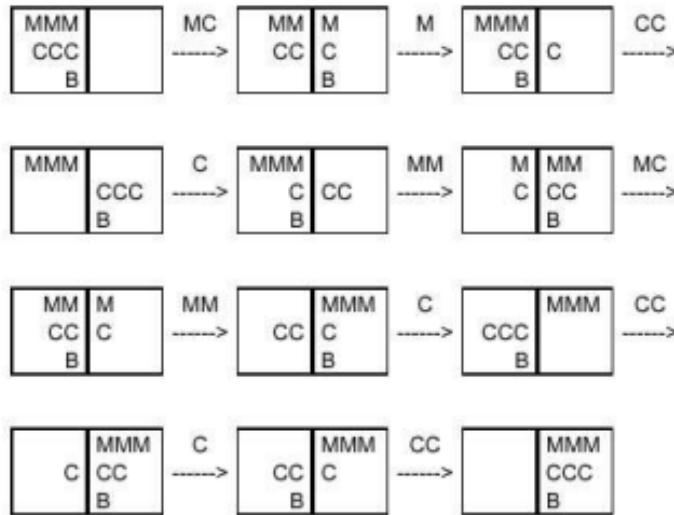
1. Les cannibales ne doivent pas être plus nombreux que les missionnaires sur les deux rives
2. Le bateau ne peut pas supporter plus de deux personnes.



34

Problème 2: résolution du problème

Solution



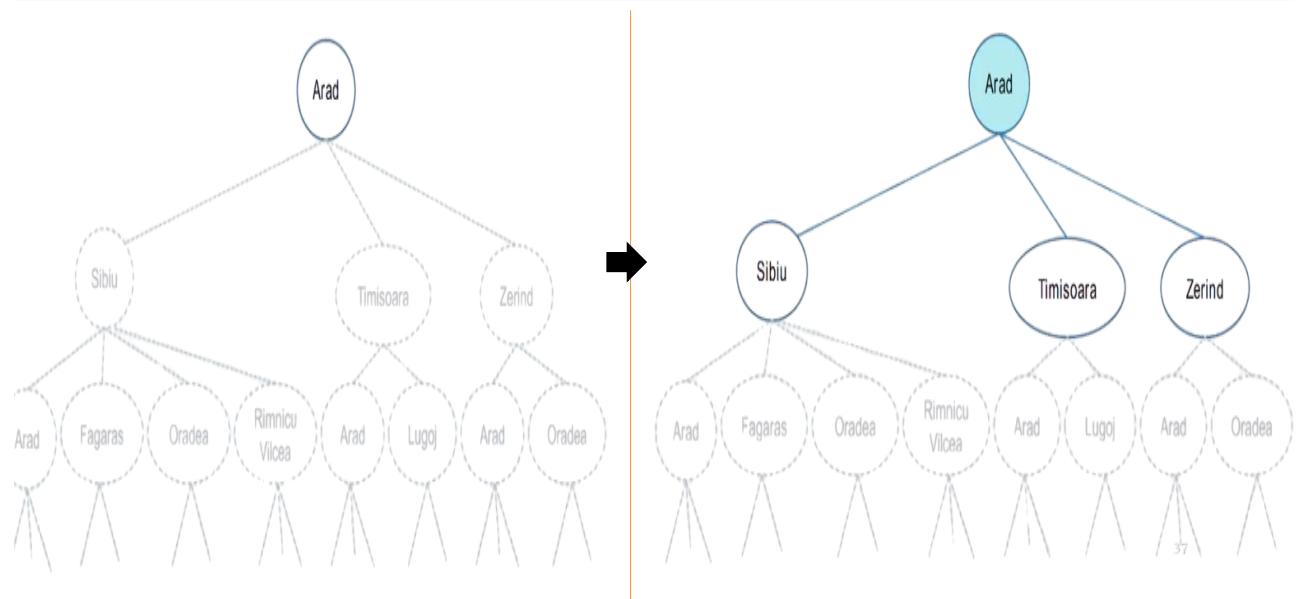
35

Recherche de solutions

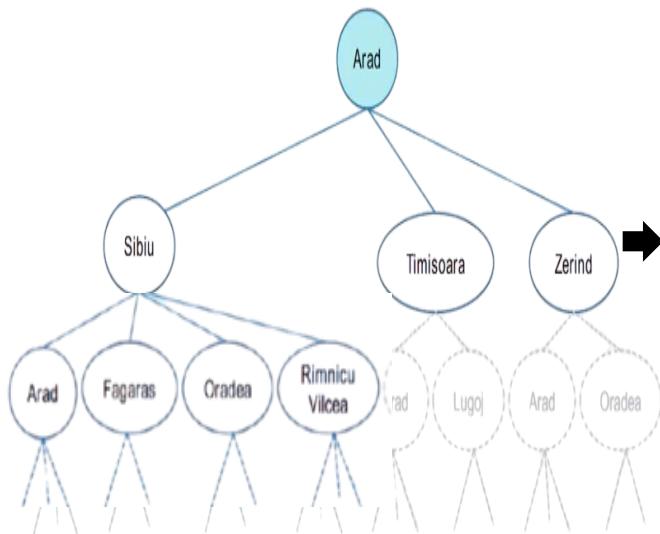
- Une solution est une séquence d'actions.
- **Les algorithmes d'exploration** examinent diverses séquences d'actions possibles à partir de l'état initial, c'est **l'arbre d'exploration** (\neq espace d'états) :
 - État initial : racine,
 - Branches : actions,
 - Nœuds : états de l'espace d'états du problème.

36

Arbre d'exploration partielle de l'itinéraire Arad-Bucarest



Arbre d'exploration partielle de l'itinéraire Arad-Bucarest

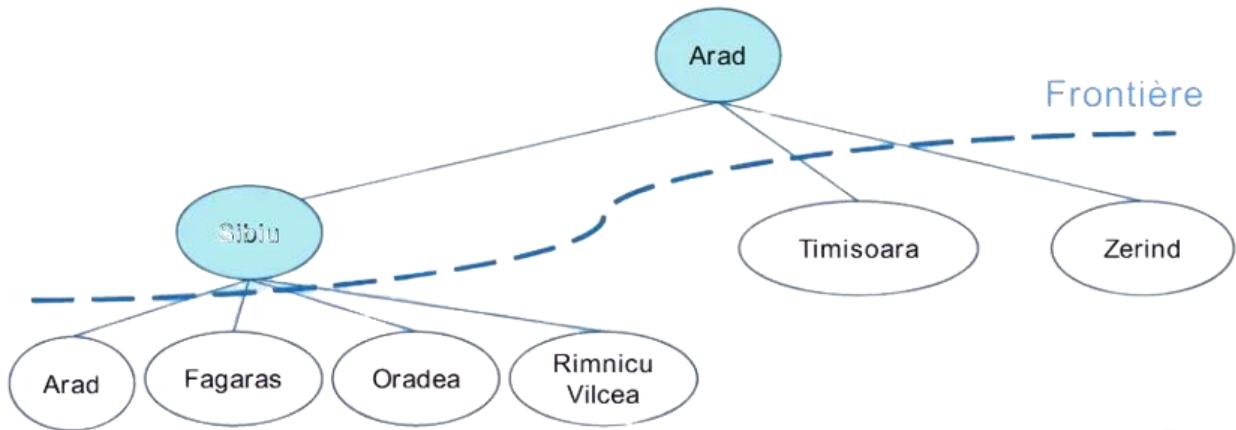


1. Le nœud racine est le nœud initial (Arad).
2. On teste si l'état courant (Arad) est un état but.
3. Si oui, on renvoie la solution et arrêt.
4. Si non, et si l'état peut être développé, on développe l'état courant (fils : Sibiu, Timisoara, Zerind).
5. On sélectionne l'état à examiner, les autres restent en attente.
6. On revient à 2 avec l'état sélectionné est l'état courant.

38

Frontières

- Nœuds feuilles générés mais non encore développés.



39

Algorithme générale d'exploration en arbre

Fonction Explorer-Arbre (problème) retourne une solution ou échec

Initialisation: Initialiser la frontière avec l'état initial du problème

// Exploration itérative:

Tant que la frontière n'est pas vide faire

 Sélectionner un nœud feuille ([à l'aide d'une stratégie](#)) et l'enlever de la frontière

Si le nœud contient un état but **alors**

 Retourner la solution correspondante

Sinon

 Développer le nœud en générant ses successeurs

 Ajouter les successeurs à la frontière

// Arrêt : Aucun état but trouvé

 Retourner échec

➔ L'idée principale de cet algorithme est d'explorer un **espace d'états** sous forme d'un **arbre de recherche**, afin de trouver une solution à un problème donné⁴⁰

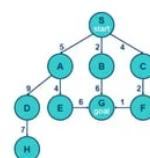
Chemins avec boucles et chemins redondants

- Dans un arbre de recherche, chaque chemin est unique car un arbre n'a pas de cycles. Cependant, si nous explorons **un graphe**, il est possible d'avoir :
 - Chemins avec boucles (cas particulier) :**
 - Présence d'états répétés,
 - Arbre d'exploration infini.
 - Chemins redondants :**
 - Plusieurs façons d'aller d'un état à un autre.
 - Problèmes impraticables même si on évite les boucles.

41

Exploration

- Les algorithmes d'exploration ont la même structure de base, ils diffèrent par **la stratégie d'exploration**.
 - Noeud de recherche :
 - ✓ **État** : l'état de l'espace des états.
 - ✓ **Noeud parent** : le noeud dans l'arbre d'exploration qui a produit ce noeud.
 - ✓ **Action** : L'action qui a été appliquée au parent pour générer ce noeud.
 - ✓ **Coût du chemin** : coût $g(n)$ du chemin à partir de l'état initial jusqu'à ce noeud.
 - Noeud ≠ état.



42

Evaluation de la résolution de problème

1. Complétude : si une solution existe, l'algorithme garanti son obtention.

Définition : Un algorithme est **complet** s'il garantit de trouver une solution si elle existe.

2. Optimalité : la solution trouvée est la meilleure.

Définition : Un algorithme est **optimal** s'il garantit de trouver la **meilleure solution** (celle avec le **coût minimal**).

3. Complexité en temps : Temps nécessaire pour trouver la solution.

4. Complexité en espace : mémoire nécessaire pour effectuer l'exploration.

Complétude: Est-ce que l'algorithme trouve toujours une solution s'il y en a une ?

Optimalité: Est-ce que l'algorithme renvoie toujours des solutions optimales ?

Complexité en temps: combien de noeuds faut-il produire pour trouver la solution?

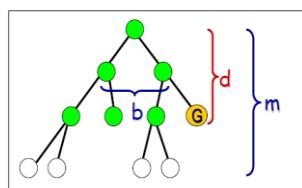
43

Complexité en espace: nombre maximum de noeuds à conserver en mémoire pour trouver la solution ?

Complexité

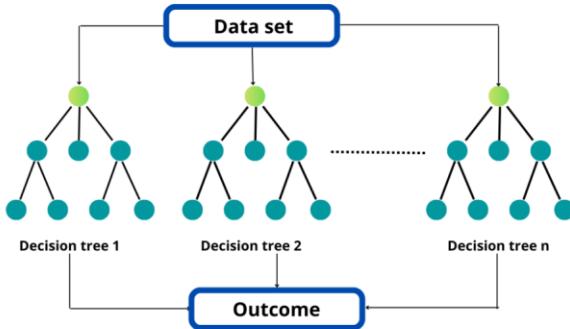
La complexité en temps et en espace dépend de :

- **b** : facteur de branchement, c'est le nombre maximal de successeurs d'un nœud donné,
- **d** : profondeur à laquelle se trouve le meilleur nœud solution (moins d'étapes depuis la racine),
- **m** : longueur maximale d'un chemin dans l'espace d'états.



44

Partie 3



Stratégies de recherche

45

Type de stratégies de recherche

1. Exploration Non-informée (Aveugle)

L'algorithme n'a aucune information supplémentaire autre que celles contenues dans la définition du problème. Il ne sait pas si un état est plus proche ou plus prometteur qu'un autre.

- => Pas d'autres informations sur les états que celles fournies dans la définition du problème.
- => Elles génèrent des successeurs et distinguent un état final d'un état non final.

2. Exploration informée (heuristique)

L'algorithme utilise une fonction heuristique pour évaluer la qualité des états non finaux et orienter la recherche vers les plus prometteurs.

- => Peuvent déterminer si un état non final est meilleur qu'un autre.

46

1. Stratégies de Recherche non-informées

- N'exploitent **aucune information** sur la structure de l'arbre ou la présence potentielle de nœuds-solution pour **optimiser la recherche** :
 - Exploration en largeur d'abord**
 - Exploration à coût uniforme
 - Exploration en profondeur d'abord
 - Exploration en profondeur avec libération de mémoire
 - Exploration en profondeur limitée
 - Exploration itérative en profondeur
 - Exploration bidirectionnelle
- La plupart des problèmes réels sont susceptibles de provoquer une **explosion combinatoire** du nombre d'états possibles.

47

Exploration en largeur

Principe

- On commence par développer le nœud racine puis tous les nœuds successeurs, puis les successeurs des successeurs, ...
- Tous les nœuds à une profondeur **i** sont développés avant ceux de niveau **i+1**.

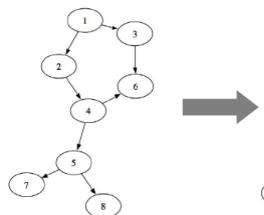


Figure 1

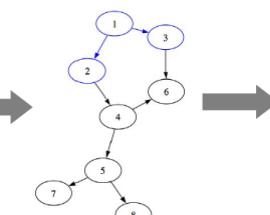


Figure 2

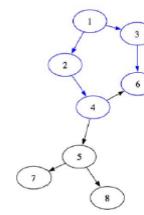


Figure 3

=> Le parcours en largeur de ce graphe donnerait la liste suivante : 1 2 3 4 6 5 7 8. Une autre solution serait la suivante : 1 3 2 6 4 5 8 7. Elles sont toutes les deux valables

Algorithme de recherche en largeur d'abord: Breadth-First Search (BFS)

Algorithme Parcours en largeur BFS(G, s, but): Données : graphe G , sommet de départ s , sommet but

Variables : File q (initialisée à vide)

Tableau marque des sommets (initialisé à Faux)

Début

marque[s] \leftarrow Vrai; // Marquer le sommet de départ comme visité

enfiler(s, q); // Ajouter le sommet s à la file

Tant que q n'est pas vide faire

$u \leftarrow \text{tête}(q)$; // Dépiler un sommet de la file

Si $u = \text{but}$ alors // Vérifier si u est le noeud but

Retourner solution(u) // Fin de l'algorithme, on arrête tout

Fin Si

Pour chaque voisin v de u faire

Si $\text{marque}[v]$ est Faux alors // Si v n'a pas été visité

marque[v] \leftarrow Vrai; // Marquer v comme visité

enfiler(v, q); // Ajouter v à la fin de la file

Fin Si

Fin Pour

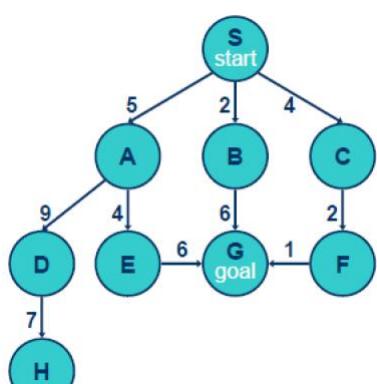
défiler(q); // Retirer u de la file

Fin Tant que Fin

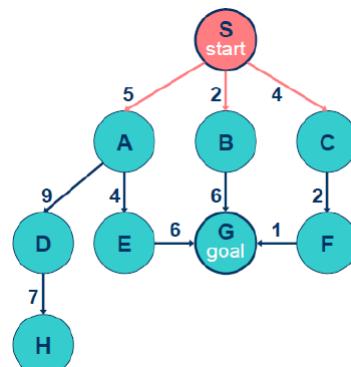
Algorithme de recherche en largeur d'abord: Breadth-First Search (BFS)

Exemple 1

expnd. node	nodes list
	{S}



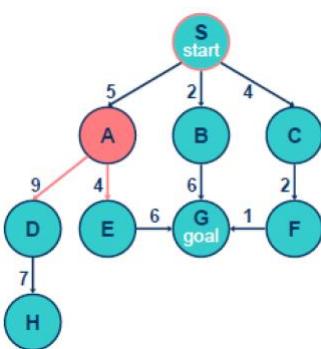
expnd. node	nodes list
	{S}
S not goal	{A,B,C}



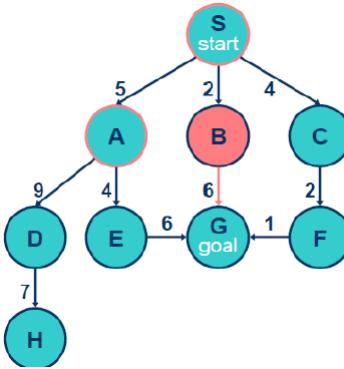
Algorithme de recherche en largeur d'abord: Breadth-First Search (BFS)

Exemple 1

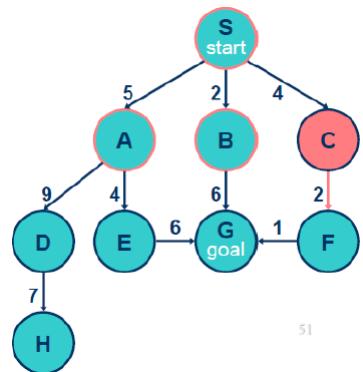
expnd. node	nodes list
S	{S}
A	{A,B,C}
B not goal	{B,C,D,E}



expnd. node	nodes list
S	{S}
A	{A,B,C}
B	{B,C,D,E}
G not goal	{C,D,E,G}



expnd. node	nodes list
S	{S}
A	{A,B,C}
B	{B,C,D,E}
C	{C,D,E,G}
D	{D,E,G,F}

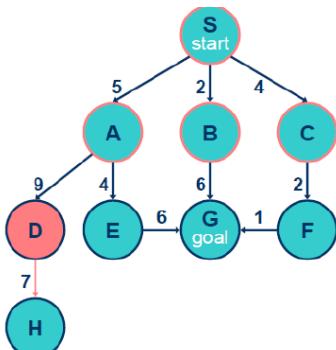


51

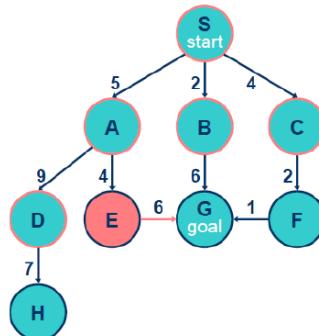
Algorithme de recherche en largeur d'abord: Breadth-First Search (BFS)

Exemple 1

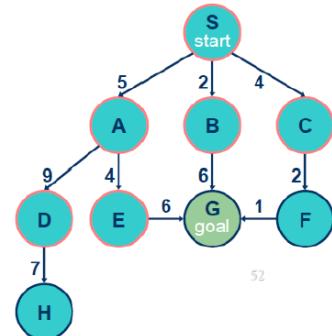
expnd. node	nodes list
S	{S}
A	{A,B,C}
B	{B,C,D,E}
C	{C,D,E,G}
D	{D,E,G,F}
E not goal	{E,G,F,H,G}



expnd. node	nodes list
S	{S}
A	{A,B,C}
B	{B,C,D,E}
C	{C,D,E,G}
D	{D,E,G,F}
E	{G,F,H,G}



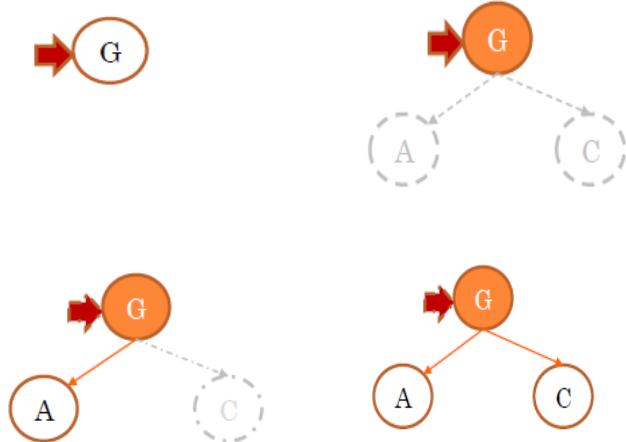
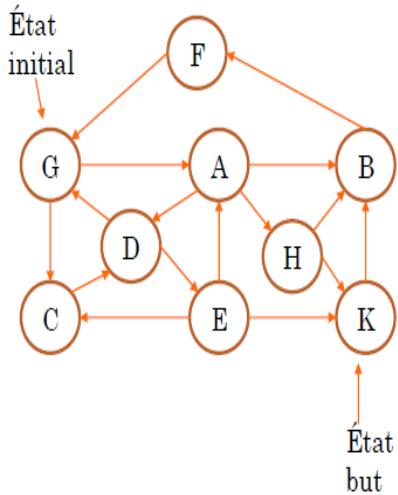
expnd. node	nodes list
S	{S}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G goal	{F,H,G} no expand



52

Algorithme de recherche en largeur d'abord: Breadth-First Search (BFS)

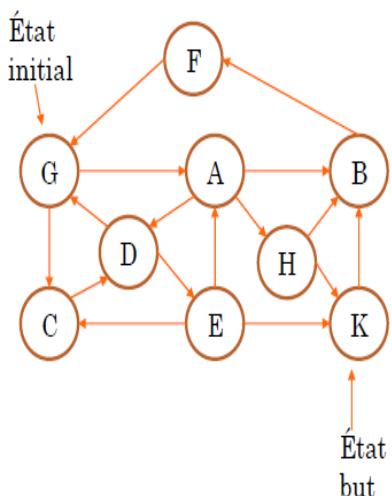
Exemple 2



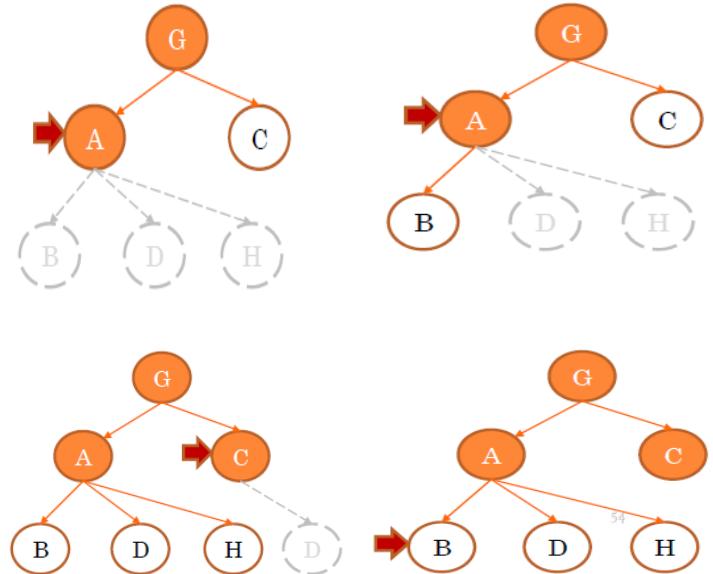
53

Algorithme de recherche en largeur d'abord: Breadth-First Search (BFS)

Exemple 2



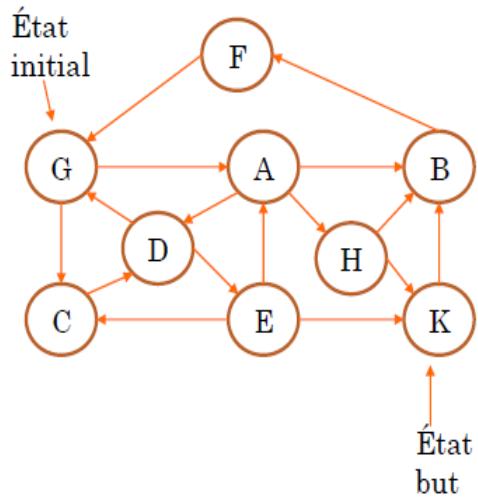
Pas d'ajout du fils de C (D) à la frontière car il y est déjà



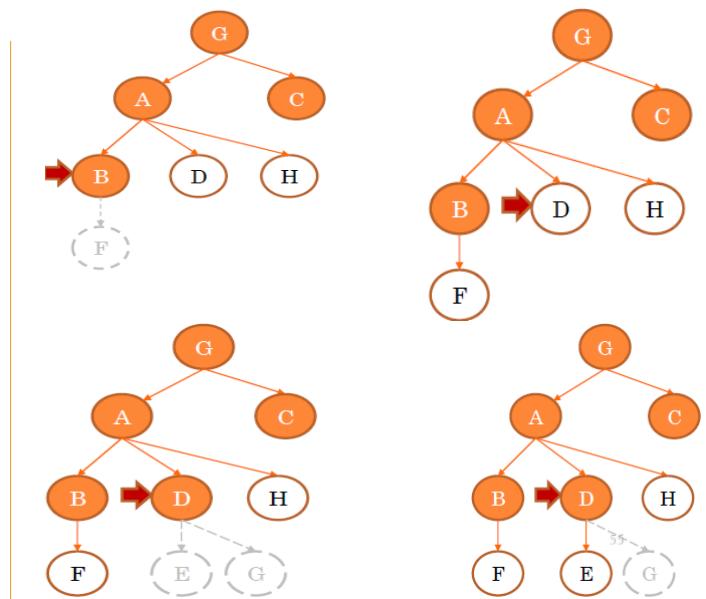
54

Algorithme de recherche en largeur d'abord: Breadth-First Search (BFS)

Exemple 2

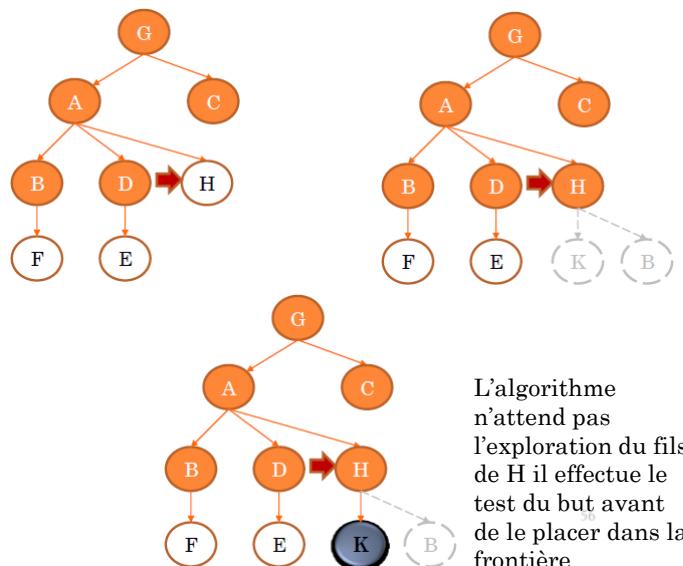
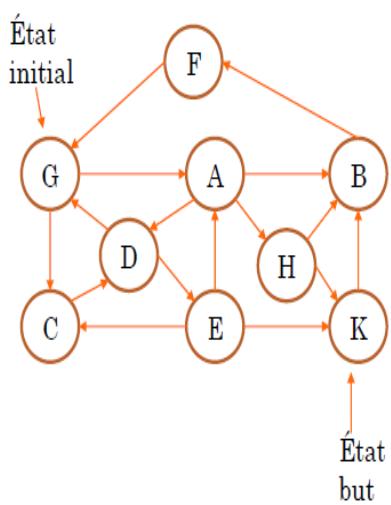


Pas d'ajout du fils de D (G) à la frontière car il a déjà été exploré



Algorithme de recherche en largeur d'abord: Breadth-First Search (BFS)

Exemple 2



x Algorithme de recherche en largeur d'abord: Performance

- **Complétude** : oui, si le nœud but se trouve à une profondeur d , l'algorithme le trouvera une fois qu'il génère les nœuds des niveaux précédents, mais b doit être fini.
==>Condition : le facteur de branchement b (nombre de successeurs par nœud) doit être fini, sinon BFS pourrait ne jamais terminer.
- **Optimalité** : pas nécessairement, si tous les coûts de transition entre les nœuds sont égaux, BFS trouve bien le chemin le plus court. Mais si les coûts varient, BFS ne garantit pas d'explorer d'abord le chemin le moins coûteux.
==>Condition pour la garantir il faut que le coût du chemin soit une fonction non décroissante de la profondeur du nœud (ex. coût fixe)

x Algorithme de recherche en largeur d'abord: Performance

Complexité temporelle : $1+b+b^2+b^3+\dots+b^d=O(b^d)$.

==> Si le test de but se fait une fois le nœud choisi (comme l'algorithme général) et non une fois produit la complexité serait $O(b^{d+1})$.

Complexité spatiale :

L'algorithme BFS utilise une **file (FIFO)** pour stocker les nœuds à explorer. Il doit mémoriser deux types de nœuds

==>(1) **Les nœuds déjà explorés** (visités et retirés de la file), Lorsque BFS atteint le dernier niveau d , il aura déjà exploré tous les niveaux précédents :

$1+b+b^2+b^3+\dots+b^{d-1}$ Cette somme est dominée par son dernier terme, soit $O(b^{d-1})$

==>(2) **Et les nœuds en attente d'exploration** (encore dans la file): $O(b^d)$ sur la frontière.

58

BFS: Résumé

- Efficace si on ne sait pas par où se trouve le but (toutes les étapes ont le même coût).
- Mais une perte de temps et d'espace si on connaît plus d'informations sur notre destination.
- N'est pas adaptée aux problèmes exponentiels :

Profondeur	Nœuds $b=10$	Temps 1 000 000 nœuds/seconde	Mémoire 1000 octets/nœud
12	10^{12}	13 jours	1 pétaoctets
16	10^{16}	350 années	10 exaoctets

59

Stratégies de Recherche non-informées

- N'exploitent **aucune information** sur la structure de l'arbre ou la présence potentielle de nœuds-solution pour **optimiser la recherche** :
 1. Exploration en largeur d'abord
 2. Exploration à coût uniforme
 3. Exploration en profondeur d'abord
 4. Exploration en profondeur avec libération de mémoire
 5. Exploration en profondeur limitée
 6. Exploration itérative en profondeur
 7. Exploration bidirectionnelle
- La plupart des problèmes réels sont susceptibles de provoquer une **explosion combinatoire** du nombre d'états possibles.

60

Stratégies d'exploration à coût uniforme

L'algorithme d'exploration à coût uniforme (Uniform Cost Search - UCS) est une variante de la recherche en largeur (BFS) qui prend en compte le coût des arcs pour garantir l'exploration du chemin le moins coûteux en premier.

Principes

- Développement du nœud qui a le coût de chemin le plus faible $g(n)$
→ $g(n)$ =coût à partir du nœud initial jusqu'au nœud développé
- La frontière est stockée dans une file triée par coût (g) croissant
- Le test du but est appliqué à un nœud une fois choisi.
- Un test vérifie si un meilleur chemin existe vers un nœud sur la frontière.
- Équivalent à l'exploration en largeur d'abord si tous les coûts des actions sont égaux.

Stratégies d'exploration à coût uniforme

Algorithme UCS(Graph G, Node start, Node goal):

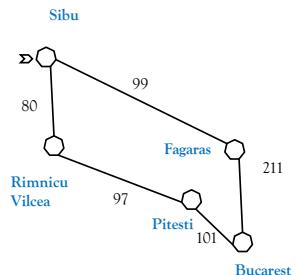
1. Initialiser une file de priorité Q avec (start, coût=0).
2. Initialiser un dictionnaire des coûts {start: 0}.
3. Tant que Q n'est pas vide:
 - a. Extraire le nœud u avec le plus petit coût de Q.
 - b. Si u est le nœud but, retourner le chemin et le coût.
 - c. Pour chaque voisin v de u:
 - i. Calculer le nouveau coût $C = \text{coût}(u) + \text{coût}(u \rightarrow v)$.
 - ii. Si v n'a pas encore été visité ou si C est inférieur au coût connu pour v:
 - Mettre à jour le coût de v.
 - Ajouter (v, C) à la file de priorité.
4. Si Q est vide et que le but n'a pas été trouvé, retourner "Échec".

Stratégies d'exploration à coût uniforme

Ajouter les nœuds dans l'ordre du coût

Sibu
»

File (frontière)	Nœud développé
[(Sibu,0)]	



Sibu
»
Rimnicu
Vilcea

File (frontière)	Nœud développé
[(Sibu,0)]	(Sibu,0)

63

Stratégies d'exploration à coût uniforme

Sibu
»
Rimnicu
Vilcea

File (frontière)	Nœud développé
[(Sibu,0)]	(Sibu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	

Sibu
»
Rimnicu
Vilcea

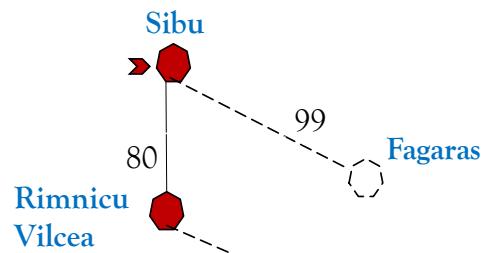
File (frontière)	Nœud développé
[(Sibu,0)]	(Sibu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	(Rimnicu Vilcea,80)

97

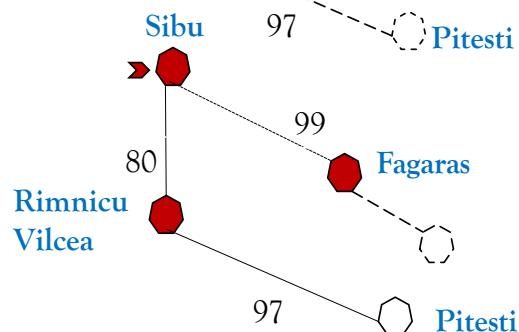
64

Pitesti

Stratégies d'exploration à coût uniforme

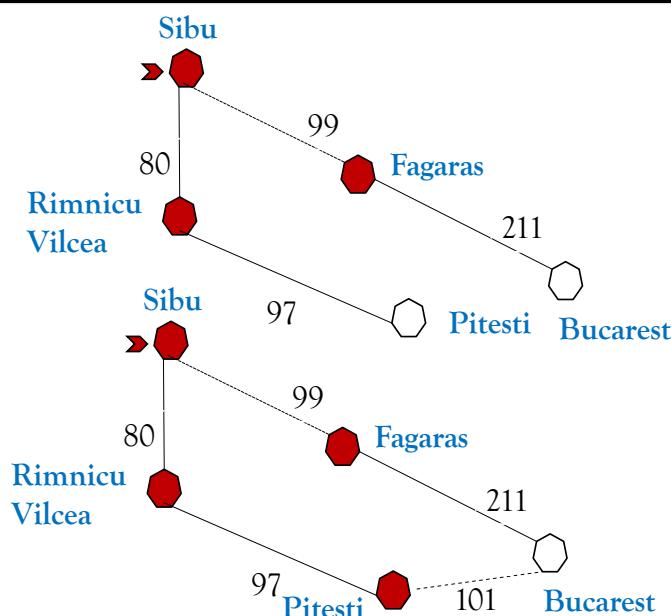


File (frontière)	Nœud développé
$[(Sibiu,0)]$	(Sibiu,0)
$[(Rimnicu\ Vilcea,80), (Fagaras,99)]$	(Rimnicu Vilcea,80)
$[(Fagaras,99), (Pitesti,177)]$	



File (frontière)	Nœud développé
$[(Sibiu,0)]$	(Sibiu,0)
$[(Rimnicu\ Vilcea,80), (Fagaras,99)]$	(Rimnicu Vilcea,80)
$[(Fagaras,99), (Pitesti,177)]$	(Fagaras,99)

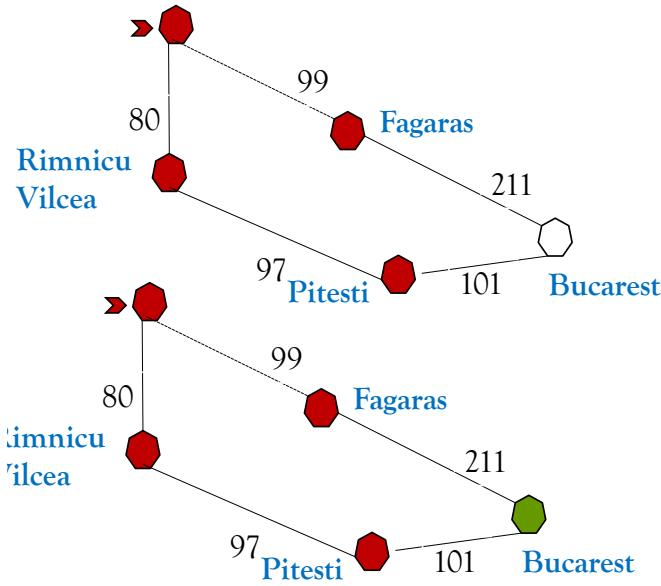
Stratégies d'exploration à coût uniforme



File (frontière)	Nœud développé
$[(Sibiu,0)]$	(Sibiu,0)
$[(Rimnicu\ Vilcea,80), (Fagaras,99)]$	(Rimnicu Vilcea,80)
$[(Fagaras,99), (Pitesti,177)]$	(Fagaras,99)
$[(Pitesti,177), (Bucarest,310)]$	

File (frontière)	Nœud développé
$[(Sibiu,0)]$	(Sibiu,0)
$[(Rimnicu\ Vilcea,80), (Fagaras,99)]$	(Rimnicu Vilcea,80)
$[(Fagaras,99), (Pitesti,177)]$	(Fagaras,99)
$[(Pitesti,177), (Bucarest,310)]$	(Pitesti,177)

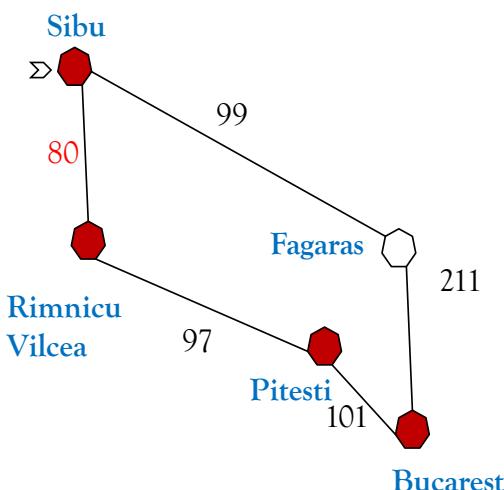
Stratégies d'exploration à coût uniforme



File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[((Rimnicu Vilcea,80), (Fagaras,99))]	(Rimnicu Vilcea,80)
[((Fagaras,99), (Pitesti,177))]	(Fagaras,99)
[((Pitesti,177), (Bucarest,310))]	(Pitesti,177)
[((Bucarest,278))]	

File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[((Rimnicu Vilcea,80), (Fagaras,99))]	(Rimnicu Vilcea,80)
[((Fagaras,99), (Pitesti,177))]	(Fagaras,99)
[((Pitesti,177), (Bucarest,278))]	(Pitesti,177)
[((Bucarest,278))]	(Bucarest,278)

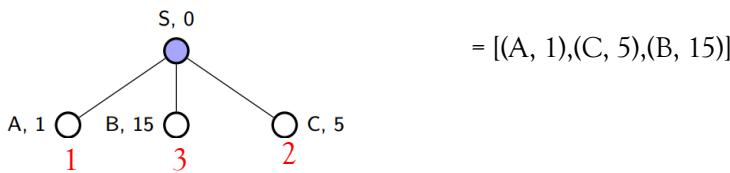
Stratégies d'exploration à coût uniforme



- Le nœud Rimnicu Vilcea (80) est développé en premier.
- Ensuite Fagaras (99), meilleur que Pitesti ($80 + 97 = 177$).
- Ensuite Pitesti ($80 + 97 = 177$), meilleur que Bucarest ($99 + 211 = 310$).
- Ensuite Bucarest à partir de Pitesti ($80 + 97 + 101$) meilleur que 310.
- Le chemin est donc : Sibiu- Rimnicu Vilcea -Pitesti-Bucarest.

Stratégies d'exploration à coût uniforme

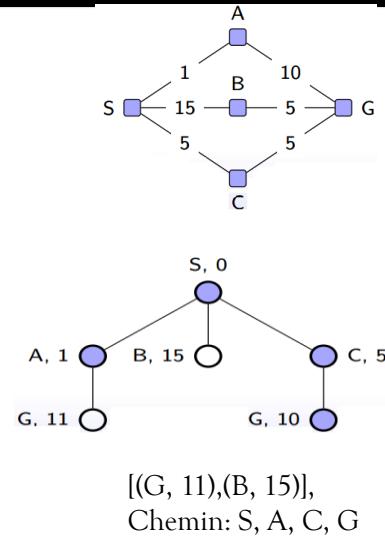
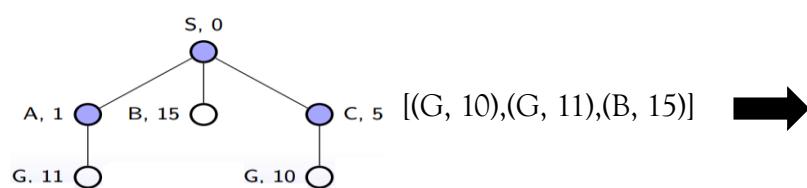
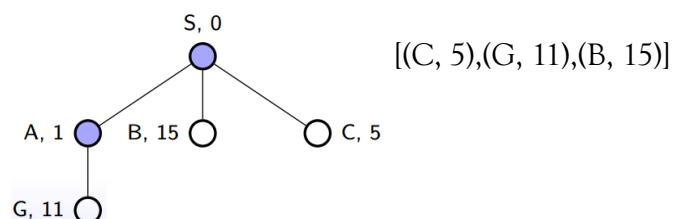
Exemple 2



69

Stratégies d'exploration à coût uniforme

Exemple 2



Contrairement à **BFS**, qui explore les nœuds en fonction de leur profondeur, **UCS** explore en fonction du coût total accumulé depuis l'état initial. Il utilise une **file de priorité**

Performance

Complétude : oui, (a) si le coût de chaque étape \geq à une constante positive ε .

(b) si b est fini

(a)

- ✓ Il existe un nombre fini d'actions avant que le coût du chemin soit égal au coût du but, donc il sera forcément atteint.
- ✓ Mais le coût des actions doit être supérieur à 0, sinon boucle infinie.



(b)

- ✓ Dans le cas de l'exploration de graphes, les espaces d'états infinis sont problématiques : imaginez que le coût des nœuds de la branche infinie partant de B ne dépassera jamais 50.



71

Performance

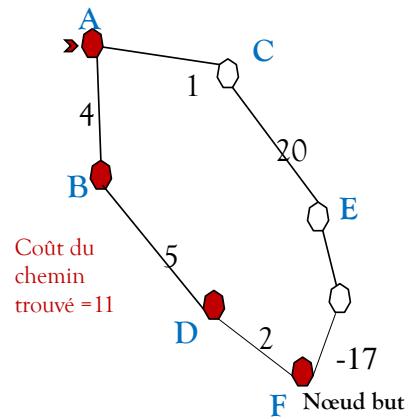
2. Optimalité : oui, les nœuds sont développés dans l'ordre de coût de chemin optimal (mais les coûts ne doivent pas être négatifs).



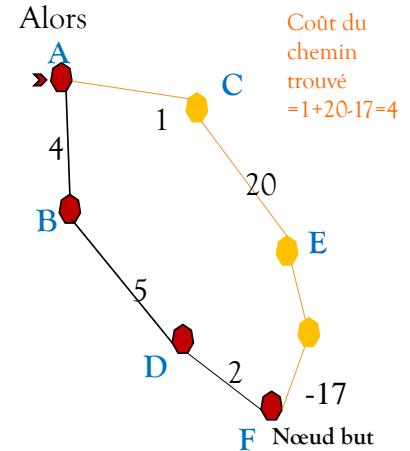
- L'optimalité est garantie car l'algorithme développe les nœuds par ordre de coût de chemin optimal :
 - Tout nœud choisi pour le développement est atteint par un chemin optimal, car sinon un autre nœud sur la frontière aurait été choisi.
 - Le coût des actions n'est jamais < 0 , donc les chemins ne deviennent jamais plus court lorsque des nœuds sont ajoutés

72

Et si les coûts étaient négatifs



File (frontière)	Nœud développé
[(A,0)]	(A,0)
[(C,1), (B,4)]	(C,1)
[(B,4),(E,21)]	(B,4)
[(D,9),(E,21)]	(D,9)
[(F,11),(E,21)]	(F,11) test du but réussi



73

Performance

3. Complexité en temps :

- ✓ Soit C^* le coût de la solution optimale
 - ✓ Soit ϵ le coût minimal de chaque action
1. Dans le pire des cas: $O(b^{1+C^*/\epsilon})$
 2. Égal à $O(b^{d+1})$ si les coûts des actions sont égaux

1. 1^{er} cas: Pourquoi $O(b^{1+C^*/\epsilon})$?

Dans le pire des cas, la complexité en temps dépend de :

- le coût total C^* : Plus le coût de la solution optimale est élevé, plus l'algorithme doit explorer de nœuds.
- Le coût minimal ϵ : Plus ϵ est petit, plus l'algorithme doit explorer de nœuds pour atteindre le coût C^* .

74

Performance

- L'algorithme explore les nœuds en ordre croissant de coût total.
 - Dans le pire des cas, il doit explorer tous les nœuds dont le coût total est inférieur ou égal à C^* .
- ==> Le nombre de nœuds explorés dépend du rapport C^*/ϵ , qui représente le nombre maximal d'actions nécessaires pour atteindre un coût total de C^* si chaque action coûte au moins ϵ .
- ==> Comme chaque nœud a b successeurs, le nombre total de nœuds explorés est de l'ordre de $b^{1+C^*/\epsilon}$.

75

Performance

2. Coûts des actions égaux

- Lorsque les coûts des actions sont égaux, on peut simplifier l'expression de la complexité.
- On a toutes les actions ont le même coût ϵ et la profondeur de la solution optimale est d (c'est-à-dire que le chemin optimal contient d actions).

==> Si toutes les actions ont le même coût ϵ , alors le coût total de la solution optimale est : $C^*=d \cdot \epsilon$

==> En substituant dans l'expression de la complexité :
 $O(b^{1+C^*/\epsilon})=O(b^{1+d \cdot \epsilon / \epsilon})=O(b^{1+d})=O(b^{d+1})$

76

Stratégies de Recherche non-informées

- N'exploitent **aucune information** sur la structure de l'arbre ou la présence potentielle de nœuds-solution pour **optimiser la recherche** :
 - Exploration en largeur d'abord
 - Exploration à coût uniforme
 - Exploration en profondeur d'abord**
 - Exploration en profondeur avec libération de mémoire
 - Exploration en profondeur limitée
 - Exploration itérative en profondeur
 - Exploration bidirectionnelle
- La plupart des problèmes réels sont susceptibles de provoquer une **explosion combinatoire** du nombre d'états possibles.

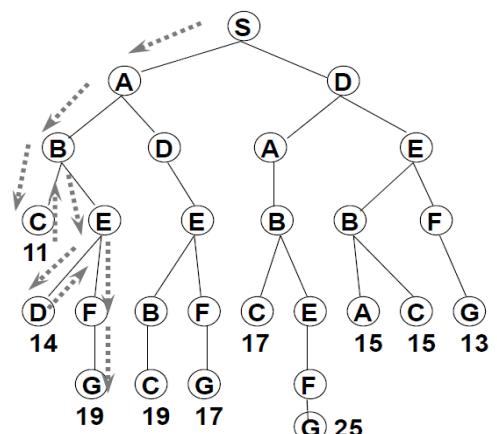
77

Stratégies d'exploration en profondeur d'abord (Depth First Search (DFS))

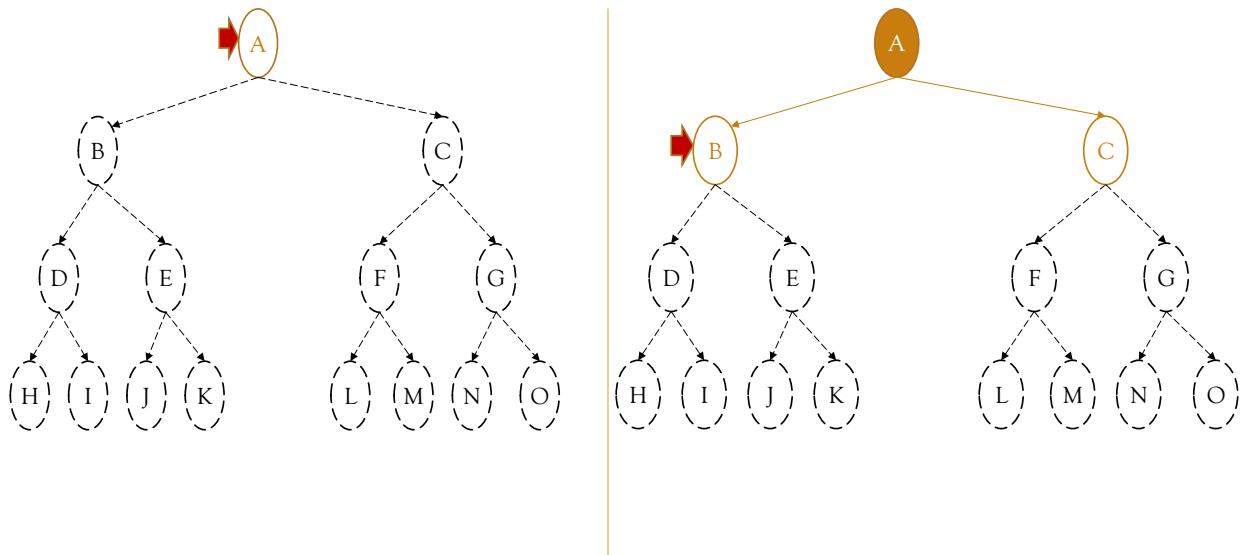
- Développer toujours le nœud le plus profond de la frontière courante.
- Variante : fonction récursive.

Principe

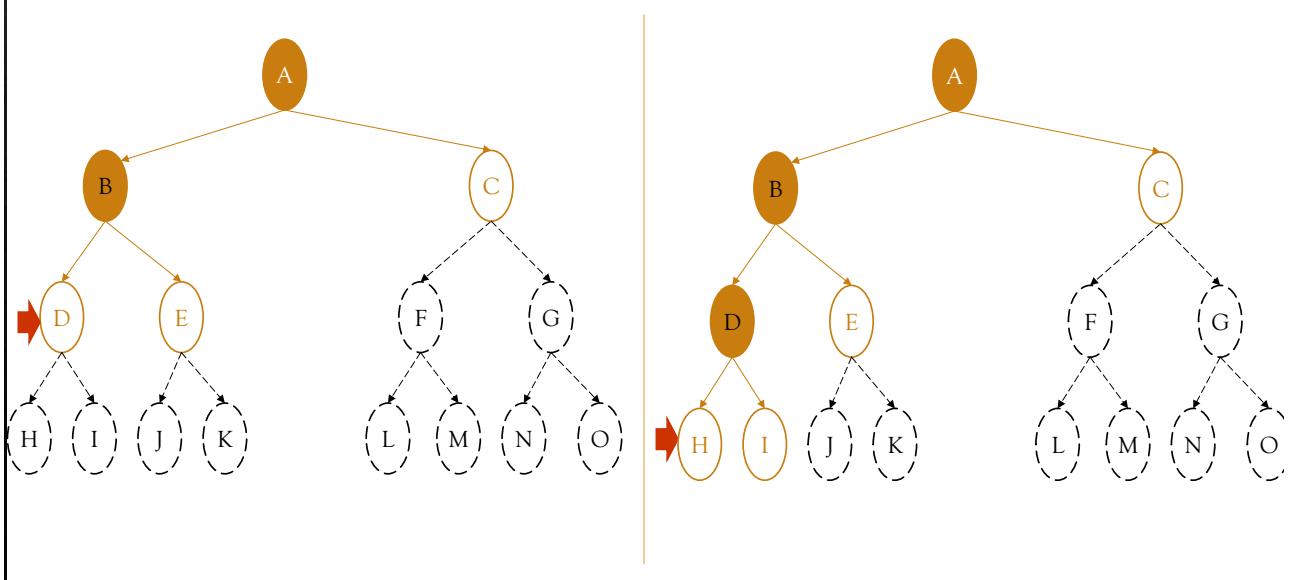
- Sélectionner une possibilité à chaque nœud
- Descendre jusqu'à ce qu'on atteigne le but ou une feuille
- En cas d'impasse, reprendre la recherche de l'ancêtre le plus proche dont au moins un fils n'a pas encore été exploré



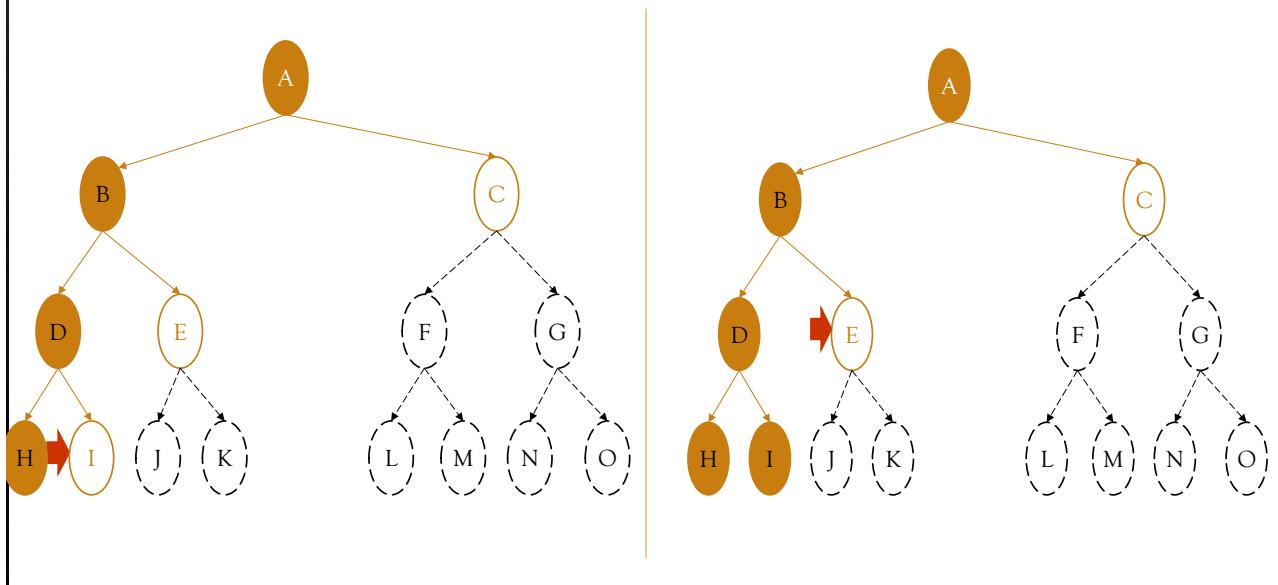
Stratégies d'exploration en profondeur d'abord (Depth First Search (DFS))



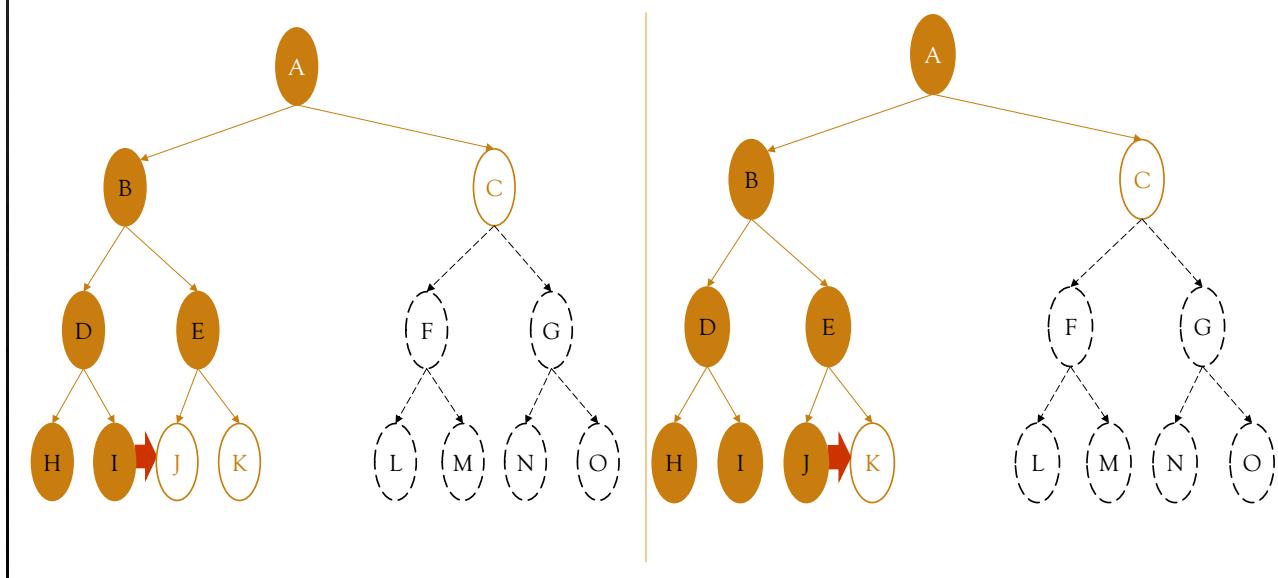
Stratégies d'exploration en profondeur d'abord (Depth First Search (DFS))



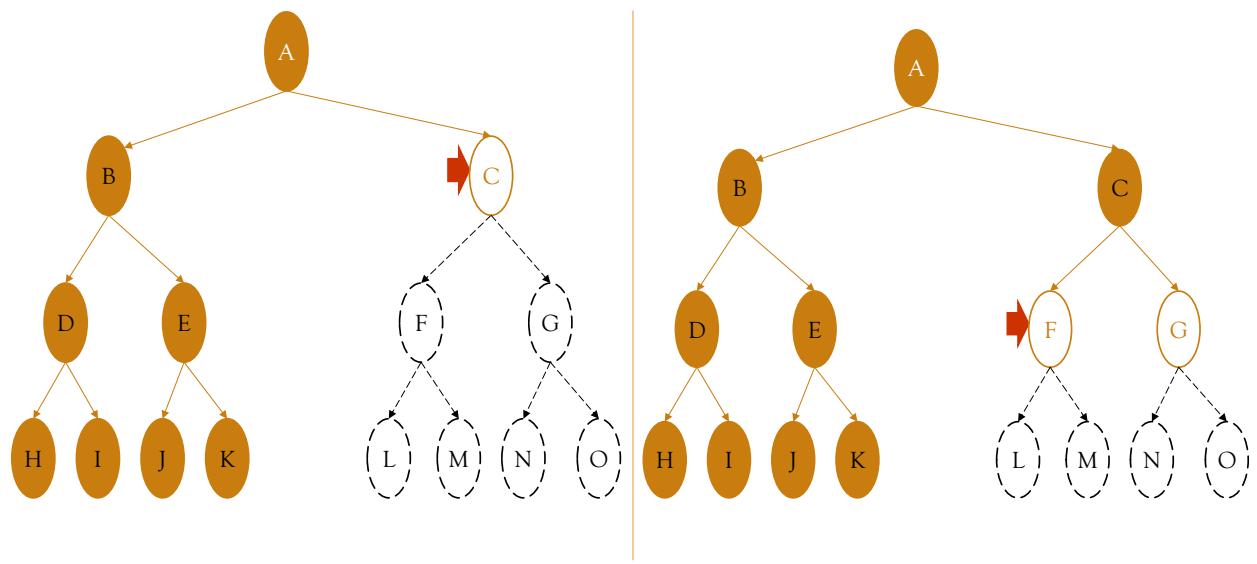
Stratégies d'exploration en profondeur d'abord (Depth First Search (DFS))



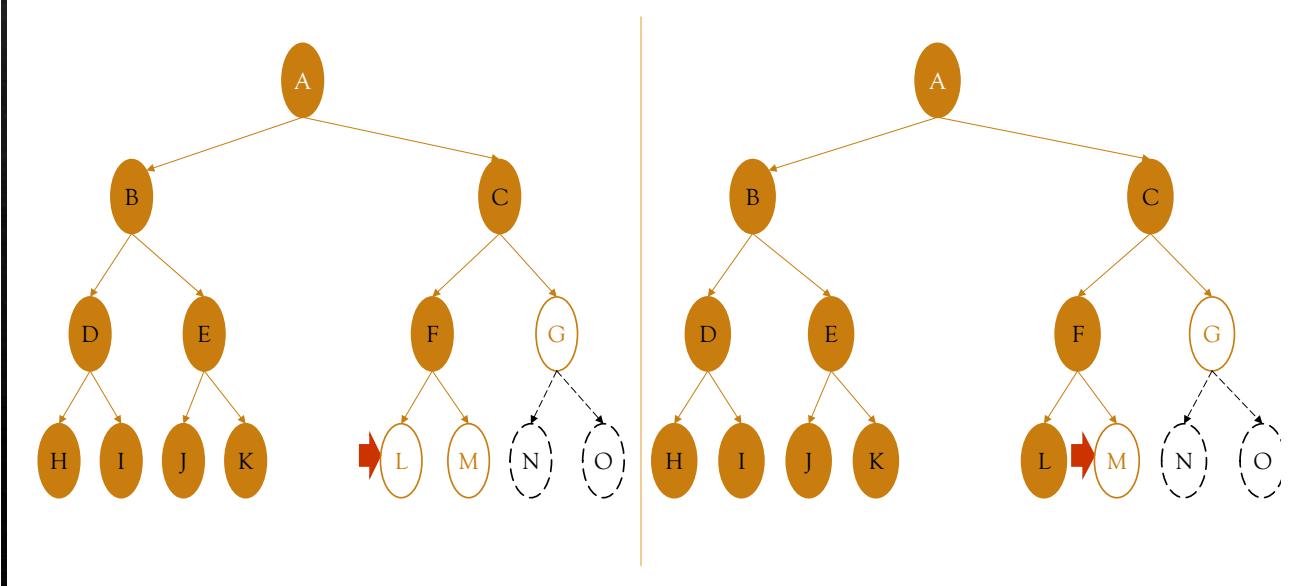
Stratégies d'exploration en profondeur d'abord (Depth First Search (DFS))



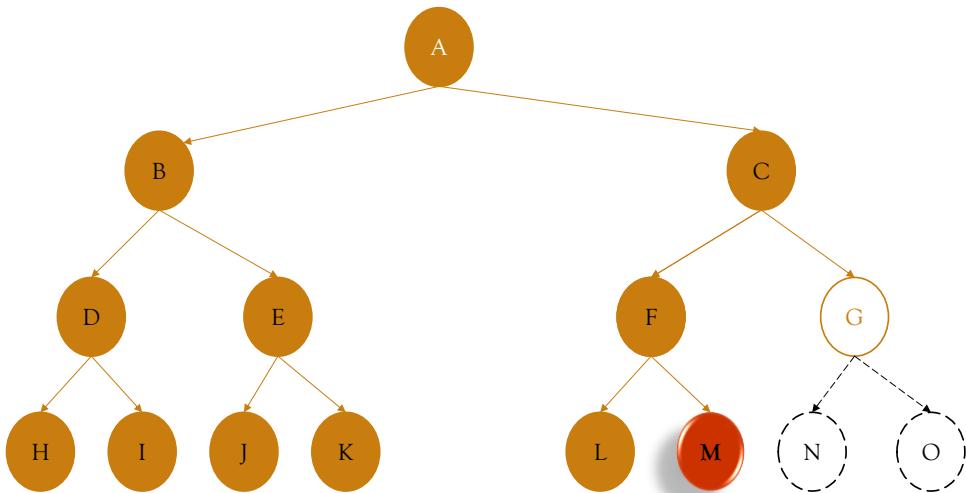
Stratégies d'exploration en profondeur d'abord (Depth First Search (DFS))



Stratégies d'exploration en profondeur d'abord (Depth First Search (DFS))

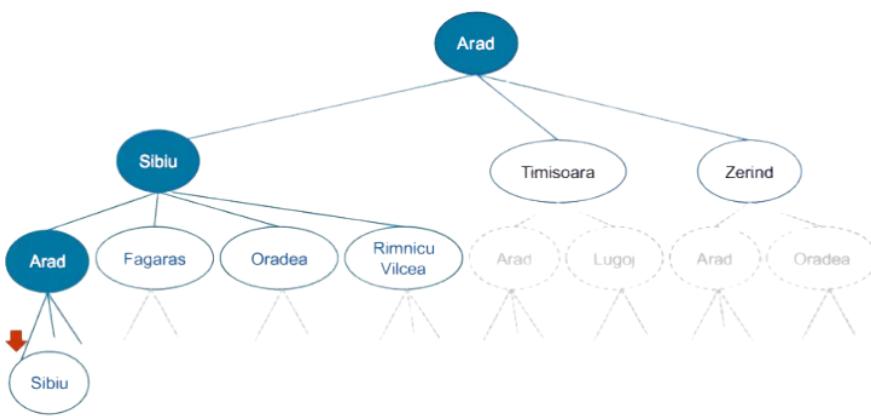


Stratégies d'exploration en profondeur d'abord (Depth First Search (DFS))



DFS: Performance

1. Complétude : oui, si l'espace d'états est fini et pas de chemins redondants (recherche en arbre) ou en maintenant une liste des nœuds visités.



DFS: Performance

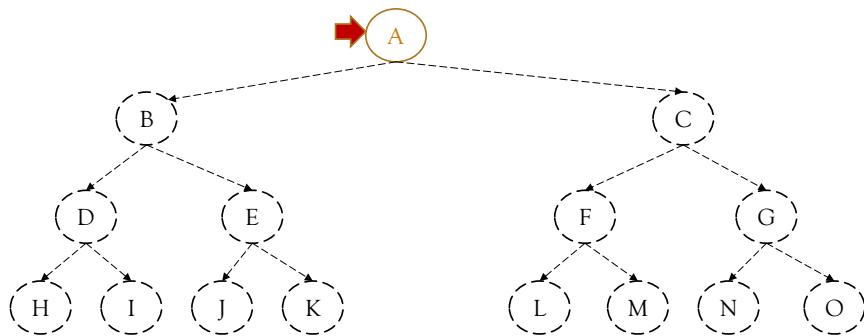
1. **Complétude** : oui, si l'espace d'états est fini et pas de chemins redondants (recherche en arbre) ou en maintenant une liste des nœuds visités.
2. **Optimalité** : non (imaginez que C ou J soit un nœud but).
3. **Complexité temporelle** : $O(b^m)$ m peut être $>d$
 ==> Dans le pire des cas, DFS explore tous les nœuds jusqu'à la profondeur m. Comme chaque nœud a en moyenne b successeurs, le nombre total de nœuds explorés est de l'ordre de b^m .
4. **Complexité en espace** : $O(b^m)$, linéaire pour l'exploration en arbre
 ==> DFS ne garde en mémoire que le chemin actuel en cours d'exploration, ainsi que les nœuds non encore explorés à chaque niveau. Ainsi, la mémoire utilisée est proportionnelle à b^m

Stratégies de Recherche non-informées

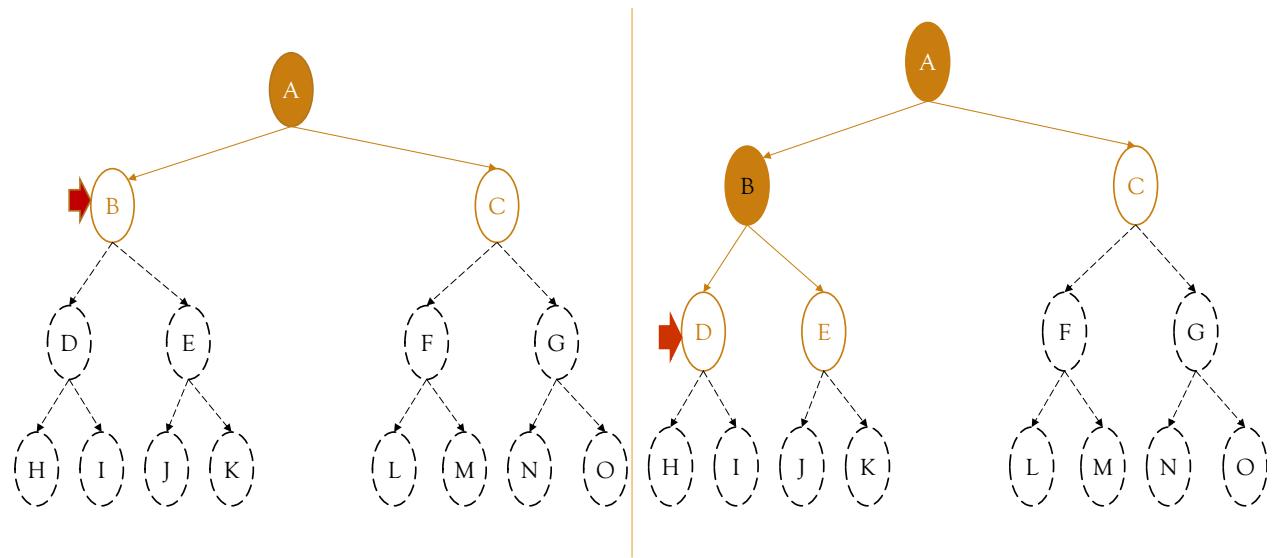
- N'exploitent **aucune information** sur la structure de l'arbre ou la présence potentielle de nœuds-solution pour **optimiser la recherche** :
 1. Exploration en largeur d'abord
 2. Exploration à coût uniforme
 3. Exploration en profondeur d'abord
 4. Exploration en profondeur avec libération de mémoire
 5. Exploration en profondeur limitée
 6. Exploration itérative en profondeur
 7. Exploration bidirectionnelle
- La plupart des problèmes réels sont susceptibles de provoquer une **explosion combinatoire** du nombre d'états possibles.

Stratégies d'exploration en profondeur d'abord avec libération mémoire

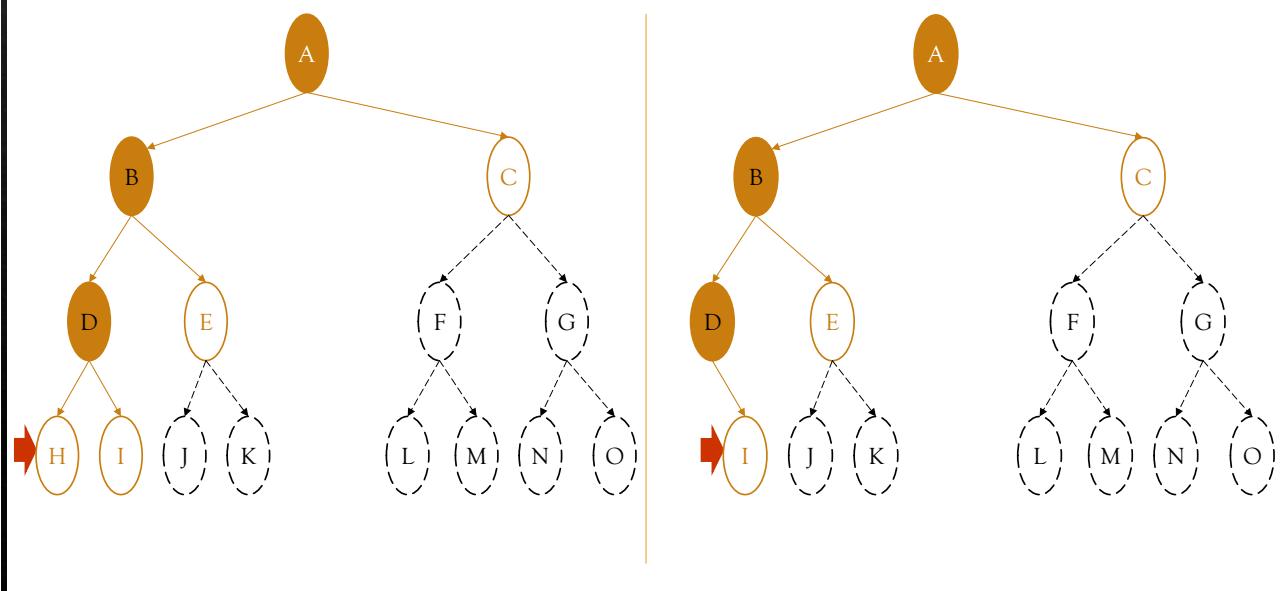
Variante de l'algorithme DFS conçues pour réduire l'utilisation de la mémoire tout en conservant les avantages de l'exploration en profondeur. Ces stratégies sont particulièrement utiles dans des environnements où la mémoire est limitée, comme dans les systèmes embarqués ou les applications à grande échelle.



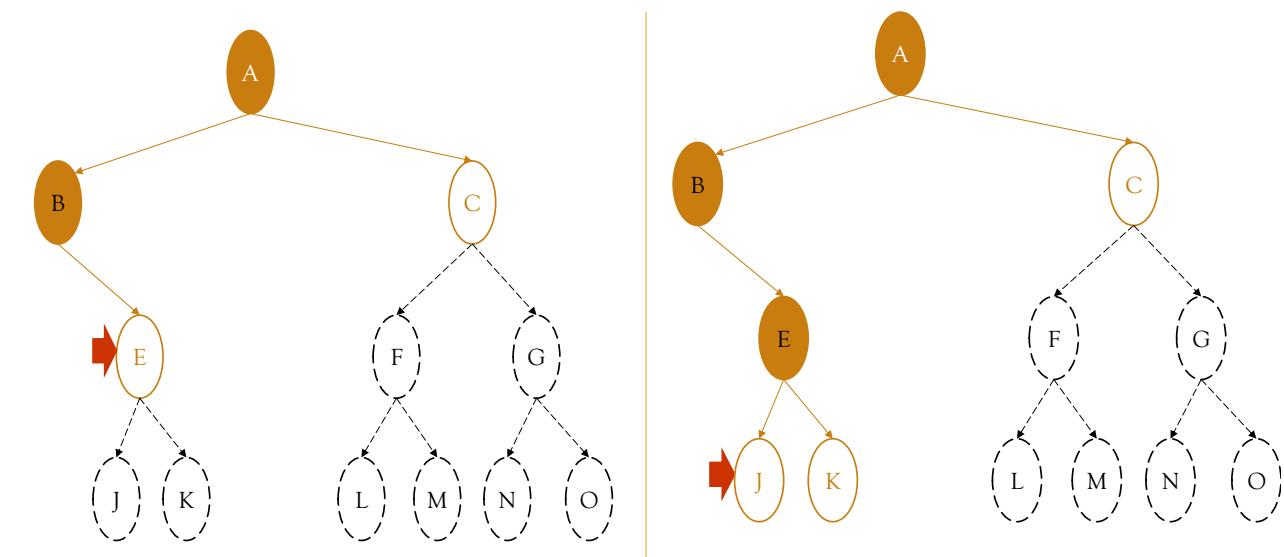
Stratégies d'exploration en profondeur d'abord avec libération mémoire



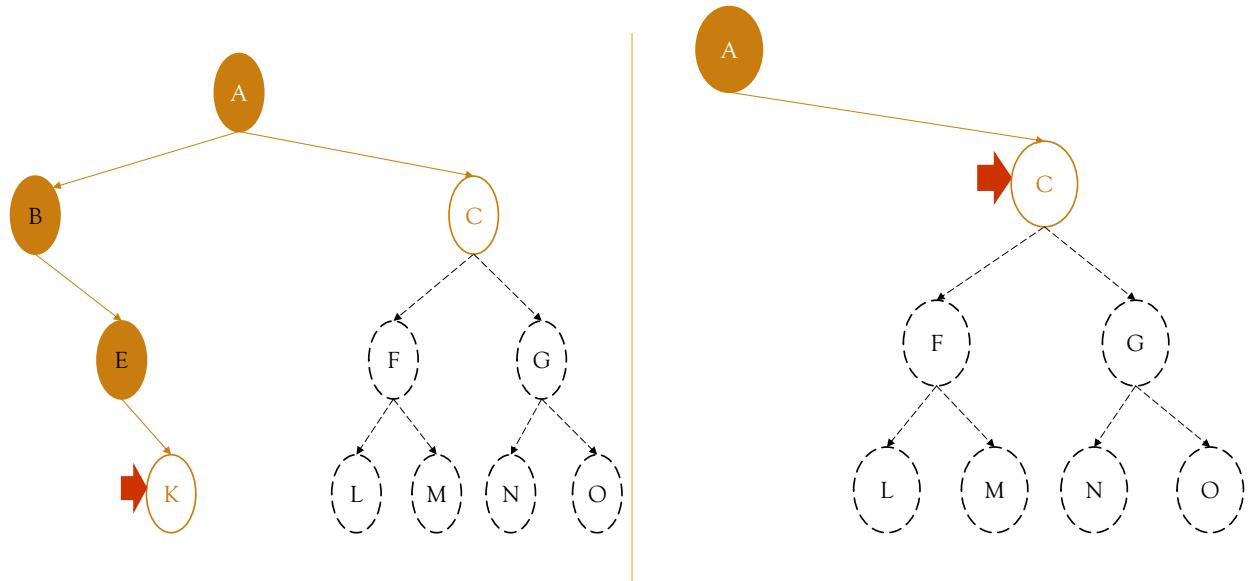
Stratégies d'exploration en profondeur d'abord avec libération mémoire



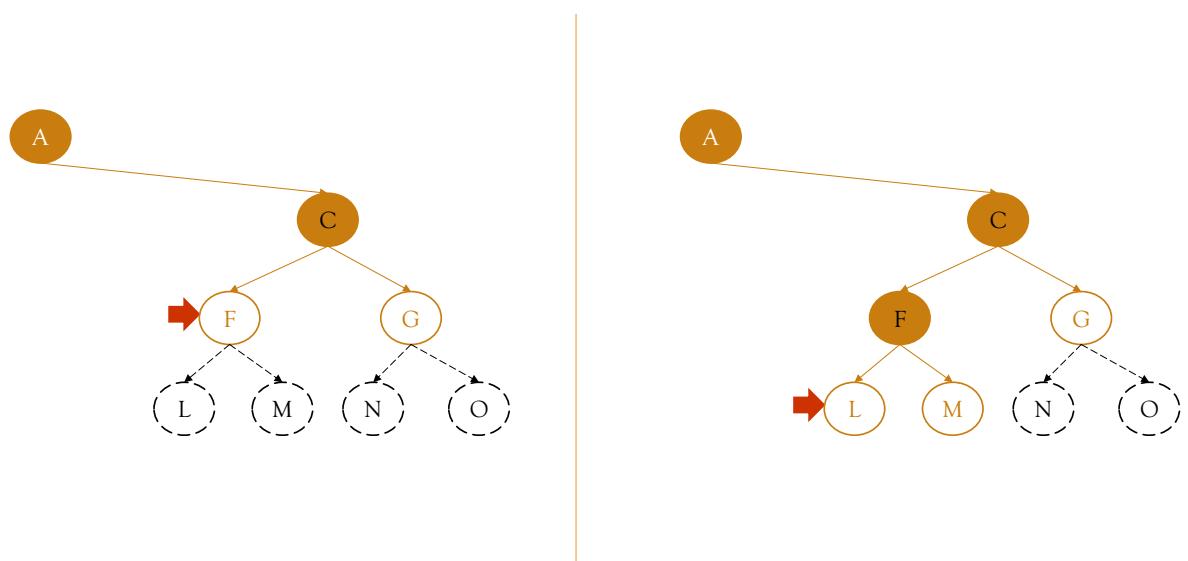
Stratégies d'exploration en profondeur d'abord avec libération mémoire



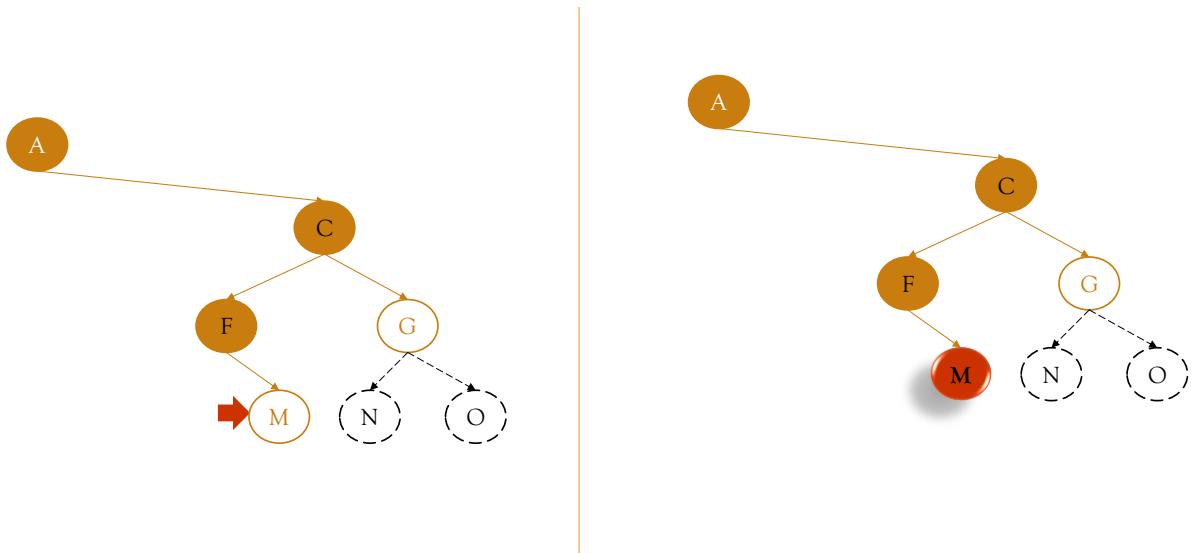
Stratégies d'exploration en profondeur d'abord avec libération mémoire



Stratégies d'exploration en profondeur d'abord avec libération mémoire



Stratégies d'exploration en profondeur d'abord avec libération mémoire



Stratégies d'exploration en profondeur d'abord avec libération mémoire

- 1. Complétude :** oui, si l'espace d'états est fini et pas de chemins redondants (recherche en arbre) ou en maintenant une liste des noeuds visités.
- 2. Optimalité :** non (imaginez que C ou J soit un noeud but).
- 3. Complexité temporelle :** $O(b^m)$ m peut être $>d$
==> Dans le pire des cas, DFS explore tous les noeuds jusqu'à la profondeur m. Comme chaque noeud a en moyenne b successeurs, le nombre total de noeuds explorés est de l'ordre de b^m .

Stratégies d'exploration en profondeur d'abord avec libération mémoire

La complexité en espace dépend fortement de la structure de l'espace d'états (arbre ou graphe) et de la manière dont l'algorithme est implémenté.

4. Complexité en espace :

- ✓ **$O(bm)$, linéaire** pour l'exploration en arbre: DFS explore un seul chemin à la fois, et la mémoire utilisée est proportionnelle à la profondeur m multipliée par le facteur de branchement b (Cela rend la complexité en espace linéaire en fonction de la profondeur m)
- ✓ **$O(b^m)$** pour l'exploration en graphe: DFS doit garder en mémoire le chemin actuel (de taille m) ainsi que les noeuds non encore explorés à chaque niveau (jusqu'à b noeuds par niveau). De plus, il doit marquer les noeuds visités pour éviter les cycles.
- ✓ **$O(m)$** en arbre avec backtracking (1 successeur généré à la fois): DFS ne garde en mémoire que le chemin actuel en cours d'exploration (de taille m). Puisqu'un seul successeur est généré et exploré à la fois, il n'est pas nécessaire de stocker tous les noeuds non encore explorés à chaque niveau.

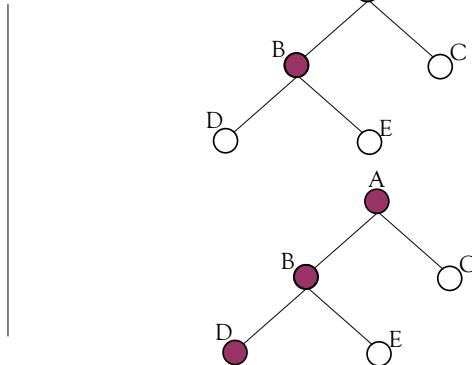
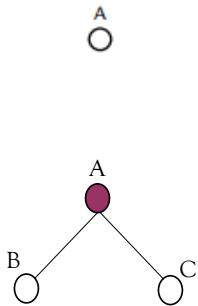
Stratégies de Recherche non-informées

- N'exploitent **aucune information** sur la structure de l'arbre ou la présence potentielle de noeuds-solution pour **optimiser la recherche** :
 1. Exploration en largeur d'abord
 2. Exploration à coût uniforme
 3. Exploration en profondeur d'abord
 4. Exploration en profondeur avec libération de mémoire
 5. **Exploration en profondeur limitée**
 6. Exploration itérative en profondeur
 7. Exploration bidirectionnelle
- La plupart des problèmes réels sont susceptibles de provoquer une **explosion combinatoire** du nombre d'états possibles.

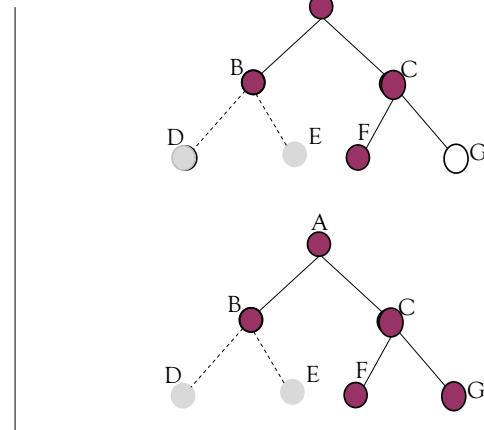
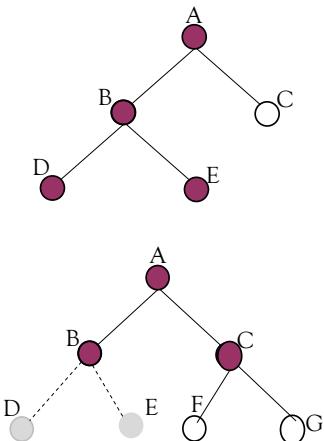
Stratégies de Recherche en profondeur limité

Principe

- ❖ Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
- ❖ Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$



Stratégies de Recherche en profondeur limité



Les stratégies de recherche en profondeur limitée sont des variantes de l'algorithme de recherche en profondeur d'abord qui imposent une limite de profondeur l à l'exploration. Cela permet d'éviter les inconvénients de DFS, comme l'exploration infinie dans des graphes sans limite de profondeur, tout en conservant les avantages de l'exploration en profondeur.

Performance

1. Complétude : uniquement lorsque $l \geq d$:

=> Si la limite de profondeur l est supérieure ou égale à la profondeur de la solution d , DLS est complet. Il explorera jusqu'à la profondeur l et trouvera la solution si elle existe

2. Complexité temporelle : $O(b^l)$

=> Dans le pire des cas, DLS explore tous les nœuds jusqu'à la profondeur l . Comme chaque nœud a en moyenne b successeurs, le nombre total de noeuds explorés est de l'ordre de bl .

3. Complexité spatiale : $O(b^l)$.

=> La complexité spatiale est linéaire en fonction de la limite de profondeur l .

On peut améliorer l'efficacité avec une bonne connaissance du problème.

Stratégies de Recherche non-informées

- N'exploitent **aucune information** sur la structure de l'arbre ou la présence potentielle de nœuds-solution pour **optimiser la recherche** :
 1. Exploration en largeur d'abord
 2. Exploration à coût uniforme
 3. Exploration en profondeur d'abord
 4. Exploration en profondeur avec libération de mémoire
 5. Exploration en profondeur limitée
 - 6. Exploration itérative en profondeur**
 7. Exploration bidirectionnelle
- La plupart des problèmes réels sont susceptibles de provoquer une **explosion combinatoire** du nombre d'états possibles.

Stratégies de Recherche en profondeur itérative

❖ Principe

- Profondeur limitée, mais en essayant toutes les profondeurs: 0, 1, 2, 3, ...
- Evite le problème de trouver une limite pour la recherche profondeur limitée

❖ Algorithme

Fonction RechercherProfondeurIterative (problème, profondeur) retourne solution

Pour profondeur \leftarrow 0 à ∞

solution \leftarrow RechercherProfondeurIterative (problème, profondeur)

Si solution = but alors

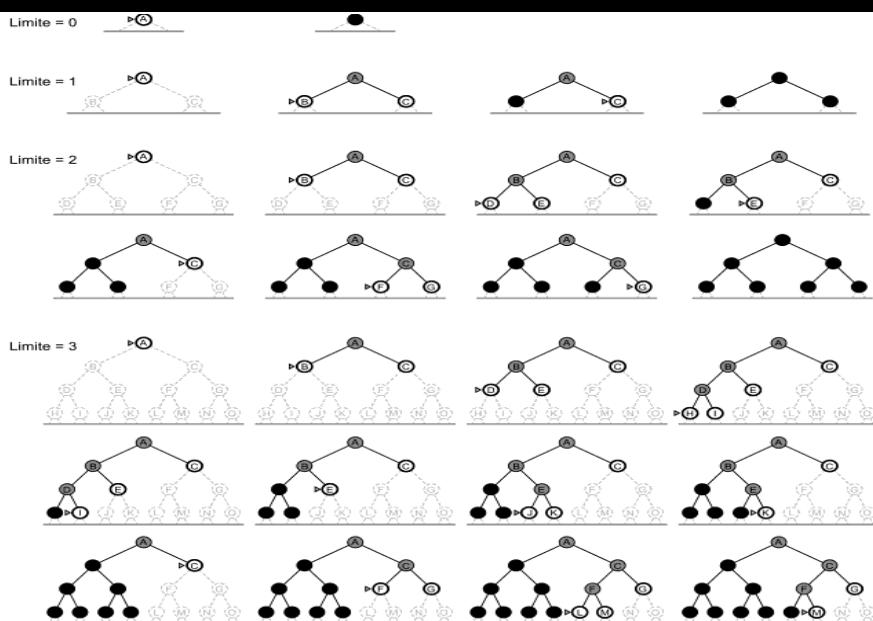
retourne solution

fin

103

fin

Stratégies de Recherche en profondeur itérative



104

Stratégies de Recherche en profondeur itérative

- ❖ **Complétude** : oui si b est fini.
- ❖ **Optimalité** : oui si coût est une fonction non décroissante de la profondeur du nœud.
- ❖ **Complexité en temps** :

$$== >(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d).$$
- ❖ **Complexité en espace** : $O(bd)$.
- ❖ Peut être modifiée pour une stratégie de coût uniforme.

105

Stratégies de Recherche non-informées

- N'exploitent **aucune information** sur la structure de l'arbre ou la présence potentielle de nœuds-solution pour **optimiser la recherche** :
 1. Exploration en largeur d'abord
 2. Exploration à coût uniforme
 3. Exploration en profondeur d'abord
 4. Exploration en profondeur avec libération de mémoire
 5. Exploration en profondeur limitée
 6. Exploration itérative en profondeur
 7. Exploration bidirectionnelle
- La plupart des problèmes réels sont susceptibles de provoquer une **explosion combinatoire** du nombre d'états possibles.

106

Stratégies de Recherche bi-directionnelle

- Exécution de deux explorations simultanément :
 1. en aval depuis l'état initial
 2. en amont à partir du but
- Arrêt lorsque les deux frontières se rencontrent au milieu.

Principe de base :

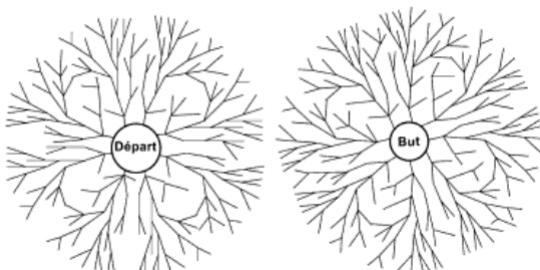
1. Deux fronts de recherche : L'algorithme commence à explorer à partir de l'état initial et de l'état cible (ou objectif) en même temps.
2. Rencontre au milieu : Les deux explorations progressent jusqu'à ce qu'elles se rencontrent à un point commun. Lorsqu'elles se rencontrent, le chemin entre l'état initial et l'état objectif peut être reconstitué.

La recherche bi-directionnelle est une technique puissante pour des problèmes où l'état but est bien défini et où les mouvements peuvent être inversés.

107

--> Difficile à appliquer si le but est une description abstraite (n reines).

Stratégies de Recherche bi-directionnelle



Exemple:

Imaginons que vous devez trouver un chemin entre les villes A (état initial) et D (état final) dans un graphe de villes connectées.

1. Vous commencez à explorer à partir de A, en visitant progressivement les villes voisines.
2. Simultanément, vous commencez à explorer à partir de D, en visitant les villes voisines de D.
3. Si à un moment donné, vous atteignez une ville que vous avez déjà visitée depuis l'autre côté, cela signifie que vous avez trouvé un chemin entre les deux villes et vous pouvez reconstituer le chemin complet.

Stratégies de Recherche bi-directionnelle

Méthode avec une table de hachage :

Pour faciliter cette détection, **une table de hachage** peut être utilisée.

1. **Création de la table de hachage** : On crée une table de hachage (ou dictionnaire) pour stocker les nœuds explorés dans l'un des arbres
2. **Exploration dans la première direction** : Au fur et à mesure que l'algorithme explore les noeuds depuis l'état initial (dans le premier arbre), il ajoute chaque noeud à la table de hachage.
3. **Exploration dans la deuxième direction** : Ensuite, lorsque l'algorithme explore les nœuds depuis l'état final (dans le second arbre), il vérifie si chaque nœud visité dans cet arbre existe déjà dans la table de hachage.
 - a. Si existe déjà dans la table de hachage , cela signifie que les deux arbres se sont rencontrés à ce nœud commun, et le chemin complet peut être reconstitué.
 - b. Sinon, l'algorithme continue à explorer dans les deux directions.

109

Stratégies de Recherche bidirectionnelle

Complétude : oui.

Optimalité : oui si les coûts des étapes sont identiques et les deux directions utilisent une exploration en largeur d'abord.

Complexité en temps : $O(b^d/2)$.

Complexité en espace : $O(b^d/2)$.

110

Comparaison des stratégies d'exploration d'arbres

Critère	Largeur d'abord	Coût uniforme	Profondeur d'abord	Profondeur limitée	Profondeur itérative	Bidirectionnelle (si applicable)
Complète?	Oui ^a	Oui ^{a,b}	Non	Non	Oui ^a	Oui ^{a,d}
Temps	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Espace	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Oui ^c	Oui	Non	Non	Oui ^c	Oui ^{c,d}

- b : facteur de branchement
- d : profondeur de la solution la moins profonde
- m : profondeur maximale de l'arbre d'exploration
- ℓ : profondeur limite
- ^a si b est fini
- ^b si les coûts des étapes sont $\geq \epsilon$ avec ϵ positif
- ^c si les coûts d'étapes sont tous identiques
- ^d si les deux directions utilisent une exploration en largeur d'abord

III

Type de stratégies de recherche

1. Exploration Non-informée (Aveugle)

L'algorithme n'a aucune information supplémentaire autre que celles contenues dans la définition du problème. Il ne sait pas si un état est plus proche ou plus prometteur qu'un autre.

=> Pas d'autres informations sur les états que celles fournies dans la définition du problème.

=> Elles génèrent des successeurs et distinguent un état final d'un état non final.

2. Exploration informée (heuristique)

L'algorithme utilise une fonction heuristique pour évaluer la qualité des états non finaux et orienter la recherche vers les plus prometteurs.

=> Peuvent déterminer si un état non final est meilleur qu'un autre.

II

Exploration informée (heuristique): Pourquoi?

- Stratégies d'exploration non informées sont généralement très peu efficaces.
- Stratégies d'exploration informées :
 - Utilisent des connaissances du problème : **une fonction heuristique**.
 - Plus efficaces que l'exploration aveugle.

Un algorithme de recherche **efficace** doit **guider** la recherche de solution pour éviter l'explosion combinatoire

- ✓ En faisant des choix
- ✓ En gérant la révision de ces choix

On utilise des **Heuristiques** pour guider ces choix en ordonnant dynamiquement la liste des successeurs selon leur "promesse de rapprocher d'un but"

Exploration informée (heuristique)

- ✓ Connaissance spécifique au problème à résoudre, indépendante de l'algorithme de recherche, et non généralisable
- ✓ Une règle d'estimation, une stratégie, une astuce, une simplification ou autre règle permettant de guider les choix non-déterministes
- ✓ Permet de détecter grâce à une fonction d'évaluation le nœud qui semble potentiellement meilleur que les autres et de se concentrer sur ce nœud par la suite en ordonnant la liste de successeurs d'un état
- ✓ La notion de complexité conduit naturellement à la notion d'heuristique
- ✓ A la différence des algorithmes aveugles, les heuristiques sont tirées de l'expérience, de l'abstraction ou d'un apprentissage plutôt que d'une analyse scientifique

Exploration informée (heuristique)

- ✓ Qui donne une « bonne » estimation du coût réel
- ✓ Facile à calculer

Exemple :

Pour le problème du plus court chemin entre les villes

$h(n)$ - la distance à vol d'oiseau (distance en ligne droite) de la ville n à la ville destination

=> Que veut dire « bonne » estimation ?

=> Comment définir une heuristique ?

Stratégies de Recherche informées

- Tout algorithme de recherche heuristique dispose d'une fonction d'évaluation f qui détermine l'ordre dans lequel les nœuds sont traités : la liste de nœuds à traiter est organisée en fonction des f -valeurs des nœuds, avec les nœuds de plus petite valeur en tête de liste.

□ Il existe plusieurs stratégies :

1. Best-first Search

1.1. Exploration gloutonne par le meilleur

1.2. A*

2. Exploration heuristique à mémoire limitée:

IDA*,RBFS,SMA*

116

Stratégie d'exploration par le meilleur d'abord

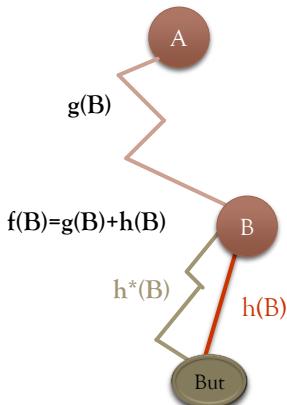
- Basée sur l'exploration en arbre et en graphe.
- Le nœud à développer est choisi selon une fonction d'évaluation $f(n)$:
 - ✓ La fonction $f(n)$ est une estimation coût.
 - ✓ Le nœud qui a un $f(n)$ le plus faible est développé en premier.
 - ✓ Cette méthode est similaire à l'exploration à coût uniforme, mais au lieu d'utiliser le coût réel accumulé depuis le nœud initial (g), elle utilise une fonction d'évaluation f , qui estime la qualité du nœud en prenant en compte une combinaison de facteurs.

si $f(n)=g(n)$, c'est-à-dire on n'utilise pas l'heuristique, on obtient alors la recherche à coût uniforme (non informée, déjà étudiée)

117

Fonction d'évaluation $f(n)$

$f(n) = g(n) + h(n)$: la fonction d'évaluation utilisée pour ordonner les nœuds.



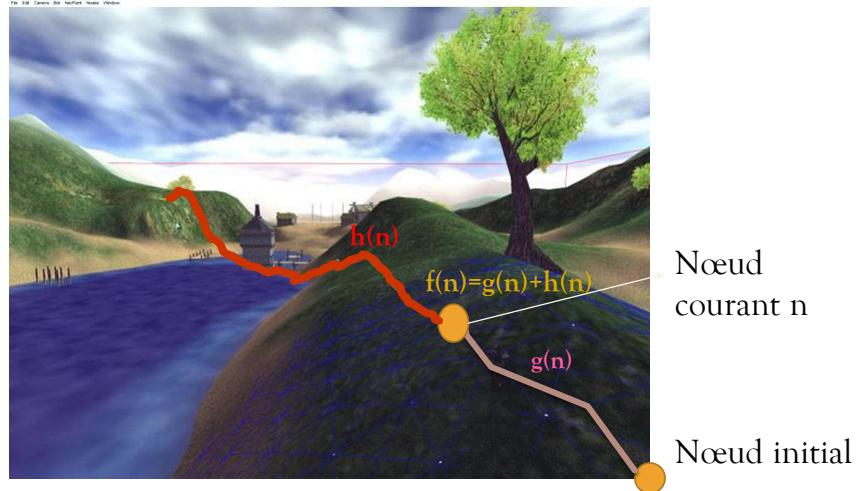
- ◊ $g(n)$: coût réel (somme des meilleurs coûts du nœud initial à n).
- ◊ $h^*(n)$: coût réel (somme des coûts du nœud n au nœud but).
- ◊ $h(n)$: coût heuristique (estimation du chemin le moins coûteux de l'état du nœud n à nœud but).

118

Fonction d'évaluation $f(n)$

$h(n)$: heuristique
distance à vol d'oiseau

Le prochain nœud est choisi selon la valeur de $f(n)$



Performance

Avantages :

- Rapide si une bonne heuristique est utilisée, car elle guide la recherche vers les nœuds les plus prometteurs.

Inconvénients :

- **Optimalité:** Non optimale, l'exploration par le meilleur d'abord peut ne pas toujours garantir de trouver la solution optimale, car l'heuristique peut induire en erreur.
- **Complétude:** Si la fonction heuristique n'est pas bien choisie, la recherche peut ne pas explorer tous les nœuds nécessaires pour trouver une solution.

Stratégies de Recherche informées

- Tout algorithme de recherche heuristique dispose d'une fonction d'évaluation f qui détermine l'ordre dans lequel les nœuds sont traités : la liste de nœuds à traiter est organisée en fonction des f -valeurs des nœuds, avec les nœuds de plus petite valeur en tête de liste.

□ Il existe plusieurs stratégies :

1. Best-first Search

1.1. Exploration gloutonne par le meilleur

1.2. A*

2. Exploration heuristique à mémoire limitée:

IDA*, RBFS, SMA*

121

Exploration gloutonne par le meilleur

L'exploration gloutonne par le meilleur (ou Greedy Best-First Search) est une stratégie de recherche qui choisit toujours le nœud **qui semble le plus prometteur**, basé sur une **heuristique** qui évalue la distance estimée **jusqu'à la solution finale**.

- Développe le nœud le plus proche du but.
- **Fonction d'évaluation :** se base uniquement sur l'**heuristique $h(n)$** , qui estime le coût restant pour atteindre l'objectif à partir du nœud n ,
- **Objectif :** Choisir le nœud qui semble le plus proche de l'objectif en termes de coût estimé restant, sans tenir compte du coût déjà accumulé pour atteindre le nœud, $f(n)=h(n)$.
- Exemple du voyageur en Roumanie :
 - ✓ Heuristique distance à vol d'oiseau (Straight-line distance SLD : H_{SLD}).

122

L'heuristique H_{SLD}

- L'heuristique H_{SLD} (ou **heuristique SLD**, pour **Straight-Line Distance**) :
 - ✓ Utiliser pour des problèmes de **recherche de chemin dans un espace**. Cette heuristique est fréquemment utilisée dans des problèmes où les déplacements dans un espace géographique ou physique sont concernés.
 - ✓ Estime **le coût restant** (ou la distance) pour atteindre l'objectif à partir d'un nœud **n** entre le nœud actuel et le nœud objectif. Cela donne une estimation directe de la distance la plus courte entre les deux points sans tenir compte des obstacles ou des contraintes du chemin.
- Prenons une carte où l'on veut aller de la ville A à la ville B
 - ✓ **Coût réel (g)** : Distance routière réelle entre A et B.
 - ✓ **Heuristique (H_{SLD})** : Distance à vol d'oiseau entre A et B.

123

L'heuristique H_{SLD}

Application:

- ✓ Recherche de chemin (Pathfinding) : GPS, planification de trajets.
- ✓ Jeux vidéo : Mouvements d'agents IA.
- ✓ Problèmes de navigation en robotique.

H_{SLD} est une heuristique simple et efficace qui fonctionne bien dans des problèmes spatiaux mais peut être moins pertinente dans des cas où le coût du chemin ne dépend pas uniquement de la distance (ex : obstacles, péages, terrains variés).

124

Exploration gloutonne par le meilleur: Performance

Complétude :

- ✓ Non , pour arbre (Il peut ignorer des chemins valides à cause de son choix basé uniquement sur l'heuristique)

Optimalité :

- ✓ Non, l'optimisation est locale. L'algorithme ignore les coûts réels observés

Complexité en temps :

- ✓ $O(b^m)$ en arbre, mais peut être améliorée par le choix d'une bonne fonction heuristique.

Complexité d'espace :

- ✓ $O(b^m)$, elle garde tous les nœuds en mémoire.

125

En résumé

Critère	Exploration gloutonne par le meilleur
Principe	Utilise uniquement une fonction heuristique $h(n)$ pour choisir le nœud le plus prometteur.
Fonction d'évaluation	$f(n)=h(n)$ (n'examine que l'heuristique)
Prise en compte du coût réel $g(n)$	Non, il ne prend en compte que la proximité estimée du but.
Optimalité	Non optimal, car elle ne prend pas en compte le coût accumulé pour atteindre le nœud actuel
Complétude	Non en général, il peut se coincer dans des cycles ou explorer infiniment un mauvais chemin.
Vitesse de convergence	Plus rapide, car elle se concentre sur les nœuds les plus prometteurs immédiatement.

126

Stratégies de Recherche informées

- Tout algorithme de recherche heuristique dispose d'une fonction d'évaluation f qui détermine l'ordre dans lequel les nœuds sont traités : la liste de nœuds à traiter est organisée en fonction des f -valeurs des nœuds, avec les nœuds de plus petite valeur en tête de liste.

□ Il existe plusieurs stratégies :

1. Best-first Search

 1.1. Exploration gloutonne par le meilleur

 1.2. A*

2. Exploration heuristique à mémoire limitée:

 IDA*, RBFS, SMA*

127

Principe de l'exploration A*

Constat

Best-first search/glotonne donne la préférence aux nœuds dont les états **semblent** les plus proche d'un état but,

(-) il ne prend pas en compte les coûts des chemins reliant l'état initial à ces nœuds.



A* combine le coût du chemin déjà parcouru et une estimation du coût restant pour atteindre l'objectif.

- Fonction d'évaluation $f(n) = g(n) + h(n)$
- Dans chaque nœud n on stock $g(n)$, et $f(n)$ sera calculé
- À chaque étape on choisit un nœud avec $f(n)$ minimal

Principe de l'exploration A*

Algorithm

1. mettre le sommet initial s dans FRONTIERE, $g(s)=0$, $f(s)=h(s)$
2. Si FRONTIERE vide, sortir avec échec
3. retirer de FRONTIERE le noeud n pour lequel $f(n)$ est minimal
4. si n est le but alors terminer et retracer la solution, le chemin de s à n
5. développer n pour engendrer tous ses successeurs. Pour tout successeur m de n :
 - A. Calculer $f(m)=g(m)+h(m)$, où $g(m)=g(n)+\text{coût}(n, m)$
 - B. Insérer m dans FRONTIERE
6. aller à 2

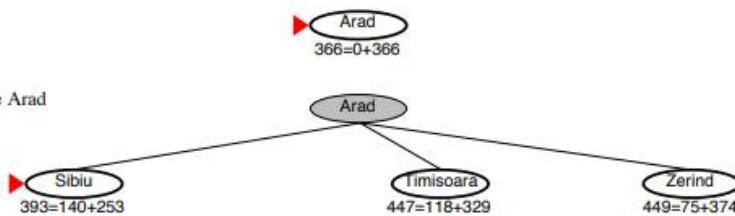
Etapes pour une exploration A* vers Bucarest

1

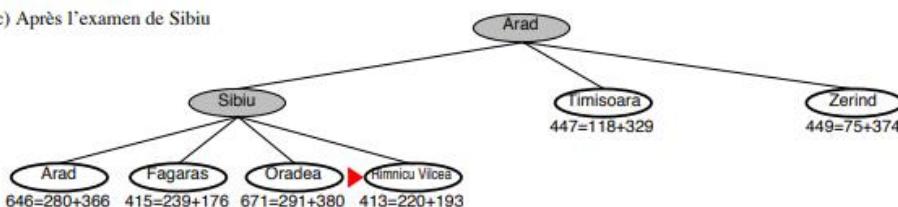
(a) L'état initial



(b) Après l'examen de Arad



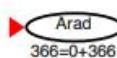
(c) Après l'examen de Sibiu



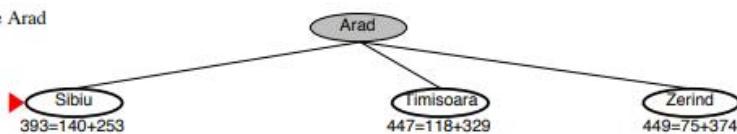
Etapes pour une exploration A* vers Bucarest

1

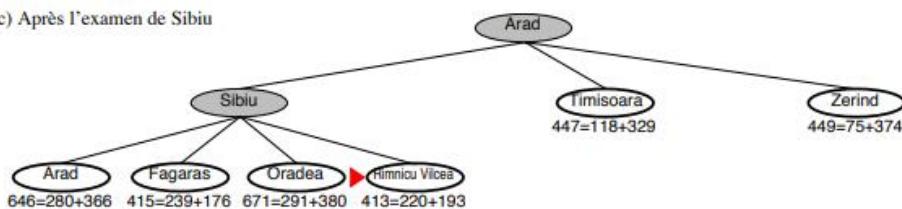
(a) L'état initial



(b) Après l'examen de Arad



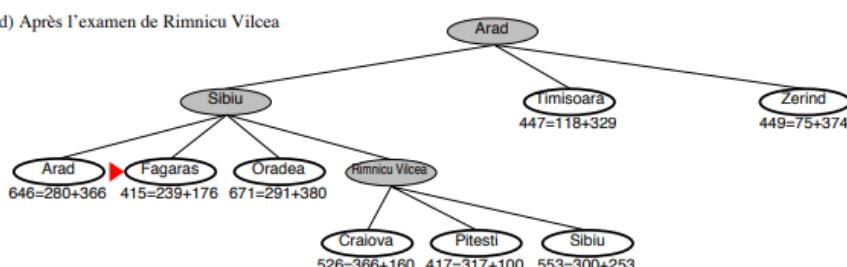
(c) Après l'examen de Sibiu



Etapes pour une exploration A* vers Bucarest

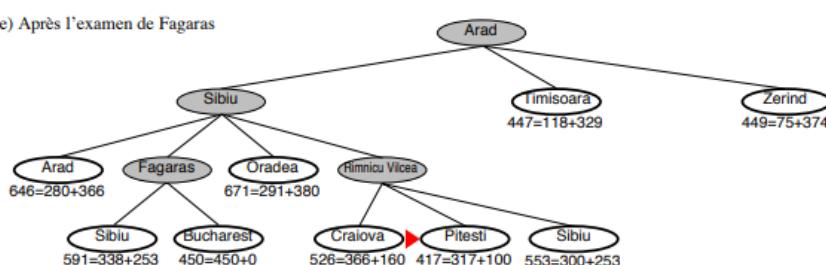
2

(d) Après l'examen de Rimnicu Vilcea



3

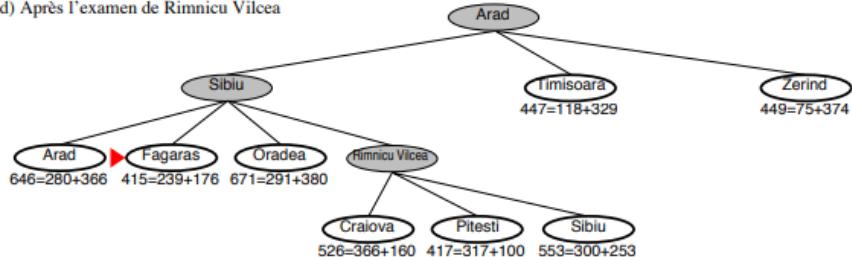
(e) Après l'examen de Fagaras



Etapes pour une exploration A* vers Bucarest

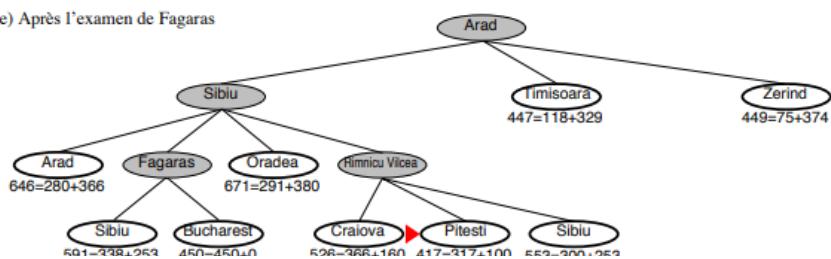
2

(d) Après l'examen de Rimnicu Vilcea



3

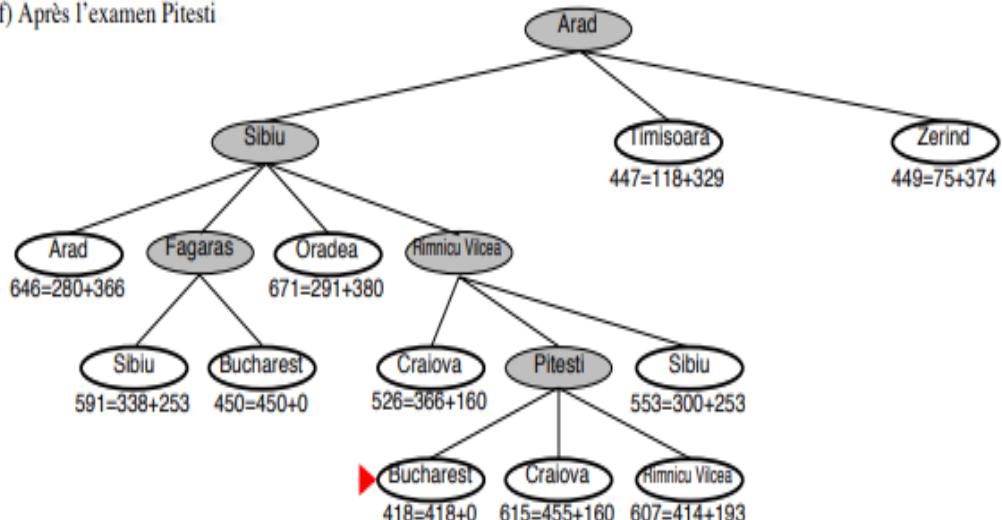
(e) Après l'examen de Fagaras



Etapes pour une exploration A* vers Bucarest

5

(f) Après l'examen Pitesti



A*: performance

- **Complétude** : Oui à moins qu'il y soit une infinité de nœuds avec $f \leq f(\text{but})$.
- **Optimalité** :
 - Arbre : **Oui si $h(n)$ est admissible.**
 - Graphe : **Oui, si $h(n)$ est consistante.**

Une heuristique admissible

Définition: la fonction heuristique **h** est admissible si pour chaque sommet n ,

$h(n) \leq$ coût réel du chemin le plus court de n vers un état but

=> Une heuristique h est **admissible** si *elle ne surestime jamais le coût réel* pour atteindre le but : **$h(n) \leq h^*(n)$**

Conditions d'optimalité de A*

- **Consistance (monotonie) : pour l'exploration de graphes :**

=> La consistance (ou monotonie) est une propriété importante pour les heuristiques utilisées dans les algorithmes de recherche de graphes, comme l'algorithme A*.

- Une heuristique $h(n)$ est dite **consistante** si elle satisfait la condition suivante:

$$h(n) \leq c(n, a, n') + h(n')$$

- ✓ **$h(n)$** : Estimation heuristique du coût restant pour atteindre l'objectif à partir du nœud n .
- ✓ **$c(n, a, n')$** : Coût réel de l'action a pour passer du nœud n au nœud n' .
- ✓ **$h(n')$** : Estimation heuristique du coût restant pour atteindre l'objectif à partir du nœud n' .

Attention

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

A*: performance

- A. Si la heuristique h est admissible alors la version tree-search de l'algorithme A* est optimale.
- B. Si la heuristique h est consistante alors la version graph-search de l'algorithme A* est optimale.

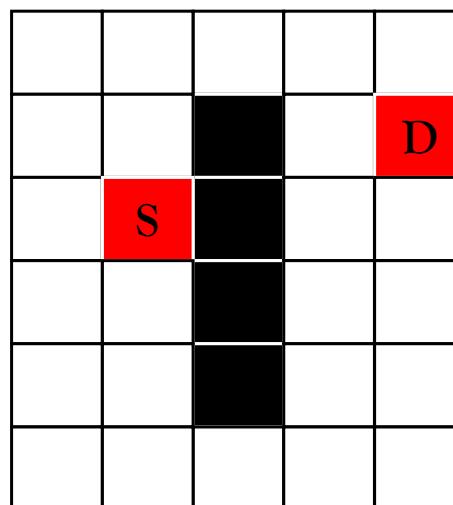
- **Complexité de temps** : Dans le pire cas, l'algorithme explore presque tous les nœuds jusqu'à atteindre l'objectif. La complexité est de l'ordre de $O(b^d)$
- **Complexité en espace** : Exponentielle $O(b^d)$, A* stocke **tous les nœuds en mémoire**, ce qui peut devenir problématique pour de grands espaces d'états

Exemple 1

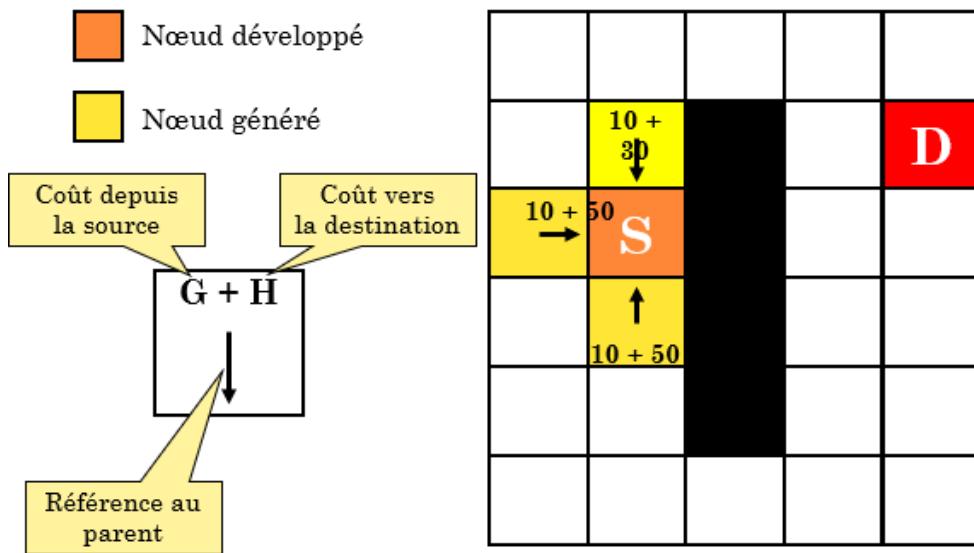
S Nœud source

D Nœud destination

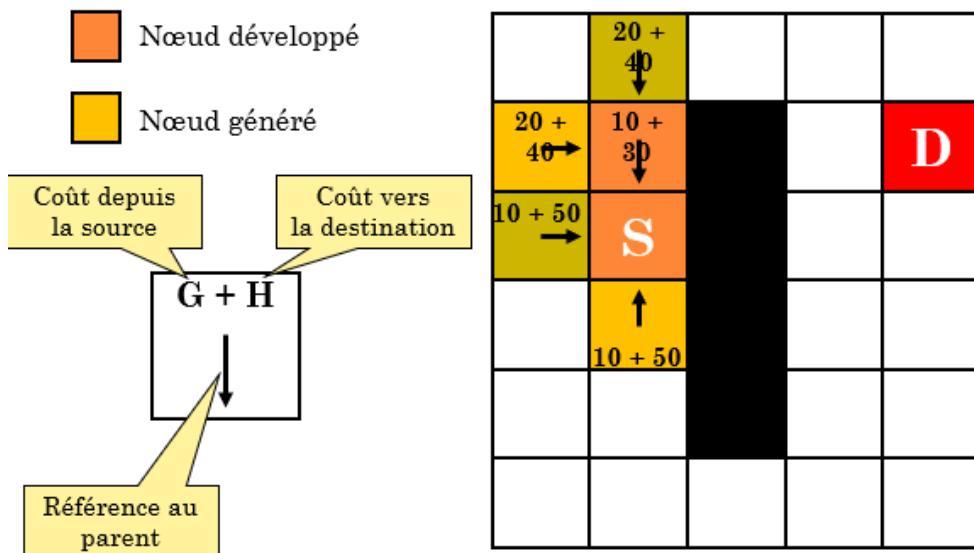
█ Obstacle



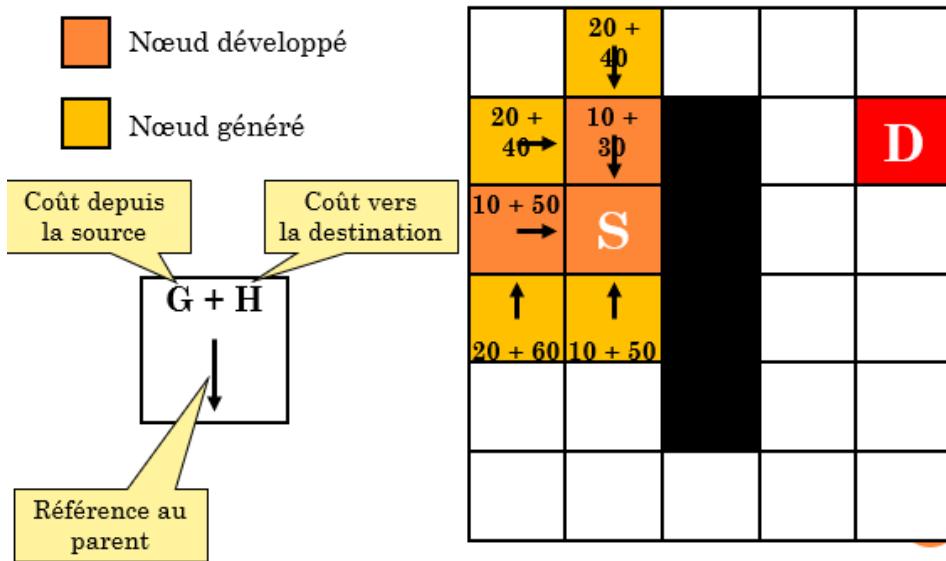
Exemple 1



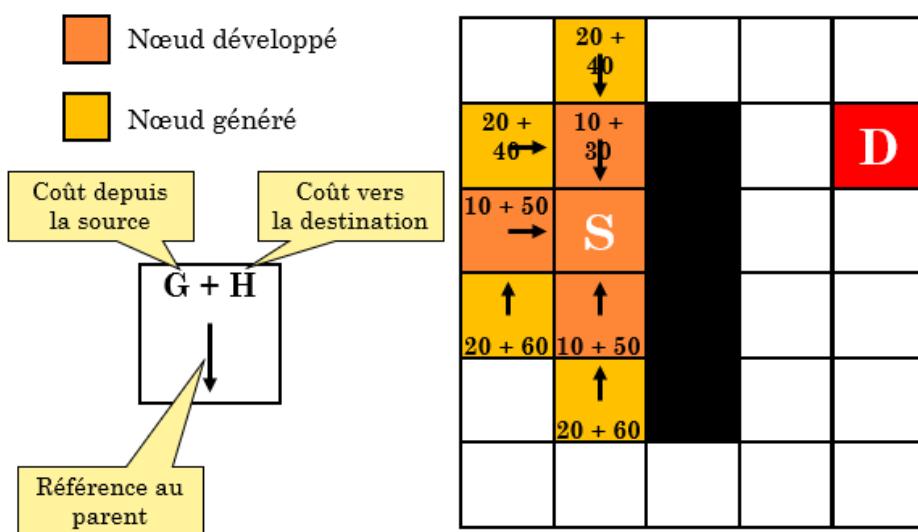
Exemple 1



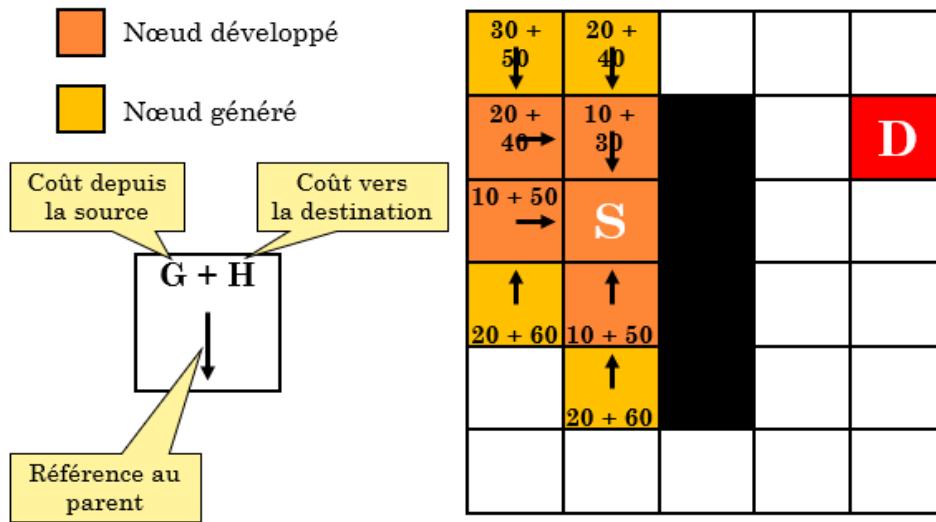
Exemple 1



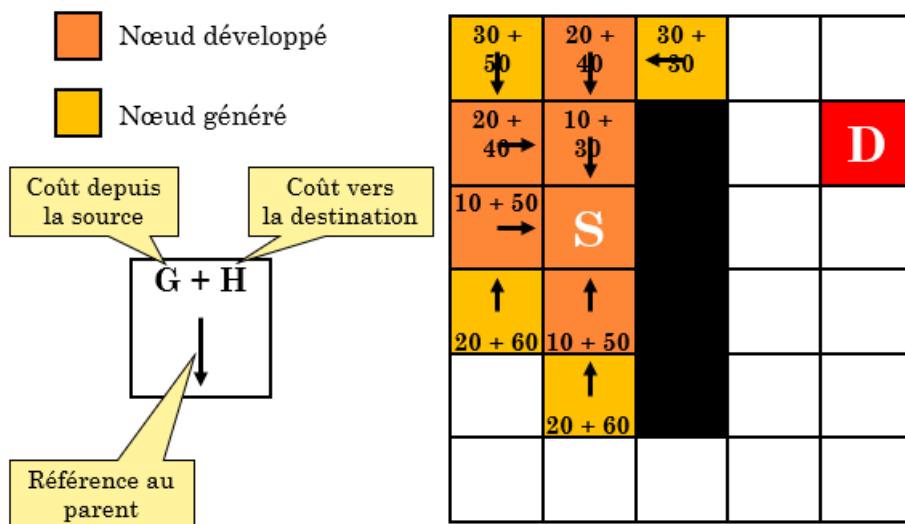
Exemple 1



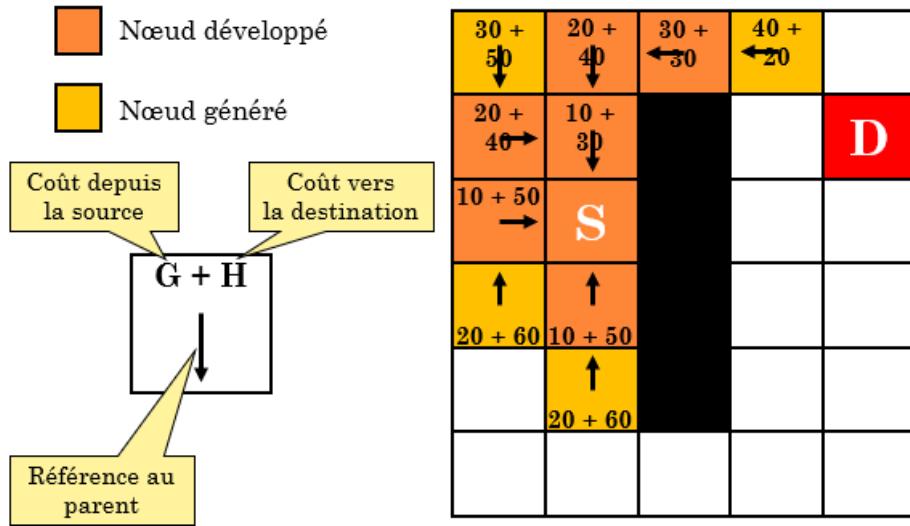
Exemple 1



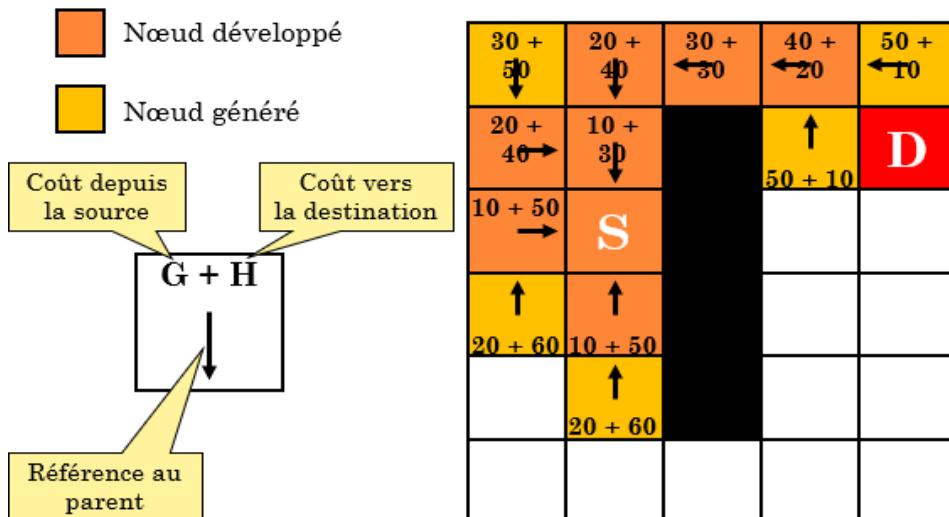
Exemple 1



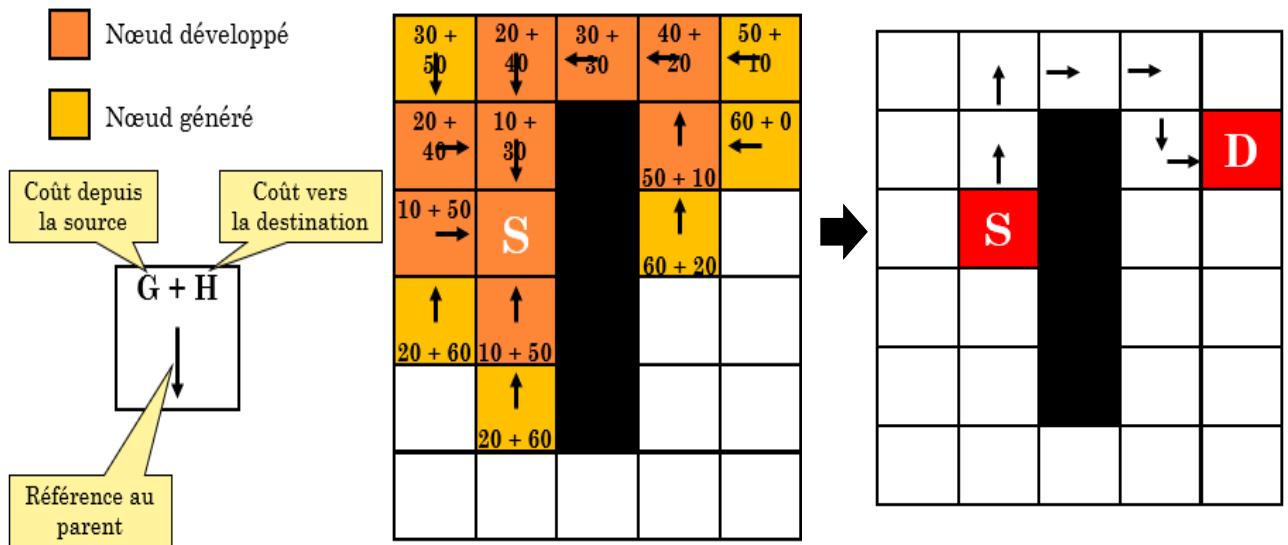
Exemple 1



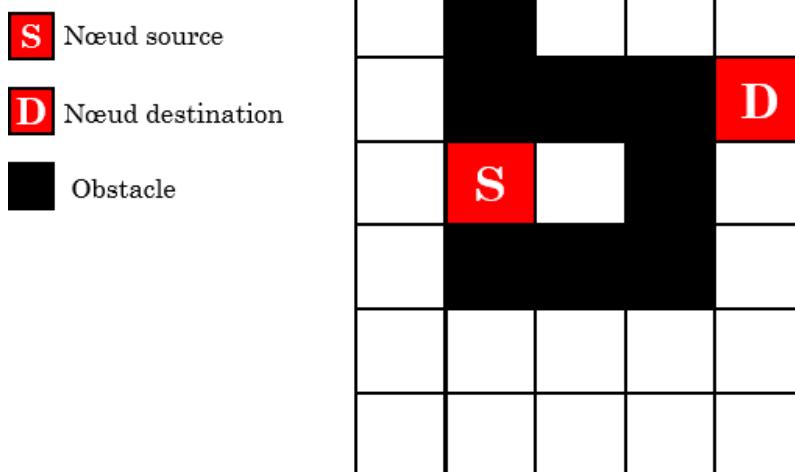
Exemple 1



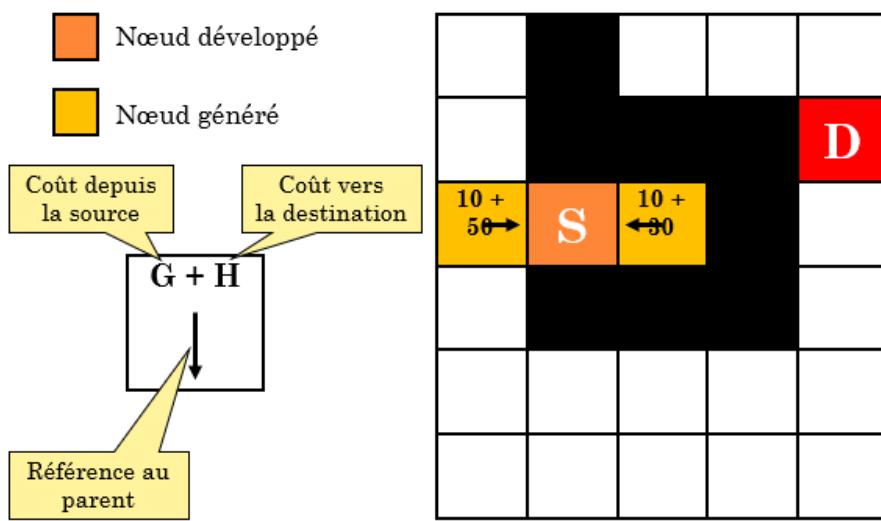
Exemple 1



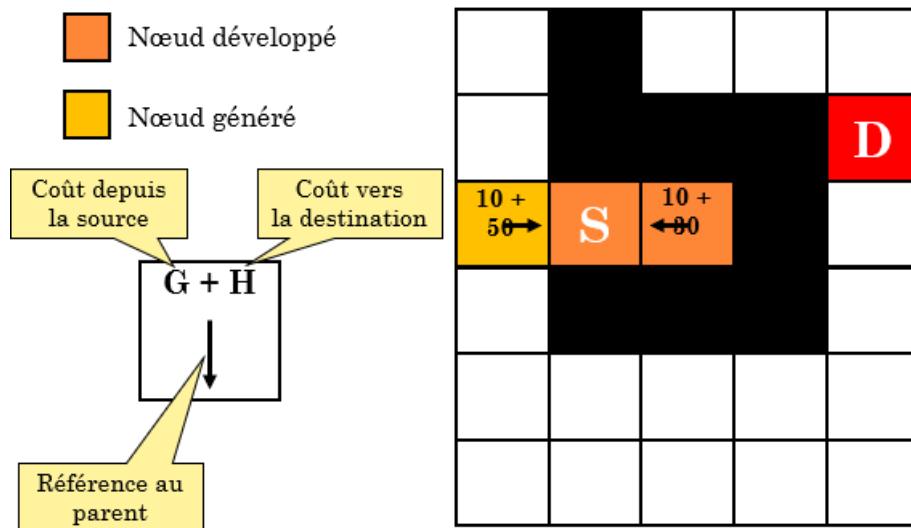
Exemple 2:



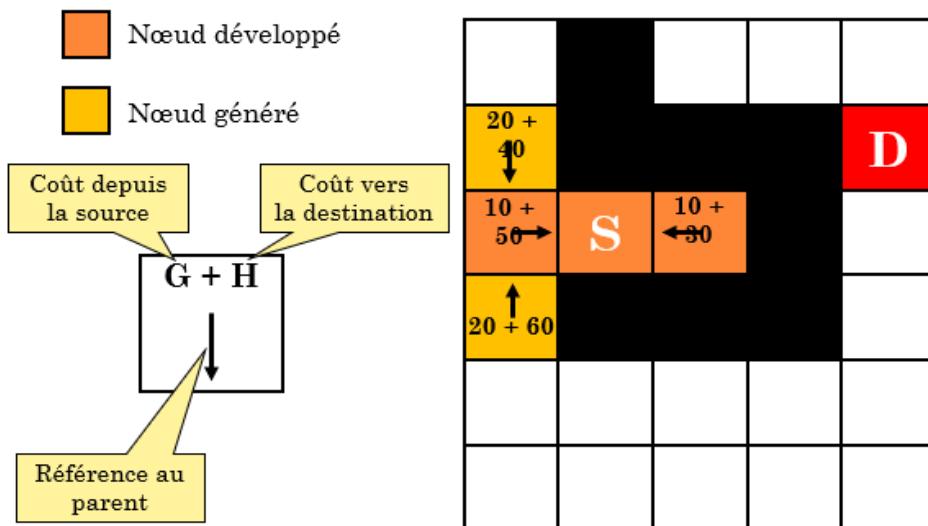
Exemple 2:



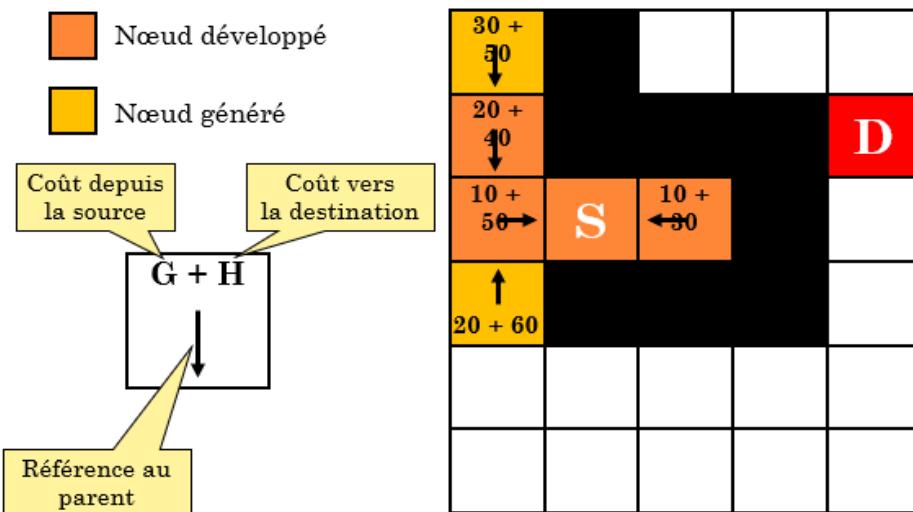
Exemple 2:



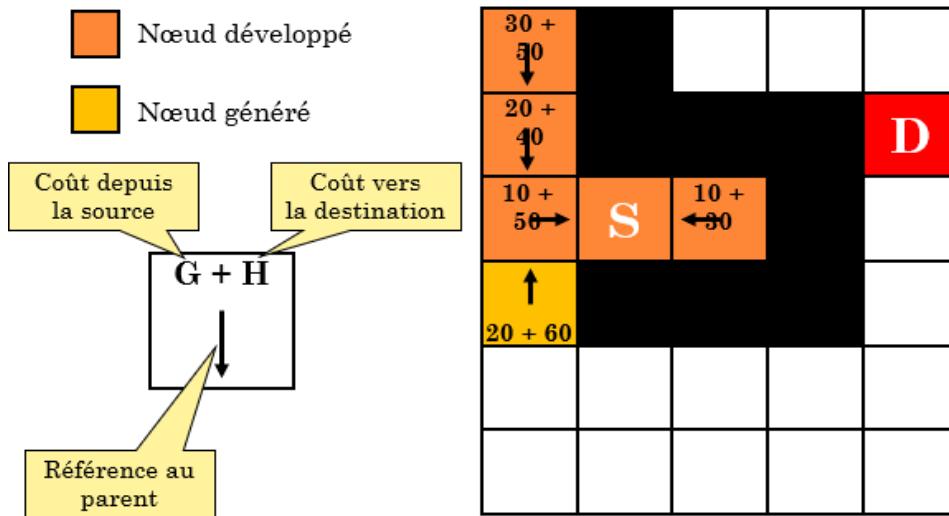
Exemple 2:



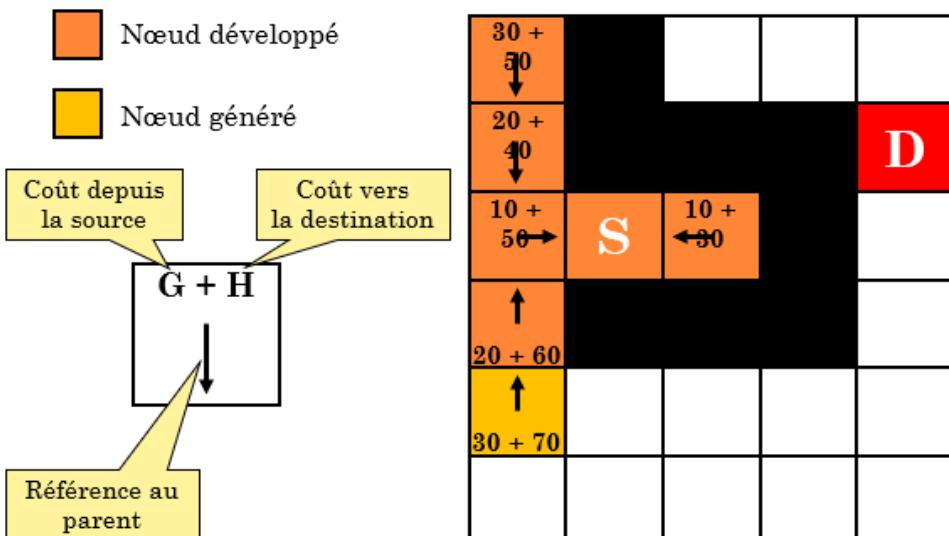
Exemple 2:



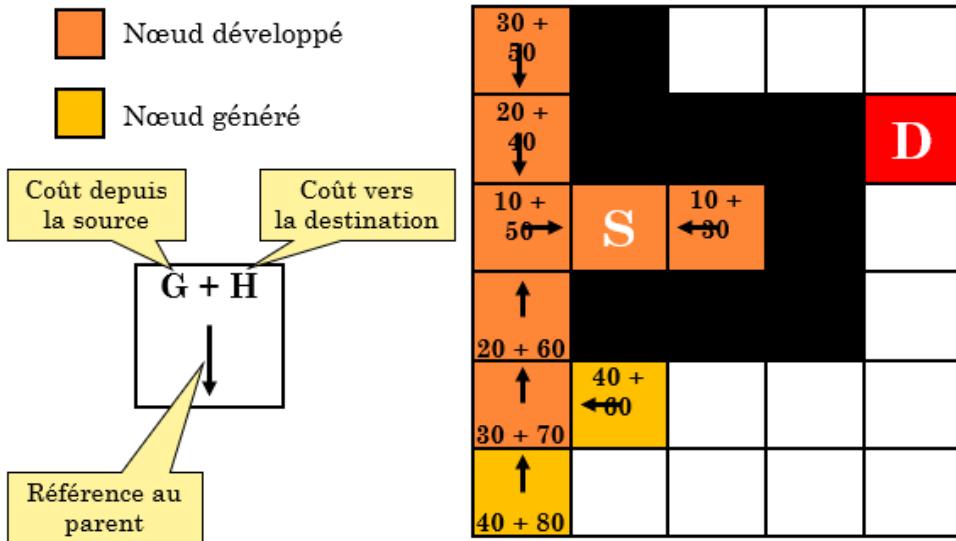
Exemple 2:



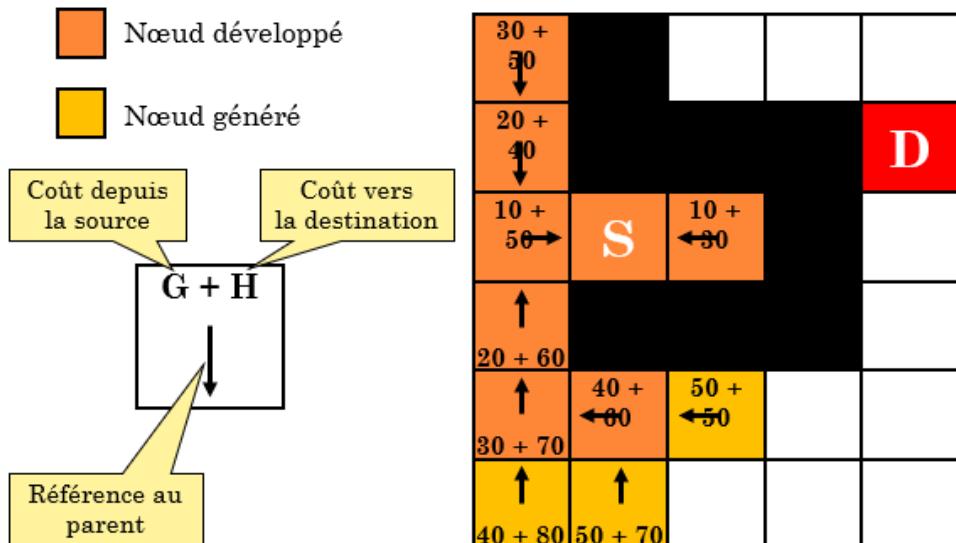
Exemple 2:



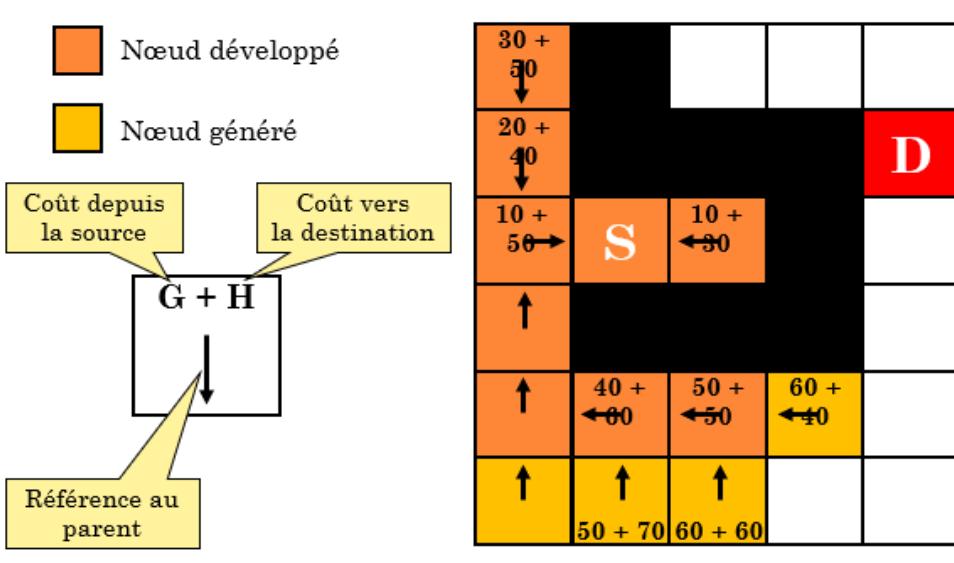
Exemple 2:



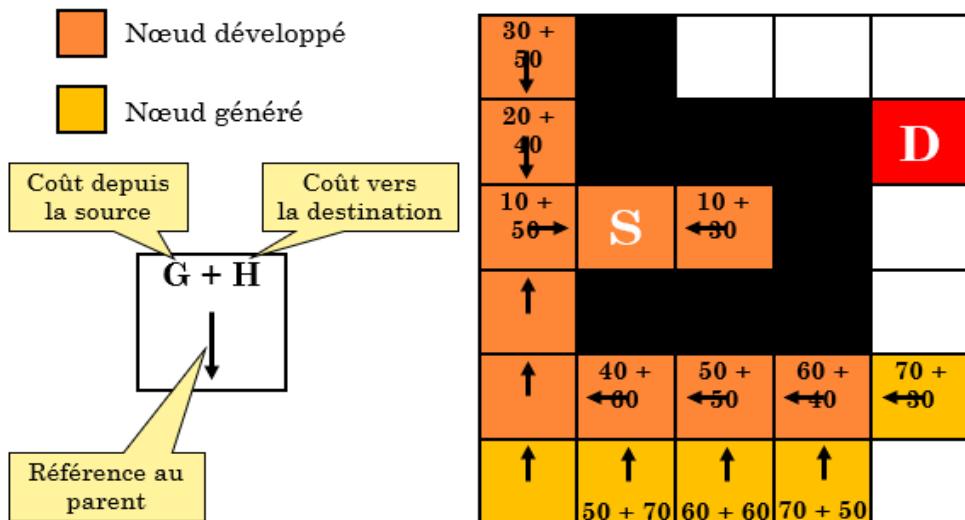
Exemple 2:



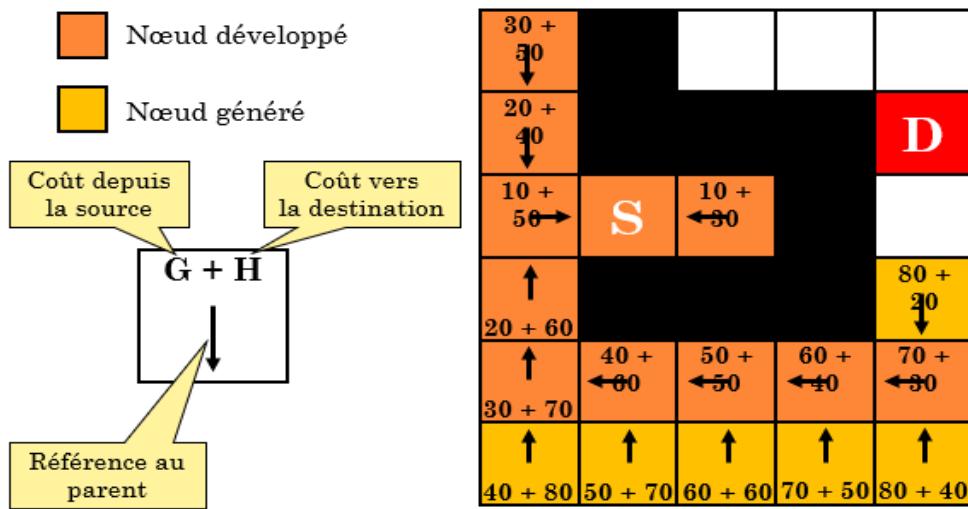
Exemple 2:



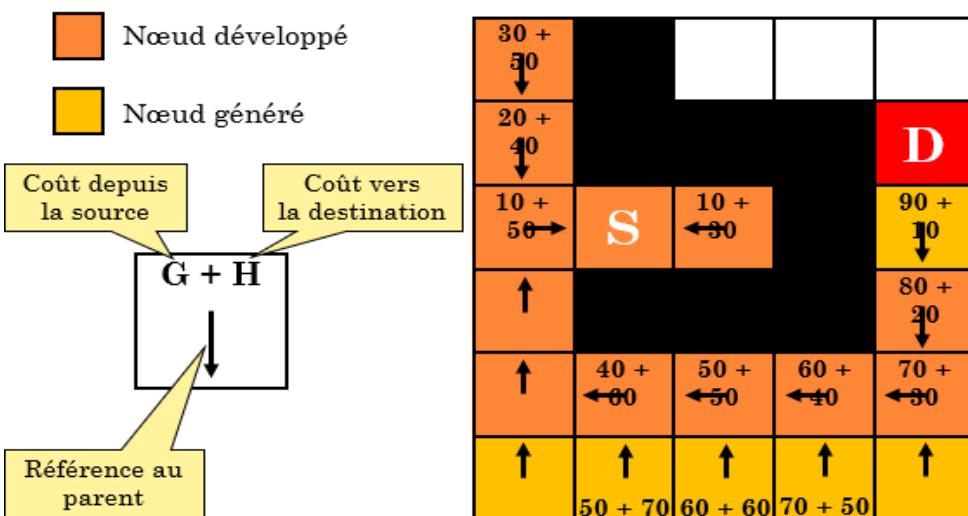
Exemple 2:



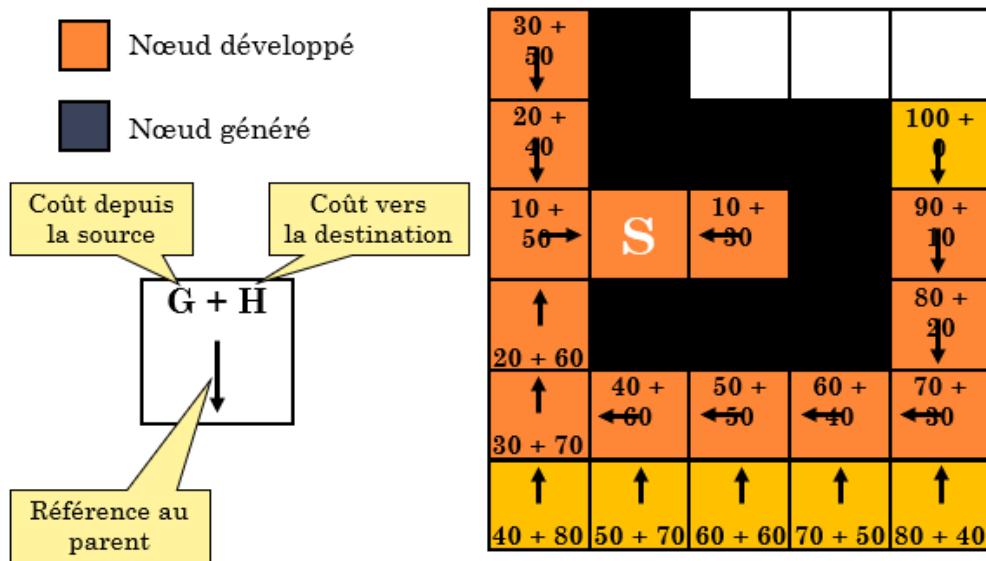
Exemple 2:



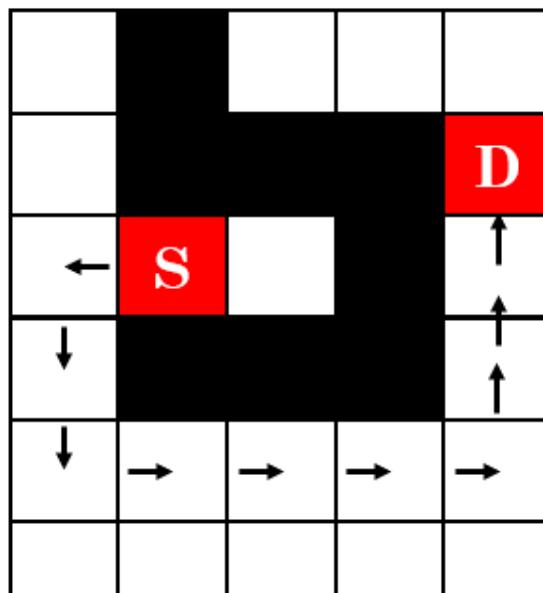
Exemple 2:



Exemple 2:



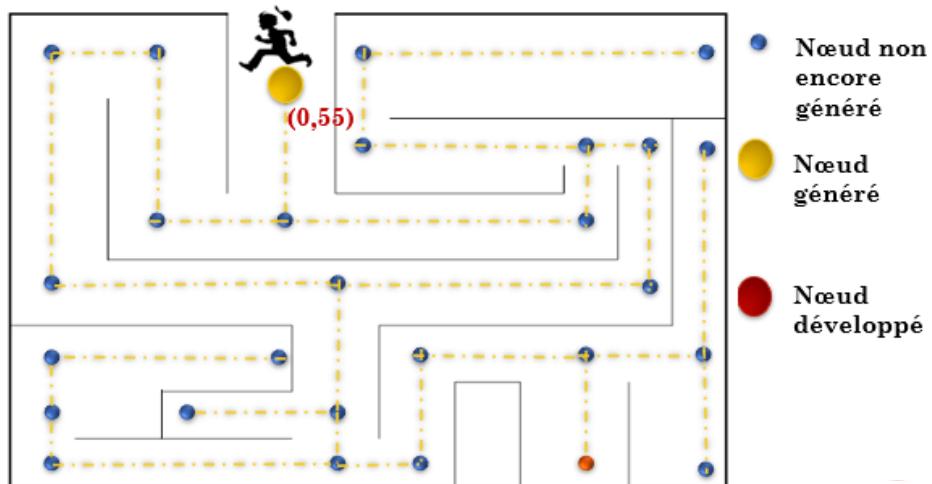
Exemple 2: chemin complet



Application (labyrinthe)

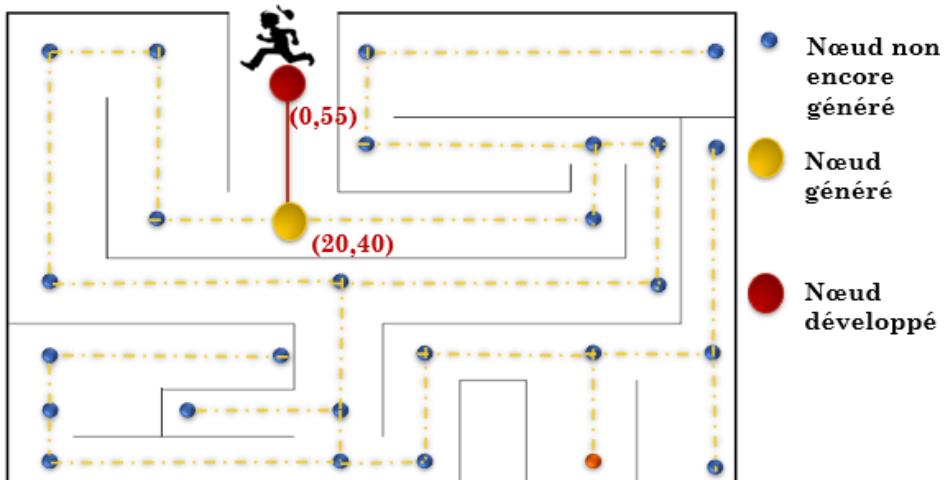
Au début la distance parcourue est 0

(Distance parcourue, Distance estimée)



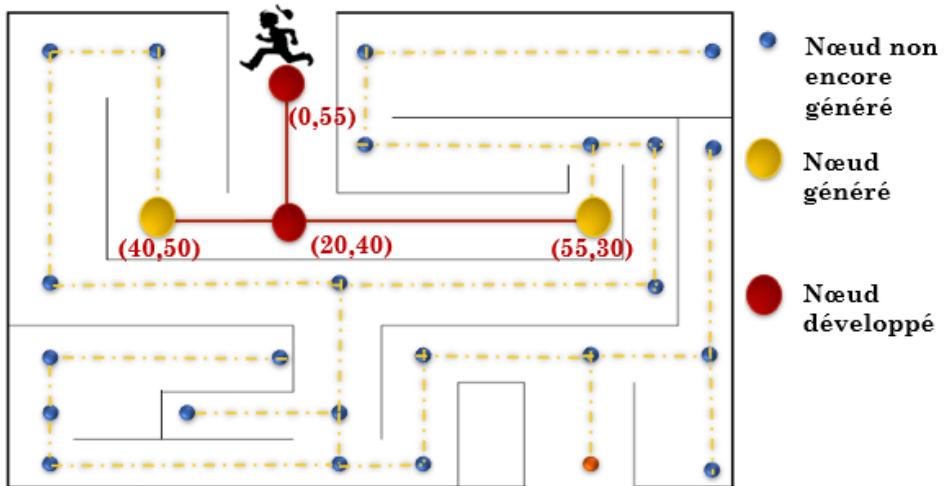
Application (labyrinthe)

(Distance parcourue, Distance estimée)



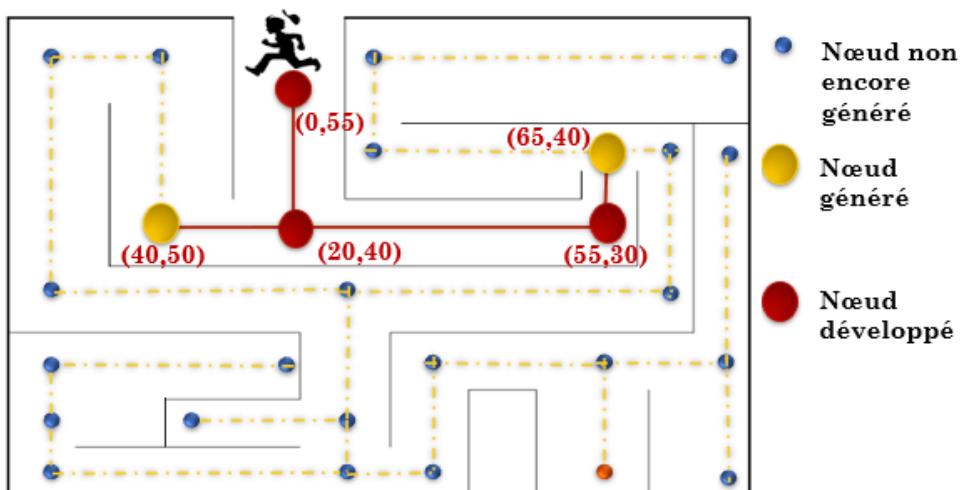
Application (labyrinthe)

(Distance parcourue, Distance estimée)



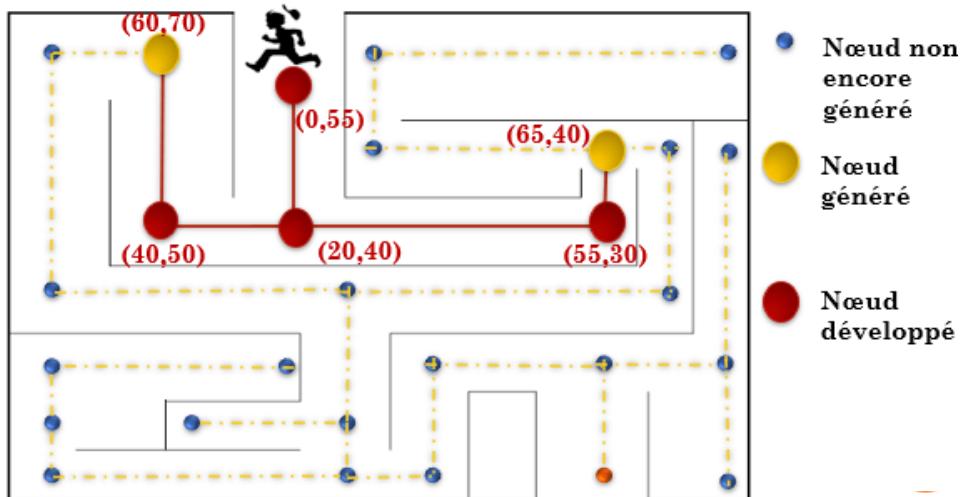
Application (labyrinthe)

(Distance parcourue, Distance estimée)



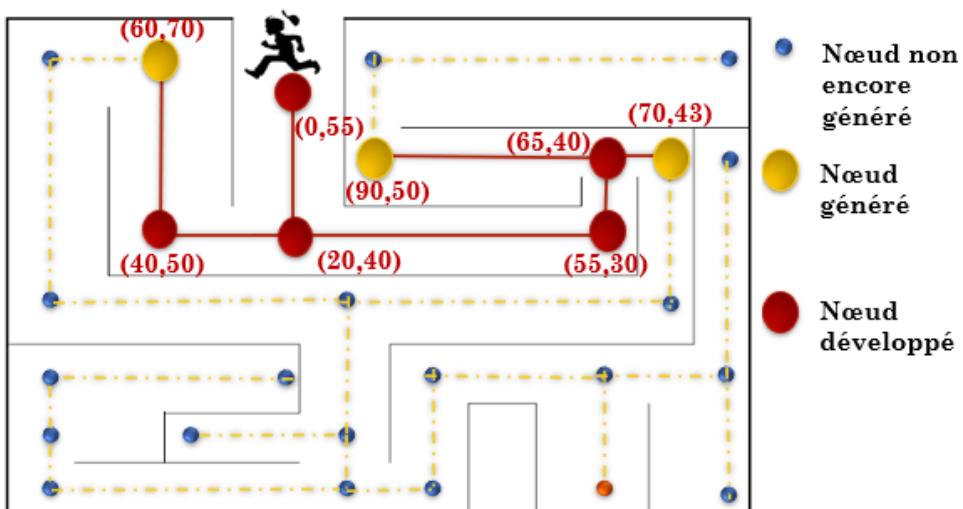
Application (labyrinthe)

(Distance parcourue, Distance estimée)



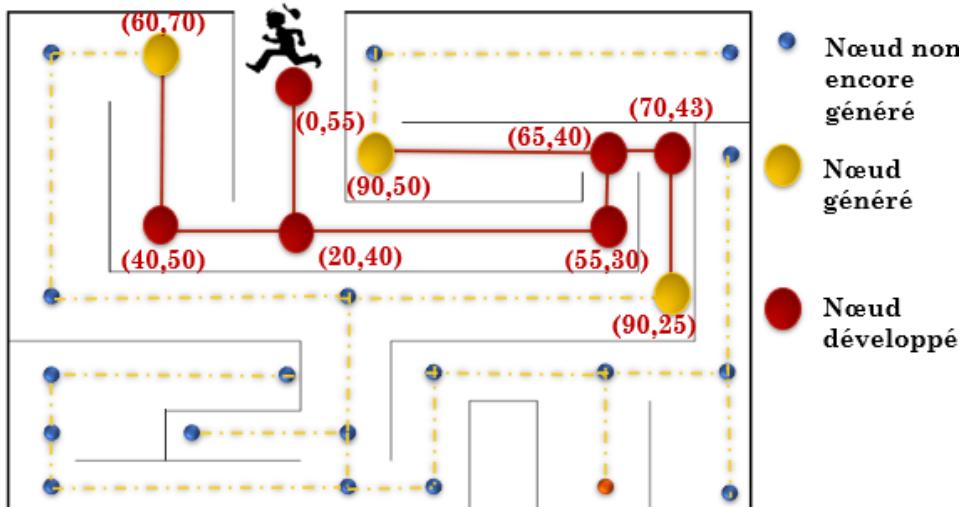
Application (labyrinthe)

(Distance parcourue, Distance estimée)



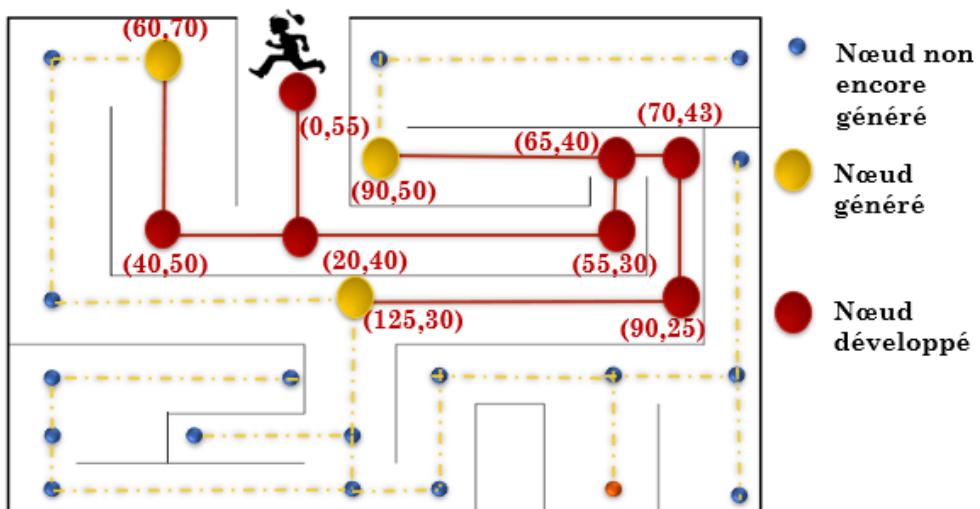
Application (labyrinthe)

(Distance parcourue, Distance estimée)

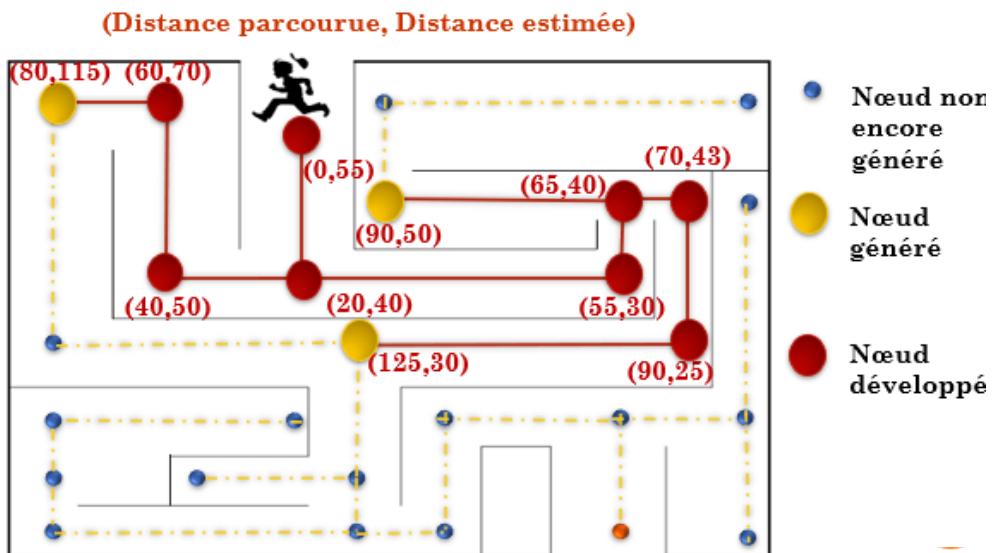


Application (labyrinthe)

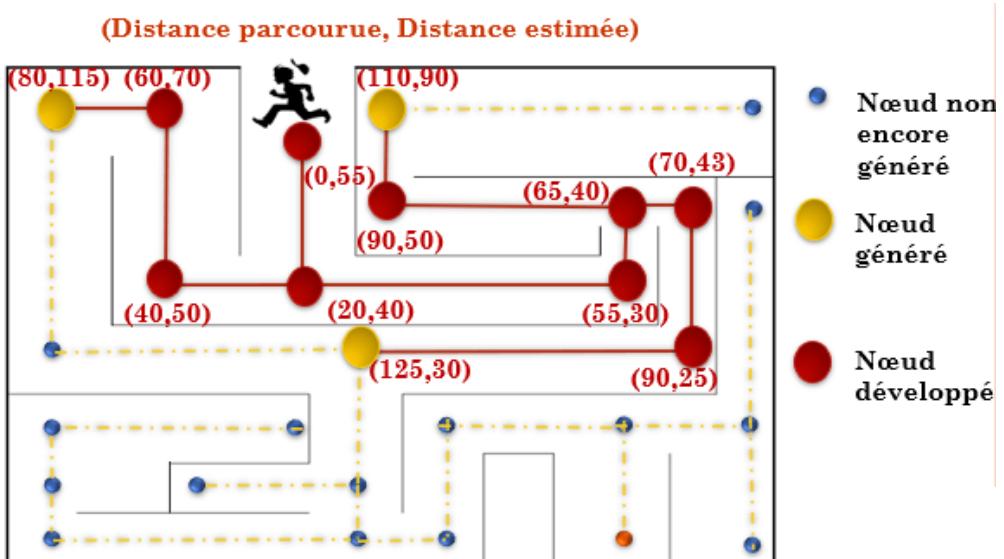
(Distance parcourue, Distance estimée)



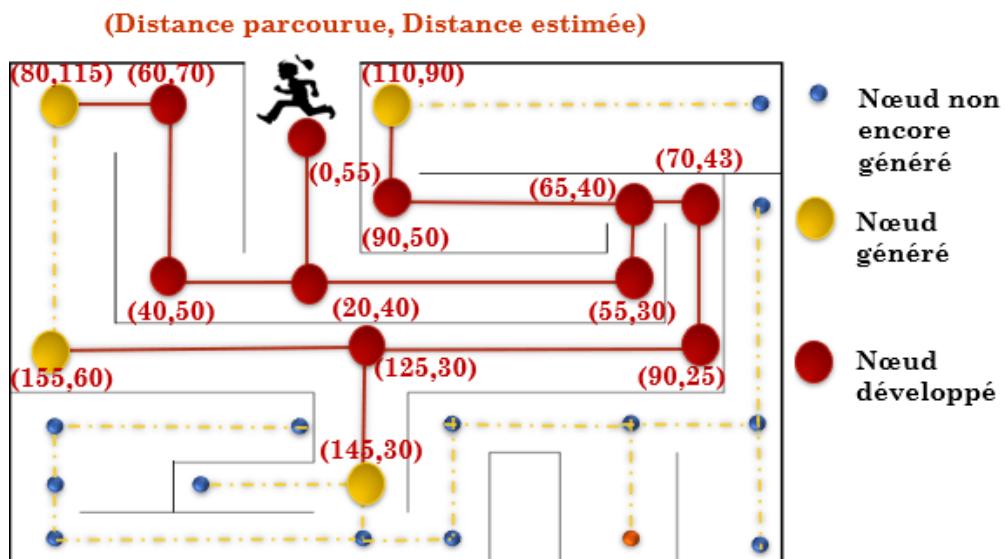
Application (labyrinthe)



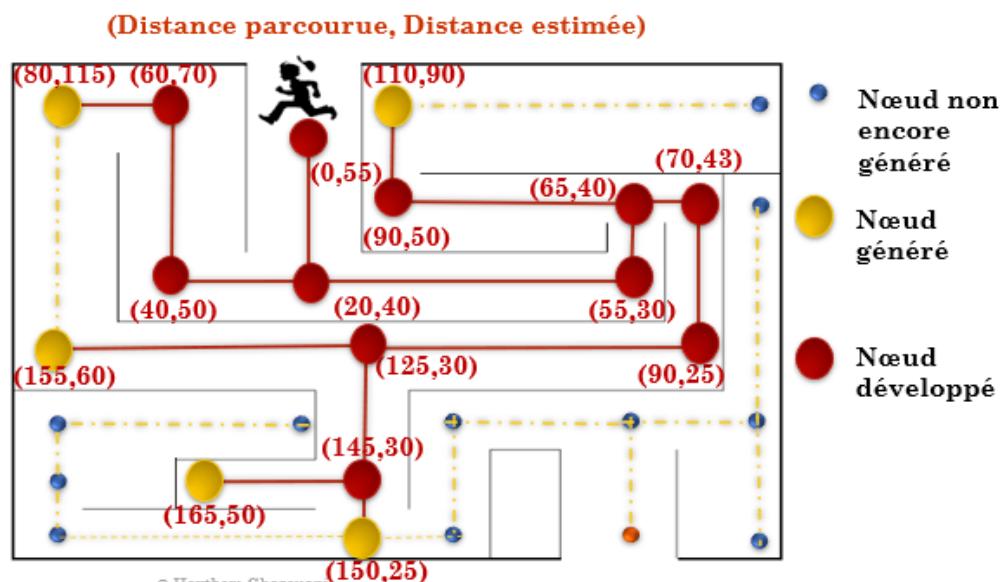
Application (labyrinthe)



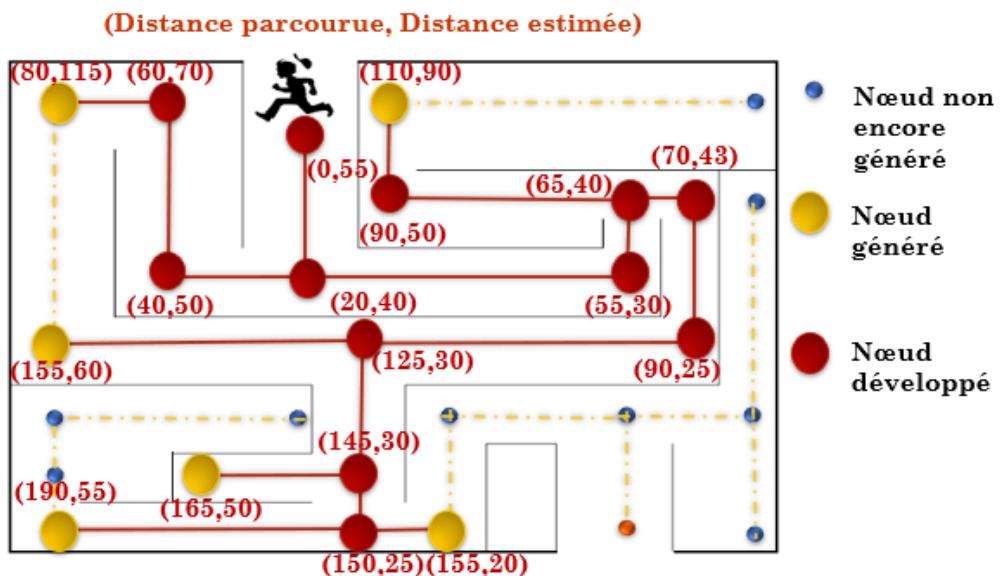
Application (labyrinthe)



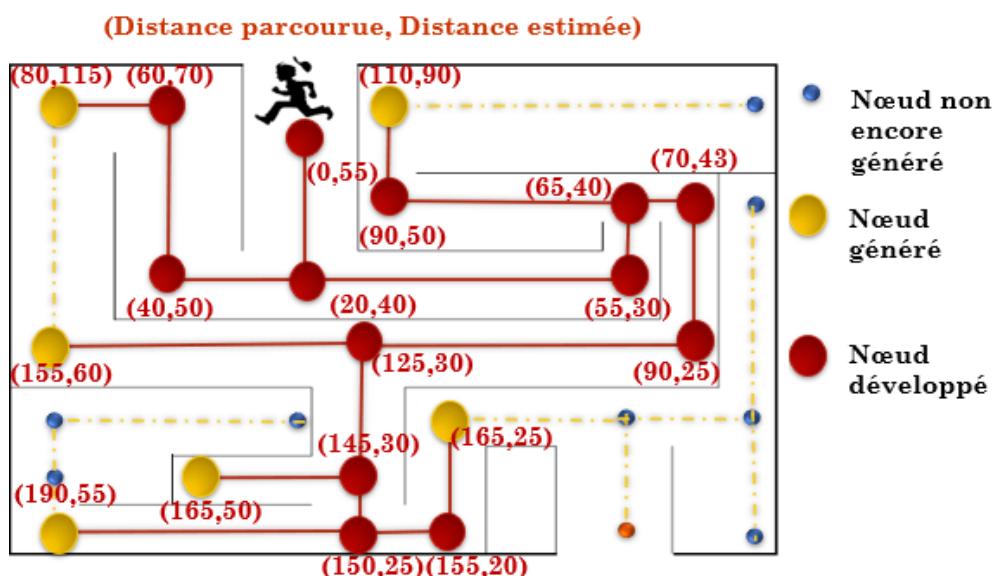
Application (labyrinthe)



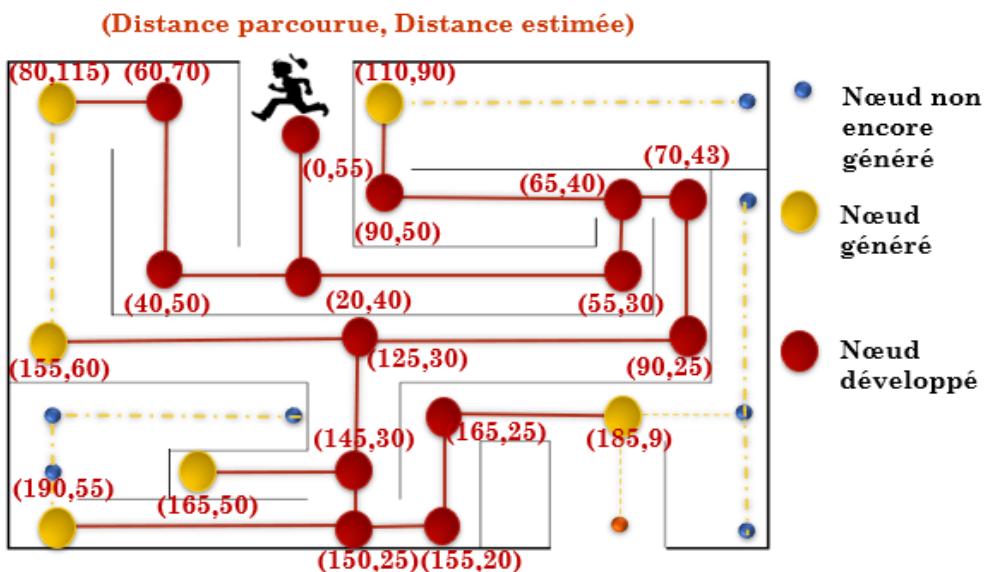
Application (labyrinthe)



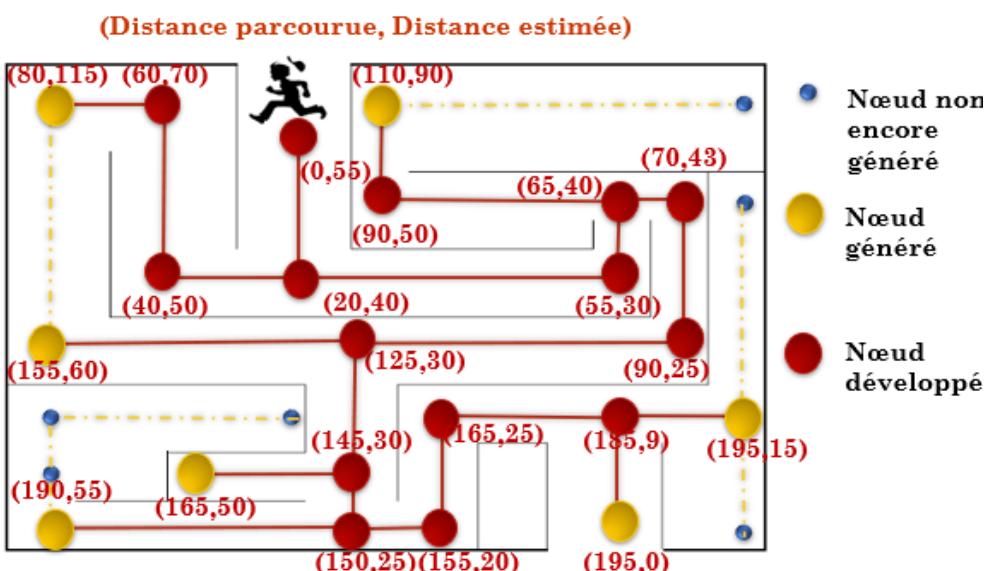
Application (labyrinthe)



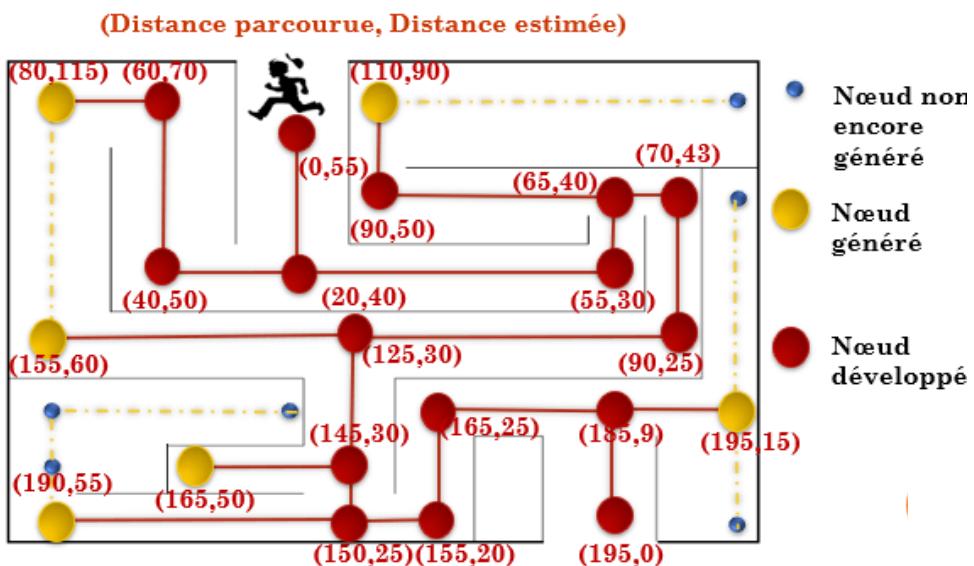
Application (labyrinthe)



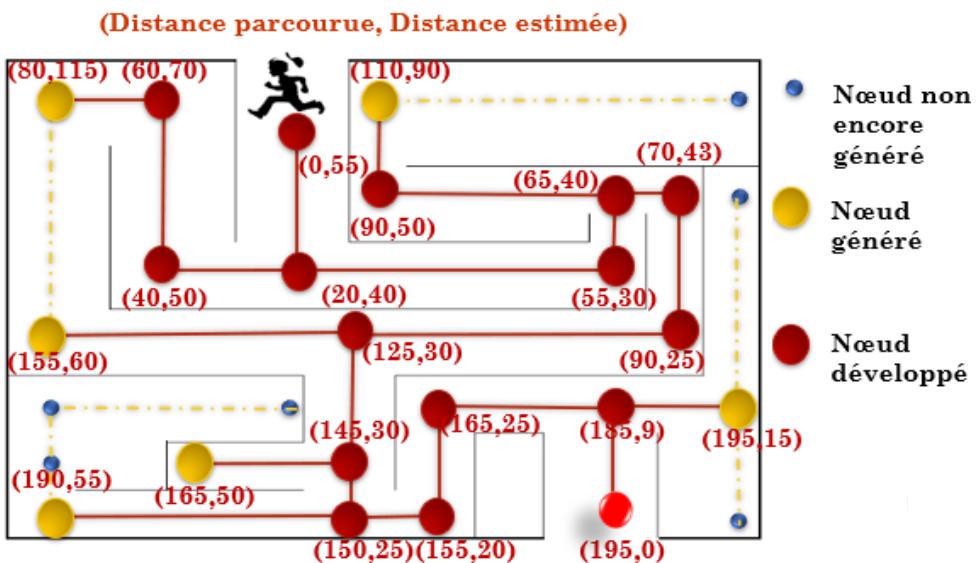
Application (labyrinthe)



Application (labyrinthe)



Application (labyrinthe)



Stratégies de Recherche informées

- Tout algorithme de recherche heuristique dispose d'une fonction d'évaluation f qui détermine l'ordre dans lequel les nœuds sont traités : la liste de nœuds à traiter est organisée en fonction des f -valeurs des nœuds, avec les nœuds de plus petite valeur en tête de liste.

□ Il existe plusieurs stratégies :

1. Best-first Search
 - 1.1. Exploration gloutonne par le meilleur
 - 1.2. A*
2. Exploration heuristique à mémoire limitée:
IDA*, RBFS, SMA*

181

A* avec approfondissement itératif (IDA*)

• Principe :

IDA* combine l'approche d'A* et la recherche en profondeur itérative. Il utilise une borne sur la fonction $f(n)=g(n)+h(n)$

==> **La limite est le coût f et non la profondeur.**

- Exemple : les nœuds n où $f(n) > 10$ ne sont pas pris en considération!

- Lors de chaque itération d'IDA* :

1. On fixe une limite f (**initiallement $f(start)=h(start)$**).
2. On explore en profondeur, en respectant cette limite.
3. Si un nœud dépasse la limite actuelle, on enregistre sa **plus petite valeur $f(n)$** .
4. À l'itération suivante, la limite devient cette **plus petite valeur** parmi les nœuds dépassant la limite précédente.
5. On répète jusqu'à trouver la solution optimale.

A chaque itération: On fixe la limite à la plus petite valeur $f(n)$ de tous les nœuds qui avaient une valeur plus grande que la limite au tour précédent.

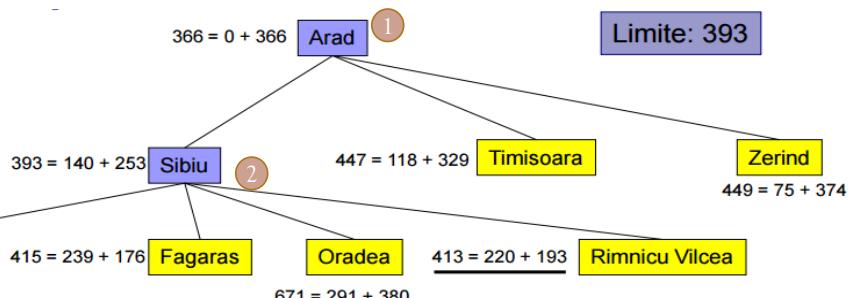
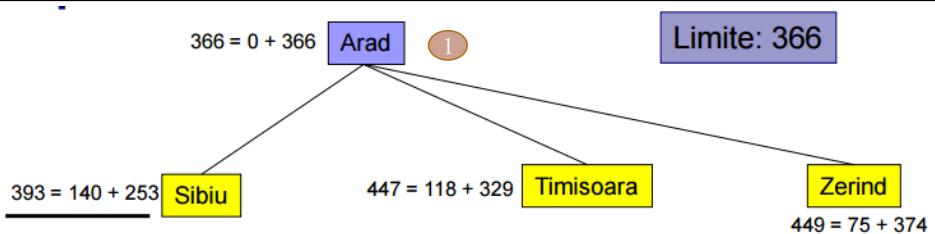
A* avec approfondissement itératif (IDA*)

- Cette approche permet à IDA* d'éviter de stocker tous les noeuds en mémoire, contrairement à A*, tout en garantissant l'optimalité si $h(n)$ est admissible.
- Prend moins de mémoire, comparable à l'exploration en profondeur d'abord car évite le maintien d'une liste triée.
- Gagne en temps d'exploration par noeud puisque il ne le place pas dans une liste triée

=> IDA* explore en profondeur en respectant une limite de coût.

=> Il ajuste la limite progressivement pour éviter d'explorer inutilement des chemins coûteux.

A* avec approfondissement itératif (IDA*): exemple



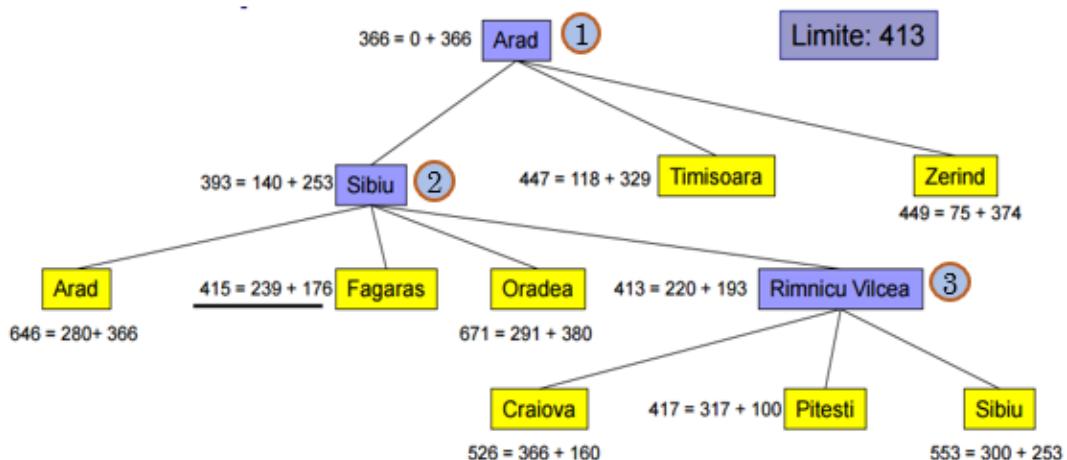
$646 = 280 + 366$

$415 = 239 + 176$

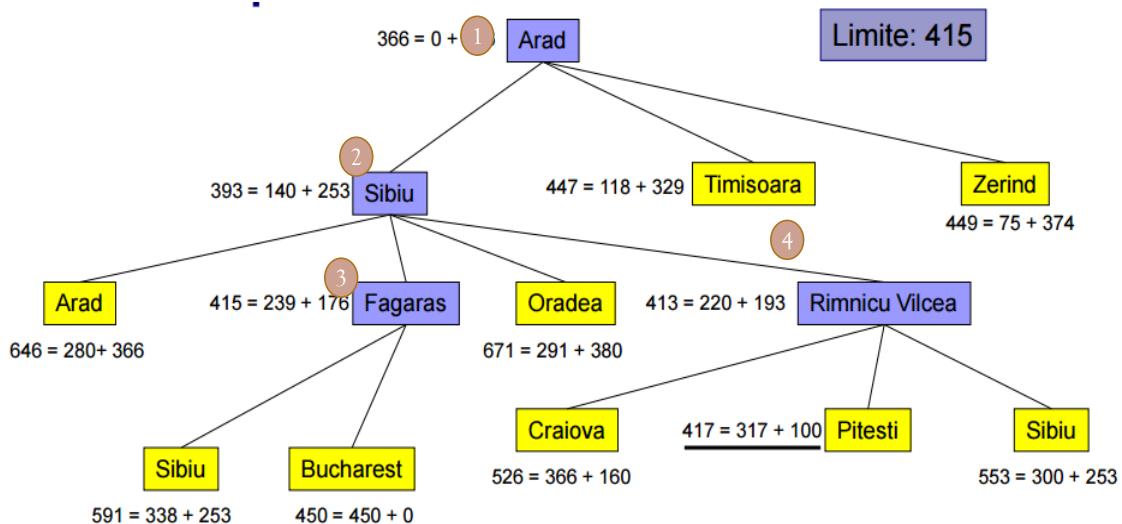
$413 = 220 + 193$

$671 = 291 + 380$

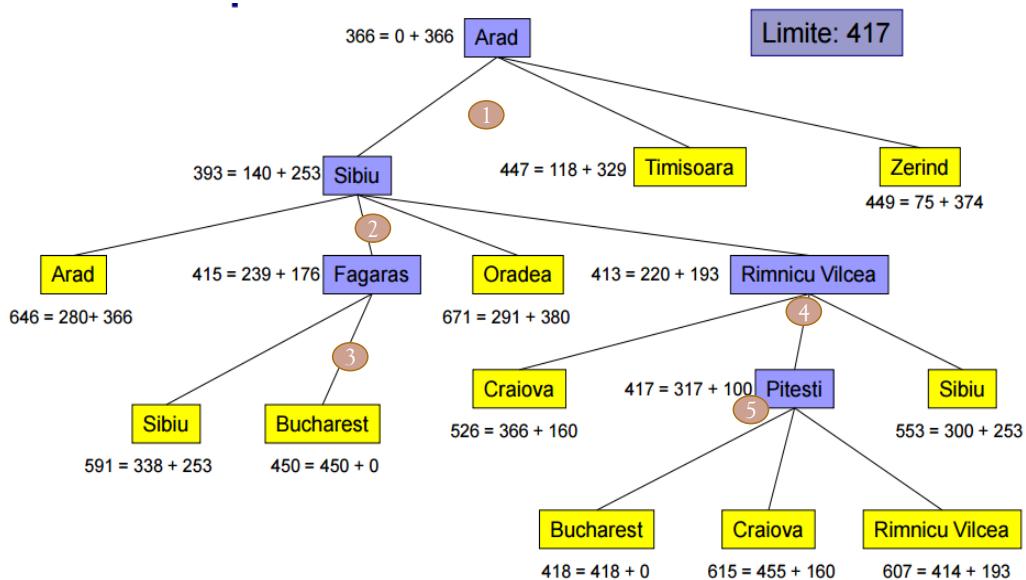
A* avec approfondissement itératif (IDA*): exemple



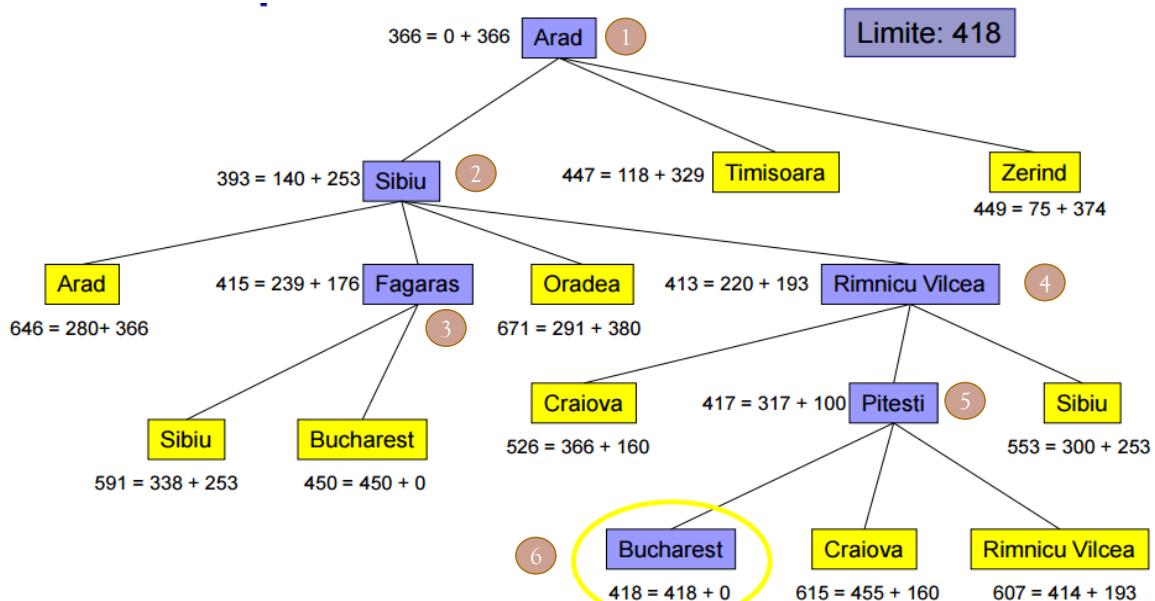
A* avec approfondissement itératif (IDA*): exemple



A* avec approfondissement itératif (IDA*): exemple



A* avec approfondissement itératif (IDA*): exemple



A* vs IDA*

A*

- Stocke tous les noeuds dans une file de priorité (open list) triée par $f(n)=g(n)+h(n)$
- Chaque itération prend le noeud avec le plus petit $f(n)$ et explore ses successeurs.
- Garde une mémoire exponentielle, car il stocke tous les noeuds explorés.
- Exploration en largeur contrôlée, car il compare plusieurs branches simultanément.

IDA*

- N'utilise pas de file de priorité : il explore les chemins **par profondeur**.
 - Fixe une **limite de coût $f(n)$** et explore uniquement les chemins en dessous de cette limite.
 - Si aucun chemin ne mène à la solution avec la limite actuelle, il **augmente la limite et recommence**.
 - Utilise beaucoup moins de mémoire, car il **ne stocke pas** tous les noeuds visités.
- ➔ IDA* explore l'arbre de recherche en augmentant progressivement la limite

Exploration récursive par le meilleur d'abord (RBFS)

- Variante de l'exploration par le meilleur d'abord.
- Utilise un espace de temps linéaire.
- L'exploration **se fait récursivement en profondeur d'abord**.
- Mais ne continue pas indéfiniment dans un chemin, elle pause **une limite** correspondant à la **valeur f du meilleur chemin alternatif parmi les ancêtres**.

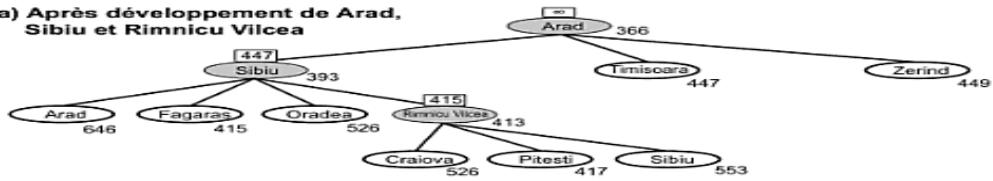
RBFS (Recursive Best-First Search) est une version améliorée de Best-First Search qui utilise moins de mémoire. Il explore récursivement le meilleur chemin tout en limitant l'espace utilisé

Idée principale

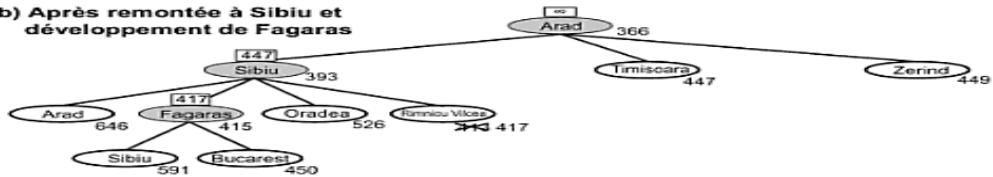
- On explore le noeud avec le plus petit $f(n)$.
- Si la solution est trouvée, on s'arrête.
- Sinon, on fixe une limite sur $f(n)$ (**le deuxième plus petit $f(n)$ des noeuds non explorés**).
- Si un noeud dépasse cette limite, on recule et essaie un autre chemin.
- La limite est mise à jour dynamiquement à chaque retour arrière.

Exploration récursive par le meilleur d'abord (RBFS)

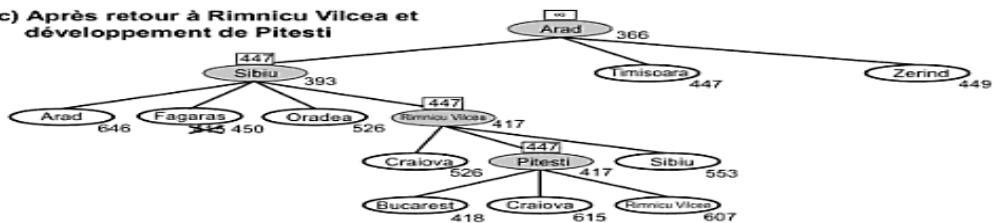
(a) Après développement de Arad, Sibiu et Rimnicu Vilcea



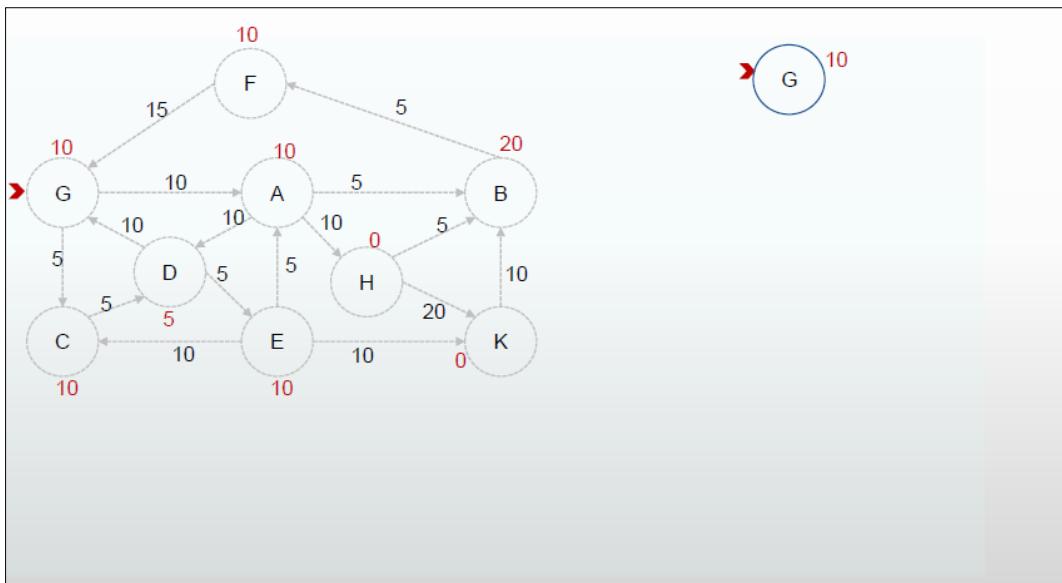
(b) Après remontée à Sibiu et développement de Fagaras



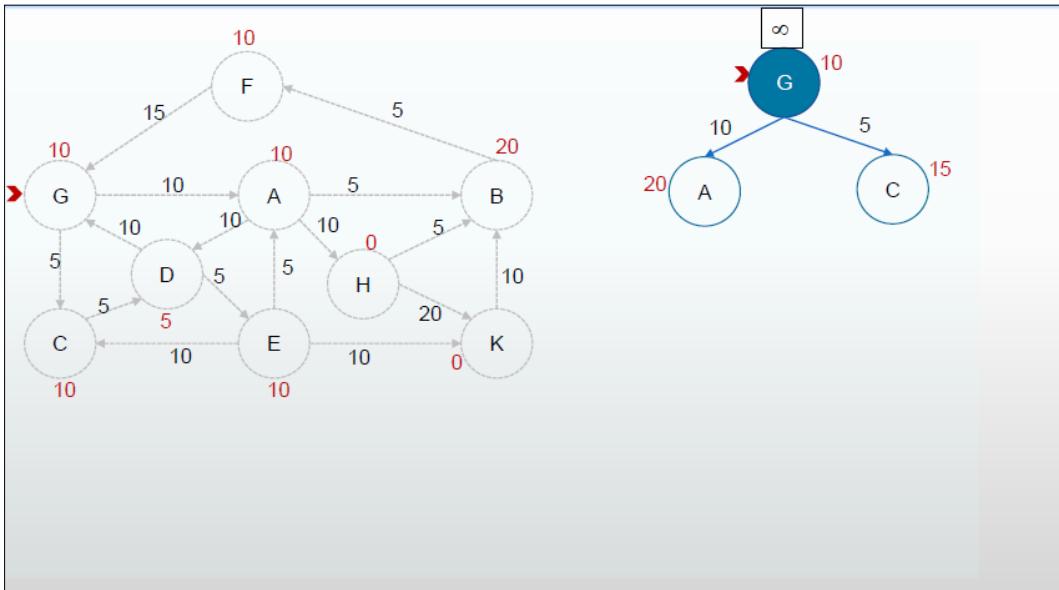
(c) Après retour à Rimnicu Vilcea et développement de Pitesti



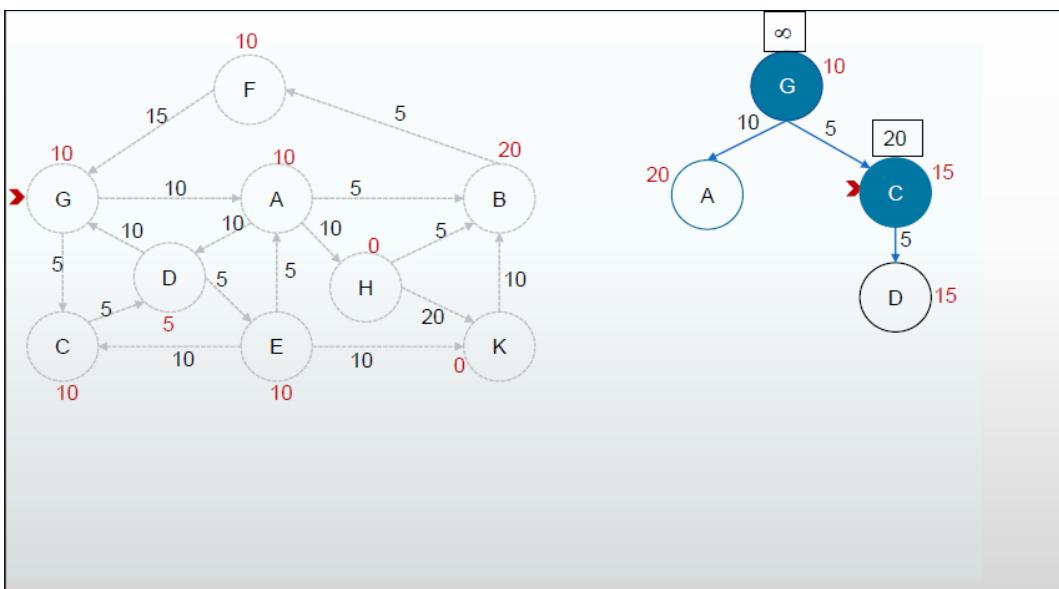
Exploration récursive par le meilleur d'abord (RBFS)



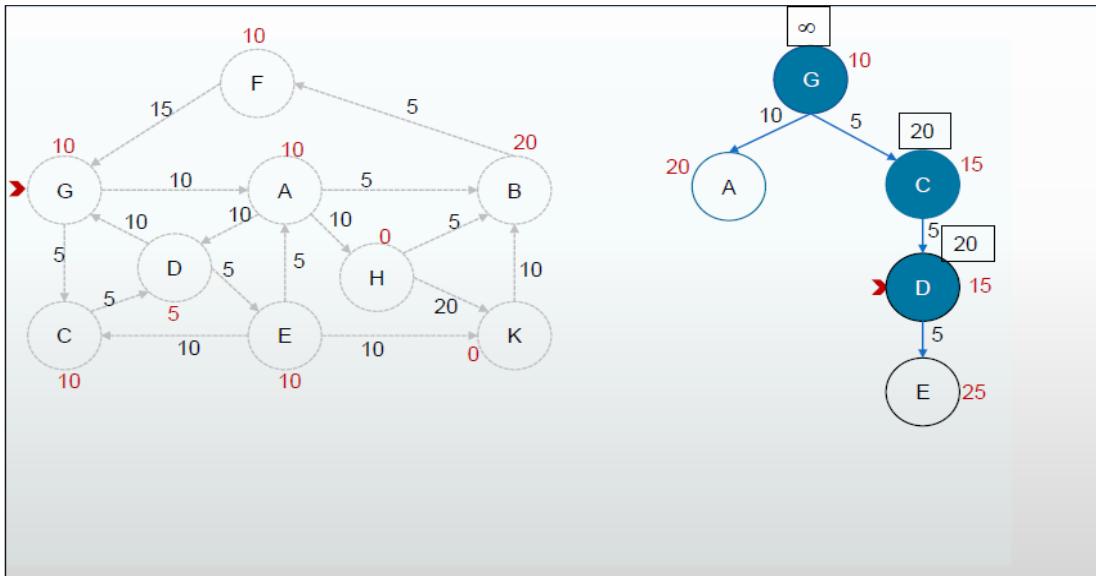
Exploration récursive par le meilleur d'abord (RBFS)



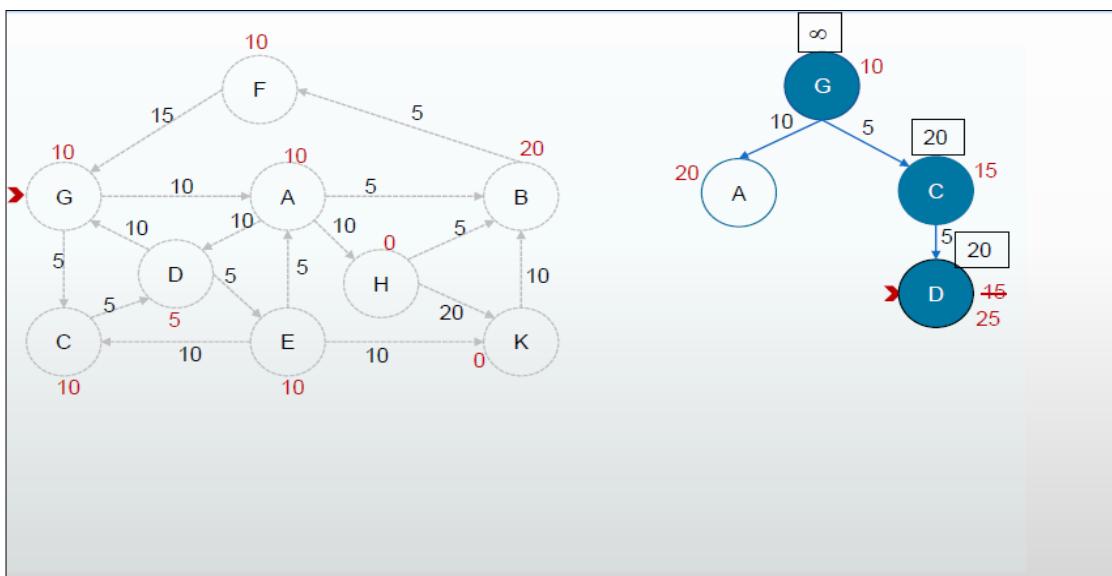
Exploration récursive par le meilleur d'abord (RBFS)



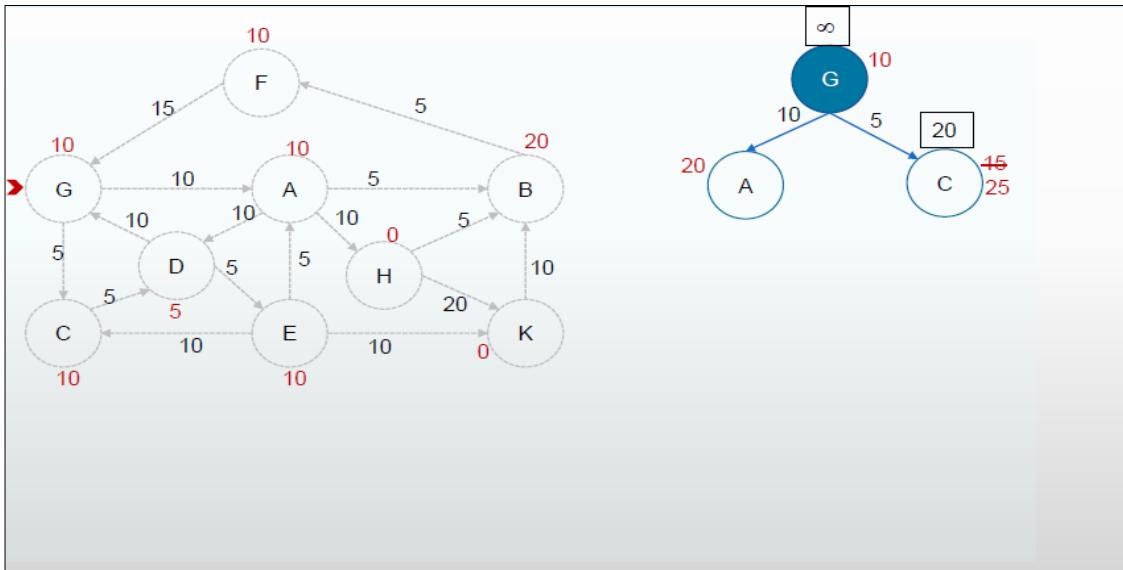
Exploration récursive par le meilleur d'abord (RBFS)



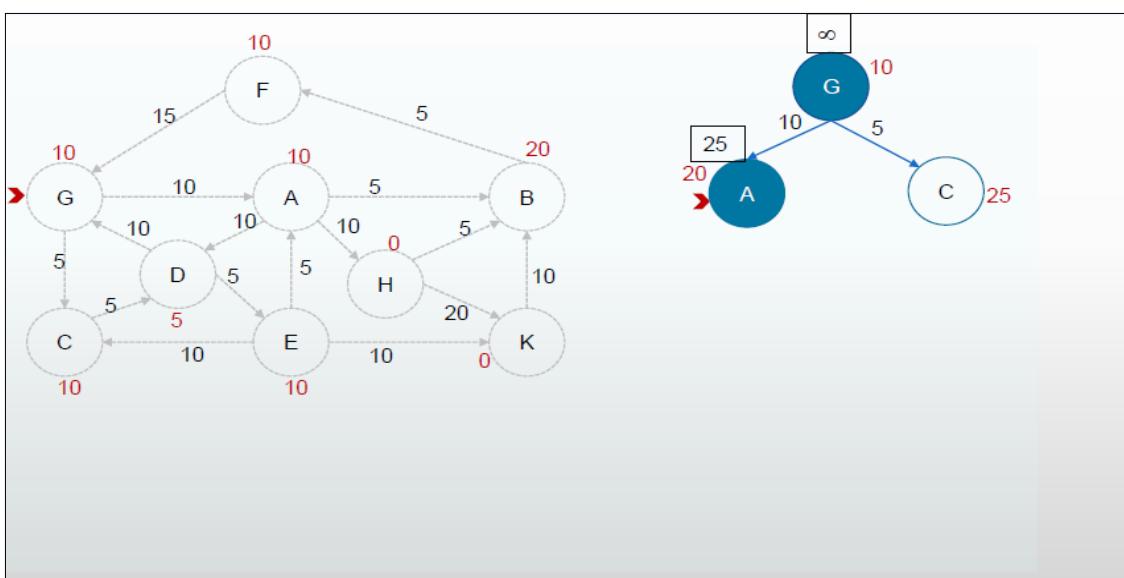
Exploration récursive par le meilleur d'abord (RBFS)



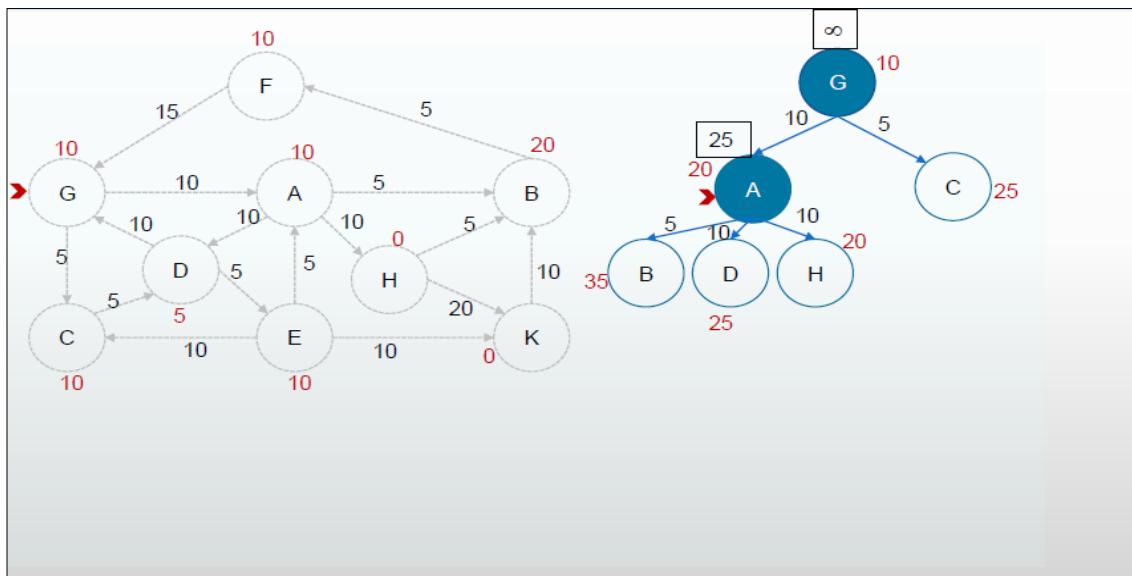
Exploration récursive par le meilleur d'abord (RBFS)



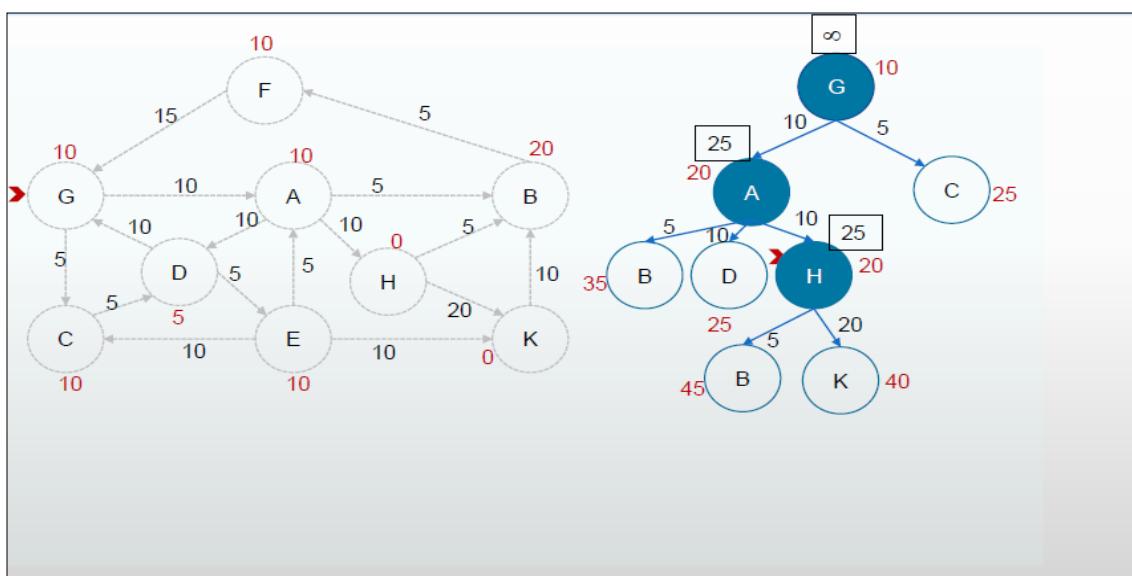
Exploration récursive par le meilleur d'abord (RBFS)



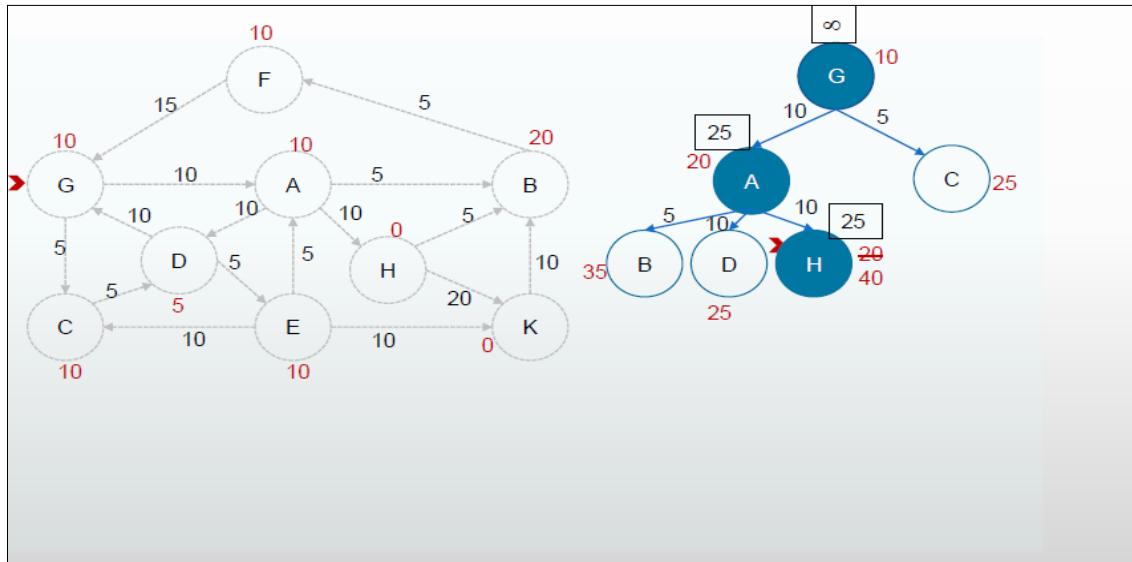
Exploration récursive par le meilleur d'abord (RBFS)



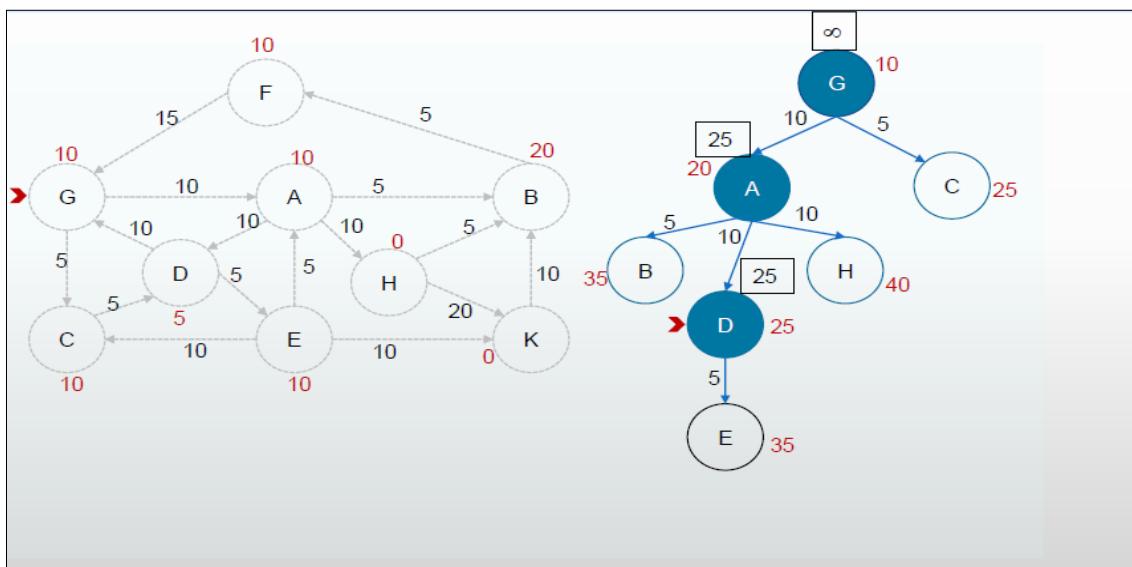
Exploration récursive par le meilleur d'abord (RBFS)



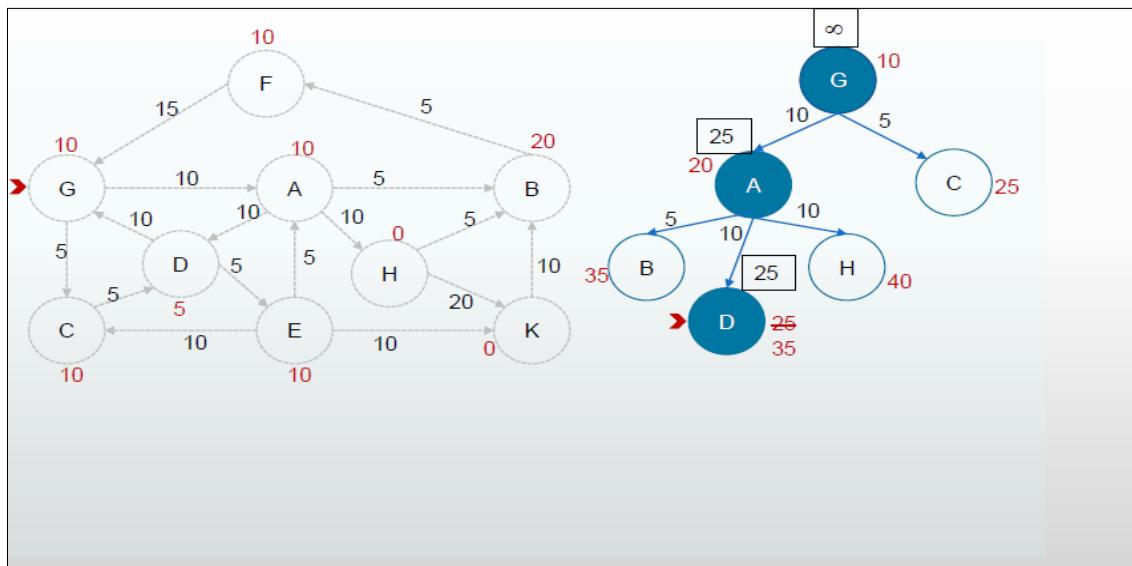
Exploration récursive par le meilleur d'abord (RBFS)



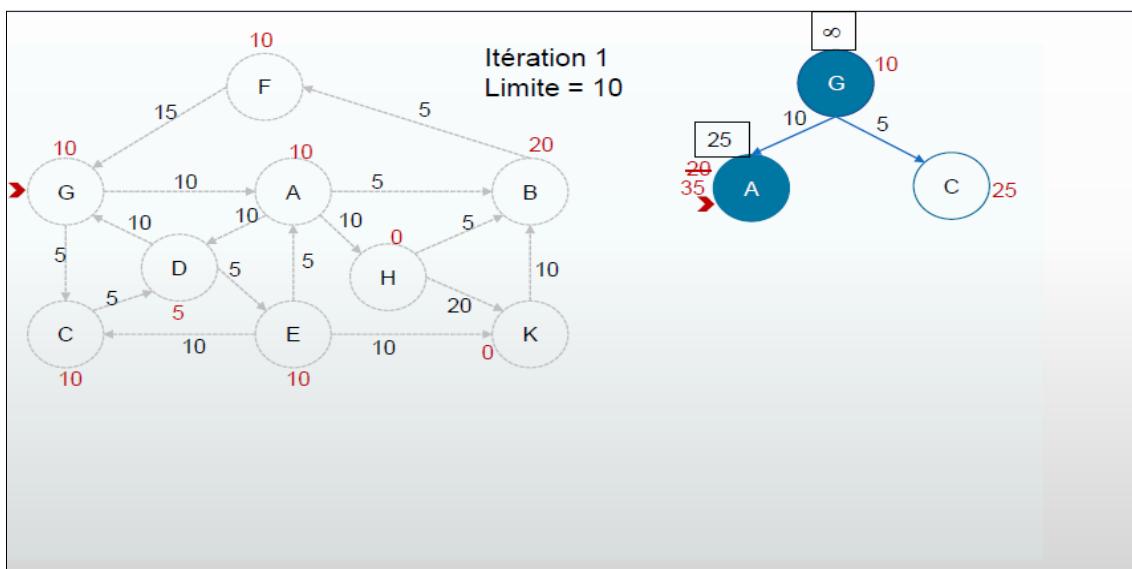
Exploration récursive par le meilleur d'abord (RBFS)



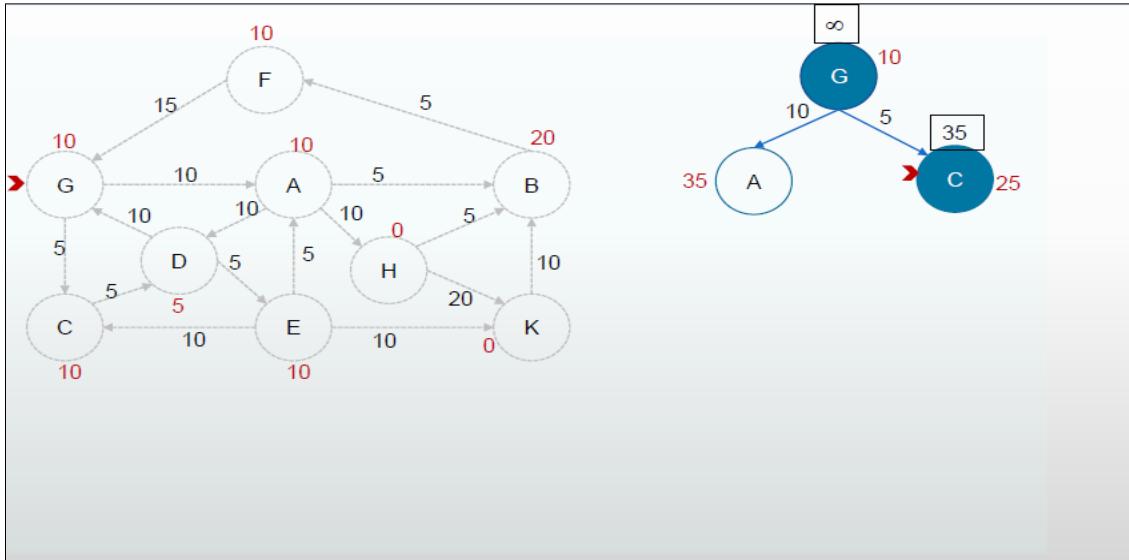
Exploration récursive par le meilleur d'abord (RBFS)



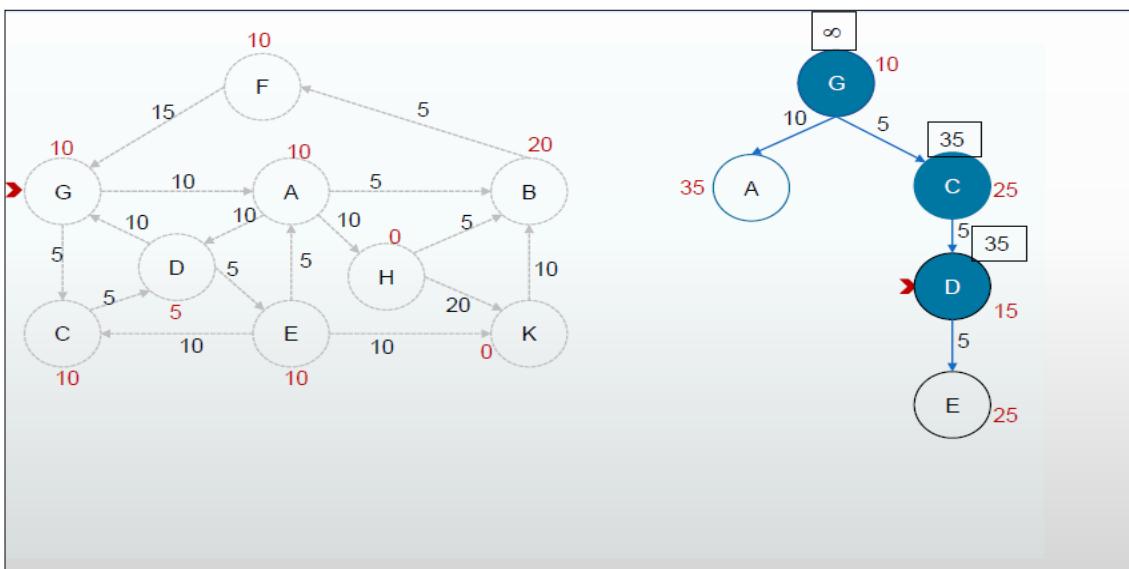
Exploration récursive par le meilleur d'abord (RBFS)



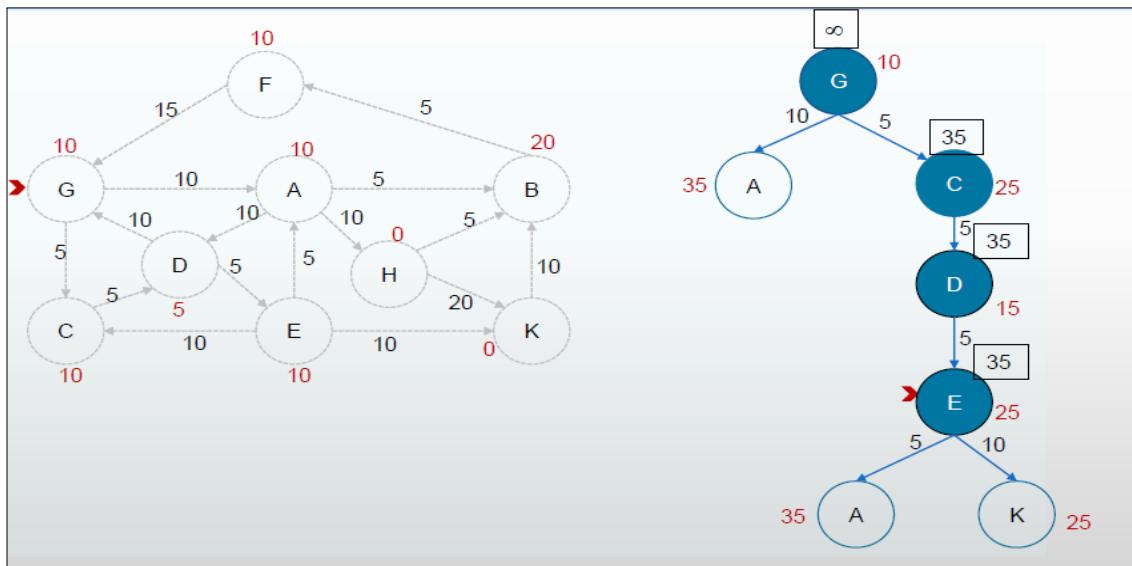
Exploration récursive par le meilleur d'abord (RBFS)



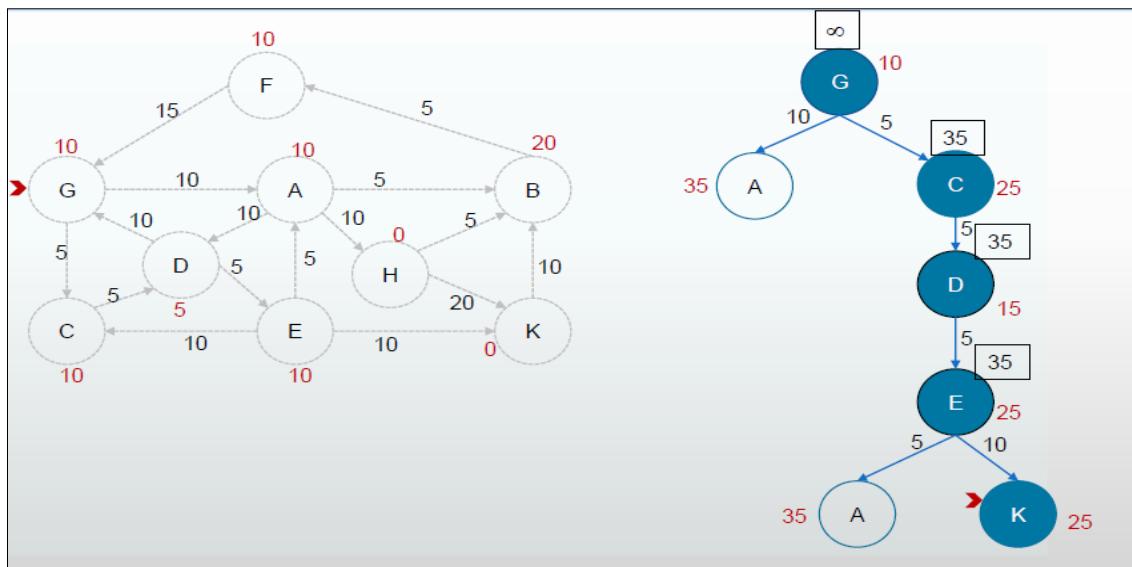
Exploration récursive par le meilleur d'abord (RBFS)



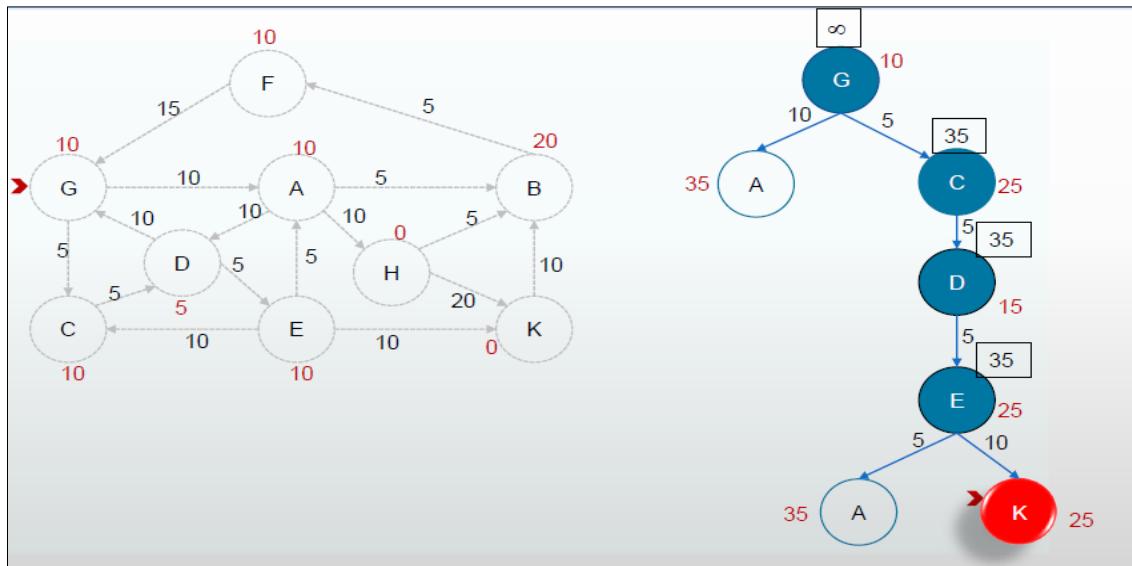
Exploration récursive par le meilleur d'abord (RBFS)



Exploration récursive par le meilleur d'abord (RBFS)



Exploration récursive par le meilleur d'abord (RBFS)



RBFS: performance

- Un peu plus efficace que IDA*; mais toujours génération excessive de nœuds.
- **Optimalité :** Optimale si $h(n)$ est admissible.
- **Complexité en espace linéaire** : $O(bd)$.
- **Complexité en temps** : dépend de l'exactitude de la fonction heuristique et de la fréquence des changements de chemins.
- Mais comme IDA* il sous-utilise l'espace mémoire.

RBFS: performance

- Un peu plus efficace que IDA*; mais toujours génération excessive de nœuds.
- **Optimalité :** Optimale si $h(n)$ est admissible.
- **Complexité en espace linéaire :** $O(bd)$.
- **Complexité en temps :** dépend de l'exactitude de la fonction heuristique et de la fréquence des changements de chemins.
- Mais comme IDA* il sous-utilise l'espace mémoire.

Exploration SMA* (Simple Memory-Bounded A*)

SMA* (Simplified Memory-Bounded A*) est une variante de l'algorithme A* qui est conçue pour fonctionner dans des environnements où la mémoire est limitée. Contrairement à A* qui conserve tous les nœuds dans la mémoire, SMA* permet une recherche optimale tout en limitant la quantité de mémoire utilisée.

- Meilleure utilisation de la mémoire disponible.
- Pareil que A* sauf qu'il développe la meilleure feuille **jusqu'à ce que la mémoire soit pleine**.
- **Pour libérer de l'espace, il supprime le nœud feuille le moins favorable.**
- Mémorise la valeur du nœud oublié au niveau du parent, il pourra alors régénérer ce sous-arbre s'il redeviendra le meilleur.

Exploration SMA* (Simple Memory-Bounded A*)

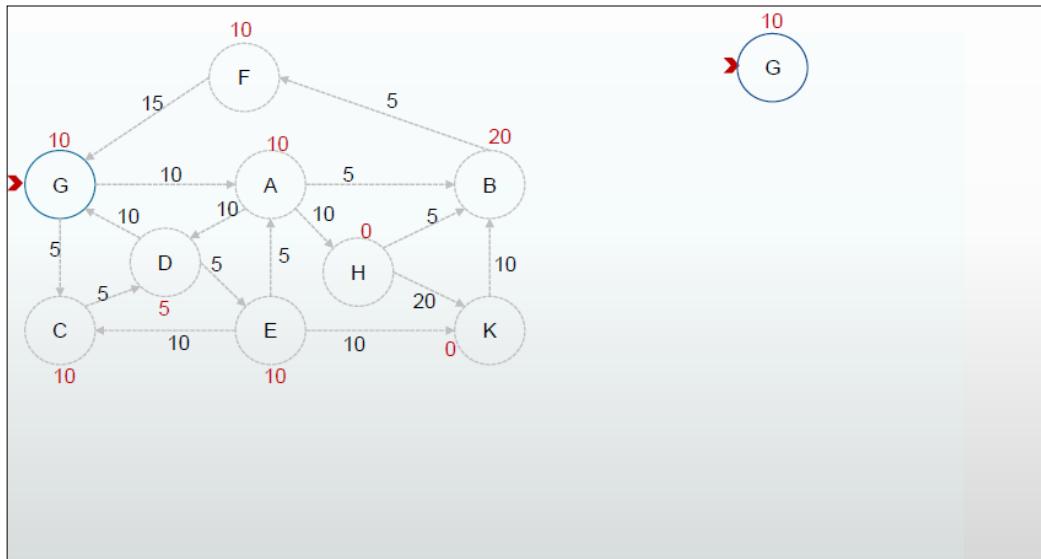
- Exactement comme A*, mais avec une limite sur la mémoire.
- S'il doit développer un nœud et que la mémoire est pleine, il enlève le plus mauvais nœud et comme RBFS, il enregistre au niveau du père la valeur du meilleur chemin.

Principe de SMA*:

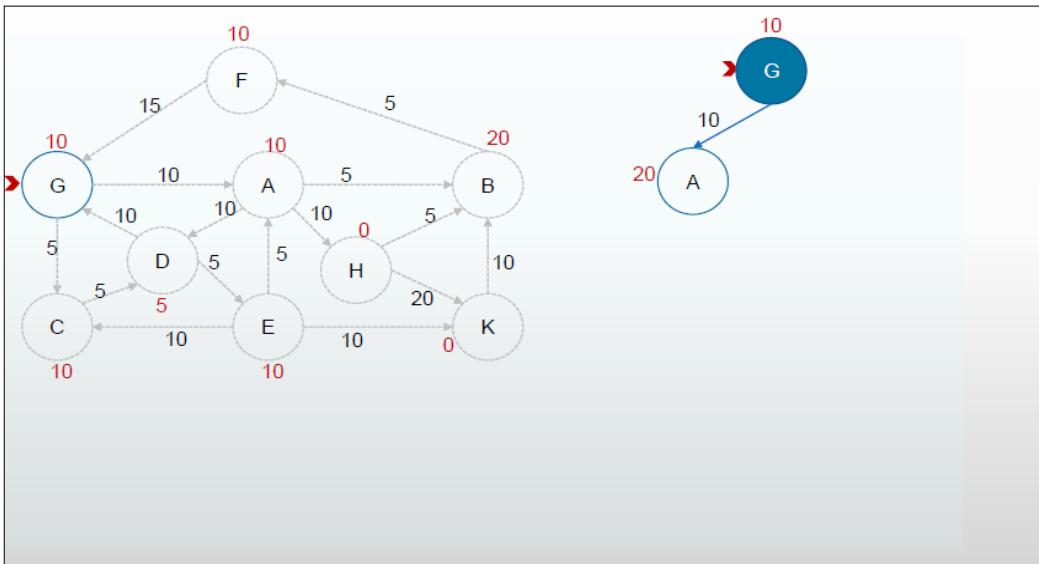
Il fonctionne comme suit :

1. **Exploration** : SMA* explore l'arbre de recherche comme A*, en utilisant la fonction d'évaluation $f(n)=g(n)+h(n)$
2. **Mémoire limitée** : SMA* maintient une **mémoire fixe** pour stocker les nœuds. Lorsque la mémoire est pleine, il **oublie** les nœuds les moins prometteurs (ceux avec la plus grande valeur $f(n)$).
3. **Récupération** : Si un nœud oublié devient nécessaire, SMA* le régénère en explorant à nouveau le chemin correspondant.

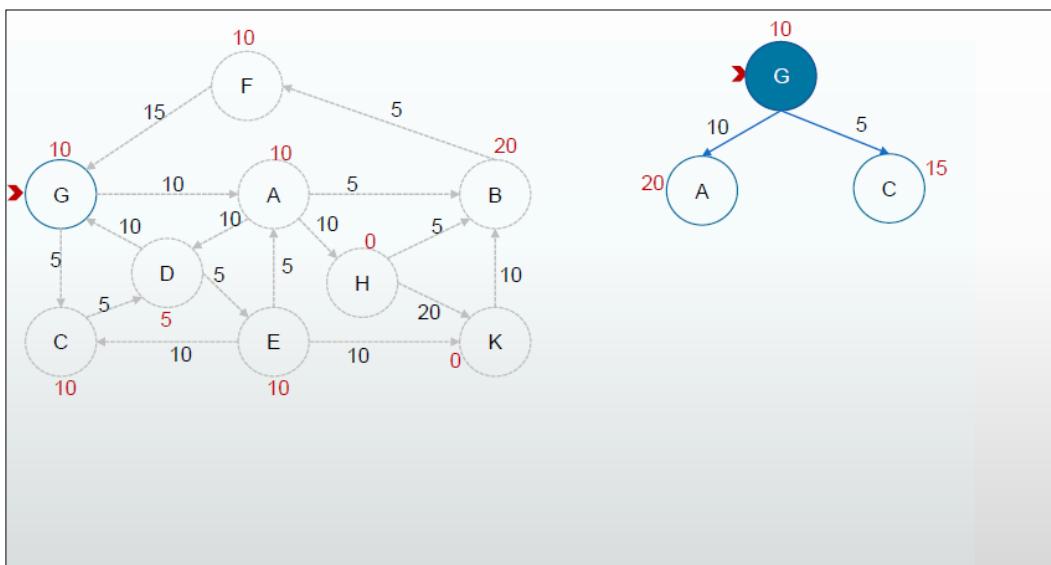
SMA*: 6 nœuds



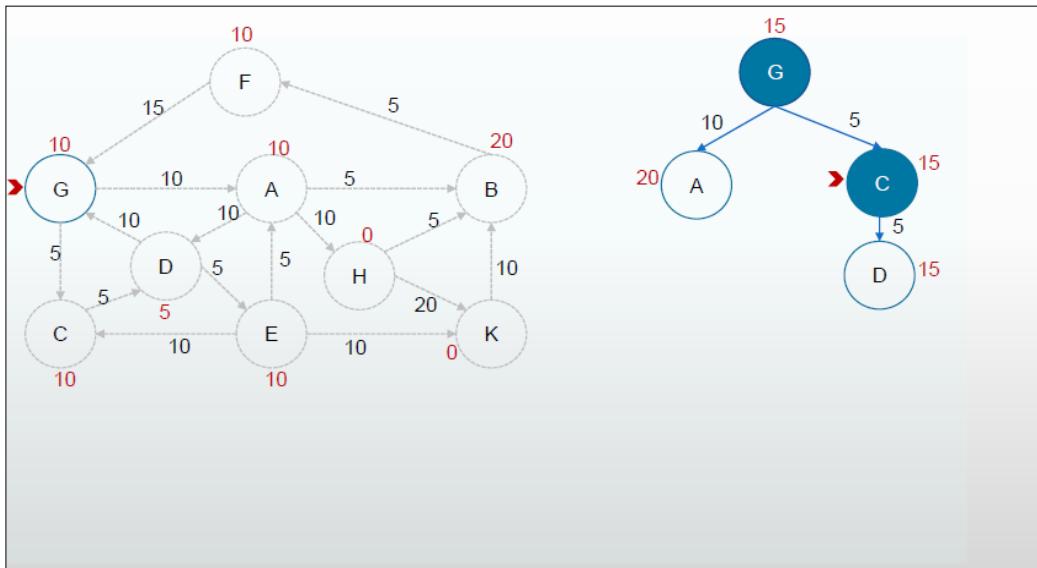
SMA*: 6 nœuds



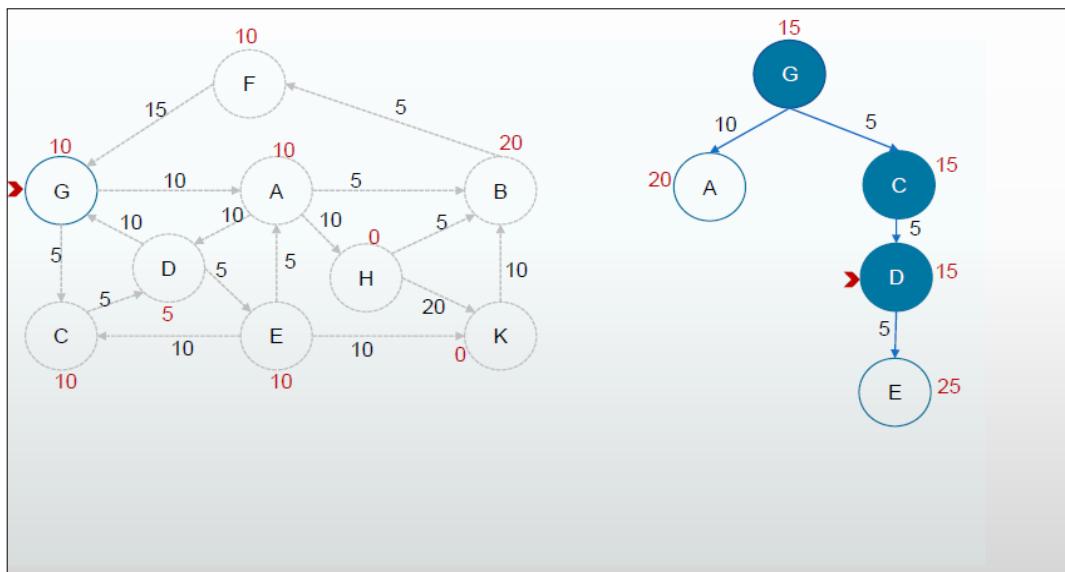
SMA*: 6 nœuds



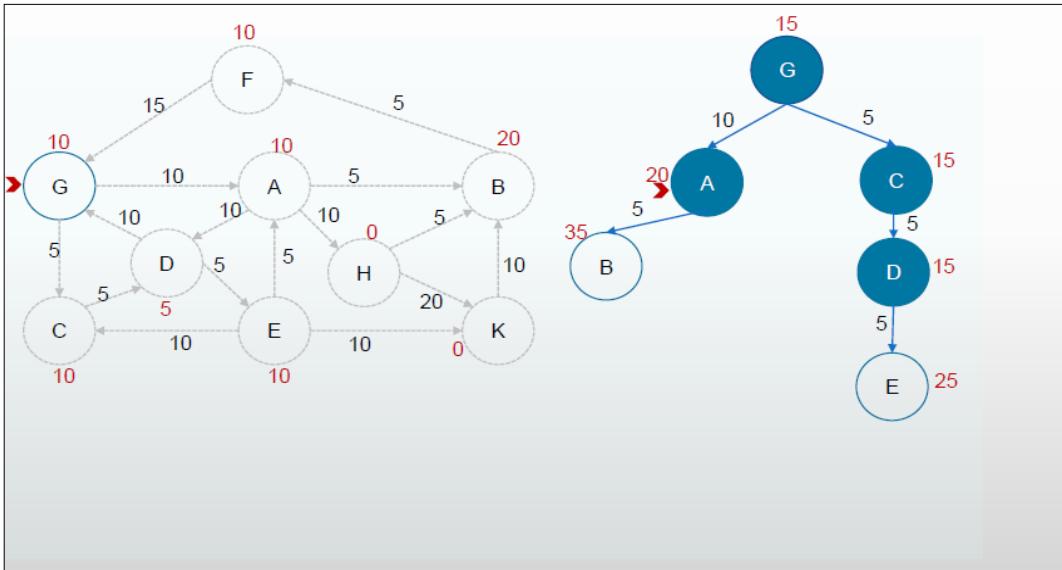
SMA*: 6 nœuds



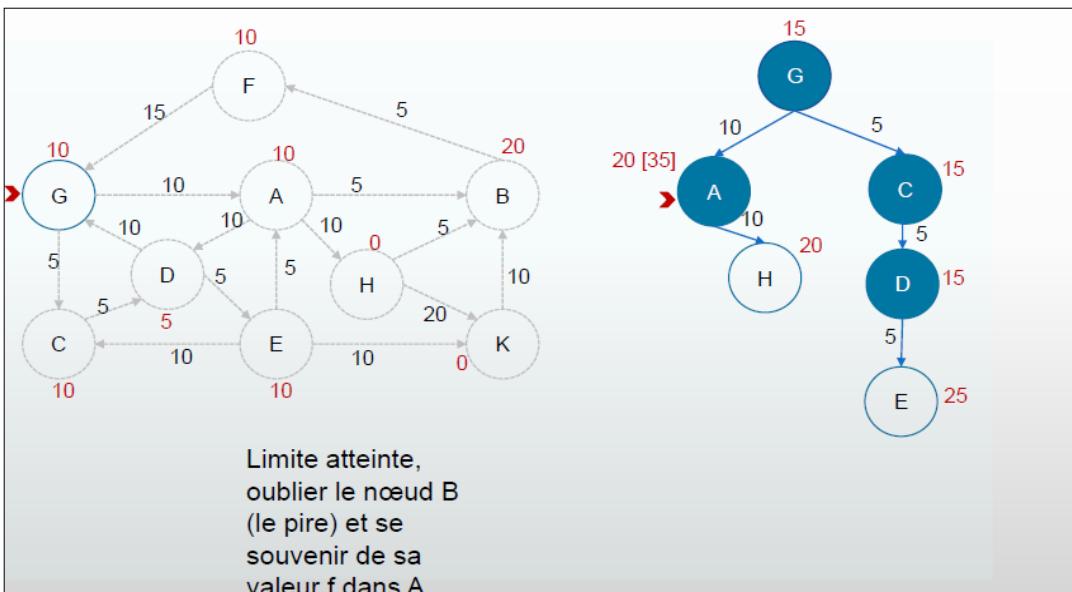
SMA*: 6 nœuds



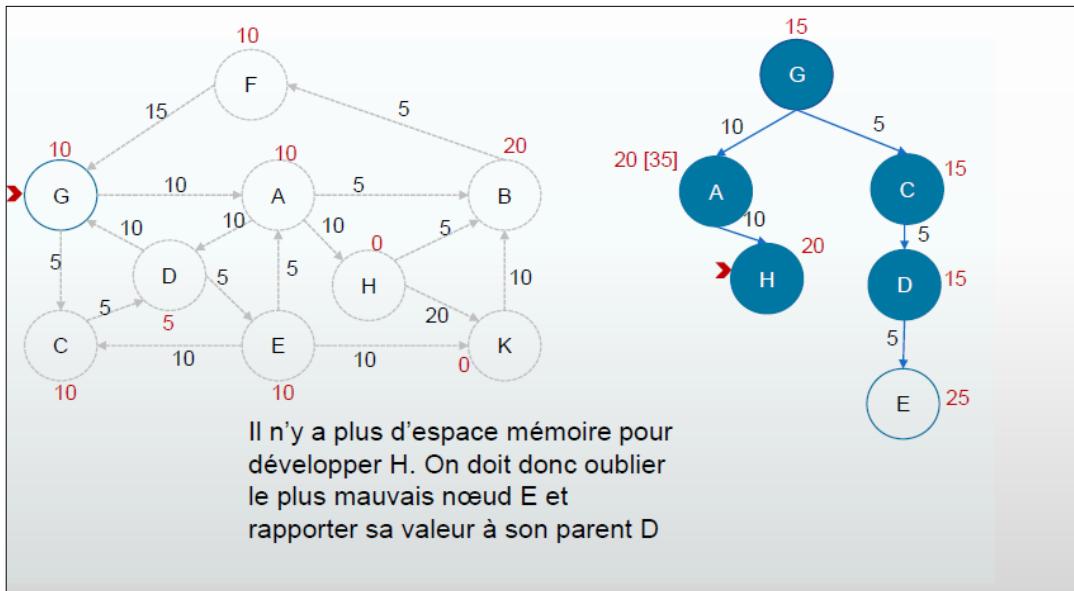
SMA*: 6 noeuds



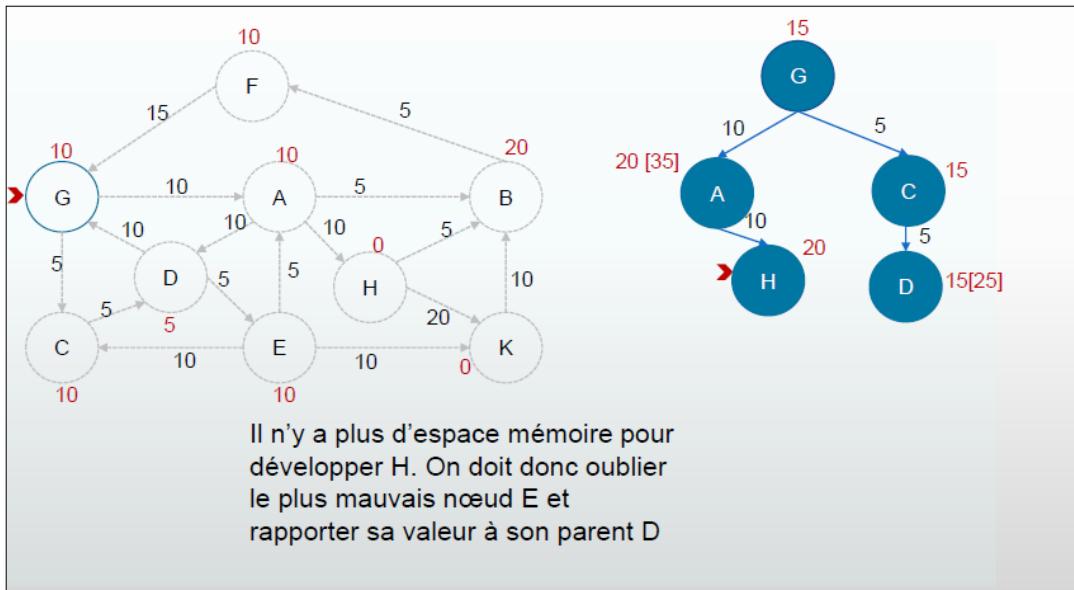
SMA*: 6 nœuds



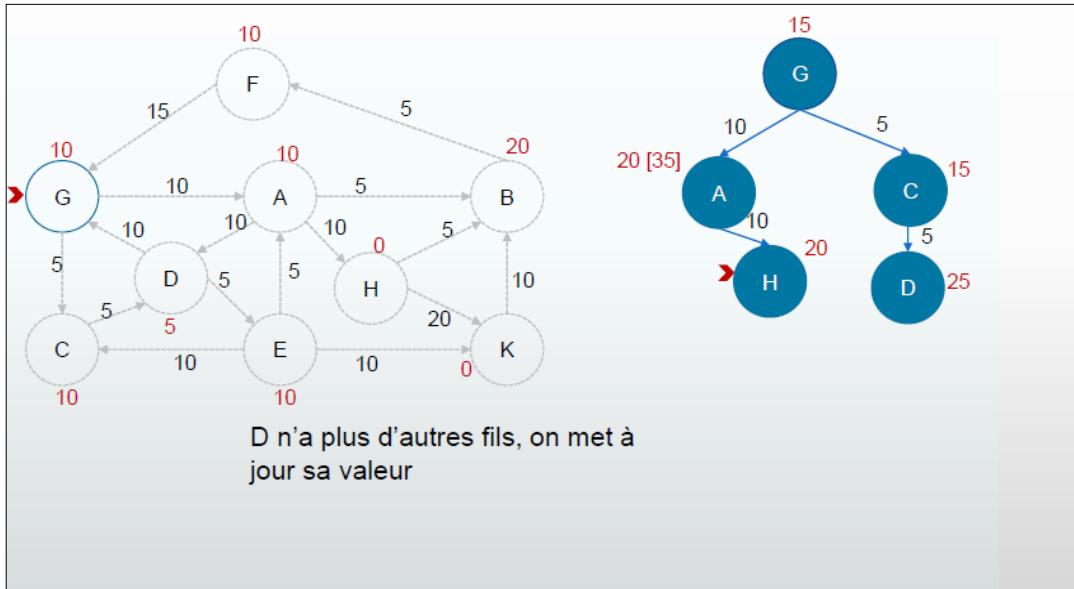
SMA*: 6 nœuds



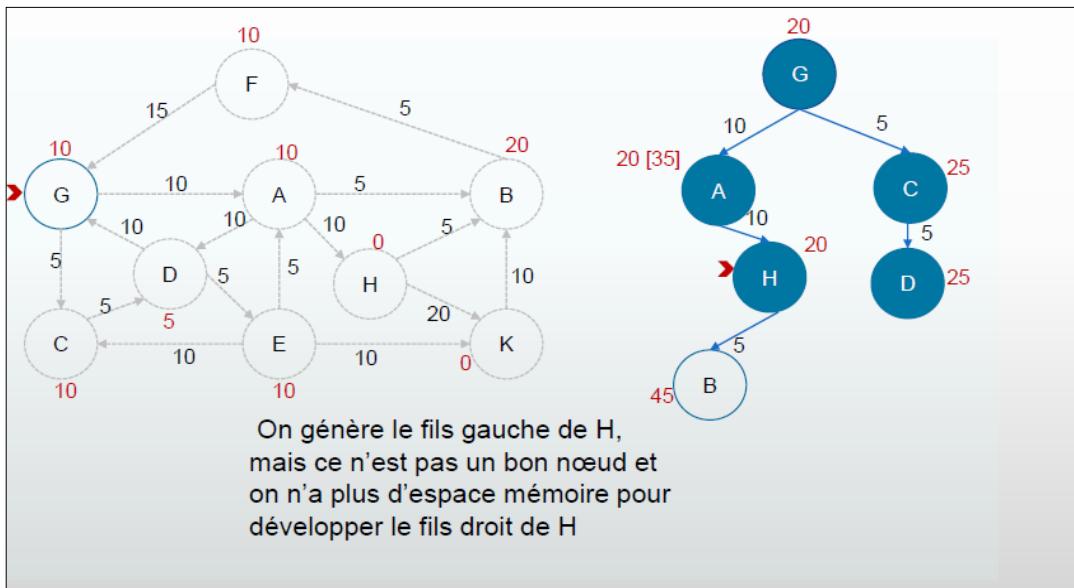
SMA*: 6 nœuds



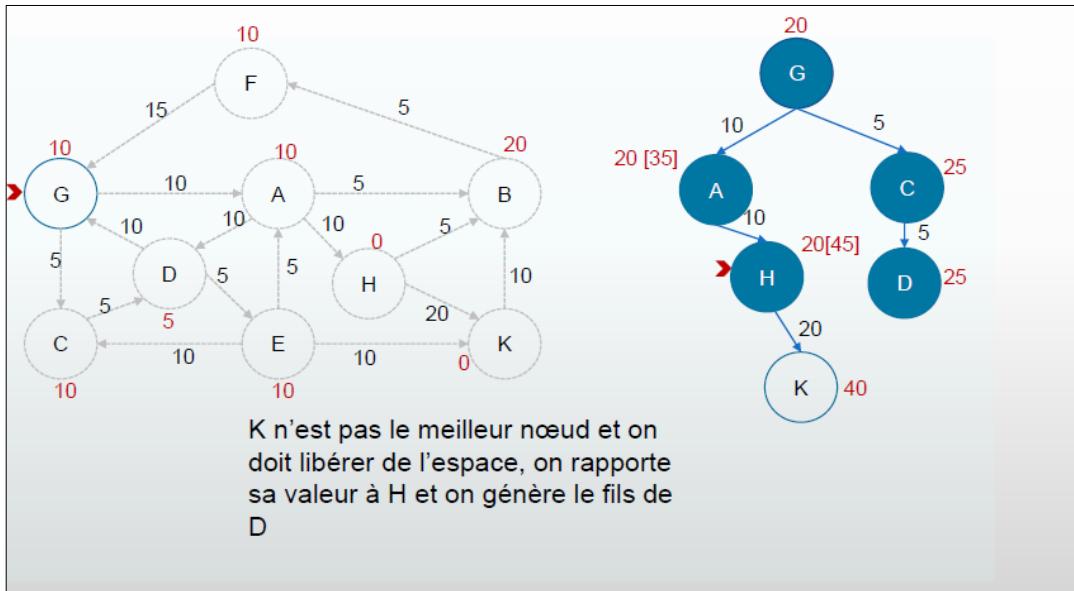
SMA*: 6 noeuds



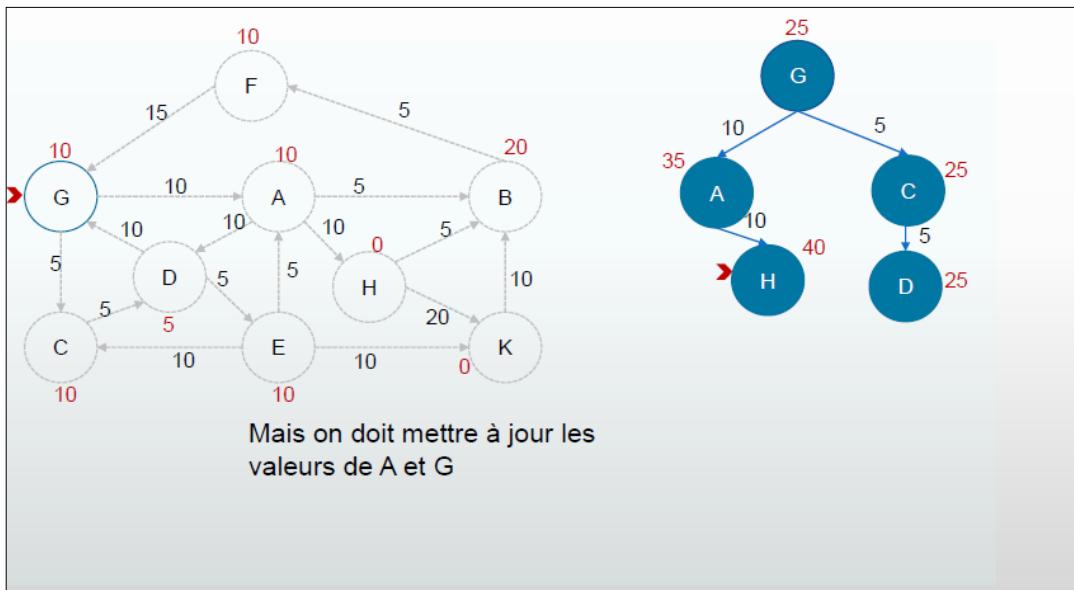
SMA*: 6 nœuds



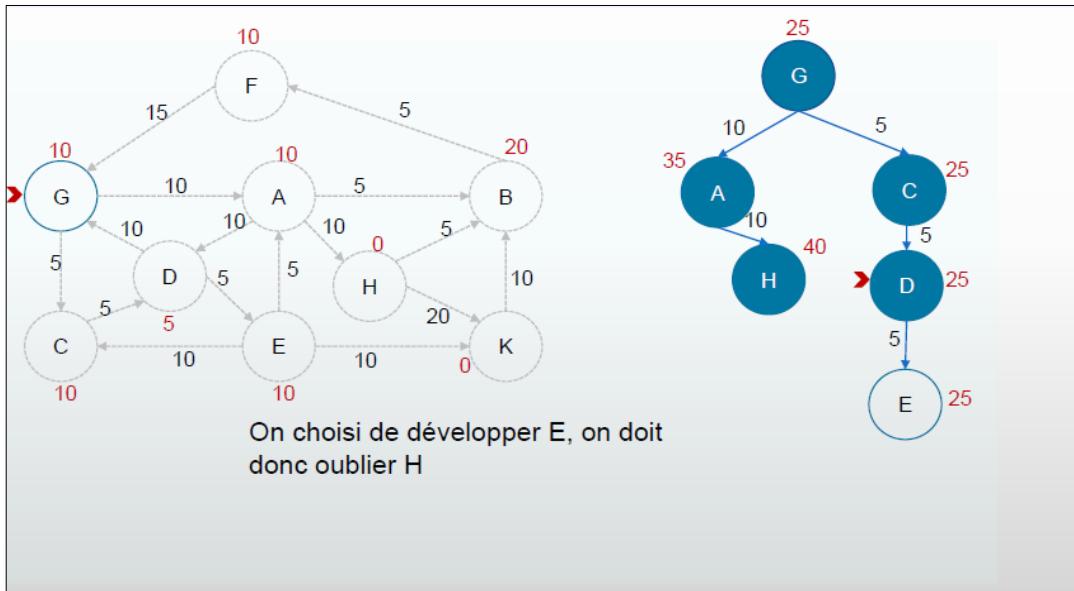
SMA*: 6 noeuds



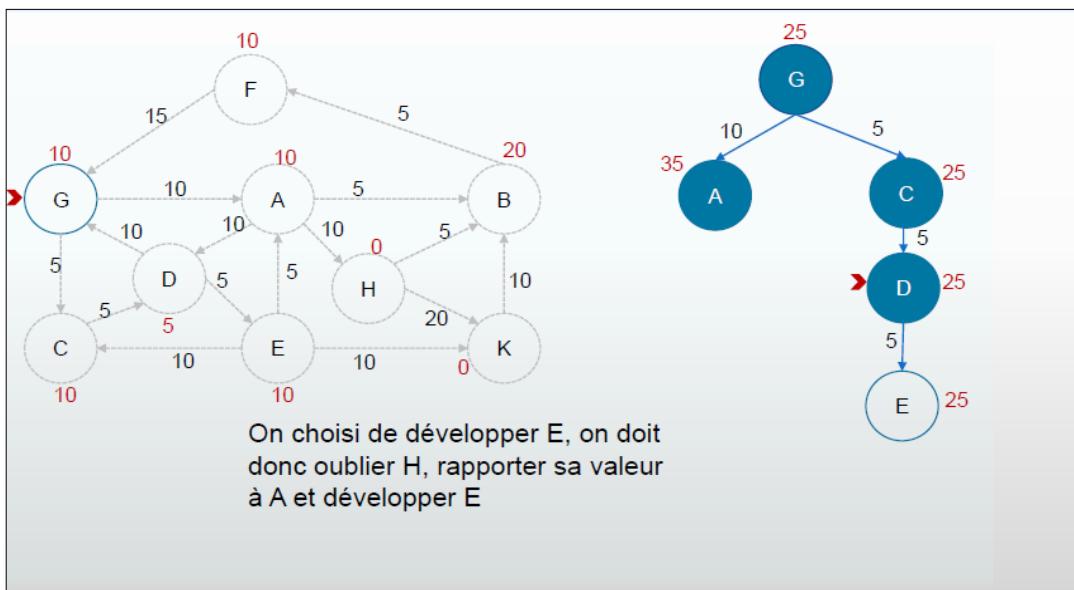
SMA*: 6 noeuds



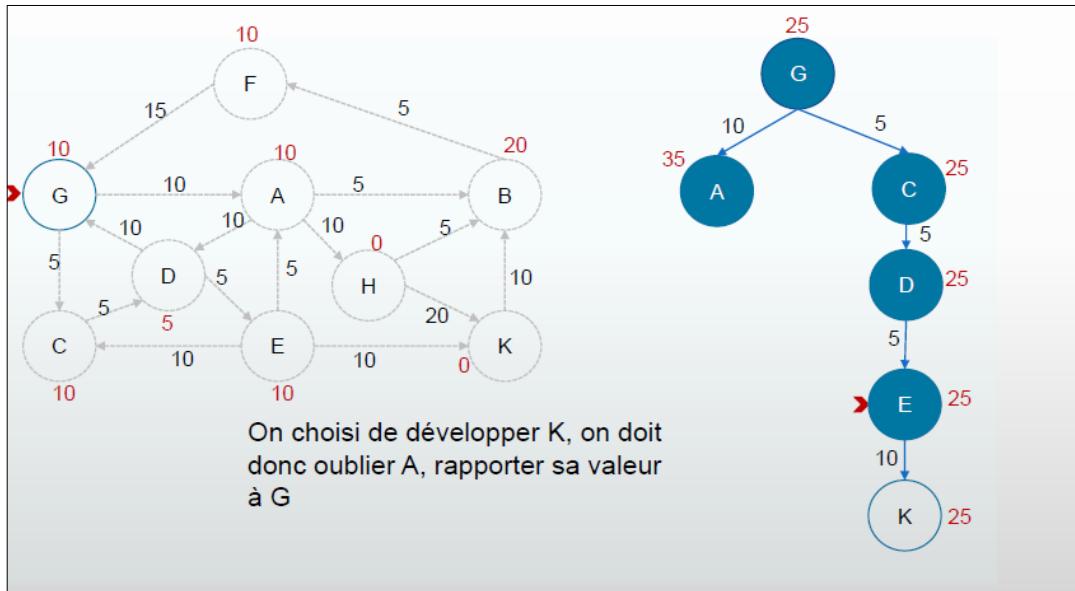
SMA*: 6 noeuds



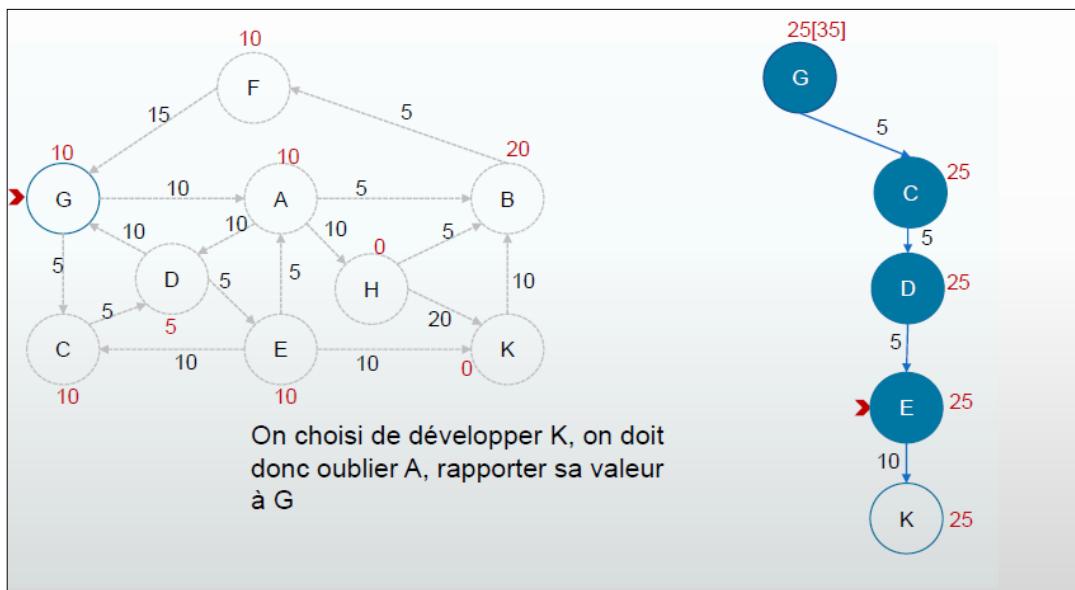
SMA*: 6 noeuds



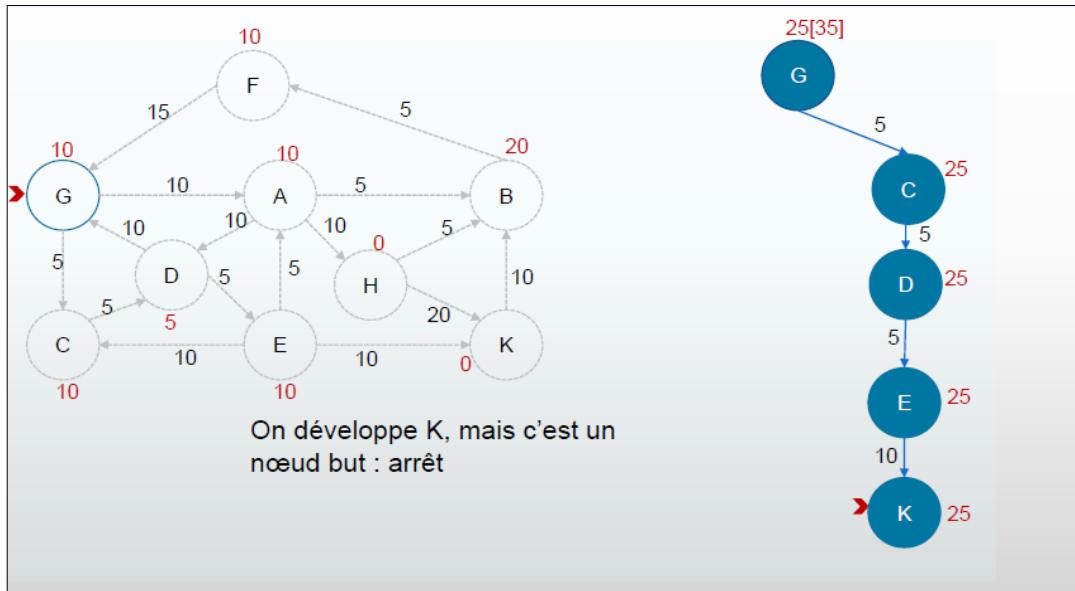
SMA*: 6 noeuds



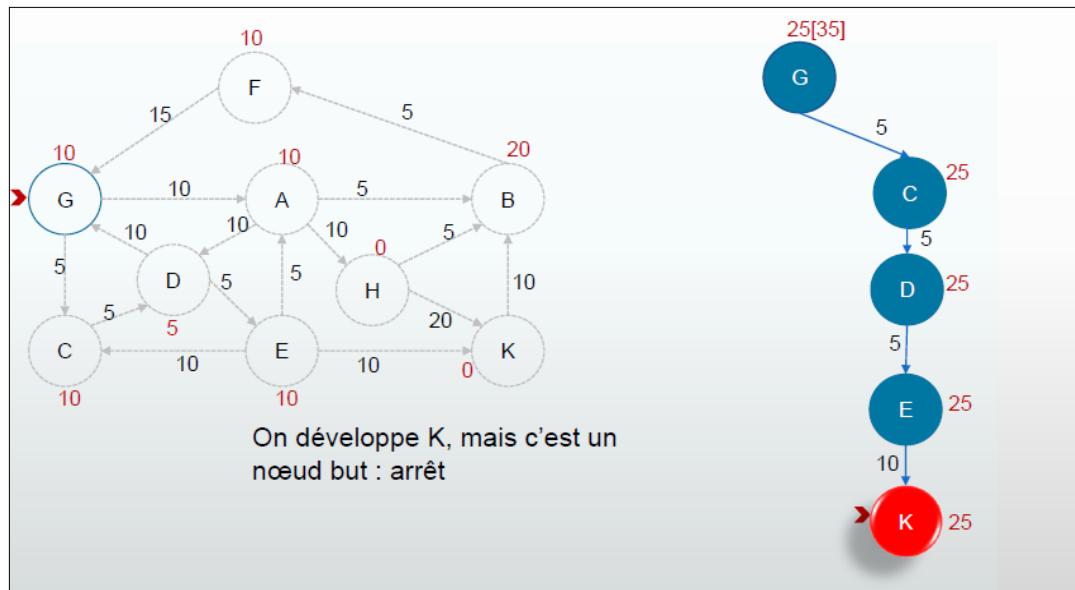
SMA*: 6 noeuds



SMA*: 6 noeuds



SMA*: 6 noeuds



SMA*: Performance

- **Complétude :** oui s'il existe une solution accessible ($d < \infty$ à la capacité de stockage).
- **Optimalité :** oui si une solution optimale est accessible (dépend de h)

Stratégies

A* :

1. Utilise une combinaison de coût réel $g(n)$ et d'heuristique $h(n)$ pour guider la recherche.
2. Nécessite une mémoire importante pour stocker tous les noeuds explorés.

IDA* :

1. Répète des itérations **de recherche en profondeur avec une limite croissante sur $f(n)$** .
2. Évite la consommation mémoire élevée de A* en ne stockant qu'un chemin à la fois.

RBFS :

1. Explore **récursivement** les noeuds en gardant une trace des meilleures alternatives.
2. Économise de la mémoire en ne stockant qu'un sous-arbre à la fois.

SMA* :

1. Limite explicitement l'utilisation de la mémoire en **supprimant les nœuds les moins prometteurs**.
2. Adapté aux environnements où la mémoire est une ressource critique.

Stratégies: comparaison

	A*	IDA*	RBFS	SMA*
Principe	Utilise une file de priorité (open list) pour explorer les noeuds en fonction de $f(n)=g(n)+h(n)$	Répète des itérations de recherche en profondeur limitée avec une limite croissante sur $f(n)$	Explore récursivement les noeuds en gardant une trace des meilleures alternatives.	Utilise une mémoire limitée pour explorer les noeuds, supprimant les moins prometteurs
Stratégie de recherche	Exploration en largeur (file de priorité sur $f(n)$)	Approfondissement progressif avec seuil sur $f(n)$	Approfondissement récursif en profondeur avec backtracking adaptatif	Similaire à A* mais supprime des noeuds pour respecter une contrainte mémoire
Gestion des successeurs	Stocke tous les noeuds dans une file de priorité	Ré-explore plusieurs fois certains chemins à cause de l'itération	N'exploré qu'une seule branche à la fois et mémorise la meilleure alternative	Supprime les moins prometteurs lorsque la mémoire est saturée

Stratégies: comparaison

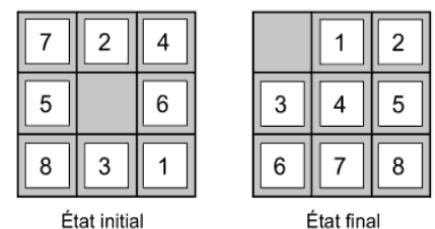
	A*	IDA*	RBFS	SMA*
Complétude	Oui (si $h(n)$ est admissible)	Oui (si $h(n)$ est admissible)	Oui (si $h(n)$ est admissible)	Oui (si la mémoire est suffisante)
Optimalité	Oui (si $h(n)$ est admissible)	Oui (si $h(n)$ est admissible)	Oui (si $h(n)$ est admissible)	Oui (si la mémoire est suffisante et $h(n)$ admissible)
Complexité en temps	Exponentielle, $O(b^d)$	Exponentielle, mais meilleure que A* en mémoire	Exponentielle, dépend de la qualité de l'heuristique	Exponentielle, mais adaptée aux contraintes mémoire
Complexité spatiale	Exponentielle, $O(b^{d^d})$ (garde tous les noeuds en mémoire)	Linéaire, $O(bd)$ (moins gourmand en mémoire que A*)	Linéaire, $O(bd)$ (évite de stocker toute la frontière)	Bornée à la mémoire disponible
Avantage	Optimal et complet. Efficace pour des graphes de petite à moyenne taille.	Moins gourmand en mémoire que A*. Idéal pour les problèmes avec un espace d'état large.	Économiseur de mémoire par rapport à A*. Gère bien les arbres de recherche larges.	Gestion explicite de la mémoire. Adapté aux problèmes avec une mémoire limitée.

Stratégies de Recherche

- **Recherche non-informée**
 - ✓ Exploration en largeur d'abord
 - ✓ Exploration à coût uniforme
 - ✓ Exploration en profondeur d'abord
 - ✓ Exploration en profondeur avec libération de mémoire
 - ✓ Exploration en profondeur limitée
 - ✓ Exploration itérative en profondeur
 - ✓ Exploration bidirectionnelle
- **Recherche informée**
 - Meilleur d'abord (Best-first)
 - ✓ Meilleur d'abord gloutonne (Greedy best-first)
 - ✓ A*
 - Algorithmes à mémoire limitée : IDA*, RDFS et SMA*
- **Fonctions Heuristiques**
- **Algorithmes de recherche local et optimisation**
 - ✓ Escalade (Hill-climbing)
 - ✓ Recuit simulé (Simulated annealing)
 - ✓ Recherche locale en faisceau (local beam)
 - ✓ Algorithmes génétiques

Jeu de taquin

- Coût moyen pour la solution : **22 étapes**
 ==> Le **coût moyen** vient d'expériences et d'analyses de recherche optimale sur cet espace d'états. Il représente le **nombre moyen de déplacements pour atteindre la solution depuis un état initial aléatoire.**



- Facteur de branchement moyen : $b = 3$
- Pour une exploration d'arbre exhaustive à une profondeur 22, examine $3^{22} \approx 3.1 \times 10^{10}$ états (sans heuristique)

Pour une exploration en graphes : une diminution par un facteur de 170000 états (avec mémorisation des états déjà visités) :

- ✓ On stocke les états déjà explorés pour éviter de les revisiter inutilement.
- ✓ Chaque état est unique dans l'espace de recherche, ce qui élimine les duplications.

Jeu de taquin

Alors que dire avec Problème du taquin 15

- Le taquin 15 (4×4) est un cas plus difficile, où une solution optimale prend souvent 30 à 50 mouvements.



→ Il faut penser à utiliser une bonne fonction heuristique.

Heuristique pour le jeu de taquin

- Dans le jeu de taquin, une heuristique est une estimation du coût restant pour atteindre l'état but à partir d'un état donné. Deux heuristiques admissibles pour le jeu de taquin :
- h_1 =nombre de pièces mal placées : $h_1=8$ pour l'état initial.

Explication: Cette heuristique est admissible, car chaque déplacement ne peut corriger qu'une seule pièce à la fois (ou aucune si le déplacement ne concerne pas une pièce mal placée). Ainsi, h_1 ne surestime jamais le nombre de déplacements nécessaires.

- h_2 =somme des distances entre la position actuelle de chaque pièce et sa position but. (distance de Manhattan ou distance city-block) : ($h_2=3+1+2+2+2+3+3+2=18$ pour l'état initial).

Explication: Cette heuristique est admissible, car chaque déplacement ne peut réduire la distance totale que d'au plus 1 (en déplaçant une pièce vers sa position but). Ainsi, h_2 ne surestime jamais le coût réel.

Dominance

Notion de dominance

On dit qu'une heuristique h_2 **domine** une autre heuristique h_1 si elle est toujours plus informative qu'elle, c'est-à-dire si pour tout nœud n , on a :

$$h^*(n) \geq h_2(n) \geq h_1(n)$$

où $h^*(n)$ est le coût réel minimal pour atteindre l'état final depuis n .

Cela signifie que h_2 donne une meilleure approximation du coût réel que h_1

Que peut-on dire des heuristiques de l'exemple précédent ?

Production d'heuristique admissible

- ❖ **Problèmes relaxés:** est une version simplifiée d'un problème original, où certaines contraintes sont supprimées. L'idée est que la résolution du problème relaxé est plus facile, et la solution obtenue peut servir d'heuristique admissible pour le problème original.
- ❖ **Base de données de motifs:** est une technique de précalcul qui stocke les coûts optimaux pour atteindre l'état but à partir de différents sous-ensembles d'états (ou "motifs"). Ces coûts sont ensuite utilisés comme heuristiques pendant la recherche.
- **Fonctionnement :**
 - ✓ On divise le problème en sous-problèmes plus petits.
 - ✓ Pour chaque sous-problème, on calcule et stocke le coût optimal pour atteindre l'état but.
 - ✓ Pendant la recherche, on combine les coûts des sous-problèmes pour obtenir une heuristique admissible.
- ❖ **Apprentissage:** peut être utilisé pour améliorer les heuristiques en apprenant à partir de données ou d'expériences passées. Cela peut se faire via des techniques d'apprentissage automatique (machine learning) ou par renforcement (reinforcement learning).

Production d'heuristique pour les problèmes relaxés

l'heuristique est une solution exacte pour le problème simplifié.

Exemple du jeu de Taquin :

- A. Problème réel :** une pièce peut passer de la case A à la case B si A est adjacent à B et si B est vide

B. Problèmes relaxés :

- **Relaxation 1 :** Une pièce peut se déplacer de A à B si A est adjacent à B (peu importe si B est vide ou non): cela correspond à l'heuristique h_2 (somme des distances de Manhattan).
- **Relaxation 2 :** Une pièce peut se déplacer de A à B si B est vide (peu importe si A et B sont adjacents): cela correspond à une heuristique moins restrictive.
- **Relaxation 3 :** Une pièce peut se déplacer de A à B sans aucune contrainte.
 - ✓ Cela correspond à l'heuristique h_1 (nombre de pièces mal placées).

Le programme **OBSERVER** est un programme qui automatise la génération d'heuristiques en se basant sur des problèmes relaxés. Il explore différentes relaxations possibles et sélectionne celles qui fournissent les heuristiques les plus informatives.

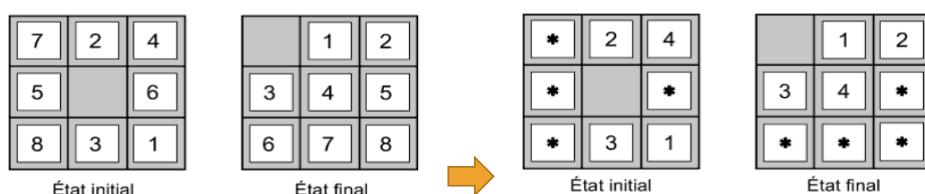
Production d'heuristique à partir des sous problèmes

Cette technique consiste à décomposer un problème complexe en sous-problèmes plus simples, à résoudre ces sous-problèmes de manière exacte, et à utiliser leurs solutions pour guider la recherche dans le problème global:

- ✓ L'heuristique est obtenue à partir du coût de la solution d'un sous-problème
- ✓ Stocker, dans une base de données de motifs, des coût exacts de solutions qui sont des instances de sous-problèmes

Exemple dans le jeu de taquin :

- ✓ Pour un groupe de 4 pièces, on calcule le nombre minimal de déplacements nécessaires pour amener ces pièces à leur position but.
- ✓ Ce coût est stocké dans une base de données de motifs.



Production d'heuristique à partir des sous problèmes

- La base de données est construite par **exploration en arrière** à partir du but et en enregistrant le coût de chaque nouveau motif rencontré.
 - ==> L'exploration en arrière consiste à partir de l'état but et à explorer les états précédents de manière récursive, en enregistrant le coût pour atteindre l'état but à partir de chaque état rencontré.
 - ==> L'exploration s'arrête lorsque tous les états possibles du sous-problème ont été visités.
 - ==> Structure de la base de données (key, value) , ou la clé c'est un état du sous-problème (par exemple, une configuration spécifique de 4 pièces). Et la valeur est le coût minimal pour atteindre l'état but à partir de cet état.

Avantages

- Pour le taquin à 15 pièces, cette heuristique réduit le nombre de nœuds générés par un facteur de 1000 par rapport à l'heuristique de la distance de Manhattan. Cela s'explique par le fait que l'heuristique basée sur les sous-problèmes est beaucoup plus informative : elle prend en compte les interactions locales entre les pièces, ce que la distance de Manhattan ne fait pas.

Apprentissage d'heuristique

- Développer des heuristiques à partir du coût des solutions optimales
- Appliquer **des algorithmes d'apprentissage** afin de construire une fonction $h(n)$ qui prédit les coûts des solutions pour d'autres états qui se produisent pendant l'exploration



- L'objectif est de construire une fonction heuristique $h(n)$ qui approxime le coût réel pour atteindre l'état but à partir d'un état n .

Étapes :

1. **Collecte de données** : Générer un ensemble de données en résolvant des instances du problème (ou des sous-problèmes) de manière optimale.
2. **Apprentissage** : Entraîner un modèle d'apprentissage automatique pour prédire le coût des solutions à partir des caractéristiques des états.
3. **Utilisation** : Utiliser le modèle appris comme heuristique pendant la recherche.

==> Des techniques basées sur les réseaux de neurones, d'arbres de décision et d'autres méthodes d'apprentissage sont adoptées.

Stratégies de recherche

- **Recherche non-informée**
 - ✓ Exploration en largeur d'abord
 - ✓ Exploration à coût uniforme
 - ✓ Exploration en profondeur d'abord
 - ✓ Exploration en profondeur avec libération de mémoire
 - ✓ Exploration en profondeur limitée
 - ✓ Exploration itérative en profondeur
 - ✓ Exploration bidirectionnelle
- **Recherche informée**
 - Meilleur d'abord (Best-first)
 - ✓ Meilleur d'abord gloutonne (Greedy best-first)
 - ✓ A*
 - Algorithmes à mémoire limitée : IDA*, RDFS et SMA*
- **Fonctions Heuristiques**
- **Algorithmes de recherche local et optimisation**
 - ✓ Escalade (Hill-climbing)
 - ✓ Recuit simulé (Simulated annealing)
 - ✓ Recherche locale en faisceau (local beam)
 - ✓ Algorithmes génétiques

Exploration locale

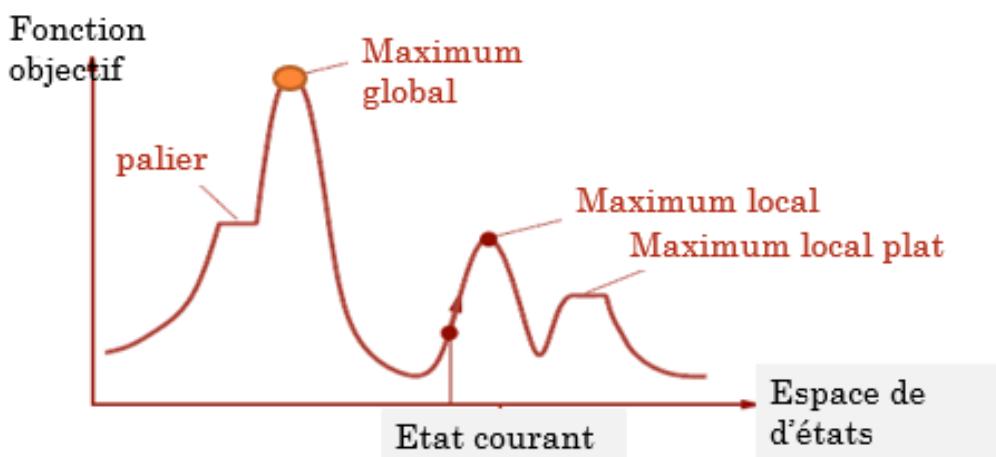
- Le chemin menant à la solution optimale n'a pas d'importance si on cherche uniquement une solution optimale ou quasi-optimale, et non la séquence exacte des étapes pour y parvenir. Ex. Jeu de n reines
- Conception de circuits intégrés, ordonnancement, optimisation de réseaux de télécommunication, routage (problèmes réels).

L'objectif est de trouver un état optimal ou une configuration finale qui satisfait les contraintes ou optimise les critères. Cela influence le choix des algorithmes de recherche et d'optimisation, privilégiant ceux qui se concentrent sur l'état final plutôt que sur la séquence d'actions

Exploration locale

- ✓ Elle opère en ne considérant qu'un seul noeud courant et non plusieurs chemins.
- ✓ **Voisinage:** Ne s'occupe que des voisins du noeud choisi.
- ✓ **Pas de mémorisation des chemins:** Les algorithmes de recherche locale ne mémorisent généralement pas les **chemins suivis** pour atteindre l'état courant.
- ✓ **Optimisation d'une fonction objectif:** Tentent de trouver la meilleure solution selon une fonction objectif (à minimiser ou à maximiser).
- ✓ **Faible consommation de mémoire:** Consomment peu de mémoire et trouvent souvent des solutions raisonnables dans des espaces d'états infinis.

Paysage de l'espace d'états pour une exploration locale



Paysage de l'espace d'états pour une exploration locale

- **Algorithmes de recherche local et optimisation**

- ✓ Escalade (Hill-climbing)
- ✓ Recuit simulé (Simulated annealing)
- ✓ Recherche locale en faisceau (local beam)
- ✓ Algorithmes génétiques

Paysage de l'espace d'états pour une exploration locale

But : « Atteindre le sommet de l'Everest dans un épais brouillard tout en souffrant d'amnésie ».

- ✓ "Atteindre le sommet de l'Everest" : Objectif d'optimisation (trouver la meilleure solution).
- ✓ "Dans un épais brouillard" : Manque d'information globale, on ne voit que les solutions voisines.
- ✓ "Tout en souffrant d'amnésie" : Pas de mémoire des états précédents, l'algorithme ne peut pas revenir en arrière.

```

fonction ESCALADE(problème) retourne un état qui est un maximum local
    courant  $\leftarrow$  CRÉER-NŒUD(problème.État-Initial)
    faire en boucle
        voisin  $\leftarrow$  un successeur de courant ayant la valeur la plus élevée
        si voisin.VALEUR  $\leq$  courant.VALEUR alors retourner courant.État
        courant  $\leftarrow$  voisin

```

Fonctionnement

Entrée :

- État initial.
- Fonction à optimiser (fonction objectif/coût noté VALUE).
- Fonction permettant de générer les états successeurs

Algorithme :

- Le nœud courant est initialisé à l'état initial.
- Itérativement, le nœud courant est comparé à ses successeurs immédiats.
 - ✓ Le meilleur voisin immédiat et ayant la plus grande/petite valeur (selon VALUE) que le nœud courant, devient le nœud courant.
 - ✓ Si un tel voisin n'existe pas, on arrête et on retourne le nœud courant comme solution

Exploration par escalade

- ✓ **Méthode itérative:** Une boucle qui se déplace vers les valeurs croissantes/décroissantes guidée par la fonction objectif.
- ✓ **Optimum local:** S'arrête quant elle atteint un pic avec aucun voisin avec une valeur plus élevée.
- ✓ **Mémoire limitée :** Pas de maintien d'un arbre d'exploration, uniquement l'état courant et la fonction objectif sont enregistrés.
- Uniquement les voisins immédiats sont observés.

Avantage :

- Faible consommation de mémoire : Hill-Climbing est très économique en mémoire, car il ne stocke pas d'informations supplémentaires.

Limite :

- Pas de retour en arrière : Comme il ne mémorise pas les états précédents, Hill-Climbing ne peut pas revenir en arrière pour explorer d'autres branches.

Illustration de hill-climbing

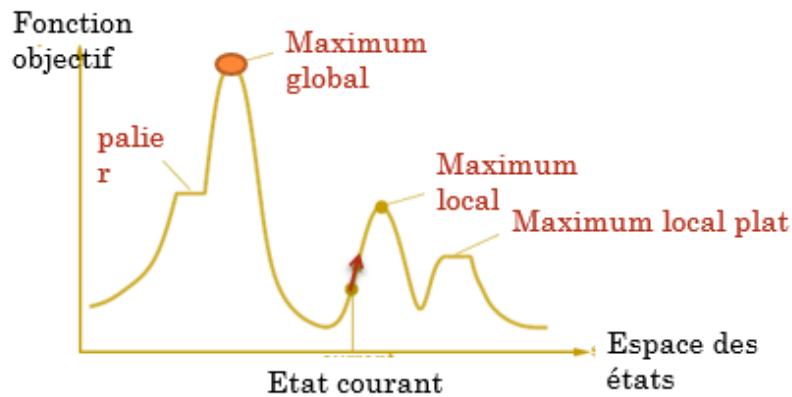


Illustration de hill-climbing

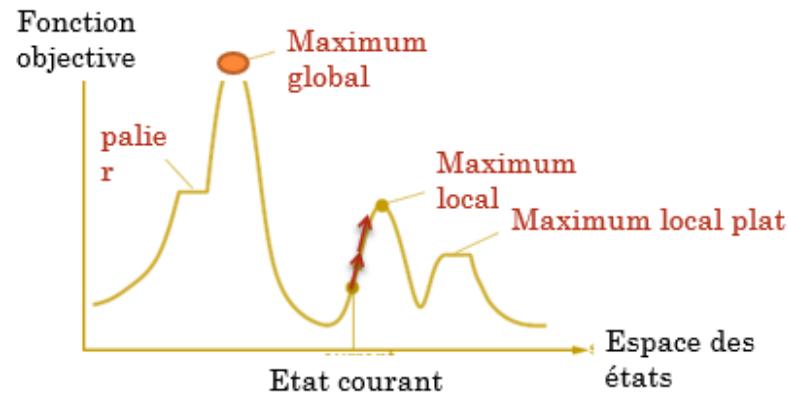


Illustration de hill-climbing

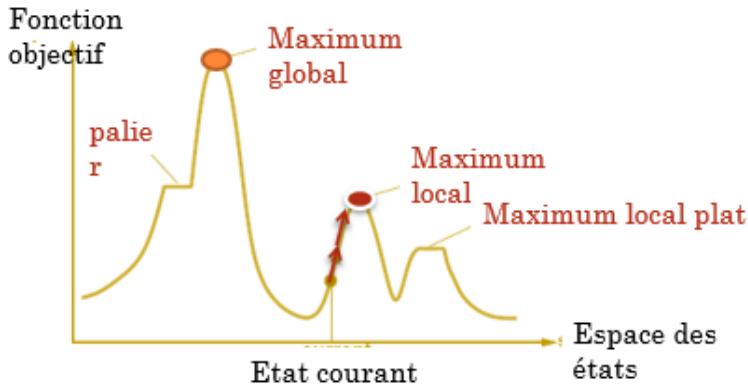
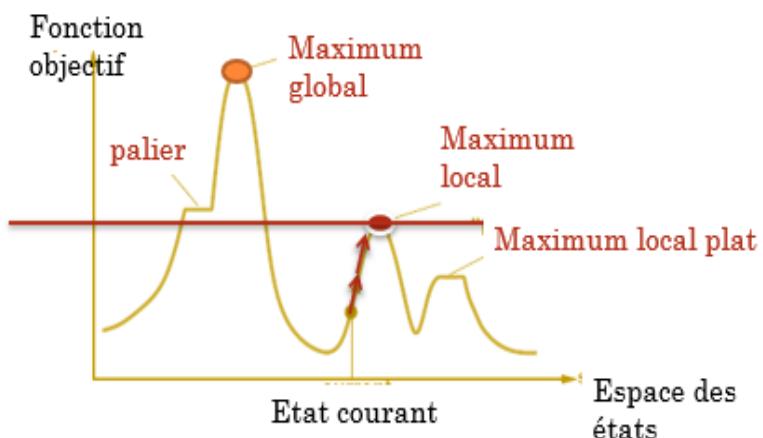


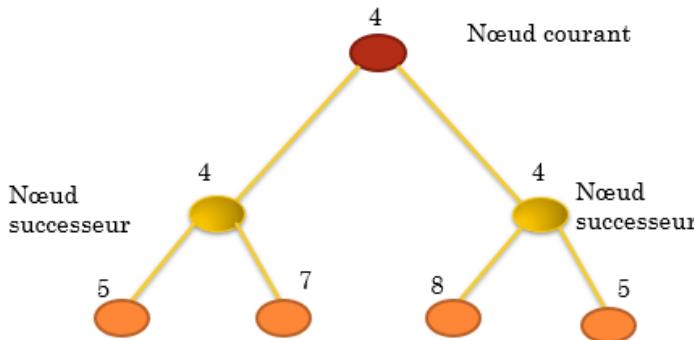
Illustration de hill-climbing



- L'algorithme *hill-climbing* risque d'être piégé dans des optimums locaux : s'il atteint un nœud dont ses voisins immédiats sont moins bons, il s'arrête !

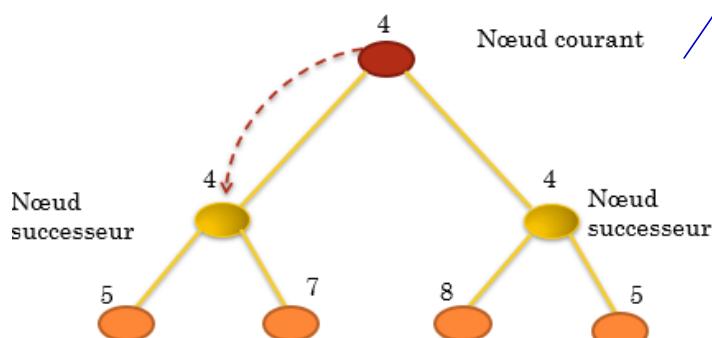
Cas d'une région plate dans l'espace de recherche

❖ Ici on cherche à minimiser la fonction objectif.



Minimum local plat

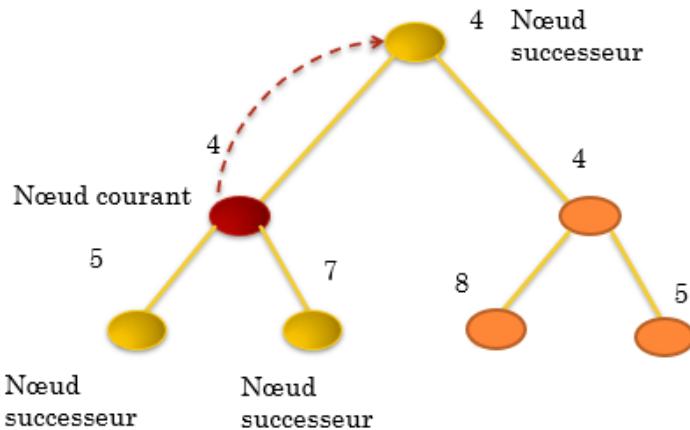
❖ Si on arrive à dépasser l'espace plat et on choisit un état successeur de façon aléatoire.



- **Minimum local** : Un état (ou point) dans l'espace de recherche où la fonction objectif a une valeur inférieure à celle de tous ses voisins immédiats.
- **Plat** : Une région de l'espace de recherche où la fonction objectif a une valeur constante ou quasi-constante (c'est-à-dire que les valeurs des états voisins sont très proches).
- **Un minimum local plat** est donc une région plate autour d'un minimum local, où la fonction objectif ne varie pas significativement.

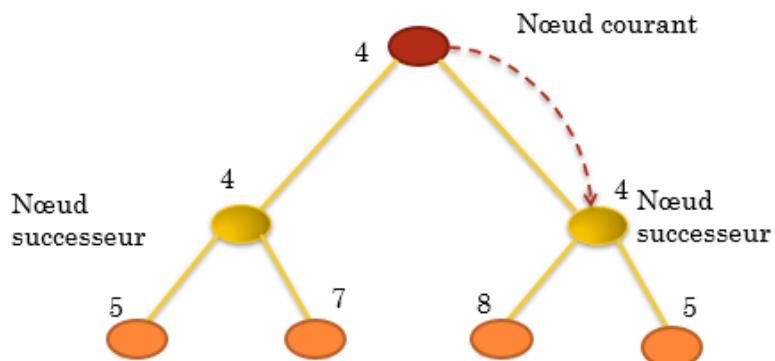
Minimum local plat

- ◆ Ici il va revenir à l'état initial puisque c'est le meilleur choix.



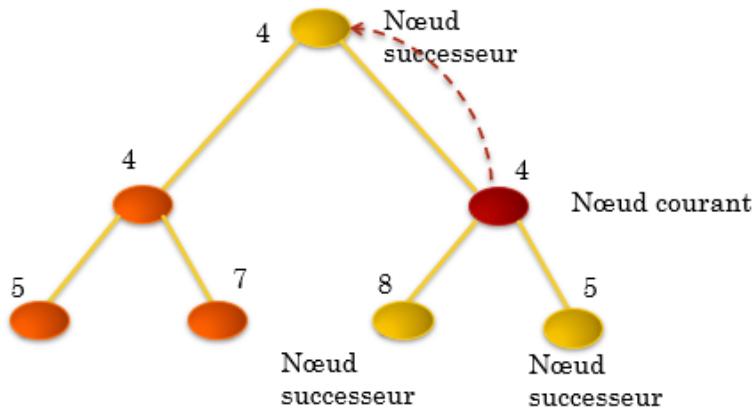
Minimum local plat

- ◆ Il peut ou bien boucler entre le noeud initial et son successeur gauche ou passer au successeur droit d'une façon aléatoire.



Minimum local plat

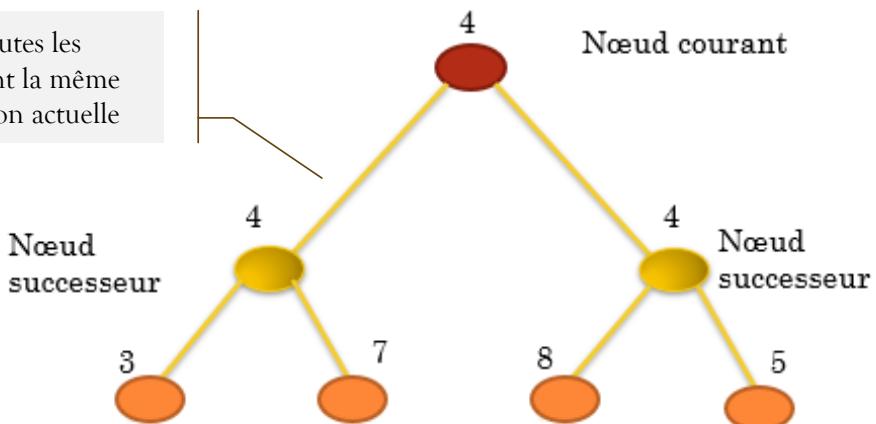
- Il va encore revenir à l'état initial puisque c'est le meilleur choix. Ainsi il bouclera à l'infini entre ses deux successeurs.



Plateau

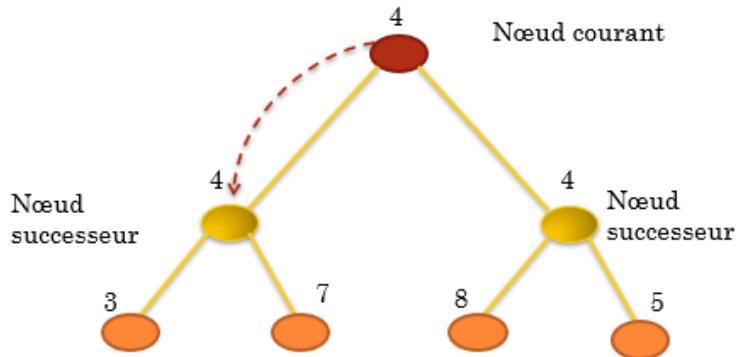
- On cherche à minimiser la fonction objectif.

Plateau plat : Toutes les solutions voisines ont la même qualité que la solution actuelle



Plateau

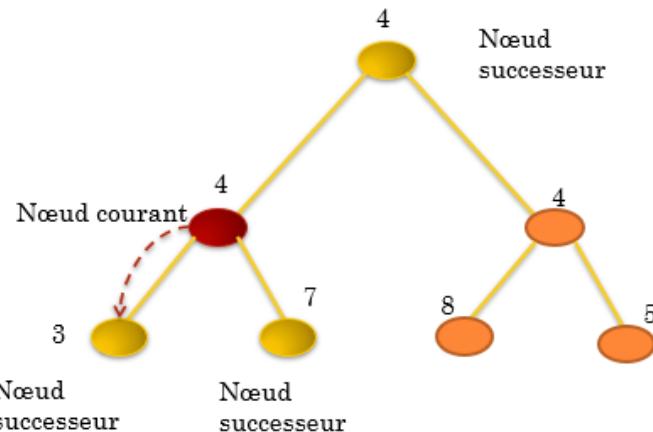
- ◇ Si on arrive à dépasser l'espace plat et on choisit un état successeur de façon aléatoire.



Les **plateaux bloquent** l'algorithme car aucune direction évidente n'est indiquée.

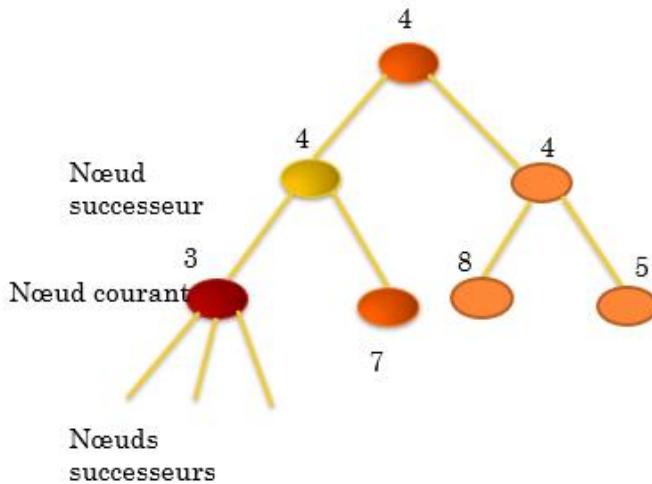
Plateau

- ◇ Le successeur gauche est le meilleur choix puisqu'il minimise la fonction objectif.



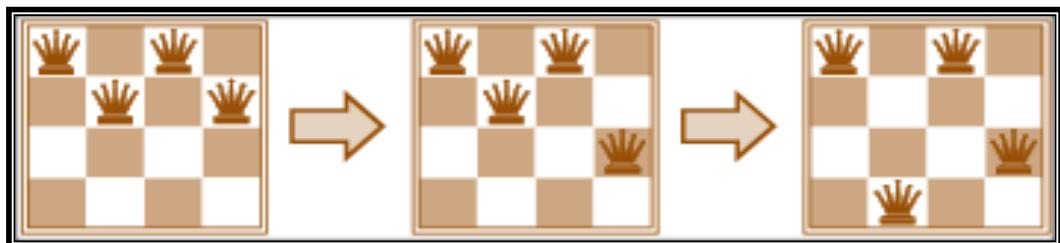
Plateau

- Il peut ainsi quitter le plateau et il pourra peut être arriver à la solution.



Exemple N-reines

- Il faut placer n reines sur un échiquier de taille $n \times n$ de sorte que deux reines ne s'attaquent mutuellement : **jamais deux reines sur la même diagonale, ligne ou colonne.**



- $n=4$: 256 configurations.
- $n=8$: 16 777 216.
- $n=16$: 18,446,744,073,709,551,616 configurations.

Hill-Climbing avec 8 reines

- ❖ n : configuration de l'échiquier avec n reines.
- ❖ F(n) (VALUE) : nombre de paires de reines qui s'attaquent mutuellement directement ou indirectement.
- ❖ On veut **minimiser F(n)**.
- Ici n=[5,6,7,4,5,6,7,6].
- F(n) pour l'état affiché: 17.
- Maintenant, pour améliorer la configuration, on doit chercher **les meilleures successeurs**, c'est-à-dire les positions où F(n) diminue en déplaçant **une seule reine** dans sa colonne.

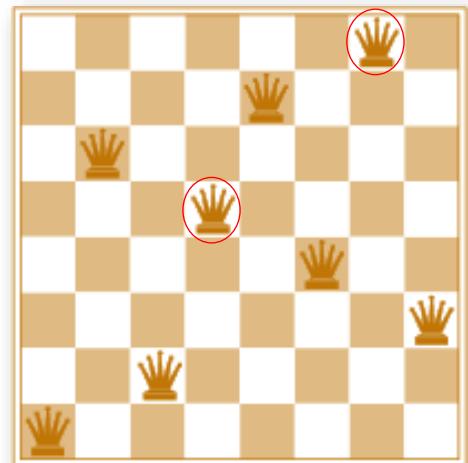
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	13	16	13	16
17	14	17	15	15	14	16	16
18	14	14	18	15	15	15	14
14	14	13	17	12	14	12	18

Minimum local dans l'espace des états des 8-reines

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	13	16	13	16
17	14	17	15	15	14	16	16
18	14	14	18	15	15	15	14
14	14	13	17	12	14	12	18

$h=17$

Après 5 étapes
→

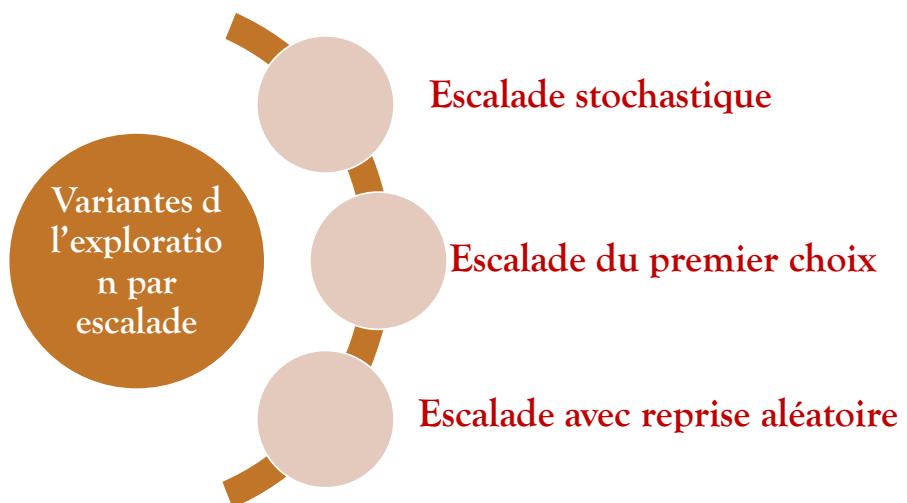


$h=1$ (minimum local)d

Surmonter le plateau

- **L'algorithme s'arrête en présence d'un plateau:** l'algorithme de Hill Climbing peut se stopper prématulement lorsqu'il rencontre un **plateau**.
- **Une amélioration vise à se déplacer latéralement dans l'espoir que le plateau s'avère être un palier : mais attention au bouclage;** l'idée est d'explorer latéralement (dans un espace où les voisins n'améliorent pas immédiatement la solution), dans l'espoir que le plateau rencontré est **un palier** et qu'il mène à une meilleure solution après quelques déplacements.
- **Une solution :** imposer une limite au nombre de déplacements.

Variantes de l'exploration par escalade



Escalade stochastique

Escalade stochastique

- **Principe:**

✓ Au lieu de toujours choisir le meilleur mouvement ascendant, cette méthode sélectionne **aléatoirement** un mouvement parmi les mouvements ascendants possibles. La probabilité de choisir un mouvement donné dépend de son **importance** (par exemple, un mouvement qui améliore significativement la solution a une probabilité plus élevée d'être choisi).

- **Avantage:**

✓ Réduit le risque de rester bloqué sur un plateau ou un maximum local.

- **Limites:**

✓ Moins efficace en termes de convergence que l'escalade classique si le paysage de la fonction est simple.

✓ Nécessite un réglage des probabilités pour être efficace

Escalade stochastique

Escalade du premier choix

- **Principe:**

Cette méthode génère des **successeurs aléatoires** jusqu'à ce qu'un successeur meilleur que l'état courant soit trouvé. Contrairement à l'escalade classique, qui évalue tous les voisins avant de choisir le meilleur, cette méthode s'arrête dès qu'elle trouve une amélioration.

- **Avantage:**

✓ Utile lorsque le nombre de voisins est très grand, car elle évite d'évaluer tous les voisins.

- **Limites:**

✓ Peut être lente si les améliorations sont rares.

✓ Risque de convergence vers un optimum local.

Escalade avec reprise aléatoire.

Escalade avec reprise aléatoire.

Principe:

Cette méthode combine l'escalade classique avec des **redémarrages aléatoires**. Si l'algorithme atteint un optimum local, il redémarre à partir d'un **point aléatoire** dans l'espace de recherche.

Avantages:

- ✓ Permet d'explorer différentes régions de l'espace de recherche, augmentant les chances de trouver un optimum global.
- ✓ Simple à implémenter et efficace pour les problèmes avec de nombreux optima locaux.

Limites:

- ✓ Peut être coûteux en temps de calcul si le nombre de redémarrages est élevé.
- ✓ Nécessite un bon équilibre entre exploration (redémarrages) et exploitation (escalade).

Hill-Climbing : verdict

- Il peut trouver une solution très rapidement.
- Il peut être **piégé** dans des maximum/minimum locaux.
- Peut **osciller** indéfiniment en revenant à un état antérieurement visité : cas du maximum local plat.

Paysage de l'espace d'états pour une exploration locale

- **Algorithmes de recherche local et optimisation**
 - ✓ Escalade (Hill-climbing)
 - ✓ Recherche locale en faisceau (local beam)
 - ✓ Recuit simulé (Simulated annealing)
 - ✓ Algorithmes génétiques

Recherche locale en faisceau: principe

La **recherche locale en faisceau** (ou **Beam Search** en anglais) est une variante de la recherche locale qui est souvent utilisée dans les problèmes de recherche d'optimisation où l'espace de recherche est trop vaste pour effectuer une recherche exhaustive.

==> Contrairement à la recherche locale classique (comme **hill climbing**), qui explore un seul chemin jusqu'à une solution ou un plateau, la recherche locale en faisceau explore plusieurs chemins **en parallèle**, mais en limitant le nombre de chemins explorés à chaque étape.

Recherche locale en faisceau: principe

Faisceau de solutions :

1. Au lieu de maintenir une seule solution courante (comme dans l'escalade classique), la recherche locale en faisceau maintient un **ensemble de k solutions** (appelé "faisceau"). Ce nombre k est un paramètre fixé à l'avance et généré aléatoirement.

Génération des voisins :

2. Pour chaque solution dans le faisceau, générez un ensemble de voisins (mouvements possibles).

Sélection des meilleures solutions :

3. A chaque étape, tous les successeurs des k états sont générés.
 - ✓ Si l'un d'eux est un but, l'algorithme s'arrête.
 - ✓ Sinon, il sélectionne les k meilleurs successeurs (former le nouveau faisceau)

Itération :

4. Répétez le processus de génération de voisins et de sélection jusqu'à ce qu'un critère d'arrêt soit atteint

Variante de Recherche locale en faisceau

C'est une variante de la **recherche en faisceau** qui introduit une composante aléatoire dans le processus de sélection des successeurs. Au lieu de simplement sélectionner les k meilleurs successeurs de manière déterministe



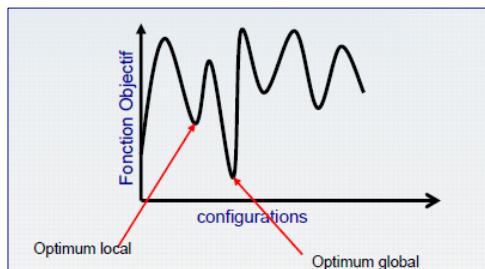
Calcul de la probabilité de sélection : Chaque successeur a une probabilité de sélection qui est une fonction de sa qualité (plus la qualité est élevée, plus la probabilité de sélection est grande). Cette probabilité peut être définie de manière croissante.

Paysage de l'espace d'états pour une exploration locale

- Algorithmes de recherche local et optimisation
 - ✓ Escalade (Hill-climbing)
 - ✓ Recherche locale en faisceau (local beam)
 - ✓ Recuit simulé (Simulated annealing)**
 - ✓ Algorithmes génétiques

Recuit simulé: principe générale

- Le recuit Simulé « *Simulated Annealing* » est une meta-heuristique destinée à résoudre au mieux les problèmes dits d'optimisation difficile.
- Evite le piégeage dans **les minima locaux** de la fonction objectif



Inspiré de la métallurgie :

- Durcissement des métaux par chauffage à haute température.
- Refroidissement progressif pour atteindre la cristallisation.

Source d'inspiration

Chauffer le matériau et le porter à l'état liquide (énergie élevée)

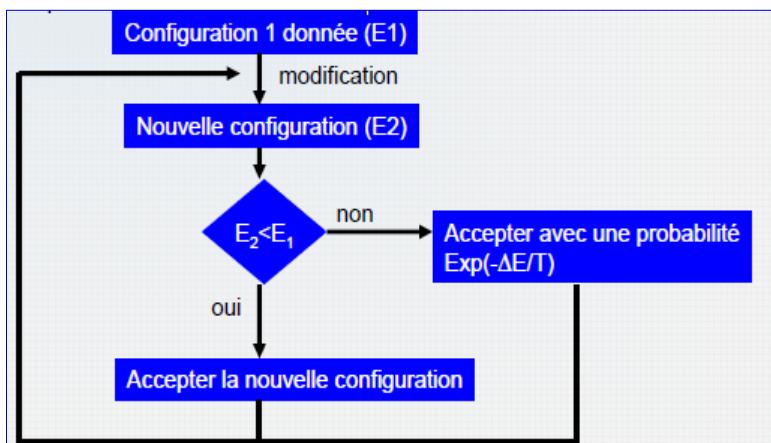
Refroidissement lent de la température en marquant des paliers de température de durée suffisante

**Refroidissement rapide de la température
Technique de la trempe**

- Etat solide cristallisé, état stable., \leftrightarrow • Apparition de défauts dans le matériau, structure amorphe, état métastable
- Minimum absolu de l'énergie \leftrightarrow • Minimum local d'énergie
- Transformation désordre ordre \leftrightarrow • Figer un état désordonné, non cristallisé

Recuit Simulé algorithme

- Metropolis[1953] a établit un algorithme qui simule l'évolution du système physique vers son équilibre thermodynamique à une température donnée T



Reavit Simulé algorithme

KirkPatrick et al 1983 proposent un algorithme qui est basé sur une analogie entre le recuit thermique et la résolution de problèmes d'optimisation combinatoire

Reavit thermique

- Les états du solide
- Les énergies des états
- L'état à énergie minimale
- Le refroidissement rapide

Reavit Simulé

- Les solutions réalisables
- Les valeurs de la fonction objectif calculées sur ces solutions
- Solution optimale du problème
- Recherche locale

Reavit Simulé algorithme

Commencer avec une solution initiale : solution courante

Température := T0 (*Température initiale*)

Tant que la condition d'arrêt n'est pas remplie faire

Pour i de 1 à N faire

Calculer nouvelle solution (perturber la solution)
 $\Delta = \text{coût (nouvelle solution)} - \text{coût (solution courante)}$

si $\Delta < 0$ alors

conserver nouvelle solution

sinon

si $\text{random}[0,1] < e^{-\Delta/\text{Température}}$ alors

conserver nouvelle solution

fin si

fin si

fin pour

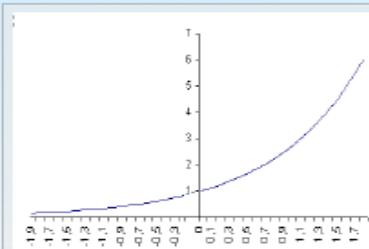
Température = $\alpha * \text{Température}$ (*nouveau palier de température*)

fin Tant que

La température actuelle dans le processus de recuit simulé, qui est une valeur qui diminue progressivement au fil des itérations.

Plus T est élevé, plus la probabilité d'accepter une solution moins bonne est grande (favoriser l'exploration)

Reduit Simulé



Température ↗

$\Delta/\text{Température} \rightarrow 0$

$$P = e^{[-\Delta/\text{Température}]} \rightarrow 1$$

0 P 1 Il est plus probable qu'une valeur aléatoire entre 0 et 1 soit inférieure à P

A température élevée la plupart des mouvements sont acceptés

L'algorithme équivaut à une simple marche dans l'espace des configurations

Température ↘

$-\Delta/\text{Température} \rightarrow -\infty$

$$P = e^{[-\Delta/\text{Température}]} \rightarrow 0$$

0 P 1 Il est plus probable qu'une valeur aléatoire entre 0 et 1 soit supérieure à P

A température faible la plupart des mouvements augmentant la température sont refusés

L'algorithme se ramène à une amélioration itérative classique

Température intermédiaire

L'algorithme autorise de temps en temps des transformations qui dégradent la fonction objectif

L'algorithme laisse une chance au système pour s'extraire d'un minimum local

Reduit Simulé

- Un paramètre T dit température, qui tend vers zéro avec le temps, influence aussi la probabilité :
 - ✓ La probabilité décroît à mesure que T baisse.
 - ✓ Plus T est grande plus un mauvais mouvement a des chances d'être pris.
- Le nombre d'itérations et la diminution des probabilités sont définis à l'aide d'un schéma (*schedule*) de températures, en ordre décroissant
- Lorsque T s'approche de 0, l'algorithme se comporte comme celui de l'escalade (aucune dégradation du coût n'est acceptée).

Paysage de l'espace d'états pour une exploration locale

- **Algorithmes de recherche local et optimisation**
 - ✓ Escalade (Hill-climbing)
 - ✓ Recuit simulé (Simulated annealing)
 - ✓ Recherche locale en faisceau (local beam)
 - ✓ Algorithmes génétiques

Algorithme génétique

- Les algorithmes génétiques sont des techniques d'optimisation stochastiques inspirées de la théorie de l'évolution de Darwin.
- Ainsi, on imite, au sein d'un programme, la capacité d'une population d'organismes vivants à s'adapter à son environnement, à l'aide de mécanismes de sélection et d'héritage génétique.
- Il utilise des principes de l'évolution naturelle :
 - ✓ La loi du plus fort.
 - ✓ Deux individus génétiquement favorisés donnent généralement des enfants génétiquement favorisés.
 - ✓ Existence de mutations.

Algorithme génétique

Un algorithme génétique cherche une bonne approximation des optima d'une fonction définie sur un espace de données.

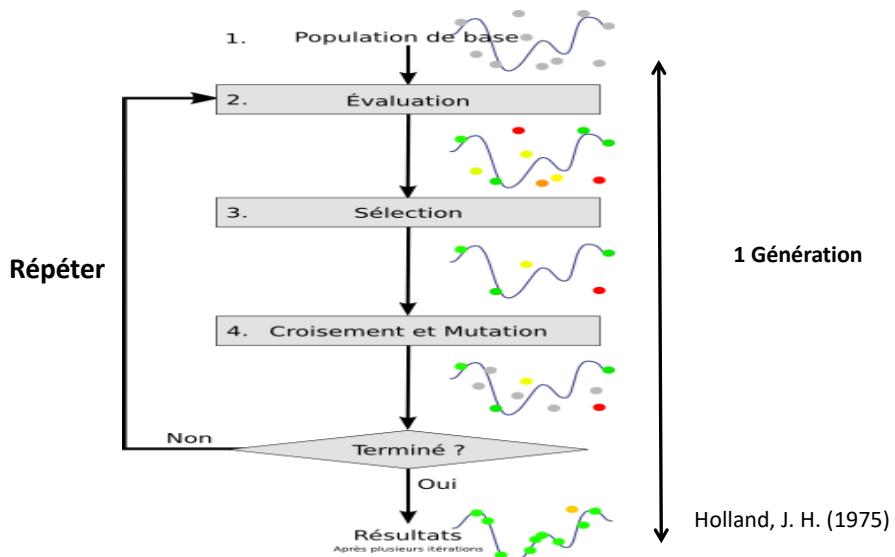
Pour l'utiliser, on doit disposer des cinq éléments suivants :

1. **Un principe de codage** de l'élément de population. Cette étape associe à chacun des points de l'espace de recherche, une structure de données
2. **Un mécanisme de génération de la population initiale:** Ce mécanisme doit être capable de produire une population d'individus non homogène qui servira de base pour les générations futures
3. **Une fonction à optimiser:** Celle-ci retourne une valeur de IR^+ appelée fitness ou fonction d'évaluation de l'individu.

Algorithme génétique

4. **Des opérateurs permettant de diversifier la population** au cours des générations et d'explorer l'espace de recherche.
 - ✓ **l'opérateur de croisement** recompose les gènes d'individus existant dans la population,
 - ✓ **l'opérateur de mutation** a pour but de garantir l'exploration de l'espace de recherche
5. **Des paramètres de dimensionnement** ; taille de la population, nombre total de générations ou critère d'arrêt, probabilités d'application des opérateurs de croisement et de mutation.

Algorithme génétique



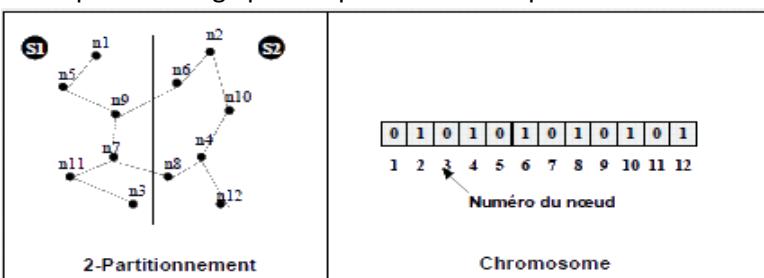
Codage d'une solution

- A chaque variable de décision x_i , nous faisons correspondre un gène.

Chromosome = Solution

Chaque solution est représentée par un chromosome

Exemple de codage pour le problème de bi-partitionnement



L'individu est représenté par n bits ou n est le nombre de noeuds de graphes

Chaque gène prend 0 si le noeud appartient à la première sous-partition, sinon 1.

AG: génération de la population initiale

- **Si la solution optimale dans l'espace de recherche est inconnue**, il est naturel de produire aléatoirement des individus en choisissant des valeurs pour les variables de décision à partir des domaines associés, en veillant à ce que les individus produits respectent les contraintes.
- **Si des informations sur le problème est connu** il est préférable d'orienter la génération de la population initiale vers un sous-domaine\sous espace de recherche afin d'accélérer la convergence.

AG: génération de la population initiale

La taille de la population est proportionnelle à sa diversité

- Population réduite Au bout de quelques générations, on atteint un optimum local
- Population large Exploration maximale de l'espace de recherche



- **La taille est généralement, entre 20 et 30. Si la taille du Chromosome (solution) est importante, il faut augmenter la population (100 individus généralement)**

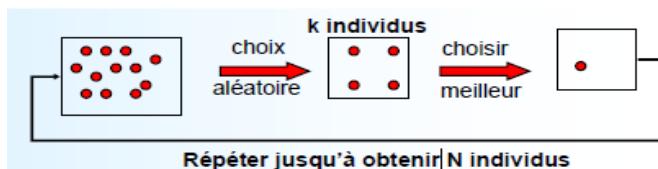
AG: Exemple de Méthode de sélection

- Le fait que les individus formant la génération intermédiaire passent leurs gènes à la génération suivante rend nécessaire la sélection de bons individus.
- Il existe deux classes de méthodes de sélection d'individus:
 1. Un individu est sélectionné selon sa valeur de fitness relativement aux valeurs des fitness des autres individus sélectionnés de la population.
 2. On trie d'abord la population selon le fitness, ensuite chaque individu se voit associer un rang en fonction de sa position. L'individu est alors sélectionné non pas selon sa valeur de fitness, mais selon son rang dans la population.

Méthode de sélection

Tournois déterministe:

- Choisir aléatoirement un nombre k d'individus dans la population
- Sélectionner pour la reproduction celui qui a la meilleure performance
- On fait autant de tournois que d'individus sélectionnés



Deux versions existent:

- Les individus qui participent à un tournoi sont remis dans la population
- Les individus qui participent à un tournoi sont retirés de la population

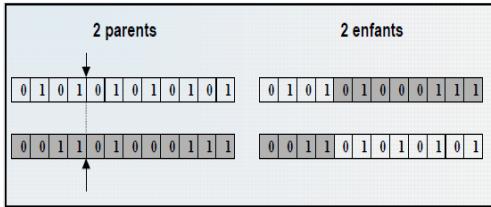
Remarque:

- Un tirage sans remise permet de faire N/K tournois avec une population de N individus
- Une copie de la population est engendrée quand elle est épuisée, autant de fois que nécessaire

Croisement

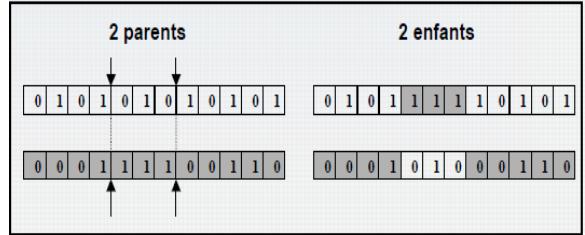
AG: Croisement

Croisement avec un seul point



Position choisi aléatoirement

Croisement avec deux points



Pos1 Pos2

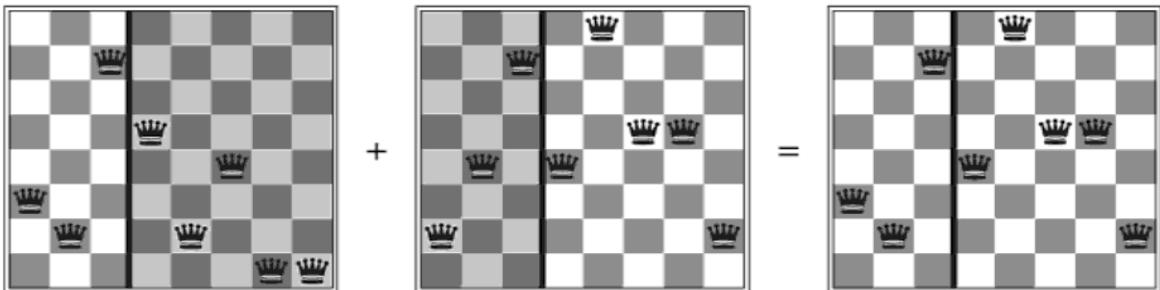
- Pos1 et pos2 sont générés aléatoirement

- En termes plus concrets, cet opérateur permet de créer de nouvelles combinaisons de solutions à partir des gènes de parents

➔ Favorise l'intensification

Croisement

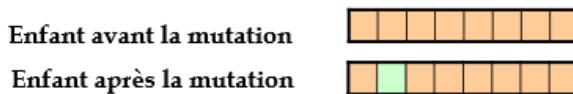
AG: Croisement



Mutation

AG: Croisement

- L'opérateur de mutation consiste généralement à tirer aléatoirement un gène dans le chromosome et à le remplacer par une valeur aléatoire.
- Cet opérateur consiste à changer la valeur allélique d'un gène avec une probabilité **PM**
- On peut aussi prendre **PM= 1 / lg** où **lg** est la longueur de la chaîne de bits codant



➔ Favorise La diversification/Exploration

Décision sous incertitude

Exploration avec actions non déterministes

Problème

L'**exploration avec actions non déterministes** est un problème courant en intelligence artificielle et en planification, où les actions d'un agent peuvent avoir des résultats **imprévisibles** ou **multiples**. Contrairement aux actions déterministes, où chaque action conduit à un seul résultat connu, les actions non déterministes introduisent de l'**incertitude**, ce qui rend la planification et l'exploration plus complexes.

- Lorsque l'environnement est non déterministe, l'agent ne connaît pas à l'avance **le résultat de ses actions**.
- **Après** chaque action, les percepts indiquent ses effets réels.
- La solution n'est plus une séquence mais **un plan contingent** (stratégie) qui indique quoi faire selon les percepts reçus.

Exploration avec actions non déterministes

L'**exploration avec actions non déterministes** signifie que lorsqu'un agent exécute une action, **le résultat n'est pas totalement prévisible**. Contrairement aux actions **déterministes** où chaque action mène toujours au même état, ici, une action peut conduire à différents états selon une distribution de probabilité.

==> Dans ce contexte, les actions non déterministes peuvent conduire à plusieurs états possibles avec des probabilités différentes.

Pourquoi?

- Modéliser l'**incertitude** du monde réel (ex: robots, jeux, systèmes stochastiques).
- Éviter les stratégies trop rigides, car le même choix peut avoir plusieurs conséquences.

Exemple d'algorithme : Q-learning

Aspirateur capricieux

- L'exemple de l'aspirateur capricieux illustre un **environnement non déterministe**, les actions de l'agent (l'aspirateur) **n'ont pas toujours le même effet**.

Cas 1 : l'aspirateur aspire une case sale

- Effet attendu : la case devient propre.
- Mais parfois, **une case adjacente est aussi nettoyée par erreur** (effet non prévu).

Cas 2 : l'aspirateur aspire une case propre

- Effet attendu : rien ne change.
- Mais parfois, **la case devient sale** au lieu de rester propre !

==> Une solution contingente à ce problème peut être représentée par un arbre ET-OU.

Exploration sans observation

- Dans un environnement **non observable**, l'agent ne connaît pas directement son état actuel. Il doit alors **raisonner sur un ensemble d'états possibles** appelés **état de croyance** (*belief state*).

Exemple de l'aspirateur :

L'aspirateur sait que l'environnement contient des cases, mais il ne sait pas où il est. Il commence donc avec un état de croyance où toutes les cases sont possibles.

- ✓ **Action 1** : Il aspire → mais il ne sait pas s'il a nettoyé la bonne case.
- ✓ **Action 2** : Il avance à droite → Il réduit son incertitude, mais ne sait toujours pas exactement où il est.

==> l'exploration des états de croyance est rarement faisable en pratique à cause de l'explosion combinatoire.

Résolution de problèmes partiellement observables

- Sachez que dans un environnement :

✓ **Entièrement observable** → L'agent connaît l'état exact de l'environnement.

Ex : Percept(s) = s (il sait exactement où il est et où est la saleté).

✓ **Partiellement observable** → L'agent n'a qu'une information partielle sur l'environnement.

Ex : Percept(s) = [A, sale] → Il sait qu'il est en A et que la case est sale, mais pas où se trouve la saleté ailleurs.

✓ **Non observable** → L'agent ne reçoit aucun feedback.

Ex : Percept(s) = nil → Il doit explorer à l'aveugle.

Recherche hors ligne

- La **recherche hors ligne** (ou **planification a priori**) consiste à explorer l'espace des solutions **avant** d'exécuter les actions dans l'environnement. L'agent construit un **plan complet** avant de commencer à agir.



Les **algorithmes d'exploration hors ligne** planifient toutes les **actions** avant de commencer l'exécution. Ils ne s'adaptent pas en fonction des percepts pendant l'exécution, ce qui signifie qu'ils supposent que l'environnement est **déterministe** et **entièrement observable**.

Recherche en ligne

- La **recherche en ligne** consiste à planifier des actions et à prendre des décisions dynamiques en fonction des **percepts** reçus en temps réel, pendant que l'agent évolue dans son environnement. Contrairement à la recherche hors ligne où l'on calcule la solution complète avant d'agir, la recherche en ligne doit être capable de s'adapter à un environnement **incertain, stochastique, ou partiellement observable**.

Recherche en ligne

- ✓ Les algorithmes d'exploration en ligne entrelacent calcul et action : l'agent doit exécuter des actions et non réaliser uniquement des calculs.
- ✓ Il ne connaît pas le résultat de ses actions avant de les effectuer réellement.
- ✓ Après chaque action l'agent reçoit un percept lui indiquant l'état atteint et enrichir sa carte de l'environnement.
- ✓ Contrairement à un algorithme hors ligne comme A* il ne peut pas développer un nœud dans une partie de l'espace ensuite développer un autre nœud dans une autre partie : les actions sont simulées et non réelles

Recherche *en ligne* vs Recherche avec actions non-déterministes

	Recherche en ligne	Recherche avec actions non-déterministes
Principe	L'agent découvre l'environnement au fur et à mesure de l'exploration	L'agent découvre l'environnement au fur et à mesure de l'exploration
Connaissance de l'environnement	L'agent ne connaît pas la carte complète à l'avance	L'agent connaît l'environnement, mais les actions ont des effets incertains
Exemple	Un robot qui explore un labyrinthe sans carte	Un robot qui essaie d'aller à droite mais peut aussi finir en haut ou en bas avec une certaine probabilité

Fonctionnement de la recherche *en ligne*

- Le développement des nœuds se fait dans un ordre local : l'exploration en profondeur possède cette propriété.
- L'état résultant de l'exécution d'une action est enregistré.
- Chaque fois qu'une action de l'état courant n'a pas été explorée, l'agent l'essaie.
- Avec une exploration en profondeur hors ligne l'état visité est supprimé de la file, mais dans l'exploration en ligne l'agent revient en arrière physiquement.

Exploration en ligne locale

- L'exploration **par escalade** possède la propriété de localité et est donc une exploration en ligne.
- Mais elle n'est pas intéressante puisque l'agent reste bloqué dans des maximum/minimum locaux.
- Les reprises aléatoires ne sont pas applicables : impossible de se transporter dans un nouvel état.
- **Une solution** est le parcours aléatoire : sélection au hasard d'une action disponible.
- Une meilleure solution consiste à mémoriser la meilleure estimation courante du coût pour atteindre le but à partir de chaque état visité : l'agent est appelé LRTA*.