CHAPITRE II

Algorithmes de recherche et résolution des problèmes

Résolution des problèmes

Etapes intuitives par un humain

- Modéliser la situation actuelle
- Enumérer les solutions possibles
- Evaluer la valeur des solutions
- Retenir la meilleure option possible satisfaisant le but

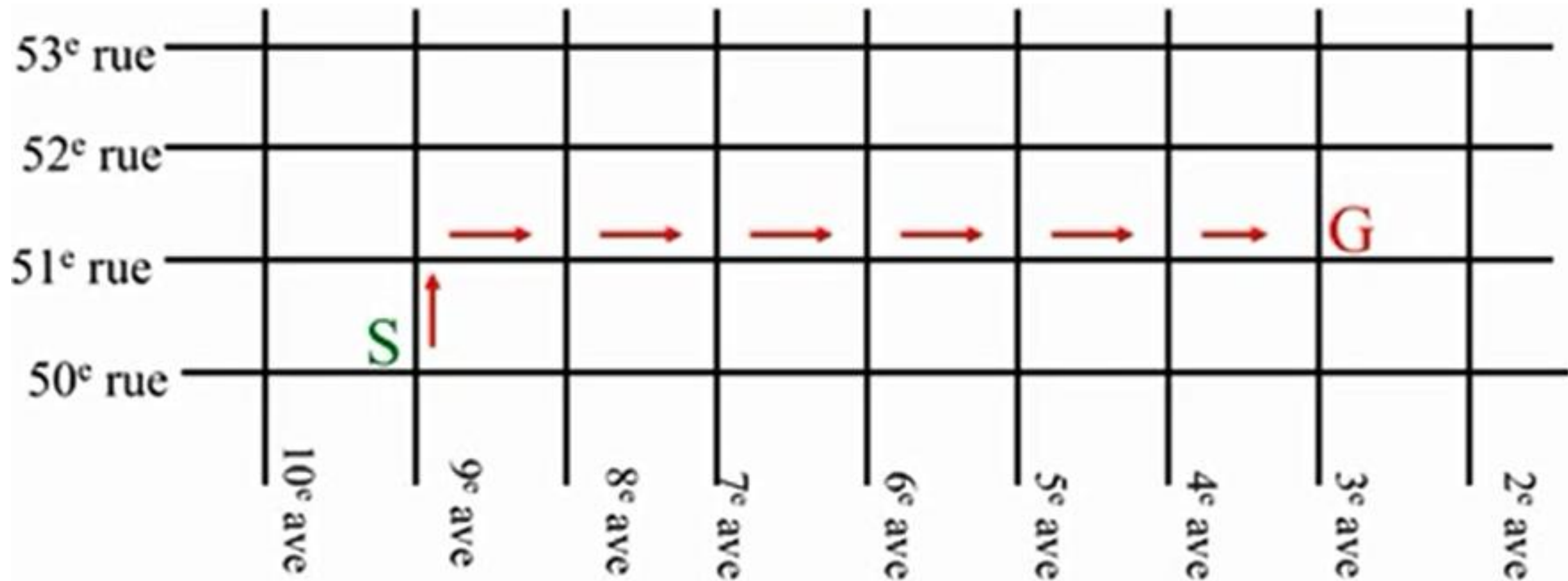
Comment parcourir efficacement la liste des solutions ?

La résolution de plusieurs problèmes peut être faite par une recherche dans un graphe :

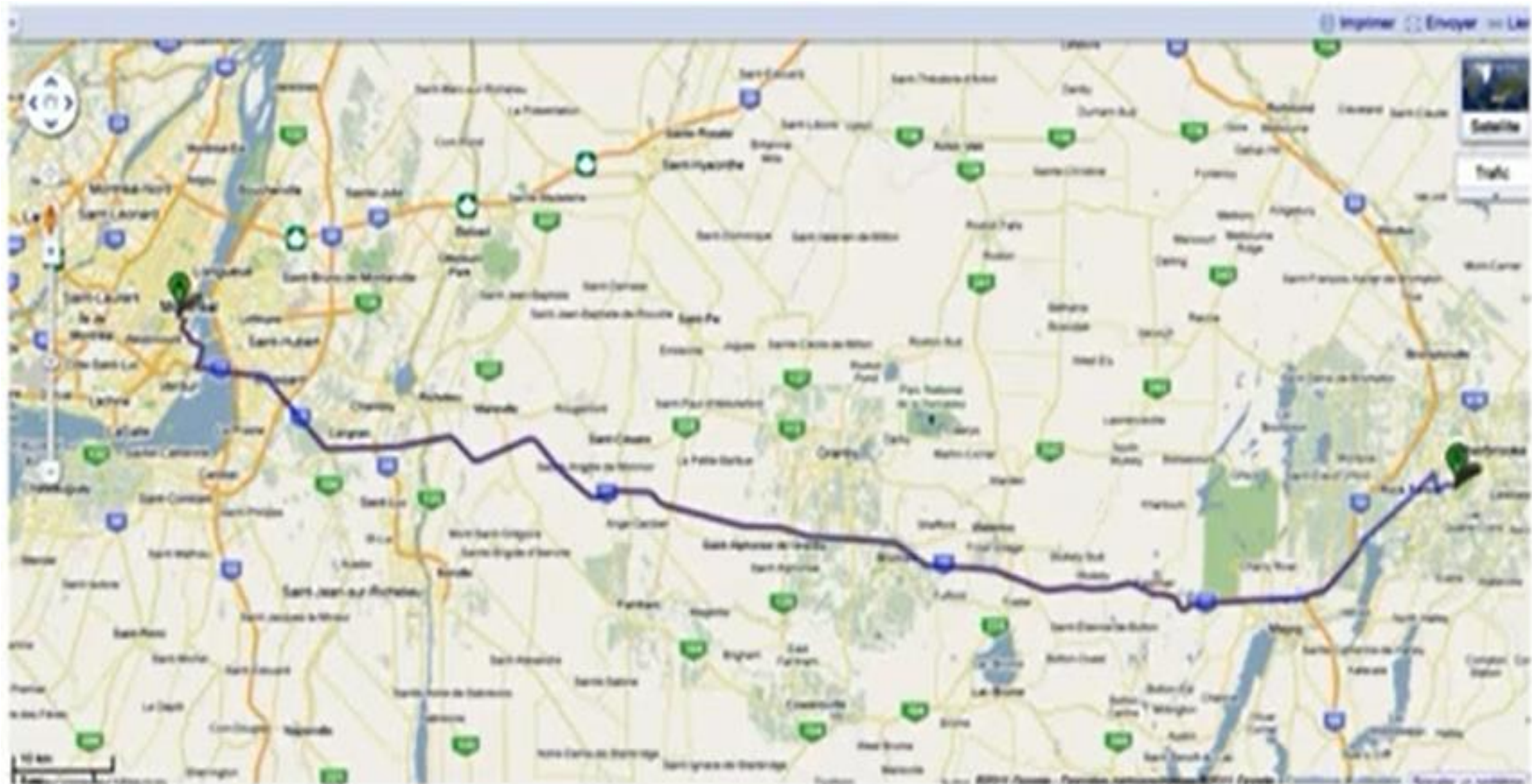
- Chaque nœud correspond à un état de l'environnement
- Chaque chemin à travers un graphe représente une suite d'actions
- La solution : il suffit de chercher le chemin qui satisfait le mieux notre mesure de performance

Exemple : trouver un chemin dans une ville

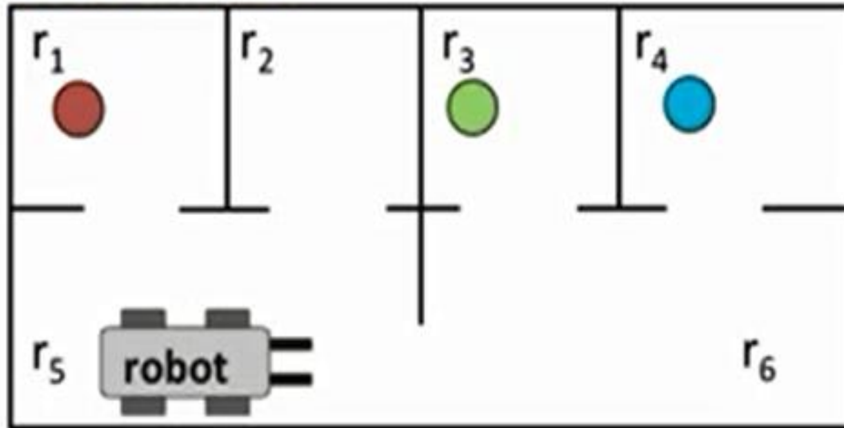
Trouver un chemin de la 9e ave & 50e rue à la 3e ave et 51e rue

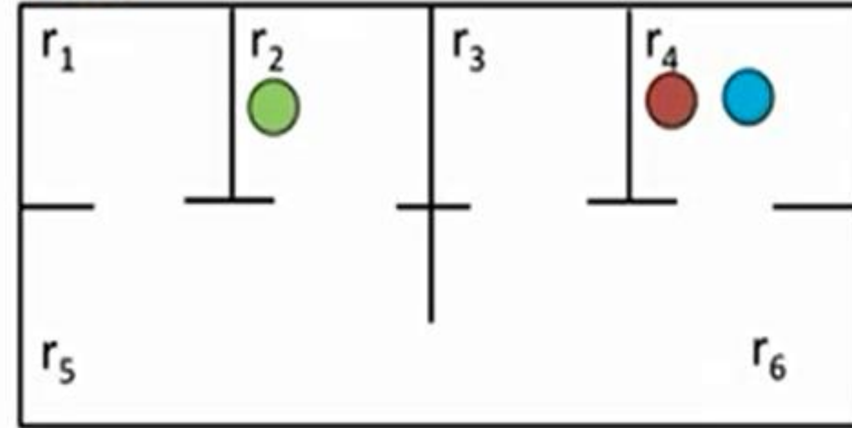


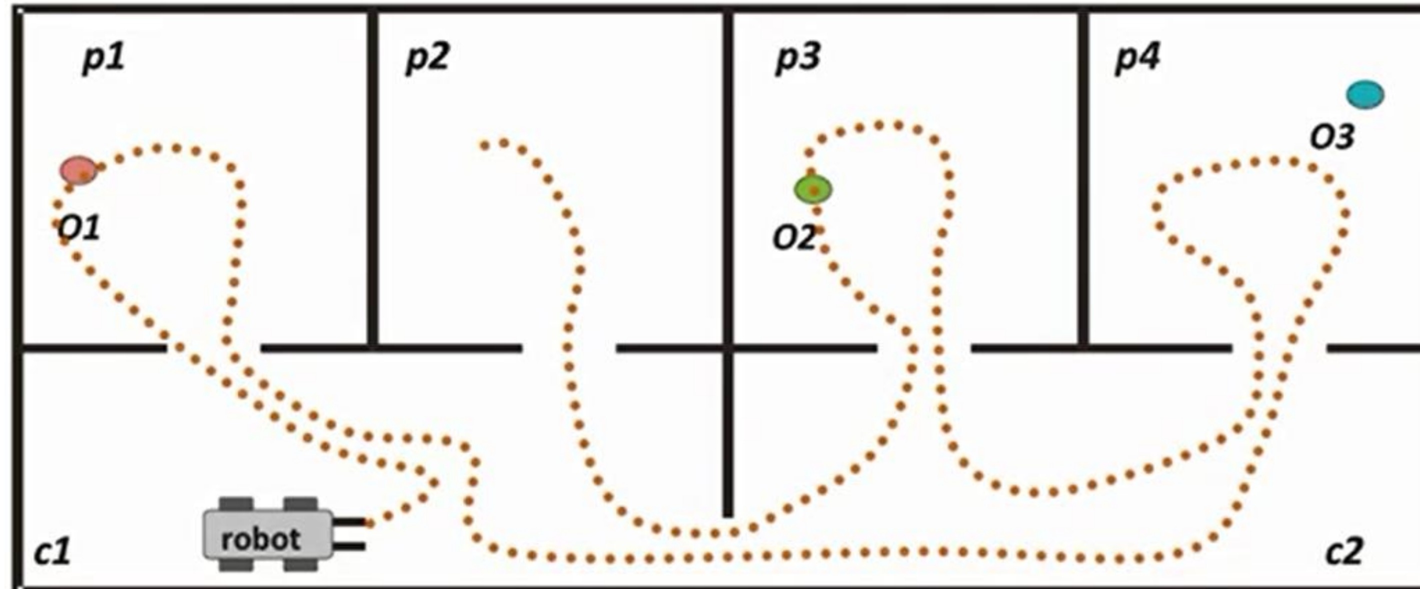
Exemple : Google Maps



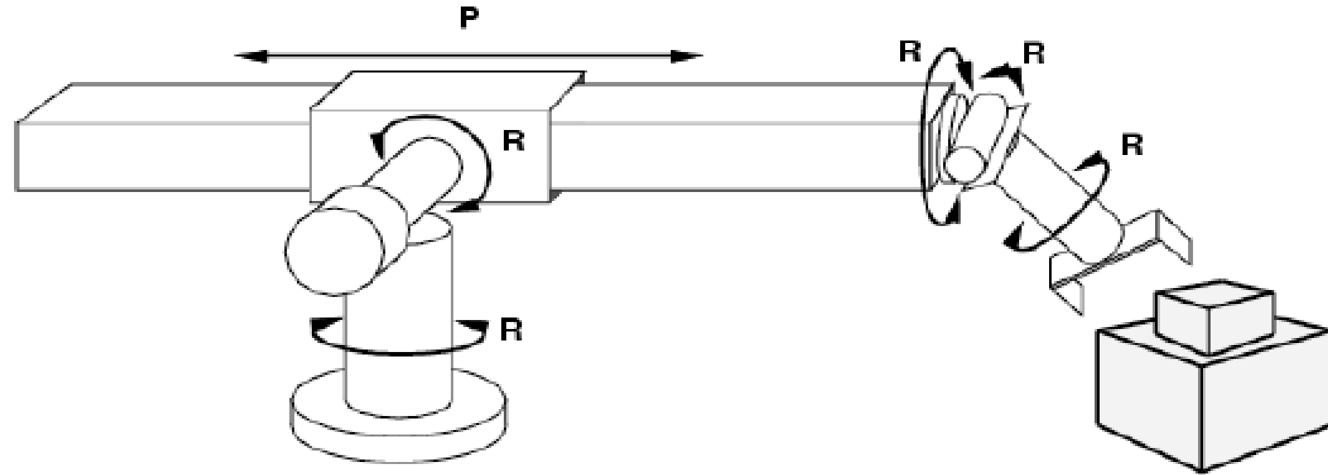
Exemple : livrer des colis







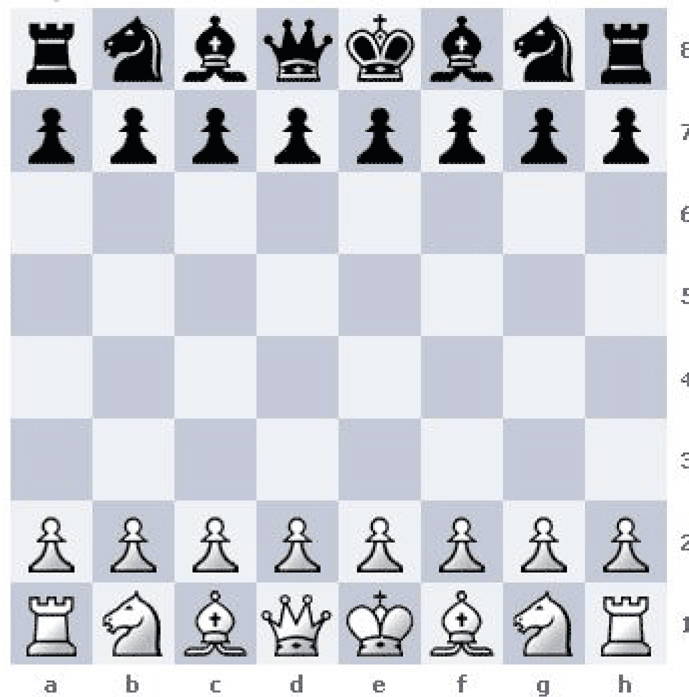
Exemple: Agent bras manipulateur



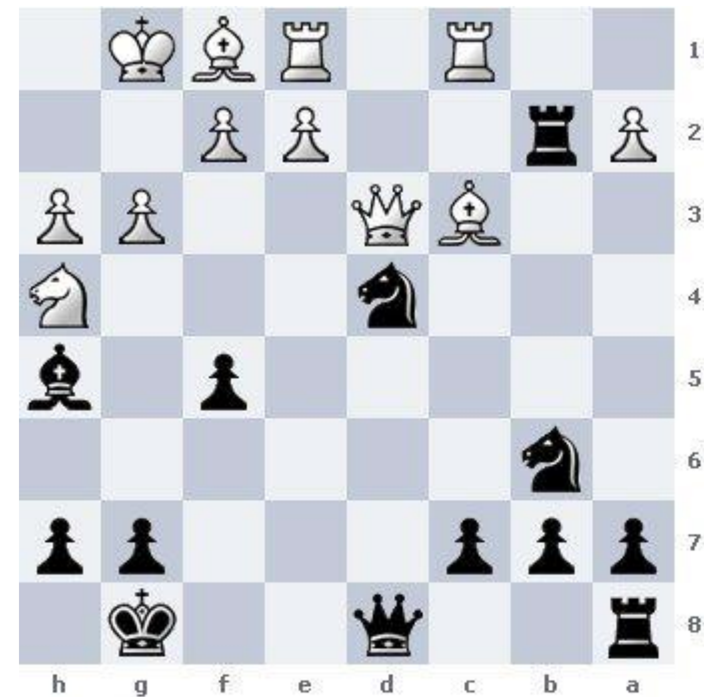
- Etats? configuration du bras, objets à assembler
- Actions? mouvements du bras
- Test état but? objet assemblé (positions relatives des pièces)
- Coût? temps; nombre d'actions

Exemple : Jeu d'échec

Etat initial



Etat but



Exemple : N-Puzzle
(Jeu de Taquin)

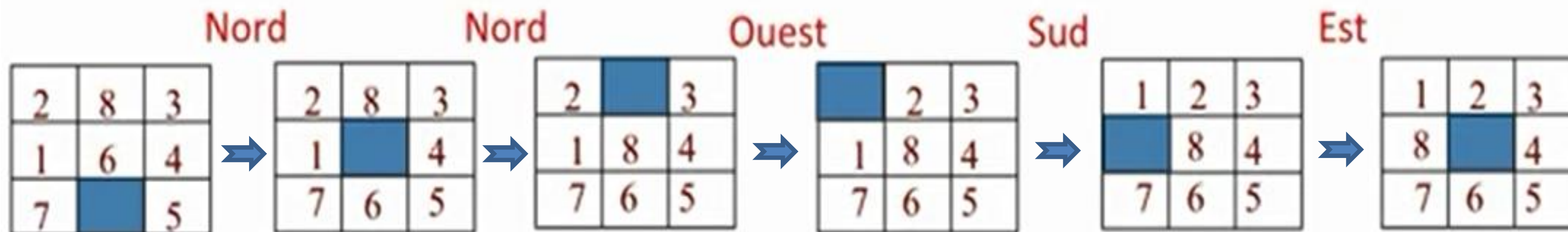
Etat initial

2	8	3
1	6	4
7		5

?

Etat but

1	2	3
8		4
7	6	5



Problème de recherche dans un graphe

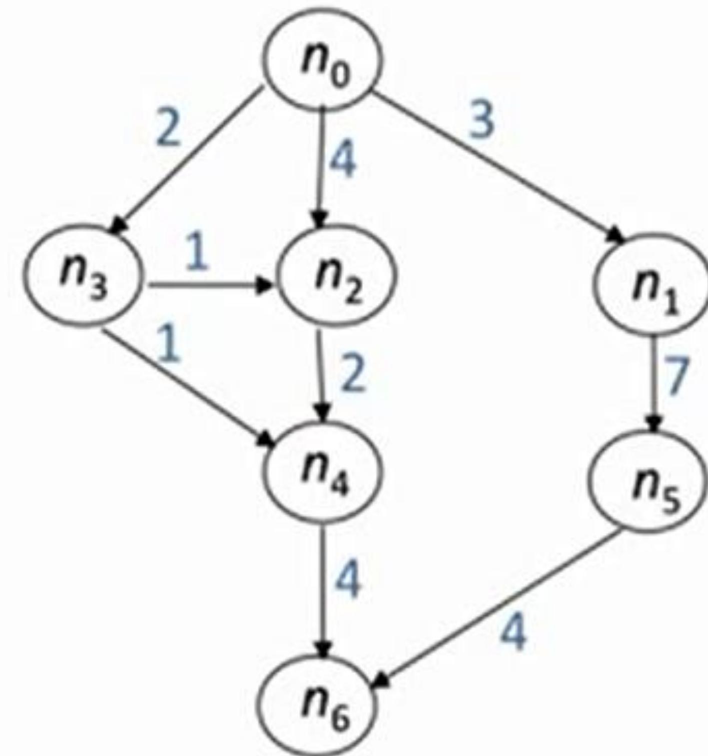
- Algorithme de recherche dans un graphe
  - Entrées :
    - un nœud initial
    - une fonction *goal*(*n*) qui retourne *true* si le but est atteint
    - une fonction de transition *transitions*(*n*) qui retourne les nœuds successeurs de *n*
    - une fonction *c*(*n*,*n'*) strictement positive, qui retourne le coût de passer de *n* à *n'* (permet de considérer le cas avec coûts variables)
  - Sortie :
    - un chemin dans un graphe (séquence nœuds / arrêtes)
  - Le **coût d'un chemin** est la **somme des coûts des arrêtes** dans le graphe
  - Il peut y avoir plusieurs nœuds qui satisfont le but

- Enjeux :
  - trouver un chemin solution, ou
  - trouver un chemin optimal, ou
  - trouver rapidement un chemin (optimalité pas importante)

Exemple : trouver chemin entre deux villes

- Villes : nœuds
- Chemins entre deux villes : arrêtes

- Ville de départ : nœud (état) initial n_0
- Routes entre les villes : *transitions*(n_0) = (n_3, n_2, n_1)
- Distances entre les villes : $c(n_0,n_2) = 4$
- Ville de destination : *goal*(n) : vrai si $n=n_6$
  (où n_6 est le nœud de la ville de destination)



Algorithmes de recherche

Tout problème de recherche est caractérisé par une situation de départ et un but à atteindre et un espace de recherche:

- L’espace de recherche est composé de l’ensemble des états possibles.
- Déterminer les opérations possibles pour passer d’un état à un autre.
- Déterminer une stratégie de recherche.

Il existe différentes stratégies :

- **Non informées (Aveugles) :**

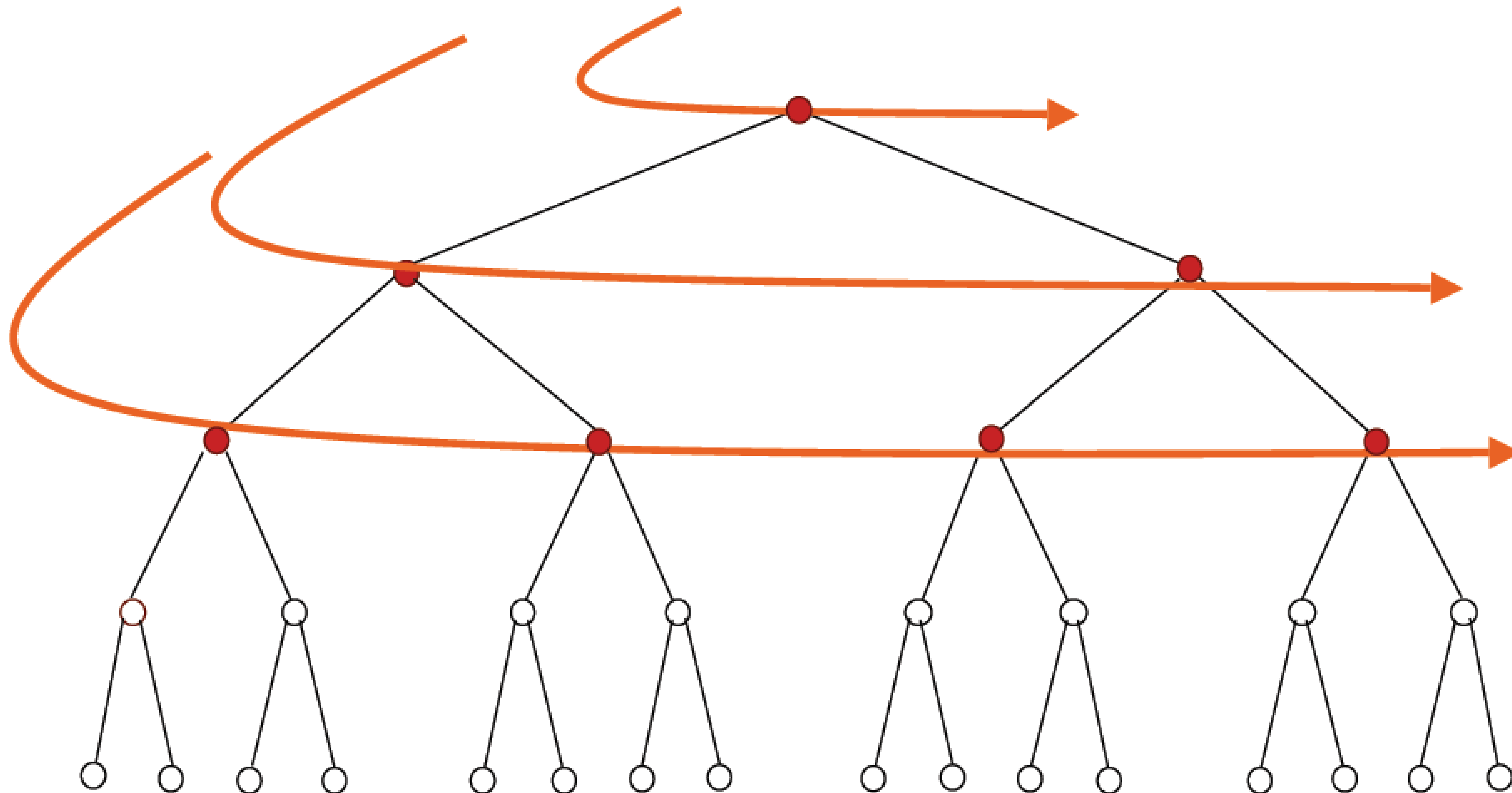
- En largeur d’abord (breadth-first search)
- En profondeur d’abord (depth-first search) et ses variantes

- **Informées :**

- A coût uniforme
- Meilleur d’abord (Best-first search),
  - Recherche gloutonne (greedy best-first search),
  - Algorithme A*

Algorithme de recherche : Largeur d'abord

- Principe : Pour un nœud donné, explorer les nœuds frères avant d'explorer leurs enfants.



Algorithme de recherche : Largeur d'abord

Algorithme Largeur d'abord :

1- Mettre le nœud *état initial* dans une file FIFO : **OPEN**

2- Si **n** correspond à l'état final alors succès

3- Si **OPEN** est vide alors Echec

4- Retirer **n** de **OPEN**

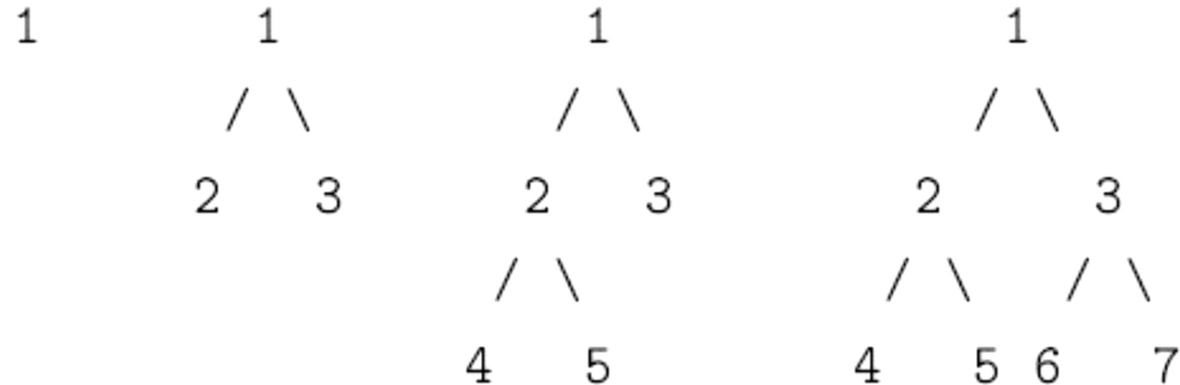
5- Si **n** n'a pas de successeur alors aller à 3 sinon :

- Développer les successeurs de **n**
- Les insérer dans **OPEN**
- Etablir le chaînage
- Insérer **n** dans **Closed** (une file contenant les nœuds déjà explorés)

6- Si parmi les successeurs il existe des états finaux alors succès sinon aller à 3

Algorithme de recherche : Largeur d'abord

- Parcours d'un arbre



1. Mettre 1 dans la liste. On obtient [1].

2. Enlever le premier élément de la liste (le 1) et rajouter ses successeurs 2, 3.

On obtient [2,3].

3. Enlever le premier élément de la liste (le 2) et rajouter ses successeurs 4, 5.

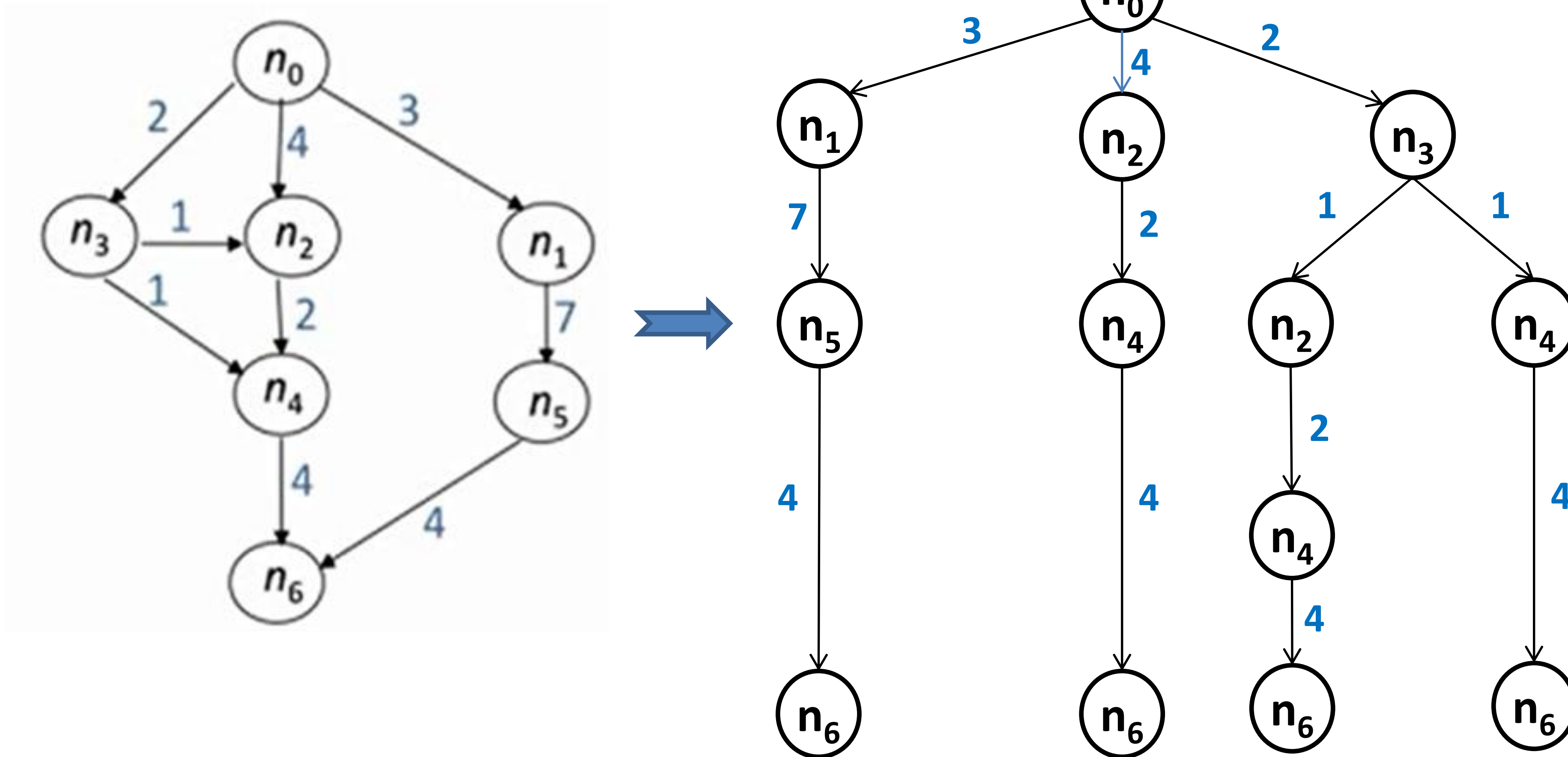
On obtient [3,4,5].

4. Enlever le premier élément de la liste (le 3) et rajouter ses successeurs 6, 7.

On obtient [4,5,6,7].

Algorithme de recherche : Largeur d'abord

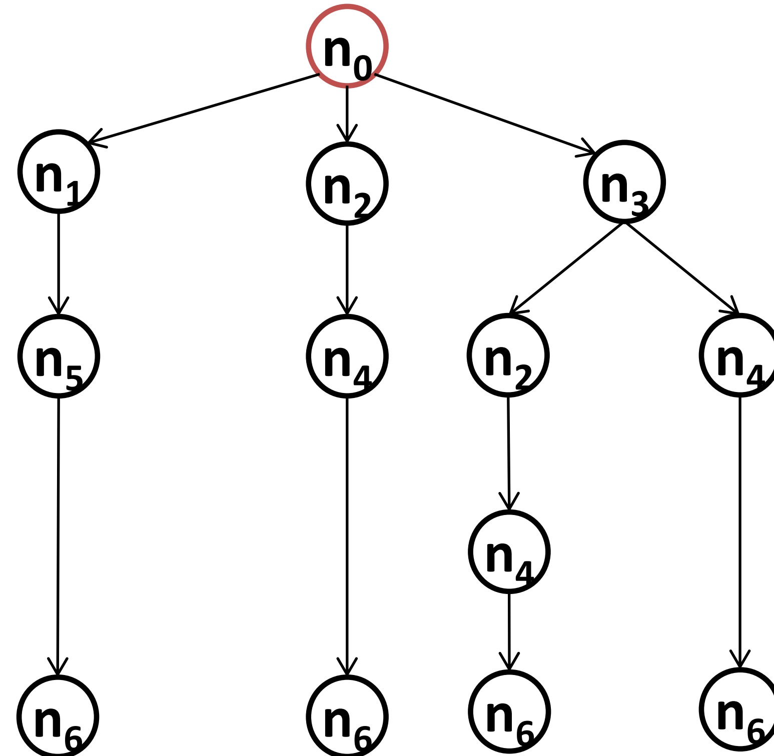
Exemple : Chemin entre deux villes n_0 et n_6



Algorithme de recherche : Largeur d'abord

- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la liste **Open**. On obtient **[$\mathbf{n_0}$]**.



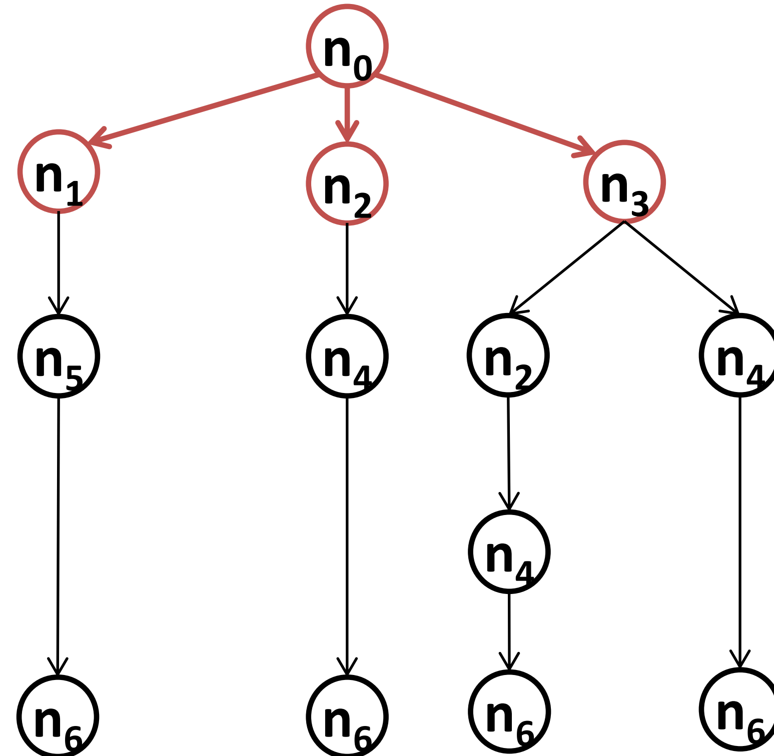
Algorithme de recherche : Largeur d'abord

- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la liste **Open**. On obtient **[$\mathbf{n_0}$]**.

2. Enlever le premier élément de la liste (le $\mathbf{n_0}$) et

rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. On obtient [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].



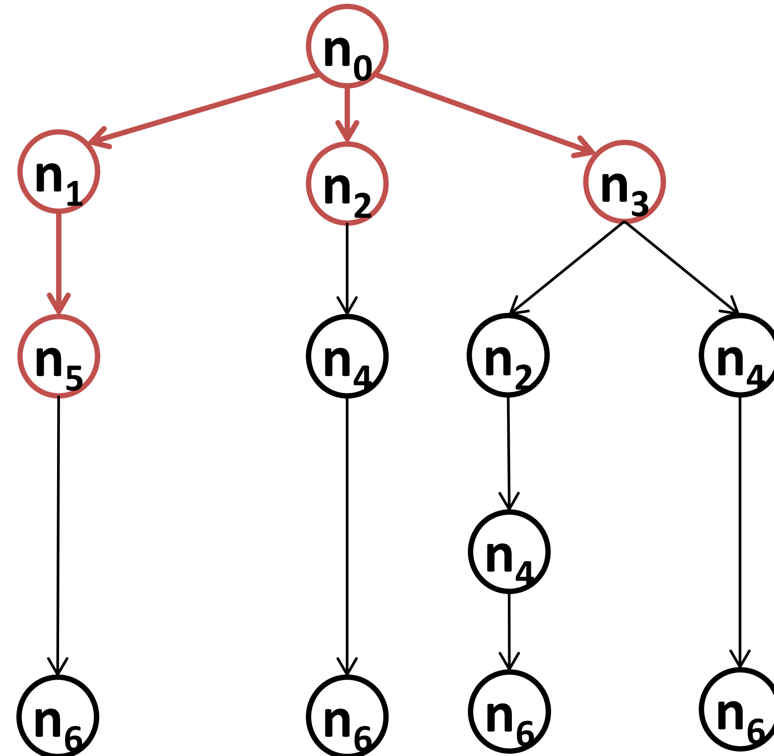
Algorithme de recherche : Largeur d'abord

- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la liste **Open**. On obtient **[$\mathbf{n_0}$]**.

2. Enlever le premier élément de la liste (le $\mathbf{n_0}$) et rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. On obtient [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].

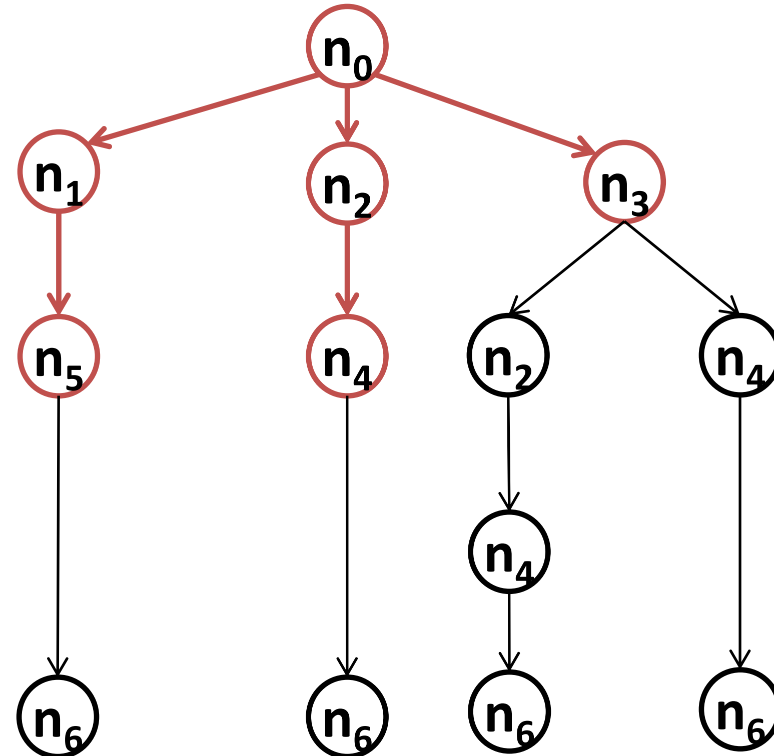
3. Enlever le premier élément de la liste (le $\mathbf{n_1}$) et rajouter son successeurs $\mathbf{n_5}$. On obtient [$\mathbf{n_2}$, $\mathbf{n_3}$, $\mathbf{n_5}$].



Algorithme de recherche : Largeur d'abord

- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la liste **Open**. On obtient **[$\mathbf{n_0}$]**.
2. Enlever le premier élément de la liste (le $\mathbf{n_0}$) et rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. On obtient [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].
3. Enlever le premier élément de la liste (le $\mathbf{n_1}$) et rajouter son successeurs $\mathbf{n_5}$. On obtient [$\mathbf{n_2}$, $\mathbf{n_3}$, $\mathbf{n_5}$].
4. Enlever le premier élément de la liste (le $\mathbf{n_2}$) et rajouter son successeurs $\mathbf{n_4}$. On obtient [$\mathbf{n_3}$, $\mathbf{n_5}$, $\mathbf{n_4}$].



Algorithme de recherche : Largeur d’abord

- **Déroulement :**

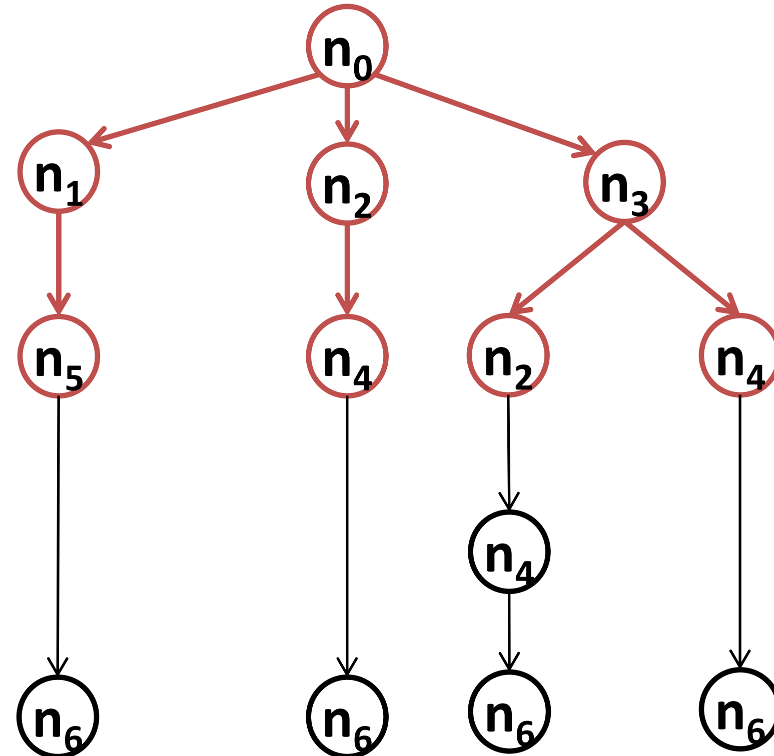
1. Mettre $\mathbf{n_0}$ dans la liste **Open**. On obtient **[$\mathbf{n_0}$]**.

2. Enlever le premier élément de la liste (le $\mathbf{n_0}$) et rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. On obtient [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].

3. Enlever le premier élément de la liste (le $\mathbf{n_1}$) et rajouter son successeurs $\mathbf{n_5}$. On obtient [$\mathbf{n_2}$, $\mathbf{n_3}$, $\mathbf{n_5}$].

4. Enlever le premier élément de la liste (le $\mathbf{n_2}$) et rajouter son successeurs $\mathbf{n_4}$. On obtient [$\mathbf{n_3}$, $\mathbf{n_5}$, $\mathbf{n_4}$].

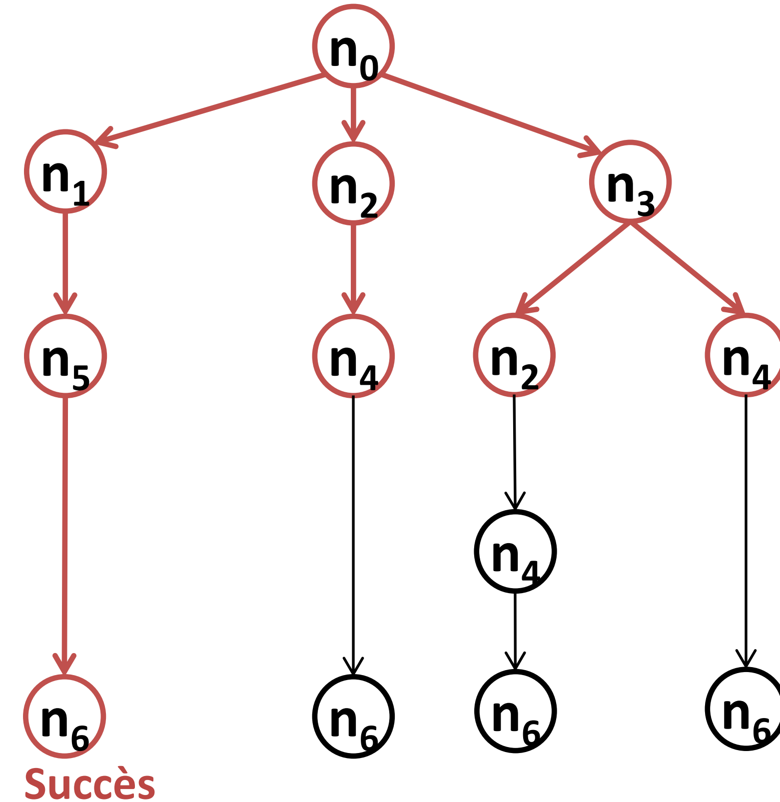
5. Enlever le premier élément de la liste (le $\mathbf{n_3}$) et rajouter ses successeurs $\mathbf{n_2}$, $\mathbf{n_4}$.et On obtient [$\mathbf{n_5}$, $\overset{2}{\mathbf{n}}_{\mathbf{4}}$, $\mathbf{n_2}$, $\overset{3}{\mathbf{n}}_{\mathbf{4}}$].



Algorithme de recherche : Largeur d'abord

- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la liste **Open**. On obtient **[$\mathbf{n_0}$]**.
2. Enlever le premier élément de la liste (le $\mathbf{n_0}$) et rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. On obtient [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].
3. Enlever le premier élément de la liste (le $\mathbf{n_1}$) et rajouter son successeurs $\mathbf{n_5}$. On obtient [$\mathbf{n_2}$, $\mathbf{n_3}$, $\mathbf{n_5}$].
4. Enlever le premier élément de la liste (le $\mathbf{n_2}$) et rajouter son successeurs $\mathbf{n_4}$. On obtient [$\mathbf{n_3}$, $\mathbf{n_5}$, $\mathbf{n_4}$].
5. Enlever le premier élément de la liste (le $\mathbf{n_3}$) et rajouter ses successeurs $\mathbf{n_2}$,$\mathbf{n_4}$.et On obtient [$\mathbf{n_5}$, $\overset{2}{\mathbf{n_4}}$, $\mathbf{n_2}$, $\overset{3}{\mathbf{n_4}}$].
6. Enlever le premier élément de la liste (le $\mathbf{n_5}$) et rajouter son successeurs $\mathbf{n_6}$.et On obtient [$\overset{2}{\mathbf{n_4}}$, $\mathbf{n_2}$, $\overset{3}{\mathbf{n_4}}$, $\mathbf{n_6}$].



$\mathbf{n_6}$ Apparait dans Open

alors arrêter

Algorithme de recherche : Profondeur d'abord

- Principe : Pour un nœud donné, explorer le premier nœud enfant avant d'explorer les nœuds frères.

Algorithme de recherche : Profondeur d'abord

Algorithme Profondeur d'abord :

1- Mettre le nœud **état initial** dans une pile LIFO : **OPEN**

2- Si pile vide alors Echec

3- Dépiler **n**

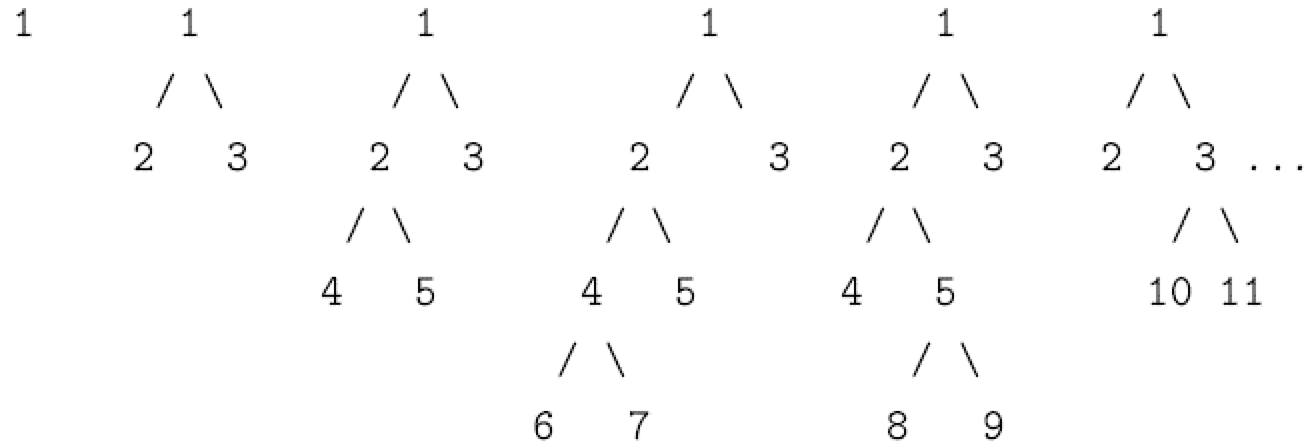
4- Développer les successeurs de **n**

- Si successeurs existent alors
  - Empiler les successeurs
  - Etablir le chaînage successeur de **n**
  - Mettre **n** dans **Closed**

6- Si parmi les successeurs il existe des états finaux alors succès sinon aller à 2

Algorithme de recherche : Profondeur d'abord

- **Parcours d'un arbre**



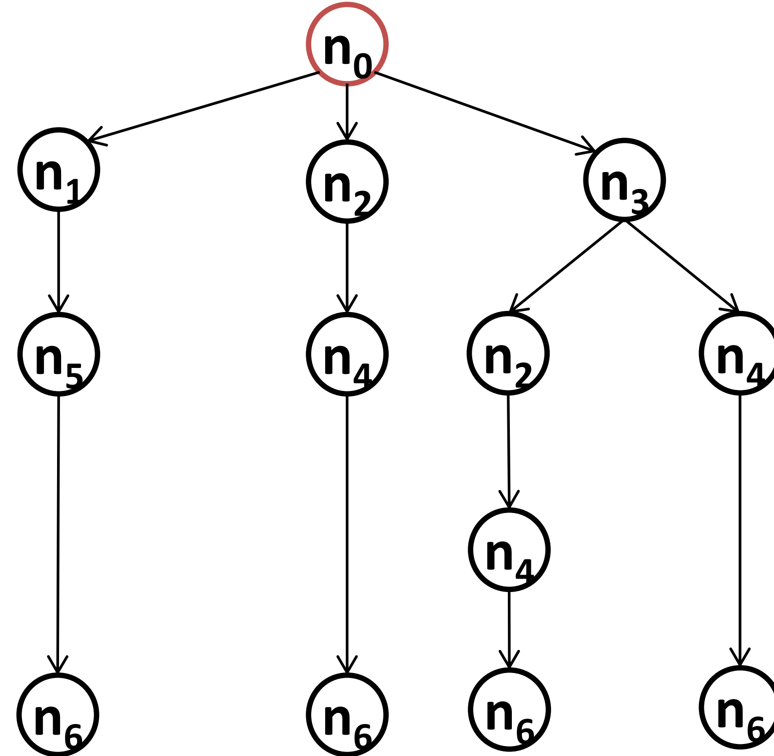
On utilise une structure de pile :

1. Mettre 1 dans la pile. On obtient [1].
2. Enlever le premier élément de la pile (le 1) et rajouter ses successeurs 2, 3. On obtient [2,3].
3. Enlever le premier élément de la pile (le 2) et rajouter ses successeurs 4, 5. On obtient [4,5,3].
4. Enlever le premier élément de la pile (le 4) et rajouter ses successeurs 6, 7. On obtient [6,7,5,3].
5. Enlever le premier élément de la pile (le 6) qui n'a pas de successeurs. On obtient [7,5,3].
6. Enlever le premier élément de la pile (le 7) qui n'a pas de successeurs. On obtient [5,3].
7. Enlever le premier élément de la pile (le 5) et rajouter ses successeurs 8, 9. On obtient [8,9,3].
8. Enlever le premier élément de la pile (le 8) qui n'a pas de successeurs. On obtient [9,3].
9. Enlever le premier élément de la pile (le 9) qui n'a pas de successeurs. On obtient [3].
10. Enlever le premier élément de la pile (le 3) et rajouter ses successeurs 10, 11. On obtient [10,11]………

Algorithme de recherche : Profondeur d'abord

- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la pile **Open**. On obtient **[$\mathbf{n_0}$]**.

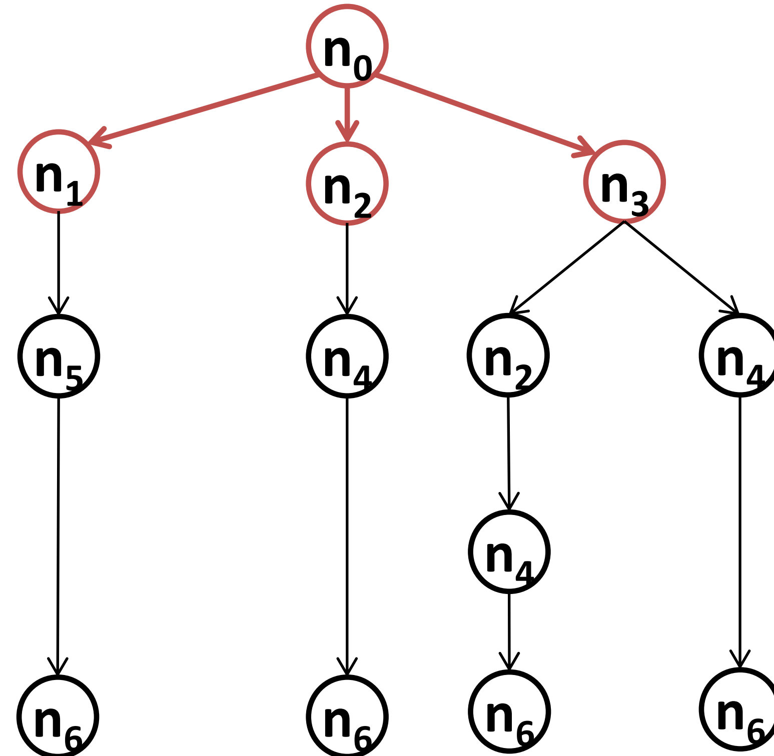


Algorithme de recherche : Profondeur d'abord

- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la pile **Open**. On obtient **[$\mathbf{n_0}$]**.
2. Enlever le premier élément de la pile (le $\mathbf{n_0}$) et

rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. On obtient [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].



Algorithme de recherche : Profondeur d'abord

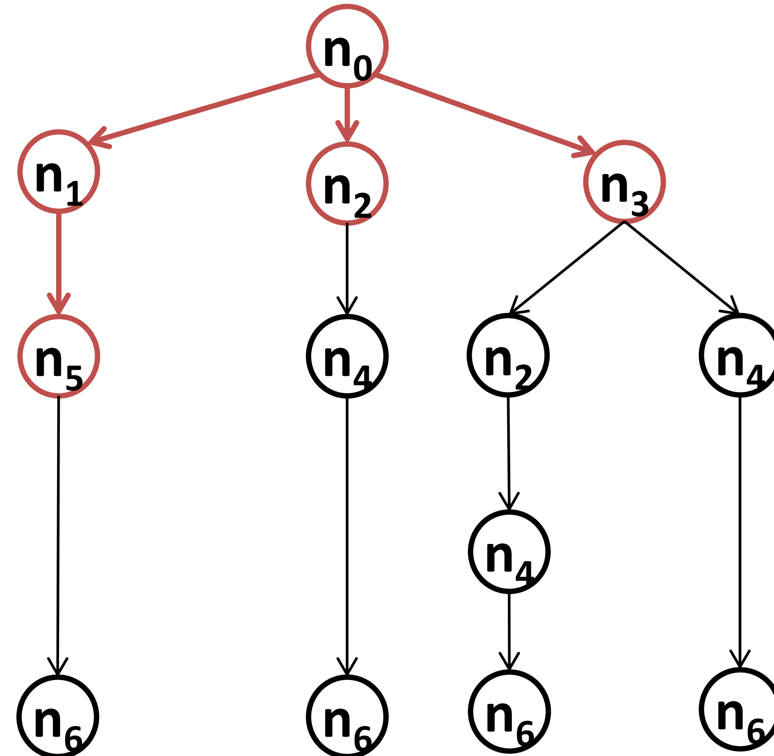
- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la pile **Open**. On obtient **[$\mathbf{n_0}$]**.
2. Enlever le premier élément de la pile (le $\mathbf{n_0}$) et

rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. On obtient [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].

3. Enlever le premier élément de la pile (le $\mathbf{n_1}$) et

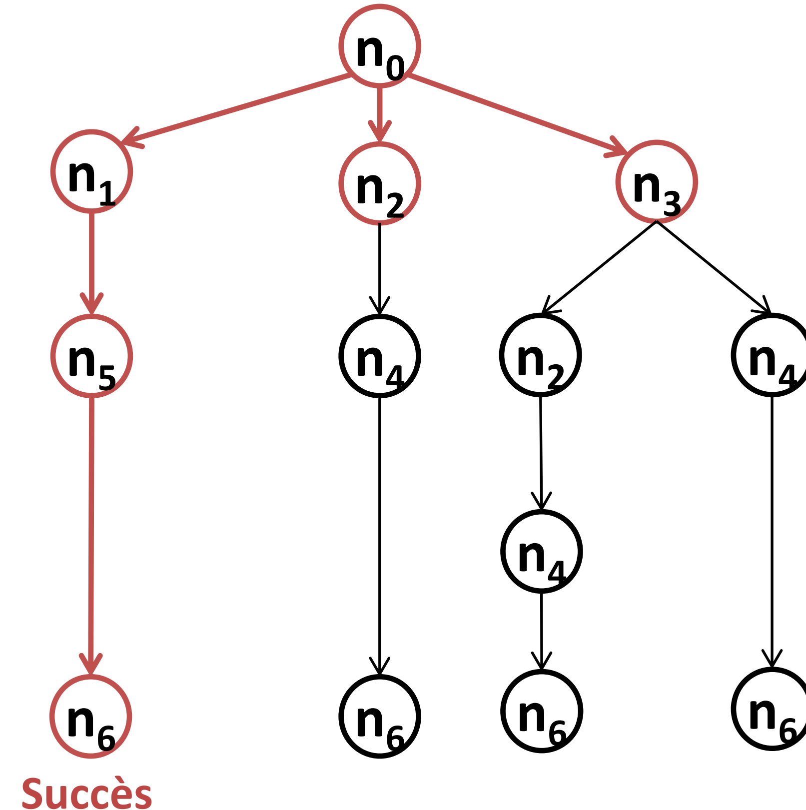
rajouter ses successeurs $\mathbf{n_5}$. On obtient [$\mathbf{n_5}$, $\mathbf{n_2}$, $\mathbf{n_3}$].



Algorithme de recherche : Profondeur d'abord

- **Déroulement :**

1. Mettre $\mathbf{n_0}$ dans la pile **Open**. On obtient **[$\mathbf{n_0}$]**.
2. Enlever le premier élément de la pile (le $\mathbf{n_0}$) et rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. On obtient [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].
3. Enlever le premier élément de la pile (le $\mathbf{n_1}$) et rajouter ses successeurs $\mathbf{n_5}$. On obtient [$\mathbf{n_5}$, $\mathbf{n_2}$, $\mathbf{n_3}$].
4. Enlever le premier élément de la pile (le $\mathbf{n_5}$) et rajouter ses successeurs $\mathbf{n_6}$. On obtient [$\mathbf{n_6}$, $\mathbf{n_2}$, $\mathbf{n_3}$].



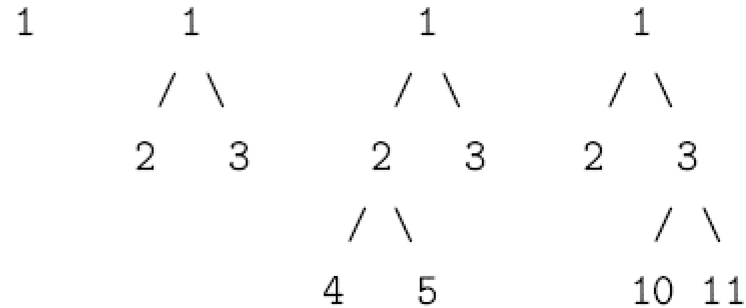
$\mathbf{n_6}$ Apparait dans Open

alors arrêter

Algorithme de recherche : Profondeur limitée

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

Exemple avec limite = 2



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

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

Algorithme de recherche : Profondeur itérative

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

Pour **p** de **0** a **infini** faire
{
recherche_profondeur_limitée(**p**)
}

Synthèse

- La stratégie en largeur d'abord est intéressante car s'il existe un chemin vers le but dans les premiers niveaux, elle trouve le plus court chemin.

- La recherche en profondeur d'abord est intéressante aussi parce qu'elle retourne le meilleur chemin Si l'état but se trouve dans les premières branches de l'arbre de recherche.

❖ Mais : ces deux stratégie sont dites aveugles car elles ne tiennent pas compte du chemin qui mène à l'état but (aucune information sur le chemin).

Algorithme de recherche : A coût uniforme

- Principe : A chaque arc du graphe est associé un coût de parcours. Cet algorithme fourni une solution de coût optimal.

Soit $C(n_i, n_j)$ le coût d'un arc allant de n_i à n_j.

La fonction de coût d'un nœud ***ns*** qui est le nœud successeur de ***n*** est calculée comme suit :

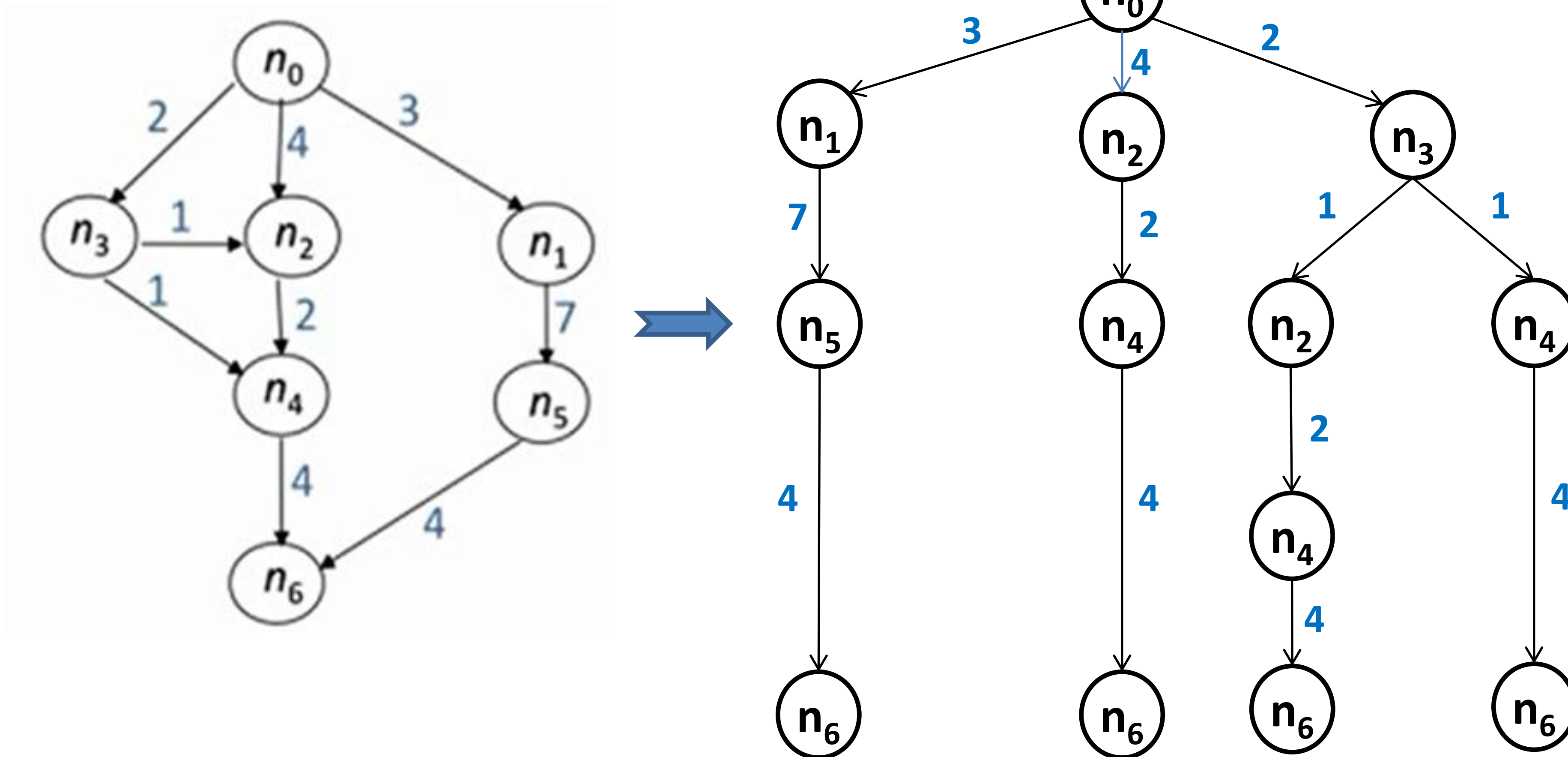
$$g(ns) = g(n) + c(n, ns)$$

g(n) est le coût jusqu'au nœud *n*

Les nœuds ouverts du graphe sont ordonnés par ordre croissant.

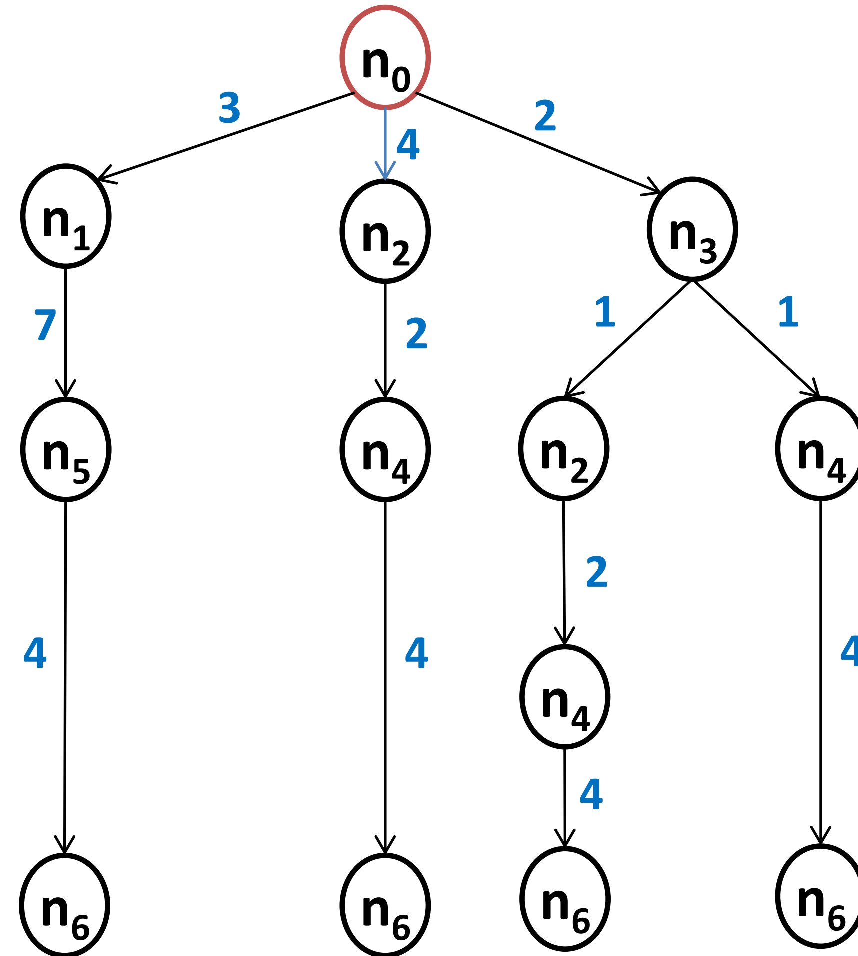
Algorithme de recherche : A coût uniforme

- **Déroulement : Chemin entre deux villes**



Algorithme de recherche : A coût uniforme

1. Mettre $\mathbf{n_0}$ dans la liste avec son coût initial. On obtient **[(n_0,0)]**.



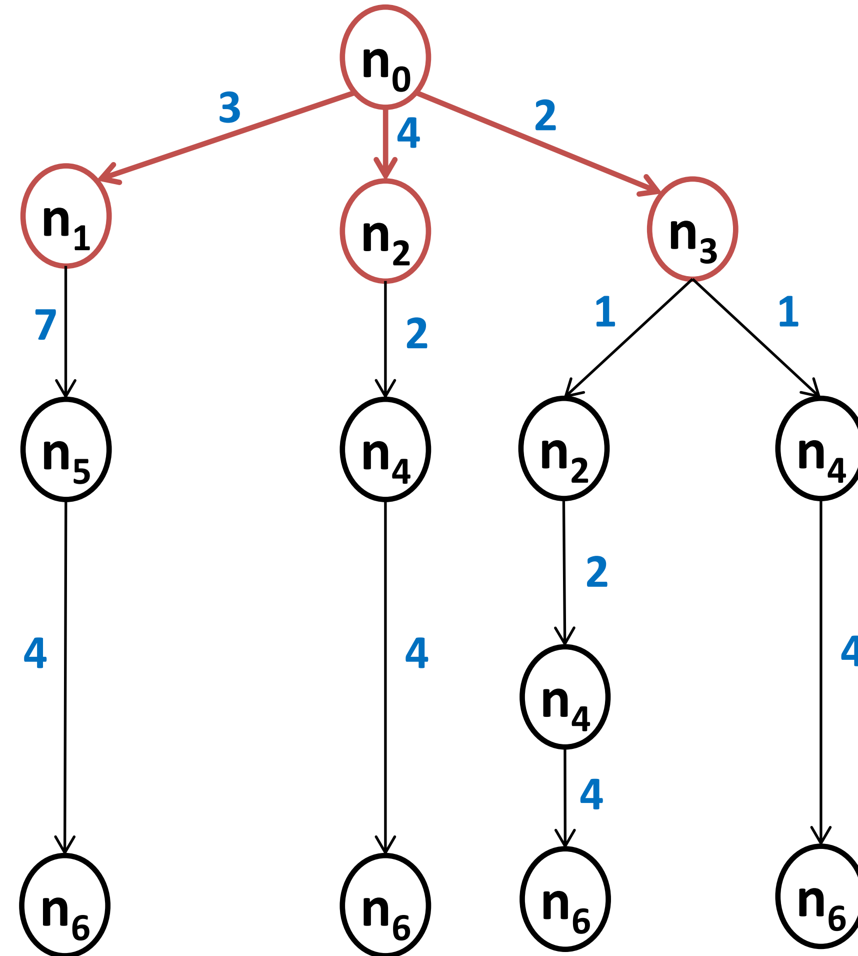
Algorithme de recherche : A coût uniforme

1. Mettre $\mathbf{n_0}$ dans la liste avec son coût initial. On obtient **[(n_0,0)]**.

2. Enlever le premier élément de la liste **(n_0,0)** et rajouter ses successeurs $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$ à la liste d'états en respectant l'ordre croissant. Pour cela on calculera le coût total de chaque successeur : *g(ns) = g(n) + c(n, ns)*

$g(n_1) = g(n_0) + c(n_0, n_1) = 3$

On obtient : **[(n_3,2, n_0), (n_1,3, n_0), (n_2,4, n_0)]**



Algorithme de recherche : A coût uniforme

1. Mettre $\mathbf{n_0}$ dans la liste avec son coût initial. On obtient $[(\mathbf{n_0},0)]$.

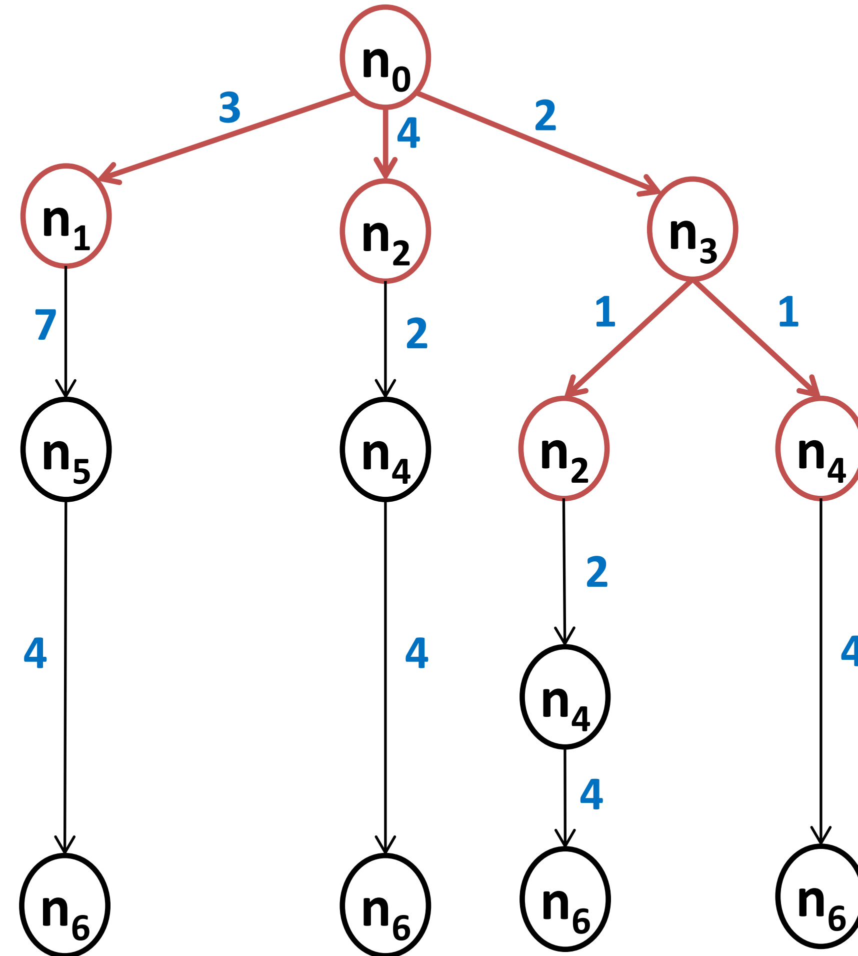
2. Enlever le premier élément de la liste $(\mathbf{n_0},0)$ et rajouter ses successeurs $\mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}$ à la liste d'états en respectant l'ordre croissant. Pour cela on calculera le coût total de chaque successeur : $g(ns) = g(n) + c(n, ns)$

$g(\mathbf{n_1}) = g(\mathbf{n_0}) + c(\mathbf{n_0}, \mathbf{n_1}) = 3$

On obtient : **$[(n_3,2, n_0), (n_1,3, n_0), (n_2,4, n_0)]$**

3. Enlever le premier élément de la liste $(\mathbf{n_3},2,\mathbf{n_0})$ et rajouter ses successeurs $\mathbf{n_2}, \mathbf{n_4}$ à la liste d'états en respectant l'ordre croissant.

On obtient : **$[(n_1,3, n_0), (n_2,3, n_3),(n_4,3, n_3), (n_2,4, n_0)]$**



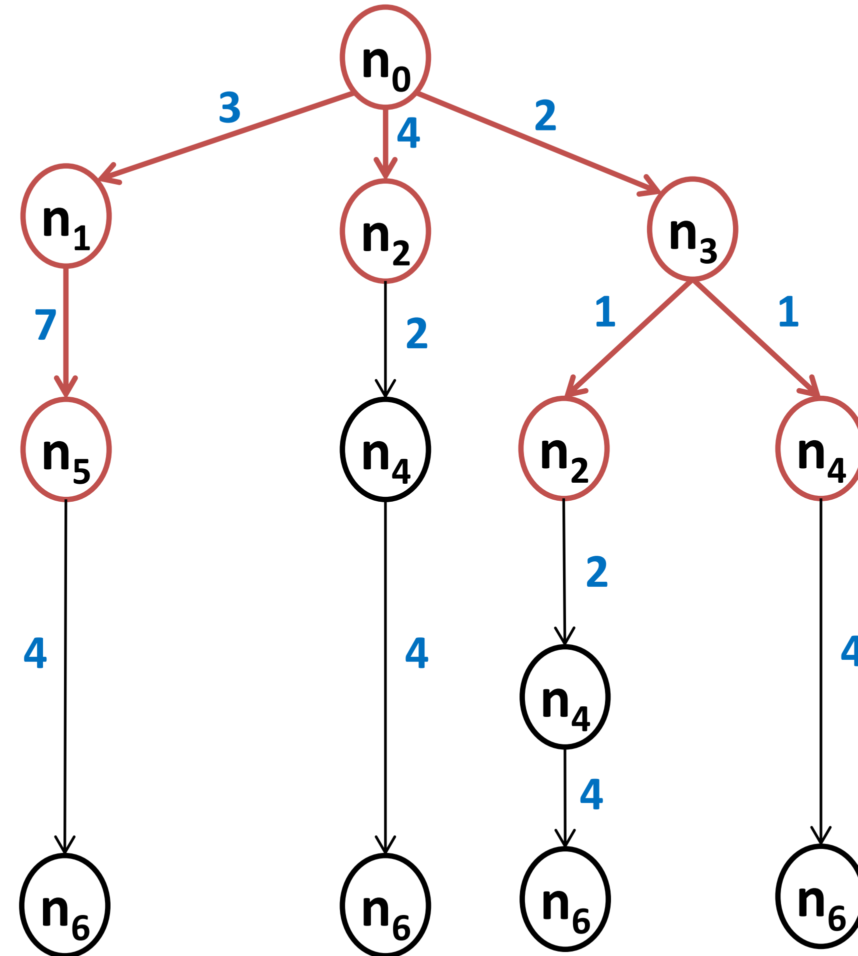
Algorithme de recherche : A coût uniforme

1. Mettre $\mathbf{n_0}$ dans la liste avec son coût initial. On obtient **[(n_0,0)]**.

2. Enlever le premier élément de la liste **(n_0,0)** et rajouter ses successeurs **n_1, n_2, n_3** à la liste d'états en respectant l'ordre croissant. Pour cela on calculera le coût total de chaque successeur : *g(ns) = g(n) + c(n, ns)*
$g(n_1) = g(n_0) + c(n_0, n_1) = 3$
On obtient : **[(n_3,2, n_0), (n_1,3, n_0), (n_2,4, n_0)]**

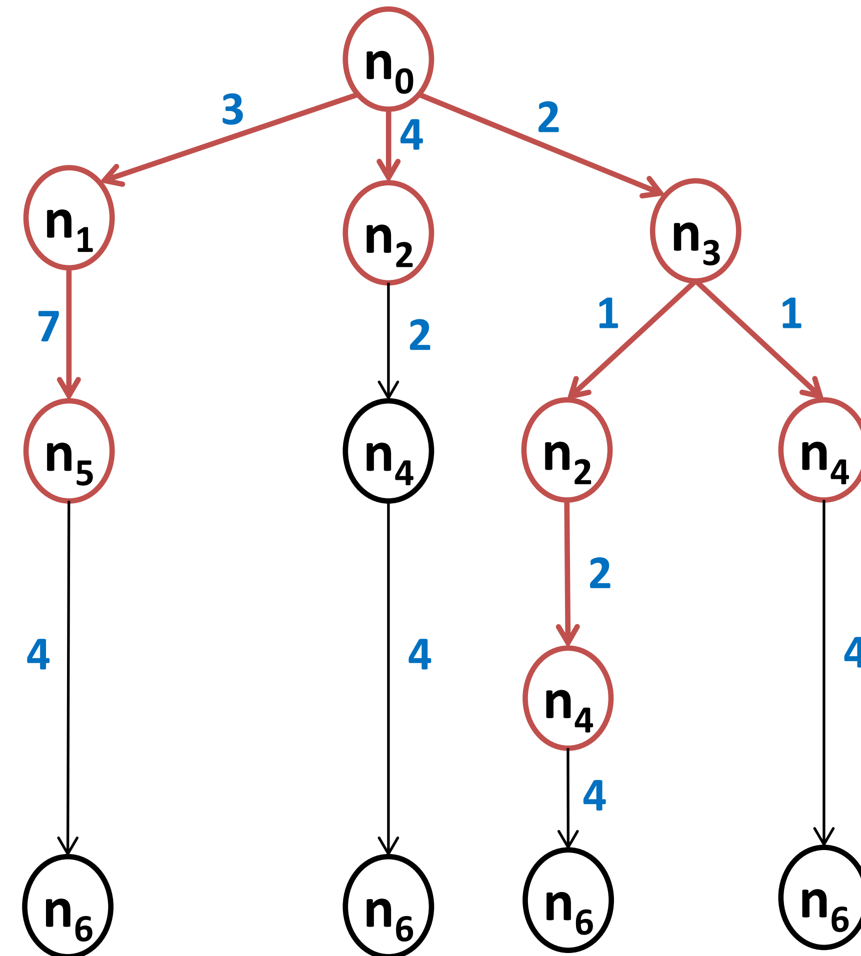
3. Enlever le premier élément de la liste **(n_3,2,n_0)** et rajouter ses successeurs **n_2, n_4** à la liste d'états en respectant l'ordre croissant.
On obtient : **[(n_1,3, n_0), (n_2,3, n_3),(n_4,3, n_3), (n_2,4, n_0)]**

4. Enlever le premier élément de la liste **(n_1,3,n_0)** et rajouter ses successeurs **n_5** à la liste d'états en respectant l'ordre croissant.
On obtient : **[(n_2,3, n_3),(n_4,3, n_3), (n_2,4, n_0),(n_5,10, n_1)]**



Algorithme de recherche : A coût uniforme

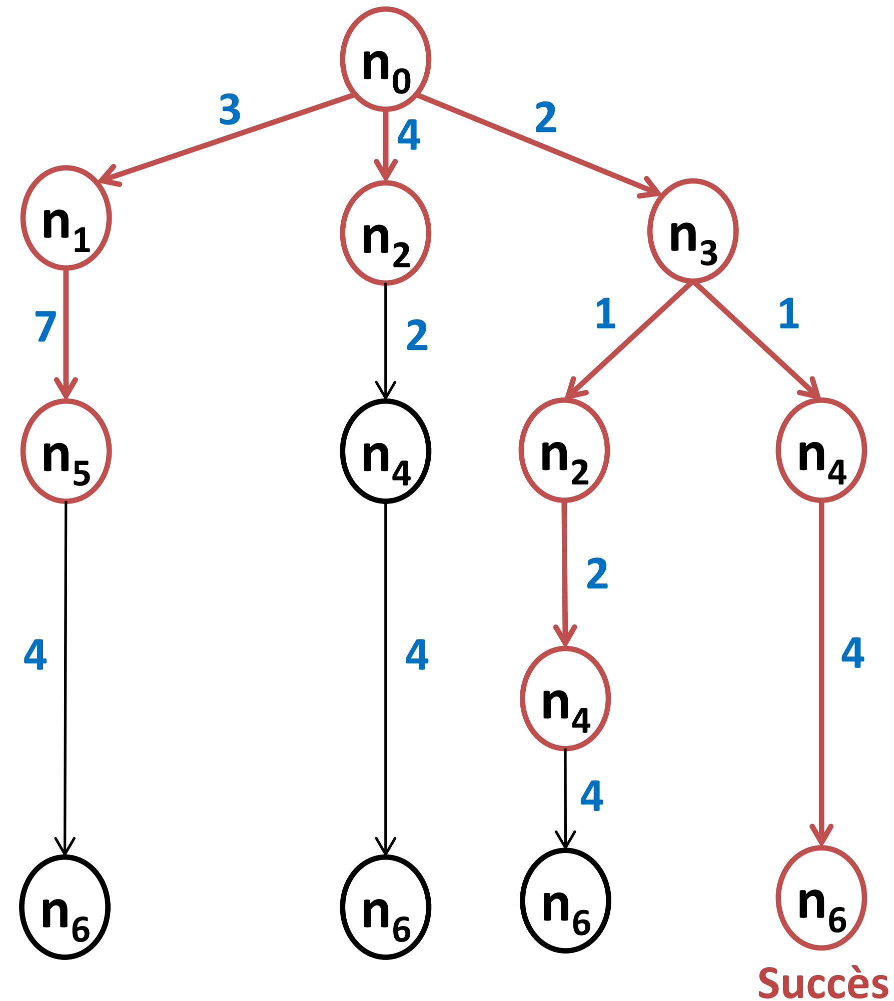
5. Enlever le premier élément de la liste **(n_2,3,n_3)** et rajouter ses successeurs **n_4** à la liste d'états en respectant l'ordre croissant.
On obtient : **[(n_4,3,n_3), (n_2,4, n_0), (n_4,5,n_2),(n_5,10,n_1)]**



Algorithme de recherche : A coût uniforme

5. Enlever le premier élément de la liste **(n_2,3,n_3)** et rajouter ses successeurs **n_4** à la liste d'états en respectant l'ordre croissant.
On obtient : **[(n_4,3,n_3), (n_2,4, n_0), (n_4,5,n_2),(n_5,10,n_1)]**

6. Enlever le premier élément de la liste **(n_4,3,n_3)** et rajouter ses successeurs **n_6** à la liste d'états en respectant l'ordre croissant.
On obtient : **[(n_6,7,n_4),(n_5,10,n_1)]**



n_6 Apparait dans Open

alors arrêter

Le chemin à coût optimal est : n_0,n_3,n_4, n_6

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

Algorithme de recherche : Heuristique

Meilleur d'abord : Best-first search

1. Démarrer la recherche avec la liste contenant l'**état initial** du problème.

2. Si la liste n'est pas vide alors :
   - Choisir un état **n** de mesure **minimale** à traiter.
   - Si **n** est un **état final** alors retourner succès
   - Sinon, rajouter tous les successeurs de **n** à la liste d'états à traiter par **ordre croissant** selon la mesure d'utilité.

   Recommencer au point 2.

3. Sinon retourner échec.

Algorithme de recherche : Heuristique

Meilleur d'abord : Best-first search

Cas particulier 1 : Recherche gloutonne (greedy best-first search**)**

- La mesure d'utilité est donnée par une fonction d'estimation ***h***.
- Pour chaque état ***n***, ***h*(*n*)** représente l'estimation du coût de ***n*** vers un état final.

Par exemple, dans le problème du chemin le plus court entre deux villes on peut prendre ***h*(*n*) = distance directe** entre ***n*** et la **ville destination**.

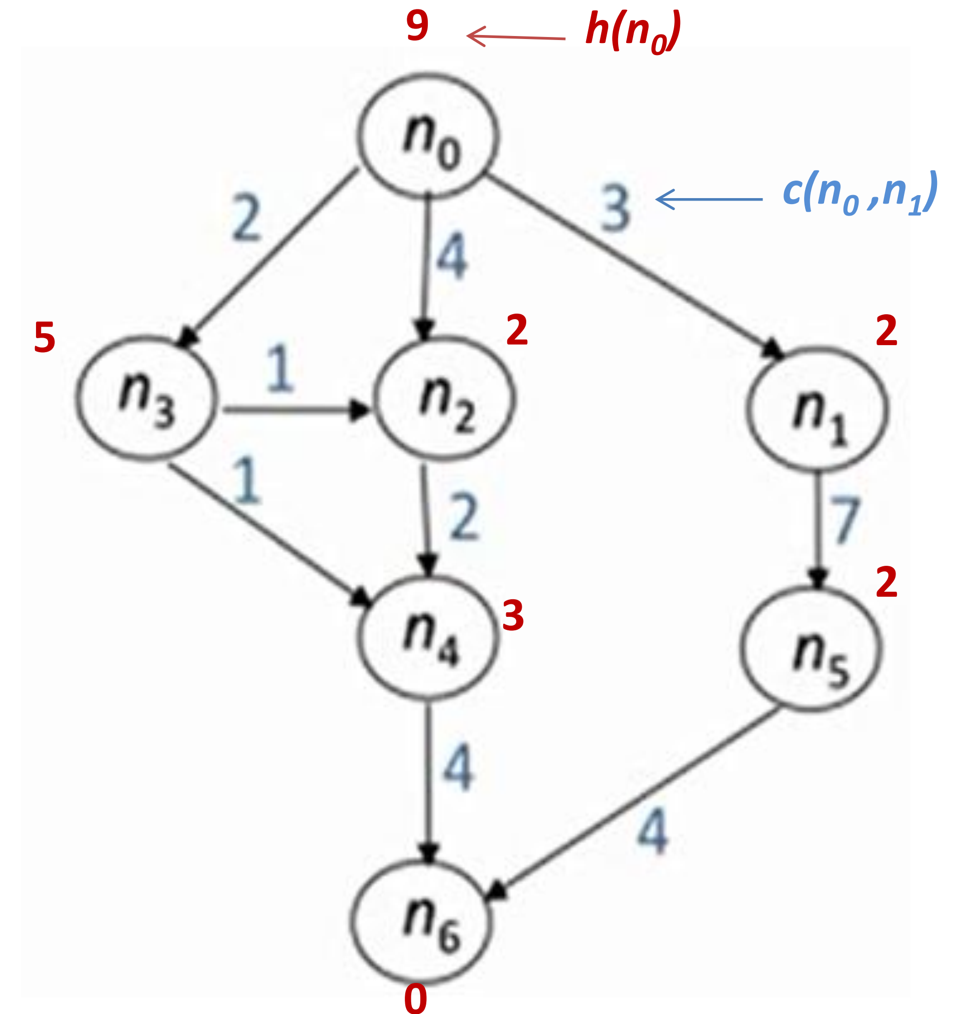
- La recherche gloutonne choisira l'état qui semble le plus proche d'un état final selon la fonction d'estimation.

Cas particulier 1 : Recherche gloutonne (greedy best-first search)

Etat de la liste Open :

- (n_0,9,void)
- (n_2,2,n_0), (n_1,2,n_0), (n_3,5,n_0)
- (n_1,2,n_0), (n_4,3,n_2), (n_3,5,n_0)
- (n_5,2,n_1), (n_4,3,n_2), (n_3,5,n_0)
- (n_6,0,n_5), (n_4,3,n_2), (n_3,5,n_0)

Chemin : $n_0 \rightarrow n_1 \rightarrow n_5 \rightarrow n_6$



Algorithme de recherche : Heuristique

Meilleur d'abord : Best-first search

Cas particulier 2 : Algorithme A*

- On évite d'explorer les chemins qui sont déjà chers.
- La mesure d'utilité est donnée par une fonction d'évaluation ***f***.
- Pour chaque état ***n*** : ***f*(*n*) = *g*(*n*) + *h*(*n*)**, où :

***g*(*n*)** est le coût jusqu'à présent pour atteindre *n*

***h*(*n*)** est le coût estimé pour aller de *n* vers un état final.

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

h est dite admissible si pout tout *n* : $h(n) \leq c(n)$

$c(n)$ étant le coût réel menant de *n* vers l'état final

Algorithme A*

- Une heuristique ***h(n)*** est une fonction d'estimation du coût restant entre un nœud ***n*** d'un graphe et le nœud ***but***.

Exemples de fonctions d'heuristique h :

- Chemin entre deux villes :

▪ Distance à vol d'oiseau entre la ville n et la ville de destination.

- Jeu de taquin :

▪ Nombre de nombres mal placés

Algorithme de recherche : A*

1. Déclarer deux nœuds ***n***, ***ns***
2. Déclarer deux listes ***Open*** et ***Closed*** (vides au départ)
3. Ajouter le nœud ***Etat initial*** dans ***Open***.
4. Si ***Open*** est vide Alors sortir de la boucle avec un ***échec***.
5. ***n=nœud*** à la tête de ***Open***
6. Enlever ***n*** de ***Open*** et l'ajouter dans ***Closed***.
7. Si ***n=but*** alors sortir de la boucle et ***retourner le chemin***.
8. Sinon : pour chaque successeur ***ns*** de ***n*** :

   - Initialiser la valeur ***g(ns) = g(n) + c(n,ns)***
   - Mettre le parent de ***ns*** à ***n***
   - Si ***Open*** ou ***Closed*** contient un nœud ***ns'=ns*** avec ***f(ns) ≤ f(ns')***

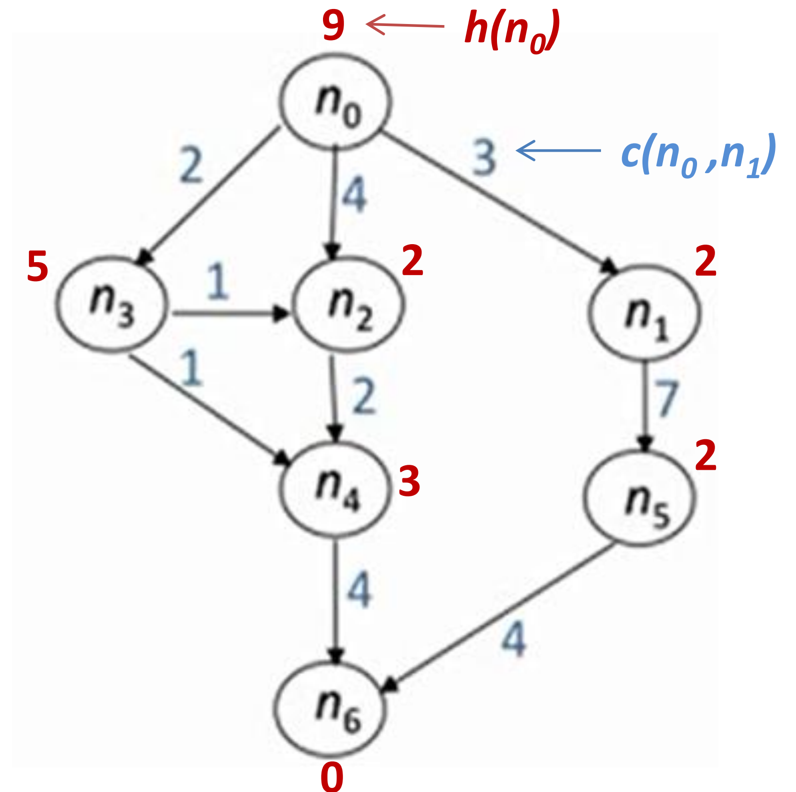
     Alors enlever ***ns'*** de ***Open*** ou ***Closed*** et insérer ***ns*** dans ***Open***

     (en respectant l'ordre croissant de ***f***)

     Sinon : Insérer ***ns*** dans ***Open*** (en respectant l'ordre croissant de ***f***)
   - Aller à 4.

Algorithme A* : Exemple de recherche d'un chemin entre deux villes

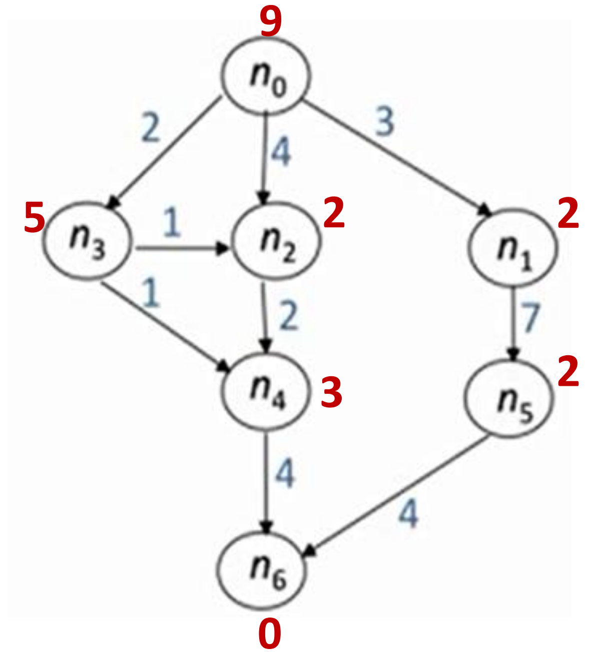
n_0 : ville de départ (nœud initial)
n_6 : ville de destination (nœud but)
h : Distance à vol d'oiseau entre une ville et la ville de destination(heuristique)
c : Distance réelle entre deux villes



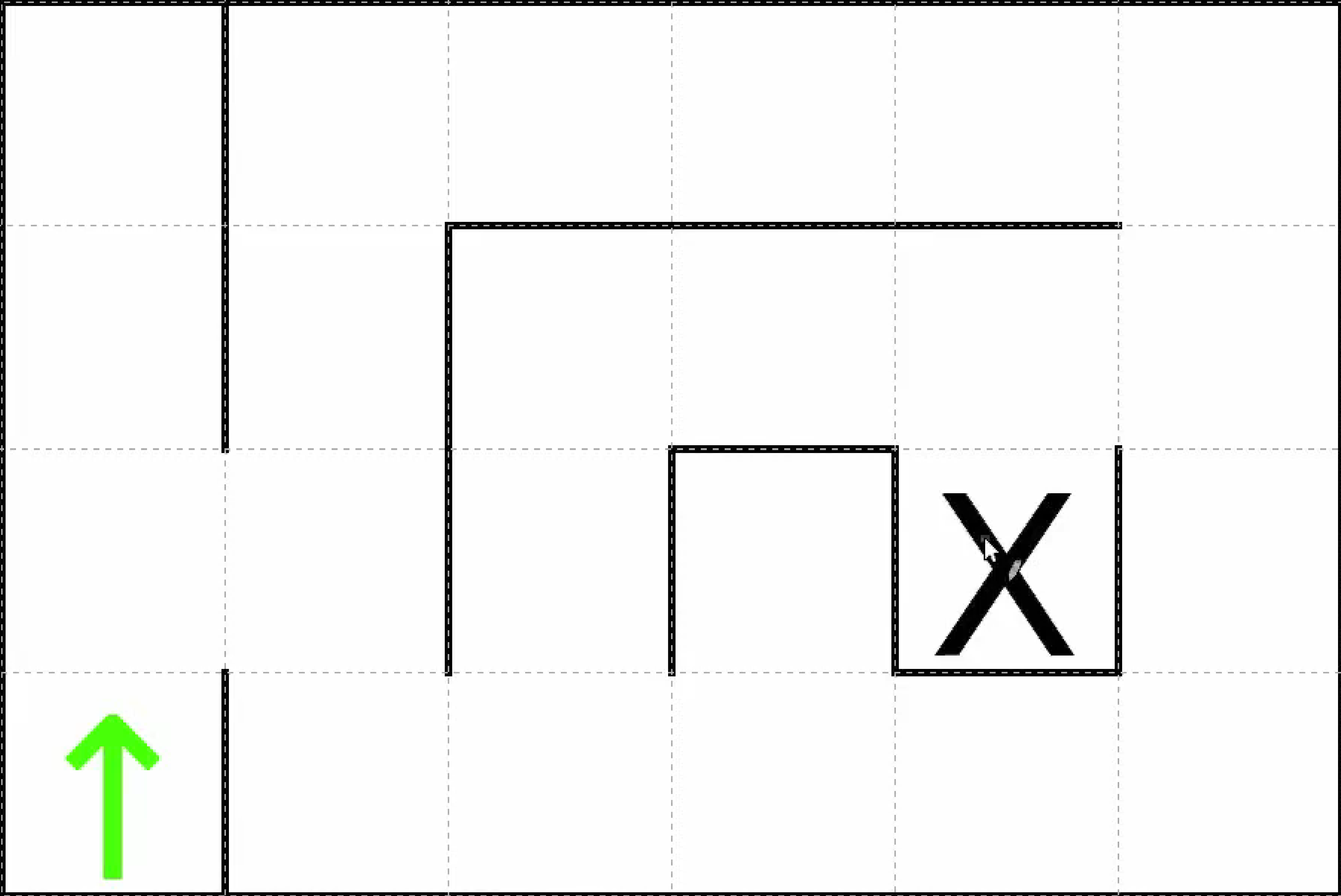
Algorithme de recherche : Heuristique

Algorithme A* : Chemin entre deux villes (Déroulement)

Contenu de *open* à chaque itération (état, f, parent) :	Contenu de *closed* à chaque itération :
1. (n_0, 9, void)	1. Vide
2. (n_1,5,n_0), (n_2,6,n_0), (n_3,7,n_0)	2. (n_0, 9, void)
3. (n_2,6,n_0), (n_3,7,n_0), (n_5,12,n_1)	3. (n_0, 9, void), (n_1,5,n_0)
4. (n_3,7,n_0), (n_4,9,n_2), (n_5,12,n_1)	4. (n_0, 9, void), (n_1,5,n_0), (n_2,6,n_0)
5. (n_2,5,n_3), (n_4,6,n_3), (n_5,12,n_1)	5. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0)
6. (n_4,6,n_3), (n_5,12,n_1)	6. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3)
7. (n_6,7,n_4), (n_5,12,n_1)	7. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3)
8. Solution : $\boldsymbol{n_0,n_3,n_4,n_6}$	8. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3), (n_6,7,n_4)



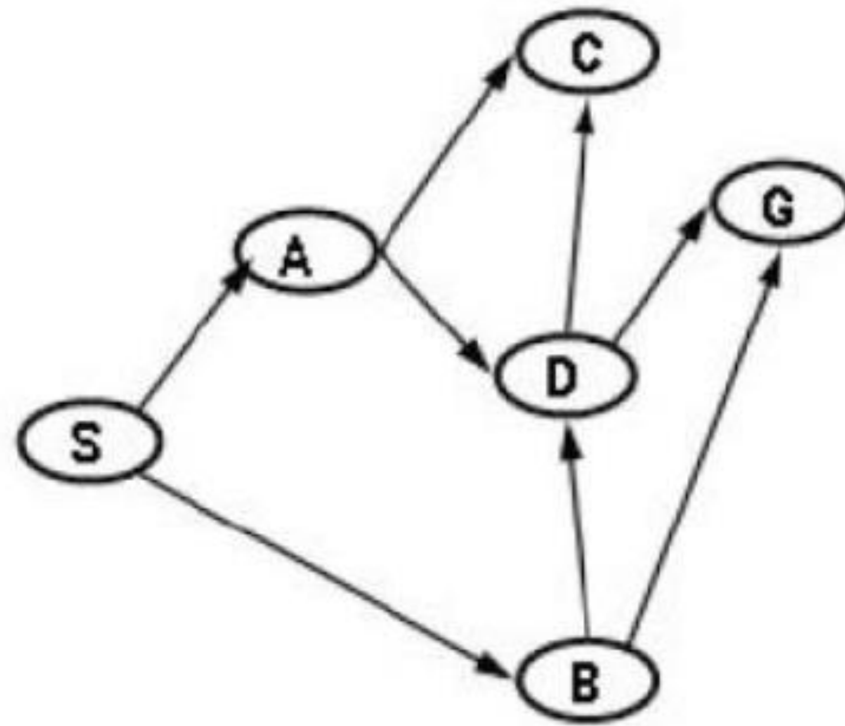
2011/08/02- 01:08:27 -Vico Tag2:TMIRC101


X

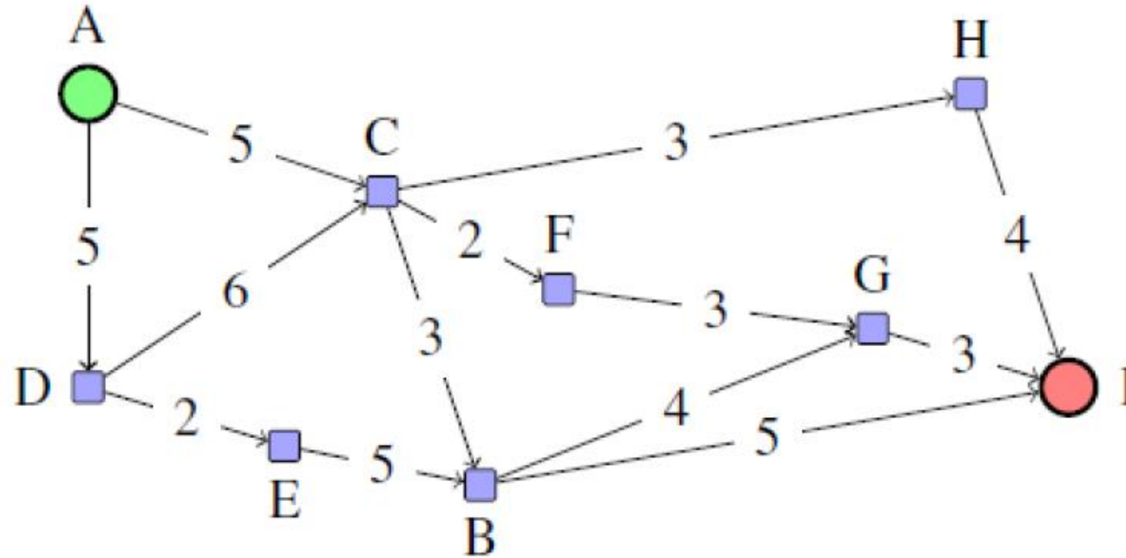
Exercice 1 : Soit l'arbre de recherche suivant :

Numérotez les différents états de l'arbre selon leurs ordres de parcours en utilisant les deux stratégies Largeur d'abord ensuite Profondeur d'abord.

Exercice 2 : Transformer le graphe d'états suivant en un arbre de recherche ensuite appliquer une recherche en largeur d'abord puis en profondeur d'abord pour chercher l'état **G** à partir de **S.**



Exercice 3 : On considère la carte suivante. L'objectif est de trouver le chemin optimal entre **A** et **I**. On donne également deux heuristiques *h1* et *h2* :



Nœud	A	B	C	D	E	F	G	H	I
h1	10	5	5	10	10	3	3	3	0
h2	10	2	8	11	6	2	1	5	0

1. Appliquez les stratégies Largeur d'abord, Profondeur d'abord, A coût uniforme
2. Appliquez un algorithme glouton en utilisant *h2* puis un algorithme A* en utilisant *h1*

(Donnez la suite des nœuds développés (l'état de la liste OPEN à chaque itération), déduire alors le chemin traversé dans les différents cas **?**)