Intelligence Artificielle

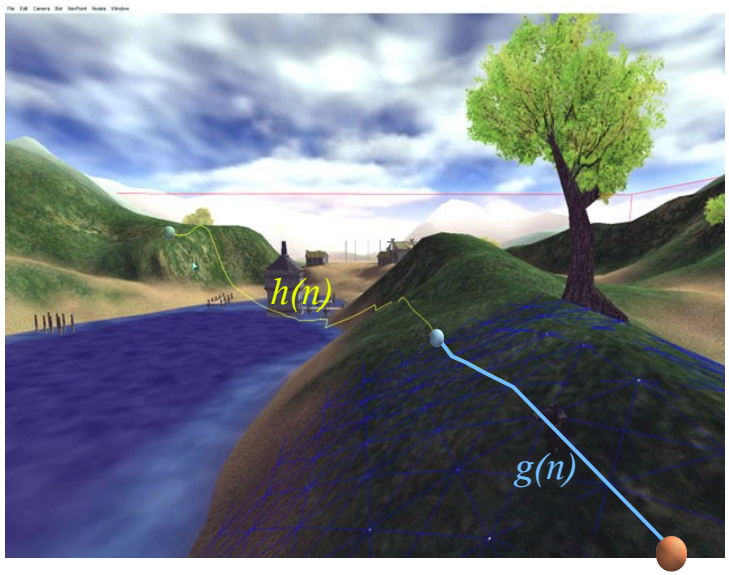
Algorithmes de Recherche

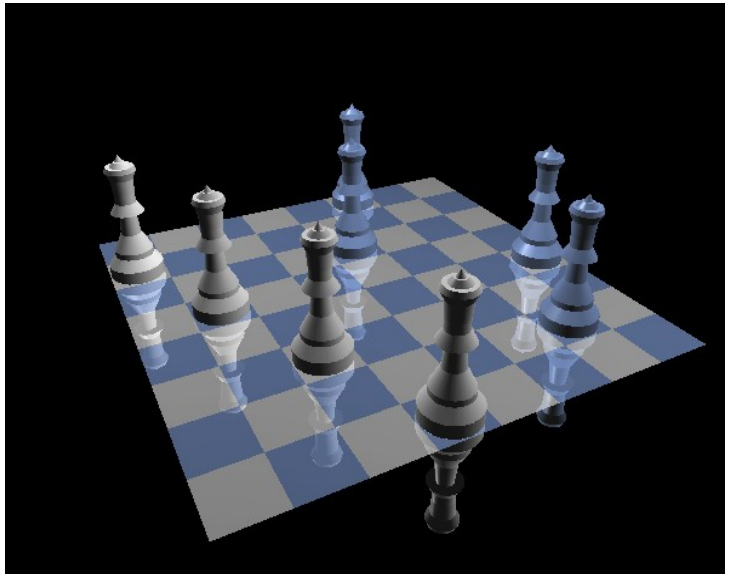
Fred Koriche

koriche@lirmm.fr

Plan

1. Résolution de Problèmes
2. Recherche Arborescente
3. Recherche Heuristique





Résolution de Problèmes

Problème

Etats

Espace de situations S

Actions

Ensemble d'actions A
Fonction d'expansion **Expand(s)**

Etat initial

Situation s_0

Coût

Fonction de test **Goal(s)**

But

Fonction de test **Goal(s)**

Solution

Solution Simple

Un chemin depuis l'état initial jusqu'à un état but

Solution Optimale

Une solution minimisant la fonction de coût

Résolution de Problèmes

Pathfinding

Etats

Graphe des points de la carte

Actions

Arêtes sortantes d'un point

Etat initial

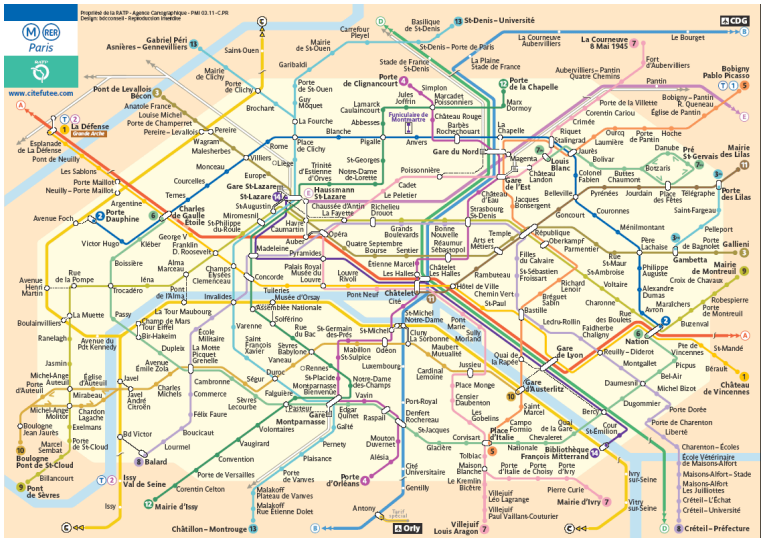
Un point arbitraire

Coût

Selon le problème:
distance, durée, difficulté, danger,

But

Sommet final ou région finale



Résolution de Problèmes

Taquin

Etats

Un état est une localisation de chaque case

Actions

Droite, gauche, haut, bas

Etat initial

Un état arbitraire

Coût

1 point par action

But

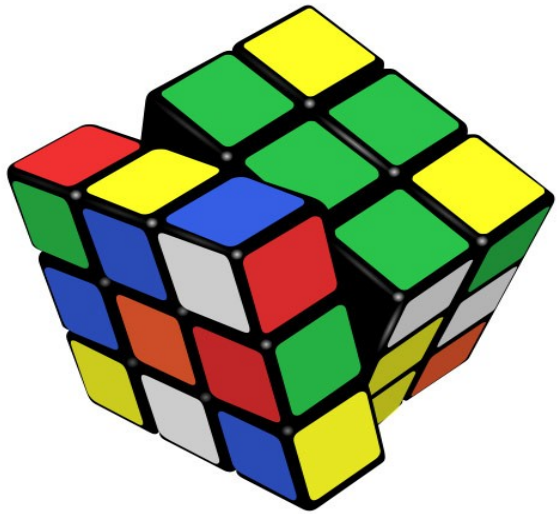
Un état arbitraire

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State



Résolution de Problèmes

Question 1

Modéliser les N reines

Résolution de Problèmes

N-Reines

Etats

Arrangement de n reines, 0 ≤ n ≤ N, sur les n colonnes de gauche

Actions

Ajout d'une reine sur la colonne vide la plus à gauche

Etat initial

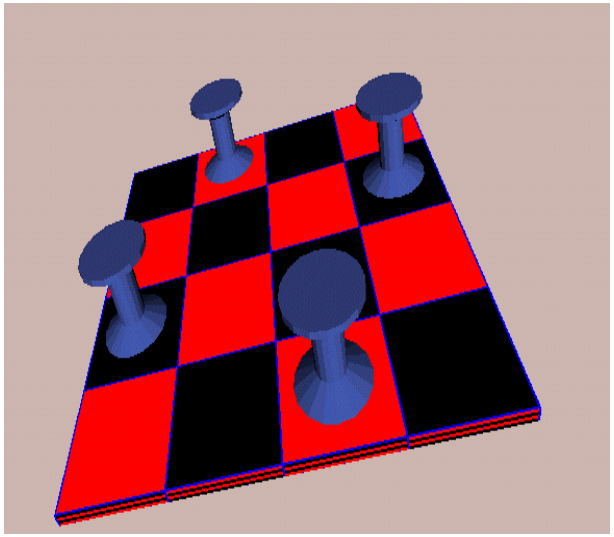
Echiquier vide

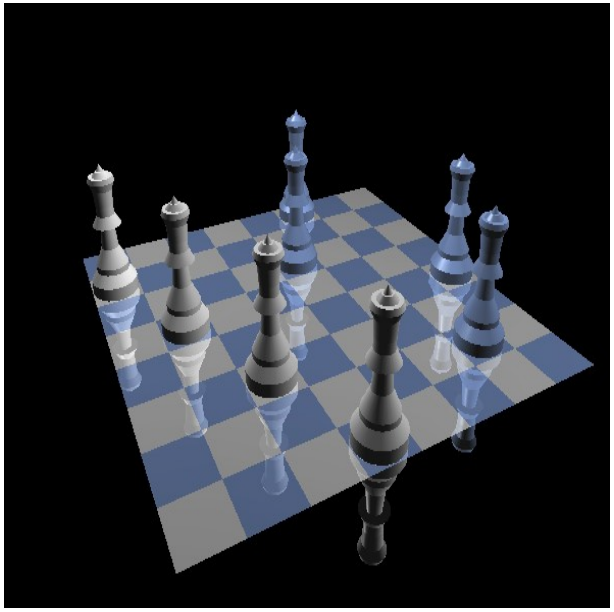
Coût

1 point par action

But

N reines non en echec





Recherche Arborescente

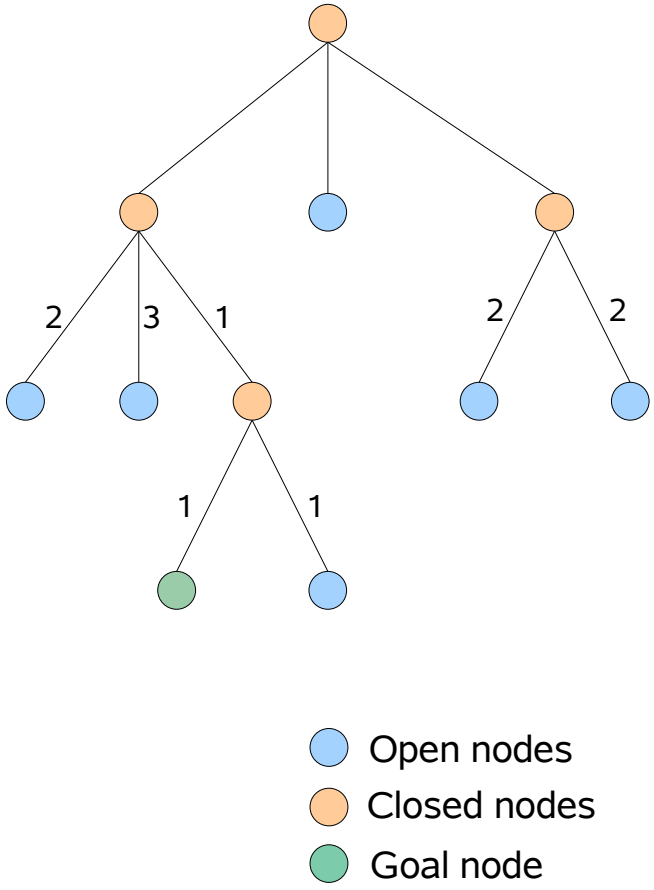
Modèle

Structure de Données

Racine: état initial
Noeuds: états
Arêtes: actions admissibles sur le noeud parent

Algorithme

Explorer l'arbre de recherche en **ouvrant** itérativement les noeuds



Recherche Arborescente

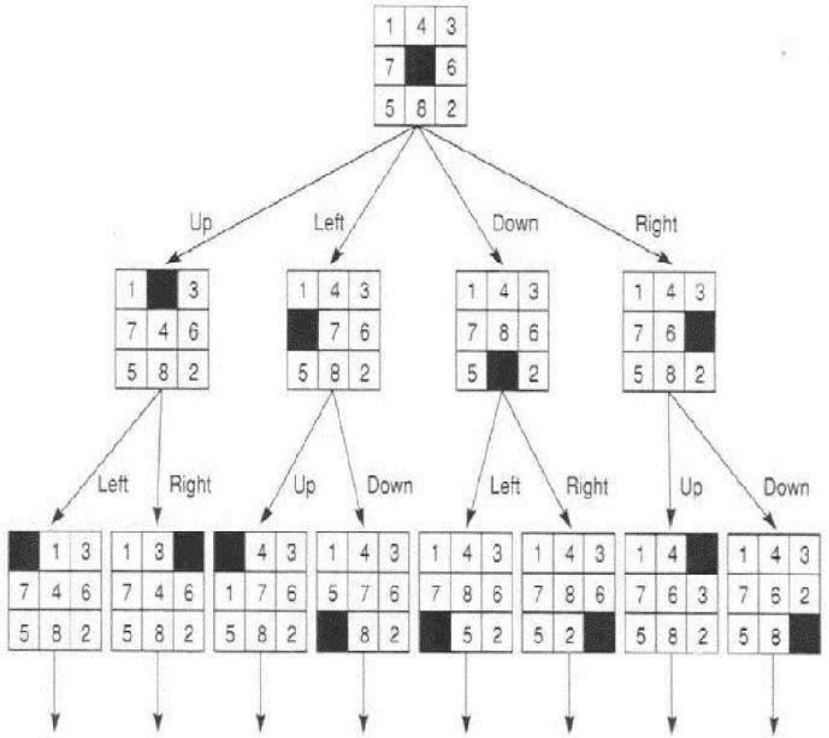
Modèle

Structure de Données

Racine: état initial
Noeuds: états
Arêtes: actions admissibles sur le noeud parent

Algorithme

Explorer l'arbre de recherche en **ouvrant** itérativement les noeuds



Recherche Arborescente

Mesures de Complexité

Branchement

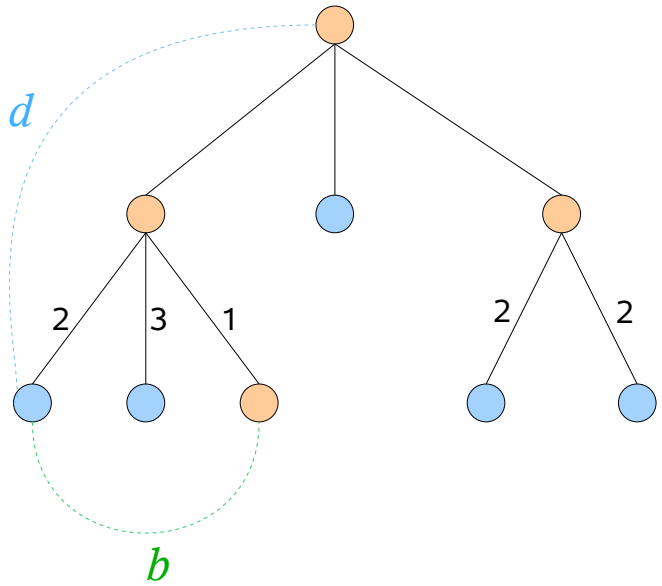
Largeur maximale d'ouverture sur un noeud

Profondeur

Longueur maximale d'un chemin dans l'arbre

Complexité et Etats

Si l'arbre explore tous les états possibles
sa complexité est en $\Omega(N)$



Recherche Arborescente

Tree Search

Entrée: un problème avec noeud initial x

Sortie: chemin depuis x jusqu'au but

Algorithme:

opens ← MakeList(x)
loop do
    if *opens* is empty **then return** failure
    node ← RemoveFront(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← Insert(*opens*, Expand(*node*))
end

MakeList(node)

Construire une liste avec le noeud initial

RemoveFront(node)

Enlever le premier noeud de la liste

set ← Expand(node)

Ouvrir un noeud, c'est-à-dire appliquer tous les actions possibles sur le noeud et retourner les états

list ← Insert(list, set)

Insérer dans la liste « list » l'ensemble de noeuds « set »

Recherche Arborescente

Breast-First Search

Utiliser la liste comme une file (FIFO)

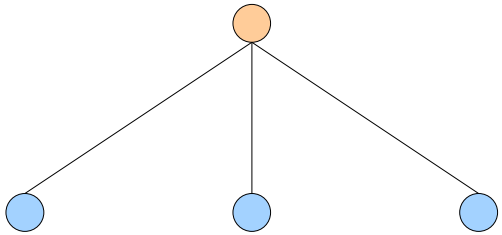
opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertLast(*opens*, Expand(*node*))
end

Recherche Arborescente

Breast-First Search

Utiliser la liste comme une file (FIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertLast(*opens*, Expand(*node*))
end

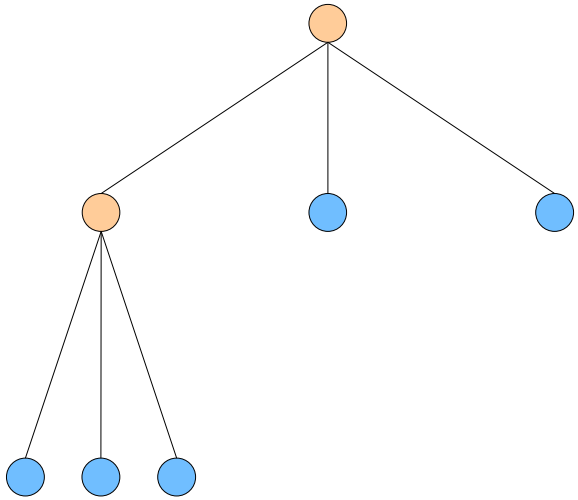


Recherche Arborescente

Breast-First Search

Utiliser la liste comme une file (FIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertLast(*opens*, Expand(*node*))
end

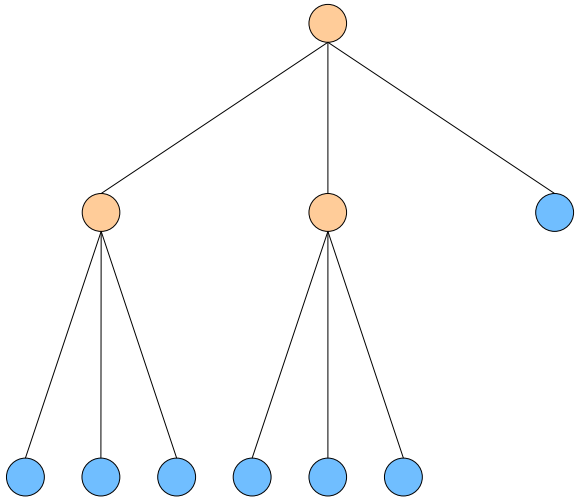


Recherche Arborescente

Breast-First Search

Utiliser la liste comme une file (FIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertLast(*opens*, Expand(*node*))
end

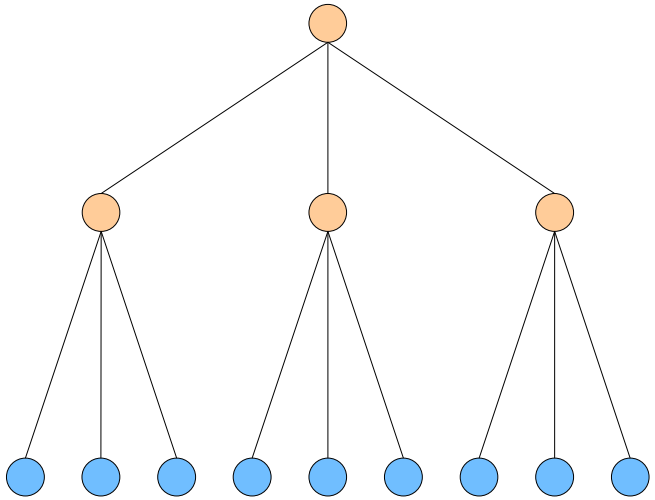


Recherche Arborescente

Breast-First Search

Utiliser la liste comme une file (FIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertLast(*opens*, Expand(*node*))
end

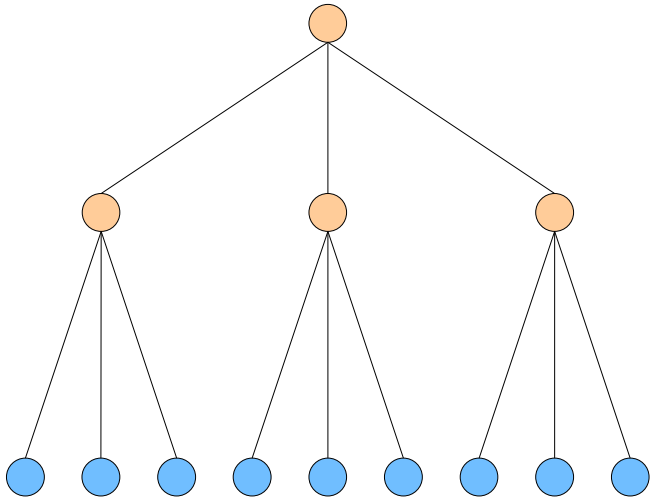


Recherche Arborescente

Breast-First Search

Utiliser la liste comme une file (FIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertLast(*opens*, Expand(*node*))
end



Complexité

Spatiale: $O(b^d)$
Temporelle: $O(b^d)$

Recherche Arborescente

Depth-First Search

Utiliser la liste comme une pile (LIFO)

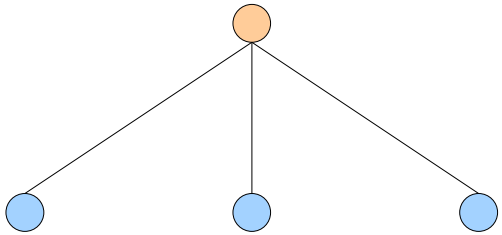
```
opens ← MakeList(x)
loop do
    if opens is empty then return failure
    node ← Pop(opens)
    if Goal(node) then return Path(node)
    opens ← InsertFirst(opens, Expand(node))
end
```

Recherche Arborescente

Depth-First Search

Utiliser la liste comme une pile (LIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertFirst(*opens*, Expand(*node*))
end

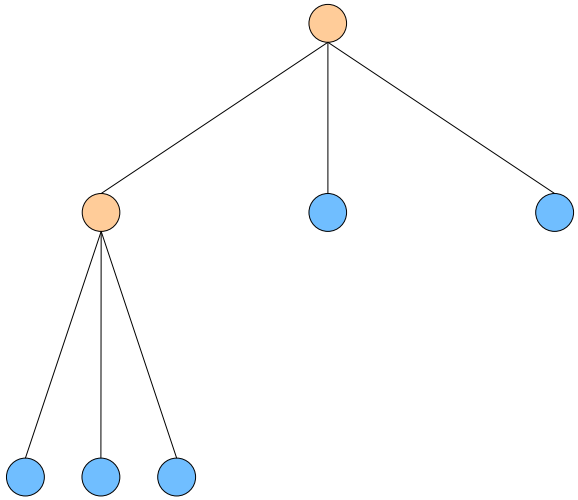


Recherche Arborescente

Depth-First Search

Utiliser la liste comme une pile (LIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertFirst(*opens*, Expand(*node*))
end

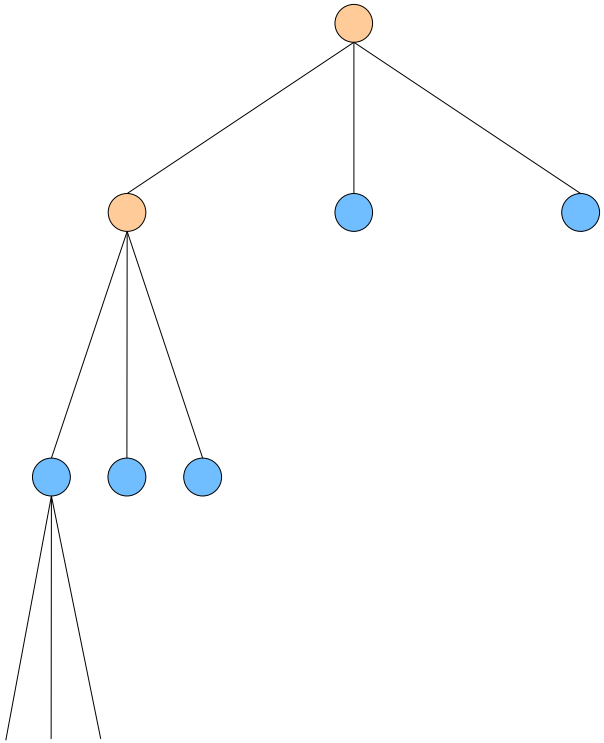


Recherche Arborescente

Depth-First Search

Utiliser la liste comme une pile (LIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertFirst(*opens*, Expand(*node*))
end

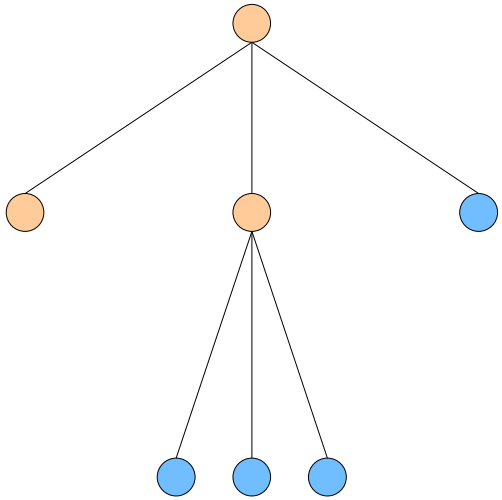


Recherche Arborescente

Depth-First Search

Utiliser la liste comme une pile (LIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertFirst(*opens*, Expand(*node*))
end

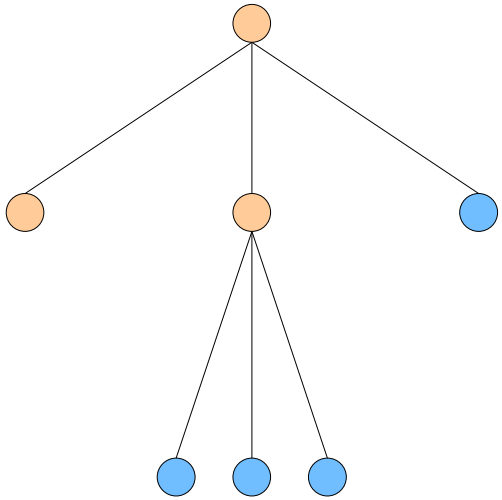


Recherche Arborescente

Depth-First Search

Utiliser la liste comme une pile (LIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertFirst(*opens*, Expand(*node*))
end

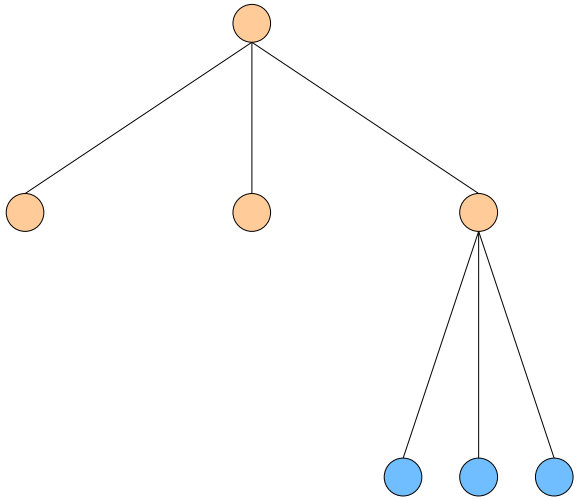


Recherche Arborescente

Depth-First Search

Utiliser la liste comme une pile (LIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertFirst(*opens*, Expand(*node*))
end

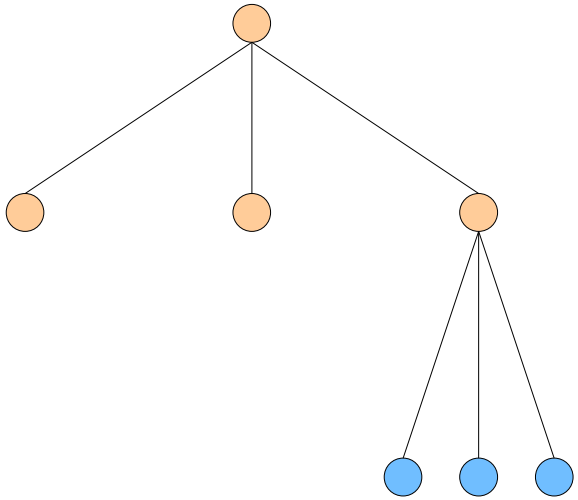


Recherche Arborescente

Depth-First Search

Utiliser la liste comme une pile (LIFO)

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertFirst(*opens*, Expand(*node*))
end



Complexité

Spatiale: $O(bd)$
Temporelle: $O(b^d)$

Recherche Arborescente

Question 2

Est-ce qu'un espace fini d'états donne toujours un arbre de recherche fini ?

Question 3

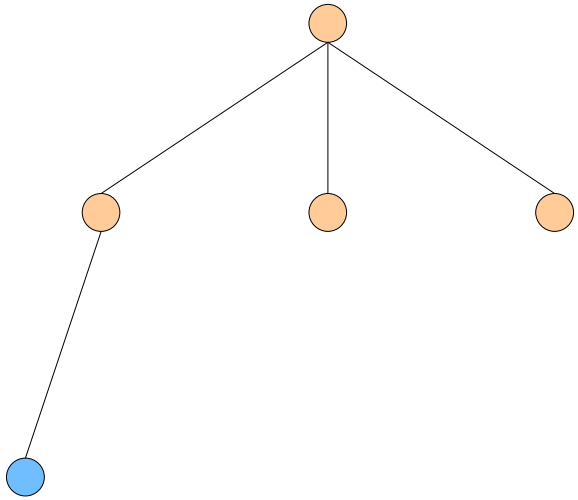
Comment éviter une recherche infinie dans depth-first-search ?

Recherche Arborescente

Backtracking Search

Ne rajouter qu'un ouvert à la fois

opens ← Expand(*node*)
if *opens* is empty **then return** failure
for each *child* **in** *opens*
    if Goal(*child*) **then**
        return Path(*child*)
    else
        return Backtrack(*child*)

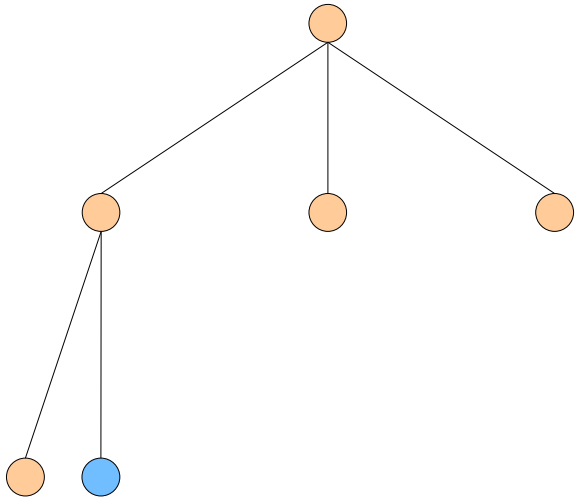


Recherche Arborescente

Backtracking Search

Ne rajouter qu'un ouvert à la fois

opens ← Expand(*node*)
if *opens* is empty **then return** failure
for each *child* **in** *opens*
    if Goal(*child*) **then**
        return Path(*child*)
    else
        return Backtrack(*child*)

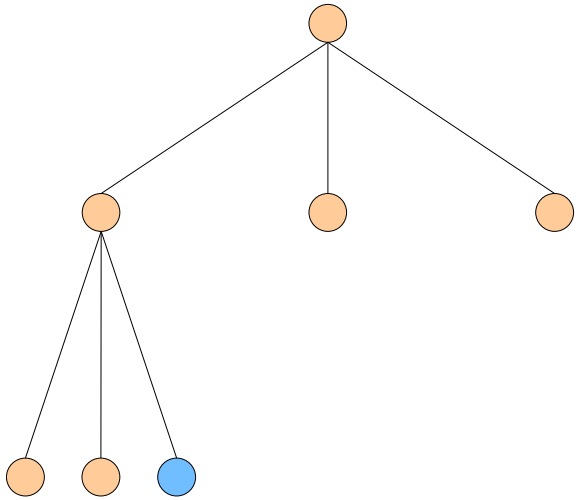


Recherche Arborescente

Backtracking Search

Ne rajouter qu'un ouvert à la fois

opens ← Expand(*node*)
if *opens* is empty **then return** failure
for each *child* **in** *opens*
    if Goal(*child*) **then**
        return Path(*child*)
    else
        return Backtrack(*child*)

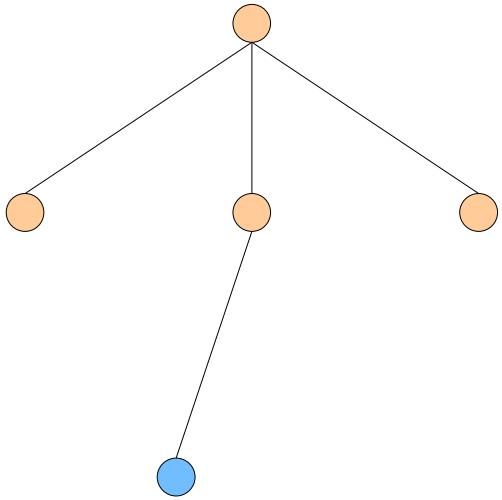


Recherche Arborescente

Backtracking Search

Ne rajouter qu'un ouvert à la fois

opens ← Expand(*node*)
if *opens* is empty **then return** failure
for each *child* **in** *opens*
    if Goal(*child*) **then**
        return Path(*child*)
    else
        return Backtrack(*child*)

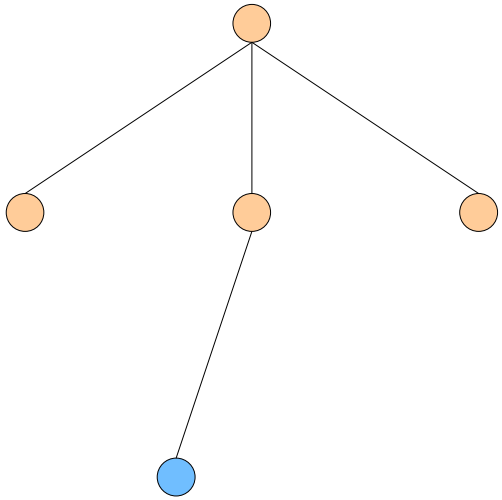


Recherche Arborescente

Backtracking Search

Ne rajouter qu'un ouvert à la fois

opens ← Expand(*node*)
if *opens* is empty **then return** failure
for each *child* **in** *opens*
    if Goal(*child*) **then**
        return Path(*child*)
    else
        return Backtrack(*child*)



Complexité

Spatiale: $O(d)$
Temporelle: $O(b^d)$

Recherche Arborescente

Best-First Search

Ouvrir toujours le noeud le moins coûteux

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertBest(*opens*, Expand(*node*))
end

InsertBest

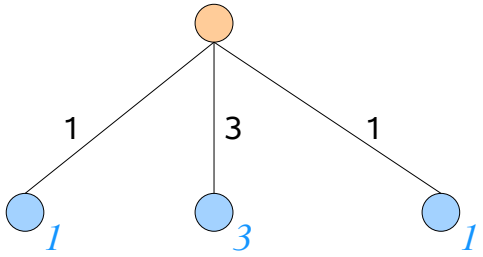
for each node **in** set **do**
    SortedInsert(opens, node, *g(node)*)

Recherche Arborescente

Best-First Search

Ouvrir toujours le noeud le moins coûteux

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertBest(*opens*, Expand(*node*))
end

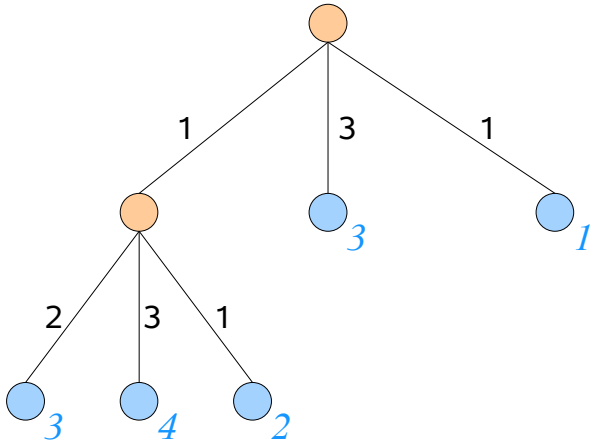


Recherche Arborescente

Best-First Search

Ouvrir toujours le noeud le moins coûteux

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertBest(*opens*, Expand(*node*))
end

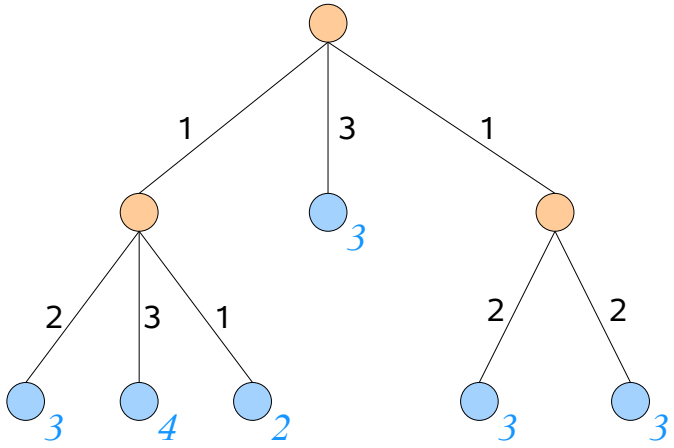


Recherche Arborescente

Best-First Search

Ouvrir toujours le noeud le moins coûteux

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertBest(*opens*, Expand(*node*))
end



Recherche Arborescente

Question 5

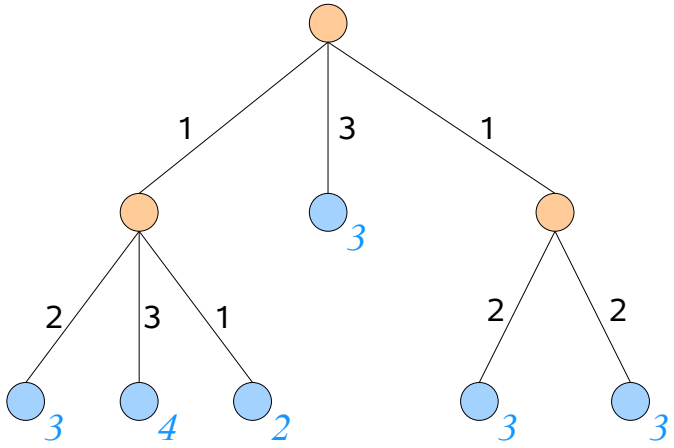
Quelle est la complexité spatiale de Best-First Search ?

Recherche Arborescente

Best-First Search

Ouvrir toujours le noeud le moins coûteux

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertBest(*opens*, Expand(*node*))
end



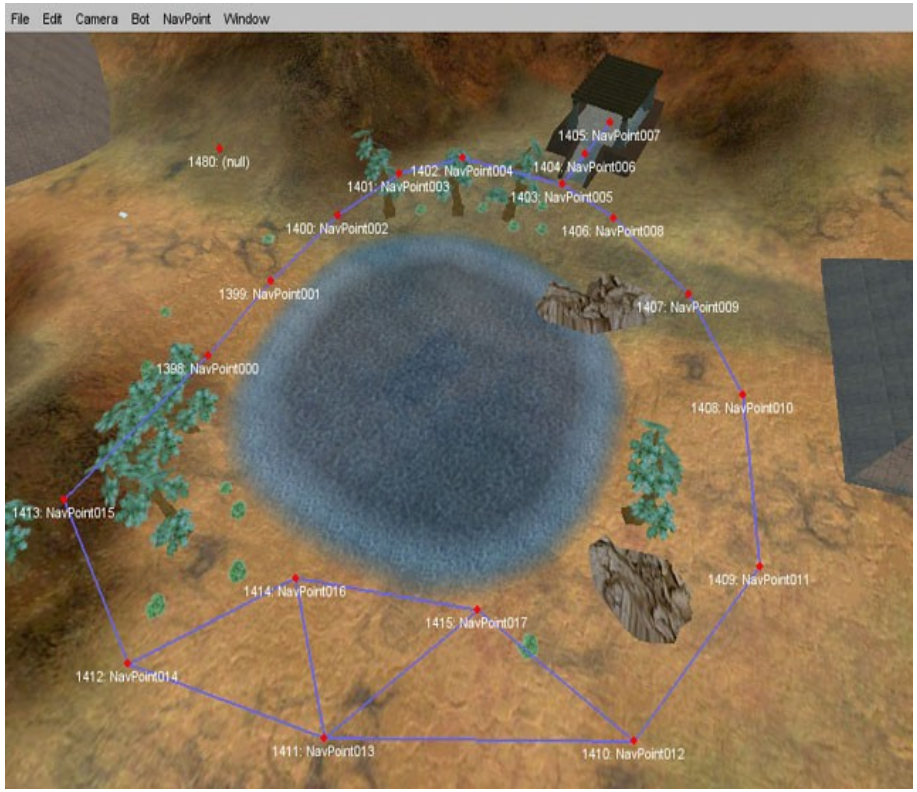
Complexité

Spatiale: $O(b^d)$
Temporelle: $O(b^d)$

Recherche Arborescente

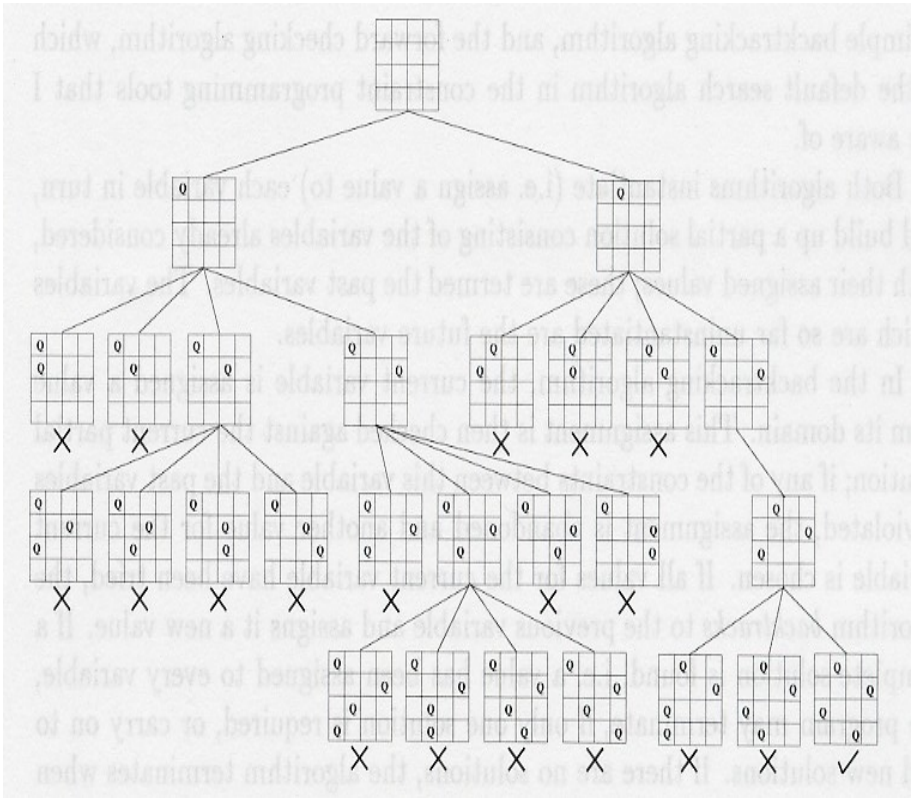
Best-First Search

Recherche de chemins dans les cartes ou jeux lorsque le graphe est **connu** de l'agent



Backtracking Search

Algorithme de base dans la recherche combinatoire (SAT, CSP, ...)



Recherche Heuristique

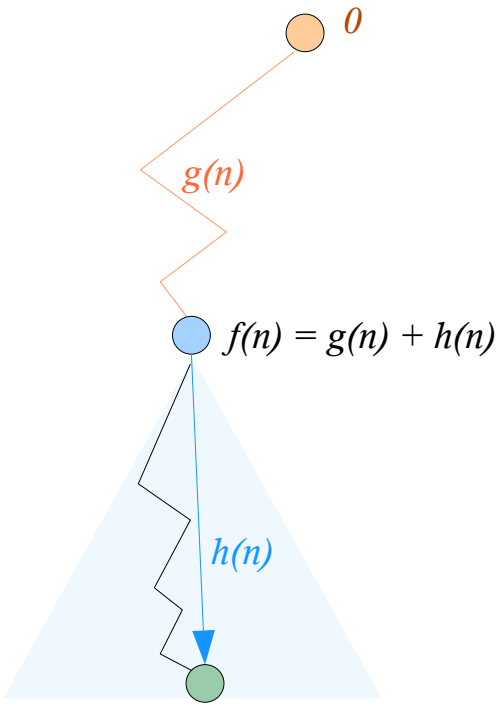
A* Search

Ouvrir toujours le noeud ***estimé*** *le moins coûteux selon* $f = g + h$

opens ← MakeList(*x*)
loop do
    if *opens* is empty **then return** failure
    node ← Pop(*opens*)
    if Goal(*node*) **then return** Path(*node*)
    opens ← InsertHeuristic(*opens*, Expand(*node*))
end

InsertHeuristic

for each node **in** set **do**
    SortedInsert(opens, node, *f(node)*)

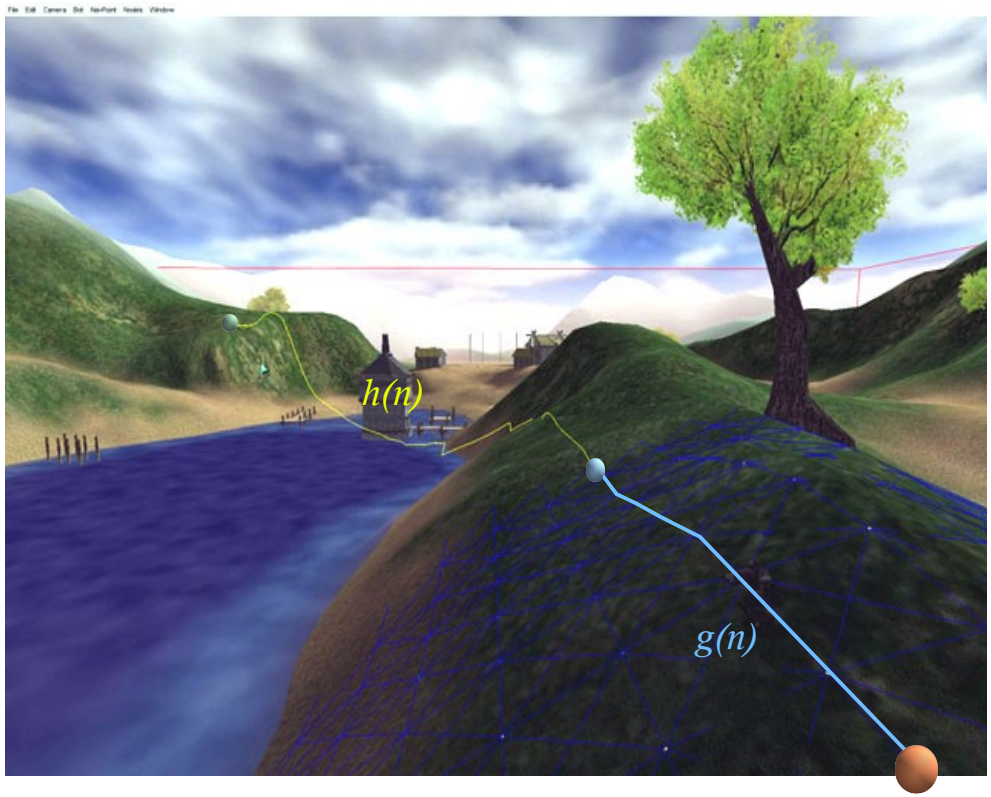


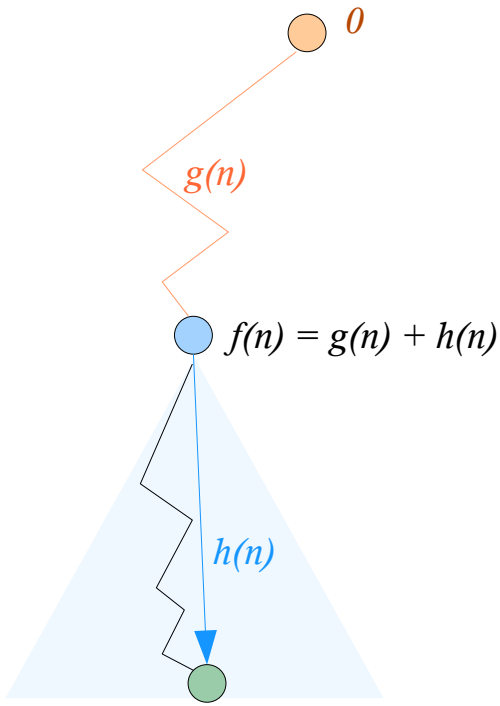
g(n) : coût réel; somme des coûts du noeud *x* à *n*
h(n) : coût heuristique; estimation du coût de *n* à *y*

Recherche Heuristique

A*Search

Recherche de chemins le graphe apparaît **progressivement** à l'agent





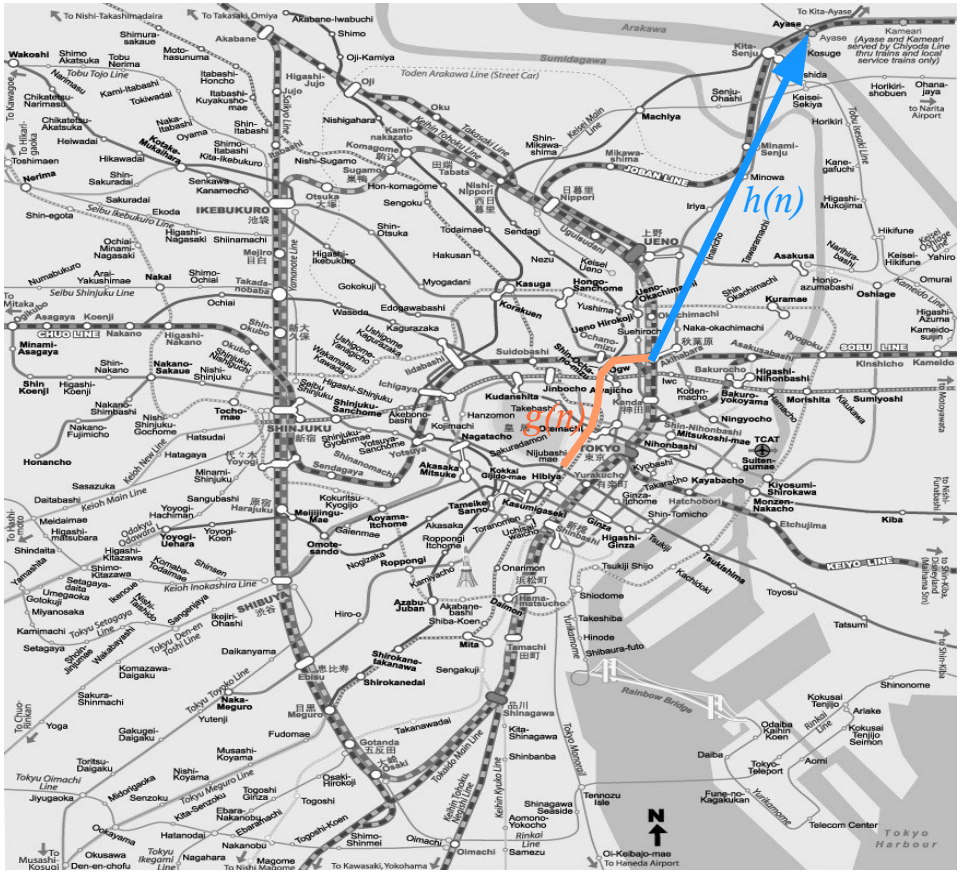
g(n) : coût réel; somme des coûts du noeud *x* à *n*

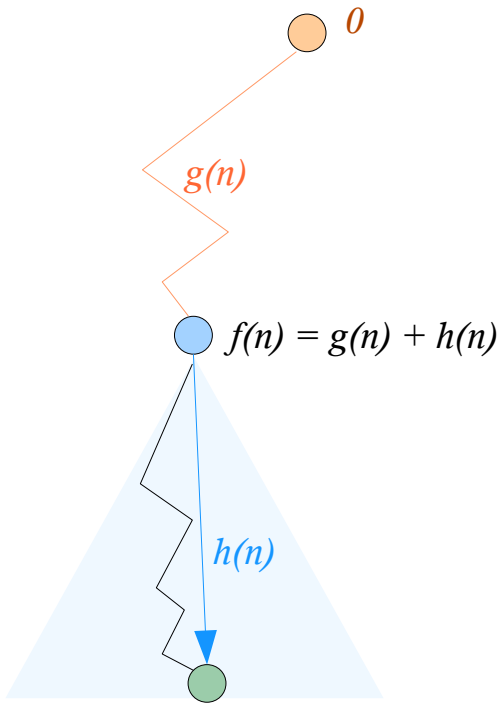
h(n) : coût heuristique; estimation du coût de *n* à *y*

Recherche Heuristique

A*Search

Recherche progressive de chemins sur de très grands graphes





$g(n)$: coût réel; somme des coûts du noeud x à n

$h(n)$: coût heuristique; estimation du coût de n à y

Recherche Heuristique

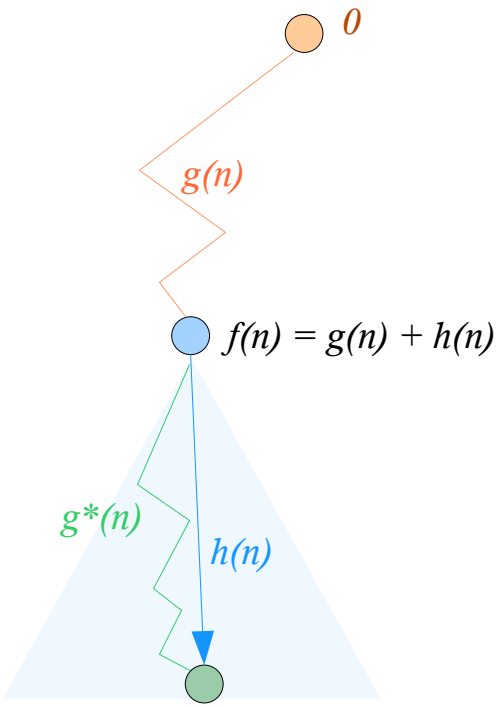
Optimalité

Heuristique admissible: $h(n) \leq g^*(n)$

Si h est une heuristique admissible, alors A* search est optimal sur la fonction de coût

Complexité

Spatiale: $O(b^d)$
Temporelle: $O(b^d)$



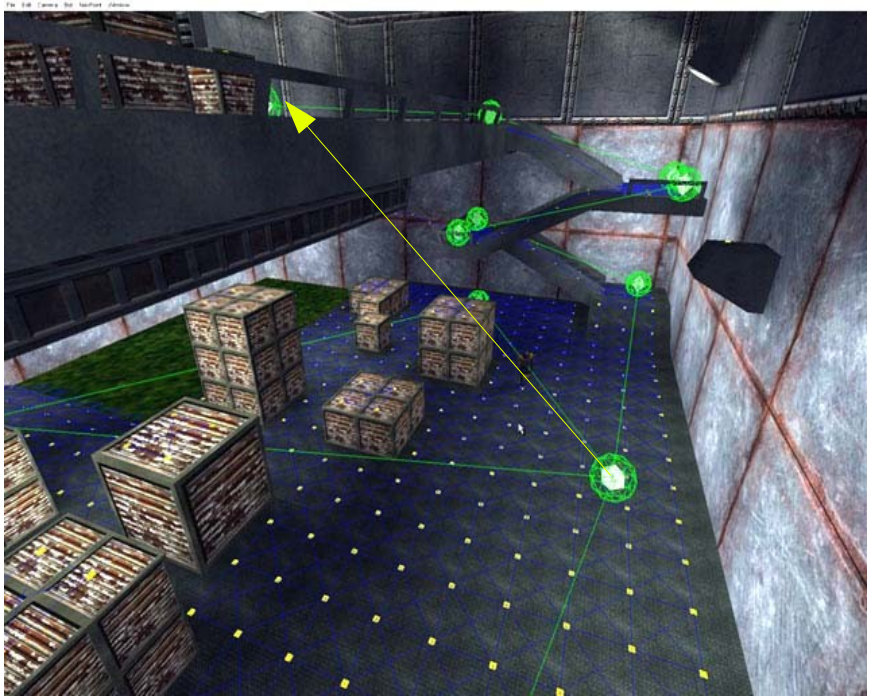
$g(n)$: coût réel; somme des coûts du noeud x à n
$g^*(n)$: coût réel; somme des coûts du noeud n à y
$h(n)$: coût heuristique; estimation du coût de n à y

Recherche Heuristique

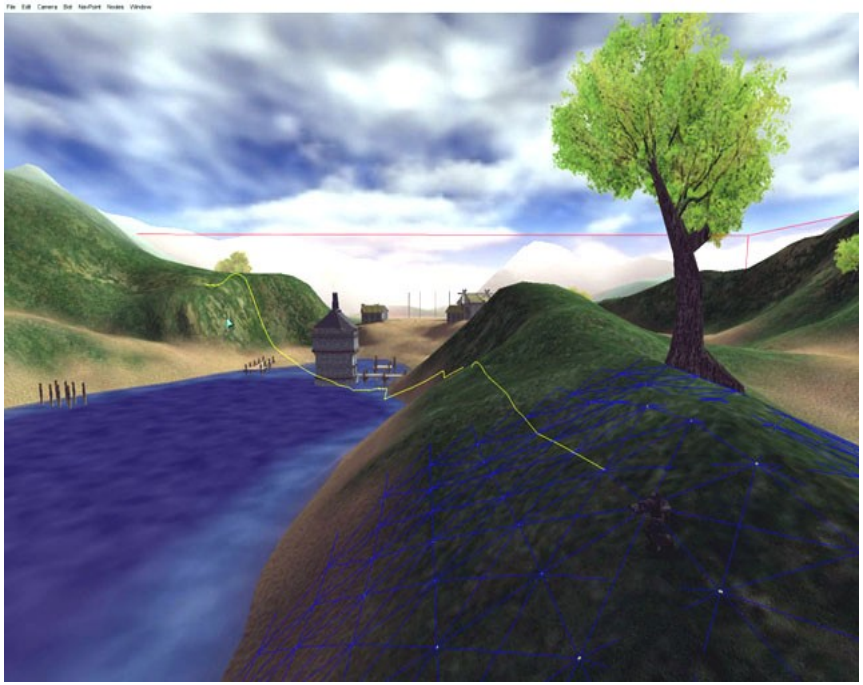
Choix Heuristique

h_2 domine h_1 si: $h_1(n) \leq h_2(n) \leq g^*(n)$

h_2 requiert toujours moins d'exploration que h_1



Distance Euclidienne



Distance Topologique

Recherche Heuristique

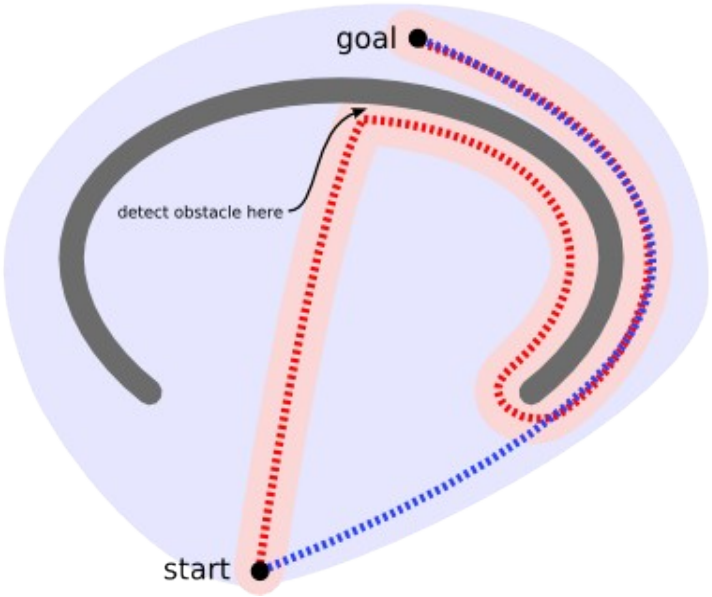
Question 6

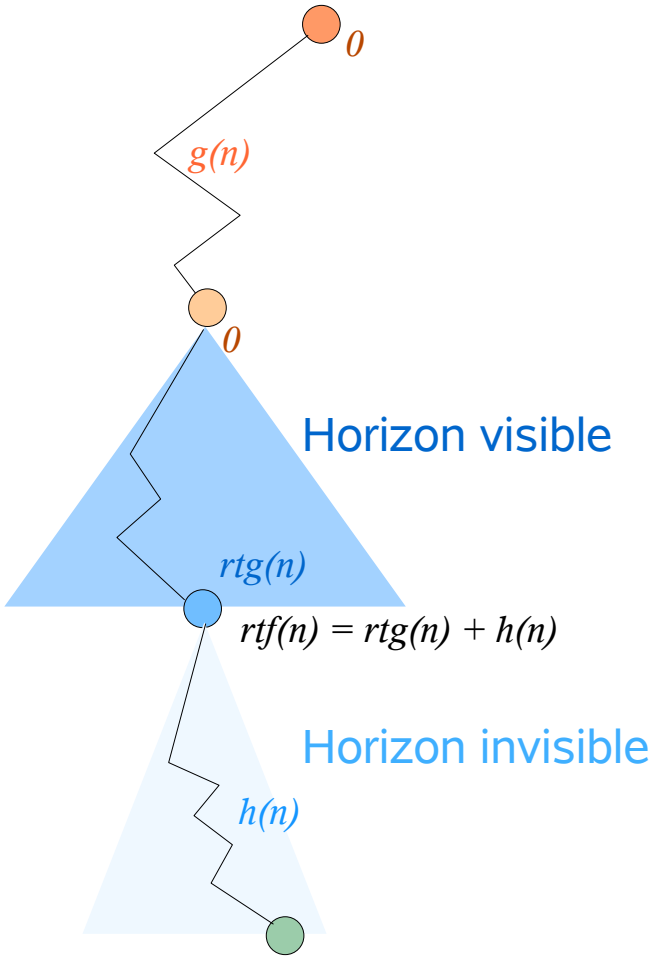
Dans le jeu de taquin quelle serait la meilleure heuristique pour A*

Recherche Heuristique

Real-Time A*

Relancer un A* tous les D temps afin
de prendre en compte le nouveau graphe

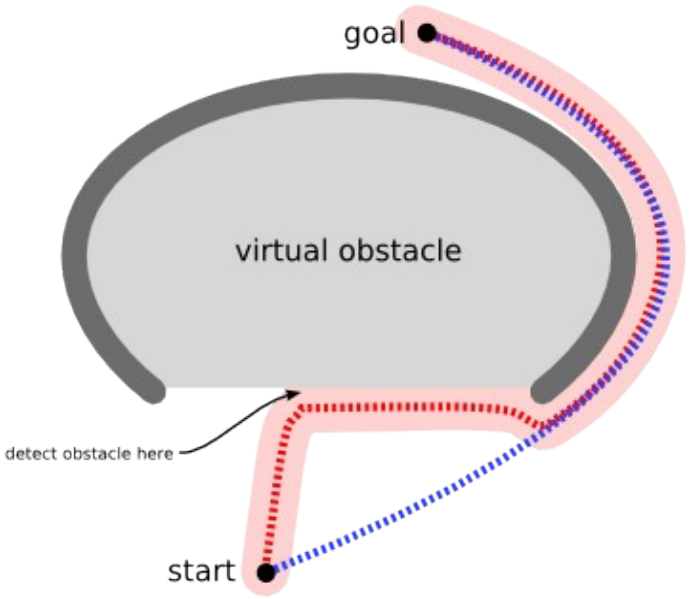


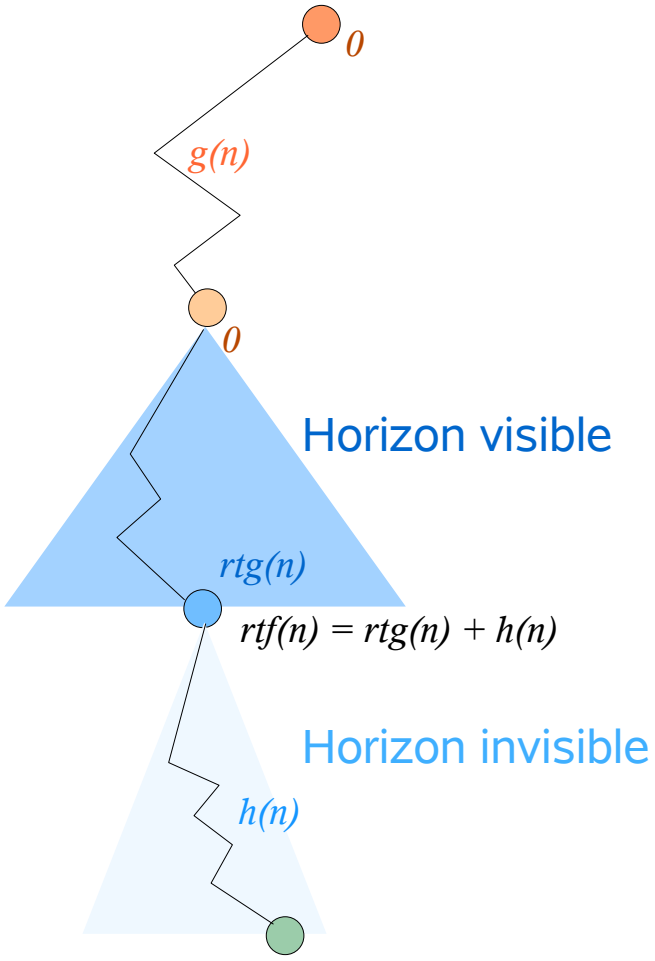


Recherche Heuristique

Real-Time A*

Relancer un A* tous les D temps afin
de prendre en compte le nouveau graphe

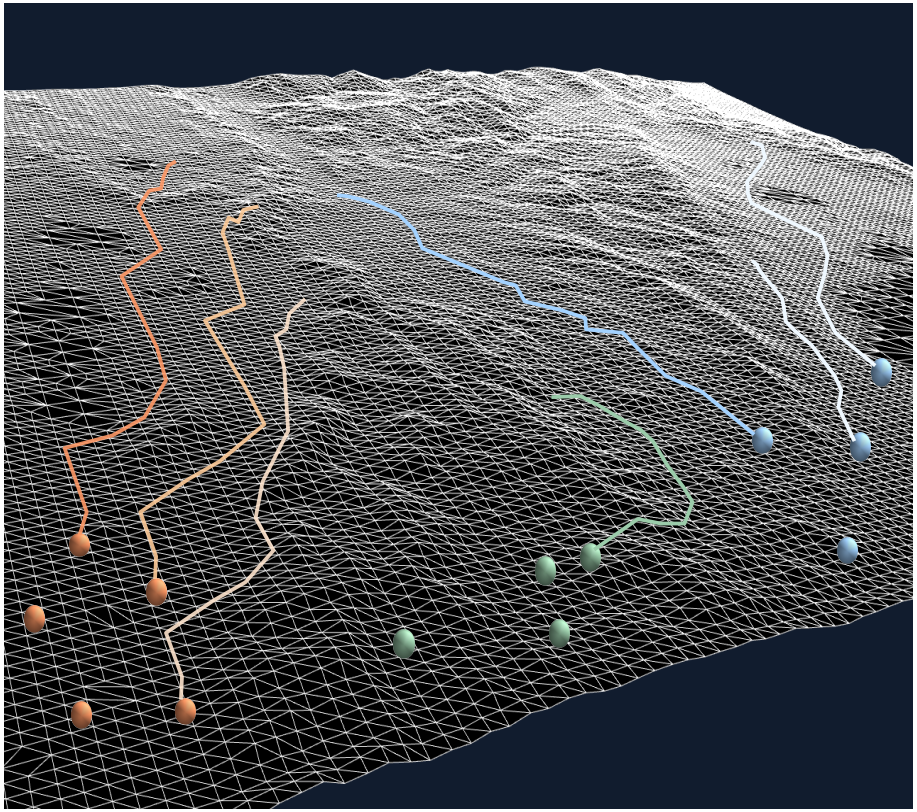


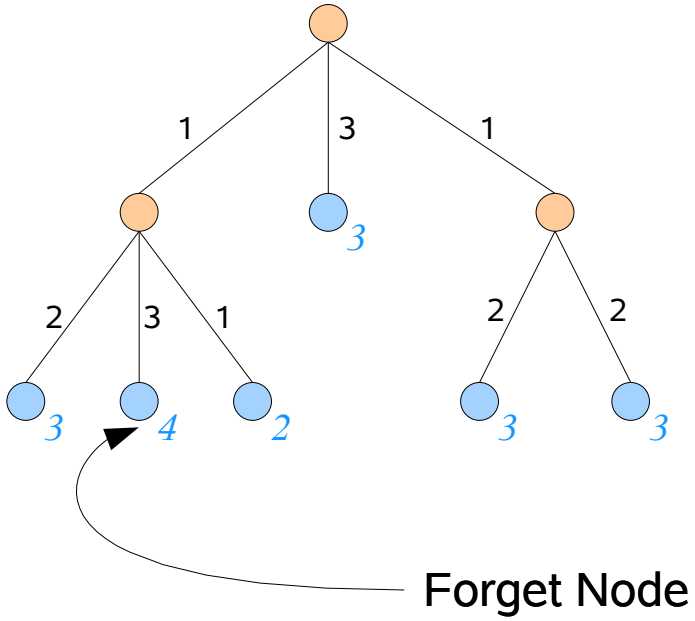


Recherche Heuristique

SMA*

A* avec une mémoire bornée. Lorsque le nombre d'ouverts atteint la mémoire, on « oublie » le noeud avec le plus fort coût.

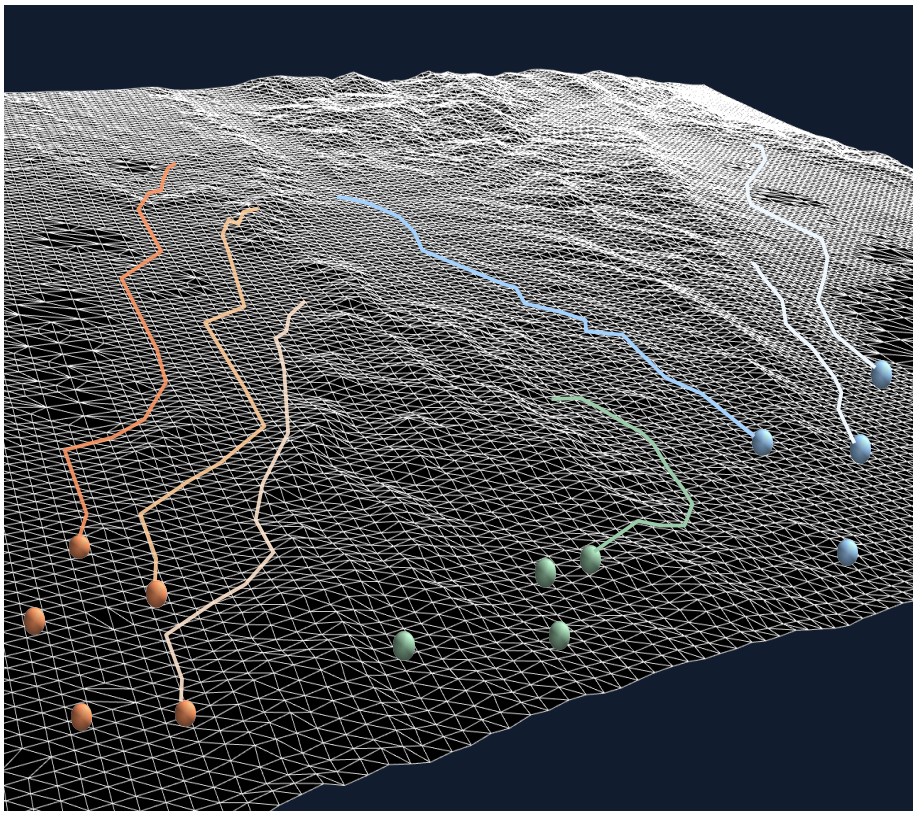


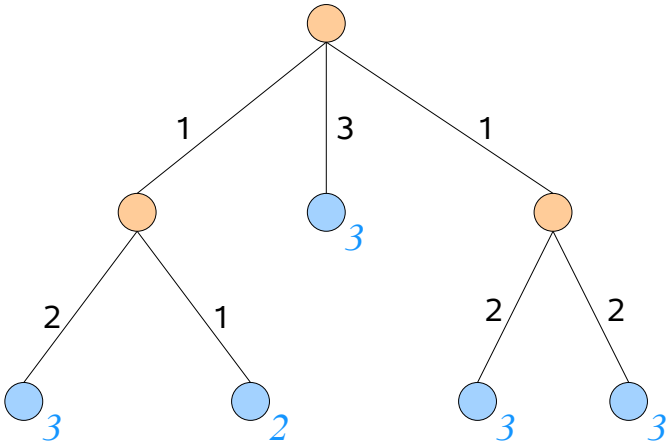


Recherche Heuristique

SMA*

A* avec une mémoire bornée. Lorsque le nombre d'ouverts atteint la mémoire, on « oublie » le noeud avec le plus fort coût.

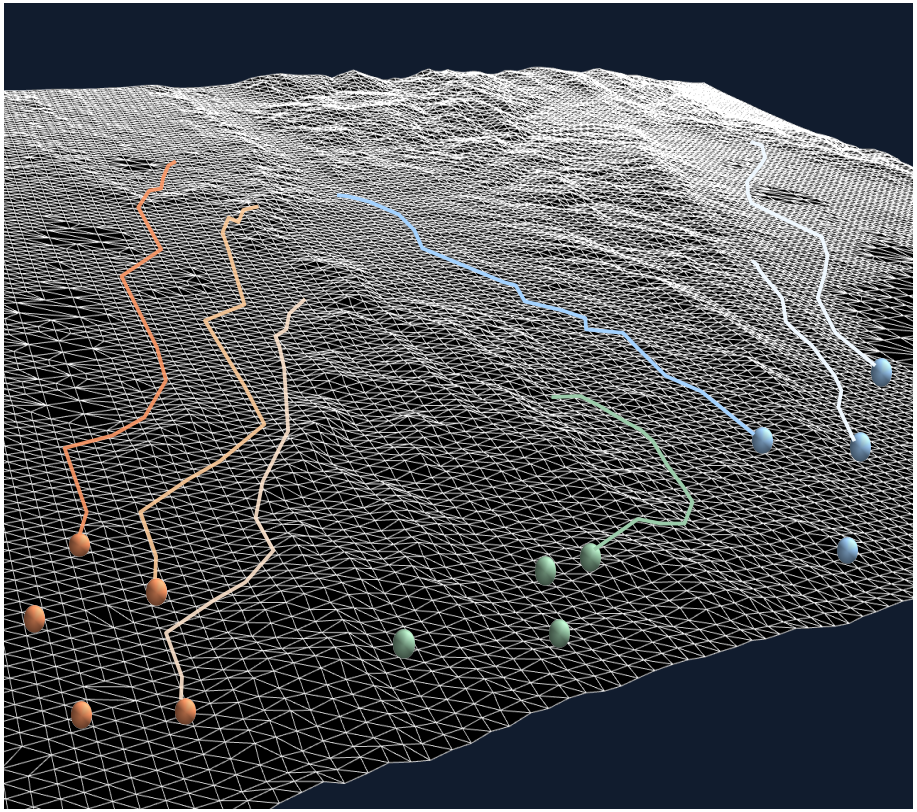


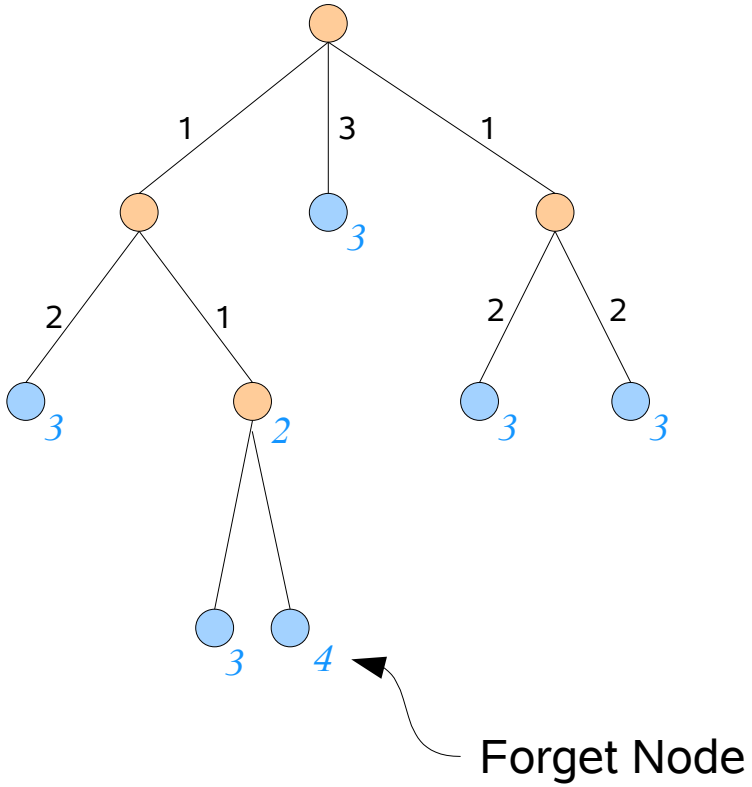


Recherche Heuristique

SMA*

A* avec une mémoire bornée. Lorsque le nombre d'ouverts atteint la mémoire, on « oublie » le noeud avec le plus fort coût, pour le rechercher plus tard si besoin

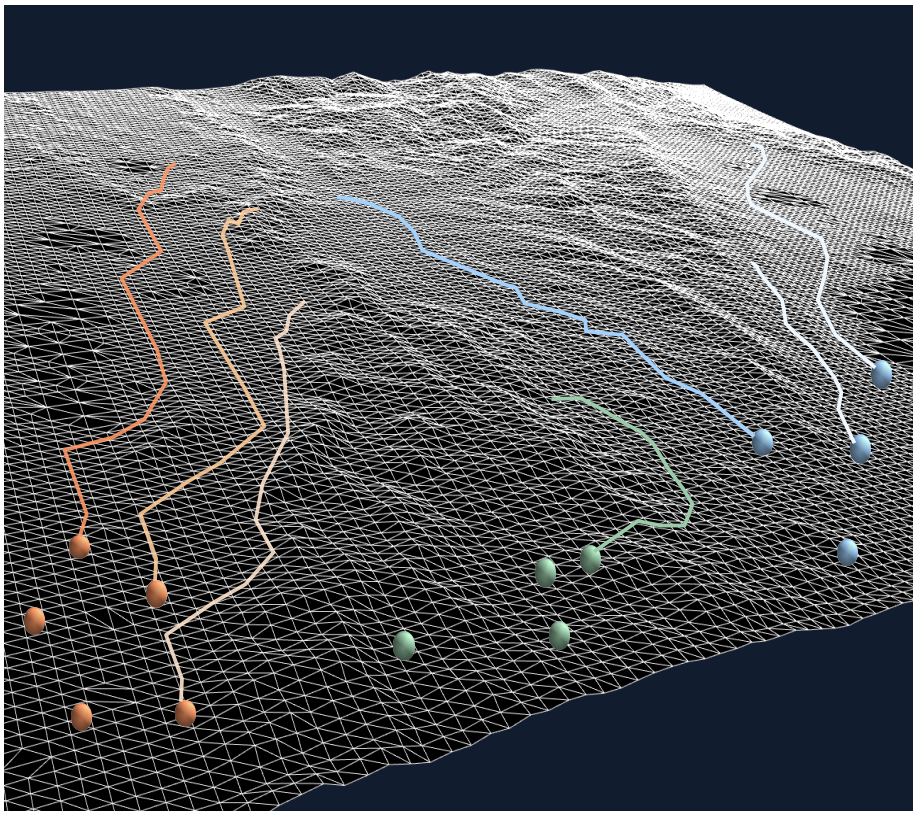


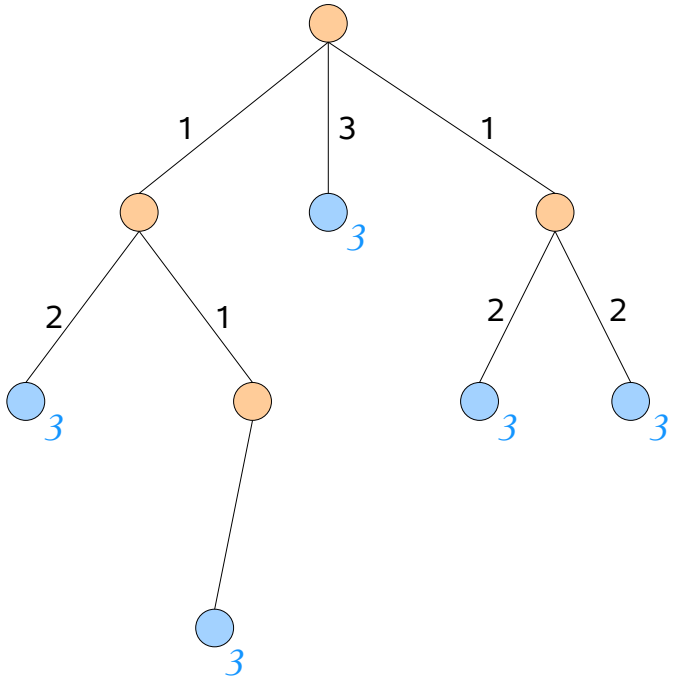


Recherche Heuristique

SMA*

A* avec une mémoire bornée. Lorsque le nombre d'ouverts atteint la mémoire, on « oublie » le noeud avec le plus fort coût, pour le rechercher plus tard si besoin





Recherche Heuristique

Question 7

Si l'on n'ouvre que le nœud le moins couteux dans A*:

- L'algorithme est-il complet ?
- Quelle est sa complexité ?

Algorithmes de Recherche