Cours Résolution de problèmes

Master Informatique visuelle
2021/2022

Prof. Slimane LARABI
USTHB-ENSIA

Cours Résolution de problèmes

Chapitre 1. Résolution de problèmes de planification

1.1. Définitions de l'IA
1.2. Représentation d’un problème par un espace d’états
1.3. Méthodes de recherche de solution dans les espaces d’états

Chapitre 1. Résolution de problèmes de planification

1.1. Définitions de l'IA

L'**intelligence artificielle** (**IA**) est l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence [4].

Dans [5], IA est classée dans le groupe des sciences cognitives,
elle fait appel à la neurobiologie computationnelle (réseaux neuronaux),
à la logique mathématique
et à l'informatique.

Elle recherche des méthodes de résolution de problèmes à forte complexité logique ou algorithmique.

Par extension elle désigne, dans le langage courant, les dispositifs imitant ou remplaçant l'homme dans certaines mises en œuvre de ses fonctions cognitives [5]

Chapitre 1. Résolution de problèmes de planification

1.2 Représentation d'un problème par un espace d'états

Définitions :

Un espace d'états est défini à l'aide de :

Etat initial d'un problème :

Opérateurs :

Ils permettent de passer d'un état à l'autre
Exprimés avec des fonctions non partout définies
Exprimables sous forme de règles de production ou de réécriture

Chapitre 1. Résolution de problèmes de planification

Exemples de représentation par espace d'états :

Problème du voyageur de commerce :

Il s'agit d'aller de la ville A et y retourner en traversant toutes les autres villes une seule fois et en minimisant le parcours (somme des distances).

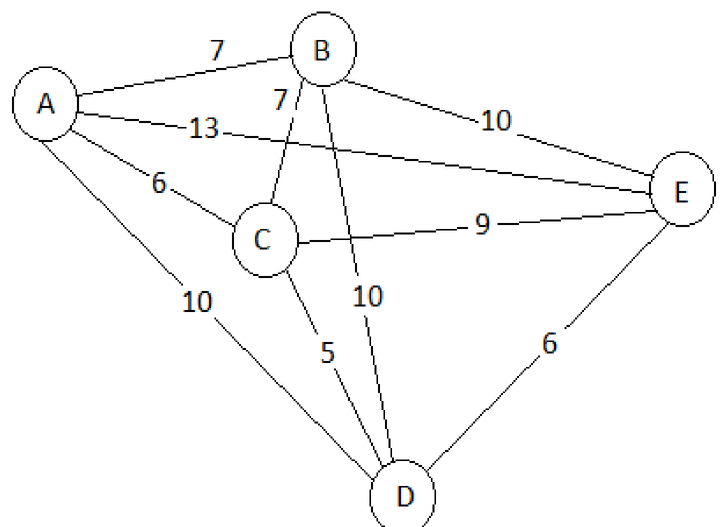


Figure 1. Problème du voyageur de commerce

Chapitre 1. Résolution de problèmes de planification

Exemples de représentation par espace d'états :

Etats : Villes
Opérateurs : Aller à Ville x avec conditions :
Aller à B ne peut pas s'appliquer à partir de B
Etat final acceptable: Tout état de la forme A$A où $ est une permutation des caractères B,..,E

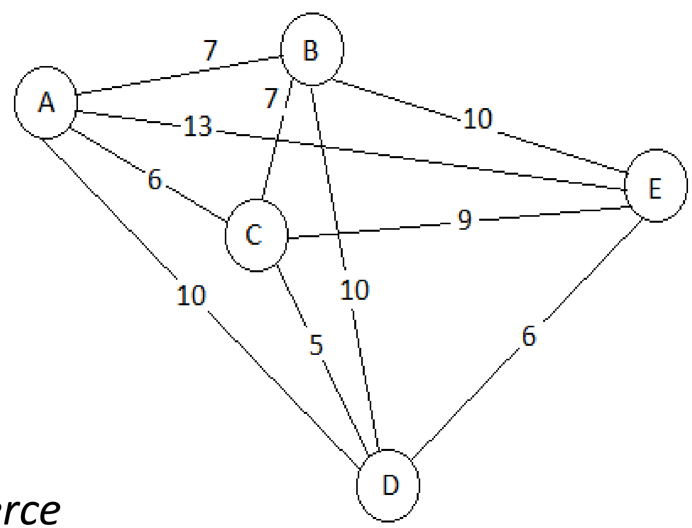


Figure 2. Problème du voyageur de commerce

Chapitre 1. Résolution de problèmes de planification

Exemples de représentation par espace d'états :

Etat objectif : Un état final acceptable de moindre coût
Ci-après le développement de l'espace d'états :
Résultat : Aller en C, Aller en D, Aller en E,
Aller en B, Aller en A, Coût : 34

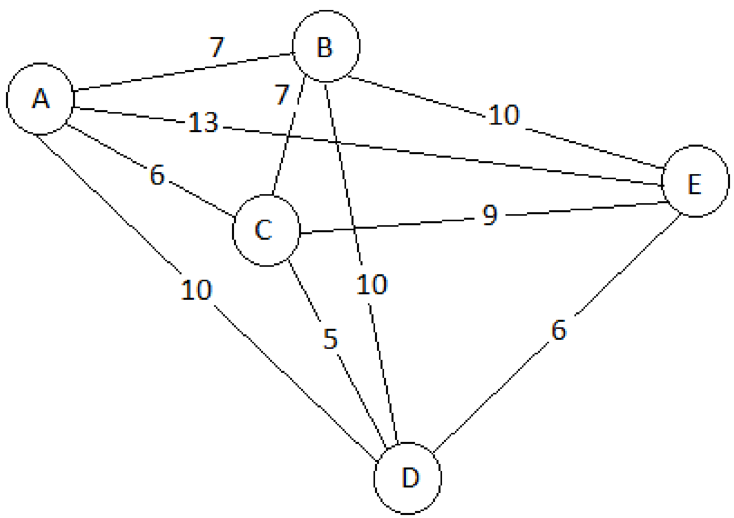


Figure 3. Problème du voyageur de commerce

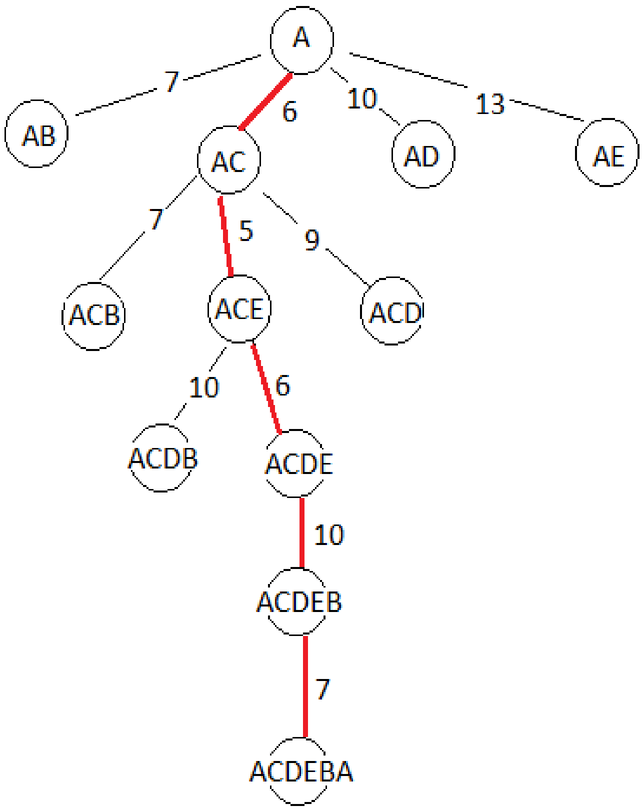


Figure 4. développement de l'espace d'états

Chapitre 1. Résolution de problèmes de planification

Exemples de représentation par espace d'états :

Problème d'analyse syntaxique

« abaabab » est-il un mot (M) du langage défini par les règles suivantes :
ab $\longrightarrow$ M opérateur de réduction n° 1
aM $\longrightarrow$ M opérateur de réduction n° 2
Mb $\longrightarrow$ M opérateur de réduction n° 3
MM $\longrightarrow$ M opérateur de réduction n° 4

Le graphe de la figure 5 illustre l'espace d'états où un état correspond à la chaine réduite, initiale ou finale, l'opérateur correspond à une règle de réduction.

Chapitre 1. Résolution de problèmes de planification

Exemples de représentation par espace d'états :

Etat initial est donné par la chaîne « abaabab ».

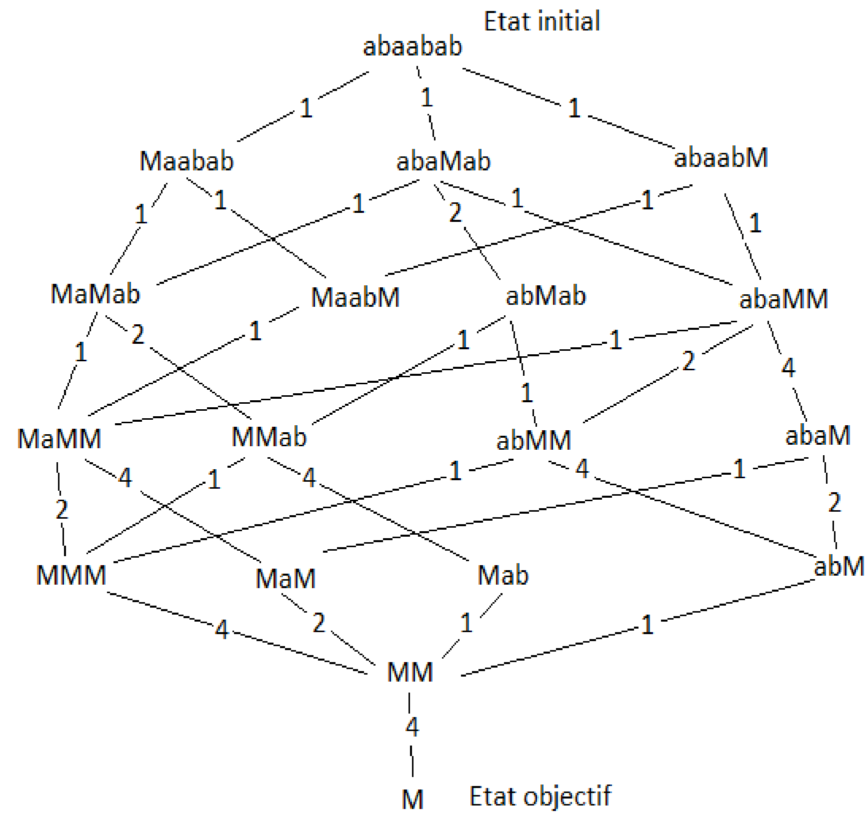


Figure 5. Espace d'états pour le problème de réduction

Chapitre 1. Résolution de problèmes de planification

Exercice 1 :
Choisir une description d'états, puis des opérateurs pour le problème suivant et le résoudre.

On dispose de deux bidons de 5 litres et 2 litres. Au départ, le bidon de 5 litres est plein d'eau, celui de 2 litres est vide. On veut obtenir 1 litre dans le bidon de 2 litres et 4 litres dans le bidon de 5 litres.

Il est possible de :

- Vider un bidon dans l'autre, s'il y a de la place
- Verser l'eau dans un troisième bidon, de volume inconnu et supérieur à 5 litres.
- Verser le contenu du troisième bidon dans le bidon de 5 litres.
- Verser le contenu du troisième bidon dans le bidon de 2 litres.

Chapitre 1. Résolution de problèmes de planification

Exercice 1 :

Solution en termes de triplets (Bidon1, Bidon2, Bidon3), voir espace d'états de la figure 4.

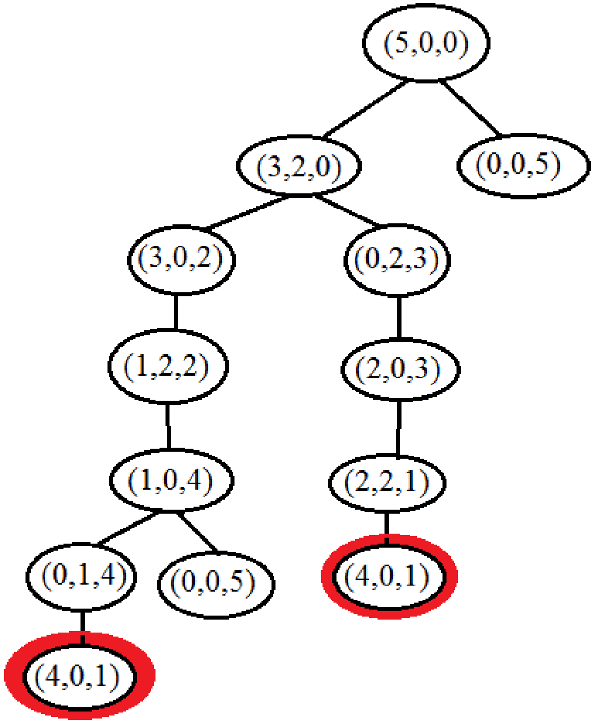


Figure 6. Espace d'états du problème de transfert du contenu des bidons.

Chapitre 1. Résolution de problèmes de planification

Exercice 2 :

Trouver un chemin dans un espace d'états qui montre que la phrase P=(((),()),(),((),())) est bien une phrase de la grammaire définie par les règles de réécriture suivantes :

( ) $\longrightarrow$ P
P $\longrightarrow$ A
A,A $\longrightarrow$ A
(A) $\longrightarrow$ P

Chapitre 1. Résolution de problèmes de planification

1.3 Méthodes de recherche de solution dans les espaces d'états

1.3.1 Mécanismes et idées de base communs aux méthodes de recherche

Un sommet (ou nœud) est associé à l'état initial **s**.

Les successeurs (fils) d'un nœud **n** sont engendrés par application de tous les opérateurs possibles, parmi ceux disponibles à **n.**

Les arêtes joignant les pères à leurs fils seront orientés des fils vers leurs pères.

Les méthodes envisagées diffèrent essentiellement quant à l'algorithme du ***choix du prochain nœud à développer.***

Chapitre 1. Résolution de problèmes de planification

1.3 Méthodes de recherche de solution dans les espaces d'états

1.3.1 Mécanismes et idées de base communs aux méthodes de recherche

Une méthode de recherche sera d'autant "meilleur" qu'elle produira un "petit" ou "peu coûteux" graphe de recherche avec un petit effort de génération.

Une solution sera obtenue dès l'instant où un nœud objectif **N** apparaîtra dans le graphe de recherche.

Chapitre 1. Résolution de problèmes de planification

1.3 Méthodes de recherche de solution dans les espaces d'états

1.3.2 Méthodes aveugles (non informées)

Deux méthodes de recherche
(en largeur d'abord) et
(en profondeur d'abord) sont qualifiées d'aveugles.

Elles sont « aveugles » en raison de:
- l'ignorance de la position de la solution
- et le choix de l'exploration de l'arbre ou graphe en entier sans information a priori du coût qui sera assigné pour accéder à la solution.

Chapitre 1. Résolution de problèmes de planification

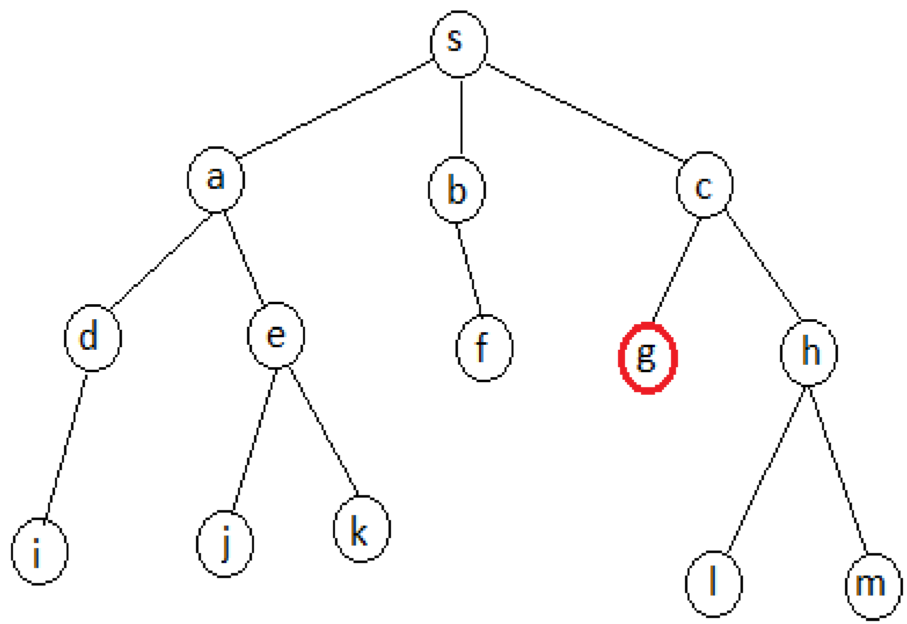
1.3.2 Méthodes aveugles (non informées)

Méthode "en Largeur d'abord" : Breadth-first

<u>a)- Cas des graphes de type arborescence</u>

Le principe est donné par l'algorithme 1 suivant :

Exemple :



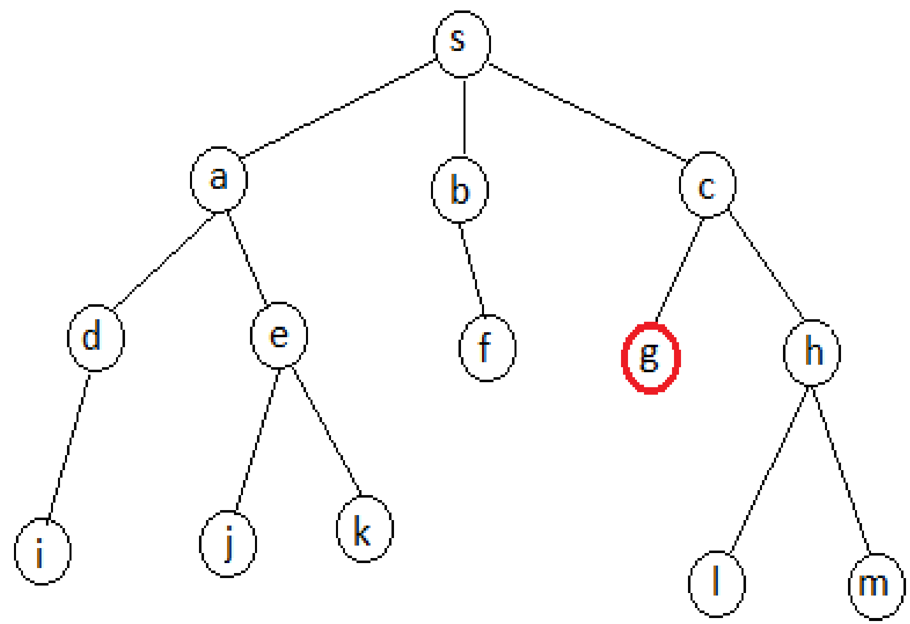
Chapitre 1. Résolution de problèmes de planification

1.3.2 Méthodes aveugles (non informées)

Méthode "en Largeur d'abord" : Breadth-first

Les nœuds explorés (nœud, nœud père),
enfilés dans la file, étape par étape :

(s, nil)
(a,s), (b,s), (c,s)
(d,a), (e,a)
(f,b)
(g,c), (h,c) Arrêt car g est un nœud objectif.



Algorithm 1
Begin
s : Etat initial
n_o: Nœud objectif
OPEN: File initialement vide
succ: fournit les successeurs d'un nœud
boolean Found= False
If(s== n_o) **Then** Found=true // Succès
  Else
    If(Succ(s)==Ø)
      Then Found= False // Echec
      Else **Enfiler** (OPEN, s)
    EndIf
EndIf
While((!OPEN_Empty) && (!Found))
Do
  n= **Défiler** (OPEN)
  If(succ(n) !=Ø)
    Then
      Trouver (n_1, n_2, .., n_k) les successeurs de n,
      Créer le lien du nœud n_j vers le nœud n.
      Enfiler (OPEN, n_j), j=1..k
      If(n_j == objective) **Then** n_o= n_j ; Found=true **EndIf**
    EndIf
EndDo
If(!Found) **then** // Echec **else** reconstruire la solution de s vers n_o **EndIf**
End

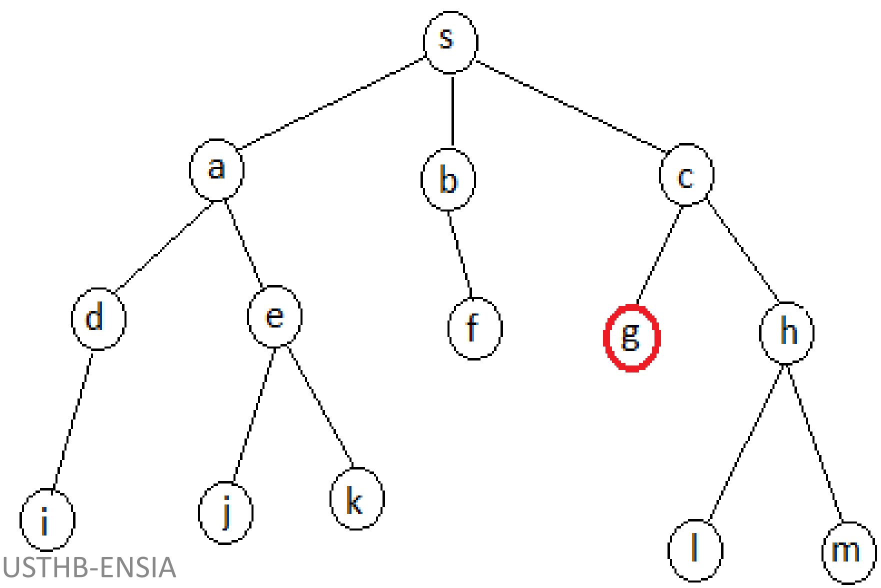
Chapitre 1. Résolution de problèmes de planification

1.3.2 Méthodes aveugles (non informées)

Méthode "en Largeur d'abord" : Breadth-first

Propriétés de la méthode :

Si l'espace complet d'états comporte un nœud objectif, Breadth-first en trouvera un



Chapitre 1. Résolution de problèmes de planification

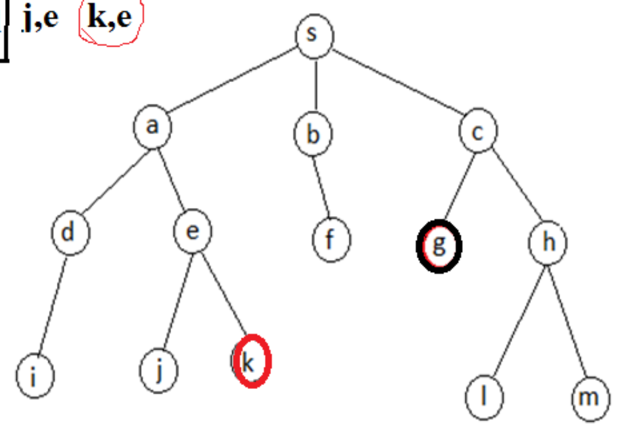
1.3.2 Méthodes aveugles (non informées)

Méthode "en Largeur d'abord" : Breadth-first

Déroulement de l'exemple du l'arbre ci-contre avec k nœud objectif

~~s,nil~~	~~a,s~~	~~b,s~~	~~c,s~~	~~d,a~~	~~e,a~~	f,b	g,c	h,c	i,d	j,e	k,e

Found=False



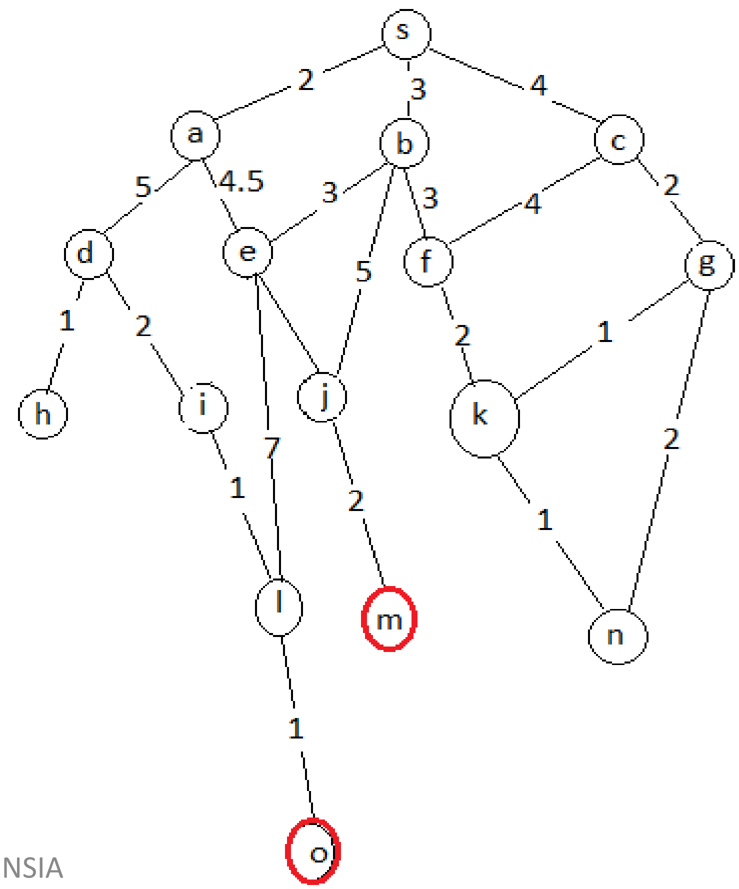
Le chemin vers le noeud objectif k,e, a,s

Chapitre 1. Résolution de problèmes de planification

1.3.2 Méthodes aveugles (non informées)

Méthode "en Largeur d'abord" : Breadth-first

<u>Cas des graphes quelconques</u>



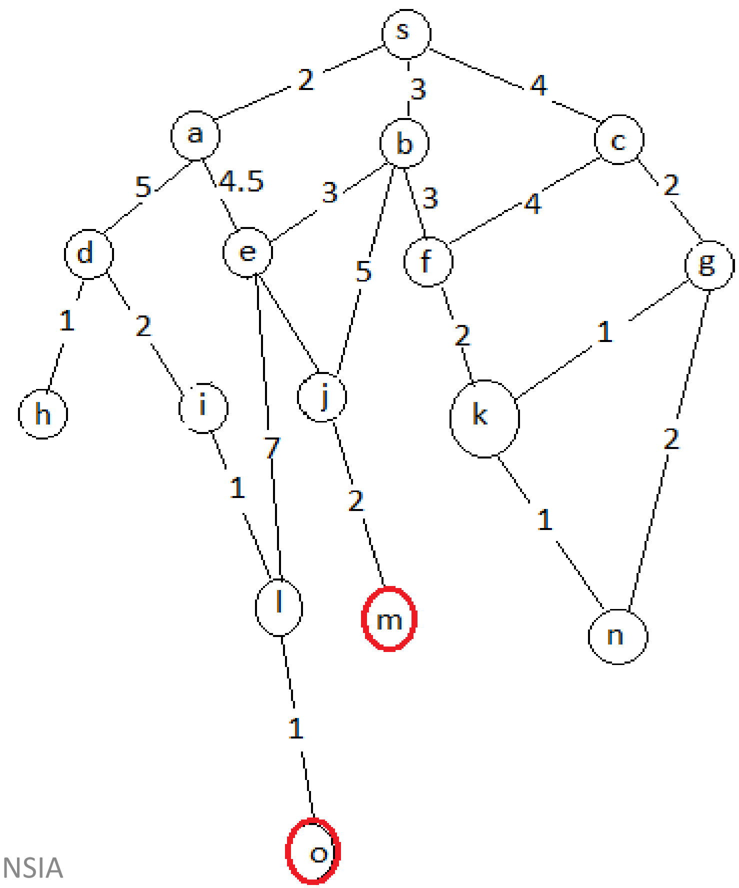
Algorithm 1
Begin
s : Etat initial
n_o: Nœud objectif
OPEN, CLOSED: Files initialement vide
succ: fournit les successeurs d'un nœud
boolean Found= False
If(s== n_o) **Then** Found=true // Succès
  Else
    If(Succ(s)==Ø)
      Then Found= False // Echec
      Else **Enfiler** (OPEN, s)
    EndIf
EndIf
While((!OPEN_Empty) && (!Found))
Do
  n= **Défiler** (OPEN); Enfiler (CLOSED, n)
  If(succ(n) !=Ø)
    Then
      Trouver (n_1, n_2, .., n_k) les successeurs de n,
      Créer le lien du nœud n_j vers le nœud n.
      Enfiler (OPEN, n_j), j=1..k tel que nj n'est pas dans OPEN and CLOSED
      If(n_j == objective) **Then** n_o= n_j ; Found=true **EndIf**
    EndIf
EndDo
If(!Found) **then** // Echec **else** reconstruire la solution de s vers n_o **EndIf**
End

Chapitre 1. Résolution de problèmes de planification

1.3.2 Méthodes aveugles (non informées)

Méthode "en Largeur d'abord" : Breadth-first

OPEN	CLOSED
s	Ø
a,b,c	s
b, c d, e	s,a
c d, e, j, f	
d, e j, f, g	s,a,b
e j, f, g, h, i	s,a,b,c
j, f, g, h, i, l	s,a,b,c,d
f, g, h, i, l, m	s,a,b,c,d,e,j



Chapitre 1. Résolution de problèmes de planification

1.3.2 Méthodes aveugles (non informées)

Méthode "en Largeur d'abord" : Breadth-first

Propriétés :

Le graphe de recherche est toujours (à toute étape) une arborescence (OPEN-CLOSED)

Si l'espace complet des états comporte un nœud objectif:
Breadth-first en trouvera un

Chapitre 1. Résolution de problèmes de planification

1.3.2 Méthodes aveugles (non informées)

Méthode "en Profondeur d'abord" : Depth-first

Propriétés :

Principe : Seul le cas des graphes complets de type arborescence est envisagé ici.
On utilise les notations suivantes :

S : nœud initial
OPEN : Pile vide initialement
L : paramètre (profondeur) limitant la distance à la racine (en nombre d'arcs)
On développe d'abord les nœuds les plus récemment engendrés.

Algorithm 2
Begin
Algorithm
Begin
s : initial state
n_o: objective node
L: level of depth
OPEN: Pile initialement vide
succ: fournit les successeurs d'un nœud
boolean Found= False
If(s== n_o) **then** Found=true // Succès
Else
If(Succ(s)==Ø) **then** Found= False // Echec
    Else push (OPEN, s)
        While((!OPEN_Empty) && (!Found))
        Do
            n= **pop** (OPEN)
            If(succ(n !=Ø) then
            If(Level<L) then
                L++;
                Soient (n_1, n_2, .., n_k) les successeurs de n, créer le lien du nœud n_j vers le nœud n.
                Push (OPEN, n_j), j=1..k
                If(n_j == objective) n_o= n_j ; Found=true EndIf
            EndIf
        EndDo
EndIf EndIf
If(!Found) **then** // Echec **else** reconstruire la solution de s vers n_o **EndIf**
End

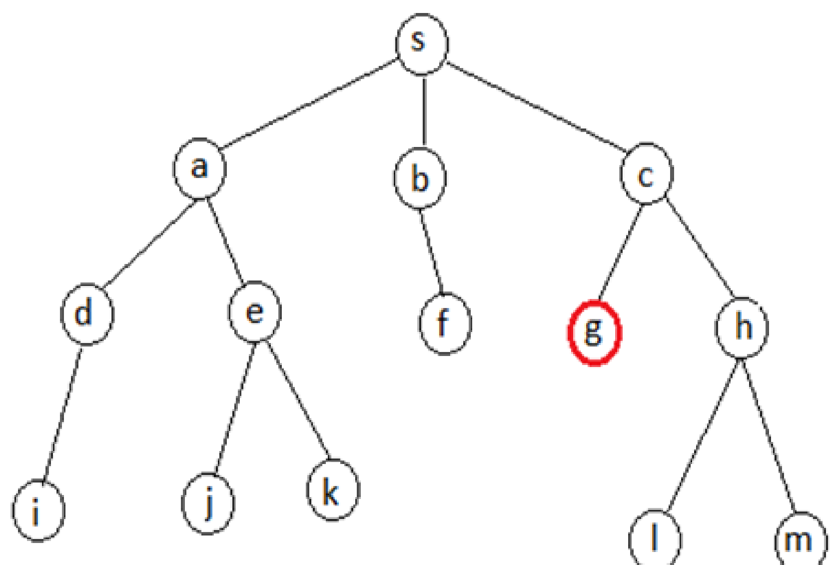
Chapitre 1. Résolution de problèmes de planification

1.3.2 Méthodes aveugles (non informées)

Méthode "en Profondeur d'abord" : Depth-first

Exemple : Appliquons l'algorithme sur l'espace d'états suivant pour L=3 :

OPEN= s
= a, b, c
= d,e b,c
= i e b,c
= j,k b,c
= f,c
= h,g



Chapitre 1. Résolution de problèmes de planification

1.3.2 Méthodes aveugles (non informées)

Méthode "en Profondeur d’abord" : Depth-first

Propriétés:

Lorsqu’on ne limite pas la profondeur de développement, “depth-first” ne garantit pas la découverte d’un nœud objectif même si celui-ci existe dans le graphe complet (cas de branche infinie).

Chapitre 1. Résolution de problèmes de planification

1.3.3 Méthodes informées

Méthode du coût uniforme (UCS)

Principe

A chaque moment de choix d'un nœud à développer, on fait l'hypothèse que tous les nœuds candidats (en position de feuilles) sont à égal distance d'un nœud objectif (coût uniforme) et on décide de développer d'abord les nœuds les moins éloignés dans le graphe de recherche courant de la racine (ayant un coût minimal).

On utilisera g(ni): coût du chemin allant de s à ni.

Algorithm 3
Begin
s : Etat initial
n_o: Nœud objectif
OPEN, CLOSED: Files initialement vide
succ: fournit les successeurs d'un nœud
boolean Found= False
If(s== n_o) **Then** Found=true // Succès
  Else
    If(Succ(s)==∅)
      Then Found= False // Echec
      Else **Enfiler** (OPEN, s), g(s)=0
   EndIf
EndIf
While((!OPEN_Empty) && (!Found))
Do
   ni= **Défiler** (OPEN) tel g(ni) est minimal; Enfiler(CLOSED, n_i)
   If(n_i == objectif) n_o= n_i ; Found=true
   Else If(succ(ni) !=∅)
     Then
         Trouver (n_{i1}, n_{i2}, .., n_{ik}) les successeurs de ni,
         Enfiler (OPEN, n_{ij}), j=1..k, g(nij)=g(ni)+Cij ....(*)
         Créer le lien du nœud n_{ij} vers le nœud ni. ....(**)
   EndIf
EndDo
If(!Found) **then** // Echec **else** reconstruire la solution de s vers n_o **EndIf**
End

Chapitre 1. Résolution de problèmes de planification

1.3.3 Méthodes informées

Méthode du coût uniforme (UCS)

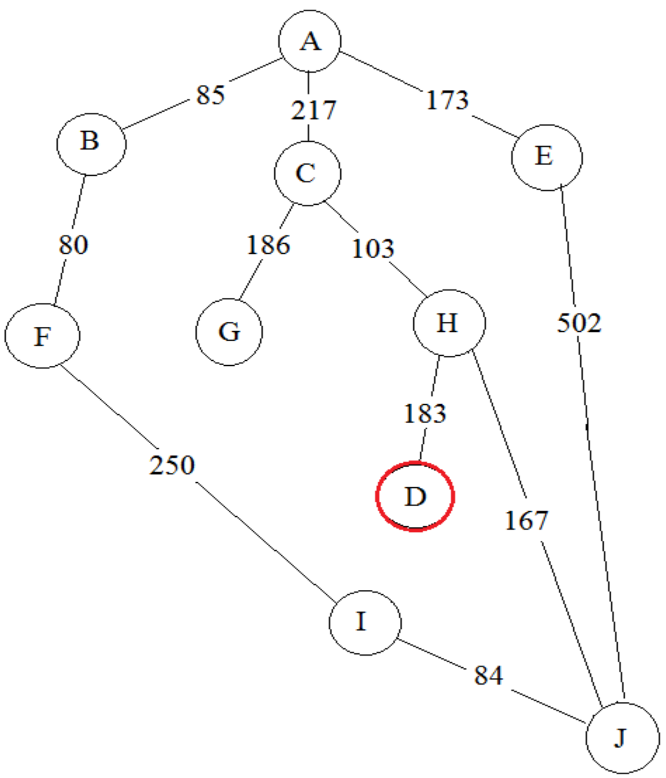
Pour un graphe quelconque, ajouter à l'algorithme 3 au niveau :
(*) : s'il n'appartient pas à OPEN et à CLOSED. Si un successeur est déjà présent dans OPEN ou CLOSED, lui associer le coût min.
à (**) : mettre à jour les liens des nœuds pour lesquels les coûts ont été mis à jour.

Propriété:
S'il existe un nœud objectif dans l'espace complet des états et tel que tous les coûts sont positifs, alors l'algorithme coût uniforme trouvera un tel nœud objectif avec un chemin minimal.

Méthode du coût uniforme (UCS)

Exemple:

OPEN	CLOSED
(A,nil,0)	
(B,A,85), (C,A,217), (E,A,173)	(A,nil,0)
(C,A,217), (E,A,173), (F,B,165)	(B,A,85)
(C,A,217), (E,A,173), (I,F, 415)	(F,B,165)
(C,A,217), (I,F, 415), (J,E,675)	(E,A,173)
(I,F, 415), (J,E,675), (G,C,403), (H,C,320)	(C,A,217)
(I,F, 415), ~~(J,E,675)~~, (G,C,403),(D,H,503),(J,H,487)	(H,C,320)
(I,F, 415), ~~(J,E,675)~~, (D,H,503),(J,H,487)	(G,C,403)
~~(J,E,675)~~, (D,H,503), (J,I, 499)	(I,F, 415)
~~(J,E,675)~~, (D,H,503)	(J,H,487)
~~(J,E,675)~~	(J,I, 499) (D,H,503)



Chapitre 1. Résolution de problèmes de planification

1.3.3 Méthodes informées

Algorithme de Dijkstra (Edsger Dijkstra, 1959)

Dijkstra, E. W., « A note on two problems in connexion with graphs », Numerische Mathematik, vol. 1, 1959, p. 269–271 (DOI 10.1007/BF01386390.).

Est identique à l'algorithme du coût uniforme à l'exception qu'il ne test pas si le nœud extrait de la file **Open est objectif**. Il s'arrête une fois OPEN est vide.

Dijkstra calcule tous les plus courts chemins depuis une source.
UCS s'arrête dès qu'un nœud objectif (optimal) est trouvé.

De ce fait, il permet d'avoir le chemin optimal de s à tout nœud du graphe.

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A

Principe :

Le choix du nœud à développer est guidé par une connaissance heuristique.

A chaque feuille produite (et à chaque étape de la recherche), on associe une valeur censée représenter la promesse du nœud. On dit qu'on évalue les nœuds. Nous noterons $\hat{f}(n)$ la valeur du nœud n, $\hat{f}$ est dite fonction d'évaluation.

La notation $\hat{f}(n)$ est trempeuse : $\hat{f}$ peut ne pas dépendre que de n mais aussi de l'état du développement de la recherche.
On écrira $\hat{f}(n)$ = $\hat{g}(n)$ +$\hat{h}(n)$ où $g(n)$ est le coût du chemin parcouru et $\hat{h}(n)$ est une estimation du coût du chemin qui reste à parcourir du nœud n vers l'objectif

Nous convenons que n est d'autant plus prometteur que $\hat{f}(n)$ est petite.

Algorithm 4
Begin
s : Etat initial , n_o: Nœud objectif, OPEN, CLOSED: Listes initialement vide
succ: fournit les successeurs d'un nœud, $\hat{f}(s)$ is une fonction heuristique associée au nœud n
bool Found= False
If(s== n_o) **then** Found=True // Succès
Else
Calculer $\hat{f}(s)$, Op(s)= $\hat{f}(s)$ // Op pour Open et Cl pour Closed
Insérer (OPEN, (s, Op(s))),
While((OPEN≠∅) && (!Found))
Do
Soit N= {n_k de OPEN tel que (n_k, Op(n_k)) est minimal}
If($n_k == n_0$) **then** **Found=True**
Else
Choisir aléatoirement n_i de N // Op(n_i) est minimal
Retirer (n_i, Op(n_i))
Insérer (n_i, C(n_i)) dans CLOSED
If(succ(n_i) ≠∅) **then**
Soient ($n_{i1}, n_{i2}, .., n_{ip}$) les successeurs de n_i
Calculer $\hat{f}(n_{il})$ for $l = 1..p$
For each n_{il}
DO
If $n_{il} \notin$ OPEN **then**
Op(n_{il})= $\hat{f}(n_{il})$
Créer un lien du noeud n_{il} vers n_i
Insérer dans OPEN (n_{il}, Op(n_{il}))

Else
Mettre à jour dans OPEN (n_{il},Op(n_{il})): Op(n_{il})=min($\hat{f}(n_{il})$, Op(n_{il}))
Mettre à jour le lien.
EndIf
If n_{il} € CLOSED **then**
Retirer de CLOSED (n_{il},C(n_{il})) tel que $\hat{f}(n_{il}) < C(n_{il})$,
Insérer le dans OPEN avec la nouvelle valeur $\hat{f}(n_{il})$
Créer le lien de n_{il} vers n_i
EndIf
EndFor
EndIf
EndIf
EndDo
If(!Found) **then** // Echec **else** reconstruire la solution de s à n_o **EndIf**
EndIf
End

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A

Remarques :

$\hat{f}$ n'est pas simplement en fonction de n, mais dépend aussi de l'étape de l'algorithme : c'est le plus court chemin de s (racine) vers n connu au moment de l'évaluation.

Si $\hat{h}(n) = 0$, l'algorithme A devient la méthode du coût uniforme.
Si le graphe est fini, l'algorithme A trouvera un chemin menant au nœud objectif. Cependant, ce chemin peut ne pas être optimal.

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A

Remarques :

Si $\hat{h}(n)$<h*(n) où h*(n) est le coût du chemin restant allant de n à l'objectif, l'algorithme A trouvera la solution optimale et il est noté algorithme A*.
Cet algorithme a été proposé pour la première fois par Peter E. Hart (en), Nils John Nilsson (en) et Bertram Raphael (en) en 1968. Il s'agit d'une extension de l'algorithme de Dijkstra de 1959.

Un exemple d'une heuristique admissible pratique est la distance à vol d'oiseau du but sur la carte.

1.3.4 Méthodes ordonnées

Algorithme de type A

Remarques :

h consistant veut dire que l'estimé du coût d'un nœud n'est pas plus élevé que l'estimé du coût de son successeur additionné avec le coût de l'arc entre les deux :$h(n)<h(n_{i+1})+C(n_i, _{ni+1})$

Même si la consistance est plus restrictive que l'admissibilité, dans les applications réelles il est rare de trouver des heuristiques admissibles qui ne sont pas consistantes.

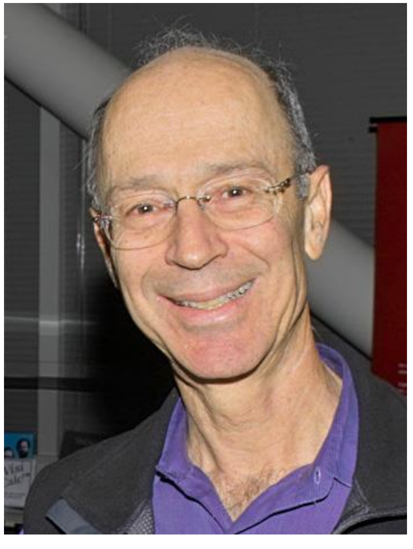
Une conséquence de la consistance de h est que les valeurs de f(n) le long de n'importe quel chemin sont croissantes. Autrement dit f est monotone (croissante).
En effet, on a f(n1) = g(n1) + h(n1) <= g(n1) + c(n1,n2) + h(n2) = f(n2).
Comme f(n) est monotone croissant, cela veut dire que si on tombe sur un nœud but, alors il est forcément optimal puisqu'il n'y a pas de meilleur chemin qui y mène.
Donc h est admissible.

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A* (extension de l'algorithme de Dijkstra)

P. E. Hart, N. J. Nilsson et B. Raphael, « A Formal Basis for the Heuristic Determination of Minimum Cost Paths », IEEE Transactions on Systems Science and Cybernetics SSC4, vol. 4, no 2, 1968, p. 100–107 (DOI 10.1109/TSSC.1968.300136)



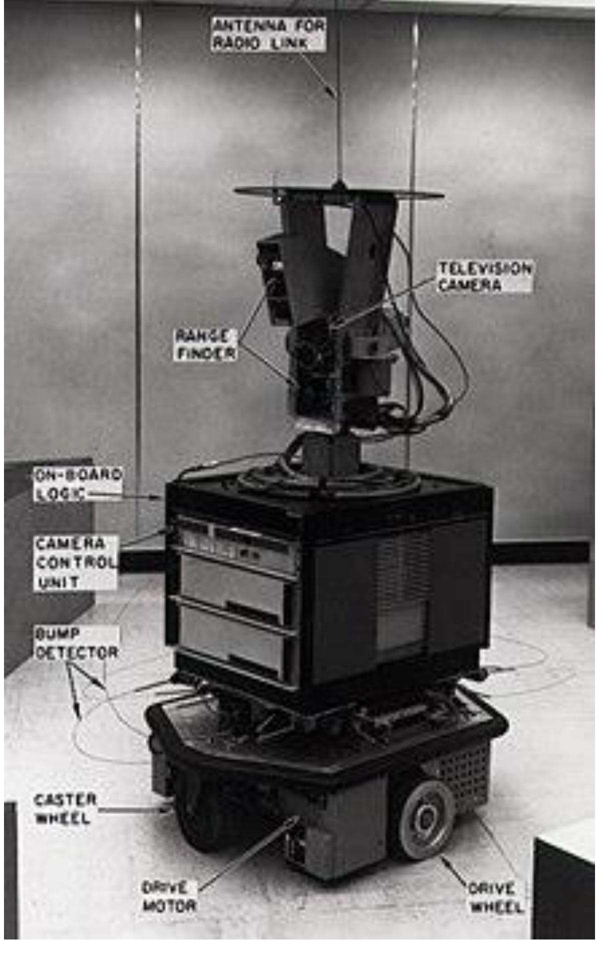
Peter E. Hart, Nils John Nilsson, Bertram Raphael en 1968

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A* (extension de l'algorithme de Dijkstra)

L’algorithme A* est le résultat des travaux de recherche pour la planification du mouvement du robot prototype Shakey dans une scène à obstacles. L'algorithme pour trouver un chemin, que Nilsson appelait A1, était une version plus rapide que la méthode la plus connue à l'époque, l‘algorithme de Djikstra.



Shakey le robot est le premier robot générique capable de raisonner sur ses actions1. Il a été créé à la fin des années 1960 en Californie par SRI International avec le soutien de la DARPA.

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A* (extension de l'algorithme de Dijkstra)

Mountain View en Californie, 1966

Shakey exposé au Musée de l'histoire de l'ordinateur.

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A* (extension de l'algorithme de Dijkstra)

L'algorithme pour trouver un chemin, que **Nilsson** appelait A1, était une version de l'algorithme de Dijkstra, pour trouver des plus courts chemins dans un graphe.

Bertram Raphael a suggéré des améliorations, donnant lieu à la version révisée A2. Puis,

Peter E. Hart a apporté des améliorations mineures à A2. **Hart, Nilsson** et **Raphael** ont alors montré que A2 est optimal pour trouver des plus courts chemins sous certaines conditions.

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A* (extension de l'algorithme de Dijkstra)

Si l'évaluation renvoie simplement toujours zéro, alors, A* exécutera une implémentation possible de l'algorithme de Dijkstra et trouvera toujours la solution optimale.

La meilleure heuristique, bien qu'habituellement impraticable pour calculer, est la distance minimale réelle (ou plus généralement le coût réel) au but. Un exemple d'une heuristique admissible pratique est la distance à vol d'oiseau du but sur la carte.

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Algorithme de type A* (extension de l'algorithme de Dijkstra)

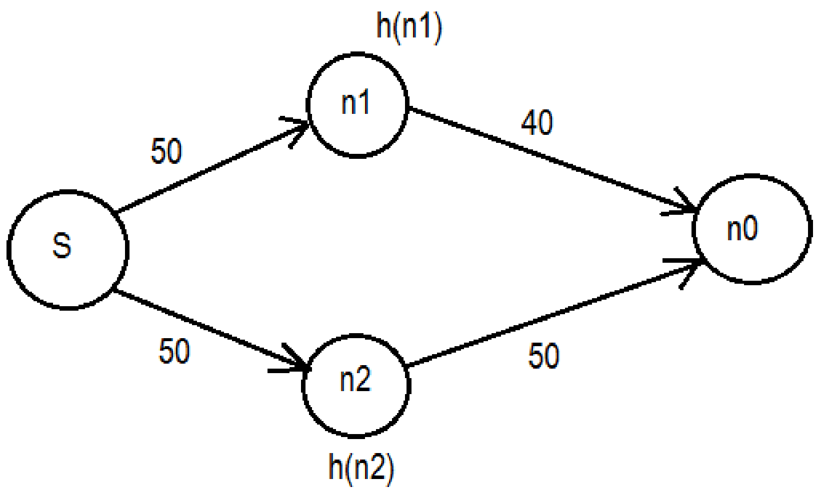
Un algorithme de recherche qui garantit de toujours trouver le chemin le plus court à un but s'appelle « algorithme admissible ».

Si A* utilise une heuristique qui ne surestime jamais la distance (ou plus généralement le coût) du but, A* peut être avéré admissible. Une heuristique qui rend A* admissible est elle-même appelée « heuristique admissible ».

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Exemple 1

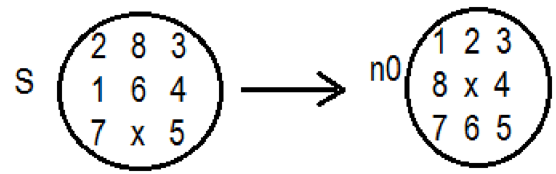


Au nœud n1 correspond f(n1)=50+h(n1)
Au nœud n2 correspond f(n2)=50+h(n2)
Si h(n1)>50 (50 est le coût de n2 à n0) et h(n2) <h(n1) alors l'algorithme A va sélectionner n2 et n0 sera atteint sans que le chemin soit optimal. Exemple h(n1)=60, h(n2)=55.

Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Exemple 2



Jeu de taquin:

Passer de la configuration s à n0.

On utilisera g(n)= niveau de profondeur du nœud

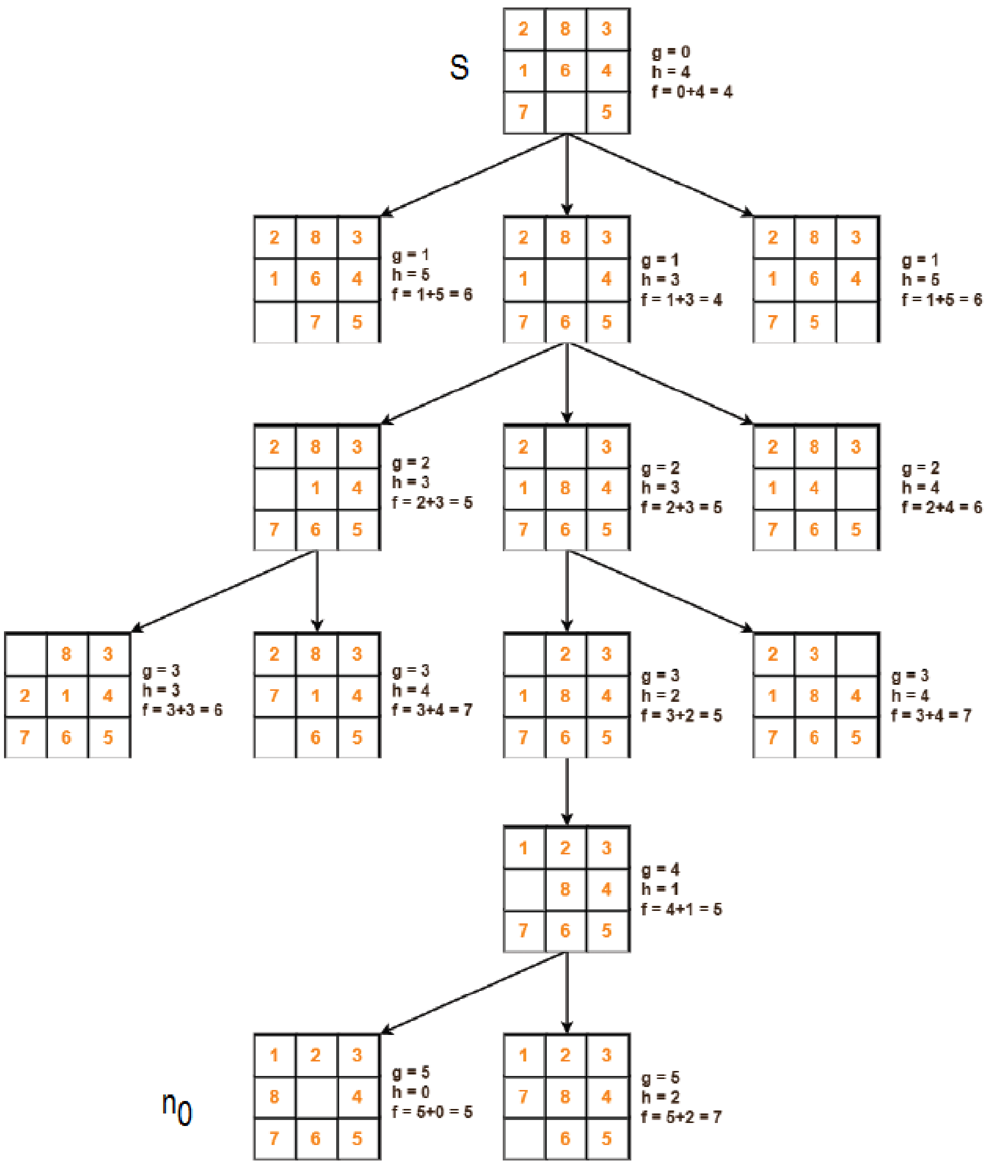
h(n)= nombre de cases qui ne sont pas à leur place

Cette heuristique vérifie l'admissibilité: h(n) <= coût du chemin qui reste à parcourir. Si m cases ne sont pas à leur places, il nous faut au minimum m déplacements pour les positionner à leur places.

L'algorithme est donc de type A*. Il trouve le chemin optimal.

Chapitre 1. Résolution de problèmes de

1.3.4 Méthodes ordonné

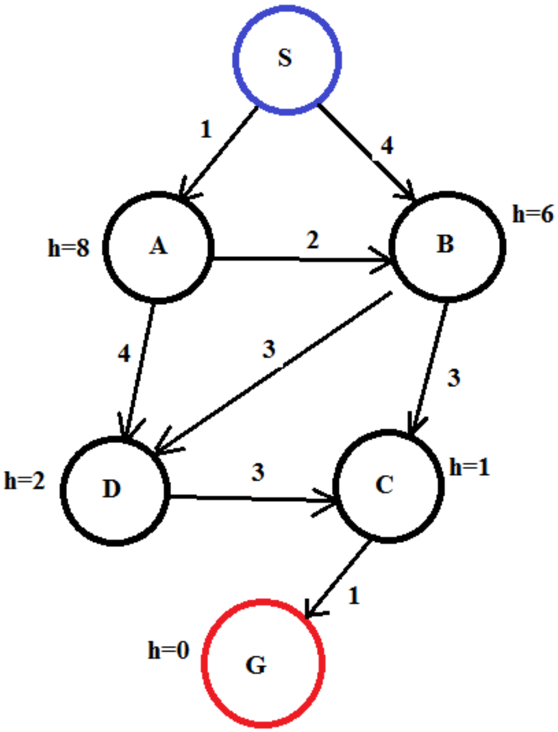


Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Exemple 3

Soit l'espace de recherche illustré par la figure ci-contre.
S et G sont respectivement les états initial et objectif.
Le coût de l'heuristique $h(n_i)$ qui donne une évaluation du coût du chemin $n_i \rightarrow G$ est donné par l'expression $h = valeur$ dans la figure.
Appliquez l'algorithme de type A avec $f(n_i) = g(n_i) + h(n_i)$, où $g(n_i)$ est le coût du chemin allant de S vers l'état n_i. Le chemin trouvé est-il optimal ? Justifiez.



Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Exemple 3

Corrigé :

S → A, B

(A, S, 9), (B, S, 10)

(A, S, 9) selected, gives the child nodes:

(D,A,S,7), (B,A,S,9)

(D,A,S,7) is selected, gives the child nodes:

(C,D,A,S,9)

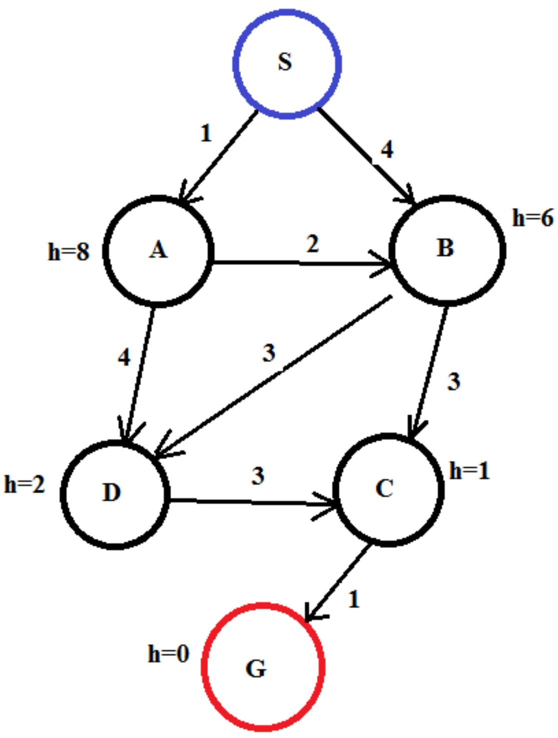
We have to choose between the nodes B or C

If we choose B:

(B,A,S,9) gives the child nodes D and C.

(D,B,A,S, 8), (C,B,A,S,7)

(C,B,A,S,7) is selected, gives the child node: **(G,C,B,A,S,7)** **the goal node.**



Chapitre 1. Résolution de problèmes de planification

1.3.4 Méthodes ordonnées

Exemple 3

Corrigé :

The path is optimal but depends on the choice of the explored node in case of equality between the values of $f(n_i) = g(n_i) + h(n_i)$.
This is explained by the fact that the applied algorithm A for this example of states space is not h^* because the constraint $h(n_i) < h^*(n_i)$ for each node n_i. is not verified for each node n_i.