Chapitre 2.

Résolution de problèmes

Chapitre 2.1 Représentation d'un problème par un espace d'états

Définitions :

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

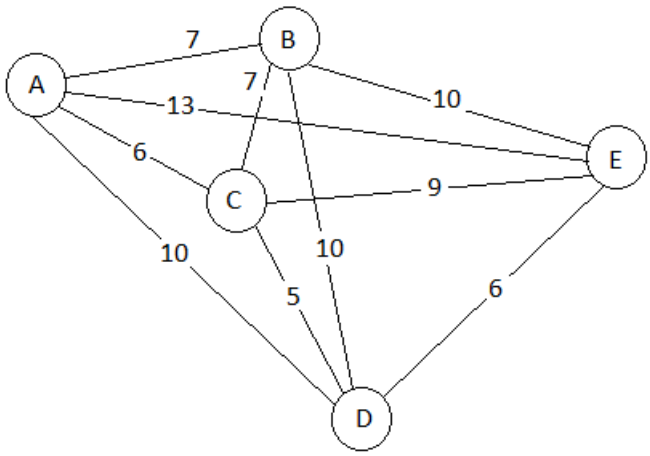
- Etat initial d'un problème :

- 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

Exemples de représentation par espace d'états :

Problème du voyageur de commerce :

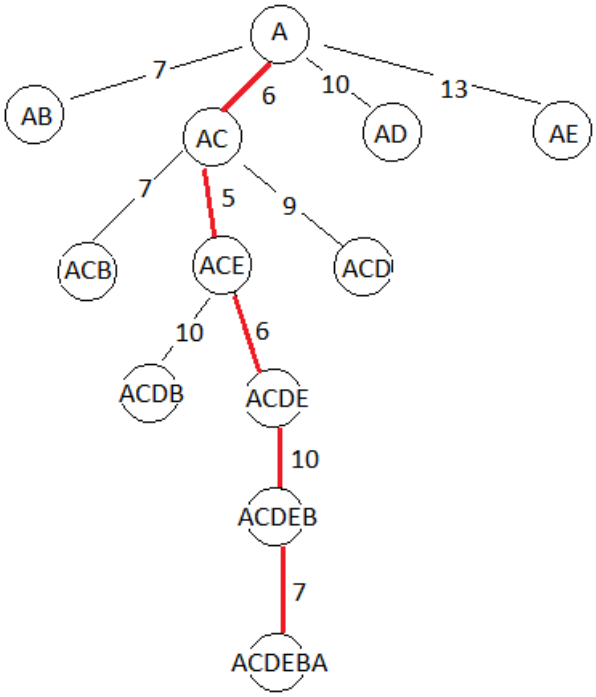
Il s'agit d'aller de la ville A et y retourner en traversant toutes les autres villes une seule fois et en minimisant le parcours (somme des distances).



Etats : Villes
Opérateurs : Aller à Ville x avec préconditions :
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

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

Problème d'analyse syntaxique

« abaabab » est-il un mot (M) du langage défini par les règles suivantes :

- ab ⟶ M
- aM ⟶ M
- Mb ⟶ M
- MM ⟶M

Ci-après le graphe complet des états où un état correspond à la chaine réduite, initiale ou finale, l'opérateur correspond à une règle de réduction.
En termes d'opérateurs, la suite 1, 1, 1, 2, 4, 4 conduit de l'état initial à l'état final.
En terme d'états : abaabab, Maabab, MaMab, MaMM, MMM, MM, M

Exercice 1 : 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 :

1- ( ) ⟶ P

2- P ⟶ A
3- A,A ⟶ A
4- (A) ⟶ P

Réponse : La solution en termes d'opérateurs : 1, 2, 3, 4, 2, 3, 3, 4

Exercice 2 :

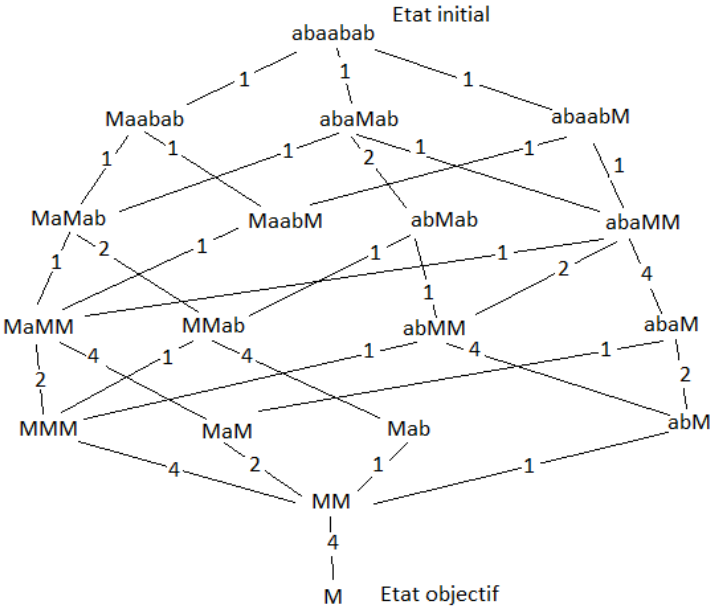
Choisir une description d'états, puis des opérateurs pour le problème suivant et le résoudre.

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 :

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

Solution en termes d'opérateurs : 1-2-1-2-1-3



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

2.2.1 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.***
- 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.

2.2.2 Méthodes aveugles

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

a)- Cas des graphes de type arborescence

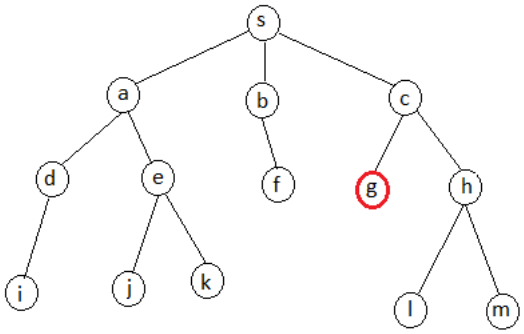
Le principe est donné par l'algorithme suivant :

Algorithm
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)
While((!OPEN_Empty) && (!Found))
Do
N= **Défiler** (OPEN)
If(succ(n) !=Ø) then
Soient (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) n_o= n_j ; Found=true
EndDo
If(!Found) **then** // Echec
else reconstruire la solution de s vers n_o **EndIf**

End

Exemple :



Les nœuds explorés, enfilés dans la file, étape par étape :
s
(a,s), (b,s), (c,s)
(d,a), (e,a)
(f,b)
(g,c), (h,c) Arrêt car g est un nœud objectif.

Propriétés de la méthode :

Si l'espace complet d'états comporte un nœud objectif :

- Breadth-first en trouvera un

b)- Cas des graphes quelconques

L'algorithme est donné comme suit:

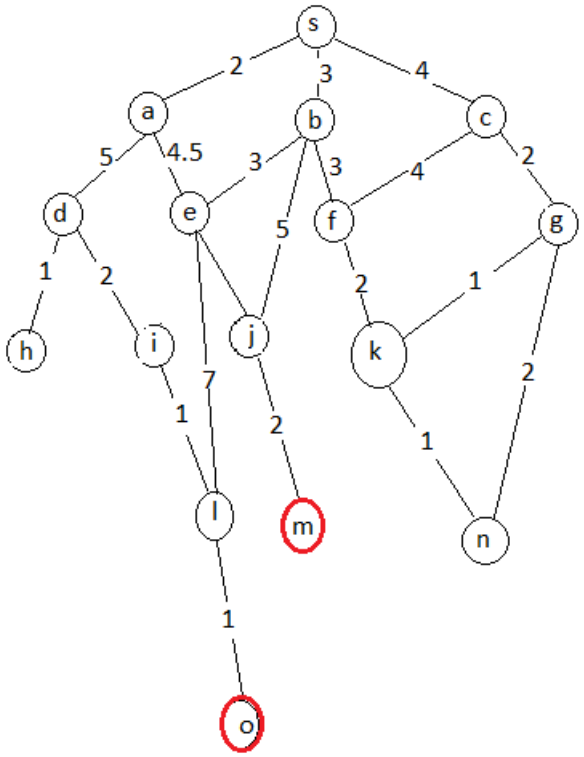
Algorithm
Begin
s : Etat initial
n_o: Nœud objectif
OPEN: File initialement vide
succ: fournit les successeurs d'un nœud
boolean Found= False
If(s== n_o) **then** Found=true // Succès

Else
If(Succ(s)==Ø) **then** Found= False // Echec
Else
Enfiler (OPEN, s)
While ((! OPEN _Empty) && (!Found))
Do

n= **Défiler** (OPEN)
Enfiler (CLOSED, n)
If(succ(n= !=Ø) then
Soient (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 == n_o) Found=true

EndDo
If(!Found) **then** // Echec **else** reconstruire la solution de s à n_o **EndIf**

End



Exemple :

OPEN

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	s,a,b,c,d,e,j

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

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

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

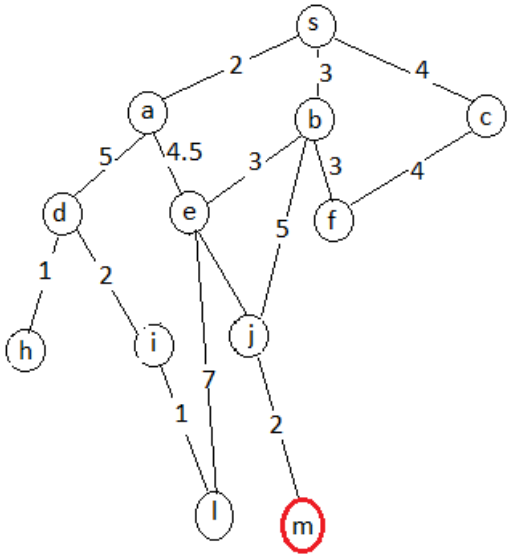
- S : nœud initial
- OPEN : Pile vide initialement
- L : paramètre (profondeur) limitant la distance à la racine (en nombre d'arcs)

On développe d'abord les nœuds les plus récemment engendrés.
L'algorithme est le suivant :

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
If(!Found) **then** // Echec
else Reconstruire la solution de s à n_o **EndIf**
EndIf
EndIf
End

Exemple : Appliquons l'algorithme sur l'espace d'états précédent.
Pour L=3 :



OPEN=
{s},
{(a,b,c)},
{(d,e), (b,c)},
{(h,i),(e,(b,c))},
{(l,j),(b,c)},
{(j,f), (c)},
{(m),(f),(c)}
m objectif

Solution : s-b-j-m, coût=10, 3 arcs, 6 développement, 12 nœuds apparus

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).

2.2.3 Méthode du coût uniforme

Principe

Dans certains problèmes, le passage d'un état n_i à un état n_j est considéré comme coûtant une quantité C_{ij}. Le coût d'une séquence d'opérateurs est calculé par cumul des coûts associés à chacun : on l'appellera distance. Nous supposons que les coûts sont positifs.

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).

L'algorithme est donné comme suivant :

Algorithm
Begin
s : Etat initial
n_o: Nœud objectif
OPEN, CLOSED: Files initialement vide
succ: fournit les successeurs d'un nœud
bool Found= False
If(s== n_o) **then** Found=true // Succès
Else
If(Succ(s)==Ø) **then** Found= False // Echec
Else
Enfiler (OPEN, s), $\hat{g}(s)=0$
While((! OPEN _Empty) && (!Found))
Do
    n_i = **Défiler** (OPEN)/ $\hat{g}(n_i)$ est minimal
    Enfiler(CLOSED, n_i)
    If(n_i == objectif) n_o= n_i ; Found=true
    Else
    If(succ(n_i !=Ø) then

Soient (n_{1i}, n_{2i}, .., n_{ki}) les successeurs de n_i
Enfiler (OPEN, n_{ji}), $g^{\wedge}(n_{ji})= g^{\wedge}(n_j) +C_{ij}$, i=1..k (*)
Créer un lien de n_{ji} vers n_i (**)
EndIf
EndDo
If(!Found) **then** // Echec **else** reconstruire la solution de s à n_o **EndIf**
End

Pour un graphe quelconque, ajouter :

- à (*) : s'il n'appartient pas à OPEN et à CLOSED. Si un successeur est dans 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.

2.2.3 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 f^(n) la valeur du nœud n, f^ est dite fonction d'évaluation.

La notation f^(n) est trempeuse : f^ peut ne pas dépendre que de n mais aussi de l'état du développement de la recherche.
Nous convenons que n est d'autant plus prometteur que f^(n) est petite.

Algorithme :

Algorithm
Begin
s : Etat initial
n_o: Nœud objectif
OPEN, CLOSED: Listes initialement vide
succ: fournit les successeurs d'un nœud
f^(n) is une fonction heuristique associée au nœud n
bool Found= False
If(s== n_o) **then** Found=True // Succès
Else
Calculer f^(s), Op(s)=f^(s)
Insérer (OPEN, (s, Op(s))),
While((! OPEN _Empty) && (!Found))
Do
Soit N= {n_i de OPEN tel que (n_i, Op(n_i)) est minimal}
If(n_i == n_o) **then** Found=True
Else
Choisir aléatoirement n_j from N
Rtirer (n_j, Op(n_j))
Insérer (n_j, C(n_j)) dans CLOSED
If(succ(n_j !=Ø) **then**
Soient (n_{1j}, n_{2j}, .., n_{kj}) les successeurs de n_j
Calculer f^(n_{lj}) for l=1..k
For each n_{lj}
DO
If n_{lj} !€ OPEN **then**

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

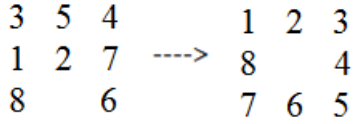
Remarques :

- 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.
- On écrira f^(n)=g^(n)+h^(n) où g^(n) est le coût du chemin parcouru et h^(n) est une estimation du coût du chemin qui reste à parcourir du nœud n vers l'objectif
  Si 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.
- Si 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.

Exercice 1:

On veut passer de la configuration initiale vers la configuration finale du jeu de Taquin en appliquant la méthode de recherche ordonnée (Algorithme A).
Nous définissons f^(n)=g^(n)+w(n) où :

- g^(n) = la distance minimale (en nombre d'arcs) de la racine s jusqu'au nœud n courant)
- w(n) est le nombre de cases (non vides) de la configuration n qui ne sont pas à leur place par rapport à la configuration objective



Exercice 2:

Reprendre l'exercice 1 où w(n) est la somme des distances (en nombre de pas) de chaque case non vide de n par rapport à son assignation dans l'objectif

Exercice 3:

Reprendre l'exemple du voyageur de commerce. Proposez une fonctions w(n)

Exemple : w(n) = nombre villes non traversées*coût min