Cours

Intelligence Artificielle

Résolution de problèmes

Auteur : Prof. Slimane LARABI
Faculté d'Informatique, USTHB
Décembre, 2022

Sommaire

Préface

Préface

Prof. Slimane LARABI

Ce polycopié du cours d'Intelligence Artificielle, dédié à la résolution des problèmes est le résultat de mes efforts consentis durant les 07 années d'enseignement de ce cours aux étudiants de première année de Master Informatique Visuelle à l'université des Sciences et Technologie Houari Boumediene.

Dans ce cours nous aborderons quatre problèmes fondamentaux de l'Intelligence Artificielle :

- Modélisation de l'espace d'états d'un problème et recherche du plus court chemin pour atteindre l'objectif. L'algorithme A* qui se base sur une heuristique est présenté et expliqué avec des exemples.
- Les algorithmes intelligents pour les jeux (MinMax, Negamax) qui permettent de faire jouer un ordinateur contre un humain en lui communiquant l'intelligence utilisée par l'humain pour gagner le jeu. L'optimisation de ces algorithmes avec l'élagage beta et ensuite alpha-beta est justifiée et expliquée. Cet élagage permet de réduire l'exploration de l'arbre de jeu lors de la recherche du meilleur coup à jouer.
- Les métaheuristiques qui sont des techniques d'optimisation algorithmiques pour résoudre des problèmes d'optimisation complexes. Elles permettent d'explorer de grandes et complexes espaces de solutions, dans lesquels les méthodes de recherche traditionnelles sont inefficaces.

Ces algorithmes heuristiques utilisent des stratégies intelligentes et des connaissances empiriques pour guider leur exploration de l'espace de recherche. Nous borderons dans ce cours dans l'ordre : La recherche aléatoire, la recherche locale, la méthode Hill-Climbing (escalade), la méthode du recuit simulé, la méthode de recherche Tabou et les algorithmes génétiques.

- Problème de satisfaction de contraintes (CSP) qui concerne les problèmes NP-Complexes pour lesquels il n'existe pas d'algorithme polynomial connu pour les résoudre de manière exacte. Pour les résoudre, on utilisera un des algorithmes heuristiques tels que les algorithmes de recherche locale, de recherche en arrière (BackTracking) et AC-3 (satisfaction de contraintes).

Nous terminons ce polycopié, par donner les énoncés des exercices réalisés en travaux dirigés et pratiques.

Chapitre 1.

Résolution de problèmes

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

1.1.1 Définitions :

Un espace d'états est défini à l'aide des éléments suivants :

- 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

1.1.2 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).

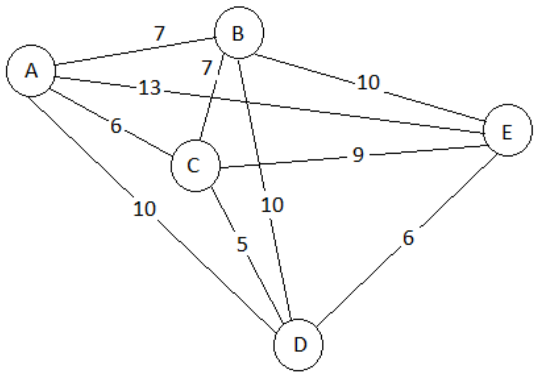


Figure 1.1. Exemple du problème du voyageur du commerce

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 :

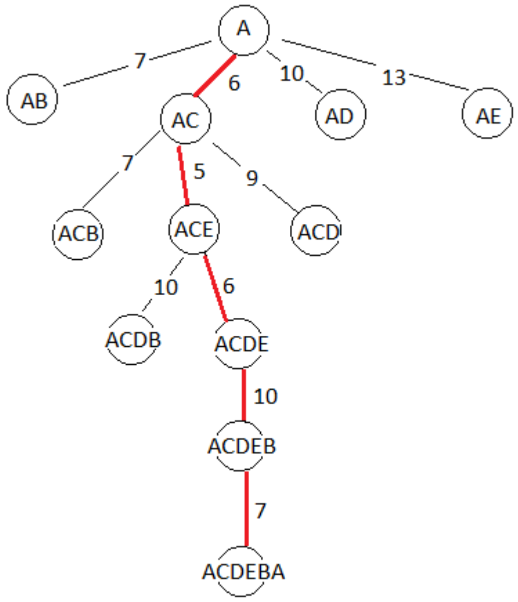


Figure 1.2. Espace d'états associé au problème du voyageur de commerce

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 opérateur de réduction n° 1
- aM ⟶ M opérateur de réduction n° 2
- Mb ⟶ M opérateur de réduction n° 3
- MM ⟶M opérateur de réduction n° 4

Le graphe de la figure 2 illustre l'espace d'états où un état correspond à la chaine réduite, initiale ou finale, l'opérateur correspond à une règle de réduction.

Etat initial est donné par la chaîne « abaabab ».

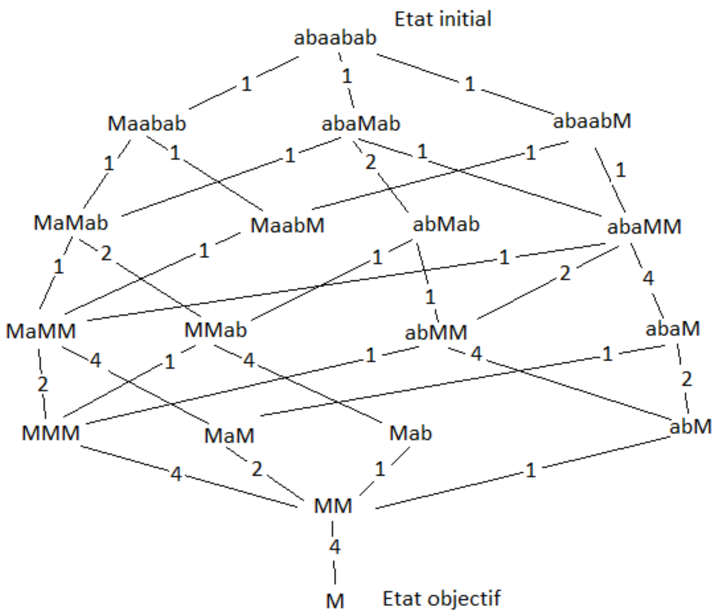


Figure 1.3. Espace d'états pour le problème de réduction

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 :

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 de triplets (Bidon1, Bidon2, Bidon3), voir espace d'états de la figure 1.4.

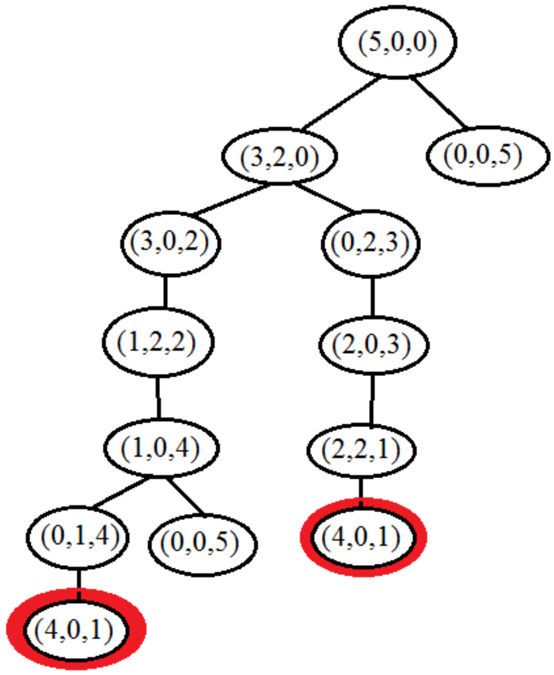


Figure 1.4. Espace d'états du problème de transfert du contenu des bidons.

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 :

1- ( ) $\longrightarrow$ P
2- P $\longrightarrow$ A
3- A,A $\longrightarrow$ A
4- (A) $\longrightarrow$ P

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

1.2.1 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 orientées 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.

1.2.2 Méthodes aveugles (non informées)

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

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

Cas des graphes de type arborescence

Le principe est donné par l'algorithme 1 suivant :

Exemple :

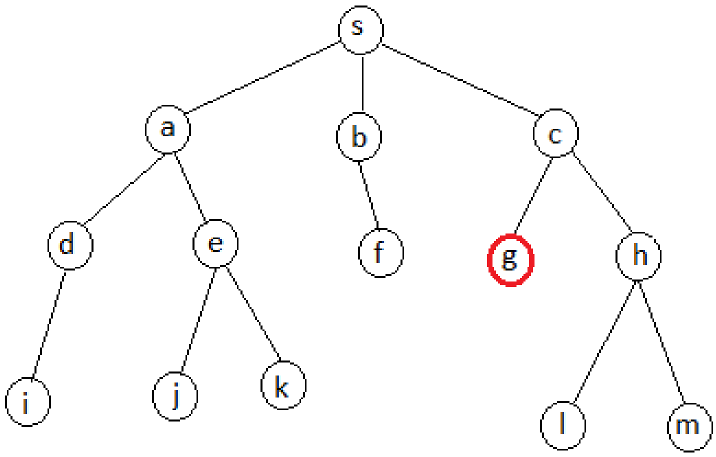


Figure 1.5. Exemple d'une arborescence

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 // Success
Else
If(Succ(s)==Ø)
Then Found= False // Echec
Else
Enfiler (OPEN, s)
While ((! OPEN_Empty) && (! Found))
Do
n= **Defiler** (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

EndIf EndIf

If(! Found) **then** // Echec

else reconstruire la solution de s vers n_o **EndIf**

End

Propriétés de la méthode :

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

Cas des graphes quelconques

L'algorithme est donné comme suit :

Algorithm

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 // Success

Else

If(Succ(s)==Ø) **then** Found= False // Echec

Else

Enfiler (OPEN, s)

While ((! OPEN _Empty) && (! Found))

Do

n= **Defiler** (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_0 **EndIf**

End

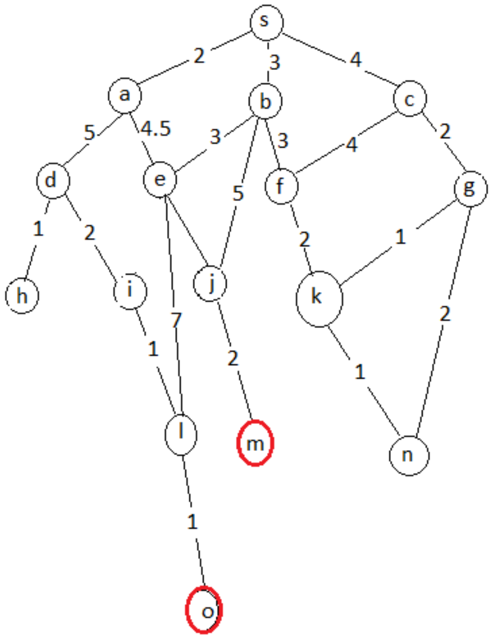


Figure 1.6

Open = S abc de jf g hi l m
Closed= S a b c d e f j

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, m	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

1.2.2.2 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 // Success
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 :

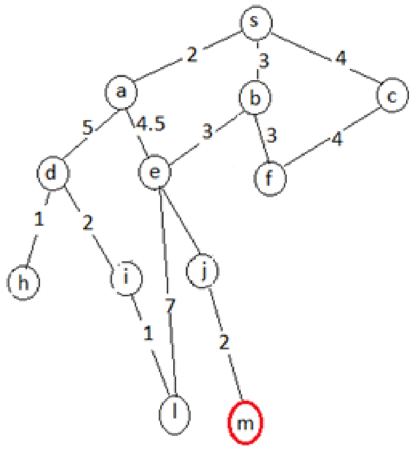


Figure 1.7

OPEN=
{s},
{(a, b, c)},
{(d, e), (b, c)},
{(h, i), (e, (b, c))}, // pour i Level=3, on ne développe pas.
{(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).

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.

1.2.3.1 Méthode du coût uniforme

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).

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 // Success
Else
If(Succ(s)==Ø) **then** Found= False // Echec
Else
Enfiler (OPEN, s), g^(s)=0
While ((! OPEN _Empty) && (! Found))
Do
    n_i = **Défiler** (OPEN)/ 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^(n_{ji})= g^(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_0 **EndIf**
End

Pour un graphe quelconque, ajouter :

- à (*) : 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.

Exemple :

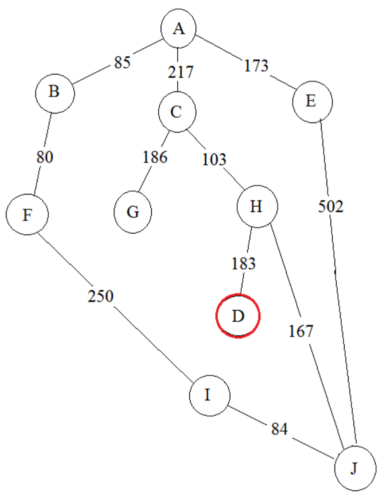


Figure 1.8

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)

~~(J,E,675)~~, (D,H,503),(J,H,487) (G,C,403)
~~(J,E,675)~~, (D,H,503), ~~(J,H,487)~~, (J,I, 499) (I,F, 415)
~~(J,E,675)~~, ~~(J,H,487)~~, (J,I, 499) (D,H,503)
FIN

Algorithme de Dijkstra [6]

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

Dijkstra et UCS sont identiques sauf que 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.

1.2.4 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.

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

Algorithm
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 // Success
Else
Calculer $\hat{f}(s)$, Op(s)= $\hat{f}(s)$

Insérer (OPEN, (s, Op(s))),
While((OPEN≠∅) && (!Found))
Do
Soit N= {n_i de OPEN tel que (n_i, Op(n_i)) est minimal}
If($n_i == n_0$) **then** Found=True
Else
Choisir aléatoirement n_j from N
Retirer (n_j, Op(n_j))
Insérer (n_j, C(n_j)) dans CLOSED
If(succ(n_j) ≠∅) **then**
Soient (n_{1j}, n_{2j}, .., n_{kj}) les successeurs de n_j
Calculer $\hat{f}(n_{lj})$ for $l = 1..k$
For each n_{lj}
DO
If n_{lj} ∉ OPEN **then**
Op(n_{lj})= $\hat{f}(n_{lj})$
Créer un lien du nœud 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($\hat{f}(n_{lj})$, Op(n_{lj}))
Mettre à jour le lien.
EndIf
If n_{lj} € CLOSED, **then**
Retirer de CLOSED (n_{lj},C(n_{lj})) tel que $\hat{f}(n_{lj})$ < 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 :

- $\hat{f}$ 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.

- 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

  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.

- 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.

- 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 :$h(n)<h(n_{i+1})+C(n_i, n_{i+1})$

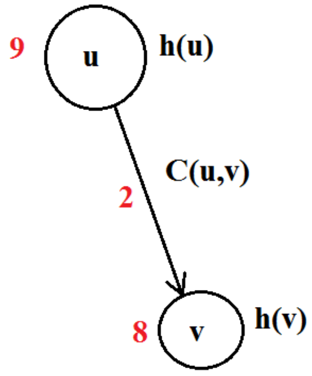
- Même si la consistance est plus restrictive que l'admissibilité, dans les applications réelles il est rare de trouver des heuristiques admissibles qui ne sont pas consistantes.

- 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.

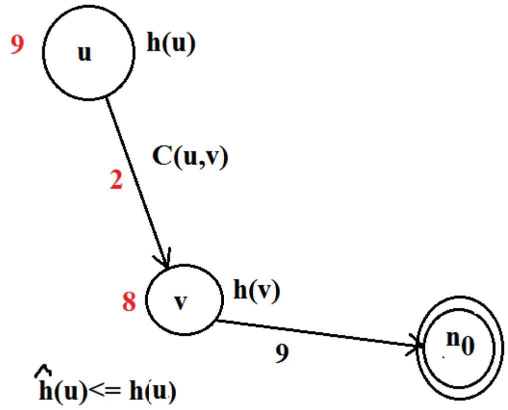
Définitions :

Une heuristique est **consistante (monotone)** si pour tout nœuds u, v connectés, si elle satisfait h(u) ≤ C(u, v) + h(v) où C(u, v) est le coût de transition de u à v.



h(u)<= h(v)+C(u,v)

Une heuristique est **admissible** si pour tout nœud n, $h(n) \ < \hat{h}(n)$, où $\hat{h}(n)$ est le coût réel de n au nœud objectif n_0.



Un algorithme A* est un algorithme de type A opérant avec une heuristique admissible.

<u>Théorème</u> :

Si une heuristique est consistante, alors elle est aussi admissible.

Démonstration

Soit N le nombre de nœuds de s à no.
Quel que soit le nœud n_i ayant comme successeur n_j, comme h est consistante, alors $h(n_i) \ \leq \ C(n_i, n_j) + \ h(n_j)$
On peut remplacer dans cette relation $h(n_j)$ par $h(n_j) \ \leq \ C(n_j, n_k) + \ h(n_k)$ où le nœud n_k est le successeur de n_j,

Nous obtenons ainsi :

$$h(n_i) \ \leq \ C(n_i, n_j) + \ h\big(n_j\big) \leq C(n_i, n_j) + \ C(n_j, n_k) + \ h(n_k)$$

En répétant le même processus, en remplaçant $h(n_k)$ puis $h(n_l)$ où n_l est le successeur de n_k jusqu'à ce qu'on arrive au dernier nœud qui mène au but $s\ tel\ que\ h(s) = 0$.

$$h(n_i) \leq C(n_i, n_j) + C(n_j, n_k) \ldots + C(n_s, s)$$

Donc $h(n_i) \leq \hat{h}(n_i)$ pour tout nœud n_i.

Propriété :

L'admissibilité signifie que, même lorsque l'espace de recherche est infini, si des solutions existent, une solution sera trouvée et le premier chemin trouvé sera une solution optimale - un chemin à moindre coût d'un nœud de départ à un nœud de but.

Pour l'algorithme A∗, l'optimalité du chemin est garantie lorsque l'heuristique est admissible ou monotone ou consistante.

Quand un nœud est-il visité deux fois ?

Considérons le graphique suivant, où l'heuristique satisfait la condition de sous-estimation de la longueur du chemin qui reste à parcourir, mais n'est pas monotone car h(b)>d(b, c)+h(c).

Lorsque nous visitons a, nous ajoutons b et c à la file d'attente avec les évaluations suivantes :

f(b)=d(a, b)+h(b)=1+8=9

f(c)=d(a, c)+h(c)=3+1=4

De toute évidence, **c** est visité en premier bien que le chemin le plus court vers c soit en fait via b. Plus tard, lorsque nous visitons b, c est à nouveau visité à partir de b et son poids total est mis à jour à 2.

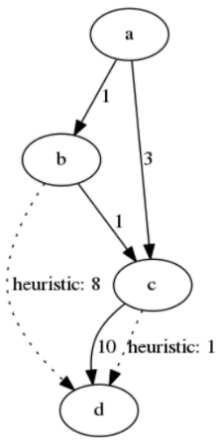


Figure 1.9 Espace d'états avec coûts et estimations h(n)

Stratégie pour trouver le chemin le plus court sans visiter un nœud deux fois.

Si a → b → c est plus court que a → c, nous voulons d'abord visiter b, donc nous atteignons c depuis b avant de l'atteindre depuis a. Si nous pouvons garantir cela, nous n'avons même pas besoin de le visiter plus tard, car a → b → c est de toute façon plus court.

Cela se résume à : Si $d(a,b)+d(b,c)<d(a,c)$

Si c'est vrai, il faut d'abord visiter b. On peut ajouter h (c) des deux côtés sans rien changer.

$$d(a, b)+d(b, c)+h(c)<d(a, c)+h(c)$$

Si la contrainte monotone est remplie, on peut remplacer d (b, c) + h (c) par h (b), car elle est inférieure ou égale. L'équation $\mathbf{d(a,b)+h(b)<d(a,c)+h(c)}$ est toujours vrai. C'est aussi le test que l'algorithme A ∗ utilise pour décider s'il visitera ou non b avant c. Si c'est vrai, il visite d'abord b. C'est exactement ce que nous voulions réaliser.

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

3	5	4		1	2	3
1	2	7	---->	8		4
8		6		7	6	5

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 fonction w(n)

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

Chapitre 2.

Algorithmes de recherche pour les jeux

2.1 Introduction

Il est possible de représenter le déroulement d'un jeu (comme exemple le jeu d'échecs), par un arbre où les nœuds correspondent aux différentes configurations du jeu (échiquier).

Par alternance, les nœuds d'un même niveau sont associés à un joueur. Les nœuds du niveau suivant sont alors associés au second joueur (voir figure 1).

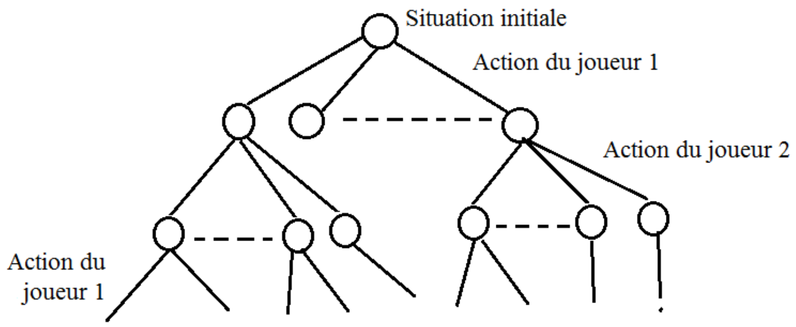


Figure 2.1. Représentation d'un jeu par un arbre

Pour chacun des joueurs, s'il veut choisir une action qui lui permettra de gagner, il doit explorer tout l'arbre (au maximum). Pour le jeu d'échecs, et pour une profondeur de 100, le nombre de nœuds visités, sachant que le nombre d'actions possibles à chaque étape est de 16, est de 16^{100}.

Il est donc impératif d'appliquer un algorithme intelligent pour espérer trouver une stratégie gagnante dans un temps raisonnable.

2.2 Procédure MinMax

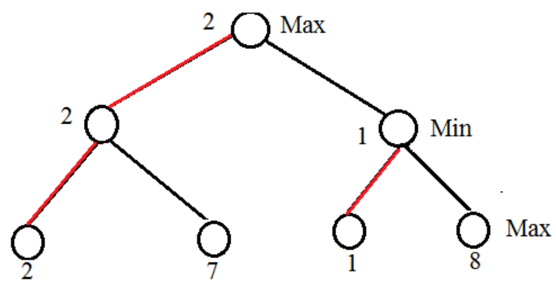


Figure 2.2. Exemple d'arbre de jeu avec Min Max

2 joueurs : J1 (jetons de couleur blanche), J2 (de couleur Noir).
J1 va récupérer les jetons Noirs, J2 récupère les Blancs.

Coût du jeu : Gain : fonction qui va Compter les jetons de couleur Noir.

J1 commence : maximiser le nombre de jetons noir (récupérés par J1)

J2 minimiser le nombre de jetons Noir (récupérés par J1)

On parlera de deux joueurs : Max Min.

Qui va commencer le jeu ?

Peu importe : Max Min (J1   J2)

Max, Min : Notation

Un joueur va maximiser un gain (minimiser celui de l'adversaire) et l'autre va le minimiser (maximiser le sien)

On peut représenter deux joueurs comme étant Max et Min où chacun va essayer d'optimiser le gain qui est donné par rapport au joueur Max.

Max va choisir les actions qui maximisent son gain alors que le joueur Min va choisir les actions qui permettront de minimiser le gain de Max.

Pour choisir l'action à opérer, Max doit explorer l'arbre est descendre jusqu'aux feuilles pour trouver quel chemin mènera au gain maximum en faisant l'hypothèse que Min tentera de minimiser ce gain à chacune de ses actions.

Cette façon d'opérer est appelée procédure MinMax et donc consiste à :

- Si le nœud est une feuille alors associer la valeur du nœud au gain du joueur se trouvant à ce niveau (Max ou Min).
- Si le nœud considéré est un nœud Max, alors appliquer la procédure MinMax à ses successeurs et affecter à ce nœud un gain égal au maximum des valeurs retournées par les nœuds fils.
- Si le nœud considéré est un nœud Min, alors appliquer la procédure MinMax à ses successeurs et affecter à ce nœud un gain égal au minimum des valeurs retournées par les nœuds fils.

L'écriture algorithmique est donnée comme suit :

Pour un nœud Max, la procédure suivante permet de retourner le max des valeurs de ses successeurs.

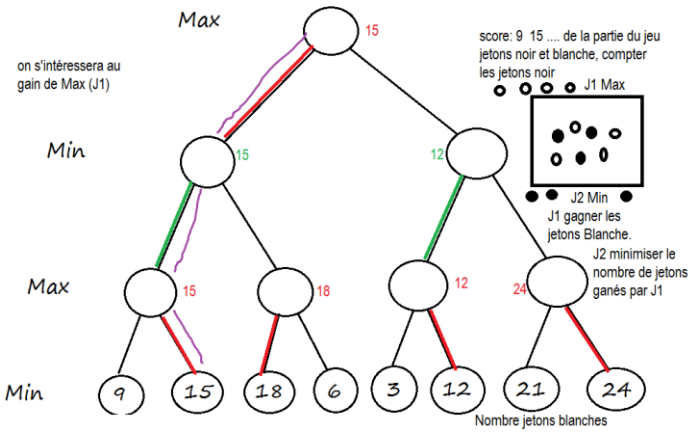


Figure 2.3

int Max(n)

{

-Déterminer les successeurs de n, soient $n_1, n_2, .., n_d$

// On va appliquer les actions possibles, et produire un nouvel état pour chaque action ce qui engendre l'ensemble des successeurs.

If(d==0) then return V(n) valeur fournie par l'heuristique choisie pour le nœud feuille

Else maximum=-∞

For (i=1 to d)

{

t=Min(n_i) // appel de la fonction min pour chacun des succ

If(t> maximum) then maximum =t;

}

return m

}

Pour un nœud Min, la procédure suivante permet de retourner le minimum des valeurs de ses successeurs.

Int Min(n)

{

-Déterminer les successeurs de n, soient $n_1, n_2, .., n_d$

If(d==0) Then return V(n)/ valeur fournie par l'heuristique choisie pour le nœud feuille

Else minimum=+∞

For (i=1 to d)

{

t=Max(n_i) // appel à max pour tous les succ

If (t< minimum) then minimum =t;

```
            }
    return minimum
    }
```

Exemple 1 : **Jeu de Grundy** :

Il s'agit de partager une pile de jetons de deux piles de hauteurs différentes. Le joueur qui est dans l'incapacité de jouer perd la partie.

Appelons les deux joueurs Max et Min, et supposons que Min joue en premier.

Le gain +1 est associé au nœud si Max gagne et -1 s'il perd.

Donnez l'arbre de jeu et donnez la stratégie pour que Min (premier joueur) gagne.

(Voir la solution sur la figure 2).

Exemple 2 : **Jeu de Fan-Tan** :

Considérons le jeu Fan-Tan à deux joueurs. On dispose de deux rangées d'allumettes contenant 1 et 3 allumettes.

Chaque joueur peut prendre d'une rangée autant d'allumettes qu'il le souhaite à la fois. Le joueur qui prend en dernier gagne.

Donnez l'arbre de jeu et donnez la stratégie pour que Max (premier joueur) gagne.

(Voir la solution sur la figure 3)

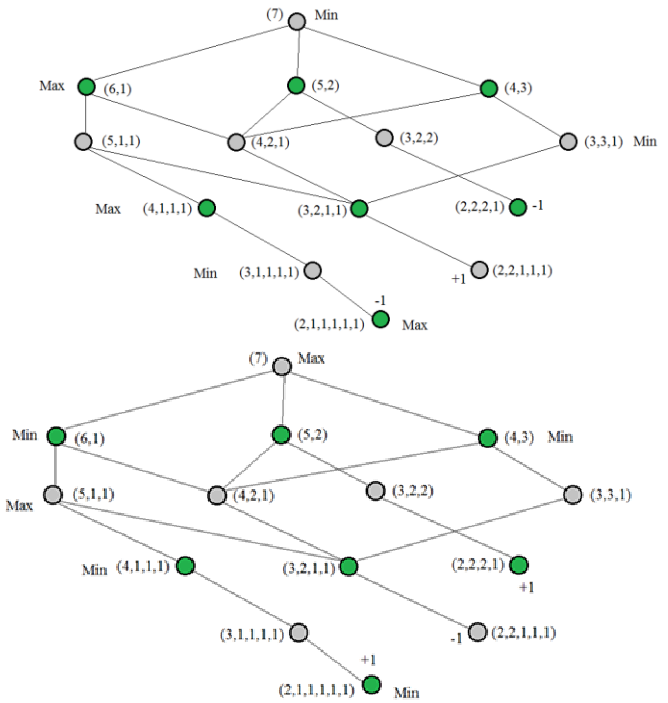


Figure 2.4 Arbre de jeu pour un état initial (Min ou Min, 7 jetons)

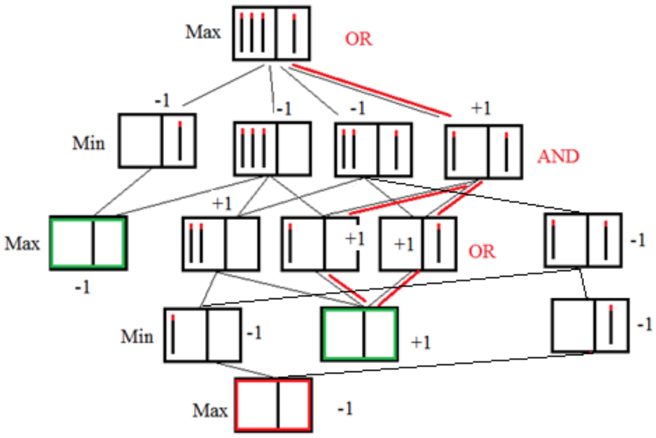


Figure 2.5. Arbre de jeu

2.3 Procédure NEGAMAX

Au lieu d'utiliser ces deux fonctions, on peut utiliser une seule NEGAMAX qui traite tous les nœuds de la même façon, on sélectionne toujours le max des opposés. Les valeurs des nœuds terminaux correspondants à Min sont initialement inversées comme illustré par les exemples des figures 4 et 5.

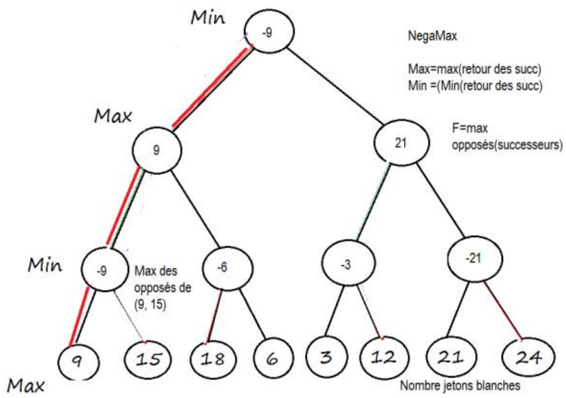


Figure 2.6

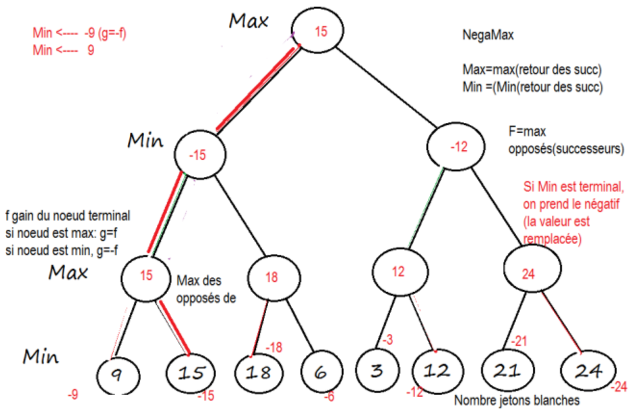


Figure 2.7.

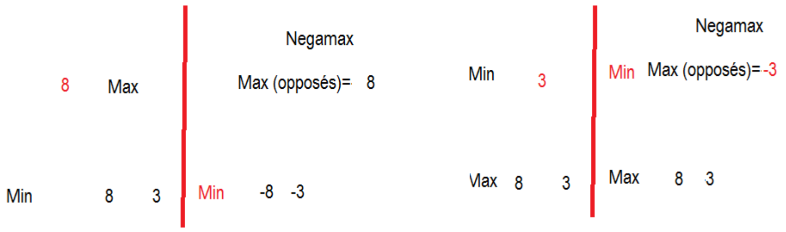


Figure 2.8. Equivalence entre les procédures minmax et Negamax

Int NEGAMAX(n)

{

-Déterminer les successeurs de n, soient n_1, n_2, .., n_d

If(d==0) then return V(n) / valeur fournie par l'heuristique choisie pour le nœud feuille

Else m=-∞

For (i=1 to d)

{

t=-NEGAMAX(n_i)

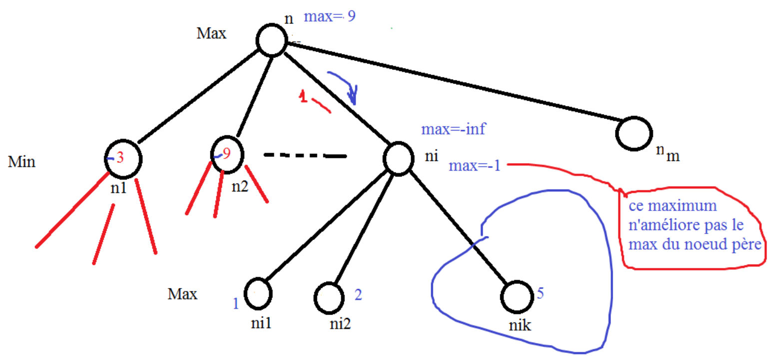
If(t>m) then m=t;

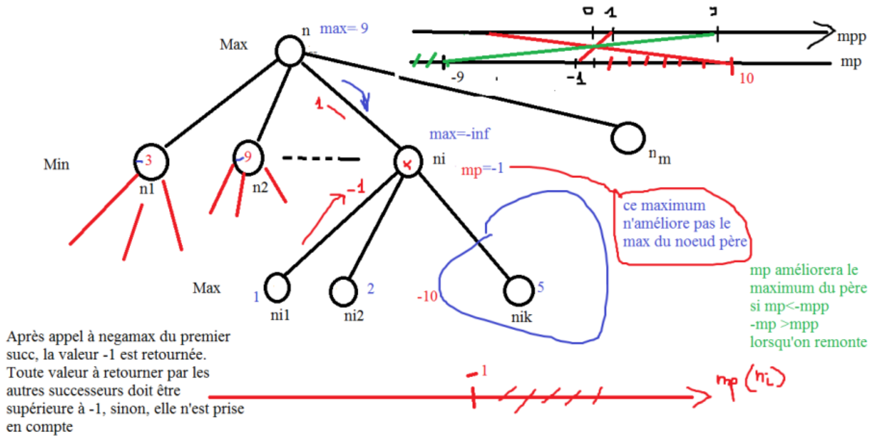
}

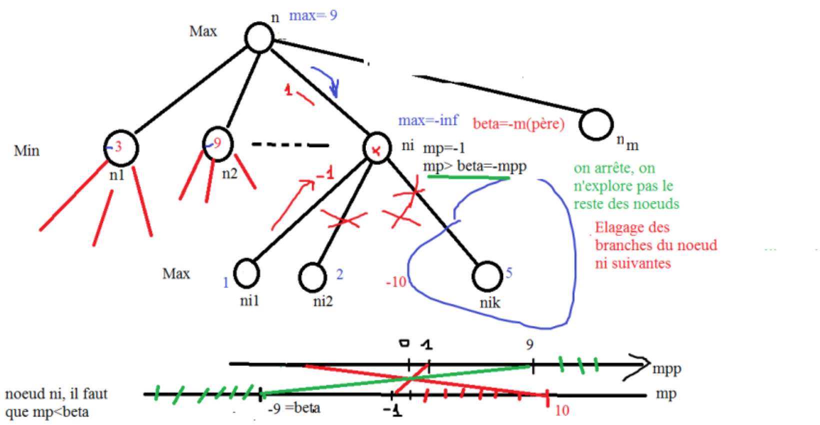
return m

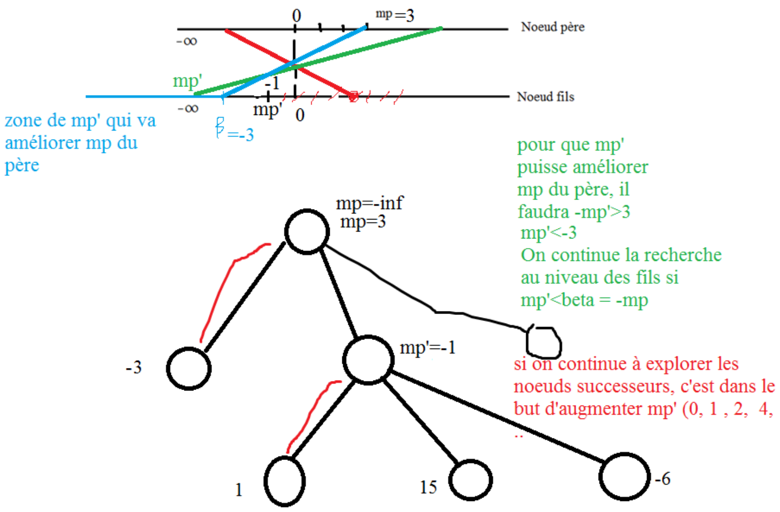
}

2.4 Borne supérieure beta (β)




Max
n max= 9
mpp
mp
-9
-1
10
1
Min
-3
n1
-9
n2
max=-inf
ni
mp=-1
nm
ce maximum n'améliore pas le max du noeud père
-1
Max
1
ni1
2
ni2
-10
5
nik
mp améliorera le maximum du père si mp<-mpp
-mp >mpp
lorsqu'on remonte
Après appel à negamax du premier succ, la valeur -1 est retournée. Toute valeur à retourner par les autres successeurs doit être supérieure à -1, sinon, elle n'est prise en compte
-1
mp (ni)


Max
n max= 9
1
Min
-3
n1
-9
n2
max=-inf
beta=-m(père)
ni
mp=-1
mp> beta=-mpp
nm
on arrête, on n'explore pas le reste des noeuds
Elagage des branches du noeud ni suivantes
-1
Max
1
ni1
2
ni2
-10
5
nik
1
9
mpp
mp
noeud ni, il faut que mp<beta
-9 =beta
-1
10


0
mp =3
Noeud père
-∞
mp'
-1
Noeud fils
-∞
mp'
0
β =-3
zone de mp' qui va améliorer mp du père
pour que mp' puisse améliorer mp du père, il faudra -mp'>3
mp'<-3
On continue la recherche au niveau des fils si mp'<beta = -mp
mp=-inf
mp=3
-3
mp'=-1
si on continue à explorer les noeuds successeurs, c'est dans le but d'augmenter mp' (0, 1 , 2, 4,
1
15
-6

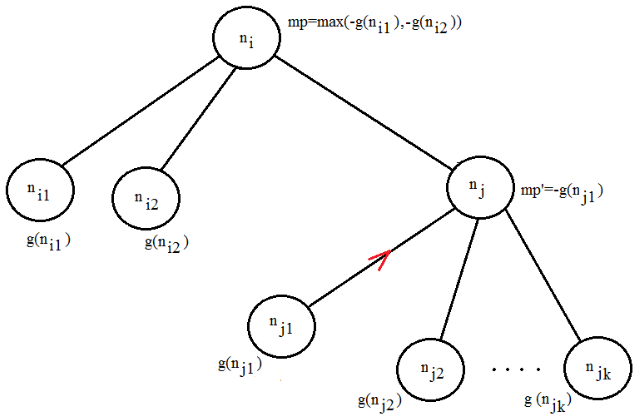


Figure 2.9. Exemple d'un arbre de jeu

Sur la figure 4, le maximum provisoire associé au nœud n_i est calculé en utilisant les nœuds n_{i1} , n_{i2} comme étant le maximum de – $g(n_{i1})$ et de – $g(n_{i2})$.

L'exploration du nœud fils n_j aura un effet sur le maximum provisoire si l'évaluation du nœud n_j donne une valeur qui mp' qui vérifie – mp' $> mp$ et donc $mp' < -mp$.

Pour que mp soit mis à jour, il faudra que l'évaluation du nœud fils n_j donne un valeur mp' qui doit vérifier : $-mp' > mp$ ce qui est équivalent à $mp' < -mp$.

Si on note $\boldsymbol{\beta} = -mp$, alors si $mp' > \boldsymbol{\beta}$ alors il n'est pas utile de continuer à évaluer les nœuds fils de n_j, car les valeurs retournées $-g(n_{jl})$ seront retenues que si elles sont supérieures à mp'. $-g(n_{jl}) > mp'$ et comme $' > \boldsymbol{\beta}$, ces nœuds vont mettre à jour mp' mais qui reste supérieur à β.

Conclusion : On évaluera le maximum provisoire du nœud n_j tant qu'il est inférieur à β. On arrêtera dès qu'il dépasse β. Les nœuds restants ne seront pas explorés et l'évaluation de nœud ni ne changera pas. On parle alors d'élagage β (voir figure 5).

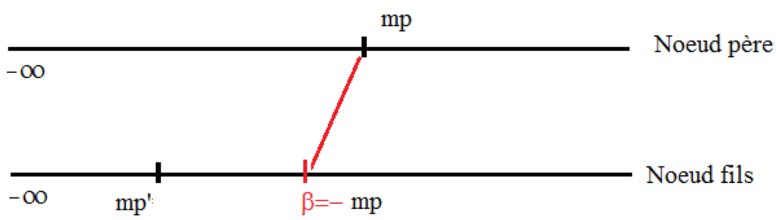
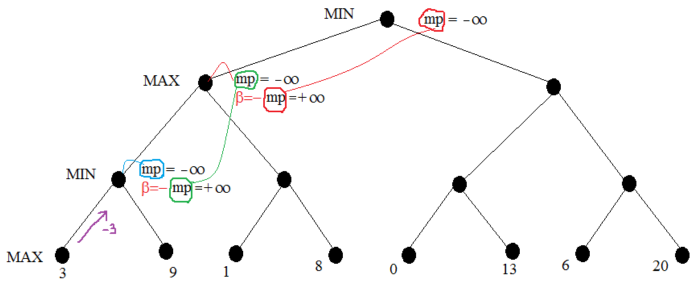


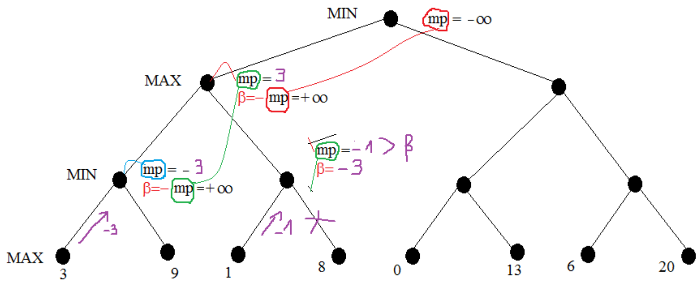
Figure 2.10 Evolution de mp' relativement à beta

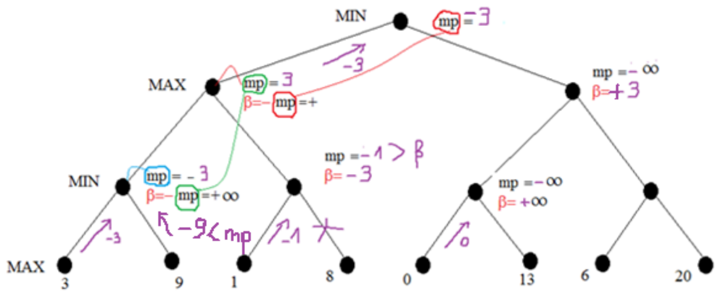
L'utilisation de la borne supérieure beta (**β**) se fait selon la procédure suivante:

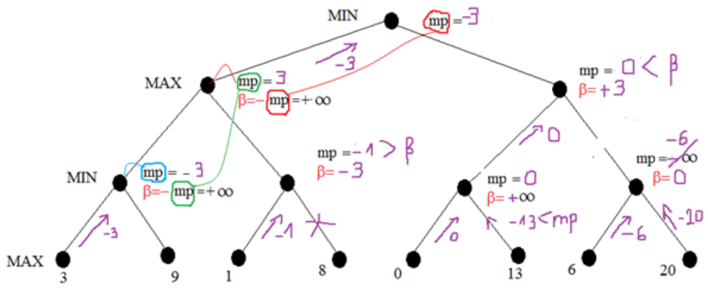
```
Int NEGAMAX(n, β)
{
-Déterminer les successeurs de n, soient n1, n2, .., nd
If(d==0) then return V(n) / valeur fournie par l'heuristique choisie pour le nœud feuille
        Else m=-∞; i=1;
        While((i≤d)&&(m<β))
                {
                t=-NEGAMAX(ni, -m)
                If(t>m) then m=t;
                i++ ;
                }
return m
}
```

Exemple :









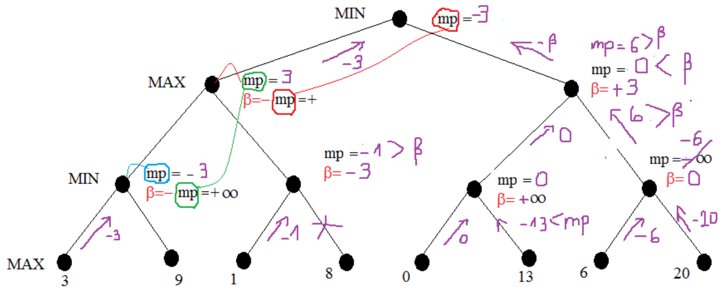


Figure 2.11. Déroulement Negamax avec élagage beta

2.5 Borne inférieure beta (α)

L'utilisation de la borne inférieure alpha (α) se fait selon la procédure suivante. L'appel du nœud racine se fait avec : $\alpha=-\infty, \beta=+\infty$

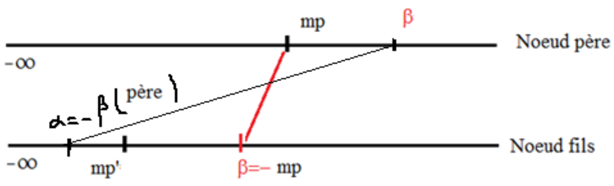


Figure 2.12

int NEGAMAX(n, α,β)
{
-Déterminer les successeurs de n, soient n_1, n_2, .., n_d
If(d==0) then return V(n) / valeur fournie par l'heuristique choisie pour le nœud feuille
Else m=α; i=1;
While((i≤d)&&(m<β))
{
t=-NEGAMAX(n_i, -β, -m)
If(t>m) then m=t;
i++ ;
}
return m
}

<u>Résultat :</u>

Si *mp* d'un nœud est inférieur à α de ce nœud, alors il y aura élagage des nœuds fils suivants du nœud père comme le montre la figure suivante.

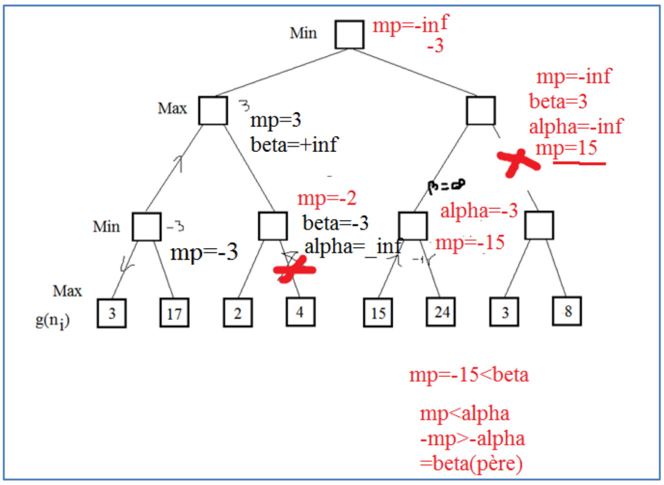


Figure 2.13

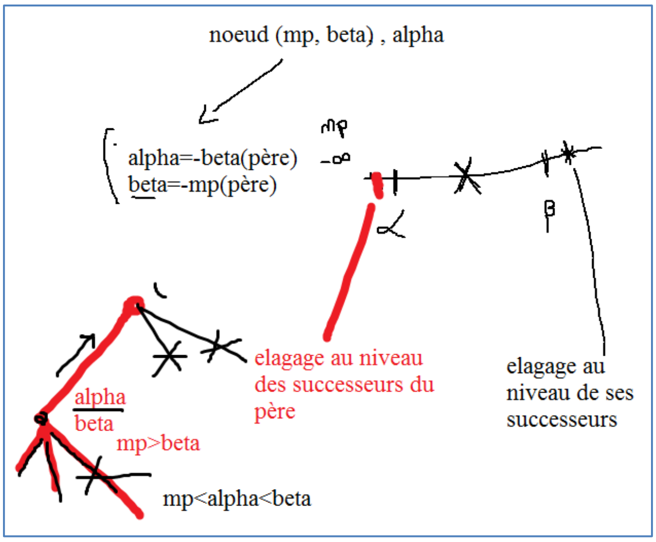


Figure 2.14

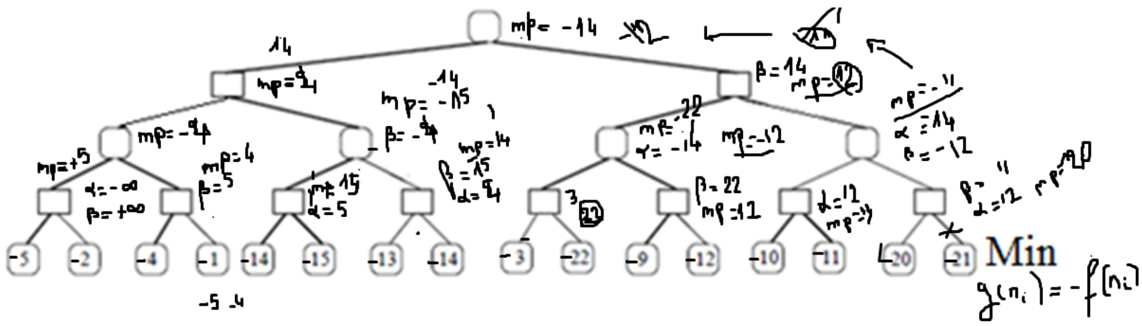


Figure 2.15

Chapitre 3.

Les méta-heuristiques

3.1 Introduction

3.1.1 Définitions

Ci-après quelques définitions :

-Une méta heuristique est un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficile pour lesquels on ne connaît pas de méthode classique plus efficace.

-Une **méta-heuristique** est une méthode générique pour la résolution de problèmes combinatoires NP-difficiles.

-Une méta heuristique est un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficile pour lesquels on ne connaît pas de méthode classique plus efficace.

-Une **méta-heuristique** est une méthode générique pour la résolution de problèmes combinatoires NP-difficiles.

-Metaheuristic is a high level problem independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms» (Sorensen and Glover 2013).

-A metaheuristic is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with imperfect information or limited computation capacity » (Bianchi et al 2009)

3.2 Problèmes d'optimisation combinatoire

Plusieurs problèmes réels consistent à rechercher une solution optimale à partir d'un espace d'états fini.

Cette tâche est souvent très difficile en raison de sa complexité.

Comme exemples, nous citons :

- Recherche du plus court chemin
- Obtention du coût minimum pour la délivrance de marchandises à des clients
- Assignation optimale de tâches

- Problème de routage d'internet

Algorithmes de recherche

Ce type de problèmes est généralement caractérisé par :

- Le besoin de groupement (clustering), tri, assignation à des objets vérifiant des contraintes
- Il est relativement facile de trouver une solution primaire
- Il est à complexité difficile (NP-hard), et donc les algorithmes classiques ne sont plus efficaces et impossible à utiliser
- Trouver une solution proche de l'optimal en un temps raisonnable est une option possible.

Problème classique : Voyageur du commerce

Pour N villes, il y a N ! routes.

Pour 10 villes, et à raison d'une micro seconde pour le calcul d'une route, il faut 3 secondes pour calculer les 10 ! Routes, 39 secondes pour 11 villes, 77 146 années pour 20 villes.

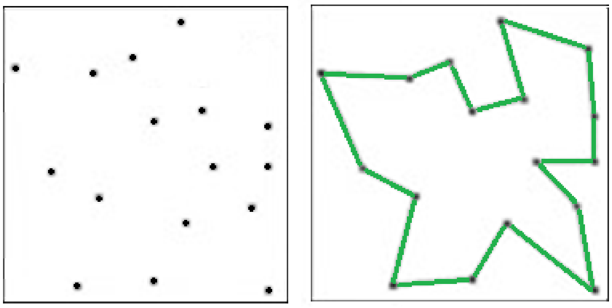


Figure 3.1. Exemple du problème du voyageur de commerce et du chemin optimal

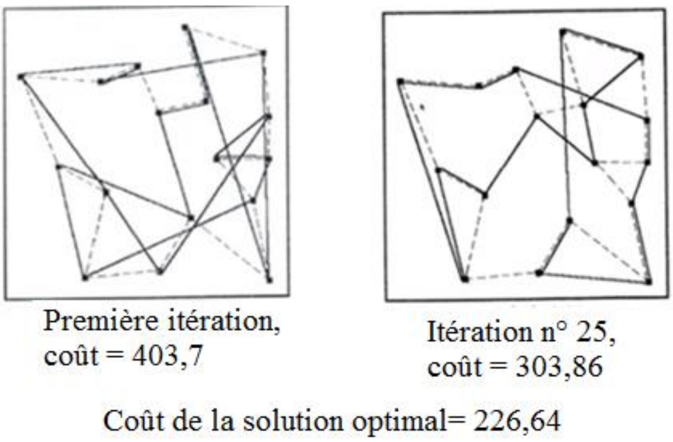


Figure 3.2. Recherche de la solution optimale pour le problème du voyageur de commerce

3.3 Meta heuristiques

Classification

Différents critères sont utilisés :

- Inspirées de la nature /non inspirée de nature
- Usage de la mémoire/ méthodes sans mémoire
- Déterministe / Stochastique
- Itérative / Gloutonne (suit le principe de réaliser, étape par étape, un choix optimum local, afin d'obtenir un résultat optimum global.)
- Recherche basée population / Recherche basée sur une seule solution

Inspirées de la nature /non inspirée de nature

Plusieurs meta heuristiques sont inspirées des processus naturels :

- Algorithmes évolutionnaires, systèmes immunitaires inspirés de la biologie
- Optimisation par essaims particulaires (simuler le déplacement d'un groupe d'oiseaux) et Algorithme de colonies de fourmis (Ant colony optimization algorithms), algorithme de colonie d'abeilles artificielles
- Recuit simulé de physique

Usage de la mémoire/ méthodes sans mémoire

Certaines méthodes (tabu search) utilisent des informations extraites durant la recherche pour (short-term) et (long-term).

Pour d'autres, pas d'informations extraites et utilisées dynamiquement durant la recherche (recherche local, recuit simulé)

Déterministe / Stochastique

Une Meta heuristique résout un problème d'optimisation en faisant des décisions déterministes (recherche local, recherche tabou)

Une meta heuristique stochastique applique des règles aléatoires durant la recherche (recuit simulé, algorithmes évolutionnaires)

Dans un algorithme déterministe, l'utilisation d'une même solution initiale, permet d'aboutir à une même solution finale.

Alors que dans le cas stochastique, différentes solutions finales peuvent être obtenues.

Recherche basée population / Recherche basée une seule solution

La recherche basée sur une seule solution (recherche locale, récuit simulé) manipule et transforme une seule solution (exploitation orientée).

La recherche basée sur une population (particle swarm, algorithme évolutionnaire) manipule un groupe de solutions (exploration orientée)

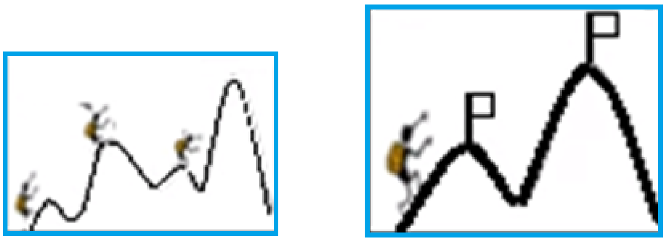


Figure 3.3 Population-based search Single solution based research

Itérative / Gloutonne

La majorité des métaheuristiques sont itératives.

Un algorithme itératif, commence par une solution (population de solutions) et la transforme à chaque itération en appliquant des opérateurs.

Algorithme Glouton, commence par une solution vide, et à chaque étape, une variable de décision est assignée jusqu'à l'obtention de la solution finale.

Mécanismes

Il y a un compromis entre deux objectifs dans la recherche :

- Diversification ou exploration : générer plusieurs solutions pour explorer l'espace de recherche sur une échelle globale.
- Intensification ou exploitation : focaliser la recherche dans une région locale sachant qu'une bonne solution a été déjà trouvée dans cette région

Il faut donc équilibrer (balance) pour :

Identifier rapidement l'espace de recherche ayant des solutions de bonnes qualité Ne pas rester longtemps dans un espace déjà exploré ou bien si la solution n'est pas prometteuse.

Chaque meta heuristique applique différentes stratégies pour concilier ces deux stratégies.

3.4 La recherche aléatoire

La recherche aléatoire est le cas extrême d'une exploration

Algorithme Début

Best <- Solution initiale déterminée aléatoirement

Répéter

S <- une solution candidate aléatoire

Si Qualité(S)> Qualité (Best) **Alors** Best <- S Fsi

Jusqu'à ce que Best est la solution idéale ou bien le temps d'exécution dépasse un seuil alloué.

Retourner Best Fin

Exemple : Problème du voyageur du commerce

Iteration	Solution
1	(1 2 4 3 8 5 9 6 7)
2	(9 6 4 7 8 5 1 2 3)
3	(2 4 1 5 8 3 9 7 6)
4	(4 7 5 1 8 3 2 6 9)
5	(7 6 9 5 8 3 1 2 4)
6	(8 3 7 2 1 5 7 6 9)
7	(2 5 1 3 9 8 4 6 7)
8	(1 4 2 3 8 5 6 9 7)
9	(3 4 2 1 5 8 7 9 6)
10	(7 4 9 3 8 5 6 2 1)

3.5 La recherche locale

La recherche locale fait partie de la famille des meta heuristiques basée sur une seule solution, elle met l'accent sur l'exploitation.

Algorithm
Begin
Input: Solution initiale so, t=0
Repeat
/ Générer des solutions candidates (voisinage partiel ou total) à partir de s_t/
Générer(C(s_t))
/ Sélectionner une solution de C(s_t) pour remplacer la solution courante
s_{t+1} =Select(C(s_t))
t=t+1
Until Critère d'arrêt vérifié
Return : Meilleur solution trouvée
End

Le voisinage C (s_t):

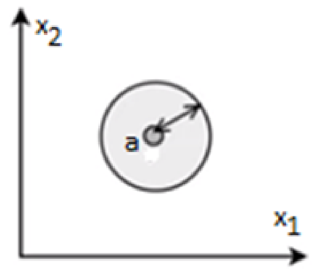


Figure 3.4. Le cercle représente le voisinage de a dans le cas de problème avec espace continu 2D

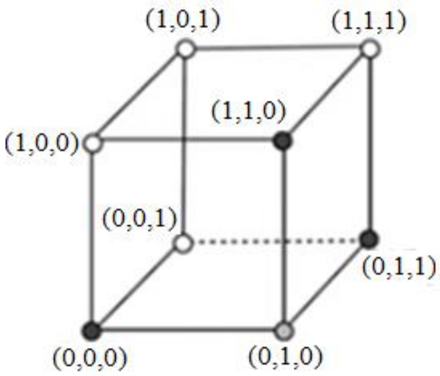


Figure 3.5. Les nœuds représentent les solutions du problème

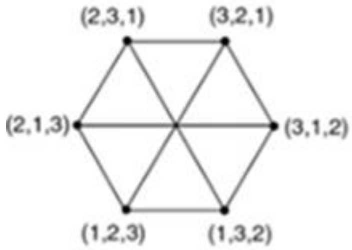


Figure 3.6. Deux solutions son voisines si elles diffèrent dans deux positions(swap)

Algorithme écrit autrement

Algorithm

Begin

s=so

While not Terminaison_Criterion **Do**

Generate (N(s)) / Generation of candidate neighbors/

If there is no better neighbor

Then Stop

s=s' // Select a better neighbor s' from N(s)

EndDo

Return Final solution found (local optima)

Meilleur Amélioration (**steepest descent**)

Dans cette stratégie, le meilleur voisinage est sélectionné. L'exploration du voisinage est exhaustive (toutes les possibilités de mouvement sont testées pour trouver la meilleur solution). L'algorithme est déterministe.

C'est un algorithme qui est time-consuming pour de voisinages larges.

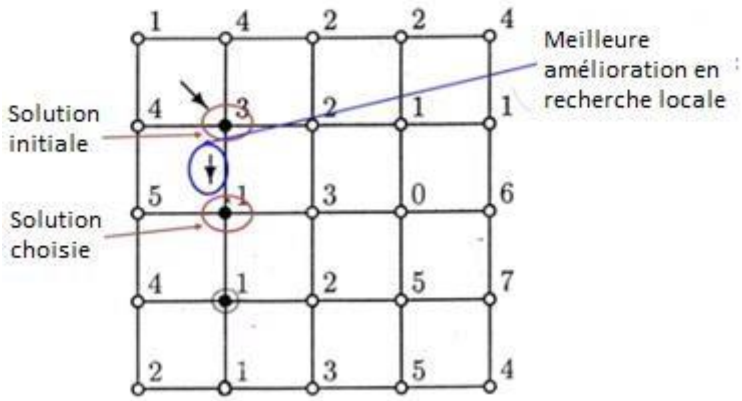


Figure 3.7. Illustration du cas de la meilleure amélioration

Première Amélioration

Dans la recherche locale de meilleure amélioration, la meilleure des solutions voisines remplace la solution actuelle à chaque itération

Dans la recherche locale, dans la première amélioration, le voisinage est exploré jusqu'à ce qu'aucune solution d'amélioration soit trouvée, qui remplace alors la solution actuelle.

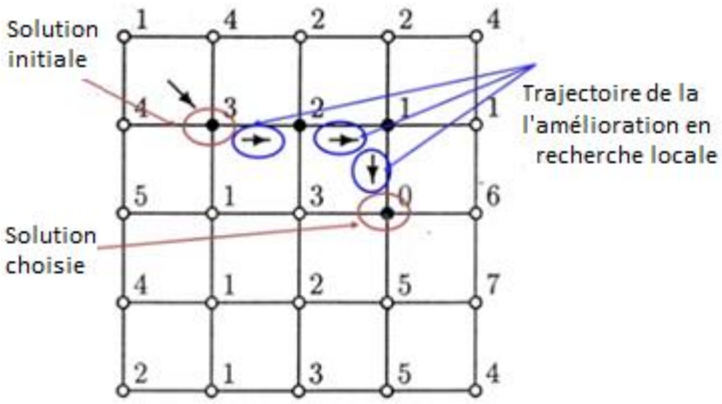


Figure 3.8. Illustration du cas de la première amélioration

3.6 Méthode Hill-Climbing (escalade)

C’est l’algorithme le plus simple de recherche locale.

Pour cet algorithme, on doit spécifier : Le nœud initial, la fonction objective F(n) à optimiser,

La fonction qui génère les successeurs d’un nœud n, Succ(ni)

Algorithme :

n_c=s (n_c est nœud courant)

Comparer n_c à ses voisins et sélectionner le meilleur n’, tel que : F(n’)>F(n_c), n_c=n’

(si on cherche à maximiser F(n) Si n' n'existe pas, on arrête la recherche
Retourner n_c comme solution.

Sur la figure 3.10, on présente un exemple où il s'agit de trouver un sommet dans une colline en cours d'exploration. F(n) est l'élévation du terrain. L'objectif est donc de maximiser F(n).

Dans le cas où la perception est difficile (présence de brouillard), le déplacement se fait pas à pas.

3.6 Méthode Hill-Climbing (escalade)

Cet algorithme ne garantit pas de trouver un maximum global. Si on démarre de s qui est à (current state), alors l'algorithme trouvera uniquement le maximum local. Par contre s'il démarre du second point de départ, il trouvera le maximum global. On peut aussi trouver un maximum sur un plateau. On peut aussi s'arrêter sur un plateau sans atteindre un maximum local.

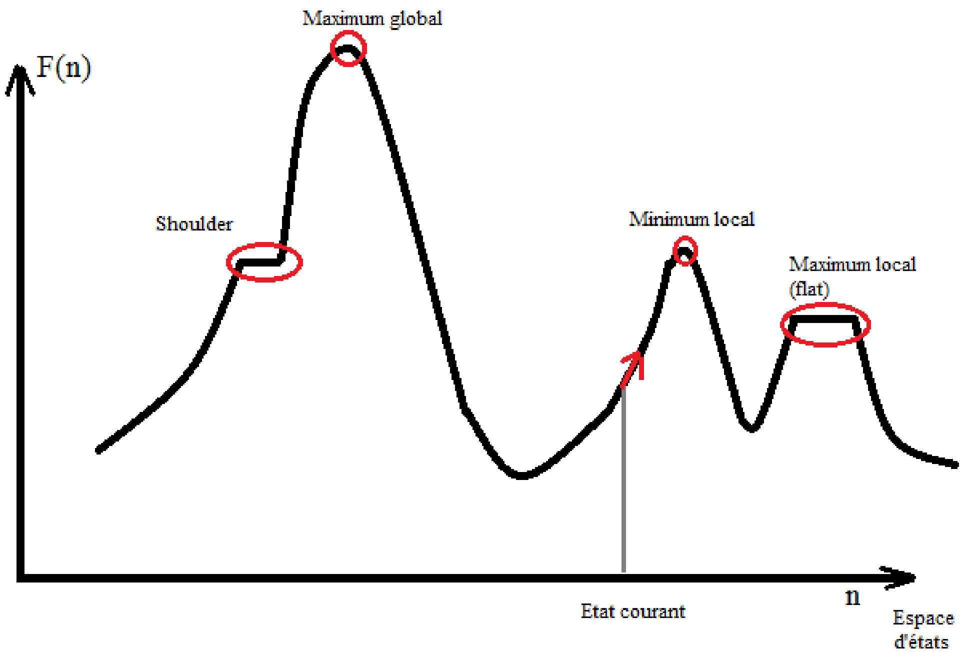


Figure 3.9. Exemple d'espace d'états à parcourir

Exemple d'exécution :

Nœud initial s=6, F(6)=2

On arrive à F(12)=10, on arrête

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(N)	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

3.7 Méthode Simulated annealing (recuit simulé)

C'est une amélioration de l'algorithme Hill-Climbing afin d'éviter de retourner un optimum local.

Il explore l'espace des solutions de telle sorte que le point de départ n'influence pas le résultat final.

En algorithmique, le recuit simulé est une méthode empirique (Méta heuristique) d'optimisation, inspirée d'un processus utilisé en métallurgie.

On alterne dans cette dernière des cycles de refroidissement lent et de réchauffage (recuit) qui ont pour effet de minimiser l'énergie du matériau.

On opère comme suit : Au lieu de choisir le meilleur voisin, on sélectionne un moins bon voisin avec une certaine probabilité.

On commence avec des probabilités élevées et les diminuer plus on avance dans l'exploration.

On définit un schéma de probabilités (schedule of temperatures) en ordre décroissant. Exemple pour 100 itérations : $[2^0, 2^{-1}, 2^{-2}, \ldots, 2^{-99}]$

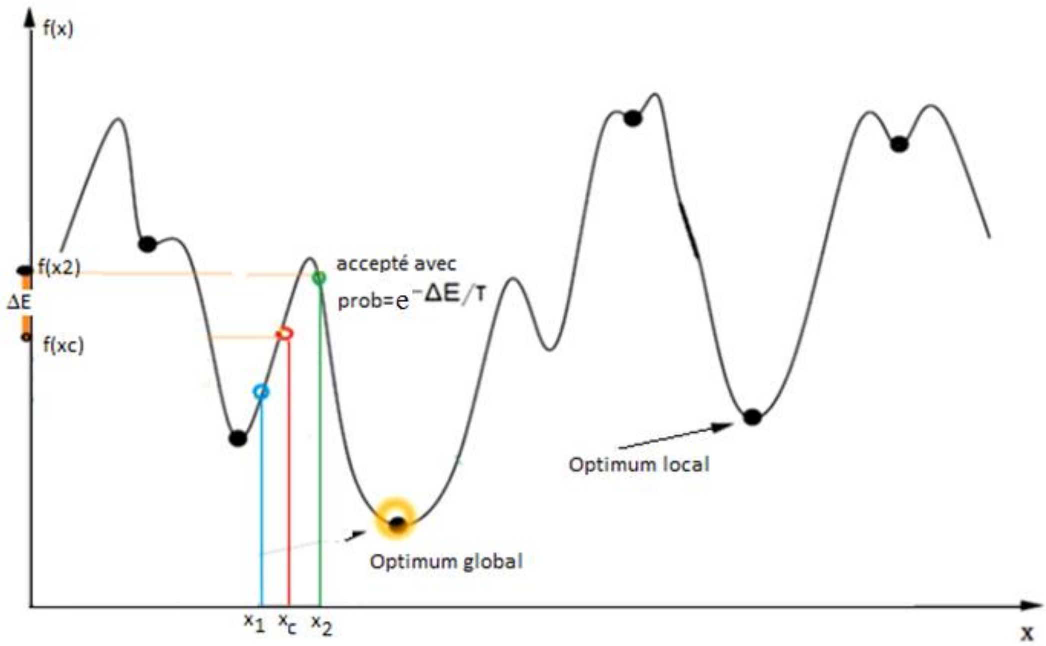


Figure 3.10. Graphe Recruit simulé

Algorithm Simulated Annealing (noeud_initial, schema)
Déclarer deux nœuds : n, n'
Déclarer : t, T, dE

N=nœud_initial
Pour t=1..taille (schema)
// le schéma de température varie d'un problème à
un autre. T=schema[t] **n'**=successeur de **n**
choisi au hasard **dE=F(n')-F(n)** si on maximise,
sinon **dE=F(n)-F(n')** Si **dE>0**, alors assigner
n=n' // amélioration par rapport à n
Sinon assigner **n=n'** seulement avec probabilité $e^{-|dE|/T} \geq r$ ($r =$
$random$ (0,1)
Si $|n - n'| < \varepsilon$ et T petit alors stop
Retourner n

3.8 Méthode Tabu Search

C'est une amélioration pour minimiser le risque de tomber dans des optimums locaux.

Le récuit simulé peut osciller indéfiniment en revenant à un nœud visité auparavant.

On peut envisager d'enregistrer les k derniers nœuds visités. On revient à A*, mais le nombre des états est important.

Algorithme Taboue, enregistre uniquement les k derniers nœuds visités (ensemble taboue). K est choisi aléatoirement. Il y aura réduction des oscillations et non élimination.

D'autres améliorations : Local Beam Search (exploration locale par faisceau)

On considère n nœuds solutions,

On commence par k nœuds choisis aléatoirement,

A chaque itération, génération de tous les successeurs des k nœuds, On choisit les k meilleurs et on recommence.

3.9 Algorithme génétique

Vocabulaire :

Individu : ensemble de chromosomes
Chromosome : chaines d'éléments d'un alphabet (AND)
Gène : élément de base du chromosome
Locus : la position du gène sur le chromosome
Génotype : l'ensemble des gènes de l'individu
Génome : génétique d'une espèce

L'algorithme génétique (AG) est un algorithme de recherche basé sur les mécanismes de la sélection naturelle et de la génétique.

Pour un problème donné, un ensemble de solutions possibles (population) est généré aléatoirement.

Les caractéristiques sont utilisées dans des séquences de gènes qui seront combinées avec d'autres gènes pour former des chromosomes [7].

La génération d'une nouvelle population se fait par :
-croisement (recomposer les gènes des individus dans la population)
-mutation (améliorer la qualité de la nouvelle génération)
Cette étape est exécutée en tenant compte de :

- La probabilité d'application du croisement
- Du nombre de générations à obtenir,
- De la taille de la population
- Du critère d'arrêt (si pas d'évolution)

Selon Lerman et Ngouenet [8], un algorithme génétique est défini par :
– Individu/chromosome/séquence : une solution potentielle du problème ;
– Population : un ensemble de chromosomes ou de points de l'espace de recherche ;
– Environnement : l'espace de recherche ;
– Fonction de fitness : la fonction à optimiser.

Les AGs sont alors basés sur les phases suivantes :

1. Initialisation. Une population initiale de N chromosomes est tirée aléatoirement.
2. Évaluation. Chaque chromosome est décodé, puis évalué.
3. Sélection. Création d'une nouvelle population de N chromosomes par l'utilisation d'une méthode de sélection appropriée.
4. Reproduction. Possibilité de croisement et mutation au sein de la nouvelle population.
5. Retour à la phase d'évaluation jusqu'à l'arrêt de l'algorithme

Pour améliorer les mauvaises solutions, les meilleures solutions sont choisies au hasard avec un mécanisme de sélection.

Cet opérateur est plus susceptible de choisir les meilleures solutions puisque la probabilité est proportionnelle à la fitness (valeur objective).

Ce qui augmente l'évitement des optima locaux est la probabilité de choisir également de mauvaises solutions. Cela signifie que si de bonnes solutions sont piégées dans une solution locale, elles peuvent être retirées avec d'autres solutions.

Les techniques de sélection :

La sélection se fait proportionnellement à la fonction de fitness. L'individu ayant une grande valeur de fitness a plus de chances de propager ses propriétés aux générations suivantes.

- Roulette wheel selection
- Tournament selection
- Rank selection
- Random selection

Exemple de sélection [9] :

Il s'agit de trouver les paramètres a, b, c, d (entre 0 et 30) qui vérifient l'équation :

F(a, b, c, d)=abs(a+2b+3c+4d-30)

1- génération d'une population aléatoire au nombre de 6.

Et évaluation de la fonction objective F.

Chromosome [1] = (12, 05, 23, 08), F(1)=93
Chromosome [2] = (2, 21, 18, 03), F(2)=80
Chromosome [3] = (10, 04, 13, 14), F(3)=83
Chromosome [4] = (20, 01, 10, 06), F(4)=46
Chromosome [5] = (01, 04, 13, 19), F(5)=94
Chromosome [6] = (20, 05, 17, 01), F(6)=55

2- pour la sélection, on prendra les chromosomes ayant une grande probabilité.

On définit la fonction fitness comme suit :

Fitness[i]=1/(1+F(i)) pour éviter de diviser par zéro.

On calculera ensuite la probabilité de chaque chromosome :

P[i]=Fitness[i]/Somme_Fitness_all_chromosomes

Somme_Fitness_all_chromosomes =
0.0106+0.0123+0.0119+0.0213+0.0105+0.0179=0.0845

P=[0.1245, 0.1456, 0.1408, 0.2521, 0.1243, 0.2118]

En adoptant la sélection par Roulette wheel, 6 nombres sont générés aléatoirement entre zéro et 1.

Les chromosomes tirés sont :

Chromosomes 2, 3, 1, 6, 3, 4.

On génère pour chacun de ces chromosomes un nombre aléatoire. S'il est inférieur à une probabilité de croisement fixée (0.25), alors il sera utilisé pour le croisement.

Après le tirage de 6 nombres aléatoires, les chromosomes sélectionnés pour le croisement sont :

Le premier, le quatrième et cinquième : 2, 6,3 sélectionnés pour le croisement sont :

Chromosome [2] = (2, 21, 18, 03), F(2)=80
Chromosome [6] = (20, 05, 17, 01), F(6)=55

Chromosome [3] = (10, 04, 13, 14), F(3)=83

Le croisement se fait en des « cut-points » obtenus après génération de nombres aléatoires (de 1 à 3) entre 2,6 et 2,3 et 6,3 donne 6 nouveaux chromosomes.

L'algorithme génétique classique est donné ci-après [5].

Algorithm : ***Classical Genetic Algorithm (GA)***

Input:
Population Size, n
Maximum number of iterations, MAX
Output:
Global best solution, Y_{bt}
begin
Generate initial population of n *chromosomes* Y_i $(i = 1, 2, \ldots, n)$
Set iteration counter $t = 0$
Compute the fitness value of each chromosomes
while $(t < MAX)$
Select a pair of chromosomes from initial population based on fitness
Apply crossover operation on selected pair with crossover probability
Apply mutation on the offspring with mutation probability
Replace old population with newly generated population
Increment the current iteration t *by 1.*
end while
return the best solution, Y_{bt}
end

Exemple de croisement (problème de reines (8x8)

On prend comme fonction d'adaptation : le nombre de paires de reines qui ne s'attaquent pas (min=0, max= 8x7/2=28)

Prenons les chromosomes suivants de la population initiale :

(1)24748552, (2) 32752411, (3) 24415124, (4) 32543213

Calculons la probabilité de sélection du premier chromosome, proportionnelle à F(n) :

Prob(1)=24/78=31%,
Prob(2)=23/78=29%,
Prob(3)=20/78=26%,
Prob(4)=11/78=14%

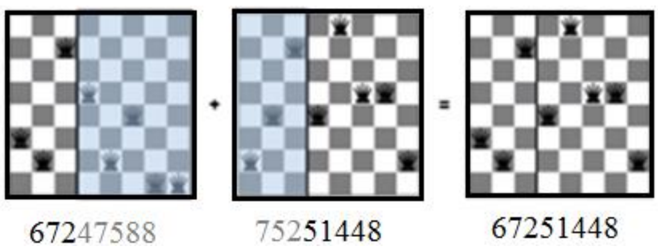


Figure 3.11. Exemple de croisement (problème de reines)

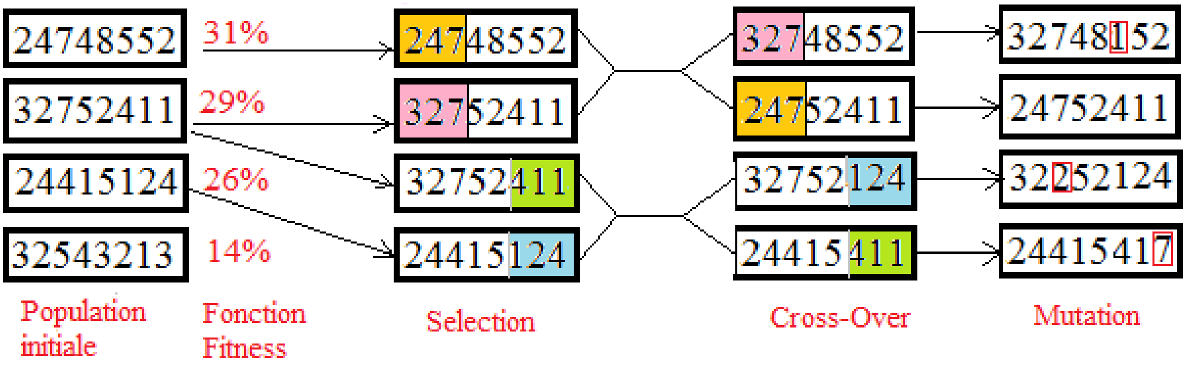


Figure 3.12. Exemple de choix de sélection

Ne pas sélectionner les chromosomes faisant partie des 25% pires

Problème de Satisfaction de Contraintes

4.1 Introduction

Soient:

- V={V1, V2, …, Vn}: Un ensemble de variables définies sur une ensemble de domaines D={D1, D2, .., Dn}
- C={C1, C2, …, Cm} un ensemble de contraintes sur les variables V1, V2, …, Vn

Le problème de satisfaction de contraintes (CSP) a pour objectif d'assigner une valeur à chacune des variables de V en respectant les contraintes C.

Exemple de problèmes de satisfaction de contraintes (CSP):

- Grille de Sudoku
- V={S[i,j], i=1..9, j=1..9}
- Di,j={1,2, .., 9} valeur possible pour S[i,j]
- C={Cligne, Ccolonne, C bloc}
- Cligne: All_Diff(S[i,j], j=1..9, i fixé}
- Ccolonne: All_Diff(S[i,j], i=1..9, j fixé}
- Cbloc: All_Diff(S[1,1], S[1,2] , S[1,3] , S[2,1], S[2,2] , S[2,3] , S[3,1] , S[3,2] , S[3,3]}
- ….
- All_Diff(S[7,7], S[7,8] , S[7,9] , S[8,7], S[8,8] , S[8,9] , S[9,7] , S[9,8] , S[9,9]}

Exemple de problèmes de satisfaction de contraintes (CSP) :

- Coloriage de graphe : Colorier les régions V= {WA, NT, SA, Q, NSW, V, T} de l'Australie moyennant trois couleurs D= {R, G, B} avec la contrainte : C= {Régions voisines ne peuvent avoir la même couleur}.
- Le graphe de la figure ci-dessous illustre la connexion des régions par des arêtes.

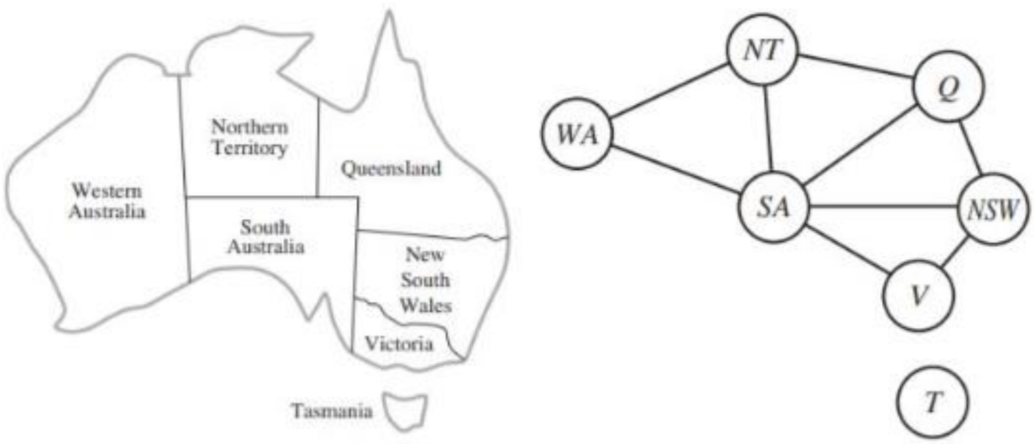


Figure 4.1. Exemple du problème de coloriage de graphe

4.2 Algorithme BackTracking

Le problème CSP peut être résolu en assignant des valeurs aux variables (une à une) en exploitant les différentes combinaisons.

Lorsqu'une contrainte n'est pas respectée suite à une assignation, l'algorithme BackTracking revient à la variable et lui assigne une nouvelle valeur.

```
Algorithm   BACKTRACK(assignment, csp)
if assignment is complete then return true
var ← SELECT-UNASSIGNED-VARIABLE(csp)
for all value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if adding {var = value} satisfies every constraint then
        add {var = value} to assignment
        result ← BACKTRACK(assignment, csp)
        if result is true then return result
    end if
    remove {var = value} from assignment
end for
return false // Aucune valeur ne satisfait les contraintes
```

ORDER-DOMAIN-VALUES :Ordre de valeurs à essayer

SELECT-UNASSIGNED-VARIABLE : permet de sélectionner la prochaine variable

Le problème CSP peut être résolu en assignant des valeurs aux variables (une à une) en exploitant les différentes combinaisons.

Lorsqu'une contrainte n'est pas respectée suite à une assignation, l'algorithme BackTracking revient à la variable et lui assigne une nouvelle valeur.

Exemple :

Variable A, DA ={1,2,3}

Variable B, DB ={1,2,3}

Variable C, DC ={1,2,3}

Contraintes : A>B, B≠C, A≠C

Variable A, DA ={1,2,3}

Variable B, DB ={1,2,3}

Variable C, DC ={1,2,3}

Contraintes : C1: A>B, C2: B≠C, C3: A≠C

Assignation	Contraintes satisfaites			Action
	C1	C2	C3	
A=1	yes	Yes	Yes	
A=1, B=1	No	Yes	Yes	Back to B
A=1, B=2	No	Yes	Yes	Back to B
A=1, B=3	No	Yes	Yes	Back to A

A=2	Yes	Yes	Yes	
A=2, B=1	Yes	Yes	Yes	
A=2, B=1, C=1	Yes	No	Yes	Back to C
A=2, B=1, C=2	Yes	Yes	No	Back to C
A=2, B=1, C=3	Yes	No	Yes	End

Amélioration

Des heuristiques sont utilisées pour améliorer l'algorithme (temps de réponse) et ceci en tenant compte des éléments suivants :

- Quelle valeur assigner ?
- Dans quel ordre les variables seront assignées
- Peut-on détecter un échec à l'avance ?

Heuristiques pour sélectionner une variable

(1) -MRV (Minimum Remaining Values)

Elle consiste à :

- Sélectionner d'abord les variables ayant le moins de valeurs possibles (MRV)
- En cas où plusieurs variables vérifient l'heuristique MRV, sélectionner la variable ayant le plus de contraintes ((2) **degree heuristic).**

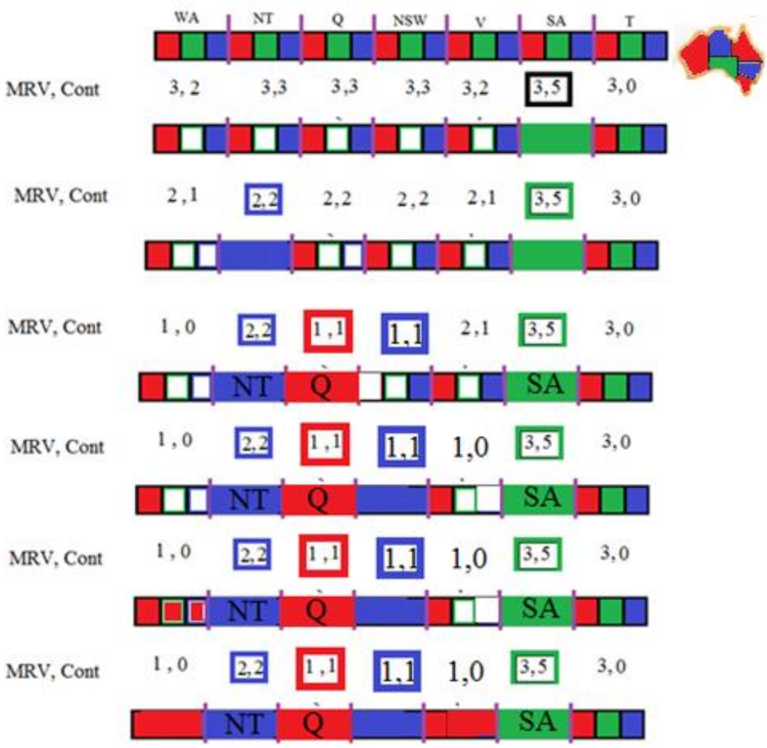


Figure 4.2. Application de MRV

(2) degree heuristic :

Choisir la variable impliquée dans le plus grand nombre de contraintes sur les autres variables non affectées.

(3) Least-constraining-value heuristic (LCV):

Sélectionnez la valeur qui exclut le moins de valeurs (ou laisse le plus grand nombre de valeurs) pour les variables non affectées voisines :

Donc la variable a moins de contraintes.

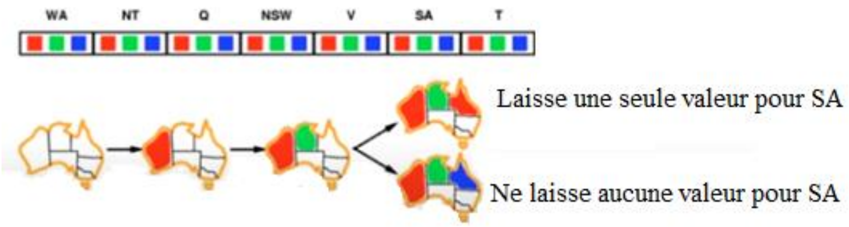


Figure 4.3. Application de LCV

(4) Heuristique d'inférence

Détection des assignations conflictuelle

Algorithme : Forward Checking

- Vérifier les valeurs compatibles des valeurs non-assignées
- Terminer la récursivité en cas de conflit : Une variable non assignée avec un ensemble de valeurs compatibles vide.

Détection des assignations conflictuelle

Algorithme : Forward Checking

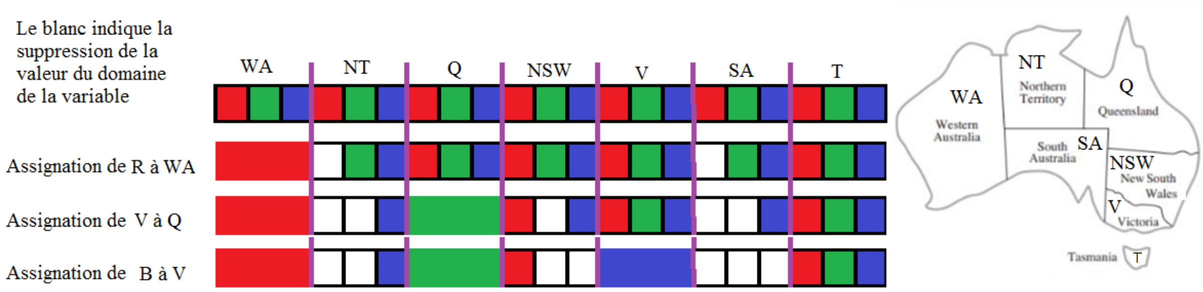


Figure 4.4. Application de l' Algorithme Forward Checking

Algorithm: Forward Checking (X, CSP)

Pour chaque Xk pris des voisins de (X, CSP)

Changé, CSP=**REVISER** (Xk, X, CSP)

// On peut changer le domaine de Xk en tenant compte des contraintes entre X et Xk)

Si Changé et Domaine(Xk, CSP) est vide

Retourner (Void, Faux)

Retourner (CSP, Vrai)

REVISER (Xi, Xj, CSP)

// Réduit le domaine de Xi en fonction de celui de Xj

Changé= Faux

Pour chaque **x** dans Domaine (Xi, CSP)

Si Aucun **y** dans Domaine(Xj, CSP) qui satisfait la contrainte entre Xi et Xj

Enlever **x** du Domaine (Xi, CSP)

Changé=Vrai

Retourner (Changé, CSP)

Algorithm BACKTRACK-INFERENCES(assignment, csp)

```
if assignment is complete then return true
var ← SELECT-UNASSIGNED-VARIABLE(csp)
for all value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if adding {var = value} satisfies every constraint then
        add {var = value} to assignment
        inf-result ← INFERENCES(assignment, csp)
        if inf-result is true then
            add the inference results to assignment
            result ← BACKTRACK(assignment, csp)
            if result is true then return result
        end if
    end if
    remove {var = value} and the inference results from assignment
end for
return false
```

4.3 Algorithme Arc Consistency 3 (AC-3

Consistance d'arc

Un CSP(X,D,C) est **consistant d'arc** si pour tout couple de variables (Xi, Xj), et pour toute valeur vi de D(Xi), il existe une valeur vj de D(Xj) telle que {(Xi,vi), (Xj,vj)} satisfait toutes les contraintes binaires de C.

Exemple :

C: $X_1+X_2>2$

$D(X_1)=D(X_2)=\{0,1,2\}$

CSP n'est pas consistant d'arc.

Si on enlève la valeur zéro du domaine de X1, car si X1=0, il n'existe pas de valeur de X2 qui satisfait la contrainte C.

D(X1)={1,2}, D(X2)={0,1,2}

Si X1=1, alors X2=2 permet de satisfaire la contrainte C.

De même pour X1=2, il existe au moins une valeur de D(X2) qui permet de satisfaire la contrainte C.

Vérifie la compatibilité entre les arcs (entre deux variables).

L'arc X->Y est compatible ssi :

- Pour chaque valeur x de X il existe au moins une valeur permise y de Y
- Si une variable perd une valeur, ses voisins (variables) doivent être revérifiés.
- Exemple, si on affecte bleu à NSW, il n'existe pas de valeur dans SA qui satisfait les contraintes.
- On supprime donc le bleu de NSW (on change le domaine)

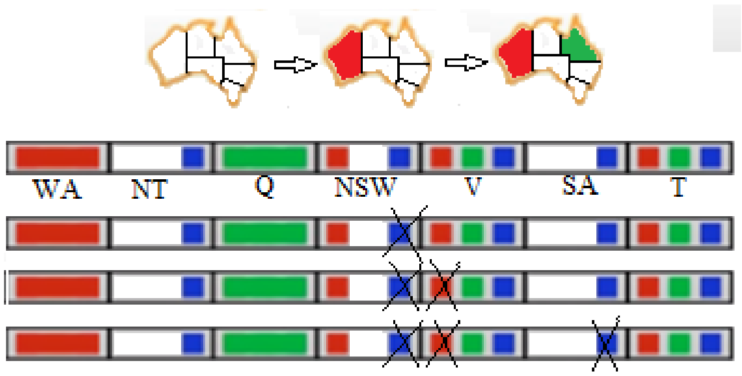


Figure 4.5. Application de l'algorithme de Consistance d'arc

Appliqué au début de l'algorithme BackTracking, ou juste après assignation d'une nouvelle valeur.

Algorithme

Ecrire chaque Contrainte binaire en deux arcs :

// exemple : A≠B s'écrira : A≠B et B≠A

Ajouter tous les arcs à **Agenda**

Répéter jusqu'à ce que **Agenda** est vide

- Prendre un arc (Xi, Xj) et vérifier que pour chaque valeur de Xi

S'il existe une valeur de Xj qui satisfait les contraintes

Supprimer toute valeur non consistante de Xi

Si Xi a été changé, alors ajouter tous les arcs de la forme (Xk, Xi) à l'Agenda

Exemple d'application

A={1, 2, 3}
B ={1, 2, 3}
C ={1, 2, 3}
A>B,
B=C

Variables		**Agenda**		**Arcs**
A={1, 2, 3}			A>B	
B ={1, 2, 3}			B<A	
C ={1, 2, 3}				B=C
			C=B	

Variables		**Agenda**		**Arcs**
A={1, 2, 3}	A>B		A>B	
B ={1, 2, 3}	B<A		B<A	
C ={1, 2, 3}		B=C		B=C
	C=B		C=B	
	A>B			
	B=C			

Chapitre 5.

Exercices de travaux dirigés et pratiques

Exercices de travaux dirigés et pratiques

Exercice 1.

Ecrire un programme écrit en langage Processing qui permet de créer une image contenant une grille où chaque élément représente soit un espace libre, ou occupé par un obstacle ou occupé par un objet « objectif » ou par un robot sujet à des déplacements.

Les couleurs des éléments de la grille sont fixées comme suit (voir figure ci-dessous) :

- Bleu pour obstacle,
- Noir pour espace libre
- Rouge pour le robot
- Vert pour l'objectif.

Ecrire le programme en Processing qui permet de déplacer le robot en suivant l'algorithme « largeur d'abord », le coût du passage d'une cellule à l'autre est égal à 1. Visualisez les déplacements et comptez le nombre de cellules explorés en variant les positions de l'objectif et du robot.

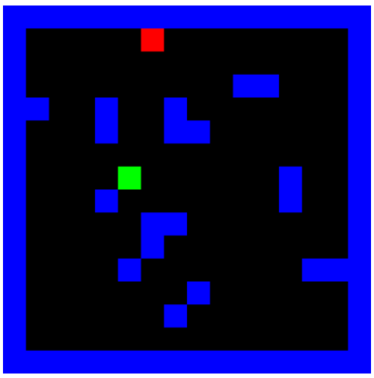


Figure 5.1

Exercice 2.

Refaire l'exercice pour les algorithmes : profondeur d'abord, et coût uniforme. Comparez les trois algorithmes pour différentes configurations (objectif, robot). Evaluez leurs complexités.

Exercice 3

Une fonction heuristique h appliquée à un espace d'états E satisfait la restriction de monotonie (consistance) si et seulement si :

Pour tout $(n_i, n_j) \in E$ où $n_j$est fils de n_i, $h(n_i) - h(n_j) \leq c(n_i, n_j)$ où $c(n_i, n_j)$ est le coût associé à l'arc reliant n_i à n_j.

Montrer qu'un algorithme de type A dont la fonction associée h satisfait la restriction de monotonie est de type A* (toute heuristique monotone est admissible).

Solution

Soit N le nombre de nœuds de s à no.

Quel que soit le nœud n_i ayant comme successeur n_j, comme h est consistante, alors $h(n_i) \leq C(n_i, n_j) + h(n_j)$

On peut remplacer dans cette relation $h(n_j)$ par $h(n_j) \leq C(n_j, n_k) + h(n_k)$ où le nœud n_k est le successeur de n_j,

Nous obtenons ainsi :

$$h(n_i) \leq C(n_i, n_j) + h(n_j) \leq C(n_i, n_j) + C(n_j, n_k) + h(n_k)$$

En répétant le même processus, en remplaçant $h(n_k)$ puis $h(n_l)$ où n_l est le successeur de n_k jusqu'à ce qu'on arrive au dernier nœud qui mène au but $s\ tel\ que\ h(s) = 0$.

$$h(n_i) \leq C(n_i, n_j) + C(n_j, n_k) \ldots + C(n_s, s)$$

Donc $h(n_i) \leq \hat{h}(n_i)$ pour tout nœud n_i.

Exercice 4

Un objet se déplace de l'état initial S vers un état final G comme indiqué par la figure 2 où l'on représente l'espace d'états. A chaque état n on associe une heuristique h(n) du chemin qui reste à parcourir de l'état courant vers G. Le coût du passage d'un état vers un autre est indiqué sur l'arc.

Donnez les nœuds visités et le chemin obtenu pour la recherche A avec f(n)=g(n)+h(n), où g(n) est le coût du chemin S vers l'état n. Le chemin trouvé est-il optimal ? justifiez.

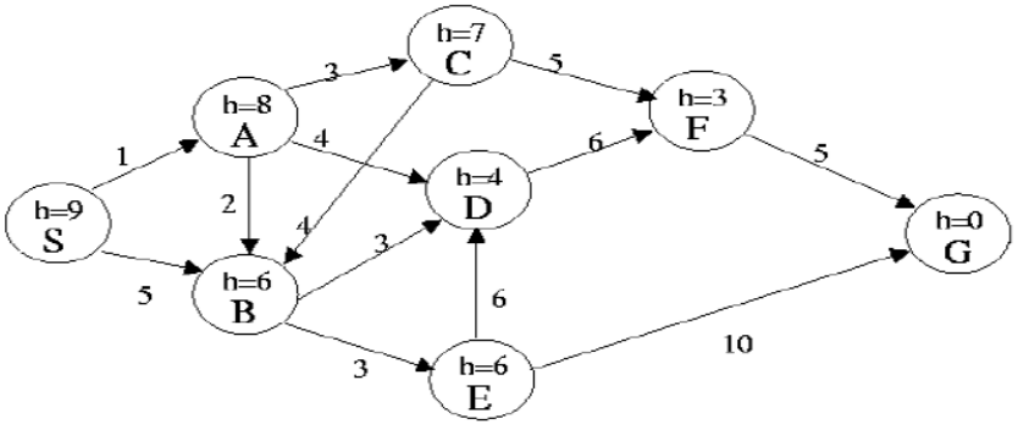


Figure 5.2. Graphe de recherche

Exercice 5

Il s'agit de trouver le chemin le plus court de l'entrée à la sortie d'un labyrinthe en utilisant un algorithme de type A* (voir figure 3). Nous supposons qu'aller d'une cellule à une autre se fait avec un coût égal à 1.

1- Si $h(n)$ est le coût estimé du chemin (de la cellule numéro n vers la cellule de sortie m) définie comme étant la distance de Manhattan (somme des distances horizontale et verticale entre n et m), l'algorithme est-il de type A* ? Cette distance est utile pour le déplacement vers les 4 voisins (horizontaux et verticaux).
2- Si $h(n)$ est le coût estimé du chemin (de la cellule numéro n vers la cellule de sortie m) définie comme étant la distance Diagonale (maximum des distances horizontale ou verticale entre n et m), l'algorithme est-il de type A* ? Cette distance est utile pour le déplacement vers les 8 voisins.
3- Refaire la question (1) en utilisant la distance euclidienne entre n et m. Quelle distance choisir ?

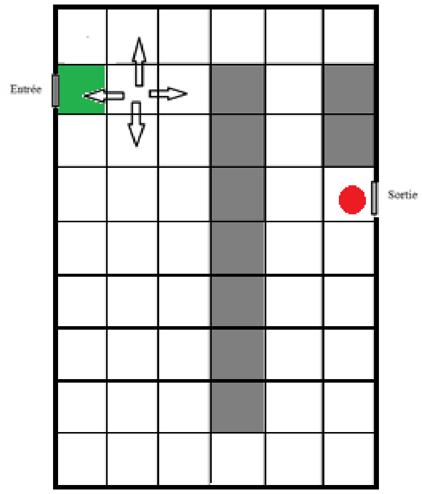


Figure 5.3. Exemple d'un labyrinthe (case crise représentent les obstacles).

Algorithme de jeu : Min-Max

Exercice 6.

On considère le jeu Mini-Othello dont les règles sont décrites ci-après :

- L'échiquier est composé d'un damier de 6x6 éléments et initialisé par les jetons (rouge, vert) comme indiqué par la figure ci-dessous.
- Le tour est donné au joueur, il ne peut placer son jeton que s'il arrive à mettre entre deux de ces jetons les jetons verts alignés. De même pour l'ordinateur. Sinon son tour passe.
- Si les jetons d'une couleur c1 sont mis entre deux jetons de couleur c2, ils seront mis à la couleur c2.
- Le jeu s'arrête si toutes les cases sont remplies ou si un joueur ne peut plus jouer.

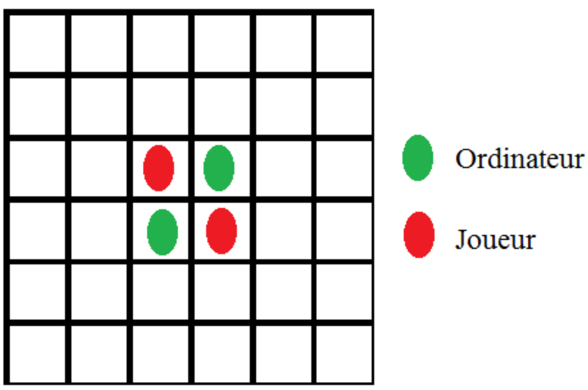


Figure 5.4

1- En premier lieu, on suppose que l'ordinateur est un second joueur. Implémentez le jeu.
2- On veut que l'ordinateur joue en utilisant l'algorithme Min-Max en prenant comme coût de l'action le nombre de jetons obtenus pour son profit. Implémentez de nouveau le jeu.

Exercice 7.

Description du jeu Othello

Les joueurs disposent de 64 pions bicolores, noirs d'un côté et blancs de l'autre. En début de partie, quatre pions sont déjà placés au centre de l'othellier : deux noirs, en e4 et d5, et deux blancs, en d4 et e5 (voir figure 1).

Chaque joueur, noir et blanc, pose l'un après l'autre un pion de sa couleur sur l'othellier selon les règles définies ci-dessous. Le jeu s'arrête quand les deux joueurs ne peuvent plus poser de pion. On compte alors le nombre de pions. Le joueur ayant le plus grand nombre de pions de sa couleur sur l'othellier a gagné.

Noir commence toujours la partie. Puis les joueurs jouent à tour de rôle, chacun étant tenu de capturer des pions adverses lors de son mouvement. Si un joueur ne peut pas capturer de pion(s) adverse(s), il est forcé de passer son tour. Si aucun des deux joueurs ne peut jouer, ou si l'othellier ne comporte plus de case vide, la partie s'arrête. Le gagnant en fin de partie est celui qui possède le plus de pions.

La capture de pions survient lorsqu'un joueur place un de ses pions à l'extrémité d'un alignement de pions adverses contigus et dont l'autre extrémité est déjà occupée par un de ses propres pions. Les alignements considérés peuvent être une colonne, une ligne, ou une diagonale. Si le pion nouvellement placé vient fermer plusieurs alignements, il capture tous les pions adverses des lignes ainsi fermées. La capture se traduit par le retournement des pions capturés. Ces retournements n'entraînent pas d'effet de capture en cascade : seul le pion nouvellement posé est pris en compte.

Par exemple, la figure de gauche ci-dessous montre la position de départ. La figure centrale montre les 4 cases où Noir peut jouer, grâce à la capture d'un pion Blanc. Enfin, la figure de droite montre la position résultante si Noir joue en d3. Le pion Blanc d4 a été capturé (retourné), devenant ainsi un pion Noir.

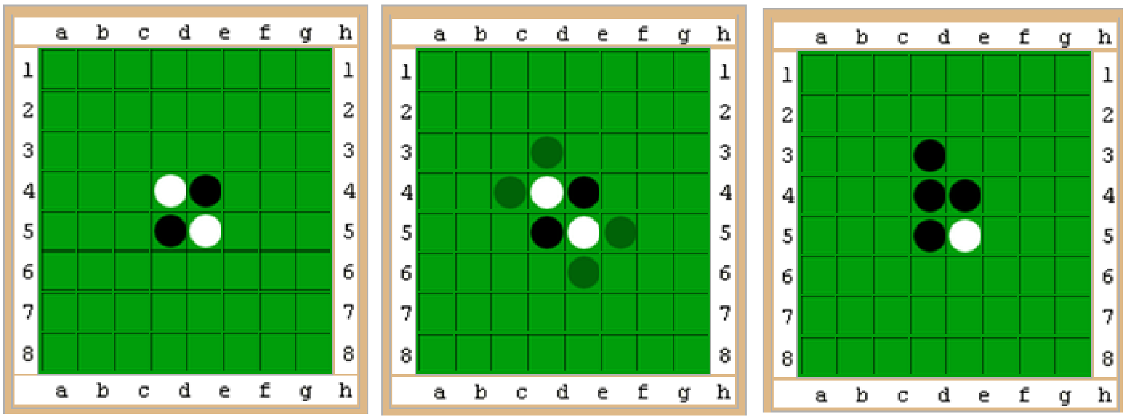


Figure 5.5

Travail demandé :

Partie 1 :

- Pour un état donné de l'othellier, trouver pour un joueur donné l'ensemble des positions possibles pour déposer son pion.
- Pour un joueur donné, et pour une position choisie dans laquelle il mettra son pion, mettre à jour l'othellier et retourner le nombre de ses pions et ceux de son adversaire

Partie 2 :

Il s'agit pour un état donné de l'othellier, et pour un joueur donné, développez un arbre de des états (espace d'états) de profondeur np (on prendra np comme paramètre). Pour chacun des états associez un coût (à définir).

Implémentez cette fonction et faites des tests pour np=1, np=2.

Partie 3 :

Réalisez une interface graphique moyennant Processing qui permet d'afficher l'othellier, les états des deux joueurs, le prochain joueur, les listes des actions pour chacun des joueurs.

Partie 4 :

Définir un coût à chacun des états (à justifier) et implémenter un algorithme MinMax de sorte que le joueur joue contre votre système (machine). Il s'agit donc de minimiser le coût du joueur (s'il joue en premier lieu) et de minimiser le score du système.

Exercice 8.

Description du jeu Abalone

Le jeu Abalone est décrit sur le document joint. Les jetons de deux couleurs différentes sont disposés comme indiqué par la figure 1.

Pour implémenter ce jeu, nous proposons d'utiliser la matrice illustrée par la figure 2. Les cases colorées indiquent les positions autorisées pour les différents jetons.

Travail demandé :

Partie 1 :

- Afficher la gille de jeu,
- Positionner les jetons sur leurs emplacements (Noir et Blanc)
- Déplacer un jeton par clic de la souris
- Déplacer un ensemble de jetons selon les six directions

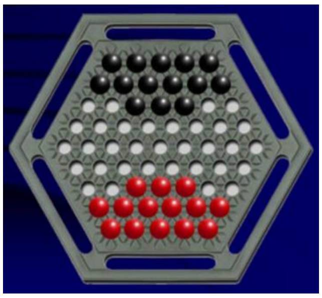


Figure 5.6. Surface du jeu Abalone.

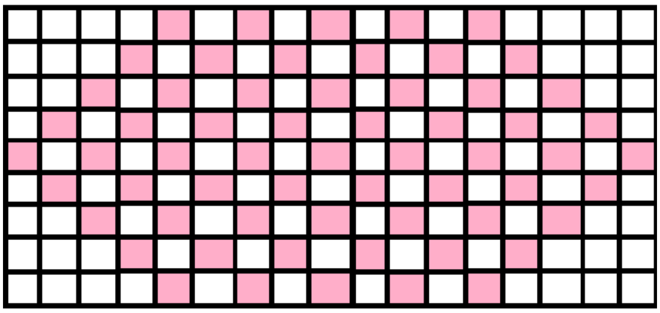


Figure 5.7 Matrice associée au jeu Abalone

Exercice 9.

Un objet se déplace de l'état initial S vers un état final G comme indiqué par la figure ci-dessous où l'on représente l'espace d'états. A chaque état n on associe une heuristique h(n) du chemin qui reste à parcourir de l'état courant vers G. Le coût du passage d'un état vers un autre est indiqué sur l'arc.

Donnez les nœuds visités et le chemin obtenu pour la recherche A* avec f(n)=g(n)+h(n), où g(n) est le coût du chemin S vers l'état n. Le chemin trouvé est-il optimal ? justifiez.

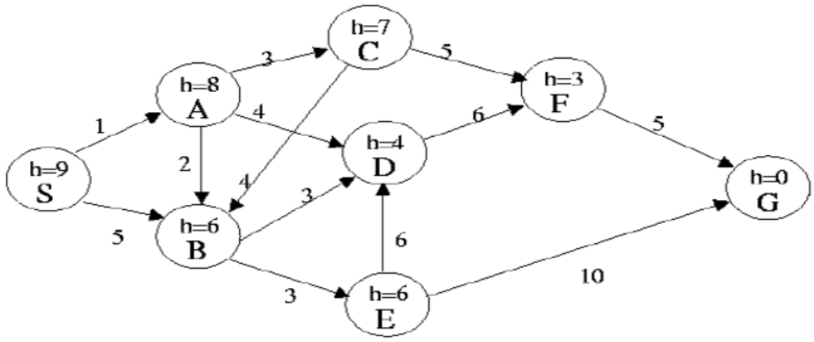


Figure 5.8 Graphe de recherche

Exercice 10.

Un objet se déplace de l'état initial S vers un état final G comme indiqué par la figure ci-dessous où l'on représente l'espace d'états. A chaque état n on associe une heuristique w(n) du chemin qui reste à parcourir de l'état courant vers S. Le coût du passage d'un état vers un autre est égal à 1.

En supposant que les chemins avec boucle sont éliminés, donnez les nœuds visités et le chemin obtenu pour chacun des algorithmes :

- Profondeur d'abord
- Largeur d'abord
- Recherche A* avec f(n)=g(n)+w(n), où g(n) est le coût du chemin S vers l'état n. Le chemin trouvé est-il optimal ? justifiez.

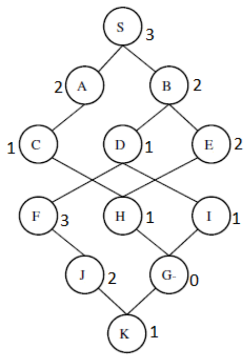


Figure 5.9 Graphe de recherche

Exercice 11.

Appliquez la procédure alpha beta pour le sous arbre suivant en utilisant Negamax.

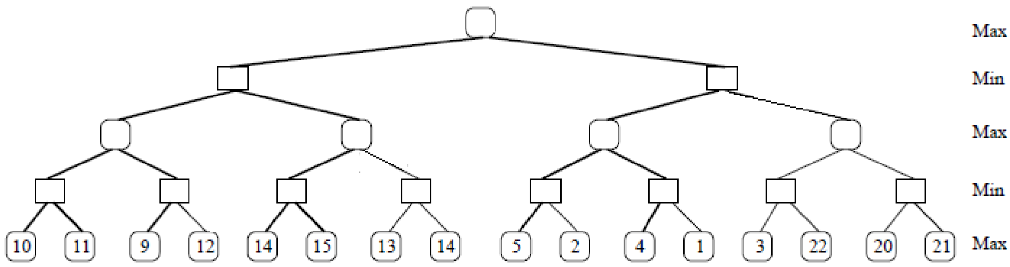


Figure 5.10 Arbre de jeu

Exercice 12.

Appliquez l'algorithme Alpha-Beta pour l'arbre de jeu donné par la figure 1 en appliquant Negamax. Donnez pour chaque nœud les valeurs de [illegible].

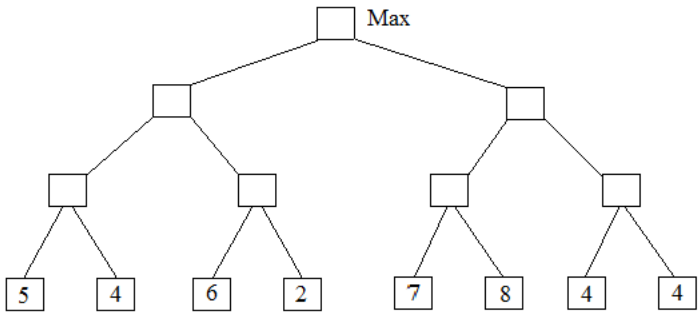


Figure 5.11 Arbre de jeu

Exercice 13.

Appliquez la procédure alpha beta pour le sous arbre suivant.

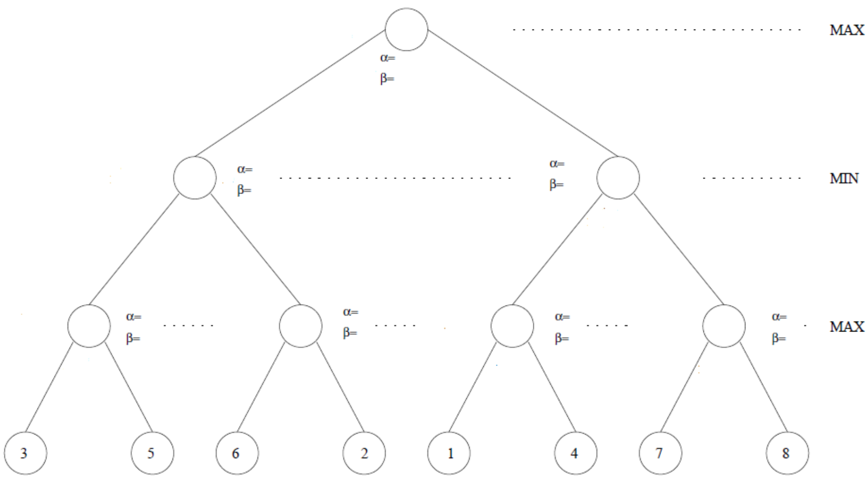


Figure 5.12 Arbre de jeu

Exercice 14. A* pour jeu de taquin

1- Ecrire un programme en langage Processing (ou Python) qui crée et affiche une image contenant une grille 3x3 où les éléments contiennent les valeurs de 1 à 8. La case centrale est sans valeur comme indique par la figure 1.

2- Ajoute au code produit la possibilité de déplacer un chiffre d'une case vers la case vide. La case vide sera à la position du chiffre déplacé.

3- Appliquer plusieurs déplacements et obtenez un état S (état de départ, voir l'exemple de la figure 2) à partir duquel nous allons chercher, en appliquant l'algorithme A* la meilleure combinaison de déplacement des chiffres pour aller à l'état initial indiqué par la figure 1.

4- Implémentez l'algorithme A star et visualisez les étapes de déroulement de l'algorithme pour trouver le chemin optimal

1	2	3
8		4
7	6	5

Figure 5.13. Taquin (Nœud no)

3	4	7
5		8
1	2	6

Figure 5.14 Taquin (Nœud S)

Exercice 15.

Appliquer l'élagage alpha-beta à l'arbre de jeu suivant en utilisant NEGAMAX. Les gains g(ni) donnés pour les nœuds terminaux sont associés au joueur Min.

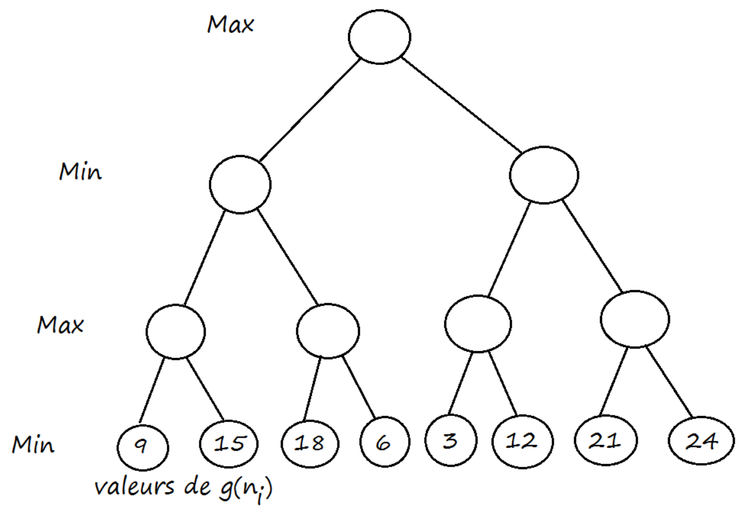


Figure 5.15

Réponse :

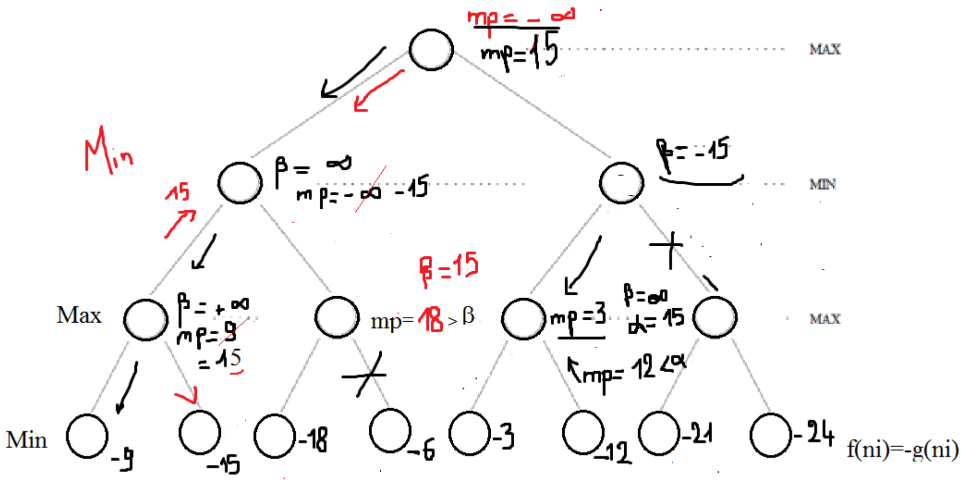


Figure 5.16

Références

[1] Principles of Artificial Intelligence par Nils J. Nilson, 1982, Springer-Verlag Berlin Heidelberg.

[2] Essentials of Artificial Intelligence par Nils J. Nilson, 1993, Copyright: © Morgan Kaufmann

[3] Artificial Intelligence: A new synthesis, Nils Nilson, 1997, Elsevier.

[4]Artificial Intelligence: A Modern Approach par Stuart Russell and Peter Norvig, Ed. Pearson, 2016

[5] Sourabh Katoh, Sumit Singh Chauhan, Vijay Kumar. A review on genetic algorithm: past, present, and future. Multimedia Tools and Applications volume 80, pages8091–8126 (2021).

[6] **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.).

[7] Jean H. Holland, Adaptation in Natural and Artificial Systems.

University of Michigan Press : Ann Arbor, 1975.

[8] Lerman, I. & Ngouenet, F. (1995), Algorithmes génétiques séquentiels et parallèles pour une représentation affine des proximités, Rapport de Recherche de l'INRIA Rennes - Projet REPCO 2570, INRIA.

[9] D. Hermawanto, Genetic algorithm for solving mathematica equality problem

https://arxiv.org/ftp/arxiv/papers/1308/1308.4675.pdf

[10] Rapport de synthèse - France Intelligence Artificielle. Ministère de l'Enseignement supérieur et de la Recherche, 2017.