Résolution de problèmes à l'aide d'algorithmes de recherche

1

Plan

- Agent de résolution de problèmes
- Stratégies de recherche
  - Recherche non-informée
    - Largeur d'abord, profondeur d'abord, etc.
  - Recherche informée
    - Meilleur d'abord, A*, algorithmes génétiques, etc.

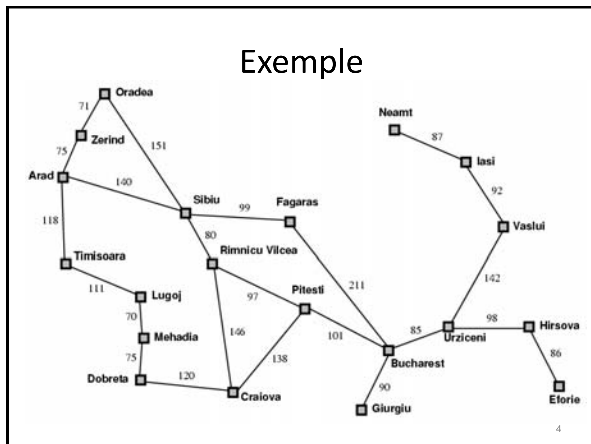
2

Agent de résolution de problèmes

1. Formulation d'un but : un ensemble d'états à atteindre.
2. Formulation du problème : les états et les actions à considérer.
3. Recherche de solution : examiner les différentes séquences d'actions menant à un état but et choisir la meilleure.
4. Exécution : accomplir la séquence d'actions sélectionnée.

3

Exemple



Exemple

- On est à Arad et on veut aller à Bucharest
  - But : être à Bucharest
  - Problème :
    - États : villes
    - Actions : aller d'une ville à une autre.
  - Solution : Une séquence de villes.
    - Ex: Arad, Sibiu, Fagaras, Bucharest
- Environnement très simple
  - statique, observable, discret et déterministe

Formulation du problème

- État initial : l'état *x* dans lequel se trouve l'agent
  - *À*(*Arad*)
- Actions : Une fonction *S*(*x*) qui permet de trouver l'ensemble des états successeurs, sous forme de pair (action, successeur)
  - *S*(*À*(*Arad*)) = (*AllerÀ*(*Sibiu*), *À*(*Sibiu*)), etc.
- Test de but : un test afin de déterminer si le but est atteint
  - But : {*À*(*Bucharest*)}
- Coût du chemin : une fonction qui assigne un coût à un chemin. Elle reflète la mesure de performance de l'agent. La valeur est non-négative.

Abstraction

- Enlever des détails de la représentation.
  - On ne parle pas de la radio, du paysage ou de la météo.
  - On conserve le minimum nécessaire pour résoudre le problème d'aller à Bucharest.
- Une abstraction est valide si l'on peut développer toutes les solutions abstraites vers une solution dans le monde réel.
- Sans abstraction, un agent ne peut rien faire, le monde est juste trop complexe. L'humain est pareil aussi.

7

Exemple de problème : 8-puzzle

- États : ?
  - Positions des huit tuiles dans les cases.
- Actions : ?
  - Déplacement du trou: droite, gauche, haut, bas.
- Test de but : ?
  - Un état qui correspond à l'état final.
- Coût du chemin : ?
  - Chaque action coûte 1.

5	7	3
8		1
4	2	6

État initial

1	2	3
4	5	6
7	8	

État but

8

Recherche de solutions dans un arbre

- Simuler l'exploration de l'espace d'états en générant des successeurs pour les états déjà explorés.
- Nœud de recherche
  - État : l'état dans l'espace d'état.
  - Nœud parent : Le nœud dans l'arbre de recherche qui a généré ce nœud.
  - Action : L'action qui a été appliquée à l'état du nœud parent pour générer l'état de ce nœud.
  - Coût du chemin : Le coût du chemin à partir de l'état initial jusqu'à ce nœud : $g(n)$
  - Profondeur : Le nombre d'étapes dans le chemin à partir de l'état initial.

9

Stratégies de recherche

- Détermine l'ordre de développement des nœuds.
- Recherches non-informées : Aucune information additionnelle. Elles ne peuvent pas dire si un nœud est meilleur qu'un autre. Elles peuvent seulement dire si l'état est un but ou non.
- Recherches informées (heuristiques) : Elles peuvent estimer si un nœud est plus prometteur qu'un autre.

10

Évaluation des stratégies

- Complétude : Est-ce que l'algorithme garantit de trouver une solution s'il y en a une ?
- Optimalité : Est-ce que la stratégie trouve la solution optimale ?
- Complexité de temps : Combien de temps ça prend pour trouver une solution ?
- Complexité d'espace : Quelle quantité de mémoire a-t-on besoin ?

11

Complexité

- Elle est exprimée en utilisant les quantités suivantes :
  - *b*, le facteur de branchement: le nombre maximum de successeurs à un nœud.
  - *d*, la profondeur du **nœud but** le moins profond.
  - *m*, la longueur maximale d'un chemin dans l'espace d'états.
- Complexité en temps : le nombre de nœuds générés pendant la recherche.
- Complexité en espace : le nombre maximum de nœuds en mémoire.

12

Stratégies de recherche non-informées

- Largeur d'abord (Breath-first)
- Coût uniforme (Uniform-cost)
- Profondeur d'abord (Depth-first)
- Profondeur limitée (Depth-limited)
- Profondeur itératif (Iterative deepening)
- Recherche bidirectionnelle (Bidirectional search)

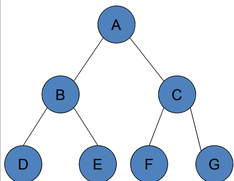
13

Largeur d'abord

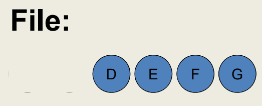
- Développement de tous les nœuds au niveau *i* avant de développer les nœuds au niveau *i+1*
- Implémenté à l'aide d'une file; les nouveaux successeurs vont à la fin.

14

Exemple largeur d'abord



File:



Ordre de visite: A – B – C – D – E – F - G

15

Propriétés de largeur d'abord

- Complétude : oui, si b est fini
- Complexité en temps : $b^0 + b^1 + b^2 + b^3 + \dots + b^d + b^{d+1} = O(b^{d+1})$
- Complexité en espace : $O(b^{d+1})$ (Garde tous les nœuds en mémoire)
- Optimal : non en général, oui si le coût des actions est le même pour toutes les actions

Profondeur	Nœuds	Temps	Mémoire
8	10^9	31 heures	1 teraoctet
12	10^{13}	35 ans	10 pétaoctets

16

Exemple de largeur d'abord

- Le temps sont à 10 000 nœuds par seconde
- 1000 octets de mémoire pour un nœud.
- Teraoctet = 1 000 milliards, soit 10^{12}
- Pétaoctets = 1 000 teraoctets, soit 10^{15}

Profondeur	Nœuds	Temps	Mémoire
8	10^9	31 heures	1 teraoctet
12	10^{13}	35 ans	10 pétaoctets

17

Coût uniforme

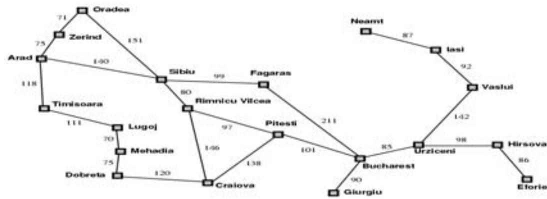
- Développe le nœud ayant le coût le plus bas.
- File triée selon le coût
- Équivalent à largeur d'abord si le coût des actions est toujours le même.

18

Exemple pour coût uniforme

Aller de Sibiu à Bucarest

- On développe le nœud à moindre coût, soit RM, ce qui ajoute Pitesti et au total ça donne 80 +97 = 177
- Le nœud de moindre coût est maintenant Fagaras; il est donc développé, ce qui ajoute Bucarest avec un coût de 99 + 210 = 310; un nœud but à été produit
- L'autre continue et produit 177 + 101 = 278
- Le moindre coût vérifie lequel est meilleur et celui-ci est retenu



Coût uniforme

- Complète : oui, si le coût de chaque action ≥ ε
- Complexité en temps : nombre de nœuds avec *g* ≤ au coût de la solution optimale

$$O(b^{\lceil C^*/\epsilon \rceil})$$

où C^* est le coût de la solution optimale. Risque d'être plus grand que $O(b^{d+1})$

- Complexité en espace : même que celle en temps
- Optimal : oui, parce que les nœuds sont développés en ordre de $g(n)$.

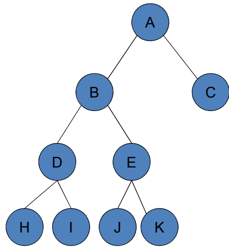
20

Profondeur d'abord

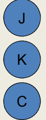
- Développe le nœud le plus profond.
- Implémenté à l'aide d'une pile. Les nouveaux nœuds générés vont sur le dessus.

21

Exemple profondeur d'abord



Pile:



Ordre de visite: A – B – D – H – I – E – J – K - C

22

Propriétés de profondeur d'abord

- Complétude : non, si la profondeur est infinie, s'il y a des cycles. Oui, si on évite les états répétés ou si l'espace de recherche est fini.
- Complexité en temps : $O(b^m)$ Très mauvais si m est plus grand que d. m est la profondeur max.
- Complexité en espace : $O(bm)$, linéaire
- Optimal : Non

23

Profondeur limitée

- L'algorithme de profondeur d'abord, mais avec une limite de l sur la profondeur.
  - Les nœuds de profondeur l n'ont pas de successeurs.
- Complétude : oui, si $l \geq d$
- Complexité en temps : $O(b^l)$
- Complexité en espace : $O(bl)$, linéaire
- Optimal : Non

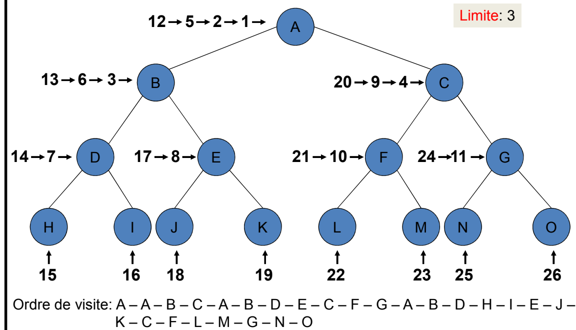
24

Profondeur itérative

- Profondeur limitée, mais en essayant toutes les profondeurs: 0, 1, 2, 3, …
- Évite le problème de trouver une limite pour la recherche profondeur limitée.
- Combine les avantages de largeur d'abord (complète et optimale), mais a la complexité en espace de profondeur d'abord.

25

Exemple profondeur itérative



Ordre de visite: A – A – B – C – A – B – D – E – C – F – G – A – B – D – H – I – E – J – K – C – F – L – M – G – N – O

26

Propriétés profondeur itérative

- Complétude : Oui
- Complexité en temps : $(d+1)b^0 + db^1 + (d-1)b^2 + b^d = O(b^d)$
- Complexité en espace : $O(bd)$
- Optimal : Oui, si le coût de chaque action est de 1. Peut être modifiée pour une stratégie de coût uniforme.

27

Comparaison entre profondeur itérative et largeur d'abord

Comparaison pour b = 10 et d = 5, avec la solution à l'extrémité droite

N(P.I.) = 6 + 50 + 400 + 3000 + 20 000 + 100 000
= 132 456 noeuds

N(L) = 1 + 10 + 1000 + 10 000 + 100 000 + 999 990
= 1 111 101 noeuds

28

Recherche bidirectionnelle

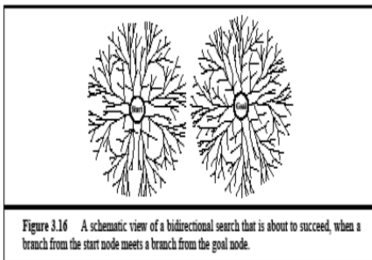


Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

29

Recherche bidirectionnelle

- Recherche simultanée du départ vers le but et du but vers le départ, afin de se rejoindre au milieu.
- Applicable si on peut faire une recherche à partir du but.
- Besoin d'une vérification efficace de l'existence d'un nœud commun aux deux arbres. Il faut conserver tous les nœuds d'au moins un des arbres.

30

Propriété bidirectionnelle

- Complétude : Oui
- Complexité en temps : $O(b^{d/2})$
- Complexité en espace : $O(b^{d/2})$
- Optimale : Oui

Stratégie de recherche non-informées : Sommaire

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Dans ce tableau:

(1) b est le facteur de branchement;

(2) d est la profondeur du but le - profond;

(3) m est la profondeur max;

(4) l est la profondeur auquelle on se limite

Stratégies de recherche informée

- Les stratégies de recherche non-informée ne sont pas très efficaces dans la plupart des cas. Elles sont aveugles car elles ne savent pas si elles s'approchent du but.
- Les stratégies de recherche informée utilisent une fonction d'estimation (heuristique) pour choisir les nœuds à visiter.

Stratégies de recherche informée

- Meilleur d'abord (Best-first)
- Meilleur d'abord gloutonne (Greedy best-first)
- A*
- Algorithme à mémoire limitée
  - IDA*, RDFS et SMA*
- Escalade (Hill-climbing)
- Recuit simulé (Simulated annealing)
- Recherche local en faisceau (local beam)
- Algorithmes génétiques

34

Meilleur d'abord

- L'idée est d'utiliser une fonction d'évaluation qui estime l'intérêt des nœuds et de développer le nœud le plus intéressant.
- Le nœud à développer est choisi selon une fonction d'évaluation $f(n)$.
- Une composante importante de ce type d'algorithme est une fonction heuristique $h(n)$ qui estime le coût du chemin le plus court pour se rendre au but.
- Deux types de recherche meilleur d'abord: A* et meilleur d'abord glouton.

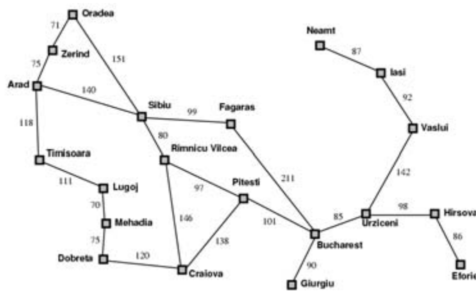
35

Meilleur d'abord gloutonne

- $f(n) = h(n)$ = distance à vol d'oiseau
- Choisit toujours de développer le nœud le plus proche du but.

36

De Arad à Bucharest



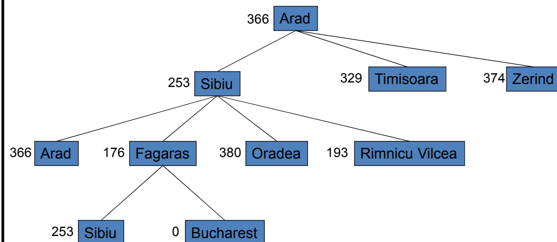
Heuristique du vol d'oiseau

- Soit *h(n)* = distance à vol d'oiseau pour la carte de Roumanie.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 4.1 Values of h_{SLD}—straight-line distances to Bucharest.

Exemple meilleur d'abord gloutonne



Propriétés meilleur d'abord gloutonne

- Complétude : Non, elle peut être prise dans des cycles. Oui, si l'espace de recherche est fini avec vérification des états répétés (pas de cycles).
- Complexité de temps : $O(b^m)$, mais une bonne fonction heuristique peut améliorer grandement la situation.
- Complexité d'espace : $O(b^m)$, elle retient tous les nœuds en mémoire.
- Optimale : non, elle s'arrête à la première solution trouvée.

A*

- Fonction d'évaluation: $f(n) = g(n) + h(n)$
  - *g(n)*: coût du nœud de départ jusqu'au nœud *n*
  - *h(n)*: coût estimé du nœud *n* jusqu'au but
  - *f(n)*: coût total estimé du chemin passant par *n* pour se rendre au but.
- A* utilise une heuristique admissible, c'est-à-dire $h(n) \leq h^*(n)$, où $h^*(n)$ est le véritable coût pour se rendre de *n* au but.
- Demande aussi que $h(n) \geq 0$ et que $h(G) = 0$ pour tous les buts *G*.

Heuristique du vol d'oiseau

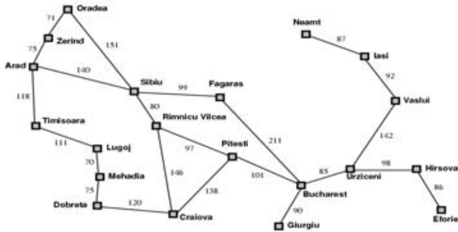
- Soit *h(n)* = distance à vol d'oiseau pour la carte de Roumanie. Cette heurisitique est admissible

Ville	h_{SLD}	Ville	h_{SLD}
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

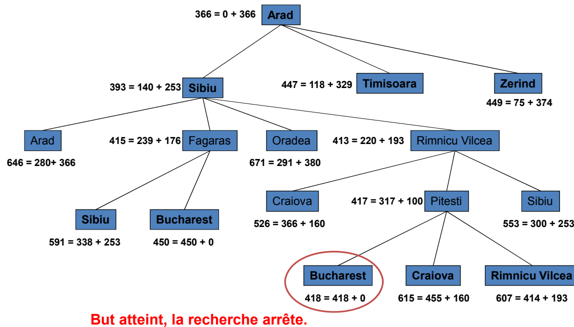
Figure 4.1 Values of h_{SLD}—straight-line distances to Bucharest.

Heuristique du vol d’oiseau (2)

- Soit *h(n)* = distance à vol d’oiseau pour la carte de Roumanie.



Exemple A*

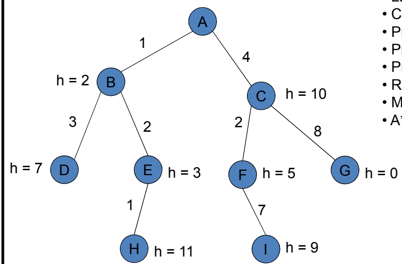


Propriétés de A*

- Complétude : oui
- Complexité de temps : exponentielle selon la longueur de la solution (dépendent de l’espace des états-- $O(b^{\epsilon d})$),
- Complexité en espace : elle garde tous les nœuds en mémoire: exponentielle selon la longueur de la solution $O(b^{\epsilon d})$.
- Optimale : Oui

Habituellement, on manque d’espace longtemps avant de manquer de temps.

Exercice



- Largeur d'abord
- Coût uniforme
- Profondeur d'abord
- Profondeur limitée
- Profondeur itérative
- Recherche bidirectionnelle
- Meilleur d'abord gloutonne
- A*

Dans quel ordre les nœuds sont développés pour chacun des algorithmes?

46

Exercice (2)

Largeur d'abord : A,B,C,D,E,F,G
Profondeur d'abord : A;B;D;E;H;C;F;I;G
Profondeur limitée : (l=2) A;B;D;E;C;F;G
Profondeur itérative :
l=0 A
l=1 A;B;C
l=2 A;B;D;E;C;F;G

47

Exercice (2)

Meilleur d'abord vorace : A;B;E;D;C;G
A* :
f(B) =3
f(C)=14
f(D)=11
f(E)=6
f(F)=11
f(G)=0
f(H)=15
f(I)=22
Par conséquent la séquence serait : A;B;E;D;C;G

48

Recherche à mémoire limitée

- IDA* (Iterative-deepening A*)
  - C'est un algorithme de profondeur itérative, mais qui utilise une valeur de la fonction d'évaluation $f=g+h$ comme limite au lieu de la profondeur.
  - À chaque itération, la limite est fixée à la plus petite valeur de fonction d'évaluation de tous les nœuds qui avaient une valeur plus grande que la limite au tour précédent.
  - Avantage: Prend beaucoup moins de mémoire

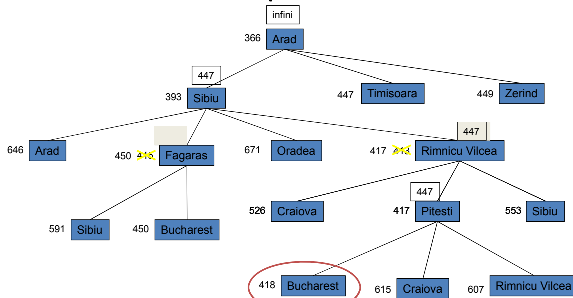
49

Recherche à mémoire limitée

- RBFS (Recursive best-first search)
  - Ressemble au meilleur d'abord, mais avec une complexité d'espace linéaire. $O(bd)$
  - Il ne garde pas tous les nœuds en mémoire
    - Il ne conserve que les options possibles sur le chemin courant.
    - À chaque itération, la limite est la 2e meilleure option disponible.
    - Si la valeur du chemin courant dépasse la meilleure 2e option, il va faire un retour arrière pour développer cette 2e option.

50

Exemple RBFS



But atteint, la recherche arrête.

51

Recherche à mémoire limitée

- IDA* et RBFS utilise trop peu de mémoire
- Ils développent souvent les mêmes nœuds
- Si le programme peut avoir accès à plus de mémoire, il serait intéressant de pouvoir l'utiliser.
- Deux algorithmes qui permettent cela :
  - MA* (Memory-bounded A*)
  - SMA* (Simplified MA*)

52

Recherche à mémoire limitée

- SMA*
  - Exactement comme A*, mais avec une limite sur la mémoire.
  - S'il doit développer un nœud et que la mémoire est pleine, il enlève le plus mauvais nœud et comme RDFS, il enregistre au niveau du père la valeur du meilleur chemin.
  - Complétude : oui, s'il y a une solution atteignable
  - Optimale : oui, si la solution optimale est atteignable
  - Complexité en temps : Peut être très long s'il doit souvent régénérer des nœuds.
  - Complexité en espace : c'est la mémoire allouée.

53

Fonctions heuristiques : Comment?

- $h_1(n)$ = nombre de tuiles mal placées.
- $h_2(n)$ = distance Manhattan totale (nombre de cases pour se rendre à la position désirée pour chaque tuile).
- $h_1(n) = 8$
- $h_2(n) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2$
  $= 18$

7	2	4
5		6
8	3	1

État initial

	1	2
3	4	5
6	7	8

État but

54

Facteur de branchement effectif

- Si le nombre de nœuds traités est de *N* avec une solution de profondeur *d*, alors le facteur de branchement effectif (b^*) est le facteur de branchement d'un arbre uniforme de profondeur *d* contenant *N + 1* nœuds.

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

- Par exemple si A* trouve une solution à la profondeur $d = 5$ en utilisant 5 nœuds, alors $b^* = 1,92$
- Plus la valeur de b^* s'approche de 1, plus l'heuristique est efficace.
- Pour déduire la valeur de b^*, il suffit de faire des tests de complexité variable et de calculer la valeur moyenne.
- b^* donne une bonne idée de l'utilité de l'heuristique et permet aussi de comparer des heuristiques.

55

Comparaison

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Pour tester *h1* et *h2*, 1200 problèmes aléatoires de longueur de solution de 2 à 24 ont été générés; et ils ont été résolus via *IDS* et *A*(h1)* et *A*(h2)*. La fig plus haut donne le nombre moyen de nœuds produit pour chaque stratégie et le facteur b^*. *Les résultats suggèrent que h2 meilleure que h1 (on dit que h1 domine h2)*

56

Dominance

- Si $h_2(n) \geq h_1(n)$ pour tout *n* (les deux étant admissibles), alors $h_2(n)$ domine $h_1(n)$ et par conséquent $h_2(n)$ est meilleure que $h_1(n)$.
- Il est toujours préférable de choisir l'heuristique dominante, car elle va développer moins de nœuds
  - Tous les nœuds développé par $h_2(n)$ vont aussi être développés par $h_1(n)$, et $h_1(n)$ peut aussi développé d'autres nœuds.
- Il vaut généralement mieux utiliser une fonction heuristique avec des valeurs élevées, pourvu qu'elle soit consistante et que son calcul ne prenne pas trop de temps.

57

Inventer une heuristique

- Simplifier (ou relaxer) le problème
- La solution exacte du problème simplifié (ou relaxé) peut être utilisée comme heuristique.
- Le coût de la solution optimale pour le problème simplifié n'est pas plus grand que le coût de la solution optimale du vrai problème.
- Voir des exemples simples dans le livre

Génération automatique d'heuristiques

- En testant plusieurs simplifications du problèmes.
- Si on trouve plusieurs heuristiques et qu'il n'y en a pas une qui en domine une autre, alors on peut utiliser l'heuristique composite :
  - *$h(n) = max\{h_1(n), \ldots, h_m(n)\}$, la fonction la plus précise sur le nœud concerné*

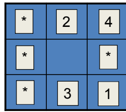
Génération automatique d'heuristiques

- On peut aussi utiliser des « pattern databases (base de données de motifs) », qui contiennent les solutions pour des sous-problèmes:
  - par ex: construire des bases de données (taquin à 8 pièces) pour 5-6-7-8 et 2-4-6-8 et ensuite combiner les heuristiques admissibles
- On peut aussi apprendre une heuristique.
  - Par ex : la caractéristique « nombre de pièces mal placées » peut être très utile pour prédire la distance réelle d'un état but:
  - On peut prendre 200 configs aléatoires et voir que si $x_1(n)$ est à 5, le coût moyen de la solution est à 14 et ainsi de suite …
  - On peut prendre $x_2(n)$ et ensuite combiner pour avoir

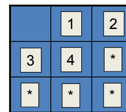
$$h(n) = C_1 X_1(n) + C_2 X_2(n)$$

Un sous-problème

- Un sous-problème du Taquin à 8 pièces .
- La tâche consiste à placer les pièces 1, 2, 3 et 4 à leurs positions correctes, sans se préoccuper des autres pièces



État initial



État but

Au-delà de l'exploration classique (Chap. 4)

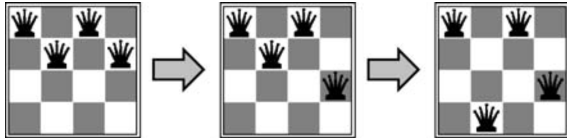
- Escalade (Hill-climbing)
- Recuit simulé (Simulated annealing)
- Recherche local en faisceau (local beam)
- Algorithmes génétiques

Algorithmes d'amélioration itérative

- Dans plusieurs problèmes d'optimisation, le chemin (pour aboutir à une solution) n'est pas important.
- Avec ce type d'algorithmes, l'espace de recherche est l'ensemble des configurations complètes et le but vise soit:
  - à trouver la solution optimale
  - ou à trouver une configuration qui satisfait les contraintes
- On part donc avec une configuration complète et, par amélioration itérative, on essaie d'améliorer la qualité de la solution.

Exemple

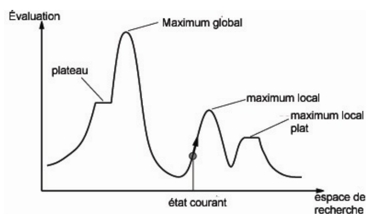
- Le problème des *n* reines
  - Mettre *n* reines sur une planche de jeu de *n* x *n* sans qu'il y ait deux reines sur la même colonne, ligne ou diagonale.



64

Algorithmes d'amélioration itérative

- Utilise très peu de mémoire
- Permet de résoudre des problèmes dans des espaces d'états très grands ou infinis.



65

Algorithmes d'amélioration itérative

- Escalade (Hill-climbing)
- Recuit simulé (Simulated annealing)
- Recherche local en faisceau (local beam)
- Algorithmes génétiques

66

Escalade (Hill-Climbing)

- À chaque tour, il choisit le successeur ayant la meilleure évaluation que l’évaluation du nœud courant.
- Il arrête quand aucun des successeurs n’a une valeur plus grande que le nœud courant.
- Il ne maintient pas d’arbre de recherche et il ne regarde pas en avant.

C’est comme escalader le mont Everest en plein brouillard en souffrant d’amnésie!

67

Exemple 8-reines

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

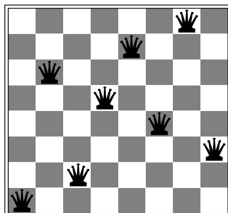
- *h:* le nombre de pairs de reines qui s’attaquent
- Ici, *h = 17*
- On voit la valeur des successeurs possibles quand on déplace chacune des reines dans sa colonne

68

Exemple 8-reines (2)

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

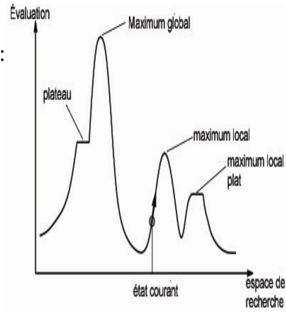
- Minimum local avec *h = 1*



69

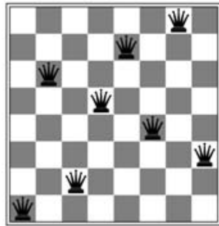
Problèmes du Hill-Climbing

- Il peut souvent rester pris pour les raisons suivantes :
  - Maximum local
  - Plateaux
  - Crête (Ridge)



Exemple minimum local

- *h = 1*
- Tous les successeurs ont une valeur plus grande: Chaque déplacement d'une seule reine fera empirer la situation



Redémarrage aléatoire

- Lorsque l'on obtient aucun progrès :
  - Redémarrage de l'algorithme à un point de départ différent.
  - Sauvegarder la meilleure solution à date.
  - Avec suffisamment de redémarrage, la solution optimale sera éventuellement trouvée.
- Très dépendant de la surface d'états :
  - Si peu de maxima locaux, la réponse optimale sera trouvée rapidement.
  - Si surface en porc-épic, on ne peut espérer mieux qu'un temps exponentiel.
- Habituellement, on peut espérer une solution raisonnable avec peu d'itérations.

Recuit simulé (Simulated Annealing)

- L'idée est de permettre des déplacements perturbateurs dans le but d'échapper aux maxima locaux.
- Principe : Similaire à la descente de gradient avec comme exemple la balle de ping-pong qu'on laisse aller sur une surface inégale. Si on la laisse aller, elle s'immobilisera à un minimum local
- Si l'on agite la surface, on peut faire rebondir la balle hors du minimum local.
- L'astuce consiste à agiter suffisamment pour extraire la balle des minimas locaux, mais pas assez pour la déloger du minimum global.
- Dans le recuit simulé, on commence par agiter fort (haute température) puis on réduit graduellement l'intensité de l'agitation (basse température). Exploration/exploitation

73

Recuit simulé

- Idée: échapper le maximum local en permettant d'aller à une moins bonne place, mais graduellement diminuer la probabilité (fréquence) de faire ceci

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] - VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(ΔE/T)
```

IA2

Recuit Simulé (suite et fin)

- La boucle interne fait la même chose que l'escalade; toutefois au lieu de choisir le meilleur déplacement; il en choisit un au hasard
  - S'il améliore la solution, il est accepté
  - Sinon l'algo accepte un déplacement avec P < 1
    - P décroit exponentiellement avec la mauvaise qualité de la solution; i.e ΔE
    - P décroit également à mesure que *T* baisse; les mauvais déplacements sont plus autorisés au début quand T est elevée; ils le sont de moins en moins quand T baisse.

IA2 75

Recherche local en faisceau

- Fonctionnement:
  - Commence avec *k* états aléatoires
  - À chaque étape, il génère tous les successeurs des *k* états.
  - S'il trouve un but, il s'arrête.
  - Sinon, il choisit les *k* meilleurs successeurs et il recommence.
- Recherche en faisceau stochastique
  - Les *k* successeurs sont choisis aléatoirement, avec une probabilité proportionnelle à la valeur de la fonction d'évaluation (une fonction de valeur pour état).

76

Algorithmes génétiques

- Les AG utilisent certains principes de l'évolution naturelle:
  - Un individu fort a plus de chance de survivre
  - Deux individus forts donnent généralement des enfants forts
  - Si l'environnement évolue lentement, les organismes évoluent et s'adaptent.
  - Occasionnellement, des mutations surviennent, certaines sont bénéfiques et d'autres mortelles.

Les AG utilisent ces principes pour gérer une population d'hypothèses.

77

Fonctionnement

- Les hypothèses sont souvent décrites par des chaînes de bits, mais elles peuvent aussi être décrites par des expressions symboliques ou des programmes informatiques.
- Fonctionnement :
  - Commence avec une population initiale.
  - La reproduction et la mutation donnent naissance à la génération suivante.
  - À chaque génération, les hypothèses sont évaluées selon une certaine fonction d'utilité et les meilleures sont celles qui ont le plus de chance de se reproduire.

78

Représentation des hypothèses

- On utilise souvent des chaînes de bits
- Exemple, représenter des règles si-alors
  - Ciel {Ensoleillé, Nuageux, Pluvieux}
    - Prend une chaîne de 3 bits
    - Un bit à 1 indique que l'attribut peut prendre cette valeur
    - Par exemple, 010 indique Ciel = Nuageux et 011 représente Ciel = Nuageux ∨ Pluvieux
  - On peut ajouter des attributs en juxtaposant deux sous chaînes
    - Par exemple, on pourrait ajouter un attribut Vent{Fort, Faible}

79

Représentation des hypothèses

On obtient, une chaîne de longueur 5

Ciel	Vent
011	10

On peut ajouter une postcondition (Comme JouerTennis = oui)

Si Vent = Fort, alors JouerTennis = Oui

Ciel	Vent	JouerTennis
111	10	1

La représentation contient toujours toutes les sous chaînes même si l'attribut n'est pas dans la règle. Ceci permet d'avoir des chaînes de longueur fixe.

80

Opérateurs génétiques

Reproduction
croisement : générer deux fils en inversant deux sections entre les parents

11101001000 → **11101010101**
00001010101 → **00001001000**

Mutation : inverser un bit au hasard

11101001000 → **11101011000**

Les opérateurs peuvent être adaptés aux représentations

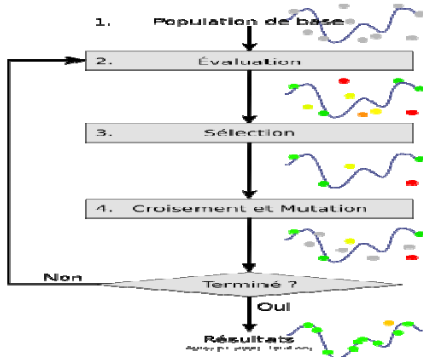
81

Algorithme

- Description de l'algorithme en pseudo-code
  - Sélectionner la population initiale
  - Répéter
    - Sélectionner une partie de la population en fonction de leur adéquation
    - Sélectionner les paires des meilleures solutions pour construire de nouvelles solutions
    - Produire une nouvelle génération en combinant les paires (croisement) des meilleures solutions et en introduisant des mutations
  - Jusqu'à l'atteinte d'une condition d'arrêt

82

Algorithme (Wikepedia)

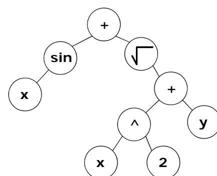


83

Programmation génétique

- Les individus sont des programmes généralement représentés par des arbres.
- Chaque appel de fonction est un nœud et les paramètres sont ses fils.

$\sin(x) + \sqrt{x^2 + y}$



84

Programmation génétique

- Reproduction: Inverser des sous arbres

85

Exemple

- Voyageur de commerce
  - Trouver le chemin le plus court en passant par toutes les villes et en revenant au point de départ

- Gêne: ville
- Individu: liste de villes
- Population: ensemble de parcours

http://lsis.univ-tln.fr/~tollari/TER/AlgoGen1/

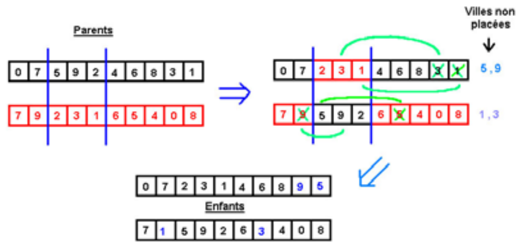
86

Exemple: Reproduction

- On choisit aléatoirement deux points de coupe.
- On intervertit, entre les deux individus, les parties qui se trouvent entre ces deux points.
- On supprime, à l'extérieur des points de coupes, les villes qui sont déjà placées entre les points de coupe.
- On recense les villes qui n'apparaissent pas dans chacun des deux parcours.
- On remplit aléatoirement les trous dans chaque parcours.

87

Exemple: Croisement



88

Exemple: Mutation

- Quand une ville doit être mutée, on choisit aléatoirement une autre ville dans ce parcours et on intervertit les deux villes.

89

Paramétrage

- Combien d'individus dans une population? Trop peu et tous les individus vont devenir semblables; trop grand et le temps de calcul devient prohibitif.
- Quel est le taux de mutation? Trop bas et peu de nouveaux traits apparaîtrons dans la population; trop haut et les générations suivantes ne seront plus semblables aux précédentes.
- Taux de reproduction, nombre maximum de générations, etc.
- Habituellement fait par essai et erreur.

90

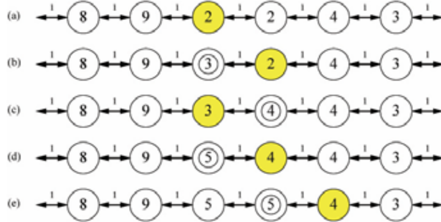
Recherche « online »

- Tous les algorithmes précédents sont des algorithmes « offline ».
  - Ils calculent une solution complète avant de l'exécuter et à l'exécution ils ne font qu'appliquer la solution trouvée.
- Un agent de recherche « online » doit entrelacer les temps de calculs et d'exécution
  - Il fait une action;
  - Il observe l'environnement;
  - Il choisit sa prochaine action.
- Les algorithmes « online » sont assez différents, parce qu'il faut tenir compte que l'agent se déplace
  - Par exemple, A* ne fonctionne pas « online »

91

Algorithmes « online »

- Recherche en profondeur d'abord
- Hill-Climbing
- LRTA*



92