Représentation et résolution de problèmes :

Intelligence artificielle II

Olivier Gasquet, Philippe Muller
IRIT

2006

Plan du cours :

Objectif : Introduire les méthodes de recherches arborescentes informées permettant de trouver une "bonne" solution à un problème à forte combinatoire et ceci en un temps acceptable.

- Introduction à la recherche opérationnelle et l'intelligence artificielle.
- Calculabilité et complexité algorithmique.
- Représentation de la connaissance : éléments de théorie des graphes.
- Représentation de problèmes
  - espaces d'états (méthodes aveugles, informées (heuristiques)).
  - réduction de problèmes en sous-problèmes (arbres ET/OU)...
  - recherche locale dans les problèmes à forte combinatoire

Thèmes des bureaux d'étude (en Camel)

- Programmation d'une procédure de recherche à stratégie paramétrable (largeur ou profondeur)
- Evolution de la procédure précédante en meilleurs d'abord, puis en A*.
- Performance du A* pour le problème du voyageur de commerce avec dix heuristiques et des instances aléatoires.
- implémentation de méthodes de recherche “locales”, expérimentation sur le problème du voyageur de commerce.

Ouvrages conseillés :

- *A la recherche de l'intelligence artificielle*. Daniel Crevier, Champs Flammarion (Poche).
- *Intelligence artificielle et informatique théorique, Alliot et Schiex*, Ed. Cepadues
- *Recherche heuristiquement ordonnée*, Henry Farreny, Ed. Masson.
- *Introduction à la calculabilité*, Pierre WOLPER, InterEdition

Résolution de problèmes et Intelligence artificielle

c'est quoi ?

- une tentative de programmation des ordinateurs pour faire ce que l'humain fait mieux (Rich & Knight)
- une tentative de programmation des ordinateurs pour faire des choses dont on dit qu'elle nécessite de l'"intelligence" quand elles sont faites par des humains
- une étude des calculs qui rendent possible la perception, le raisonnement et l'action (Winston)
- une façon d'étudier le cerveau humain
- une quête du "sens commun"
- une menace pour l'humanité

Le match homme-machine

Les ordis c'est bon pour	Les gens sont meilleurs pour
l'arithmétique	perception, vision, parole, etc.
manipuler des chaînes de caractères	l'action
les tâches répétitives	comprendre et produire des phrases en langage naturel
faire ce qu'on leur dit	apprendre par exemples, observation,...
	le raisonnement de sens commun
	les jeux
	les mathématiques et la logique formelle
	la conception et l'analyse (ingénierie, diagnostic médical, architecture, planification, etc)

Souvent, ce qui est facile pour l'humain et difficile pour la machine (et réciproquement).

Les domaines couverts par l'IA

raisonnement	démonstration automatique, planification
jeux	
apprentissage	
langage	interfaces, recherche d'informations
vision	extraction, reconnaissances de formes, surveillance
diagnostic	pannes, diagnostic médical
base de données	base de connaissances

Domaines concernés

- problèmes mal posés ou mal définis (langage,...)
- domaines en friche (apprentissage,...)
- problèmes complexes ou très larges
- problèmes pour lesquels on ne connaît pas d'algorithmes "efficaces" (rapides) ou ne donnant pas de "bonnes" solutions.

Une méthodologie commune ?

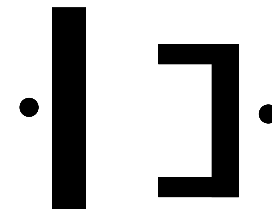
1. modélisation d'un domaine ou d'un problème particulier (mathématique ou logique)
2. représentation
3. résolution (algorithmique, raisonnement, optimisation)
4. évolution

L'étape (3) nécessite des recherches dans des ensembles de possibilités très grands → "recherche opérationnelle". La première partie du cours concernait le point 1 et en partie 2. Cette deuxième partie concerne les points 2 et 3.

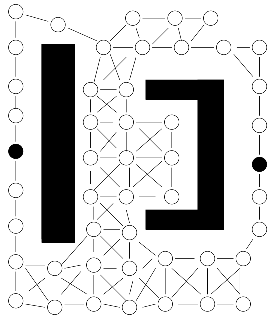
Un exemple: chercher son chemin

Un exemple: chercher son chemin

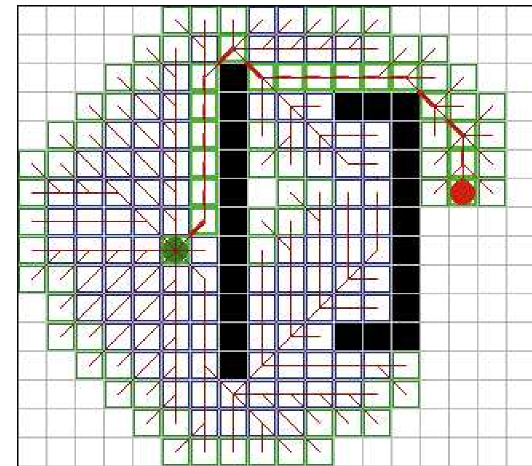
Un exemple: chercher son chemin



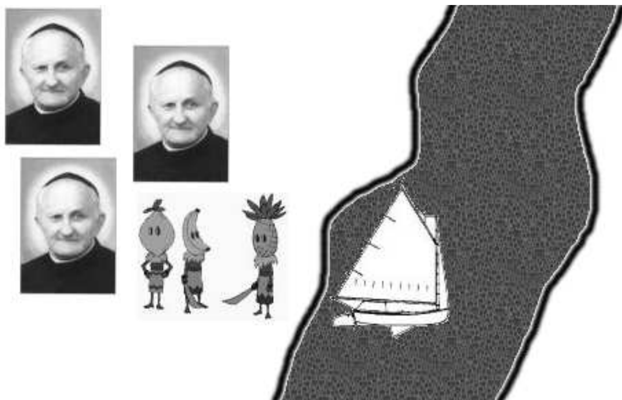
Un exemple: chercher son chemin



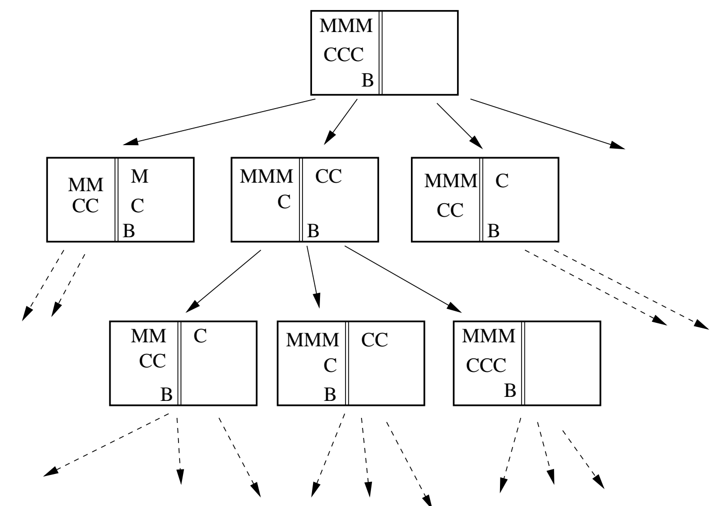
Un exemple: chercher son chemin



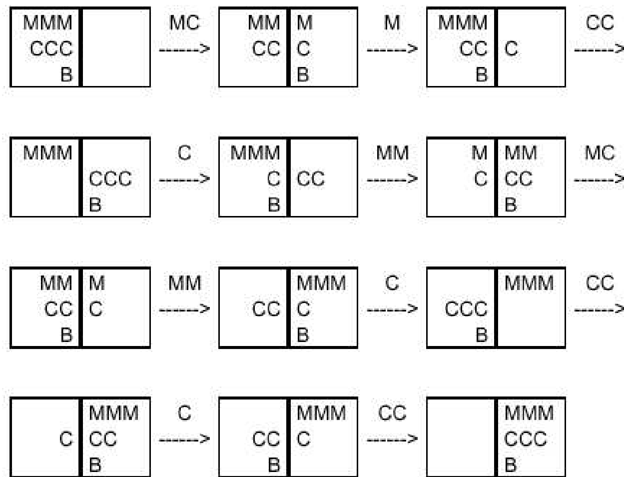
Un autre exemple ? les missionnaires et les cannibales



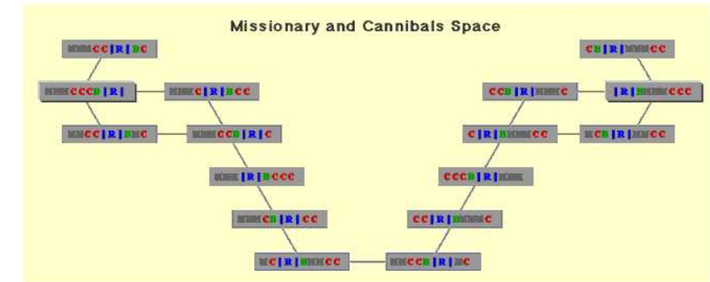
Les missionnaires et les cannibales: exploration



Les missionnaires et les cannibales: solution



Les missionnaires et les cannibales: l'espace de recherche



Complexité algorithmique

Analyse des algorithmes : évaluation des ressources consommées par les algorithmes, en temps d'éxécution et en espace mémoire.

→ moyen de comparaison des algorithmes = "complexité" des algorithmes.

le temps de calcul d'un programme dépend :

- des performances du processeur
- du compilateur
- des données d'entrées du problème
- de l'algorithme

Calculabilité et complexité algorithmique

Questions :

- comment l'évaluer indépendemment de la machine ?
- comment tenir compte des données différentes ?

→ la taille des données d'entrée est un paramètre du temps de calcul.

On notera $T(n)$ le temps de calcul d'un algorithme en fonction de la taille des données d'entrées (variable suivant les instances du problème traité).

notions à retenir:
- **l'ordre de grandeur** du temps de calcul.
- son **évolution**
- **le pire cas**

Taux de croissance

On évalue le temps de calcul en ordre de grandeur, notion mathématique:

$$T(n) = O(f(n)) \quad ssi \quad \lim_{n\to\infty} \frac{T(n)}{f(n)} = cte$$

On dira donc, plutôt que $T(n) = O(2n^2)$ ou $T(n) = O(n^2 + 3n)$, que $T(n) = O(n^2)$.

Plus le taux de croissance est d'ordre inférieur, meilleur est l'algorithme, mais pour des petites tailles de problèmes le temps de calcul peut être meilleur pour un taux supérieur

Déprimant

T(n) / n	10	20	30	40	50	60
n	10μs	20 μs	30 μs	40 μs	50 μs	60 μs
$\log n$	1μs	1.3 μs	1.5 μs	1.6 μs	1.7 μs	1.8 μs
$n \log n$	10μs	26 μs	44 μs	64 μs	85 μs	107 μs
n^2	100μs	400 μs	900 μs	1.6ms	2.5 ms	3.5 ms
n^3	1ms	8 ms	27 ms	64 ms	125 ms	216 ms
n^5	0.1s	3.2s	24.3s	1.7 mn	5.2 mn	13 mn
2^n	1ms	1s	18mn	13jours	36 ans	366 siècles
3^n	59ms	58 mn	6 mn	3855 siècles	2.10^8 siècles	$1.3.10^{13}$ siècl

NB: l'âge de l'univers est estimé à 10^8 siècles.

Encore plus déprimant

N=taille du problème qu'on peut résoudre aujourd'hui

T(n)	aujourd'hui	si 100 fois plus vite	si 1000 fois plus vite
n	N	100N	1000N
n^2	N	10N	32N
n^3	N	4.6N	10N
n^5	N	2.5N	4N
2^n	N	N + 7	N + 10
3^n	N	N + 4	N + 6

Analyse d'algorithmes : Algorithmes itératifs

- constantes
- boucles (simples/doubles)
- tests (majorant)

Exemple: tri à bulle.

```
for i:=1 to n-1 do
      for j:=n downto i+1 do
           if A[j-1] > A[j] then
              begin
                temp = A[j-1];
                a[j-1] = A[j];
                A[j] = temp
              end;
```

Résultat = $O(n^2)$

Exercice : tri par selection:

```
procedure trisel (var a : elem ; n : integer) ;
var i,j,p,min : integer;

begin
    for i := 1 to n-1 do
         begin
            min := a[i] ;
            p := i ;
            for j := i+1 to n do
                if a[j] < min then
                      begin
                          min:=a[j] ;
                          p := j;
                      end;
            a[p] := a[i] ;
            a[i] := min
         end
end
```

Algorithmes récursifs

Exemple : le tri par fusion (mergesort) d'une liste de n=2^m nombres.

```
trifusion(L,n) =
     si n = 1 retour L
     sinon partager L en deux listes L1 et L2 ;
            retour fusion(trifusion(L1,n/2),
                          trifusion(L2,n/2))
```

En supposant que le partage et la fusion sont en $O(n)$, on estime un majorant :
$T(n) \leq c_1$ si n = 1
$T(n) \leq 2T(n/2) + c_2 n$ si n > 1

Pour résoudre les équations récurrentes classiques : la solution générale de $T(n) = an^b + c * T(n/2)$

- si $c < 2^b$ alors l'algo est en $O(n^b)$
- si $c = 2^b$ alors l'algo est en $O(n^b \log n)$
- si $c > 2^b$ alors l'algo est en $O(n^{\log_2 c})$

Résultat tri fusion: $O(n \log n)$

Pour les autres cas:

- trouver un majorant, preuve par récurrence.
- substitutions/ simplifications

  (exemple:

  $T(n) = 2(T(n-2)) + 4) => T(n) = 2^n T(1) + 4n + a$)

Exercices

1. fibonacci doublement récursif (indice: $a*(3/2)^n \leq fib(n) \leq b*(5/2)^n$)

2. fibonacci récursif terminal

3. puissance efficace (avec resursif pair sur n div 2)

4. recherche nb occurences chaînes dans sous-chaînes

Complexité théorique

Au delà des algorithmes : classification des *problèmes*. Il peut y avoir plusieurs algorithmes qui résolvent un problème : on va alors caractériser les problèmes en fonction des algorithmes que l'on peut leur appliquer.

$\rightarrow$ classement des problèmes.

$\rightarrow$ "complexité" d'un problème.

$\rightarrow$ "complexité théorique".

Exemples de problèmes :

- trier une liste d'entiers.
- trouver le plus court chemin passant par n villes (voyageur de commerce).
- trouver un modèle d'une formule en logique propositionnelle.
- montrer un thèorème de mathématique.

On peut diviser les problèmes en trois grandes catégories après ce que l'on a déjà vu (n étant la dimension des données d'entrées) :

- la classe P : ceux pour lesquels il existe un algorithme en $O(n^k)$.
- la classe E : ceux pour lesquels tous les algos sont minorés par a^n
- la classe I : les problèmes indécidables.

La machine de Turing

La machine de turing est une idéalisation d'un processus de calcul mécanisé. Elle consiste en :

- un **ruban** consitutée de cellules contenant les données.
- une **tête** qui pointe sur une cellule du ruban
- un automate qui peut être dans un certain nombres d'**états**, et qui peut faire les actions suivantes : se déplacer le long du ruban, lire ou écrire le ruban, ou s'arrêter.

Un algorithme est la définition exacte du comportement de l'automate en fonction du ruban. Sa complexité en temps est le nombre d'opérations qu'il doit effectuer avant de terminer avec succès.

La machine de Turing

On distingue deux sortes de machine de Turing:

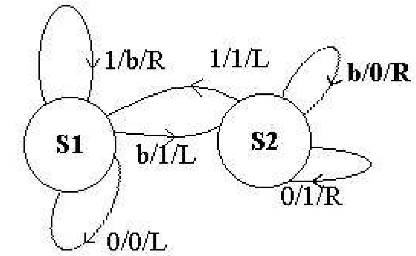
déterministe à chaque instant il est dans un certain état et l'action qu'il effectue ne dépend que de cet état. (correspond +/- à l'ordinateur moderne idéalisé)

non déterministe il existe une instruction de choix non déterministe entre deux états pour déterminer l'état suivant.

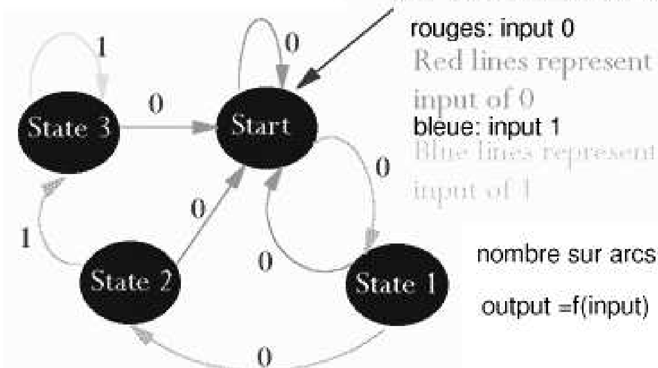
Un algorithme non déterministe accepte un donnée s'il termine avec succès pour au moins un calcul possible. Sa complexité en temps est la longueur du plus petit calcul qui accepte la donnée.

Exemple

State	Read	Write	Move	Next State
S1	0	0	L	S1
	blank	1	L	S2
	1	blank	R	S1
S2	0	1	R	S2
	blank	0	R	S2
	1	1	L	S1



Un autre exemple



L'exemple de SAT

– algo déterministe stupide : tester toutes les instanciations des n variables propositionnelles ($O(2^n)$) et vérifier la formule (en $O(m)$, taille de la formule)

– algo non déterministe : pour toute variable x_i, le choix est entre affecter la variable à vrai ou l'affecter à faux. Quand c'est fini, on teste la formule (si vrai alors succès).

Donc l'algo non déterministe est $< C_1 * n + C_2 * m$ donc en $O(m)$.

En fait l'algo non-dét. est similaire en complexité à celui, déterministe, consistant à vérifier qu'une instanciation du problème est solution ($\rightarrow$ "vérification").

Autre problème: comment ramener un problème d'optimisation à un problème soluble par la MT ? (la MT résoud des problèmes de "reconnaissance")
sur ex. du voyageur de commerce :

reconnaissance existe-t-il un circuit de longueur $\leq c$ donné.

optimisation trouver le circuit optimal

valeur optimale trouver seulement la longueur du circuit optimal

témoin trouver un circuit de longeur $\leq c$ donné

complexité : rec$\leq$ témoin,val. optimale $\leq$ optimisation

Catégories de problèmes

La classe P est la classe des problèmes qui admettent un algorithme déterministe avec un temps d'éxécution polynomial en fonction de la taille du ruban d'entrée.

La classe NP est la classe des problèmes qui admettent un algorithme non-déterministe avec un temps d'éxécution polynomial en fonction de la taille du ruban d'entrée.

Attention: NP veut donc dire "non-deterministe polynomial" et pas "non polynomial".

La confusion est d'autant plus facile que les ordinateurs étant jusqu'à présent uniquement déterministes, les problèmes de NP sont résolus par des algos déterministes non-polynomiaux.

On a donc de façon évidente P $\subseteq$ NP.

La question à 1 million de $ est : a-t-on NP = P ?

NP complétude

retour sur SAT : SAT est dans NP

et il a étémontré (Cook, 1971) que :
P = NP $\Leftrightarrow$ SAT $\in$ P

SAT est un "prototype" des problèmes de NP.

il a été prouvé en fait que tout problème de NP pouvait se ramener à SAT en temps polynomial.

On dit alors que SAT est **NP-complet** $\rightarrow$ tout un ensemble de problèmes de NP sont NP-complet.

Un problème est NP-complet

s'il est dans NP et si tout problème de NP lui est réductible en temps pôlynomial.

exemple de pbs NP complets:

- le voyageur de commerce (Travelling Salesman Problem)
- le sac à dos (Knapsack problem)
- le déménageur (Binpacking)
- vehicle routing problem
- ordonnancement (scheduling)
-

Maîtriser l'explosion combinatoire

quand on n'a pas le choix...

- garder la taille des données petite
- approximations (quand c'est possible) ex: bin packing
- approche probabiliste: algo aléatoires + prières + instances chanceuses $\rightarrow$ on évite le pire cas. TSP, SAT
- heuristiques (cf. méthodes locales): proba + approximation
- méthodes de recherche optimisées (cf résolution de pb.)

pourquoi arrive-t-on à cette différence théorie-pratique ?

Représentation de la connaissance : éléments de théorie des graphes

Définition

Graphe dirigé $G = (N, A)$
N est un ensemble de noeuds (ou sommets)
A est un ensemble d'arêtes, $A \subseteq N \times N$
$(v, w) \in A$ est une arête de v à w (les extrémités de l'arête).
Un graphe est donc une relation binaire sur l'ensemble N. 1.

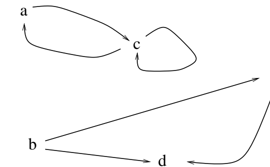


Figure 1: Représentation graphique d'un graphe

L'**ordre** du graphe est son nombre de noeuds.

Un **graphe simple** est le graphe d'une relation irréflexive (aucun noeud n'est relié à lui-même), symétrique.

Graphe non dirigé $(i, j) \in A \rightarrow (j, i) \in A$
c'est un graphe pour lequel l'orientation n'a pas d'importance (on représente tout lien par un lien simple).

Graph complet
On appelle **degré** d'un sommet le nombre d'arêtes dont ce sommet est une extrémité.

Si tous les sommets d'un graphe ont le même degré k, le graphe est dit k-**régulier**.

Pour un graphe orienté, on définit le **degré entrant** (d^+) et le **degré sortant** (d^-) d'un noeud.

Un graphe simple d'ordre n qui est (n-1)-régulier est dit **complet**. Deux sommets quelconques sont alors adjacents.

Propriétés

- Pour un graphe non orienté (N, A) :

$$\sum_{x \in N} d(x) = 2|A|$$

- Pour un graphe orienté (N, A) :

$$\sum_{x \in N} d^+(x) = \sum_{x \in N} d^-(x) = |A|$$

- Un graphe simple a au plus $\frac{n(n-1)}{2}$ arêtes.

Sous-graphe soit un graphe $G = (N, A)$, $N' \subseteq N$, $A' \subseteq A$. $G' = (N', A')$ est un sous-graphe de G si pour tout arête de A', les extrémités de l'arête dans G sont dans N'.

Parcours dans les graphes

Chemin Un chemin entre deux arêtes v et w d'un graphe (N, A) est une suite $C = (v, u_1, \ldots, u_n, w)$ de noeuds de N tels que $(v, u_1) \in A$, $(u_i, u_{i+1}) \in A$ et $(u_n, w) \in A$.

Si $v = w$, le chemin C est fermé.
Si $\forall i, j u_i \neq u_j$, le chemin est simple.
Un **cycle** est un chemin fermé simple.

Fermeture transitive la fermeture transitive d'un graphe dirigé $G = (N, A)$ est un graphe dirigé $G^* = (N, A^*)$ tel que : $(v, w) \in A^* \leftrightarrow$ il y a un chemin de v à w dans G

Graphe non dirigé connexe $\forall i, j \in N, \exists$ un chemin de i à j

composante connexe d'un graphe non dirigé: sous-graphe connecté maximal

composante fortement connexe idem pour graphe orienté

graphe fortement connexe graphe dirigé connexe : il existe un chemin pour toute paire de noeuds (u, v), u et v distincts (de u à v et de v à u).

Exemple : vérification de la forte connexité d'un graphe de rues

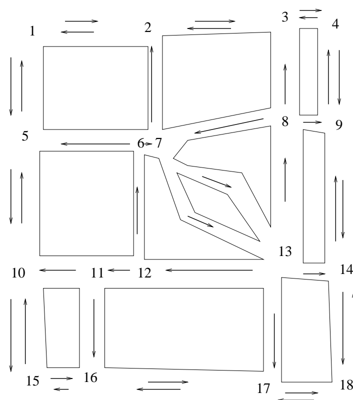


Figure 2: Plan de rues

Arbres

Un arbre est un graphe connexe sans cycle.

Un arbre de n noeuds a donc exactement n-1 arêtes.

Tout graphe connexe contient un graphe partiel (N, A'), avec $A' \subseteq A$, qui est un arbre. On dit que l'arbre est un arbre recouvrant du graphe (en anglais *spanning tree*).

Un arbre pointé est un arbre où on a distingué un sommet qcq. Ce sommet est la racine de l'arbre.

Pour un graphe orienté, la racine de l'arbre est un noeud qui n'a pas d'arcs entrants (il est unique).

Une feuille est un noeud de degré 1 (graphe orienté : uniquement un arc entrant)

Explorations

On appelle exploration ou parcours d'un graphe un procédé déterministe qui permet de choisir un sommet à partir de l'ensemble des sommets déjà visités de manière à passer par tous les sommets du graphe.

Deux types simples de parcours exhaustif : le parcours **en profondeur** et le parcours **en largeur**.

Parcours en profondeur

Il consiste, à partir d'un noeud à toujours explorer un noeud voisin jusqu'à ce qu'il n'y en ait plus, puis à remonter dans le parcours pour visiter les voisins laissés de côté au fur et à mesure.

Pour cela il faut marquer les noeuds déjà visités. Un algorithme type est par exemple :

```
profondeur(G : graphe , x: point de départ)

  marquer(x);
  traitement(x);
  Pile <-  ensemble des y tels que (x,y) est une are
  tant que (Pile pas vide) faire
       enlever un element y de Pile
       si y pas marqué alors profondeur(G,y);
  fin tant que;
```

Ce parcours trouve donc un arbre recouvrant (il y en a plusieurs possibles en général).

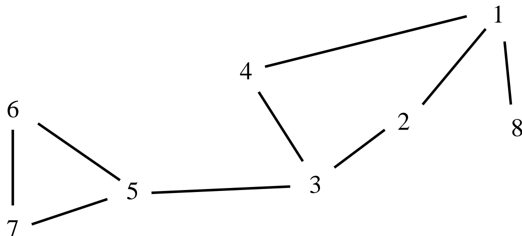


Figure 3: Un exemple de parcours en profondeur

Complexité ? en espace $O(n+m)$

En temps : traitement d'un noeud proportionnel à son degré. Le temps total est proportionnel à la somme des degrés = 2m donc $O(m)$.

Parcours en largeur

```
largeur(G : graphe ; x: point de départ)
  marquer(x);
  mettre x dans une  file F;
  tant que  (file pas vide) faire
          enlever le premier sommet y de la  file;
          traiter(y);
          pour toutes les aretes (y,z)avec z non mar
                  marquer(z);
                  mettre z dans F;
```

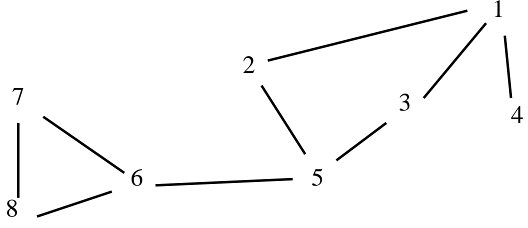


Figure 4: Parcours en largeur

Théorème : si G est connexe tous les sommets seront marqués par l'agorithme et toutes les arêtes seront explorées au moins une fois.

Théorème : soit $G = (N, A)$ un graphe connexe non orienté, soit $T = (N, B)$ un arbre recouvrant en profondeur G, construit par l'algorithme. Toute arête $a \in A$ appartient aussi à B ou bien connecte deux sommets de G tels que l'un des deux soit un ancête de l'autre.

Complexité de l'algorithme :

- en espace :

  $O(n + m)$ pour G,

  $O(n)$ pour T (n sommets et n-1 arêtes) et $O(n)$ pour la file (mais mieux en moyenne) et le marquage.

- en temps : l'exploration du sommet i (le marquage de ses voisins) est fait en parcourant la liste des voisins (d(i) opérations), d'où temps en 2m = O(m).

Partition des sommets en couches

Définition : la **distance** de deux sommets d'un graphe est la longueur du plus court chemin reliant les deux sommets, en convenant que cette distance est infinie s'ils ne sont pas reliés.

on fixe un sommet initial r. Une partition en couches associée à r est la séquence d'ensembles définie comme suit :

$C_0 = \{r\}$
$C_1 = \{x \in N / d(r, x) = 1\}$
$C_i = \{x \in N / d(r, x) = i\}$

Algorithme : on visite les sommets en construisant les couches dans l'ordre en partant de la racine r, et en marquant successivement tous les sommets des couches. La nouvelle couche est créée en prenant la liste des voisins non marquée de la liste précédente.

Représentation et résolution de problèmes

Pour résoudre un problème dans sa généralité, il est nécessaire de suivre une méthodologie assurant que la solution fonctionne dans tous les cas envisageables du problème.

approche rigoureuse :

1. poser correctement le problème : définition du problème

2. analyser la structure du problème

3. définir une stratégie de résolution à partir de l'analyse.

On peut diviser les méthodes de résolution en deux grandes classes, étudiées ici :

1. les méthodes systématiques

2. les méthodes heuristiques (= empiriques).

Résolution sur des graphes d'états simples

On part du principe que tous les problèmes étudiés sont composés d'un ensemble de situations ou objets que l'on peut décrire de façon univoque à l'aide d'un ensemble de variables appelés variables d'états du problèmes.

l'état d'un problème est alors l'ensemble des valeurs prises par les variables à un instant donné.

l'espace d'état d'un problème est l'ensemble de tous les états possibles de ce problème

On peut considérer que la résolution d'un problème est la découverte d'un état du problème ayant des caractéristiques données (=**la solution**).

L'analyse de la structure du problème va généralement permettre d'identifier des liens entre états qui permettent de les visiter tous de façon systématique. Dans cette optique on va chercher :

- un langage formel (un ensembles de symboles) qui permet de représenter le problème.

- des outils de calcul capables de transformer un état en un autre état et donc de permettre de balayer potentiellement tous les états possibles de l'espace d'état.

- une méthode d'organisation de ces transformations pour trouver l'état solution le plus vite possible.

On peut dire que la dernière partie est la seule partie algorithmique (c'est la stratégie de contrôle des opérations) alors que la résolution d'un problème implique aussi fortement les deux premières : la modélisation des données du problème et les opérateurs sur ces données.

Retour sur les cannibales et les missionaires

Première étape de résolution : le codage du problème. Plusieurs solutions s'avèrent possibles :

- Un triplet (X,Y,P) où X est le nombre de missionnaires sur la rive gauche, Y celui de cannibales, et P la position du bateau (G ou D).

- une liste des présents sur chaque rive (M, C ou B).

Dans le premier cas quels sont les codages des états initiaux et finaux et les contraintes portant sur les données ?

1. $0 \leq X \leq 3$ et $0 \leq Y \leq 3$, et P=D ou G
   - état initial = (3,3,G)
   - état final = (0,0,D)
2. - état initial rg=(M,M,M,C,C,C,B) rd=()
   - état final rg=() rd=(M,M,M,C,C,C,B)

Quelle est :

- la plus compacte ? (1!!)

- la plus pratique ? (? peut pas savoir avant de définir les opérateurs)

Exemple des M & C:
Les opérateurs consistent à faire passer 1 ou 2 personnes à la fois de l'autre côté du fleuve en respectant les contraintes du problème :

– Il faut ($Y \leq X$) ou X=0 et $3-Y \leq 3-X$ ou 3-X=0 (au moins autant de M et de C, quand il y a des missionnaires)

Les opérations valides sont donc par exemple :

$(X, Y, G) \rightarrow (X-2, Y, D)$ si $X \geq 2$ et $X-2 \geq Y$ ou $X-2=0$

ou $(2, Y, G) \rightarrow (0, Y, D)$

Opérateurs sur les données/ système de production

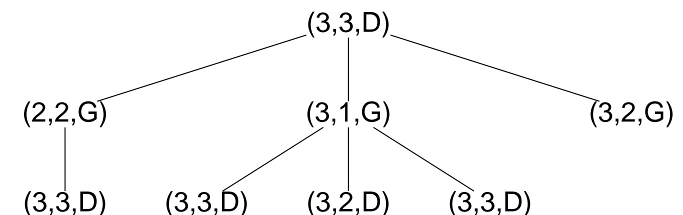
Le système de production est la description de l'ensemble des mécanismes permettant de passer d'un état à un autre du problème. Chaque état possède un ensemble d'états qui lui sont acessibles dans le problème. Les opérateurs permettent de définir cet ensemble, et c'est la stratégie de résolution qui donnera la façon de choisir parmi ces états accessibles celle menant à une solution.

Il y aura deux façons de procéder suivant les opérateurs que l'on définit :

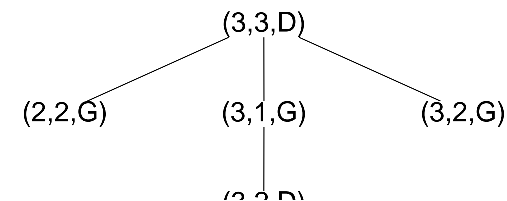
- soit on définit la liste des états accessibles à partir d'un état antérieur du problème et on tente de passer ainsi de l'état initial à l'état final (Raisonnement dit en **chaînage avant**)

- soit on définit la liste des états à partir desquels on peut accéder à un état donné du problème et on tente de remonter de l'état final à l'état initial (Raisonnement dit en **chaînage arrière**)

Représentation formelle plus pratique : la représentation par arbre. A chaque fois que l'on génère un état on le relie à l'état précédent. (sans tester si on génère un état déjà rencontré)



Avec un graphe : même chose avec un test d'occurrence : si un état a déjà été rencontré on ajoute un arc au graphe, sinon un noeud et un arc.



Propriétés des systèmes de production

Monotone un système de production est dit monotone si l'application d'une règle à l'instant t laisse applicable à l'instant t'>t toutes les autres règles qui étaient applicables à l'instant t. (M & C ? non)

Partiellement commutatif un système est partiellement commutatif s'il vérifie la condition suivante :
si X est un état et $(s_1, s_2, ..., s_n)$ une séquence de règles de production telle que
$X \xrightarrow{s_1} X1 \rightarrow ... \rightarrow Y$
alors pour toute permutation σ, la séquence $(s_{\sigma(1)}, ..., s_{\sigma(n)})$ transforme l'état X en Y pourvu que les conditions d'applications des règles soient vérifiées à chaque étape.

Un système est commutatif s'il vérifie cette condition et qu'il est aussi monotone.

Quelques rappels de dénombrement

- le nombre de parties d'un ensembles à n éléments : 2^n
- le nombre de permutations d'un ensemble à n éléments : $n!$
- le nombre de sous ensembles de p éléments d'un ensemble à n éléments: C_n^p

Exercices

Représenter par un espace d'état les problèmes suivants :

le sac à dos on dispose d'un ensemble d'objets, chacun ayant un volume, et un prix. On veut les mettre dans un sac de volume fixé. On peut se poser les questions suivantes:

1. y'a-t-il un moyen de remplir complètement le sac ?
2. comment remplir le sac en maximisant le prix du contenu ?

le déménagement

le gardien de musée

la couverture d'ensemble

Graphes ET/OU

Certains problèmes peuvent être représentés par la technique de réduction de problème, c'est-à-dire que le problème peut être considéré comme **une conjonction de plusieurs sous-problèmes indépendants** les uns des autres, que l'on peut résoudre séparément.

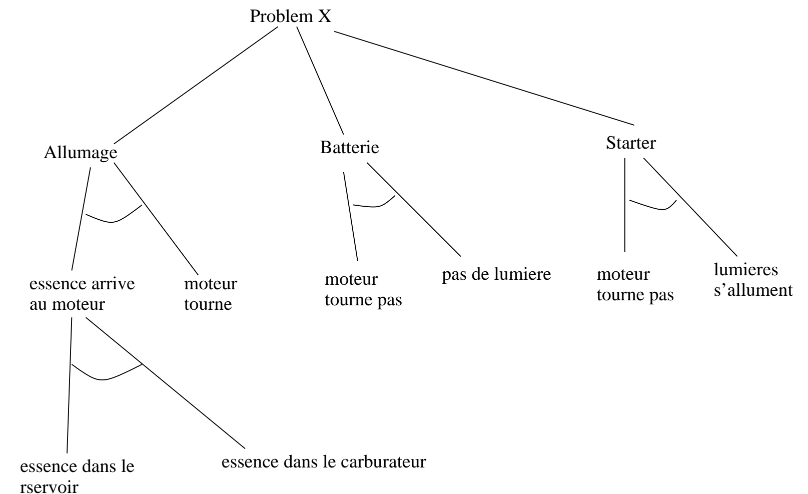
D'autre part, il peut se présenter dans la résolution d'un problème un ensemble de **possibilités indépendantes** les unes des autres (donc exclusives) telles que la résolution d'une seule de ces possibilités suffit à résoudre le problème.

On peut représenter ces deux possibilités par des arcs différents dans le graphe représentant l'espace de recherche :
- Les arcs qui représentent différentes possibilités dans la résolution d'un problème associés au noeud dont ils découlent sont des arcs OU
- Les arcs qui représentent différents sous-problèmes composant la résolution d'un problème associés au noeud dont ils découlent sont des arcs ET.

Un tel graphe est solution d'un problème si :

- il contient le sommet de départ (état initial du pb)
- tous les sommets terminaux (feuilles) sont des problèmes primitifs (indécomposables) résolus.
- s'il contient un arc ET il doit aussi contenir le groupe entier d'arcs ET qui sont frères de cet arc.

Exemple : diagnostic



Un sommet d'un arbre d'exploration d'un espace de recherche est dit résolu si :

- c'est un sommet terminal, ou bien
- c'est un sommet OU non terminal et au moins un de ses axes pointe sur un sommet résolu, ou bien
- c'est un sommet ET non terminal et tous ses arcs pointent sur des sommets résolus.

exemples:

0) M et C que des OU
1) tours de Hanoi, avec n disques. ? que des ET
2) planification

Stratégies de contrôle

On associe à la génération des états une stratégie de contrôle. Elle est dite systématique si elle vérifie les principes suivants :

1. elle permet de générer tous les états sans en oublier.
2. elle ne génère chaque état qu'une fois. Le premier principe garantit la complétude de la procédure (on finit par tomber sur une solution si elle existe.

Le deuxième évite les calculs répétitifs.
Ex: l'algo. suivant parfois appelé "algo du British Museum" (origine ?) est il systématique:

à partir d'un état qcq on génère au hasard l'état suivant jusqu'à trouver une solution. si on est dans une impasse on recommence au début.

Résumé de vocabulaire:

génération d'un noeud calcul du nouveau code de la représentation d'un noeud à partir d'un noeud qcq l'ancien noeud est alors "exploré", le nouveau "généré".

développement d'un noeud génération de tous ses successeurs.

Lors d'une exploration l'ensemble des sommets du graphe de recherche peut être divisé en quatre ensembles.

- les sommets développés déjà vus
- les sommets explorés non encore développés
- les sommets générés mais pas encore explorés (en attente)
- les noeuds toujours pas générés.

L'exemple du taquin

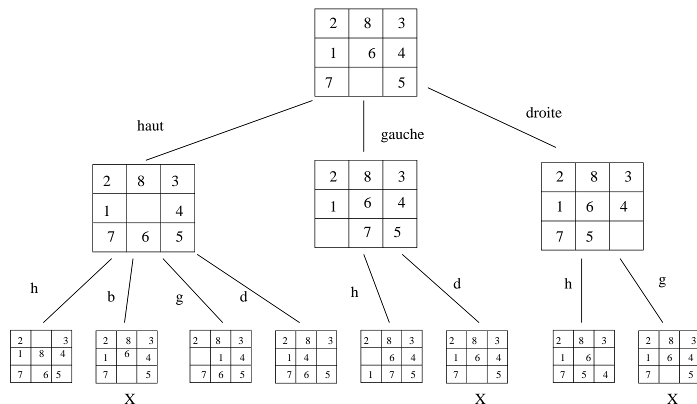


Figure 5: Le taquin

Recherche systématique aveugle (ou non informée)

Recherche en profondeur Chaque sommet choisi pour être exploré voit tous ses successeurs générés avant qu'un autre sommet ne soit exploré. Dans la liste OUVERTS le sommet le plus profond est celui le plus récemment généré. Cette ensemble aura donc une structure de pile.

Une recherche en profondeur peut se révéler dangereuse : l'algorithme risque de développer une branche infinie stérile sans que le mécanisme de retour en arrière puisse se déclencher.On ajoute dans ce cas une limite de profondeur.

Recherche en largeur: Cette stratégie donne une plus grande priorité aux sommets les moins profonds du graphe de recherche en explorant les sommets de même profondeur. L'ensemble OUVERT aura une structure de file.

Méthodes informées / Heuristiques

Une heuristique est un critère ou une méthode permettant de déterminer parmi plusieurs lignes de conduite celle qui promet d'être la plus efficace pour atteindre un but.

Exemples:

Hill-climbing

Meilleur d'abord

Coût uniforme

A*

Hill-climbing (méthode de l'escalade)

Cette méthode de recherche en profondeur utilise une fonction heuristique pour choisir à chaque étape le noeud à générer. C'est la stratégie de l'alpiniste (soi-disant) qui choisit la direction de la pente la plus forte pour arriver au sommet le plus vite.

Cette méthode comporte quelques dangers de dévissage :

- les fonctions d'évaluation peuvent être trompeuses, et certaines améliorations peuvent entraîner dans des branches infinies (danger du pifomètre).
- si on arrive à un maximum local de la fonction d'évaluation (un noeud dont l'évaluation est supérieure à tous ses successeurs), il n'y a plus de choix possibles et l'algo s'arrête, sans avoir trouvé la solution. Il faut alors décider arbitrairement d'une reprise pour relancer la recherche au risque de reprendre un sommet déjà visité.

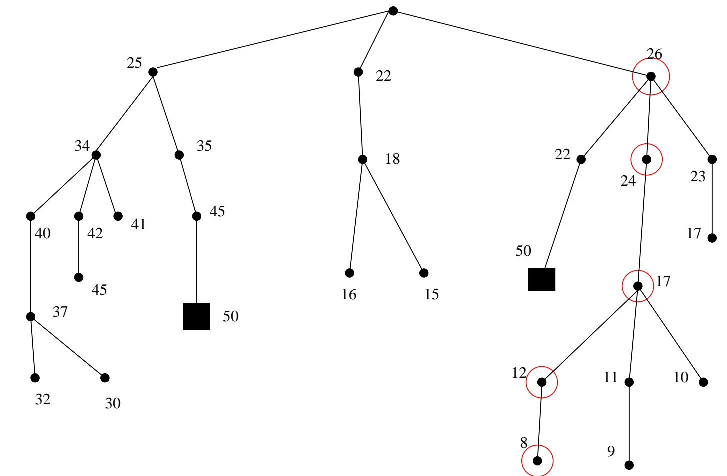


Figure 6: Escalade et profondeur

Stratégie du meilleur d'abord (Best first)

On cherche dans cette stratégie à sélectionner le sommet le plus prometteur par rapport à l'heuristique, parmi tous les sommets rencontrés depuis le début, quelle que soit sa positon dans l'arbre partiellement développé.
Le caractère prometteur d'un sommet peut être estimé de diverses manières :

- évaluer la difficulté de résolution du sous-problème représenté par le sommet

- estimer la qualité de la branche entière qui contient le sommet

- estimer la quantité d'information que l'on s'attend à gagner par rapport à la solution cherchée.

Dans tous les cas l'estimation est numérique et elle est calculée au moyen d'une fonction heuristique d'évaluation f(n) qui peut donc dépendre de n (le noeud), du but, de l'information de toute la branche.

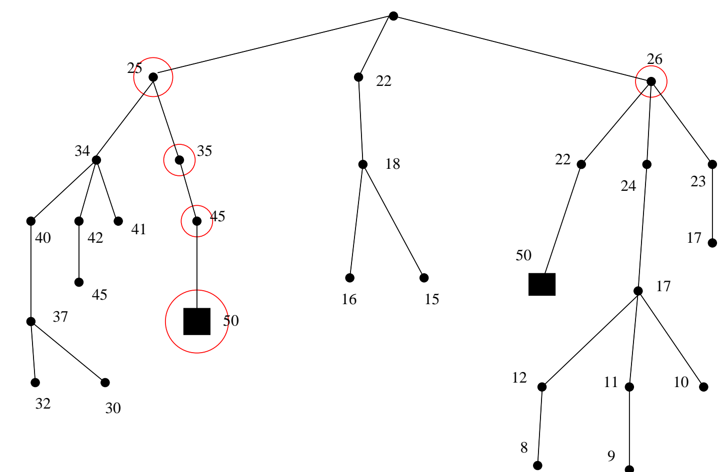


Figure 7: Meilleur d'abord

Graphes valués

chaque arc du graphe est muni d'un poids qui peut correspondre

- au coût d'une opération pour passer d'un état à un autre
- une distance entre deux noeuds d'un graphe
- etc

Le coût uniforme

exemple somme colonnes matrices

trouver la colonne de somme minimum

12	5	7	29
6	23	45	
2			
3			

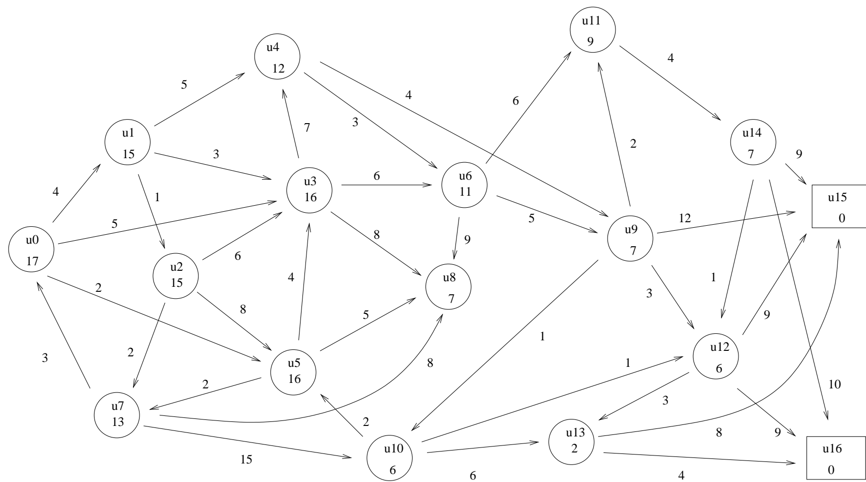
exploration en développant uniquement la colonne minimum courante.

quand on arrive à la fin d'une colonne on peut savoir si c'est le minimum

Stratégie A^* (spécialisation du meilleur d'abord)

- u_0 l'état initial.
- T l'ensemble des états terminaux.
- $P = \{p_1, p_2, ..., pn\}$ l'ensemble des opérateurs
- $h(u)$ h est la fonction heuristique qui estime le coût du passage de l'état u à l'état final.
- $k(u, v)$ k est la fonction qui donne le coût du passage de l'état u à l'état v (arc u,v)
- $g(v)$ est le coût du trajet pour atteindre l'état v
- $f(v)$ est le coût total estimé
- $pere(v)$ est le tableau indexé par les états qui donne pour chaque état celui qui l'a généré.
- OUVERTS= noeuds en attente / FERMES = noeuds déjà vus

mettre u_0 dans OUVERTS
g(u_0) ← 0
f(u_0) ← 0
trouve ← faux
tantque OUVERT≠Ø et not(trouve) **faire**
  n ← enlever_tete(OUVERTS)
  mettre n dans FERMES
  si n∈ T **alors**
    trouve←vrai
    sortir en retraçant les pointeurs père de n à u_0
  sinon
    développer n avec les p_i → successeurs de n
    pour tout $n' = p_i(n)$ **faire**
      si n'∉ (OUVERTS ∪ FERMES) ou g(n')>g(n)+k(n,n') **alors**
        calculer h(n')
        g(n') ← g(n)+k(n,n')
        f(n') ← g(n')+h(n')
        pere(n') ← n
        insere selon f (OUVERTS,n')



Cas particuliers :

- Si la longueur du chemin n'a pas d'importance on prend k=0
- Si on veut minimiser le nombre d'étapes, on prend k=1 pour tout arc.

La fonction h(n) tente d'estimer le minimum sur tous les chemins de n à l'état final de la somme des k(n,..) sur le chemin

La valeur réelle de ce miminum est notée $h^*(n)$

Plusieurs cas d'heuristiques :

heuristique parfaite h est parfaite ssi $h(u) = h(v) \leftrightarrow h^*(u) = h^*(v)$

heuristique presque parfaite $h(u) < h(v) \leftrightarrow h^*(u) < h^*(v)$

heuristique monotone (ou consistante) si pour tout v descendant de u, $h(u) - h(v) \leq k(u, v)$

heuristique minorante ou admissible $h(u) \leq h^*(u)$

Propriétés formelles des méthodes heuristiques

Notations:
n_j est un successeur de n_i sera noté $n_j \in succ(n_i)$

Γ est l'ensemble des sommets terminaux du graphe de l'espace de recherche.

$P_{n_i - n_j}$ ={ chemins de n_i à n_j}

$P_{n-\Gamma}$ ={ chemins de n à Γ}

de même $p_{n_i-n_j}$, $p_{n-\gamma}$ $P^*_{n_i-n_j}$, chemins de coût minimum.

$k(n_i, n_j)$ coût d'un chemin minimum de n_i à n_j.

Les coûts de chemin contenant u_0 l'état initial où un élément de Γ portent des noms spéciaux :

$g^*(n) =$ coût minimum des chemins reliant u_0 à un sommet n.

$g^*(n) = k(u_0, n)$

$h^*(n) =$ coût minimum des chemins allant de n à Γ

$$h^*(n) = min_{\gamma \in \Gamma} k(n, \gamma)$$

Γ^* st l'ensemble des buts optimaux γ accessibles depuis u_0 par un chemin de coût minimum c^*.
$c^* = k^*(u_0)$

Chaque arc porte une étiquette de coût positif $c(n_i, n_j) > delta > 0$.

A^*: recherche optimale d'une solution optimale :

La fonction f de A^* consiste à ajouter deux parties g(n) et h(n) où :

- g(n)= le coût du chemin courant de u_0 à n avec $g(u_0) = 0$
- h(n)= une estimation du chemin restant $h^*(n)$ avec h(solution)=0

Notons γ une solution. Lorsqu'on voudra spécifier les coûts réels de P_{u_0-n} et de $P_{n-\gamma}$ sur un chemin particulier P on écrira $g_p(n)$ et $h_p(n)$. Pour tout chemin p on $g_p(n) \geq g^*(n)$ et $h_p(n) \geq h^*(n)$

Lorsque g et h coïncident avec leur valeurs optimales, la fonction f résultante prend une signification particulière.

Soit $f^*(n) = g^*(n) + h^*(n)$, $f^*(n)$ est le coût optimal pour tous les chemins solution passant par u_0.

En particulier $f^*(u_0) = h^*(u_0) = g*(\gamma) = f^*(\gamma) = c^*, \forall \gamma \in \Gamma^*$

Propriétés algorithmiques de A^*

Définitions

Complétude : un algo est dit complet lorqu'il débouche sur une solution quand il en existe une.

Admissibilité : un algorithme est dit admissible lorsqu'il donne une solution optimale à chaque fois qu'li en existe une.

Dominance : un algorithme A_1 domine un algo A_2 si tout sommet développé par A_1 l'est aussi par A_2. A_1 domine strictement A_2 si A_1 domine A_2 mais A_2 ne domine pas A_1. (algo plus efficace, car il développe moins de sommets).

Optimalité : un algorithme est optimal dans une classe d'algorithme s'il domine tous les autres membres de sa classe.

Proposition : Si n^* est un sommet quelconque sur un chemin optimal allant de u_0 à γ il doit satisfaire : $f^*(n^*) = c^*$.

Preuve : soit $n^* \in P_{n^*-\gamma}$, il existe un chemin solution p passant par n dont le coût est c^*. Sur ce chemin $g^*(n) + h^*(n) \leq c^*$. Si jamais on avait $g^*(n) + h^*(n) < c^*$, il existerait un chemin p' tel que $g'_p(n) + h'_p(n) < c^*$, ce qui est en contradiction avec l'optimalité de c^*.

Si n est un sommet qui n'appartient pas à un chemin optimal solution on a $f^*(n) > c^*$

Terminaison et complétude

A^* s'arrête toujours sur les graphes finis.

A^* est complet pour tous les graphes.

Chaque fois que h(n) est une estimation optimiste de $h^*(n)$ A^* retourne une solution optimale. Cette classe d'heuristiques est dite heuristiques admissibles. (cad quand $h(n) \leq h^*(n)$).

A^* est **admissible** ssi il utilise une heuristique admissible

Si l'heuristique est **parfaite**, l'algo convergera immédiatement vers le but (sans explorer de branches inutiles).

Si h est **minorante** alors A^* trouvera toujours le chemin optimal.

Dans le pire des cas la complexité en temps de A^* est en 2^N, N étant le nombre de sommets du graphe d'état.

Si h est minorante la complexité est en N^2

Si h est monotone la complexité est linéaire en N

Preuve optimiste⇒ admissible

les grandes lignes :

1. pour toute itération de A^* il y a un état ouvert qui $\in$ à un chemin optimal et pour lequel $g(u) = g^*(u)$
2. la tête de ouvert $\hat{u}$ est telle que $f(\hat{u}) \leq f^*(u_0)$
3. à l'arrêt de l'algorithme, $\hat{u} \in \Gamma$, et donc $f(\hat{u}) = f^*(\hat{u})$
4. $f^*(u_0) = min_{u \in \Gamma}(f^*(u))$
5. à l'arrêt $f(\hat{u}) = f^*(\hat{u_0})$

Détail des preuves

Preuve (1):

soit $(u_0, u_1, ...u_t)$ un chemin optimal. si u_1 a été rencontré, il est soit dans ouvert soit dans fermé. s'il est dans fermé, u_2 a été rencontré, il est soit dans ouvert soit dans fermé, ... de même pour les autres u_i. comme u_t n'est pas encore développé, il y a un u_q avec $q < t$ dans ouvert. le chemin $(u_0, ...u_q)$ est connu, et optimal de u_0 à u_q. donc $g(u_q) = g^*(u_q)$

Preuve (2)

$$f(\hat{u}) \leq f(u_q) = g(u_q) + h(u_q) \leq g^*(u_q) + h^*(u_q) = f^*(u_q) = f^*(u_0)$$

Preuve (5) à l'arrêt, $f(\hat{u}) \leq f^*(\hat{u_0})$ (2) et donc d'après (3) $f^*(\hat{u}) = f(\hat{u})$ et $f^*(\hat{u}) \leq f^*(\hat{u_0})$

d'après (4) comme $\hat{u} \in \Gamma$ on a $f^*(\hat{u_0}) \leq f^*(\hat{u})$

Monotone ⇒ admissible

monotone: $\forall u, v h(u) - h(v) \leq k(u, v)$

minorante ?: $h(u) \leq h^*(u)$

or $h^*(u) = k(u, v_1) + k(v_1, v_2) + ... + k(v_{f-1}, v_f)$ avec $v_f \in \Gamma$

et $h(u) = h(u) - h(v_1) + (h(v_1) - h(v_2)) + ... + h(v_{f-1}) - h(v_f)$ (car $h(v_f) = 0$)

si h monotone, pour tout v_i, $(h(v_i) - h(v_{i+1})) \leq k(v_i, v_{i+1})$

d'où $h(u) \leq k(u, v_1) + k(v_1, v_2) + ... + k(v_{f-1}, v_f) = h^*(u)$

cqfd

Monotone $\Rightarrow$ f toujours croissante

$f(n) = g(n) + h(n)$

n' successeur de n :

$f(n') = g(n') + h(n') = g(n) + k(n, n') + h(n')$

d'où $f(n) - f(n') = h(n) - h(n') - k(n, n') \leq 0$

Exemple d'application de A^*: le taquin.

On veut minimiser la longueur des solutions. Tous les arcs de l'espace d'état auront donc un coût associé de 1.

L'heuristique h choisie est :

h(n)=distance_de_manatthan(n)+3*score de déplacement(n).

score de déplacement de n= somme des scores des cases du taquin:
- le pavé central a une valeur de 1
- un pavé non central vaut 0 si son successeur se trouve juste après lui en tournant dans le sens des aiguilles d'une montre
- tous les autres valent 2

Exemple:

1	3	4
8		2
7	6	5

manhattan=4
score de dep=7

Arbres de jeux

Plan:

- Représentation
- Le minmax
- Amélioration du minmax: alpha-beta

Les jeux d'opposition

- deux joueurs, chaque joueur à un certain nombre de coups possibles $\Rightarrow$ arbre de jeux, niveau impair= joueur A, niveau pair=joueur B
- on suppose que les joueurs sont rationnels et ont la même stratégie
- Comment trouver le meilleur coup dans une situation donnée ?

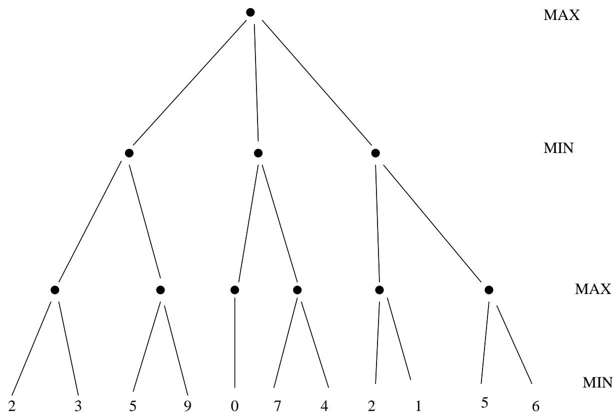
  Si on peut développer l'arbre en entier, on cherche une suite de coups ("stratégie") qui mène toujours à une situation de victoire.

  Si elle existe le premier joueur peut toujours gagner

  Si elle n'existe pas, on parle de jeu à somme nulle.
- en pratique on ne peut pas développer l'arbre entier

  $\rightarrow$ on s'arrête à une profondeur donnée et on **évalue** les situations avec une **heuristique** (encore !)
- On cherche alors à maximiser ses chances d'arriver dans une "bonne" situation

Exemple de stratégie: le minmax



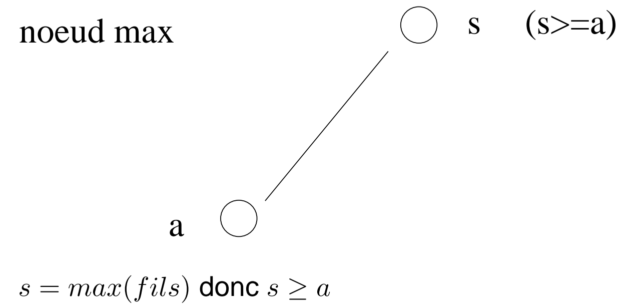
Un “vrai” jeu : le morpion

heuristique: h(n)=(nb de lignes faisables par A)-(nb de lignes faisables par B)

Amélioration du minmax: l’alpha-beta

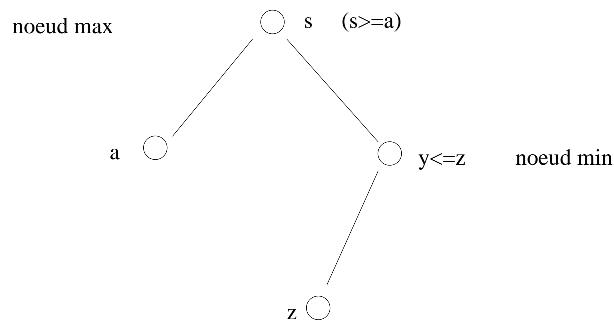
ne pas explorer de branche inutile ...

Alpha-beta: schéma général



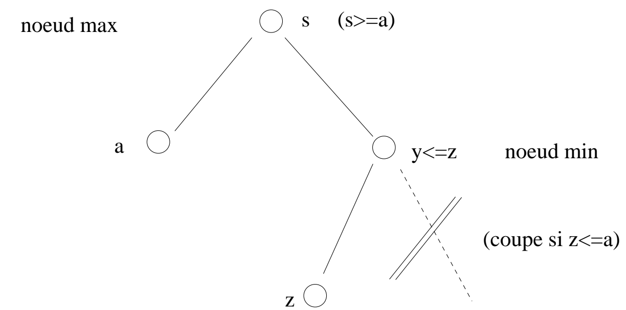
$s = max(fils)$ donc $s \geq a$

Alpha-beta: schéma général



$y = min(fils)$ donc $y \leq z$

Alpha-beta: schéma général



si $z \leq a$, on a en fait $y \leq z \leq a \leq s$
$\Rightarrow$ inutile de regarder les autres fils de y

Alpha-beta: schéma général

A chaque niveau on garde 2 valeurs, et l'intervalle $[\alpha, \beta]$ contraint la suite:

- une valeur choisie au niveau max devient la borne sup (le beta) du niveau min au-dessus
- une valeur choisie au niveau min devient la borne inf (alpha) du niveau au-dessus.
- si la valeur est en dehors, on peut arreter l'exploration du noeud père
- initialement, $\alpha = \beta = \infty$

Alpha-beta: exemple

exos avec autre arbre, 1) minmax, 2) a-b de G a D 3) a-b de D à G

Recherche locale dans les problèmes à forte combinatoire

Quelques Problèmes des méthodes exactes/complètes

1. Les méthodes complètes/exactes explorent de façon systématique l'espace de recherche. $\rightarrow$ ne marche pas dans de nombreux problèmes à cause explosion combinatoire.

2. ces méthodes sont généralement à base de recherche arborescente
   $\rightarrow$ contrainte par l'ordre des noeuds dans l'arbre
   $\rightarrow$ impossibilité de compromis qualité/temps ($\neq$ algo "anytime")

Or il y a souvent des cas ou on peut se contenter d'une solution approchée, pourvu qu'elle soit suffisament "bonne". (ex du TSP, on ne veut pas forcement le minimum mais quelque chose d'assez court, par exemple $<=$ distance donnée).

Un autre principe: les méthodes incomplètes

- si on ne peut prouver que l'on a la meilleure solution,
- et si ce n'est pas grave si on trouve un optimum le plus rapidement possible, de façon assez bonne / à un critère déterminé.

on peux adopter deux types de stratégie :

- garder une méthode complète et stopper après un temps déterminé (mais suivant les problèmes ça n'est pas toujours possible)
- plonger dans l'arbre de recherche avec des heuristiques sans jamais revenir en arrière (mais: comment être sur qu'on va trouver vite une bonne solution ?)

Une famille de méthodes incomplètes: les méthode "locales"

Le principe d'une **recherche locale** est de

1. partir d'une solution sinon approchée du moins potentiellement bonne et d'essayer de l'améliorer itérativement.

   Pour améliorer une solution on ne fait que de légers changements (on parle de changement **local**, ou de solution **voisine**).

2. relancer la méthode plusieurs fois en changeant le point de départ pour avoir plus de couverture

3. tout problème est considéré comme un problème d'optimisation (même les problèmes de satisfaction : le coût à optimiser est alors le nombre de contraintes insatisfaites).

Quelques définitions

une solution est une affectation de toutes les variables du problème.

une solution optimale est une solution de coût minimal

un mouvement est une opération élémentaire permettant de passer d'une solution à une solution voisine (exemple: changer la valeur d'une variable, échanger deux variables).

le voisinage d'une solution est l'ensemble des solutions voisines, c'est à dire l'ensemble des solutions accessibles par un mouvement (et un seul).

un essai est une succession de mouvements.

une recherche locale est une succession d'essais.

Un algorithme de recherche locale typique

```
A* := creer_une_solution()
pour tout  t=1 a max_essais  faire
   A:=nouvelle_solution()
   pour tout  m=1 a max_mouvements faire
      A':=choisir_voisin(A)
      d := (f(A')-f(A))
      si  acceptable(d) alors
         A:=A'
   si  f(A*)>f(A) alors
      A*:=A
retourner A*,f(A*)
```

f: fonction à optimiser

Les paramètres à régler

- max_essais : le nombre d'essai à réaliser
- max_mouvements : ...
- creer_une_solution : génère une solution (en général aléatoirement)
- nouvelle_solution : même chose
- choisir_un_voisin : choisit dans le voisinage de la solution courante (c'est généralement ce qui caractérise principalement une méthode de recherche locale)
- acceptable: accepte ou pas la nouvelle solution générée

Exemples de recherches locales

le hill-climbing Aussi appelé descente en gradient suivant le sens de l'optimisation (min ou max recherché).
choisir_un_voisin : choix aléatoire dans le voisinage courant.
acceptable : seulement s'il il y a une amélioration ($d \leq 0$).
méthode **opportuniste**

la version gloutonne (*greedy*)
choisir_un_voisin : après avoir déterminé l'ensemble des meilleures solutions voisines de la solution courante (exceptée celle-ci), on en choisit une aléatoirement.
acceptable : toujours.
il peut y avoir des dégradations (provisoires) de solutions.

minimisation de conflits

choisir_un_voisin :

1. déterminer l'ensemble des variables en conflit (eg. liées dans un ensemble de contraintes).
2. choisir une de ces variables au hasard.
3. déterminer l'ensemble de ses meilleures valeurs (recherche complète ou pas ?).
4. en choisir une au hasard, affecter la variable.
5. retourner la solution.

acceptable : toujours.
il ne peut y avoir de dégradation de solutions ($\rightarrow$ attention aux optimum locaux).

Les classiques en question(s)

Comportement général :

1. une majorité de mouvements améliorent la solution courante.
2. le nb d'améliorations devient de plus en plus faible.
3. il n'y a plus d'améliorations : on est dans un optimum, qui peut être local.

Les questions à se poser (1)

- quand faut-il s'arrêter ?
- faut-il être opportuniste ou gourmand ?
- comment ajuster les paramètres ?
- comment comparer les performances de deux méthodes différentes ? (qualité de la solution vs. temps consommé)

Illustration (1)

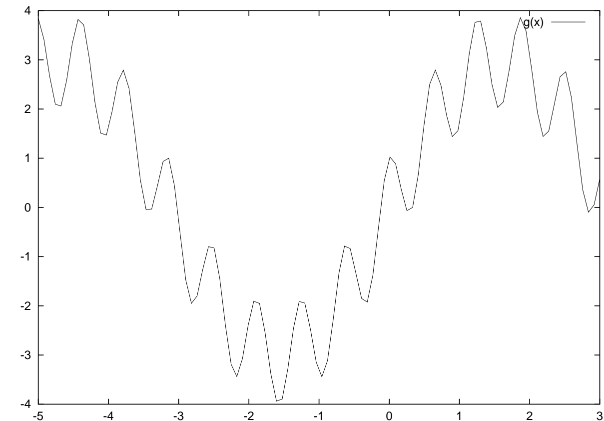


Illustration (2)

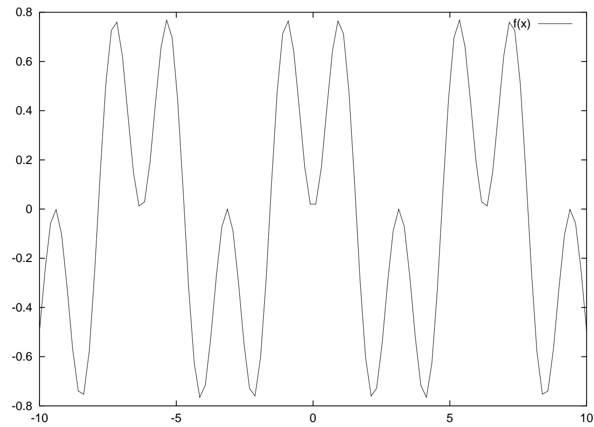


Illustration (3)

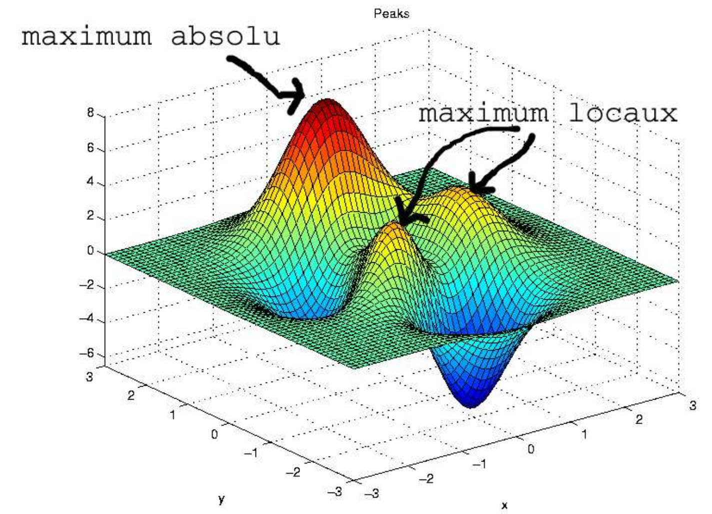


Illustration (4)

en montant le plus tôt possible

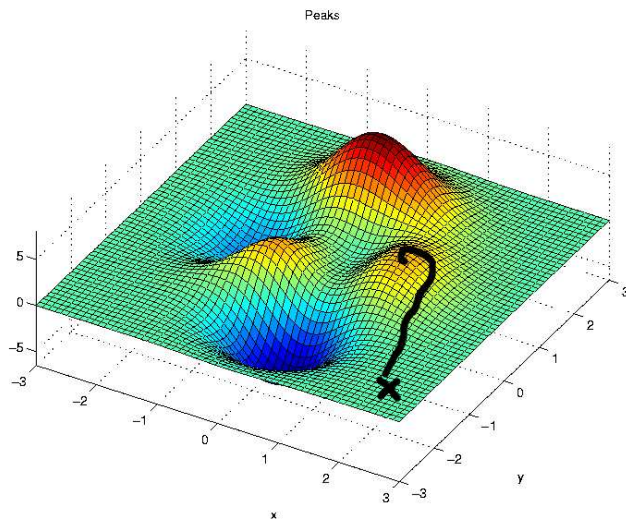
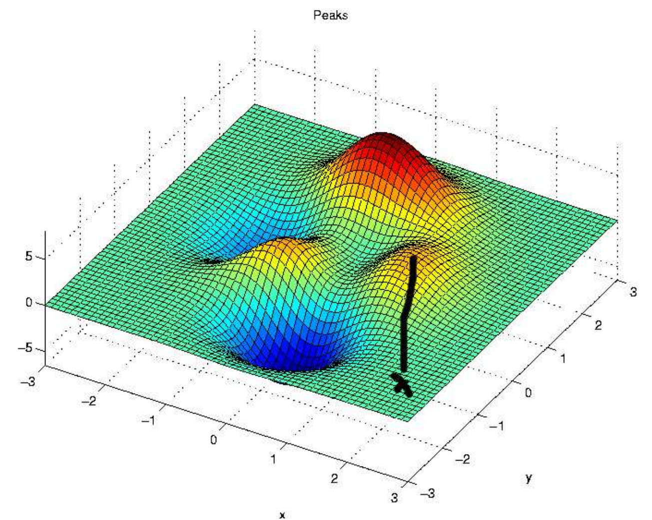


Illustration (5)

en suivant la pente la plus forte



Les questions à se poser (2)

Pour le critère d'arrêt :

- pour l'essai en cours : nb de mouvements max ?
  comment détecter un optimum local ?
- pour la recherche elle-même : le nb max d'essai ?
  coût de la solution convenable ?
  limite de temps atteinte.

Avidité ou opportuniste :

- si aucune recherche d'amélioration : comment trouver les bonnes solutions ?
- si recherche d'amélioration, comment éviter optimum local ?

Exercice: adaptation au TSP

- initialisation ?
- quel voisinage ? (échange de villes, échange d'arcs)
- complexité selon le choix opportuniste/gourmand

Amélioration des classiques : l'approche probabiliste

Pour éviter d'être coincé dans un optimum local, on peut ajouter du **bruit** : une part de mouvements aléatoires pendant une recherche classique.

Principe :

- avec une proba p, faire un mouvement aléatoire.
- avec une proba $1 - p$, suivre la méthode originale.

Reste le problème : comment régler p ?

Amélioration des classiques : le Tabou

basée sur une recherche par gradient

- choix dans le voisinage
- possibilité de détérioration de la solution courante
- Pour améliorer la recherche, on mémorise les k dernières solutions courantes et on interdit le retour sur une de ces solutions.
- on autorisera en plus toujours un mouvement conduisant à une meilleure solution que la meilleure obtenue auparavant.

En pratique, au lieu de mémoriser les k dernières solutions courantes (parfois trop coûteux en temps et mémoire), on mémorise les k derniers mouvements (plus restrictif mais peu coûteux en temps)

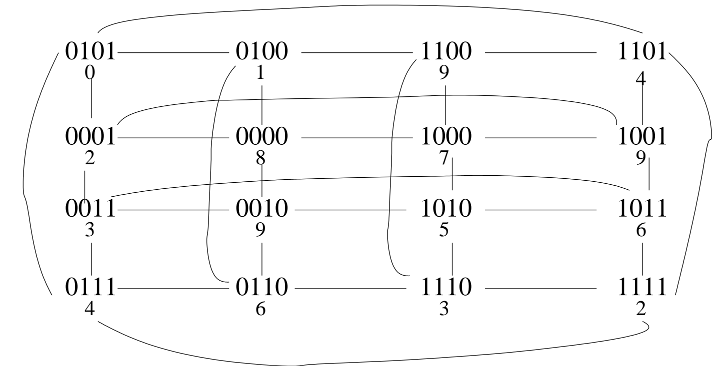
L'algorithme

A ← solution_initiale()
Am ← A
T ← Ø
tantque non(condition de fin) **faire**
  C ← voisinage(A)-T
  Y ← min pour f sur C
  si f(Y)≥f(A) **alors**
    T ← T ∪{A}
  si f(Y)<f(Am) **alors**
    Am ← Y
  A ← Y
retourner Am, f(Am)

La condition de fin est soit une limite en temps ou en itération, soit une condition de non amélioration de la solution.

Exercice

Développer l'algorithme avec $|T| = 1$, puis $|T| = 3$, puis en conservant les mouvements au lieu des états.



Amélioration des classiques : le recuit simulé

méthode inspirée par une analogie avec un phénomène de physique statistique, l'obtention d'un état stable d'un métal.

En termes de recherches locales, la probabilité d'acceptation d'un mouvement de A à A' :
$p(\phi(A') - \phi(A)) = 1$ si $\phi(A') \leq \phi(A)$
$p(\phi(A') - \phi(A)) = e^{\frac{-(\phi(A')-\phi(A))}{T}}$ si $\phi(A') > \phi(A)$
(issu de lois thermodynamiques)

Principe : on simule le processus de recuit des métaux :

- on part d'une température T élevée (instabilité/ loin de l'optimum).
- on refroidit lentement par plateaux (réduction du bruit).

A* := creer_une_solution()
A := creer_une_solution()
T := T_initiale
tantque $(T > T_{min})$ **faire**
  pour m:=1 a m:=longueur_des_plateaux **faire**
    A':=choisir_voisin(A)
    d := (f(A')-f(A))
    si d≤0 **alors**
      A:=A'
    sinon
      si vrai avec proba $p = e^{-d/T}$ **alors**
        A:=A'
  si f(A*)>f(A) **alors**
    A*:=A
  T:=T*α
retourner A*,f(A*)

Réglage des paramètres

- valeur de T_initiale : si la valeur T_init permet d'accepter x% des configurations de départ, alors on la garde sinon on la double et on recommence (x au moins plus de 50, certains auteurs : 80).
- la durée des paliers (longueur du plateau) : le plus simple est de le borner par le nombre de configurations atteignables en un mouvement élémentaire. là encore subtilités possibles
- la décroissance de la température : le plus simple est facteur constant, de l'ordre de 0.9 / 0.95 (décroissance lente). pour raffiner, le réglage se joue entre ce paramètre (+/- rapide) et le précédent (plateau +/- long).
- condition d'arrêt : la température mini ; soit 0 mais pas très efficace, soit quand améliorations très petites (très empirique).

Avantages des recherches locales

- méthodes rapides (et temps paramétrable) ;
- faciles à implémenter ;
- donnent souvent de bonnes solutions ;
- fonctionnement intuitif

Inconvénients des recherches locales

- manque de modélisation mathématiques (chaque cas est différent : il faut adapter la recherche à chaque problème particulier)
- difficiles à paramétrer (=bidouille !)
- aucune évaluation de la distance à l'optimum (pas une approximation, trouve au pire un optimum local qui n'a rien à voir)

Les Algorithmes génétiques

Les algorithmes génétiques sont une forme de recherche locale qui se démarque un peu des méthodes précédentes. Le principe est inspiré d'une analogie biologique : on considère un ensemble de solutions comme une **population d'individus** susceptible d'évoluer et de se croiser, en suivant certaines règles proches de la sélection naturelle.

L'intérêt principal est de couvrir l'espace de recherche plus largement.

Trois opérations sont réalisées successivement sur la population :

- la sélection des meilleures solutions/individus en proportion/ de leur adaptattion ou avec une proba proportionnelle ;
- le croisement d'individus entre eux ;
- la mutation éventuelle d'un ou de plusieurs individus : changements aléatoires ;

L'algorithme

```
population initiale P avec n solutions
  pour  g:=1 a max générations  faire
    C := vide
    pour  m:=1 a n par pas de 2 faire
      parent1 := selection(P)
      parent2 := selection(P)
      enfant1,enfant2:=croisement(parent1,parent2)
      C:= C + mutation(enfant1) + mutation(enfant2)
    P := C
```

Quelques caractéristiques des Algorithmes génétiques

– Historiquement, un individu était codé par une chaîne de bits (son "patrimoine génétique"). En pratique chaque problème appelle un codage particulier.

– La sélection se base sur l'adaptation (ou **fitness**) d'un individu, donnée par une fonction d'utilité (on cherche à maximiser l'adaptation des individus).

– La sélection permet de focaliser la recherche sur les zones intéressantes, alors que mutations et croisements permettent de diversifier la recherche (on retrouve, plus lâchement, le paradigme des recherches locales).

– Les algorithmes génétiques sont semblables à des recherches locales effectuées +/- en parallèle.

– Utilité dans le cas de problèmes dont on ignore la structure (espace de recherche mal connu par exemple

Quelques inconvénients des algorithmes génétiques

Les problèmes techniques importants sont :

- celui du codage des configurations en binaire
- celui de la recombinaison de solutions pour garder quelque chose de pas trop loin de l'optimum
- performances (temps, mémoire)

Conclusion

les méthodes incomplètes fournissent un ensemble de résolutions empiriques variées, parfois trop, car l'absence de modélisation les rend difficiles à adapter d'un problème à un autre (bricolage...).

Par contre elles sont à peu près indifférentes à la taille du problème, et donnent de bons résultats (approchés) à condition de bien les paramétrer.

→ peuvent fournir un bon complément aux méthodes complètes.