Algorithmes de recherche heuristique

A. Belaïd
Université de Nancy 2

Références

- http://turing.cs.pub.ro/auf2/html/chapters/chapter3/chapter_3_2_1.html
- http://www.loria.fr/~bougrain/support/cnam_heuristiques.pdf
- http://www.irit.fr/~Philippe.Muller/Cours/RO_si/poly_ia_2004.pdf
- http://www.cours.polymtl.ca/inf4215/documentation/CoursRechercheSol.pdf

Plan

- Recherche du meilleur d'abord
- Recherche gourmande
- Algorithme A*
- Variantes de A*: IDA* et SMA*

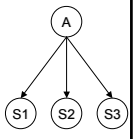
Introduction

- Les algorithmes de recherche aveugle ou "non-informés"
  - n'exploitent aucune information présente dans l'arbre de recherche pour optimiser la recherche
  - mènent une recherche systématique à travers l'espace jusqu'à trouver une solution
  - ne peuvent pas résoudre beaucoup de problèmes réels à cause de la complexité de leurs espaces d'états
- Les algorithmes de recherche heuristique
  - utilisent l'information disponible pour rendre le processus de recherche plus efficace
  - une information heuristique est une règle ou une méthode qui permet d'évaluer la probabilité qu'un chemin allant du nœud courant au nœud solution soit meilleur que les autres

Fonction heuristique

- Une fonction heuristique : h : E → R
  - fait correspondre à un état s ∈ E (espace d'états) un nombre h(s)∈R qui est (généralement) une estimation du rapport coût/bénéfice de suivre un chemin vers la solution qui passe par l'état *s dans E*
  - Propriété :
    h(solution) = 0
- Exemple :
  - Le nœud A a 3 successeurs pour lesquels :
    - h(s1) = 0.8 h(s2) = 2.0 h(s3) = 1.6

  ➔ la poursuite de la recherche par s1 est "heuristiquement" la meilleure

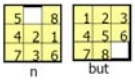


Fonction heuristique

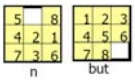
- Idée de base
  - consiste à déterminer pour chaque nœud exploré *n* :
    H(n) = *{h(s) / s ∈ successeurs(n)}*
  - et de continuer la recherche avec le nœud s_0, tel que $h(s_0)$ soit le minimum/maximum de ***H(n)***, selon l'heuristique utilisée
- Parce que les algorithmes de recherche heuristique tirent profit d'une information partielle, en dehors de l'espace d'états, ils sont appelés aussi des :
  - *algorithmes informés*

Fonction heuristique

- Exemples de fonctions heuristiques : puzzle-8



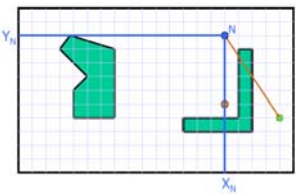
h(n) = nombre de plaquettes mal placées
= 6



h(n) = somme des distances (Manhattan) de chaque plaquette à sa position finale
= 3 + 1 + 3 + 0 + 2 + 1 + 0 + 3
= 13

Fonction heuristique

- Autre exemple : déplacement d'un robot



h(n) = distance en ligne droite vers l'objectif
= $[(Xg - XN)^2 + (Yg - YN)^2]^{1/2}$

Recherche du meilleur d'abord ou Best-First Search (BFS)

- L'algorithme BFS
  - explore le meilleur chemin (en étendant à chaque fois le nœud qui a la meilleure valeur h(s) mais garde les autres chemins en attente
  - Si à un moment donné, le chemin choisi devient moins intéressant que les autres chemins, l'algorithme peut revenir sur les autres chemins
- = combinaison entre la recherche en profondeur et en largeur
  - En profondeur
    - Avantage : cherche toujours un seul chemin, le plus prometteur selon la fonction heuristique utilisée
  - En largeur
    - Avantage : ne risque pas de rester pris dans un "cul-de-sac", à chaque pas il prend en considération tous les successeurs d'un nœud pour déterminer l'ensemble ***H(n)***

Recherche du meilleur d'abord ou Best-First Search (BFS)

```
Fonction RechercheMeilleurD_abord (étatInitial, ensemble_opérateurs, h) res : nœud
    S ← initFile(S, (Nœud(étatInitial), 0) )
    tant que non fileVide(S) faire
        NœudCourant ← extraire ( S )
        si Test_But(NœudCourant)= vrai alors retourne NœudCourant
        sinon
            pour chaque op dans ensemble_opérateurs faire
                x ← Successeur(NodeCourant,op)
                si Valide(x) alors S←insérer(S,x, h(x)) fsi
            fpour
        fsi
    fin tant que
    retourne vide
Fin
```

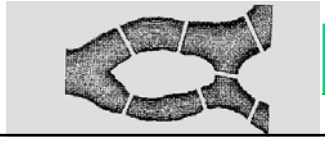

Recherche du meilleur d'abord ou Best-First Search (BFS)

- Insérer(S,x,h(x))
  - insère le nœud *x* dans S, de façon à ce que tous les éléments de S soient ordonnés dans l'ordre croissant des valeurs de *h(x)*
  - Utilise *inserFile*
- Extraire(S)
  - retourne le nœud *x* ayant la plus petite valeur *h(x)* de tous les nœuds qui se trouvent en S puis élimine *x* de S
  - Dans le cas où il y a un ensemble de nœuds ayant la même valeur minimum :
    - elle décide en faveur du nœud solution, s'il est contenu dans cet ensemble, sinon elle choisit au hasard
  - Utilise extraFile

Recherche du meilleur d'abord ou Best-First Search (BFS)

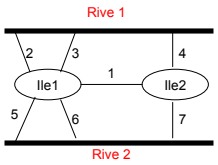
http://www.recreomath.qc.ca/dict_konigsberg_p.htm

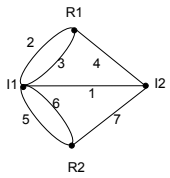
- Exemple : problème des ponts de Königsberg
  - *La ville de Königsberg est sur 2 îles. Elle est reliée à la terre ferme par sept ponts. On doit tracer un itinéraire partant d'un point quelconque de la ville et permettant de traverser une et une seule fois chacun des sept ponts pour revenir au point de départ.* Le problème consiste à déterminer s'il existe un chemin ne passant qu'une seule fois par chaque ligne du graphe associé à ce problème



Recherche du meilleur d'abord
ou Best-First Search (BFS)

- Problème des ponts de Königsberg
  - Représentation du problème en graphe d'états





Recherche du meilleur d'abord
ou Best-First Search (BFS)

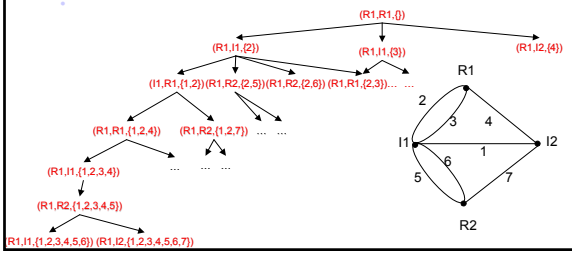
- Problème des ponts de Königsberg
  - La façon classique de résoudre ce problème est par l'utilisation de la théorie de graphes
  - Dans cette théorie, il est prouvé que pour pouvoir faire le tour en utilisant tous les arcs (ponts) une seule fois, il faut que chaque nœud possède un nombre pair d'arcs (Euler)
  - Or ce n'est pas le cas dans ce graphe
  - Ainsi, la réponse est qu'il n'y a pas de solution possible à ce problème
    - Prouvez ce résultat en construisant l'espace d'états et en montrant qu'il n'y a pas de chemin possible

Recherche du meilleur d'abord
ou Best-First Search (BFS)

- Solution : problème de ponts de Königsberg
  - La recherche explore l'espace d'états pour tenter de trouver une solution
  - Pour ce problème, on trouve une solution si en partant de l'état de départ (X, X, {}), on arrive à un état (X, X, {1,2,3,4,5,6,7})
  - La question est de savoir si cela est possible
  - Pour cela, il faut essayer tous les points de départ possibles
  - Prenons d'abord R1 comme le point de départ, on a la structure partiellement développée suivante

Recherche du meilleur d'abord
ou Best-First Search (BFS)

- Solution : problème de ponts de Königsberg
  - On voit bien qu'il n'y a pas de chemin possible



Recherche du meilleur d'abord
Cas particuliers

- Recherche gourmande (Greedy Search)
  - Stratégie la plus simple du BFS
    - h(n) = estimation du coût du nœud n au but
  - Consiste à minimiser le coût estimé pour atteindre le but
    - le nœud qui semble être le plus proche du but sera étendu en priorité
  - Fonctions heuristiques classiques
    - distance à vol d'oiseau
    - distance "Manhattan" :
      = somme des déplacements élémentaires horizontaux et verticaux

Recherche gourmande

- Modification de **RechercheMeilleurD_abord**
  - *h(n)*
    - estime le coût du chemin entre le nœud courant et le nœud solution
  - Insérer(S,x, h(x))
    - insère le nœud *x* dans la structure S, de façon à ce que tous les éléments de S soient ordonnés dans l'ordre croissant des valeurs de la fonction heuristique *h(x)*
  - Extraire(S)
    - retourne et élimine de S le nœud le plus proche du nœud solution

Recherche gourmande

- Qualité des heuristiques
  - *soit* N = nombre total d'états produits pour obtenir la solution
  - soit d = profondeur à laquelle la solution a été trouvée
    - alors b* est le facteur de branchement effectif d'un arbre fictif parfaitement équilibré tel que :

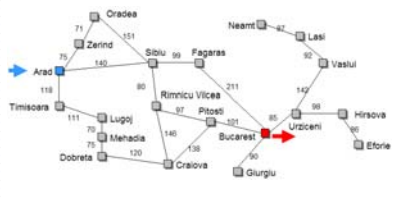
      $N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$
  - Une bonne fonction heuristique aura une valeur de b* proche de 1
    - Trouve la solution tout de suite

Recherche gourmande

- *Complétude*
  - NON, on peut rester pris dans une boucle ; il peut être complet pour les espaces d'états finis et acycliques
- *Complexité en temps*
  - exponentielle : $O(b^p)$ où p est la profondeur et b est le facteur de branchement
- *Complexité en espace*
  - exponentielle : $O(b^p)$
- *Optimalité*
  - NON

Recherche gourmande

- *Exemple1 : voyage en Roumanie*

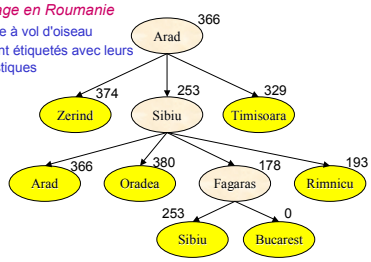


Distance directe (vol d'oiseau) à Bucarest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Recherche gourmande

- *Exemple1 : voyage en Roumanie*
  - *h(n)* = distance à vol d'oiseau
  - les nœuds sont étiquetés avec leurs valeurs heuristiques



Recherche gourmande

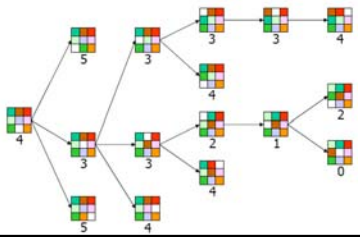
- *Commentaires*
  - *La solution*
    - *Arad > Sibiu > Fagaras > Bucarest*

    n'est pas optimale; elle est de 32 km plus longue que
    - *Arad > Sibiu > Rimnicu > Pitesti > Bucarest*

    car "greedy search" ne considère pas la distance déjà parcourue !
  - Stratégie :
    - toujours enlever le plus grand morceau du coût restant pour atteindre le but (greedy = gourmand), c.à.d minimiser le coût estimé pour atteindre la solution
  - relativement efficace, quoique pas toujours optimal
    - susceptible de faux départ
  - Exemple : pour aller de Iasi à Fagaras, Greedy-Search considère Neamt avant Vaului, même si c'est un cul-de-sac

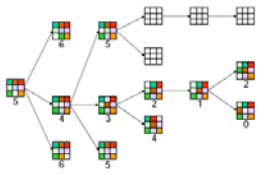
Recherche gourmande

- *Exemple2 : puzzle-8 (taquin)*
  - *h(n)* = nombre de plaquettes mal placées



Recherche gourmande

- *Exemple2 : puzzle-8 (taquin)*
  - *h(n)* = ∑ déplacements des plaquettes



Recherche heuristique

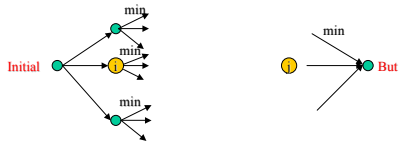
- *La recherche en coût uniforme (Uniform Cost Search)*
  - Cet algorithme utilise une heuristique qui calcule pour chaque nœud *n* le coût chemin *g(n)* depuis l'état initial jusqu'au nœud *n*
  - Le coût chemin *g(n)* est une fonction croissante le long d'un chemin : ***chacune** n **dans** E, s **dans Successeurs**(n),*
    $g(n) \leq g(s)$
  - Les nœuds sont insérés dans la file ordonnée S dans l'ordre croissant de la fonction *g(n)*, de façon qu'à chaque itération le nœud qui minimise *g(n)* soit développé pour obtenir ainsi un chemin de coût minimum entre l `état initial et la solution
  - L'algorithme est *complet* et *optimal*, mais a une efficacité réduite

Recherche heuristique

- *Programmation dynamique asynchrone : recherche du plus court chemin*
  - La programmation dynamique est basée sur le principe d'optimalité qui affirme qu'un chemin est optimal si tous ses sous chemins le sont
  - Pour évaluer le coût d'un chemin qui part d'un nœud i et va jusqu'à un nœud but il faut évaluer les coûts de tous les chemins qui partent de i et passent par les nœuds j, successeurs de i :
    - f*(j) = d(i,j) + h*(j)
  - où h*(j) est la distance la plus courte de i jusqu'à un nœud but
    - On aura h*(i) = min {f*(j) / **∀j dans Successeurs**(i)}
  - Un algorithme de programmation dynamique utilise le principe d'optimalité pour construire pas à pas le chemin vers la solution en étendant des sous-chemins de longueur croissante, qui sont tous optimaux

Recherche heuristique

- *Programmation dynamique asynchrone*



- *Idée de la construction*
  - *Construire tous les chemins*
  - *Arrivé au but, remonter le minimum afin de découvrir le chemin optimal*

Recherche heuristique

- *Algorithme général*

```
pour chaque op dans ensemble_opérateurs, avec op valide faire
    j = Successeur(i,op)
    f(j) = d(i,j) + h(j)
    si h(i) > f(j)
        alors
        h(i)=f(j)
        jmin(i) = j
    fin si
fin pour
```

  - *Le chemin sera représenté par la chaîne :*
    - *s, jmin(s), jmin(jmin(s)) , … , but*

Recherche heuristique

- Exemple : comparaison de chaînes
  - Exemples :
    - séquences d'ADN, mots manuscrits, etc.
  - La comparaison avec la référence se ramène à calculer des distances entre chaînes
  - Sachant que les deux chaînes ne sont pas identiques, il faudra définir des opérations élémentaires (caractère à caractère) et des coûts associés pour transformer ces chaînes afin de les rendre identiques et pour évaluer numériquement le coût de cette transformation

Recherche heuristique

- Levinstein
  - Propose 3 opérations par caractère (ou symbole) :
    - Remplacement ou substitution
    - Ajout
    - Suppression

x= M I L L E R
y= H U L L I E R

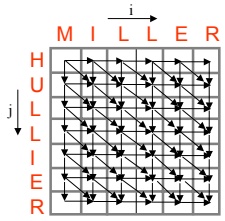
Recherche heuristique

- Distance d'édition (Levinstein)
  - Si on donnait une distance à chaque opération dont la valeur serait une fonction de la correspondance :
    - Ajout : distance forte d_1
    - Retrait : distance moyenne d_2
      - Plus facile de gommer une différence que de l'accentuer
    - Correspondance : distance faible d_3
      - Voire nulle, reflétant une certaine proximité locale
  - On pourrait calculer une distante totale égale à
    - $D = \Sigma(d_i)$
      - Où d_i est la distance minimale retenue à chaque étape i

Recherche heuristique
Comparaison dynamique

- Calcul de la distance : Wagner et Fisher
  - Etablir une table (graphe) de distances (avec toutes les possibilités)

i → i+1, j → j+1 : Substitution
i → i+1, j → j : Destruction
i → i, j → j+1 : Ajout



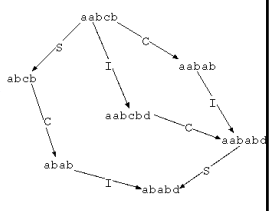
Recherche heuristique
Comparaison dynamique

- Algorithme de Wagner et Fisher en O(m.n) :

On note m et n les longueurs respectives de X et Y
$\delta(0,0)=0$;
Pour i variant de 1 à n effectuer :
$\delta(i,0)= \delta(i-1,0) + \gamma(a_i,\lambda)$
Pour j variant de 1 à m effectuer :
$\delta(0,j)= \delta(0,j-1) + \gamma(\lambda, b_j)$
Pour i variant de 1 à n effectuer :
Pour j variant de 1 à m effectuer :
$m_1 = \delta(i-1,j-1) + \gamma(a_i,b_j)$
$m_2 = \delta(i-1,j) + \gamma(a_i,\lambda)$
$m_3 = \delta(i,j-1) + \gamma(\lambda,b_j)$
$\delta(i,j)=\min(m_1, m_2, m_3)$
La distance d'édition est le résultat final de l'algorithme $\delta(m,n)$

Recherche heuristique
Comparaison dynamique

- Cet exemple :
  - Donne quelques-uns des chemins permettant de passer de aabcb à ababd
  - En prenant un coût de 1 pour chacune des transformations, la distance d'édition de aabcb à ababd est de 3
  - Elle serait différente si le coût d'une suppression était de 0,5, celui d'une insertion de 0,75 et celui d'un changement de 0,25. Sa valeur serait alors de 1,5



Recherche heuristique
Comparaison dynamique

- Exercice :
  - Soient deux listes X={a a b a c} et Y={a b d}, et les coûts : $c(\lambda,y_k) = 0.5\ \forall k$, $c(x_r,\lambda) = 0.5\ \forall r$, $c(x_r,y_k) = 0\ \forall r,k$ si $x_r = y_k$ et $c(x_r,y_k) = 1\ \forall r,k$ si $x_r \neq y_k$
  - Construire le tableau des distances, puis calculer la distance totale entre les deux chaînes

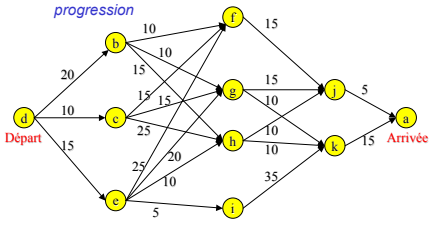
Y \ X	λ	a	a	b	a	c
λ						
a						
b						
d						

Recherche heuristique

- *Exercice*
  - Utiliser l'algorithme de programmation dynamique pour trouver le chemin le plus court entre deux villes

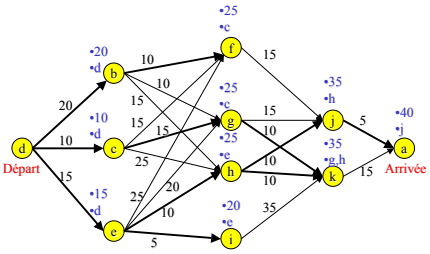
Recherche heuristique

- *Exercice : Étape 1 :*
  - *construire l'arbre des connexions, si possible par niveau de progression*



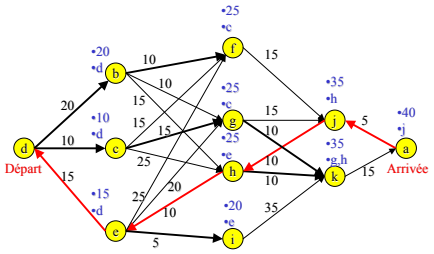
Recherche heuristique

- *Exercice : Étape 2 :*
  - *construire le minimum par ville et indiquer le prédécesseur amenant le minimum (flèche noire)*



Recherche heuristique

- *Exercice : Étape 3 :*
  - *Remonter le chemin de coût minimum en utilisant les états stockés par ville*



Recherche heuristique

- *Exercice : Algorithme*

```
Fonction rechercheChemin ( Graphe : tgraphe, OpS : topS, OpP : topP, NoeudDépart : tnoeud) res : echec ou chemin
  Listenœuds ← insertListe(créerNoeud(NoeudDépart))
Répéter
  Si listeVide(Listenoeuds) alors res ← echec
  Sinon Noeudcourant ← extraireListe(Listenoeuds)
  Fsi
  Si testBut(Noeudcourant)
alors  Noeudcourant.chemin ← Noeudcourant.nom
       Noeudcourant.distance ← 0
       Dmini ← 0
       res ← cheminMinimum(Noeudcourant, Graphe, OpP, NoeudDépart)
  Sinon
    Pour chaque NoeudSuc dans successeur (Graphe, Noeudcourant, OpS)
    insererListe(Listenoeuds, NoeudSuc)
    Fpour
  Fsi
Frépéter
FrechercheChemin
```


Recherche heuristique

- *Exercice : Algorithme*

```
Fonction cheminMinimum(NœudCourant : tnoeud, Graphe : tgraphe, OpP : topP, NoeudDépart : tnoeud) res : CheminMini
  Pour chaque NoeudPred dans prédécesseur(Graphe, OpP, NoeudCourant)
  NoeudPred.distance ← longueur(NoeudCourant, NoeudPred) + NoeudCourant.distance
  NoeudPred.chemin ← NoeudCourant.chemin+ NoeudPred.nom
  Si NoeudPred=NoeudDépart alors
          Si NoeudPred.distance < Dmini ou Si Dmini=0
          alors Dmini←NoeudPred.distance
          CheminMini ← NoeudPred.chemin
          Fsi
  Sinon cheminMinimum(NoeudPred, Graphe, OpP, NoeudDépart)
  Fsi
  Fpour
FcheminMinimum
```

Recherche heuristique

- *Exercice : Algorithme*

  Lexique

  tnoeud : nom : tcaractère
  chemin : tchaine
  distance : tentier

  longueur(Noeud, NoeudPred) : renvoie la distance entre le nœud et un nœud predecesseur

Recherche heuristique

- *Autre version (due à G. Koziel)*

  où la fonction minimum est appliquée sur tous successeurs jusqu'au nœud Solution
  La fonction minimum calcule le plus court chemin pour chaque état en "regardant" le plus court chemin de chaque nœud prédécesseur

Recherche heuristique

- *Algorithme*

```
Fonction rechercheChemin ( Graphe : tgraphe, OpS : topS, OpP : topP, NoeudDépart : tnoeud) res : echec ou chemin
Listenœuds ← initialiserListe(créerNoeud(NoeudDépart))
Répéter
   Si listeVide(Listenoeuds) alors res ← echec
   Sinon Noeudcourant ← extraireListe(Listenoeuds)
   Fsi

   Si testBut(Noeudcourant) alors
res ← Noeudcourant.distance
   Sinon
     Pour chaque NoeudSuc dans successeur (Graphe, Noeudcourant, OpS)
     minimum(NoeudSuc, Graphe, OpP)
     insererListe(Listenoeuds, NoeudSuc)
     Fpour
   Fsi
Frépéter
FrechercheChemin
```


Recherche heuristique

- *Lexique*
  - *tnoeud : nom : tcaractère*
  - *chemin : tchaine*
  - *distance : entier*
  - *longueur(Noeud, NoeudPred) : renvoie la distance entre le nœud et un nœud prédécesseur*
  - *Listedistances : liste de paires (distance : entier, chemin : chaine)*
  - *distanceMin(Listedistances) : renvoie la distance minimale dans Listedistances*
  - *cheminMin(Listedistances) : renvoie le chemin correspondant à la distance minimale dans Listedistances*

Recherche heuristique
Algorithme A*

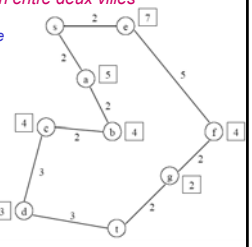
- Algorithme A*
  - Cet algorithme est aussi un cas particulier de l'algorithme de recherche du meilleur d'abord
  - Il combine les avantages des algorithmes de recherche en coût uniforme et de recherche gourmande (Greedy Search), en utilisant une heuristique f()
    f(n) = g(n) + h(n)
  - où
    - n est un nœud représentant un état dans l'espace d'états
    - g(n) = coût du meilleur chemin jusqu'à n
    - h(n) = une fonction heuristique admissible
    - h(n)≤h*(n) où h*(n) est le vrai coût pour aller de n vers un état final
  - Une heuristique est admissible si elle ne surestime jamais le coût pour atteindre le but

Recherche heuristique
Algorithme A*

- *Algorithme A**
  - *Cette heuristique estime le coût total du chemin entre l'état initial et l'état solution qui passe par n : = f(n)*
  - *On remarque qu'il s'agit d'une estimation f(n) obtenue par la somme :*
    - *d'une valeur exacte g(n) issue du chemin parcouru de l'état initial jusqu'au nœud n*
    - *et d'une estimation h(n) du coût du chemin optimal qui lie le nœud n avec le nœud solution*
  - *Pour chaque nœud exploré n, on va étendre le successeur x qui minimise la fonction heuristique f()*
    - *f(x) = min {f(s) / s **dans Successeurs**(n)}*

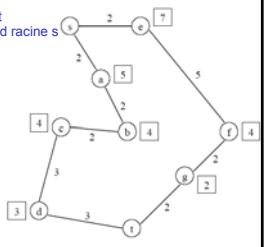
Recherche heuristique
Algorithme A*

- *Exemple 1 : recherche d'un chemin entre deux villes*
  - *Considérons le problème suivant : nous devons nous rendre de la ville s à la ville t en passant par le plus court chemin*
  - *Nombres associés aux divers segments =*
    - *distances physiques séparant les villes adjacentes*
  - *Nombres encadrés =*
    - *estimations h(n) du chemin qu'il reste à parcourir à partir de chaque ville*



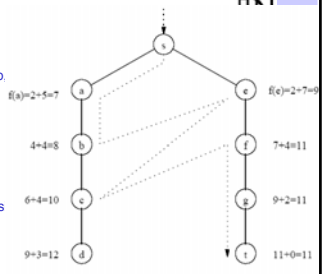
Recherche heuristique
Algorithme A*

- *Développement*
  - Initialement, l'unique chemin pouvant être développé est constitué du nœud racine s
  - Nous avons le choix entre les nœuds a et e
  - On obtient :
    - f(a) = 2 + 5 = 7
    - f(e) = 2 + 7 = 9
  - Nous choisissons le nœud a car
    - f(a) < f(e)
  - L'ensemble des chemins candidats est maintenant composé de
    - [s] et [s; a]



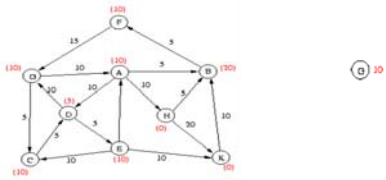
Recherche heuristique
Algorithme A*

- *Arbre de recherche*
  - Nous pouvons maintenant développer soit [s; a] vers b soit [s] vers e
    - f(b) = 4 + 4 = 8
    - f(e) = 2 + 7 = 9
  - C'est b qui a la plus petite valeur, c'est donc vers ce nœud que nous nous dirigeons
  - L'exécution se poursuit ainsi jusqu'a obtention de la solution optimale
    - [s; e; f; g; t]

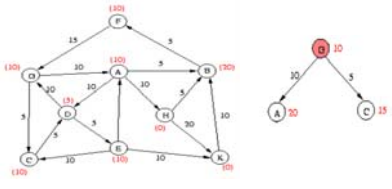


Recherche heuristique
Algorithme A*

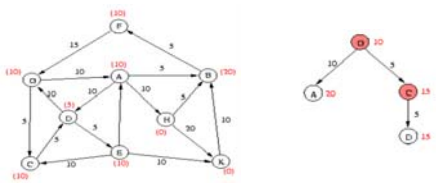
- Exemple 2



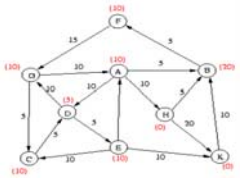
Recherche heuristique
Algorithme A*

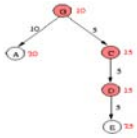


Recherche heuristique
Algorithme A*

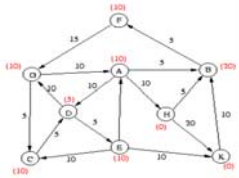


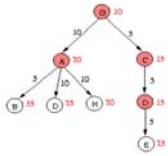
Recherche heuristique
Algorithme A*



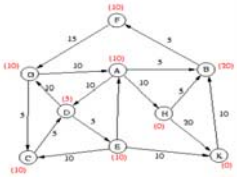


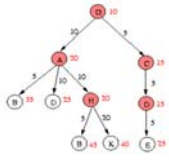
Recherche heuristique
Algorithme A*



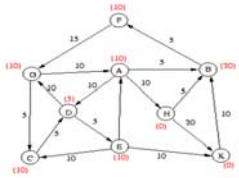


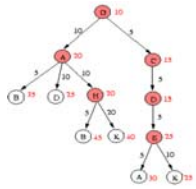
Recherche heuristique
Algorithme A*



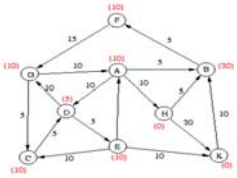


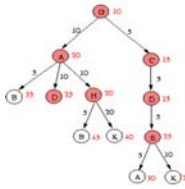
Recherche heuristique
Algorithme A*





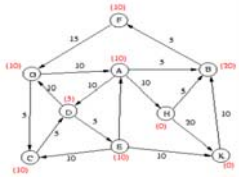
Recherche heuristique
Algorithme A*

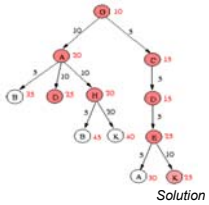




Remarque: on ne fait pas l'expansion de D puisqu'il a déjà été visité

Recherche heuristique
Algorithme A*

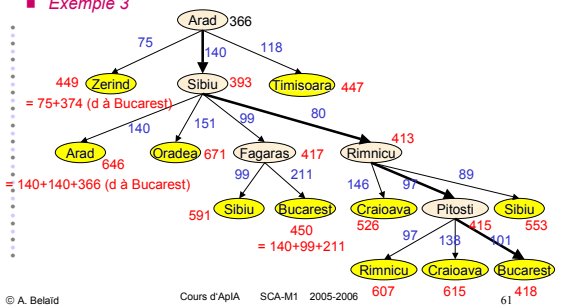




Solution

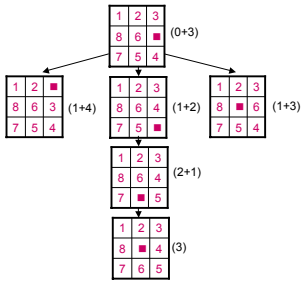
Recherche heuristique
Algorithme A*

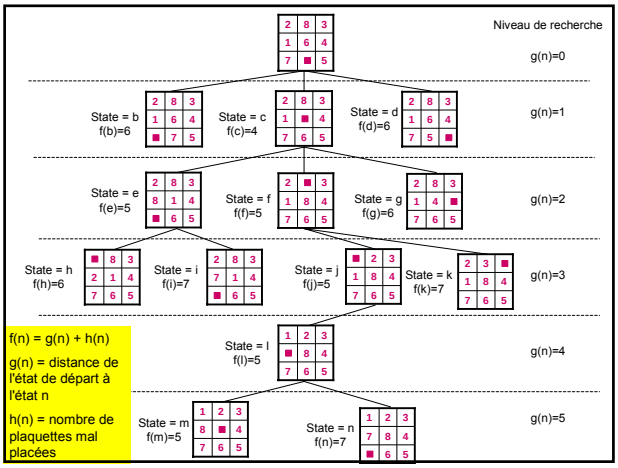
- *Exemple 3*



Recherche heuristique
Algorithme A*

- *Exemple 4 : puzzle-8*
  - *f(n) = g(n) + h(n)*
  - *h(n) = nombre de plaquettes mal placées*
  - *g(n) = longueur du chemin parcouru (+1 à chaque niveau)*





- *Algorithme A* : version sans le chemin*

```
Q ← InsertListe(créerNoeud(État-initial [problème]))
C ← InitListe // sert à repérer les nœuds déjà traités pour soit éviter les cycles ou
     faire des choix quand le nœud est atteint par plusieurs chemins
Répéter
   si listevide(Q) retourne échec
     sinon
        n ← extratListe(Q) //élément le moins coûteux
        si n =(État-final [problème]))
                alors
                   retour avec succès et n
                sinon
                   si nappartientpas (n,C) alors
                                C ← insertListe(C,n)
                                S ← extraSucceurs(n,problème) // liste contenant tous les
                                successeurs de n
                                S ← trierListe(S,f) // trier S par rapport à f(n) = g(n)+h(n)
                                Q ← mergeListe(Q,S,f) // fusionner Q et S en triant Q/f
                   finsi
        Finsi
   Finsi
fin
```


Recherche heuristique
Algorithme A*

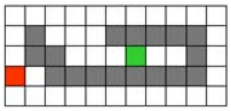
- *Lexique*
  - *Q : liste des nœuds en attente de traitement, triés par f croissant, extratListe prendra à chaque fois le nœud le moins cher par rapport à f*
  - *C : liste des nœuds traités*
  - *n : nœud courant*
  - *nappartientpas : fonction qui vérifie que n ∉ C ou a un coût inférieur au même n∈C (atteint par un autre chemin)*
  - *extra*Succeurs : fonction qui extrait du graphe d'état tous les successeurs d'un nœud
  - *trierListe(S,f) : trie S par rapport à f(n) = g(n)+h(n)*
  - *mergeListe(Q,S,f) : fusionne Q et S en triant Q/f*

- *Algorithme A* : version avec le chemin*

```
// idée : disposer dans C des états et des chemins y conduisant
Nœud ← créerNoeud(État-initial [problème])
Chemin ←Noeud
Q ← InsertListe((Nœud, chemin))
C ← InitListe // sert à repérer les nœuds déjà traités pour soit éviter les cycles ou
     faire des choix quand le nœud est atteint par plusieurs chemins
Répéter
   si listevide(Q) retourne échec
     sinon
        n,chemin ← extratListe(Q) //élément le moins coûteux
        Q ← reste(Q)
        si n =(État-final [problème]))
                alors
                   retourner n,chemin
                sinon
                   si nappartientpas (n,C) alors
                                C ← insertListe(C,(n,chemin))
                                S ← extraSucceurs(n,problème) // liste contenant tous les
                                successeurs de n. Chaque nœud est accompagné de son
                                chemin = celui de n + lui-même
                                S ← trierListe(S,f) // trier S par rapport à f(n) = g(n)+h(n)
                                Q ← mergeListe(Q,S,f) // fusionner Q et S en triant Q/f
                   finsi finsi finsi
   Finsi
fin
```

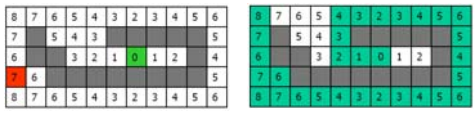
Recherche heuristique
Algorithme A*

- *Exemple5 : navigation robotique*
  - *Conduire le robot de la case de départ (verte) à la case d'arrivée (rouge) en utilisant uniquement les cases blanches et en se déplaçant de case en case*



Recherche heuristique
Algorithme A*

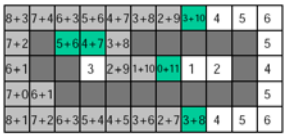
- *Premier cas :*
  - *f(n) = h(n), avec h(n) = distance de Manhattan = somme des déplacements unitaires verticaux et horizontaux*
    - *Si on s'éloigne de l'origine, on ajoute 1*
    - *Si on se rapproche de l'origine, on enlève 1*



Chemins possibles Chemin retenu

Recherche heuristique
Algorithme A*

- *Deuxième cas :*
  - f(n) = g(n)+h(n), avec h(n) = distance de Manhattan
  - Ex : 0 + 11 : 11=distance de l'origine à l'arrivée

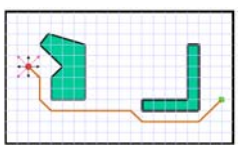


Recherche heuristique
Algorithme A*

- *Complétude et optimalité*
  - *Affirmation*
    - S'il existe un chemin allant de l'état initial au but, A* (sans élimination des états multiples) se termine et trouve le meilleur chemin
  - Donc A* a ces particularités :
    - Complétude : oui sauf s'il y a un nombre infini de nœuds avec une valeur de $f \leq f(G)$, G : *but*
    - Complexité en temps : *exponentielle*
    - Complexité en espace : *garde tous les nœuds en mémoire*
    - Optimalité : Oui

Recherche heuristique
Algorithme A*

- *Exercice*
  - *Écrire l'algorithme puis le programme permettant de faire passer le robot de la position verte à la position rouge, en utilisant :*
  - *f(n)=g(n)+h(n) où h(n)= distance à vol d'oiseau*
  - *Paramétrez votre programme par :*
    - *la position des obstacles et les positions de début et fin du robot*



Coût d'un pas horizontal/vertical = 1

Coût d'un pas en diagonale = $\sqrt{2}$

Recherche heuristique
Variantes de A*

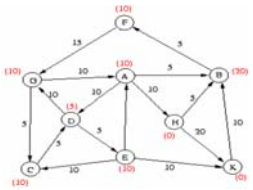
- *Les problèmes réels sont souvent très complexes*
  - *L'espace de recherche devient très grand*
  - *Même les méthodes de recherche heuristiques deviennent inefficaces*
    - *A* connaît alors des problèmes de place-mémoire*
  - *D'où la recherche d'algorithmes "économes" en place-mémoire*
- *2 variantes de A* :*
  - *IDA* = A* avec approfondissement itératif*
  - *SMA* = A* avec gestion de mémoire*

Recherche heuristique
Meilleur d'abord incrémental IDA*

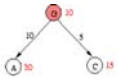
- Rappel:
  - IDS effectue l'approfondissement par rapport à la profondeur limite de recherche (L = 0, 1, 2, 3, ….)
- IDA*
  - Quant à lui, utilise comme limite de l'horizon de recherche la valeur de la fonction heuristique $f(n)$
  - Chaque itération de IDA* est une recherche en profondeur qui étend tous les nœuds à l'intérieur d'un contour délimité par la valeur courante de f_{limite} en regardant par dessus cette limite pour savoir où se trouvera la prochaine valeur de f_{limite}
- IDA* est :
  - complet et optimal
  - complexité en place de IDA*: $O(bd)$

Recherche heuristique
Meilleur d'abord incrémental IDA*

- Exemple



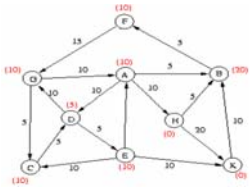
Première itération, limite = 10



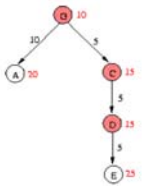
f(A) = g(A)+h(A) =
somme des arcs menant à A +10

Recherche heuristique
Meilleur d'abord incrémental IDA*

- Exemple

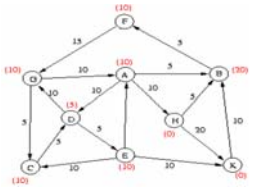


Deuxième itération, limite = 15

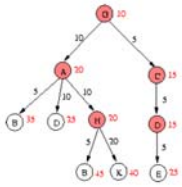


Recherche heuristique
Meilleur d'abord incrémental IDA*

- Exemple

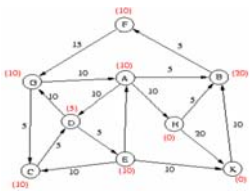


Troisième itération, limite = 20

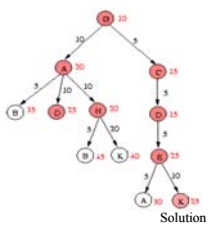


Recherche heuristique
Meilleur d'abord incrémental IDA*

- Exemple



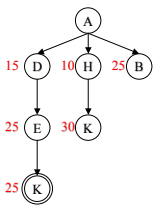
Quatrième itération, limite = 25



Solution

Recherche heuristique
Variantes de A*

En partant de A :



Recherche heuristique
Variante de IDA*

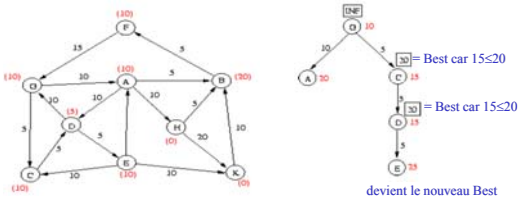
- Meilleur d'abord récursif
  - On fait un parcours en profondeur récursif
    - Lorsqu'on fait l'expansion d'un nœud n, on identifie la valeur best = f(n) du meilleur nœud parmi les autres nœuds ouverts
      - Pour ce faire, on identifie la plus petite valeur f(n) parmi ses nœuds frères
      - Si cette valeur est plus petite que la valeur best associée au parent de n, c'est elle qu'on choisit, sinon on conserve la valeur best du parent

Recherche heuristique
Variantes de A*

- Meilleur d'abord récursif (suite)
  - Lorsqu'on arrive à une situation où tous les nœuds successeurs de *n* ont une valeur *f(n)* supérieure à celle de la valeur *best* qui lui est associée, on remonte
    - En remontant, on met à jour la valeur *f(n)* en la remplaçant par celle de son meilleur fils
    - Si un des frères de *n* a une valeur *f(n)* plus petite que la valeur *best* du parent de *n*, on étend ce nœud, sinon on remonte

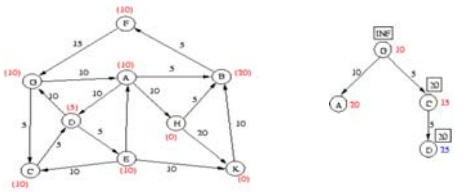
Recherche heuristique
Meilleur d'abord récursif

- Exemple



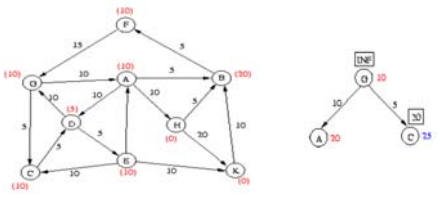
Recherche heuristique
Meilleur d'abord récursif

- Exemple



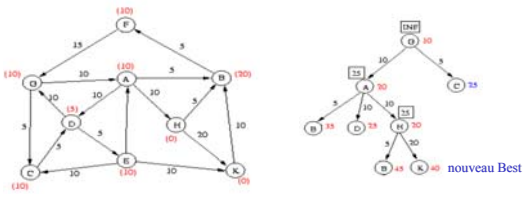
Recherche heuristique
Meilleur d'abord récursif

- Exemple



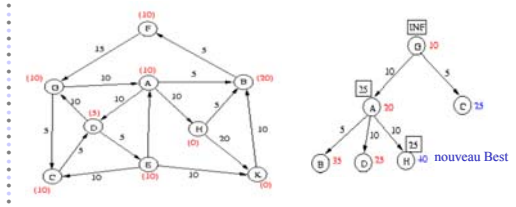
Recherche heuristique
Meilleur d'abord récursif

- Exemple



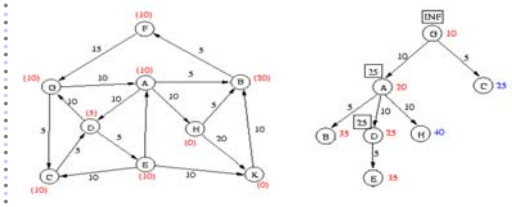
Recherche heuristique
Meilleur d'abord récursif

- Exemple



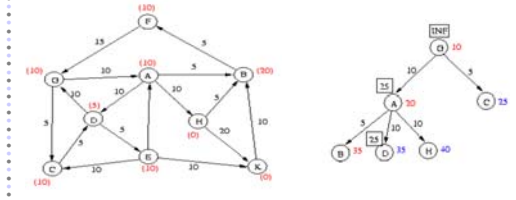
Recherche heuristique
Meilleur d'abord récursif

- Exemple



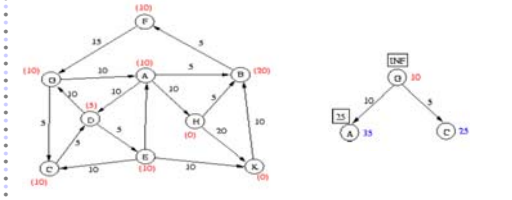
Recherche heuristique
Meilleur d'abord récursif

- Exemple



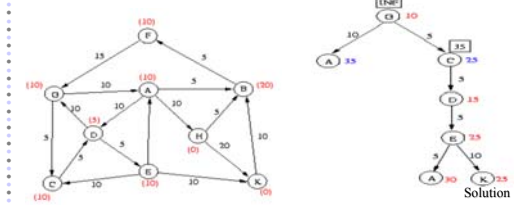
Recherche heuristique
Meilleur d'abord récursif

- Exemple



Recherche heuristique
Meilleur d'abord récursif

- Exemple

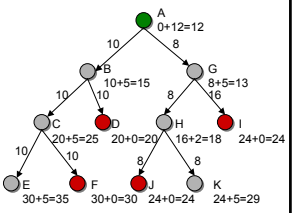


Recherche heuristique
SMA* (Simplified Memory-Bounded A)

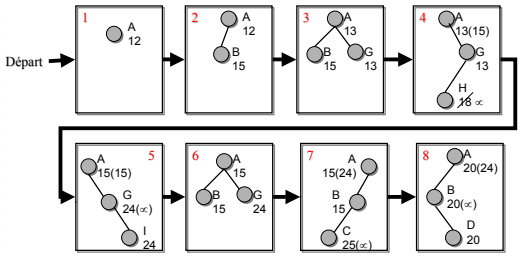
- *SMA* : A* avec gestion de la mémoire*
  - SMA* effectue sa propre gestion de la mémoire
- Principe
  - si la mémoire (file d'attente) est pleine, alors faire de la place en éliminant le nœud le moins intéressant (celui avec une valeur *f* élevée)
  - retenir dans le nœud-ancêtre la valeur du meilleur descendant oublié
    - il évolue dans un espace-mémoire alloué par avance
    - il évite les états dupliqués
    - il est complet si l'espace-mémoire alloué est suffisant pour contenir le chemin-solution le moins profond
    - il est optimal si l'espace-mémoire alloué est suffisant pour contenir le chemin-solution optimal le moins profond
  - dans tous les cas il retourne la meilleure solution accessible étant donné l'espace-mémoire alloué

Recherche heuristique
SMA*

- *Exemple*
  - Chaque nœud est étiqueté avec sa valeur de f=g+h
  - les nœuds-solutions sont D, F, I, J
  - Objectif : trouver le nœud-solution de moindre coût avec un espace-mémoire capable de contenir au maximum 3 nœuds



Recherche heuristique
SMA*



Recherche Locale

- Idée de cette recherche
  - Souvent, le chemin qui mène vers une solution n'est pas important
  - L'état lui-même est la solution
    - Utile pour des problèmes d'optimisation
  - Idée :
    - Modifier l'état en l'améliorant au fur et à mesure

Recherche Locale

- Convient surtout aux problèmes où seul l'état final nous intéresse
  - On ne conserve qu'un seul état en mémoire
    - ➔soit le nœud courant
  - On considère tous les nœuds successeurs du nœud courant qui constituent le *voisinage*
  - On choisit un de ces nœuds qui deviendra le nœud courant
  - On répète le processus jusqu'à l'obtention de l'état désiré

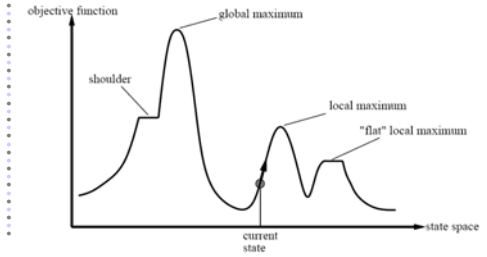
Recherche locale

- Les algorithmes dédiés
  - Ascension/descente de gradient (hill climbing)
  - Descente de gradient stochastique
  - Recuit simulé (Simulated annealing)
  - Recherche en faisceau (Beam search)
  - Recherche en faisceau stochastique
  - Algorithmes génétiques

L'algorithme Hill Climbing

- Principe
  - L'algorithme de recherche "Hill Climbing" se base sur le principe de choisir le nœud suivant comme étant le nœud le plus proche de la solution
  - L'idée est relativement triviale et peut être illustrée par le fait de choisir le chemin le plus court pour gravir une colline
  - En contrepartie, cette technique comprend d'importants défauts qu'il ne faut pas manquer de prendre en considération
  - Par exemple, l'algorithme échoue dans le cas où il existe des maxima locaux dans l'espace de recherche, mais qui ne font pas partie de la solution
  - C'est pourquoi l'utilisation de cet algorithme est restreinte à un petit nombre de problèmes et applications

L'algorithme Hill Climbing



Descente de gradient

- Problème
  - On est coincé à un moment de la résolution dans un minimum local (situation locale)
  - On cherche à s'en sortir en donnant un coup de pousse
- Idée
  - On utilise une fonction *h(n)* qu'il faut minimiser
    - On calcule *h(n')* pour tous les nœuds *n'* successeurs du nœud courant *n*
    - On choisit le nœud *n'* qui minimise *h(n')* et tel que *h(n')<h(n)*

Descente du gradient

- Algorithme

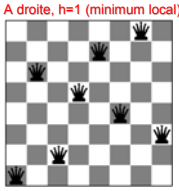
function Descente-Gradient(*problème*) **retourne** un état solution
  Courant ←créer-nœud(état-initial[problème])
  Répéter
    suivant ← valeur-la-plus-basse (suivant(courant))
    si valeur(suivant)>valeur(courant) **retourne** courant
    sinon courant ← suivant
  fin

Descente du gradient

- Exemple : problème des 8-reines
  - On choisit comme coût heuristique h le nombre de paires de reines qui s'attaquent

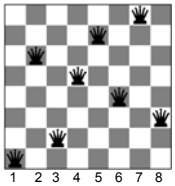
A gauche, h=17

A droite, h=1 (minimum local)



Descente du gradient

- Explications
  - Ici, seulement 2 reines sont en conflit : 4 et 7

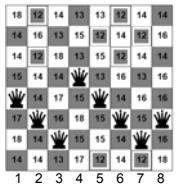


Descente du gradient

- Explications
  - Ici, il y a 17 cas de prise

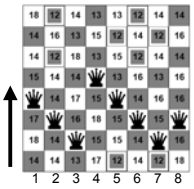
1.	1-2	10.	4-5
2.	1-3	11.	4-6
3.	1-5	12.	4-7
4.	2-3	13.	5-6
5.	2-4	14.	5-7
6.	2-6	15.	6-7
7.	2-8	16.	6-8
8.	3-5	17.	7-8
9.	3-7		

Les chiffres dans chaque case indiquent le nombre de cas de conflits créé en mettant la reine de la colonne sur cette case



Descente du gradient

- Méthode
  - Faire avancer la reine, le long de sa colonne, dont le nombre de paires de conflit est le plus petit pour toutes les reines
  - Dans la configuration ci-contre, on a le choix entre 2 possibilités : avec h=13 concernant les reines 4 et 5
  - On calcule après déplacement les nombres de conflits autour de la reine déplacée
  - Si le nombre de conflits total ne baisse pas autour de la reine, on revient en arrière pour choisir un autre minimum



Descente du gradient

- Exercice
  - Proposez une structure de données pour représenter l'espace
  - Proposez l'algorithme de résolution complet