Stratégies d'exploration informées

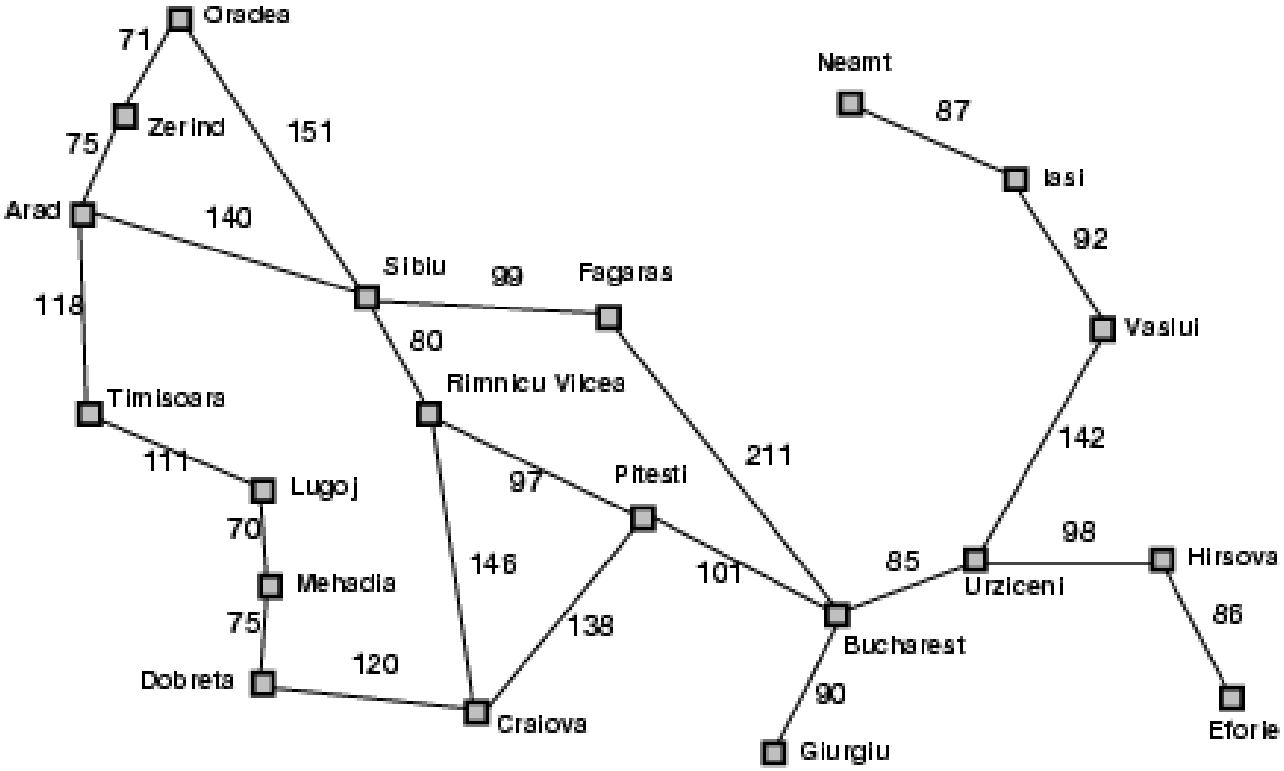
- Stratégies d'exploration non informées
  - Ne sont pas très efficaces dans la plupart des cas.
  - Elles ne savent pas si elles approchent du but.
- Stratégies d'exploration informées
  - Elles utilisent une fonction d'estimation
    - Fonction heuristique.
    - Pour choisir les nœuds à visiter.

Stratégies d'exploration informée

- Meilleur d'abord (BFS - *Best-first*)
- Meilleur d'abord gloutonne (*Greedy best-first*)
- A* (*A-Star*)
- Algorithmes heuristiques à mémoire limitée
  - IDA*, RDFS et SMA*
- Par escalade (*Hill-climbing*)
- Par recuit simulé (*Simulated annealing*)
- Exploration locale en faisceau (*Local beam*)
- Algorithmes génétiques

Exemple d'exploration:

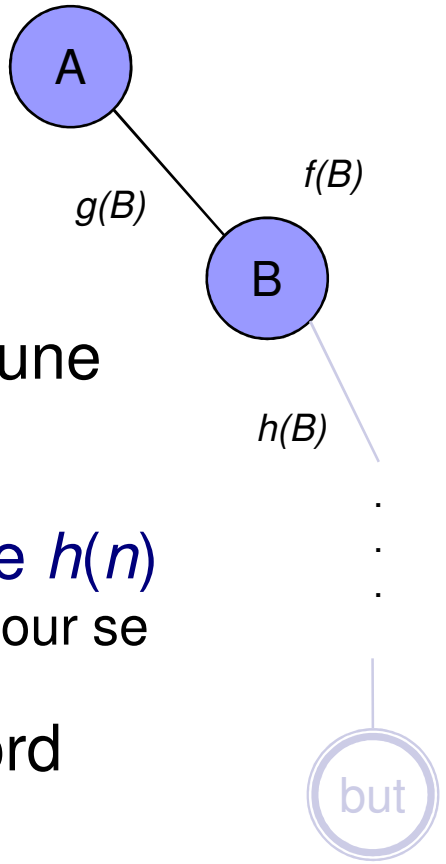
Voyage en Roumanie (avec coûts en km)



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

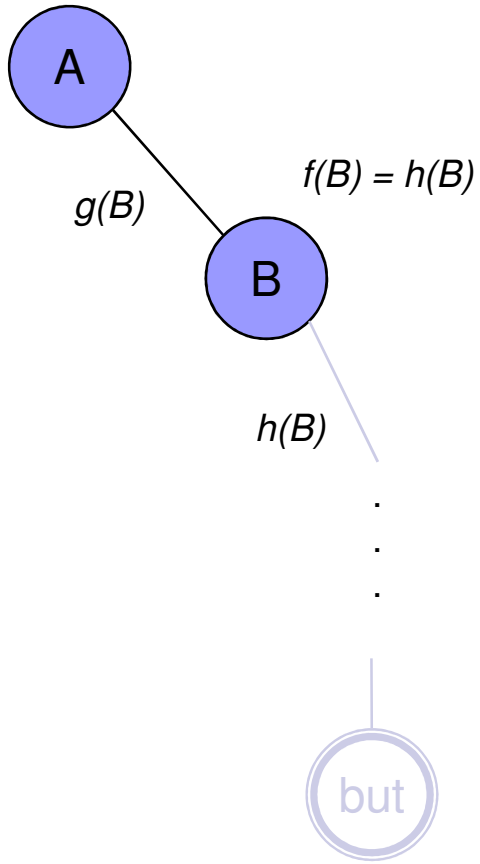
Meilleur d'abord

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

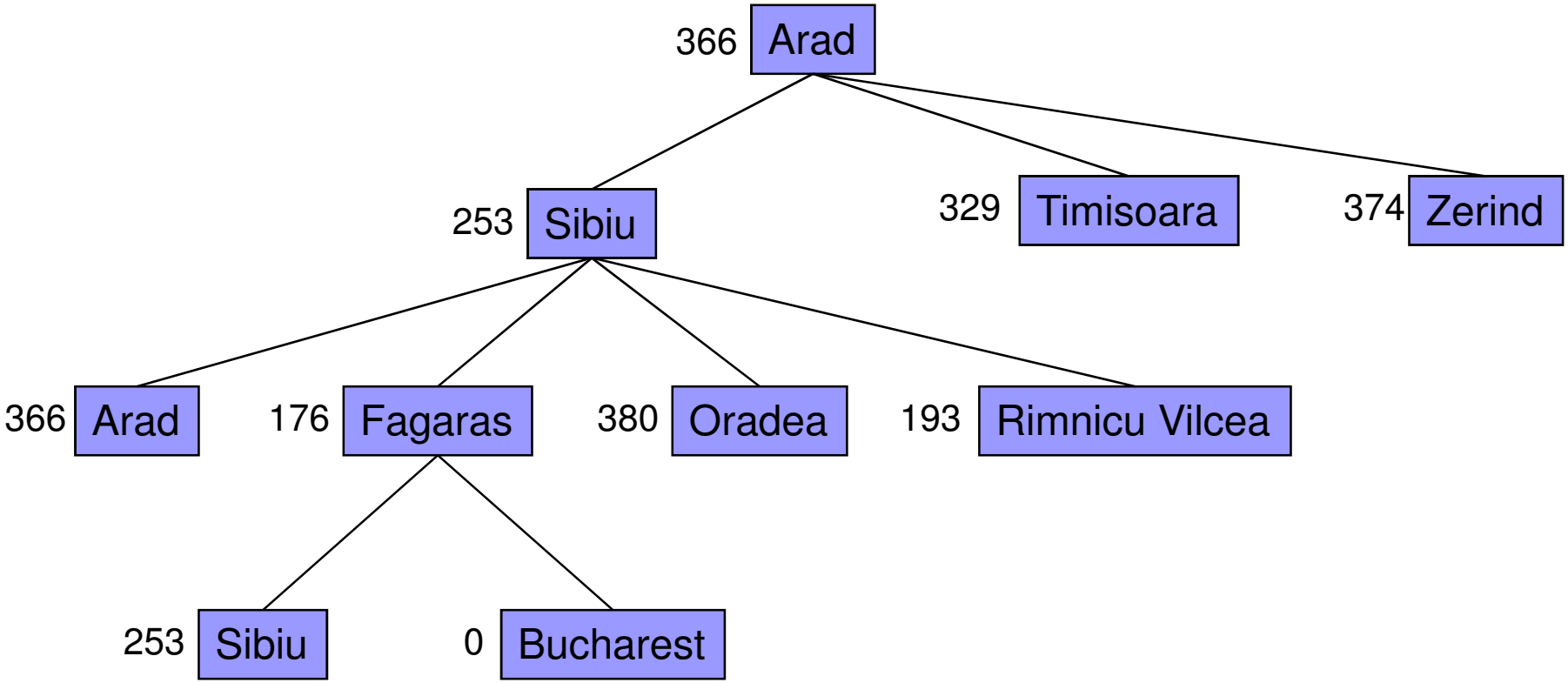


Meilleur d'abord gloutonne

- $f(n) = h(n)$
- Donc on choisit toujours de développer le nœud le plus proche du but.



Exemple meilleur d'abord gloutonne



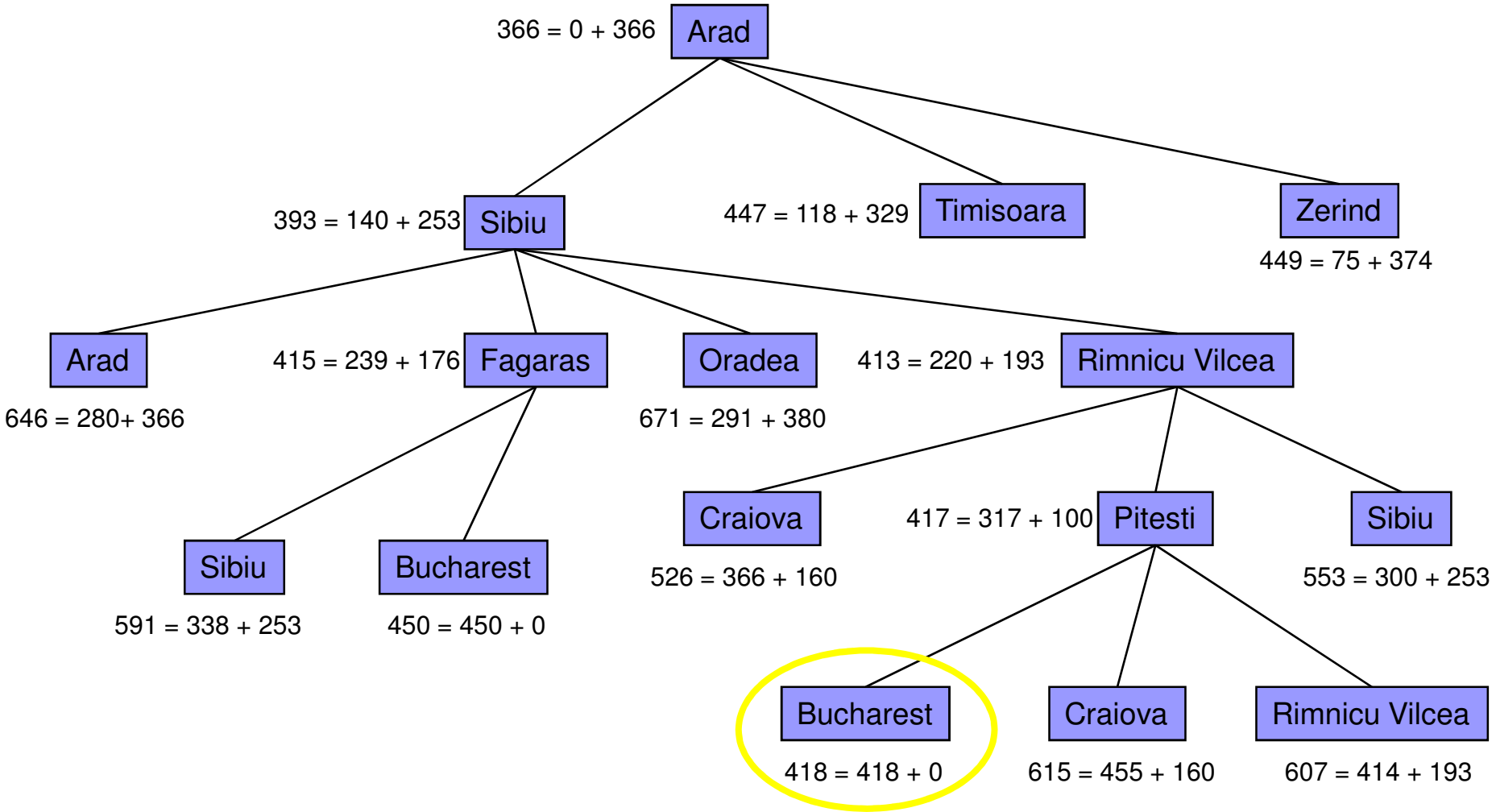
Propriétés - Meilleur d'abord gloutonne

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

A* (*A-star*)

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

Exemple A*



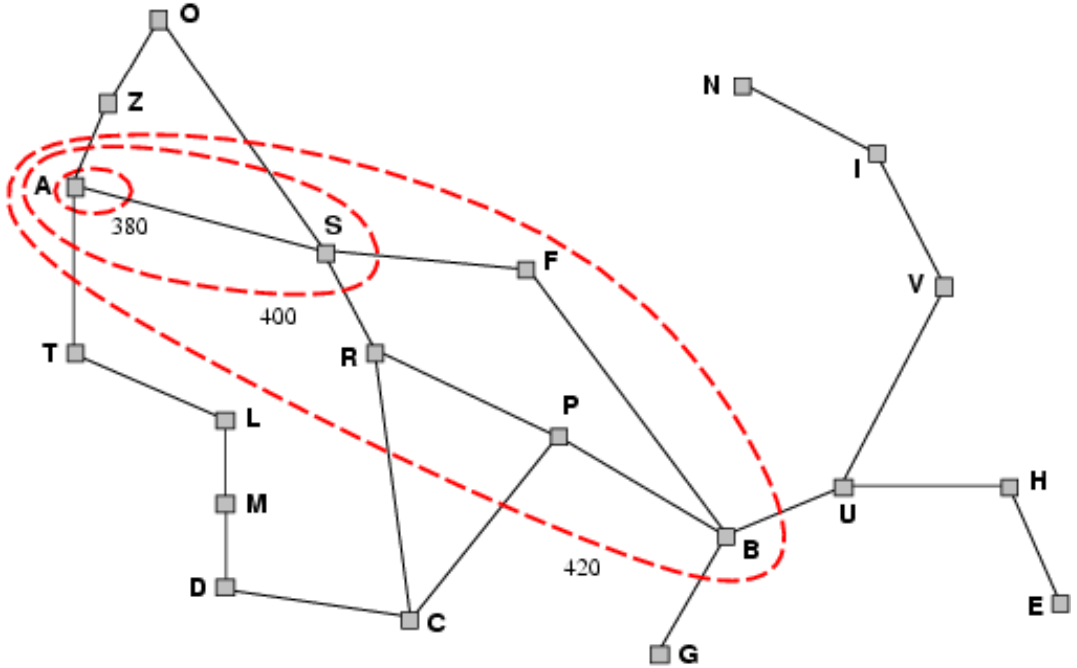
But atteint, la recherche arrête.

Propriétés de A*

- Complétude : Oui
  - À moins qu'il y est une infinité de nœuds avec $f \leq f(but)$.
- Complexité de temps : Exponentielle
  - Selon la longueur de la solution.
- Complexité en espace : Exponentielle
  - Selon la longueur de la solution.
  - Elle garde tous les nœuds en mémoire.
- Optimale : Oui
  - Habituellement, on manque d'espace longtemps avant de manquer de temps.

Optimalité de A*

- A* développe les noeuds en ordre croissant de valeur *f*
- Ajoute graduellement des “f-contours” aux noeuds
- Le contour *i* a tous les noeuds avec $f=f_i$, tel que $f_i < f_{i+1}$



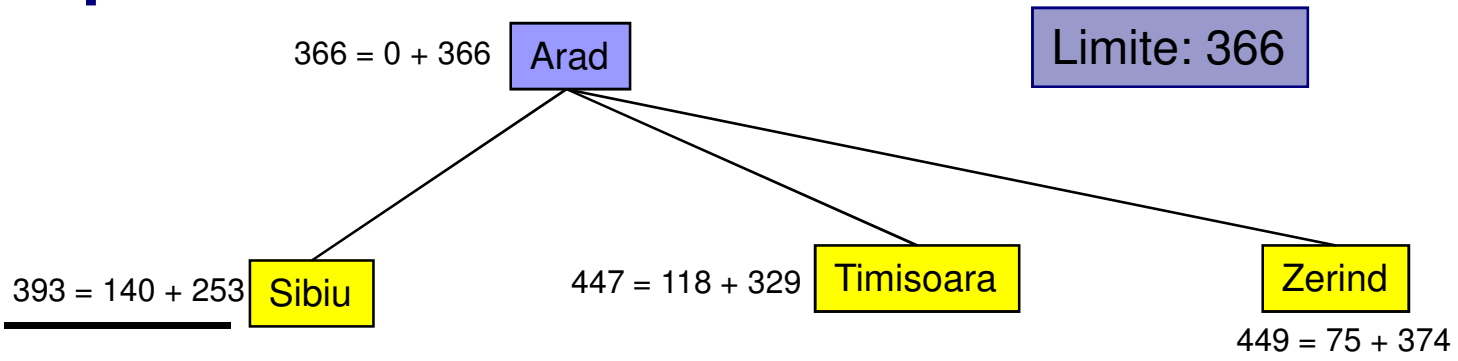
Exploration heuristique à mémoire limitée

- A* est parfois trop gourmand en mémoire.
- Il existe des algorithmes pour surmonter ce problème dont:
  - IDA*;
  - RBFS;
  - SMA*.
- Ces algorithmes permettent de préserver l'optimalité et la complétude.
- L'augmentation du temps d'exécution est raisonnable.

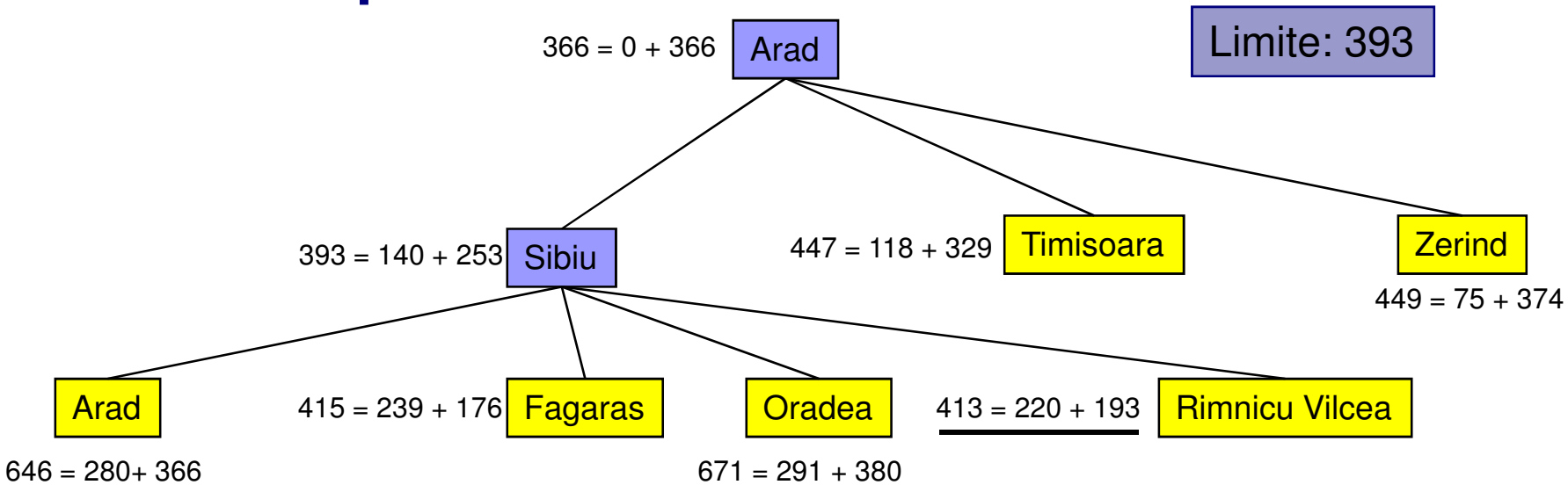
Exploration heuristique à mémoire limitée

- IDA* (*Iterative-deepening A**)
  - C'est un algorithme de profondeur itérative
  - Utilise la valeur *f(n)* comme limite
    - Contrairement à la profondeur pour IDS.
  - À chaque itération :
    - On fixe la limite à la plus petite valeur *f(n)* de tous les nœuds qui avaient une valeur plus grande que la limite au tour précédent.

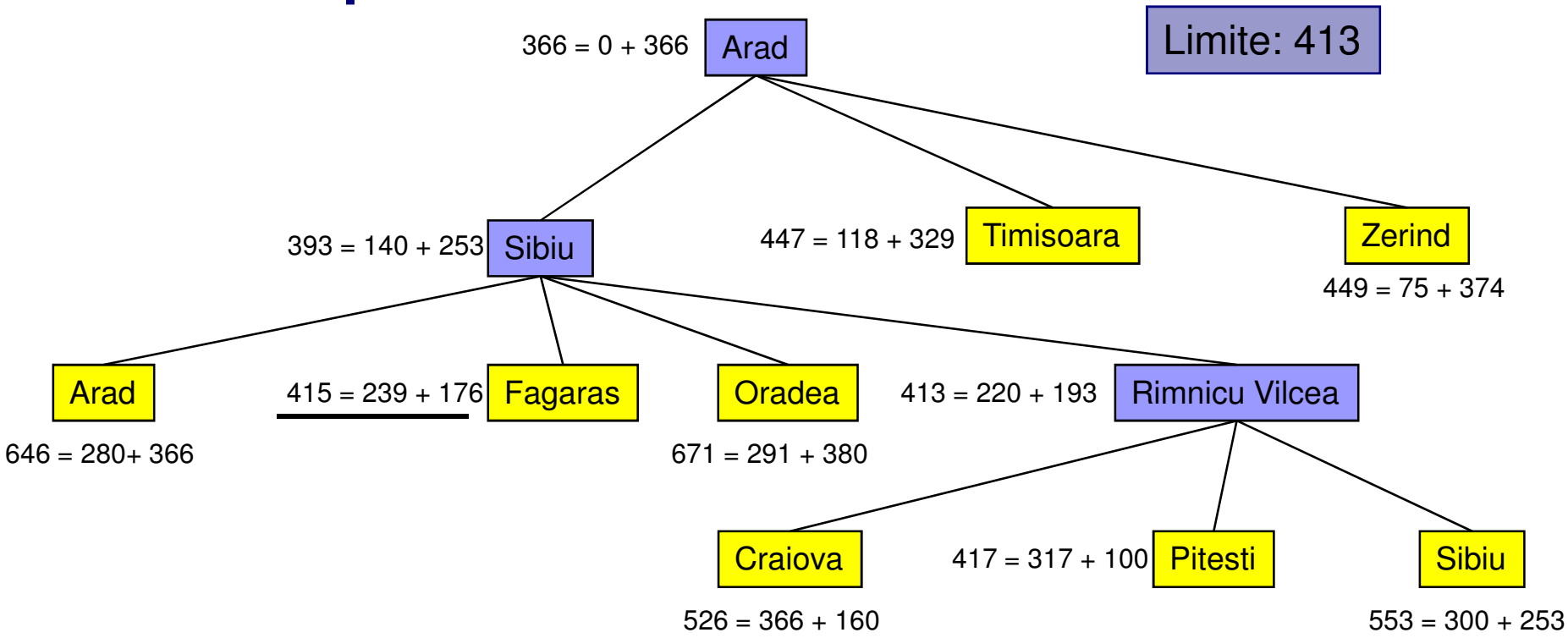
Exemple IDA*


366 = 0 + 366
Arad
Limite: 366
393 = 140 + 253
Sibiu
447 = 118 + 329
Timisoara
Zerind
449 = 75 + 374

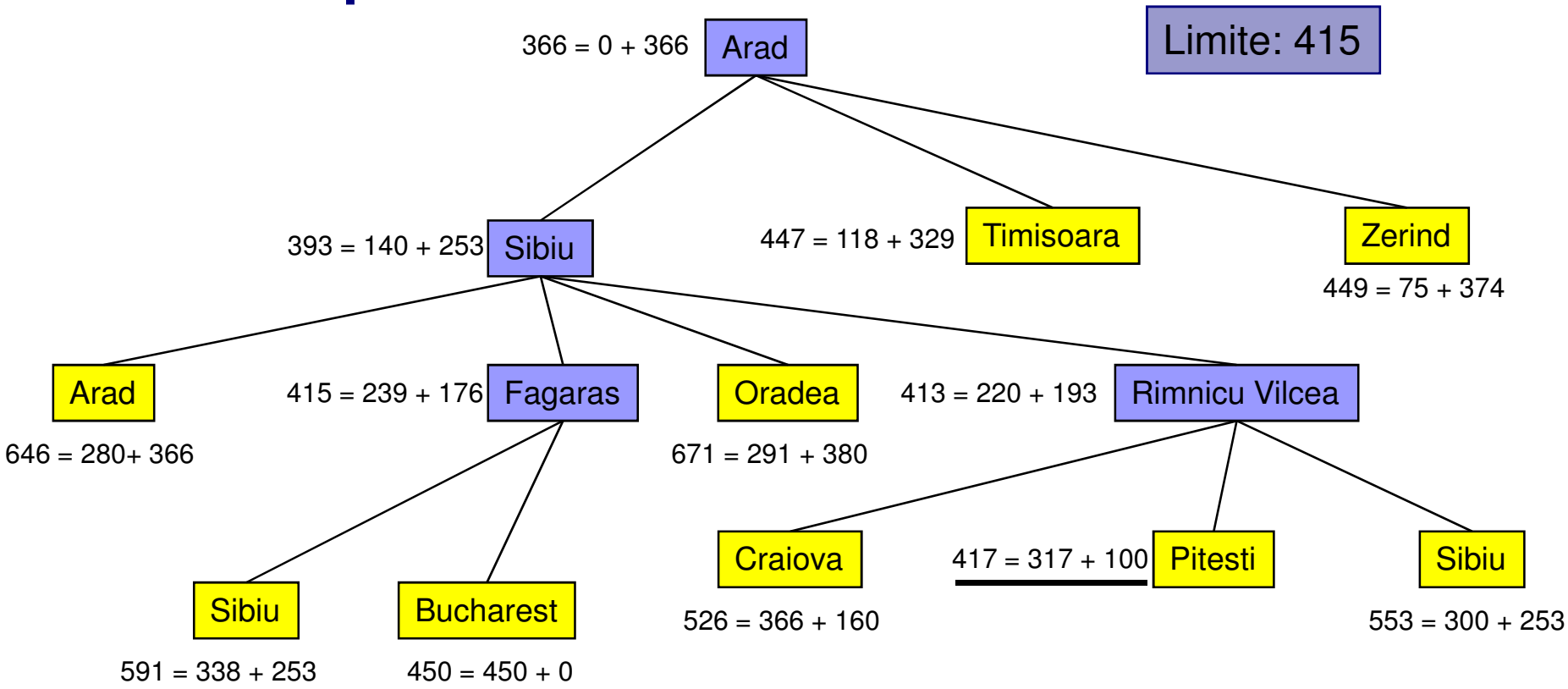
Exemple IDA*


366 = 0 + 366
Arad
Limite: 393
393 = 140 + 253
Sibiu
447 = 118 + 329
Timisoara
Zerind
449 = 75 + 374
Arad
646 = 280+ 366
415 = 239 + 176
Fagaras
Oradea
671 = 291 + 380
413 = 220 + 193
Rimnicu Vilcea

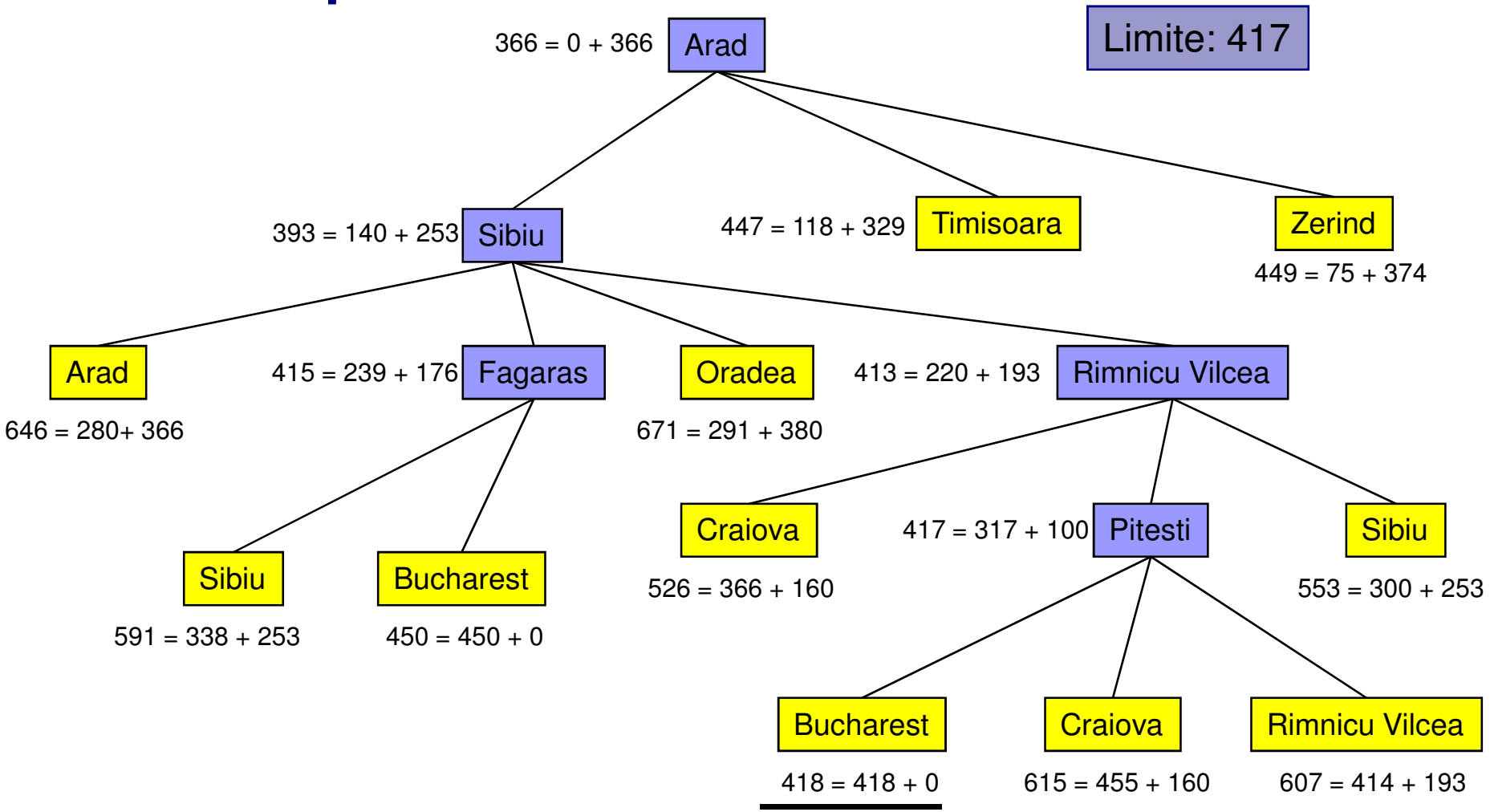
Exemple IDA*


366 = 0 + 366
Arad
Limite: 413
393 = 140 + 253
Sibiu
447 = 118 + 329
Timisoara
Zerind
449 = 75 + 374
Arad
646 = 280+ 366
415 = 239 + 176
Fagaras
Oradea
671 = 291 + 380
413 = 220 + 193
Rimnicu Vilcea
Craiova
526 = 366 + 160
417 = 317 + 100
Pitesti
Sibiu
553 = 300 + 253

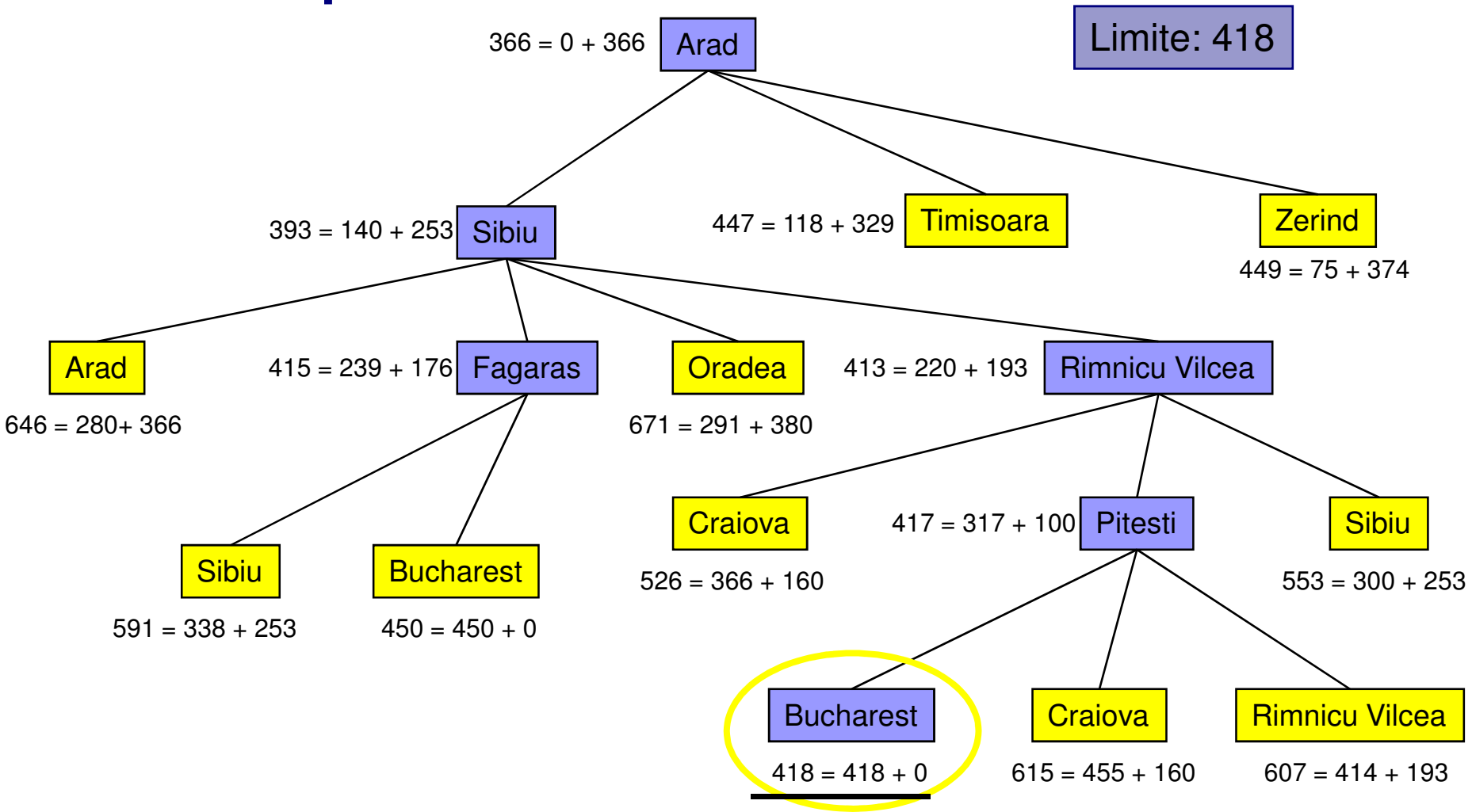
Exemple IDA*


366 = 0 + 366
Arad
Limite: 415
393 = 140 + 253
Sibiu
447 = 118 + 329
Timisoara
Zerind
449 = 75 + 374
Arad
646 = 280+ 366
415 = 239 + 176
Fagaras
Oradea
671 = 291 + 380
413 = 220 + 193
Rimnicu Vilcea
Sibiu
591 = 338 + 253
Bucharest
450 = 450 + 0
Craiova
526 = 366 + 160
417 = 317 + 100
Pitesti
Sibiu
553 = 300 + 253

Exemple IDA*


Limite: 417
366 = 0 + 366
Arad
393 = 140 + 253
Sibiu
447 = 118 + 329
Timisoara
Zerind
449 = 75 + 374
Arad
646 = 280+ 366
415 = 239 + 176
Fagaras
Oradea
671 = 291 + 380
413 = 220 + 193
Rimnicu Vilcea
Sibiu
591 = 338 + 253
Bucharest
450 = 450 + 0
Craiova
526 = 366 + 160
417 = 317 + 100
Pitesti
Sibiu
553 = 300 + 253
Bucharest
418 = 418 + 0
Craiova
615 = 455 + 160
Rimnicu Vilcea
607 = 414 + 193

Exemple IDA*

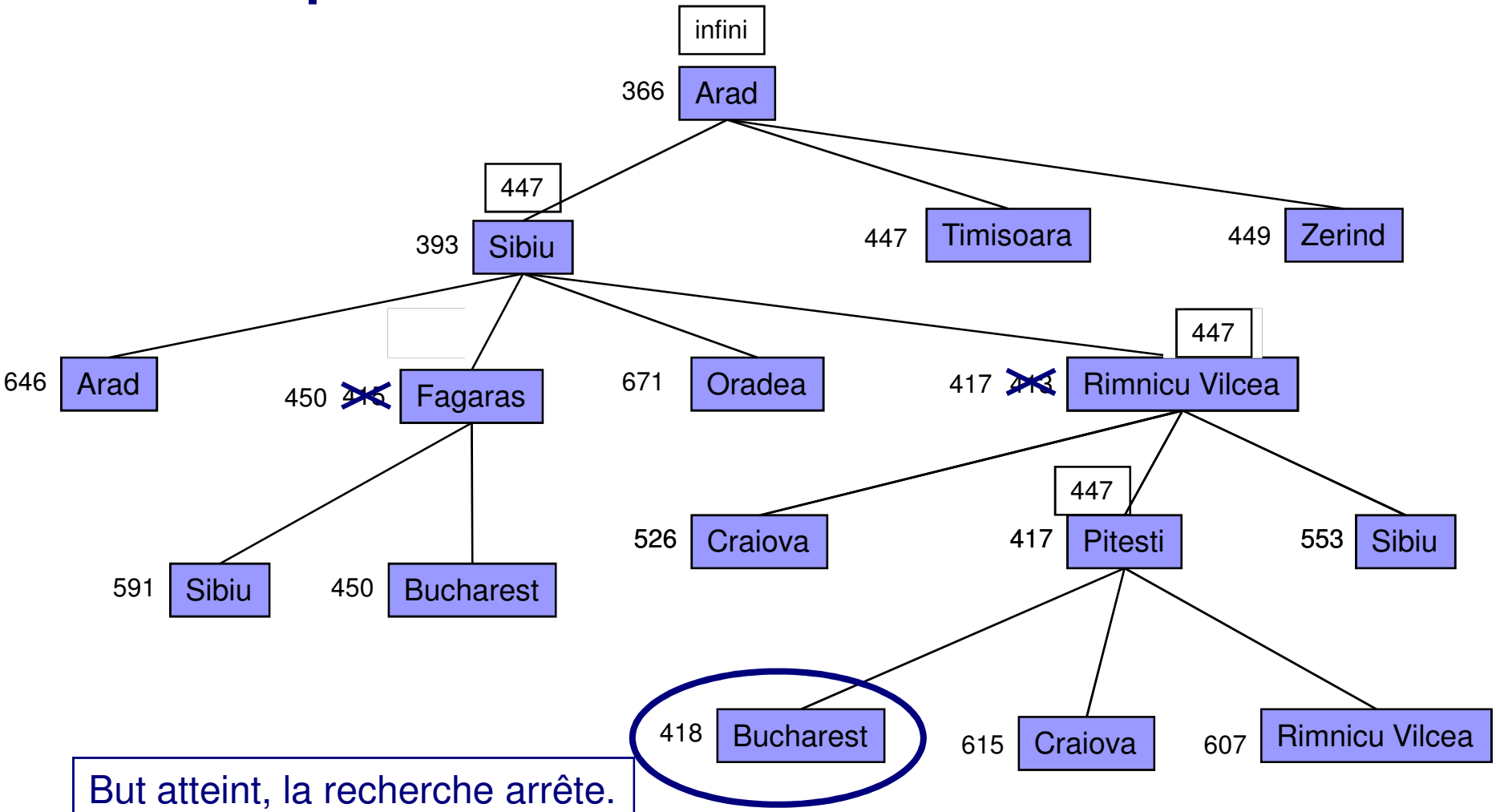


But atteint, la recherche arrête.

Exploration heuristique à mémoire limitée

- RBFS (*Recursive best-first search*)
  - Ressemble au meilleur d'abord.
  - Idée:
    - Conserve en mémoire la valeur f(n) de la meilleure alternative d'un des ancêtres.
    - Si le nœud courant excède cette valeur :
      - on garde en mémoire la valeur f(n) du meilleur descendant
      - on essaie une autre branche.
  - Optimal : comme A* si *f(n)* est admissible.
  - Complexité d'espace : linéaire -- *O*(*bd*).
  - Plus efficace que IDA*.

Exemple RBFS



Algorithme RBFS

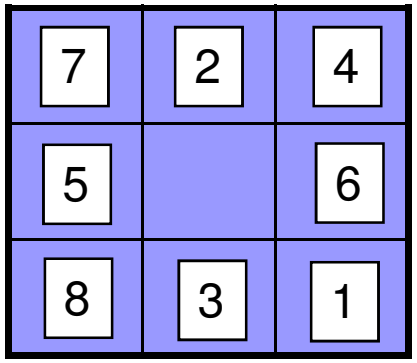
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if GOAL-TEST[problem](state) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) + h(s), f[node])
    repeat
        best ← the lowest f-value node in successors
        if f[best] > f_limit then return failure, f[best]
        alternative ← the second-lowest f-value among successors
        result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```

Recherche à mémoire limitée

- SMA* (*Simplified memory bounded A**)
  - Exactement comme A*, mais avec une limite sur la mémoire.
  - S'il doit développer un nœud et que la mémoire est pleine, il enlève le plus mauvais nœud et comme RBFS, il enregistre au niveau du père la valeur du meilleur chemin.
  - Complétude : Oui
    - S'il y a une solution atteignable
  - Optimale : Oui
    - Si la solution optimale est atteignable
  - Complexité en temps : Peut être très long
    - S'il doit souvent regénérer des nœuds.
  - Complexité en espace : C'est la mémoire allouée.

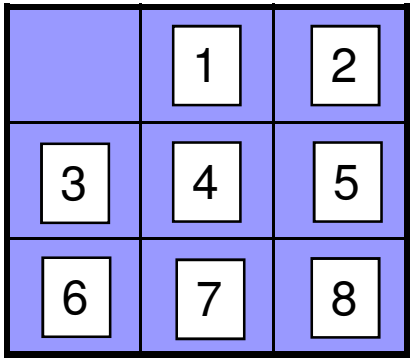
Fonctions heuristiques

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



État initial

- $h_1(n) = 8$
- $h_2(n) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2$
  $= 18$



État but

Facteur de branchement effectif

- Si *N* nœuds traités avec une solution de profondeur *d*
  - Alors le facteur de branchement effectif (b^*) est le facteur de branchement d'un arbre uniforme de profondeur *d* contenant *N* nœuds.
- $N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$
- Plus la valeur de b* s'approche de 1, plus l'heuristique est efficace.
- Pour déduire la valeur de b*, il suffit de faire des tests de complexité variable et de calculer la valeur moyenne.
- b* donne une bonne idée de l'utilité de l'heuristique et permet aussi de comparer des heuristiques.

Comparaison

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

Dominance

- Si $h_2(n) \geq h_1(n)$ pour tout n (les deux étant admissibles)
  - Alors $h_2(n)$ domine $h_1(n)$
  - Et par conséquent $h_2(n)$ est meilleure que $h_1(n)$.
- Il est toujours préférable de choisir l'heuristique dominante, car elle va développer moins de noeuds
  - Tous les nœuds avec f(n) < C* vont être développé.
  - Donc, tous les nœuds avec h(n) < C* - g(n) vont être développé.
  - Par conséquent, tous les nœuds développé par $h_2(n)$ vont aussi être développé par $h_1(n)$.
  - Et $h_1(n)$ peut aussi développé d'autres nœuds.

Inventer une heuristique

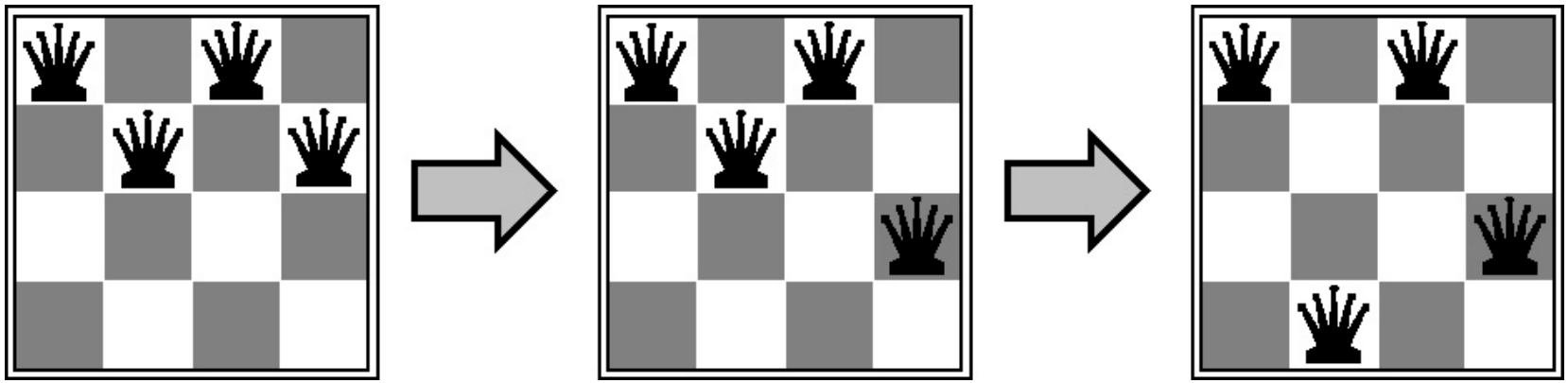
- Simplifier le problème
- La solution exacte du problème simplifié peut être utilisée comme heuristique.
- $h_1(n)$ est exacte, si une tuile peut être déplacée à tout endroit.
- $h_2(n)$ est exacte, si une tuile peut être déplacée dans toute cellule adjacente.
- Le coût de la solution optimale pour le problème simplifié n'est pas plus grand que le coût de la solution optimale du vrai problème.

Algorithmes d'amélioration itérative

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

Exemple

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



Algorithmes d'amélioration itérative

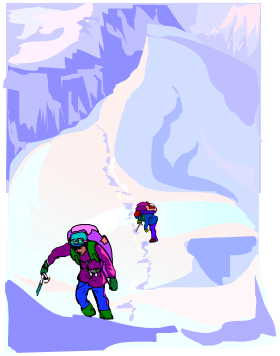
- Par escalade (*Hill-climbing*)
- Par recuit simulé (*Simulated annealing*)
- Exploration locale en faisceau (*Local beam*)
- Algorithmes génétiques

Algorithmes d'amélioration itérative

Par escalade (*Hill-climbing*)

- À chaque tour, on choisit le successeur ayant la meilleure évaluation
  - Si celle-ci est meilleure que l'évaluation du noeud courant.
- Condition d'arrêt
  - Quand aucun des successeurs n'a une meilleure valeur que le nœud courant.
- Avantage:
  - Pas d'arbre de recherche à maintenir
  - Pas de retour en arrière.

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

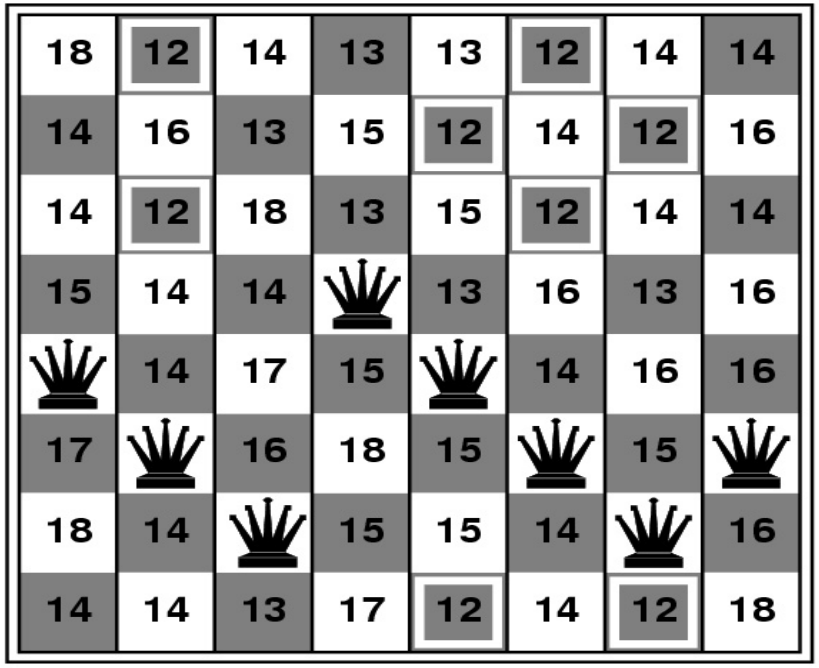


Algorithmes d'amélioration itérative

Par escalade (*Hill-climbing*)

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[ problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

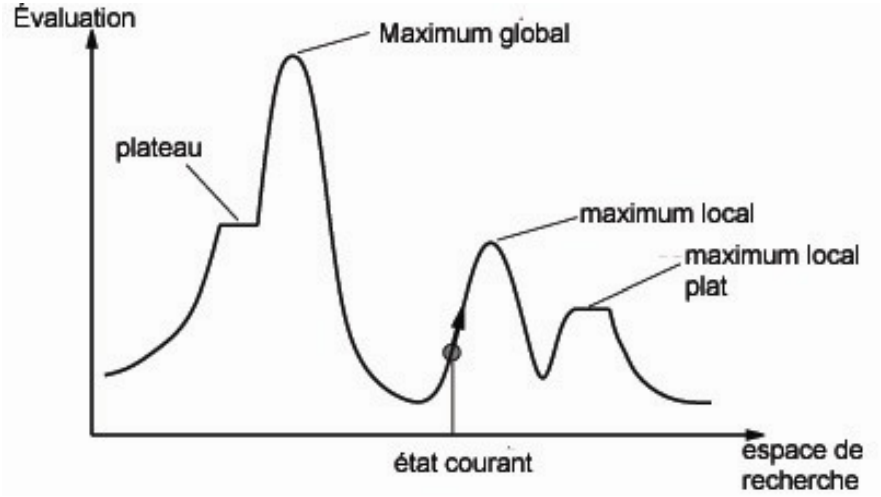
Exemple 8-reines

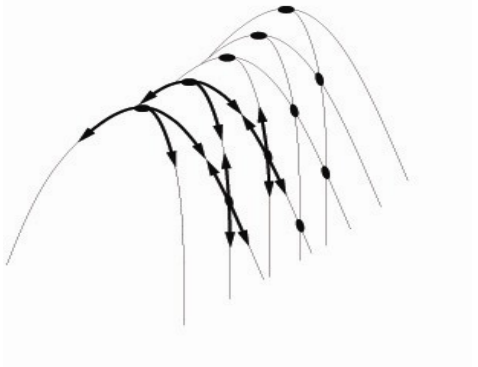


- h: le nombre de pairs de reines qui s'attaquent
- Pour cet état : h = 17
- On détermine la valeur des successeurs possibles
  - c.-à-d. le coup qui permet de minimiser le nombre d'attaques

Problèmes de *Hill-climbing*

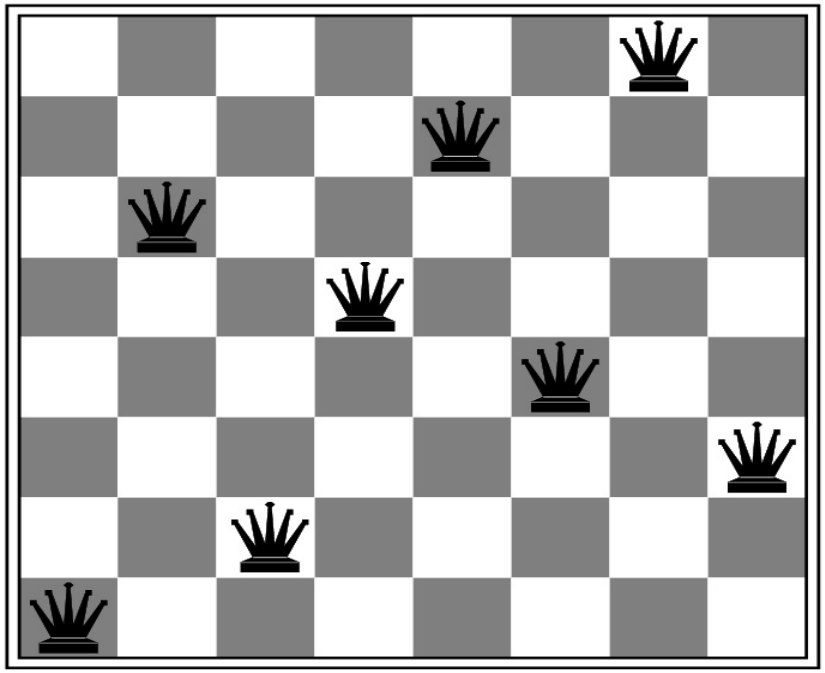
- Reste pris souvent :
  - Maximum local
  - Plateaux
  - Crêtes (*Ridges*)





Exemple minimum local

- Pour cet état : h = 1
- Tous les successeurs ont une valeur plus grande.



Redémarrage aléatoire

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

Algorithmes d'amélioration itérative

Recuit simulé (*simulated annealing*)

- L'idée est de permettre de mauvais déplacements dans le but d'échapper aux maxima locaux.
- Principe :
  - On sélectionner un déplacement aléatoire.
  - S'il améliore la situation, il est exécuté.
  - Sinon il est exécuté avec une probabilité inférieure à 1.
- Plus le mouvement empire la situation, plus la probabilité que le mouvement soit choisi est faible.
- Un paramètre *T*
  - Il tend vers zéro avec le temps.
  - Il est utilisé pour déterminer la probabilité d'un mauvais mouvement.
  - Plus la valeur de *T* est grande, plus un mauvais mouvement a des chances d'être exécuté.
- L'algorithme se comporte comme *hill-climbing* lorsque *T* tend vers zéro.

Algorithmes d'amélioration itérative

Recuit simulé

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

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

Recherche locale en faisceau

- Fonctionnement:
  - Commence avec k états aléatoires
  - À chaque étape, l'algorithme génère tous les successeurs des k états.
  - Si on trouve un but, on arrête.
  - Sinon, on choisit les k meilleurs successeurs et on recommence.
- Recherche en faisceau stochastique
  - Les k successeurs sont choisis aléatoirement
    - Selon une probabilité proportionnelle à la valeur de la fonction d'évaluation.

Algorithmes génétiques

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

Algorithmes génétiques :

Fonctionnement

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

Algorithmes génétiques :

Algorithme

- Initialiser:(*P* ← *p* individus aléatoires)
- Évaluer (Calculer la valeur d'adaptation de chaque hypothèse h)
- Tant que max *Fn-Adaptation(h)* < Borne utilité
  - Créer une nouvelle génération *Ps*
  - Sélectionner:
    - Choisir *(1-r)p* individus de *P* et les ajouter à *Ps* selon la probabilité *prob(h)*
  - Reproduire:
    - Choisir *r*p/2* paires d'individus selon la probabilité *prob(h)*.
    - Pour chaque paire produire deux enfants et les ajouter à *Ps*
  - Muter:
    - Choisir *m%* individus de *Ps* avec une probabilité uniforme
    - Modifier un gêne choisi aléatoirement
  - Mettre à jour: *P* ← *Ps*
  - Évaluer: Calculer la valeur d'adaptation de chaque individu
- Retourner l'individu avec la meilleure valeur d'adaptation

$$prob(h) = \frac{fn\_adaptation(h)}{\sum_{j=1}^{p} fn\_adaptation(h_j)}$$

***p* : le nombre d'individus dans la population**
***r* : le taux de reproduction**
***m* : le taux de mutation**

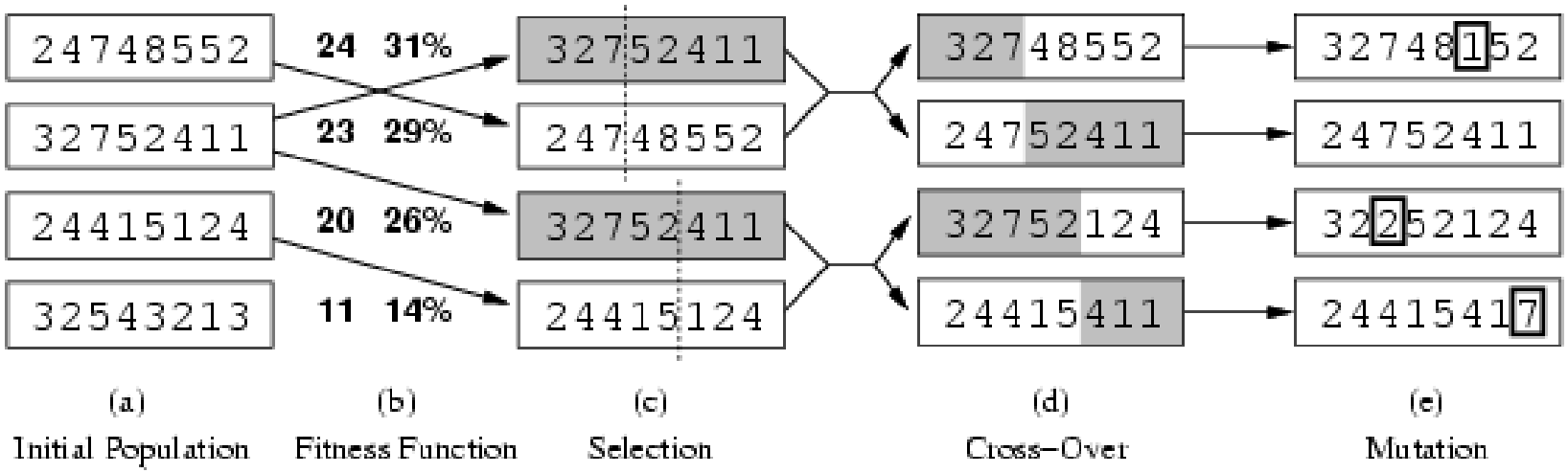
Algorithmes génétiques :

Algorithme

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    loop for i from 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

---

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Algorithmes génétiques :

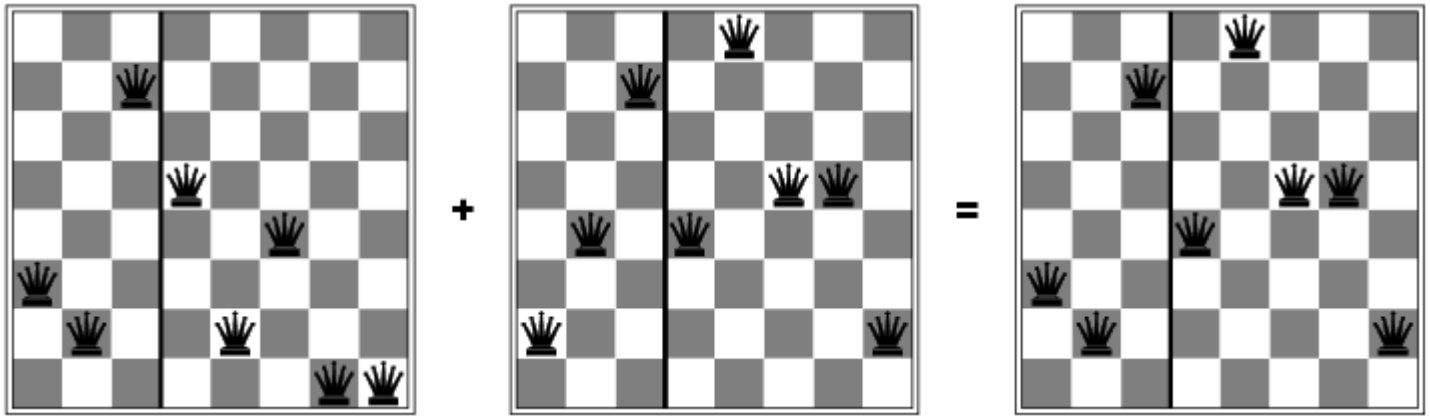
Illustration de l'algorithme



- Exemple de construction de population pour le problème des 8-reines

Algorithmes génétiques :

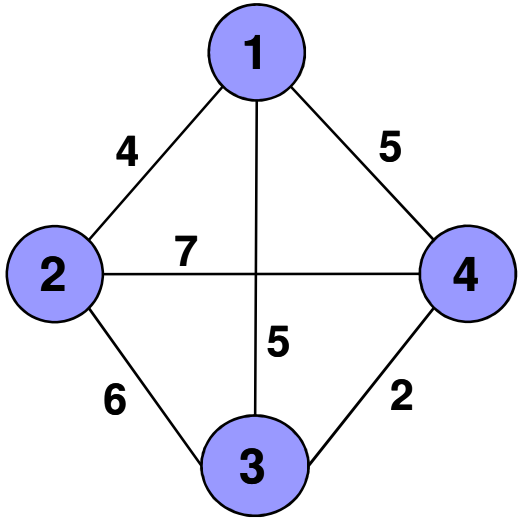
Illustration du croisement



Algorithmes génétiques :

Exemple

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



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

http://wwwsi.supelec.fr/yb/projets/algogen/VoydeCom/VoyDeCom.html

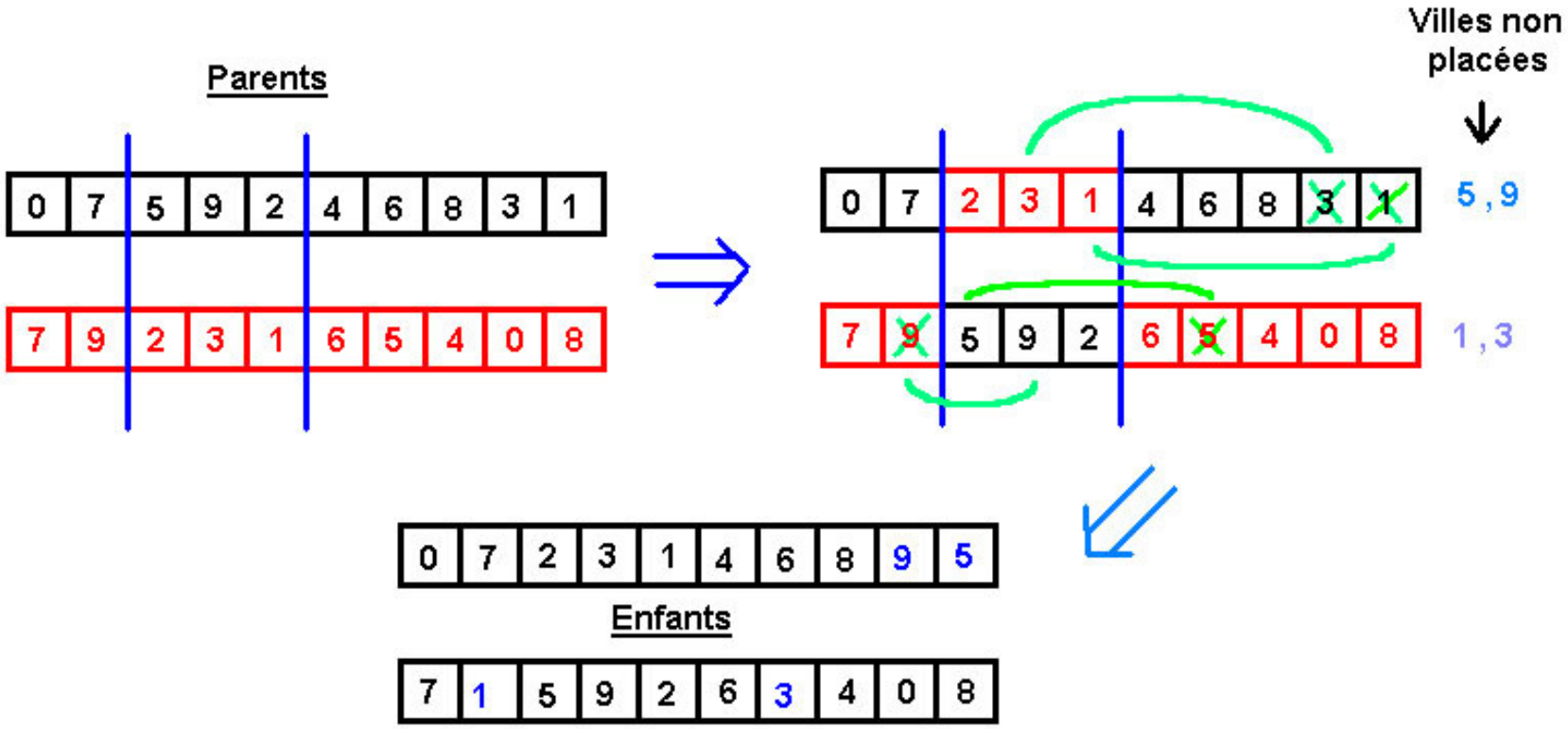
Algorithmes génétiques :

Exemple: Reproduction

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

Algorithmes génétiques :

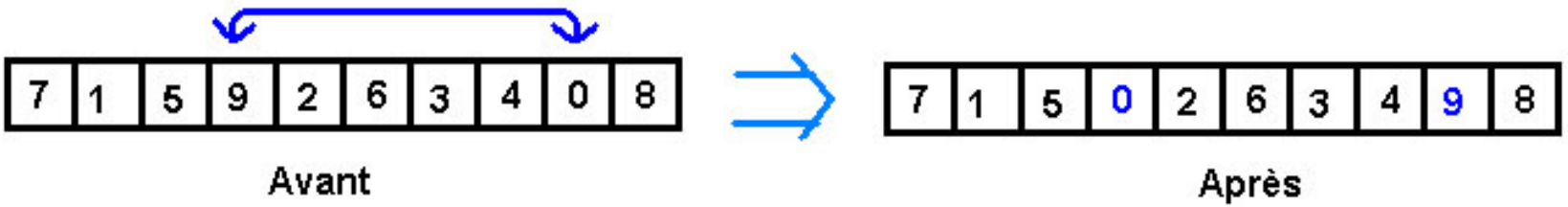
Exemple: Reproduction



Algorithmes génétiques :

Exemple: Mutation

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



Algorithmes génétiques :

Exemple: Fonction d'adaptation

- Si le parcours est invalide, c'est l'opposée du nombre de chemin inexistant
  - 3 chemins inexistants => Utilité = -3
- Si le parcours est valide, c'est l'inverse de la somme de la distance
  - Chemin de longueur 100 => Utilité = 0.01

Algorithmes génétiques :

Paramétrage

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

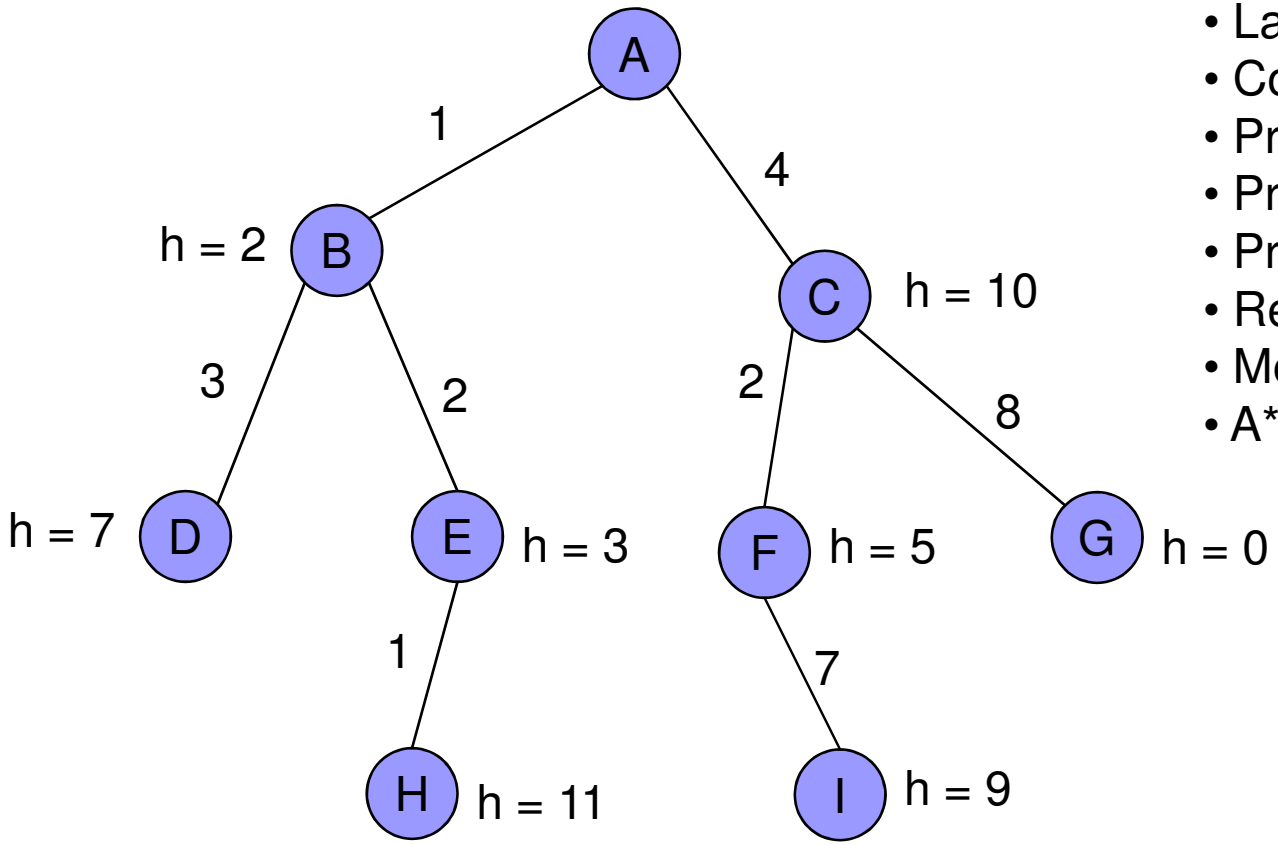
Algorithmes génétiques parallèles

- Séparer la population en sous-populations
- Appliquer les algorithmes génétiques indépendamment sur chacune des sous-populations
- On effectue des migrations entre les sous-populations (copie d'individus d'une population à une autre)
  - Permet une certaine diversité entre les individus
  - Permet d'éviter que le système tombe dans un maximum local dû à l'apparition trop hâtive d'un individu dominant

Recherche « online »

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

Exercice



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