Résolution de problèmes

Algorithmes IDA*, RBFS, min-max et alpha-beta

1 Extensions de l'A*

1.1 IDA* (iterative deepening A*)

Appliquer l'algorithme suivant avec l'heuristique correspondant au nombre de tuiles mal placées pour le taquin (la figure 1 rappelle l'état initial et l'état-but).

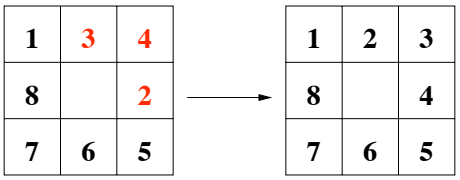

FIGURE 1 – Un état initial (à gauche) et l'**état-but** (à droite) du jeu du taquin

```
node                current node
 g                   the cost to reach current node
 f                   estimated cost of the cheapest path (root..node..goal)
 h(node)             estimated cost of the cheapest path (node..goal)
 cost(node, succ)    path cost function
 is_goal(node)       goal test
 successors(node)    node expanding function

 procedure ida_star(root, cost(), is_goal(), h())
   bound := h(root)
   loop
     t := search(root, 0, bound)
     if t = FOUND then return FOUND
     if t = infinity then return NOT_FOUND
     bound := t
   end loop
 end procedure

 function search(node, g, bound)
   f := g + h(node)
   if f > bound then return f
   if is_goal(node) then return FOUND
   min := infinity
```

```
  for succ in successors(node) do
    t := search(succ, g + cost(node, succ), bound)
    if t = FOUND then return FOUND
    if t < min then min := t
  end for
  return min
end function
```

1.2 Algorithme de recherche RBFS

L'algorithme de recherche recursive du meilleur d'abord (en anglais Recursive Best First Search ou RBFS) décrit par l'algorithme 1 est un autre algorithme sous contrainte de mémoire.

Algorithme 1 Algorithme de recherche récursive du meilleur d'abord. L'appel de la fonction est RBFS(probleme, CREER-NOEUD(probleme.ETAT-INITIAL), ∞)

Fonction RBFS(probleme, nœud, f_limite) retourne une solution ou (échec, une nouvelle limite de f)

si probleme.TEST-BUT(nœud.ETAT) **alors**
  retourner SOLUTION(nœud)
fin si
successeurs $\leftarrow \emptyset$
pour tout action $\in$ probleme.ACTIONS(nœud.ETAT) **faire**
  ajouter NOEUD-FILS(probleme, nœud, action) à successeurs
fin pour
si successeurs est vide **alors**
  retourner (échec, ∞)
fin si
pour tout s $\in$ successeurs **faire**
  s.f $\leftarrow$ max (s.g+s.h, noeud.f)
fin pour
tant que Vrai **faire**
  meilleur $\leftarrow$ nœud dans les successeurs avec la plus petite valeur f
  si meilleur.f > f_limite **alors**
    retourner (échec, meilleur.f)
  fin si
  alternative $\leftarrow$ la seconde plus petite valeur f parmi les successeurs
  (resultat, meilleur.f) $\leftarrow$ RBFS(probleme, meilleur, min(f_limite, alternative)
  si resultat $\neq$ échec **alors**
    retourner resultat
  fin si
fin tant que

Appliquer l'algorithme RBFS au problème de la recherche d'un itinéraire d'Arad à Bucarest (voir figure 2).

Remarques :
— Pour faciliter la représentation de l'arbre de recherche vous pouvez n'utiliser que la première lettre des villes.
— Faites figurer la valeur f et sa limite actuelle auprès de chaque nœud.
— Vous trouverez le coût réel des distances sur la figure 2. et la valeur de l'heuristique pour aller à Bucarest dans la table 1.
— Vous pouvez vous contenter de représenter les étapes clefs qui montrent l'effet de la contrainte de mémoire.

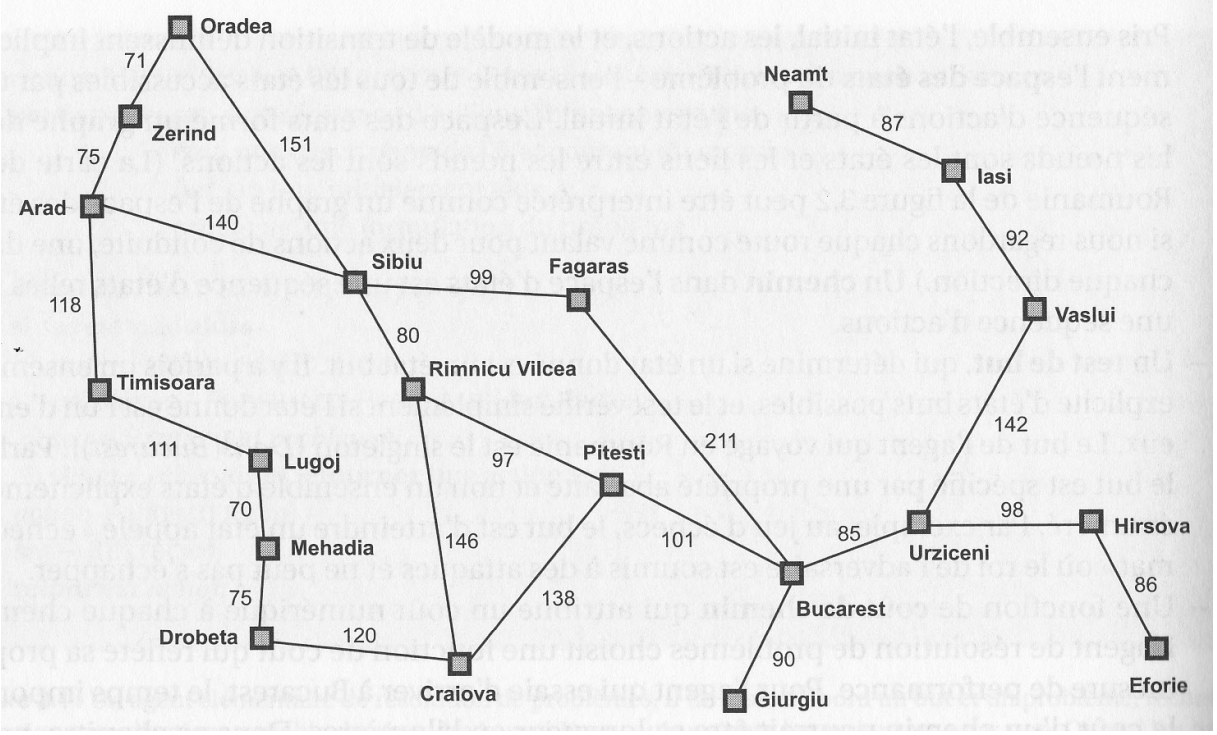

FIGURE 2 – Carte routière simplifiée d'une partie de la Roumanie

TABLE 1 – Estimation de la distance à vol d'oiseau jusqu'à Bucarest.

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

2 Algorithme de jeux

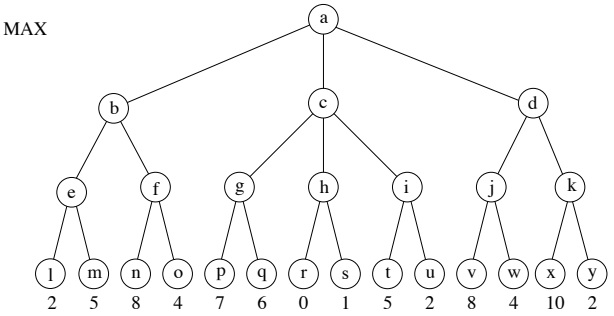

FIGURE 3 – Arbre de jeu. La racine est un nœud maximisant. Chaque lettre identifie un nœud. Chaque chiffre correspond à l'évaluation d'une feuille.

1. Appliquer l'algorithme minmax à l'arbre de jeu donné par la figure 3.
2. Appliquer l'algorithme α-β à l'arbre de jeu donné par la figure 3.
   a) Supposer d'abord que les fils des nœuds sont parcourus de gauche à droite et indiquer tous les nœuds qui ne sont pas considérés par l'algorithme.
   b) Faire la même chose en supposant que les fils sont parcourus de droite à gauche.
3. Commenter

limit correspond à la profondeur limite de recherche.
Elle réduit l'explosion combinatoire en diminuant l'espace de recherche.

Algorithme 2 Min-Max

```
int minmax (Node s, int depth, int limit)
Vector v = new Vector() ;
si isTerminal(s) OR depth == limit alors
  return (GainEvaluation(s)) ;
sinon
  // appliquer minmax sur les fils de s et mémoriser leurs valeurs
  tant que s.hasMoreSuccessors() faire
    v.addElement(minmax(s.getNextSuccessor(),depth+1,limit)) ;
  fin tant que
  si isComputerTurn(s) alors
    return maxOf(v) ; // tour du programme
  sinon
    return minOf(v) ; // tour de l'adversaire
  fin si
fin si
```

Algorithme 3 Alpha-Beta

```
α-β(Node s, int alpha, int beta, int depth, int limit) //La valeur d'un nœud est encadrée par alpha et beta.
si isTerminal(s) OR depth == limit alors
  return (GainEvaluation(s))
sinon
  // appliquer alpha-beta sur les fils de s et mémoriser leurs valeurs
  s.alpha = -∞ ;
  s.beta = +∞ ;
  si isComputerTurn(s) alors
    pour tout successeur si de s faire
      val = α-β(si, max(alpha, s.alpha),beta, depth+1, limit)
      s.alpha = max(s.alpha, val)
      si s.alpha ≥ beta alors
        break
      fin si
    fin pour
    return s.alpha
  sinon
    // tour de l'adversaire
    pour tout successeur si de s faire faire
      val = α-β(si, alpha, min(beta, s.beta),depth+1,limit)
      s.beta = min(s.beta, val)
      si s.beta ≤ alpha alors
        break
      fin si
    fin pour
    return s.beta
  fin si
fin si
```