TD 2 - Recherche de solution optimale

Fondements de l'IA symbolique - HAI710I

Soient les structures et fonctions générales de recherche (cf. annexe) permettant de parcourir les états d'un espace d'états par priorité croissante. On précise que :

— `Explorer` peut ré-explorer plusieurs fois un même état ;

— `ExplorerOptimise` évite de ré-explorer un état déjà exploré et ne conserve dans la frontière qu'un nœud par état.

Question 1 Préciser pour chacune des stratégies suivantes comment doivent être choisis les coûts et priorités $\mathbf{c_0}$, $\mathbf{c_{sn}}$, $\mathbf{p_0}$ et $\mathbf{p_{sn}}$ des deux fonctions `Explorer` et `ExplorerOptimise` :

— recherche par coût min (le coût étant défini comme la somme du coût des actions ayant conduit de l'état du nœud racine à l'état du nœud courant) ;

— recherche gloutonne ;

— recherche A*.

Question 2 Soit le problème de calcul d'une route de *Arad* à *Bucharest* (cf. document "Problème de recherche de route" sur Moodle). Dessiner les arbres de recherche correspondant à l'exécution de ces 3 stratégies en distingant le cas où la fonction `Explorer` est utilisée de celui où l'on utilise la fonction `ExplorerOptimise`.

Question 3 Même question de *Iasi* à *Fagaras*. On pourra prendre les distances à vol d'oiseau suivantes pour Fagaras :

Neamt 170	Eforie 380	Craiova 220
Iasi 220	Bucharest 190	Rimnicu Vilcea 100
Vaslui 260	Giurgu 250	Sibiu 120
Urziceni 230	Fagaras 0	
Hirsova 310	Pitesti 110	

Question 4 Dire pour chacune des deux versions de l'algorithme d'exploration, si les 3 stratégies précédentes sont complètes. Préciser éventuellement sous quelles conditions elles le sont.

Question 5 Une heuristique est dite **admissible** si pour tout état e elle ne sur-estime jamais le coût du chemin optimal de e à l'état but : $\forall e\ h(e) \leq g^*(e)$ ($g^*(e)$ dénote le coût d'un des chemins optimaux de e à l'état but le plus proche).

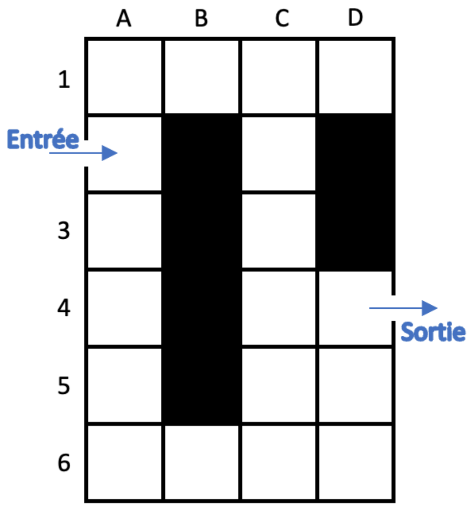
On considère la stratégie A* avec l'algorithme `Explorer`.

1. Exhiber un exemple montrant que si l'heuristique utilisée n'est pas admissible alors la solution trouvée peut ne pas être optimale.
2. Montrer que si l'heuristique est admissible alors la solution trouvée est optimale.
3. Est-ce vrai pour l'algorithme `ExplorerOptimise` ?

Question 6 Une heuristique est dite **monotone** si pour toute couple d'état (e, e') tel que e' est accessible de e par une action a, elle vérifie l'inégalité triangulaire suivante : $h(e) \leq c(a) + h(e')$.

1. Montrer que si l'heuristique est monotone, alors elle est admissible.
2. Exhiber un exemple montrant que si l'heuristique utilisée est admissible mais pas monotone alors la solution trouvée par la stratégie A* et l'algorithme `ExplorerOptimise` peut ne pas être optimale.
3. Montrer que si l'heuristique est monotone alors la solution trouvée par la stratégie A* et l'algorithme `ExplorerOptimise` est optimale.

Question 7 On considère le problème de labyrinthe suivant, où les 4 actions autorisées (quand elles sont possibles) sont les déplacements d'une seule case dans les 4 directions :



1. Proposez une modélisation par espace d'états pour ce problème.
2. On souhaite utiliser la stratégie A* pour résoudre ce problème. On envisage les deux heuristiques suivantes :
   — distance à vol d'oiseau ;
   — distance de Manhattan.

   Dites si ces heuristiques sont admissibles et/ou monotones ?
3. Une heuristique h_1 **domine** une autre heuristique h_2 si pour tout état e on a $h_1(e) \geq h_2(e)$. L'une de ces heuristiques domine t-elle l'autre ?
4. Exécutez l'algorithme `ExplorerOptimise` avec la stratégie A* en prenant la "meilleure heuristique".
5. Montrer que si deux heuristiques sont monotones et que l'une domine l'autre alors le nombre de nœuds explorés par A* par l'heuristique dominante est inférieur ou égal au nombre de nœuds explorés par A* par l'heuristique dominée.

Question 8 On reprend le problème du taquin du TD précédent. Proposez des heuristiques et caractérisez leur admissibilité, monotonie et relations de dominance ?

Annexe

```
Interface Etat {
};

Interface Action {
    resultat(e : Etat) : Etat ;
};

Interface Probleme {
    etatInitial : Etat ;
    actions(e : Etat) : Ensemble d'Action ;     // actions possibles pour un état donné
    resultat(e : Etat, a : Action) : Etat ;
    but?(e : Etat) : Booleen ;
    cout(a : Action) : Reel ;
    heuristique(e : Etat) : Reel ;
};

Interface Noeud {
    etat : Etat ;
    parent : Noeud ;
    action : Action ;
    cout : Reel ;
    priorite : Reel;
    Noeud(e : Etat, p : Noeud, a : Action, c : Reel, p : Reel) : Noeud ;
};

Interface Liste<Noeud> {
    Liste() : Liste ;           // constructeur de liste vide
    vide?() : Booleen ;
    oterTete() : Noeud ;
    oterNoeud(n : Noeud) : void ;
    insererTete(n : Noeud) : void ;
    insererQueue(n : Noeud) : void ;
    insererCroissant(n: Noeud) : void ;
    rechercher(e : Etat) : Noeud (ou null) ;
};

Interface Ensemble<Etat> {
    Ensemble() ; // constructeur d'ensemble vide
    contient?(e : Etat) : Booleen;
    ajouter(e : Etat) : void ;
};
```

Fonction Explorer(p : Probleme) : Noeud (ou null)

racine ← new Noeud(p.etatInitial, null, null, $\mathbf{c_0}$, $\mathbf{p_0}$) ;
frontiere ← new Liste<Noeud>() ;
frontiere.inserer(racine) ;
tant que non *frontiere.vide ?()* **faire**
    n ← frontiere.oterTete() ;
    si *p.but ?(n.etat)* **alors retourner** *n*;
    pour toute *action a* **dans** *p.actions(n.etat)* **faire**
        sn ← new Noeud(a.resultat(n.etat), n, a, $\mathbf{c_{sn}}$, $\mathbf{p_{sn}}$) ;
        frontiere.insererCroissant(sn) ;
retourner *null* ;

Fonction ExplorerOptimise(p : Probleme) : Noeud (ou null)

```
racine ← new Noeud(p.etatInitial, null, null, c0, p0) ;
frontiere ← new Liste<Noeud>() ;
frontiere.inserer(racine) ;
explore ← new Ensemble<Etat>() ;
tant que non frontiere.vide ?() faire
    n ← frontiere.oterTete() ;
    explore.ajouter(n.etat) ;
    si p.but ?(n.etat) alors retourner n;
    pour toute action a dans p.actions(n.etat) faire
        se ← resultat(n.etat, a) ;
        si non explore.contient ?(se) alors
            sn ← new Noeud(se, n, a, csn, psn) ;
            sosie ← frontiere.rechercher(se) ;
            si sosie = null alors
                frontiere.insererCroissant(sn) ;
            sinon si sn.priorite < sosie.priorite alors
                frontiere.oterNoeud(sosie) ;
                frontiere.insererCroissant(sn) ;
retourner null ;
```