Rappel : Notations utilisées pour décrire les paramètres des problèmes à espace d'états :

- **cout(a)** désigne le coût de l'action a ;
- **g(n)** désigne le coût du nœud n, i.e. le coût du chemin de l'état initial à l'état courant $n.etat$ en réalisant toutes les actions (et donc passant par tous les états) des nœuds de la branche de l'arbre de recherche qui va du nœud racine au nœud n ;
- **h(e)** désigne l'heuristique de l'état e ; c'est une estimation du coût d'un chemin optimal de l'état e à l'état but le plus proche ;
- **h(n)** désigne l'heuristique de l'état du nœud n, c'est-à-dire que $h(n) = h(n.etat)$;
- **f(n)** désigne le coût estimé d'une solution commençant par les actions sur le chemin la racine à n ; c'est donc le coût réel de cette première partie de solution permettant d'atteindre l'état de n augmenté de l'estimation du coût permettant d'aller à l'état but le plus proche de l'état de e, c'est-à-dire que $f(n) = g(n) + h(n)$;
- **g*(e)** désigne le coût d'un chemin optimal de l'état e à l'état but le plus proche ;
- **g*** désigne le coût d'une solution optimale ; on a donc $g^*=g^*(e_i)$ où e_i est l'état initial.

```
Interface Etat {
};

Interface Action {
    resultat(e : Etat) : Etat ;
    cout : Reel ;                              // coût de réalisation de l'action
};

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

Interface Noeud {
    etat : Etat ;      // état courant
    parent : Noeud ;   // chemin parcouru pour atteindre l'état précédent
    action : Action ;  // action ayant permis de passer de l'état précédent à l'état courant
    cout : Reel ;      // coût des actions réalisées sur le chemin de la racine à ce nœud
    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() : 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, c0, p0) ;
    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, csn, psn) ;
            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 ← a.resultat(n.etat) ;
            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 ;
```