# Module TRP
# Techniques de Résolution de Problèmes
# Travaux pratiques

*Semestre 1 - Année Universitaire 2013-2014*

# Introduction

Les TP du module TRP sont obligatoires. Les notions étudiées dans les cours et TD de TRP sont considérées comme étant acquises. Vous allez travailler pendant les TP pour implémenter en langage `CAML` certains des algorithmes et des concepts que vous avez étudiés en cours et TD. Les TP ne doivent pas être un simple exercice de programmation mais doivent vous permettre d'acquérir une compréhension plus profonde du cours.

## Organisation

Il vous faut systématiquement apporter vos documents de travail. Il vous sera impossible de réaliser le travail demandé si vous n'avez pas travaillé les cours et TD avant le TP. Il y aura 5 séances de TP, en voici le planning :

| *Séance* | *Titre* |
|---|---|
| **S1 TP1** | **Implémentation de l'algorithme A*** |
| **S2 TP2** | **Implémentation de l'algorithme A*** |
| **S3 TP3** | **Implémentation de l'algorithme A*** |
| **S4 TP4** | **Application : Représentation interne,** |
| **S5 TP5** | **Application : heuristique et tests** |

## Modalités de contrôle des connaissances

Le travail réalisé en TP donne lieu à une note (sur 20) calculée de la manière suivante :

- 10% de la note est fondée sur l'assiduité aux TP.
- 90% de la note est fondée sur une épreuve écrite d'1h en amphi qui porte sur tous les TP.

La note de TP compte pour 30% de la note TRP en 1ère session, et est reportée en 2ème session avec un poids de 25%. Notez que :

- 2 absences non justifiées auprès de l'enseignant de TP entraînent automatiquement une note d'assiduité de 0.
- Une demande de dispense d'assiduité peut être adressée au responsable de l'UE et déposée, accompagnée de pièces justificatives (certificat médical, bulletin de salaire…) le plus tôt possible au secrétariat. Si la dispense est accordée, le poids de l'assiduité est reporté sur l'épreuve écrite de TP.
- Une absence non justifiée lors de l'épreuve écrite terminale entraîne un 0 pour cette épreuve écrite. En cas d'absence justifiée, la note attribuée à l'épreuve sera 0 à la 1ère session mais ne sera pas prise en compte en 2ème session. *Une absence est considérée comme justifiée si un certificat médical ou de travail est fourni au secrétariat dans les 10 jours ouvrables.*

## Conventions à respecter pour la programmation

**Les fichiers :** pour chaque fichier, vous donnerez en en-tête les nom, prénom et nom d'utilisateur du programmeur, ainsi que la date de la dernière modification.

**Les programmes :** ils devront être commentés « intelligemment » et sans faute d'orthographe ; l'algorithme A* doit être implémenté en style fonctionnel uniquement. Pour chaque fonction, le nom doit être suivi de la liste des paramètres **typés**, avec, en commentaire, pour chaque paramètre, son rôle ; vous devrez ensuite indiquer en commentaire le rôle de la fonction, ainsi que les variables globales utilisées s'il y en a ; vous spécifierez enfin le type de l'objet retourné par la fonction. Voici un exemple d'en-tête de fonction :

```
(*************************************************************)
let rec insereA
(*************************************************************)
((n,x) : int * 'a)     (* une association                    *)
(l : (int * 'a) list) (* une liste d'associations ordonnées *)
                       (* en ordre croissant                 *)
(*************************************************************)
(* retourne la liste l augmentée de (n,x) rangée à sa place;*)
(* s'il y avait déjà une occurrence de x associée à un      *)
(* entier m, il n'en reste qu'une associée au minimum       *)
(* de m et n                                                 *)
(*************************************************************)
: (int * 'a) list
(*************************************************************)
= ....(* code de la fonction *)...
```

Commentez les parties les plus délicates du code. Indentez de manière systématique avec des espaces en début de ligne.

## Mise en route

Les projets sont à réaliser en *Objective Caml*. Le manuel de référence du langage est disponible en ligne à l'adresse suivante : `http://caml.inria.fr/pub/docs/manual-ocaml`. Le langage `CAML` peut être téléchargé à l'adresse : `http://caml.inria.fr/download.fr.html`. Un manuel d'initiation à OCaml est disponible à l'adresse suivante : http://www-igm.univ-mlv.fr/~beal/Enseignement/Logique/ocaml.pdf.

Logez-vous et appelez `CAML` en tapant la commande « `ocaml` » dans un shell. On charge un fichier en tapant l'instruction « `#use "fichier.ml" ;;` ». On sort de l'interpréteur CAML en tapant l'instruction « `exit 0 ;;` ». (Vous pouvez également lancer ocaml depuis Emacs en passant d'abord en mode commande : ESC-X puis en tapant « `run-caml` ».)

L'énoncé fait référence à des fichiers que vous pouvez (à condition d'avoir un nom d'accès et un mot de passe) récupérer sur Moodle (moodle.ups-tlse.fr) dans la zone TP du module TRP du M1 d'informatique.

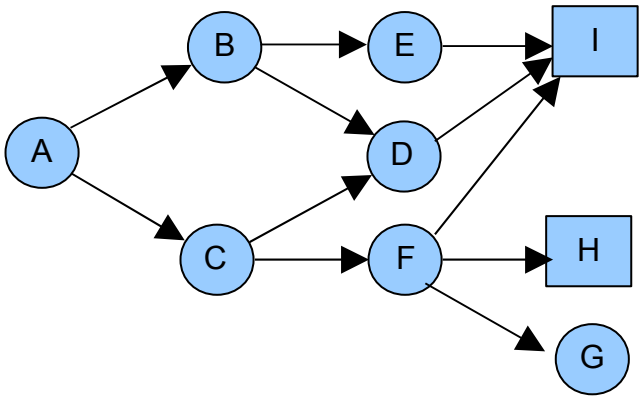
**Pour réaliser les fonctions demandées, nous vous recommandons d'écrire des fonctions auxiliaires et de les tester en vous servant des exemples de problèmes donnés sur moodle sous forme de graphes.**

# TRP – TP1, TP2 et TP3
# Recherche dans les espaces d'états, algorithme A*

**Durée : 5** h environ

## Recherche en profondeur d'abord dans un espace d'états

Voici ci-contre un exemple d'espace d'états donné « en **extension** ». Nous utiliserons ce graphe, qui représente un problème de petite taille, pour illustrer l'exécution de l'algorithme que vous devez écrire. D'autres graphes de ce type sont disponibles sur moodle. Ils doivent uniquement vous servir à vérifier le bon fonctionnement de votre algorithme de recherche. Dans de véritables applications, le graphe d'états est souvent de taille exponentielle et il ne peut évidemment pas être représenté « en extension ». On le représente donc « en **intention** », c'est-à-dire en donnant les trois caractéristiques du problème qu'il représente :



*graphe g1 : Exemple d'espace d'état*

- l'**état initial** ;
- la fonction qui caractérise **le but** ;
- la fonction qui définit les **états successeurs** d'un autre état.

Dans le graphe précédent, chaque sommet représente un état **`(A,B,C…I)`**, les états buts sont **`H`** ou **`I`**. Chaque arc permet de passer d'un état à un état-fils. Ainsi, en partant de **`A`** on arrive à **`I`** en passant par **`C`** puis par **`D`**.

Une représentation CAML de ce graphe g1 est donnée sur Moodle dans le fichier "*graphes.ml*". Cette représentation se décompose, pour ce graphe **`g1`**, en deux fonctions qui sont les suivantes :

- **`estBut1`**, de type **`string -> bool`**, la fonction qui teste si un état est un but ;
- **`etatsSuivants1`**, de type **`string -> string list`**, la fonction qui associe à chaque état la liste des états successeurs possibles depuis cet état.

On accède aux différentes fonctions caractérisant ce graphe **`g1`** ainsi :

```
# estBut1 "H";;
- : bool = true
# etatsSuivants1 "A";;
: string list = ["C"; "B"]
```

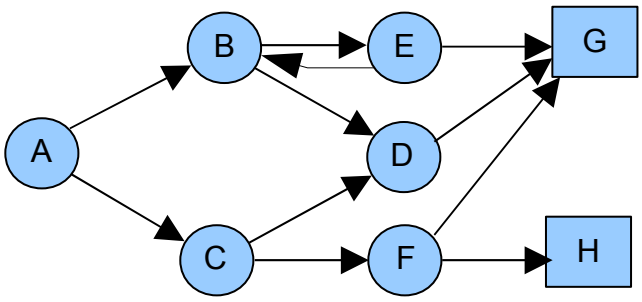
On vous demande, dans un premier temps, d'implémenter, en `CAML` et **en style fonctionnel**, un **algorithme de recherche dans les espaces d'états** qui travaille **en profondeur d'abord**. Ce type de recherche est systématique et ne nécessite donc pas l'utilisation d'une heuristique. La fonction **`profondeur`** devra être récursive et travailler sur une liste représentant la file d'attente. Cette fonction utilisera les fonctions **`estBut`** et **`etatsSuivants`** supposées définies pour le problème courant.

**1.** Etant donné la description d'un problème grâce à la définition des deux fonctions **`estBut`** et **`etatsSuivants`**, écrire une première version d'une fonction **`profondeurV1`** qui renvoie le premier état But atteint par un parcours en profondeur. Vous pourrez tester votre fonction sur le graphe **`g1`** (en définissant les fonctions **`estBut`** et **`etatsSuivants`** comme égales à **`estBut1`** et **`etatsSuivants1`** définies dans le fichier "*graphes.ml*").

```
# profondeurV1 ["A"];;
- : string = "H"
```

Vous pourrez ensuite la tester avec les deux autres fonctions de but proposées : **`estBut1bis`** et **`estBut1ter`**.

**2.** Dans un espace d'état, il peut y avoir des **boucles**. Par exemple, avec un jeu de taquin, on peut revenir à un état donné après plusieurs mouvements et, en refaisant la même suite de mouvements, tourner indéfiniment sans jamais trouver de solution. Pour éviter à l'algorithme de boucler, vous rajouterez à la fonction précédente **`profondeurV1`** une liste **`vus`** dans laquelle vous mémoriserez, au fur et à mesure les états déjà développés. Si l'algorithme retrouve l'un de ces états pendant la recherche, il évitera de le développer à nouveau en coupant ainsi la branche de l'arbre de recherche qui lui correspond. Dans un souci d'optimisation de votre algorithme de recherche, chaque état **`e`** devra être mémorisé au plus une fois dans la file d'attente. Vous pourrez tester votre fonction sur le graphe **`g2`**, décrit par **`estBut2`** et **`etatsSuivants2`** dans le ficher "*graphes.ml*" :



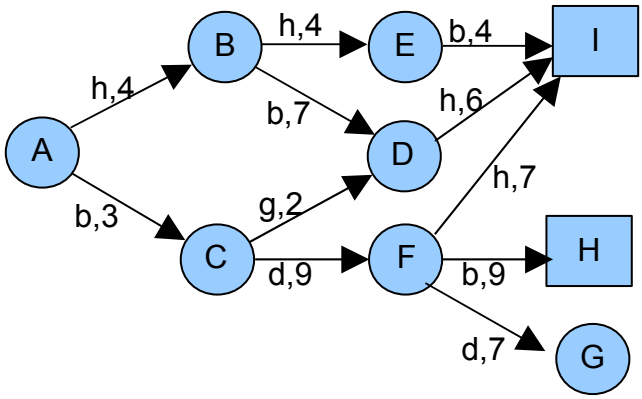
*Graphe g2*

```
# profondeurV2 ["A"] [];;
- : string = "G"
```

**3.** Faites une nouvelle version **`profondeurV3`** qui renverra le premier **chemin** menant au but (sous forme d'une liste d'états), la file d'attente devra maintenant contenir des **chemins** et non pas simplement des états. En testant avec le graphe **`g2`**, on obtient :

```
# profondeurV3 [["A"]] [];;
-: string list = ["G"; "E"; "B"; "A"]
(* chemin vers "G" stocké à l'envers *)
```

On considère maintenant que chaque opération a un coût. Dans le graphe précédent, chaque arc est associé à une opération de changement d'état, ici **`h`** (pour « haut ») ou **`b`** (pour « bas ») ou **`d`** (pour « droite ») ou **`g`** (pour « gauche ») et porte le coût réel de celle-ci. Ainsi, en partant de **`A`** et en réalisant la suite d'opérations **`b, g, h`** le coût total est de 11. Dans la suite du texte on note **`coutParcouru(ch)`** le coût des opérations d'un chemin **`ch`** ; par exemple, le coût du chemin **`ACD`** est **`coutParcouru(ACD) = 3 + 2 = 5.`** Écrire une fonction **`profondeur2`** qui renvoie le **chemin solution** (sous la forme de la liste des opérations utilisées) ainsi que son coût. Ainsi, en partant de **`A`** et en faisant la suite d'opérations **`b, d, b,`** on arrive à **`H`** en passant par **`C`** et **`F`**, pour un coût de 21. Cette fonction utilisera les mêmes fonctions que la fonction **`profondeur`** à part la deuxième qui sera une fonction (appelée **`opPoss`** dans le fichier d'exemples) qui à partir d'un état donne une liste d'éléments où chaque élément décrit l'opération réalisable, son coût et l'état résultant. Vous pourrez tester votre fonction sur les différents graphes fournis dans le fichier "*graphes.ml*". Pour le graphe 1, on vous donne la fonction **`opPoss1`**, de type **`string -> (char * string * int) list`**, qui associe à chaque état la liste des opérations possibles à partir de cet état, associées aux états résultants et aux coûts réels de ces opérations :



*graphe g1 avec coûts des opérations*

```
# opPoss1 "C";;
- : (char*string*int) list =[('d',"F", 9);('g',"D",2)]
```

**4.** Ecrire une dernière version de la fonction, **`profondeurV4`** qui renverra les trois informations suivantes : l'état but, la liste des **opérations** pour l'atteindre, le coût de ce chemin-solution. Cette fonction devra donc stocker ce genre d'information dans la file d'attente.

## Rappels sur l'algorithme A*

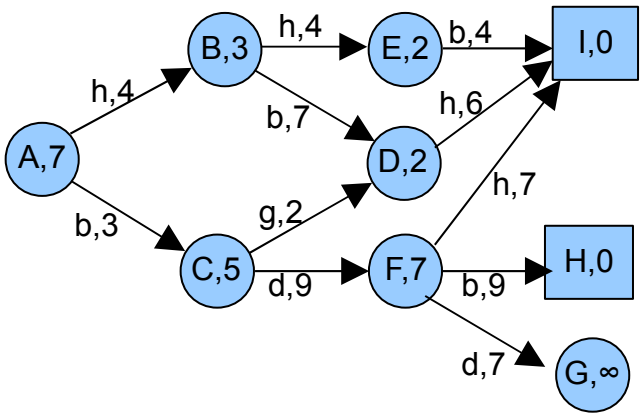
L'algorithme **A*** est un algorithme de recherche dans les **espaces d'états**. Pour un problème donné, l'algorithme développe une partie de l'espace d'états du problème. L'espace qui est développé (et que l'on souhaite être le plus petit possible) est appelé **espace de recherche**. Partant de l'état initial, l'algorithme développe donc cet espace de recherche en essayant, de trouver un **chemin-solution** (composé d'une suite d'opérations) qui permette d'arriver à un état satisfaisant le but du problème. Pour réduire la taille de l'espace de recherche, l'algorithme est guidé par une **fonction heuristique qui estime, pour chaque état étudié**, le **coût minimal** de la suite des opérations (ou chemin) qui permettrait, à partir de cet état, d'arriver à un état-but. Si cette heuristique est **minorante,** c'est-à-dire si l'estimation du coût pour arriver au but est toujours inférieure au coût réel, alors le chemin-solution retourné par **A*** est un chemin de coût minimum pour le problème posé.

L'algorithme **A*** utilise une **file d'attente** qui mémorise chaque état atteint ainsi que le chemin partant de l'état initial qui a permis de l'atteindre. Au départ la file d'attente ne contient qu'un seul élément : un « chemin vide »[1] depuis l'état initial. Pendant tout le déroulement de l'algorithme les chemins de la file d'attente y sont **classés dans l'ordre croissant de leur coût estimé** (les chemins de plus faible coût sont considérés en premier). **Le coût estimé d'un chemin `ch` d'état terminal `e`, est égal au coût réel du chemin parcouru pour arriver en `e` + l'estimation (donnée par l'heuristique) du chemin qu'il reste à parcourir depuis `e` pour arriver à un état-but.** À chaque étape de l'algorithme, le chemin le plus prometteur (c'est-à-dire celui de plus faible coût, et qui se trouve donc en tête de la file d'attente) est **développé** : il est retiré de la file et ses « chemins fils » sont insérés dans cette même file à une position qui dépend de leur coût estimé (chemins de plus faible coût d'abord). L'algorithme stoppe quand le chemin qui est retiré de la file d'attente mène à un état-but.

Un exemple d'espace de recherche développé en **extension** par l'algorithme A* avec une fonction heuristique est donné à la page suivante. Dans l'algorithme A*, l'espace d'états peut être représenté en **intention** en précisant les quatre caractéristiques du problème qu'il représente :

- l'**état initial** ;
- la fonction qui caractérise **le but** ;
- la fonction qui détermine les **opérations réalisables** à partir de chaque état, leur **coût**, et les **états qui en résultent** ;
- la fonction **heuristique** qui estime le coût minimal d'un chemin entre chaque état et un état-but.

Dans le graphe ci-contre, chaque sommet porte la valeur de la fonction heuristique **`hEtat`** qui estime le coût minimal d'un chemin entre cet état et l'un des états buts **`H`** ou **`I`**. Par exemple, **`hEtat(A)=7`** signifie que l'heuristique estime que le coût minimal pour aller de **`A`** à l'un des deux états-buts est de 7. Pour un chemin **`ch`** dont l'état terminal est **`e`**, on note **`estimChemin(ch) = coutParcouru(ch) + hEtat(e)`** : cela représente le coût estimé du prolongement du chemin **`ch`** jusqu'à un état-but ; ainsi, **`estimChemin(ACD) = 5 + hEtat(D) = 5 + 2 = 7`**. On peut vérifier (en examinant le graphe de manière exhaustive) que l'estimation heuristique sur cet exemple est minorante, c'est-à-dire que le coût estimé est systématiquement inférieur au coût minimal réel pour tout état du graphe.

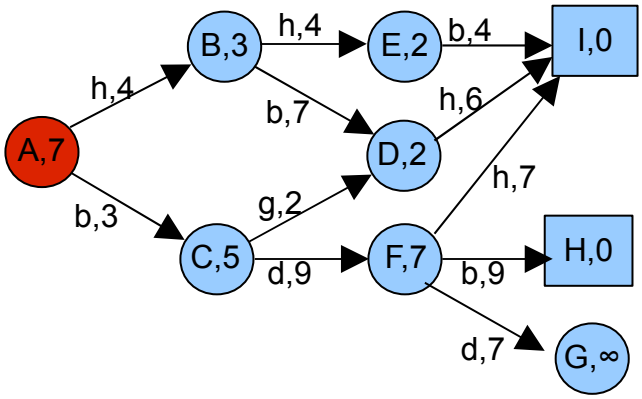



*Espace d'états du problème (à gauche : l'état initial, en rectangle : les deux états buts)*

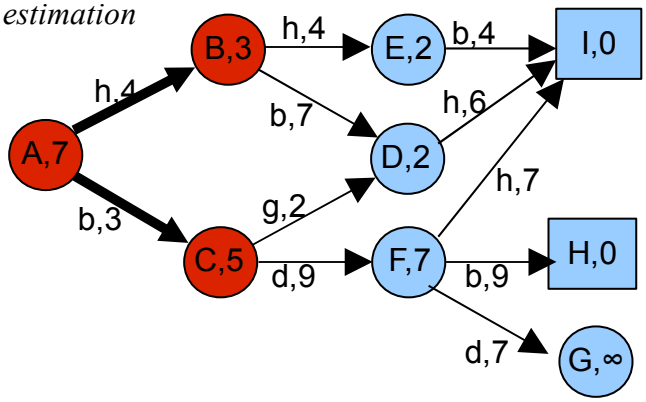
Partant de l'état initial, l'algorithme développe l'arbre de recherche comme le montrent les schémas suivants (à lire de gauche à droite et de haut en bas). Le nœud qui va être développé est le premier de la file d'attente (en rouge les nœuds qui ont été mis dans la file d'attente).

1 Par abus de langage ce terme désigne un « chemin » sans aucun arc (contenant seulement un état).

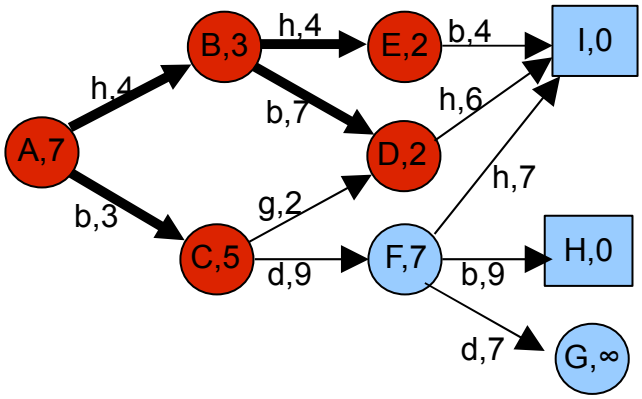
*[(A,0+7)] : au début de l'algorithme, seul l'état initial est dans la file d'attente*



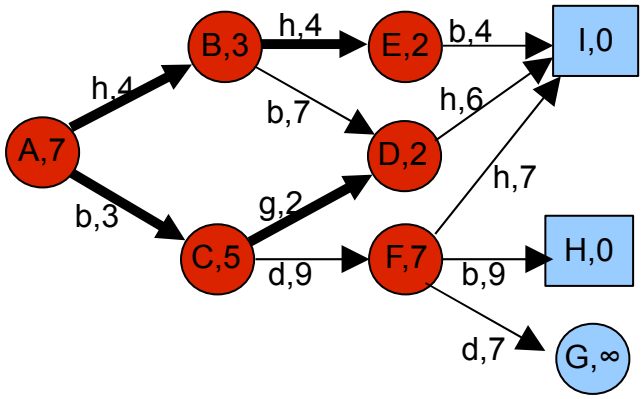

*[(B,4+3);(C,3+5)] : on développe le meilleur nœud de la file et on insère ses fils suivant leur estimation*



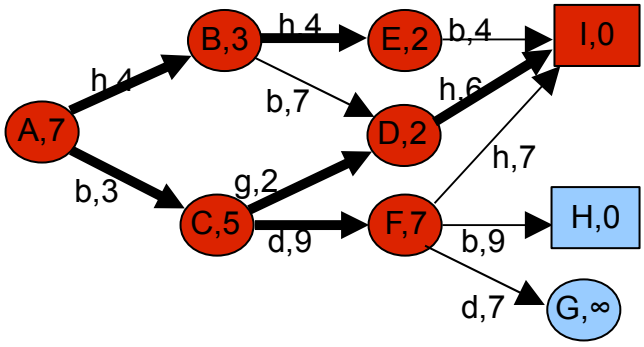

*[(C,3+5);(E,8+2);(D,11+2)] : idem*



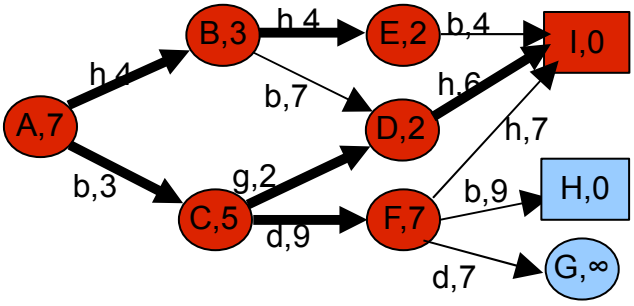

*[(D,5+2);(E,8+2);(F,12+7)] : idem et D est mis à jour avec un meilleur chemin*




*[(E,8+2);(I,11+0);(F,12+7)] : idem*




*[(I,11+0);(F,12+7)] : I est un état but fin de l'algorithme retour du chemin [b;g;h] de coût 11*




Une représentation CAML de ce graphe g1 est donnée sur Moodle dans le fichier "*graphes.ml*". Cette représentation contient la fonction **`hEtat1`** :

- **`hEtat1`**, de type **`string -> int`**, la fonction heuristique, qui associe à chaque état une estimation du coût minimal des chemins allant de cet état à un état but.

On accède à cette fonction ainsi :

```
# hEtat1 "A";;
- : int = 7
```

## Implémentation de l'algorithme A*

**1.** En modifiant votre fonction **profondeur**, **implémentez l'algorithme A*** en CAML en style fonctionnel. La fonction **aEtoile** devra être récursive et travailler sur une liste représentant la file d'attente. Pour lancer la recherche, on appellera **aEtoile** avec une file d'attente ne contenant que le chemin constitué du seul état initial. Elle utilisera les trois fonctions qui caractérisent le problème :

- **estBut : ('a -> bool)** le prédicat caractérisant les buts (renvoie **vrai** si l'état est un but);
- **opPoss : ('a->('b*'a*int) list)** la fonction qui retourne la liste des opérations possibles à partir d'un état, associées aux états d'arrivée et aux coûts de ces opérations ;
- **hEtat : ('a -> int)** la fonction heuristique qui donne une valeur numérique entière à tout état ;

Etant donnée la description d'un problème au moyen des 3 fonctions qui le caractérisent, cette fonction **aEtoile renvoie un chemin-solution.**

La fonction **aEtoile** appliquée au graphe **g1** depuis son état initial **A** est appelée ainsi :

```
#aEtoile ?? (*file d'attente contenant le chemin associé à A*);;
```

Elle utilise les fonctions **estBut**, **opPoss** et **hEtat** fixée pour les tests à **estBut1**, **opPoss1** et **hEtat1**. Elle donne comme résultat le chemin **['b';'g';'h']** de coût 11.

## Amélioration de l'algorithme A* : utilisation de la liste « vus »

Comme il existe généralement plusieurs chemins qui permettent d'arriver à un même état, il est possible que l'algorithme, au cours de sa recherche, arrive à un état **e** par un premier chemin puis, un peu plus tard, par un autre, de coût différent, potentiellement plus intéressant. Pour ne pas stocker des états déjà traités et non intéressants, on utilise une liste **vus** où l'on mémorise chaque état que l'on vient de développer (c'est-à-dire que l'on vient de retirer de la file d'attente pour créer ses fils), associé au meilleur coût du chemin (que l'on a trouvé jusqu'ici) qui permet d'y aboutir. Cette liste **vus** est optimisée en y stockant chaque état une seule fois.

La liste **vus** est utilisée lors du traitement d'un état que l'on extrait de la file d'attente. Cet état est développé, c'est-à-dire que l'on produit ses états-fils et leurs chemins respectifs associés. Chaque état-fils est ensuite considéré afin de savoir s'il doit ou non être inséré dans la file d'attente :

i) si l'état-fils considéré a déjà été vu avec un chemin de coût inférieur (il figure dans la liste **vus** avec ce dernier coût), il est inutile de le mémoriser dans la file d'attente (on coupe ainsi la recherche d'un chemin-solution passant par ce nouvel état-fils et on s'occupe du fils suivant) ;

ii) si l'état-fils considéré n'a jamais été vu, on l'insère dans la file d'attente en fonction du coût de son chemin associé (le nouveau chemin que l'on vient de trouver) ;

iii) si l'état-fils considéré a déjà été vu mais avec un chemin de coût supérieur, on l'élimine de la liste **vus** et on l'insère dans la file d'attente en fonction du coût de son chemin associé (le nouveau chemin, de coût plus faible, que l'on vient de trouver). C'est ce qui se passe à l'étape 4 du développement de l'exemple.

**2.** Améliorez votre implémentation de l'algorithme A* en lui ajoutant une liste **vus**.

# Affichage de statistiques et tests de l'algorithme A*

**3.** Pour finir, vous implémenterez une fonction **recherche** qui réalisera le premier appel à la fonction **aEtoile** avec une file d'attente ne contenant que le chemin constitué de l'état initial : les paramètres de la fonction recherche devront respecter les types suivants :

- **('a -> bool)** le prédicat caractérisant les buts (renvoie **vrai** si l'état donné est un but.
- **('a->('b*'a*int) list)** la fonction qui retourne la liste des opérations possibles à partir d'un état, associées aux états d'arrivée et aux coûts de ces opérations.
- **('a -> int)** la fonction heuristique qui donne une valeur numérique entière à tout état.
- **'a** l'état initial de la recherche (de type quelconque)

Etant donné la description d'un problème au moyen des 4 paramètres qui le caractérisent, cette fonction recherche devra renvoyer un chemin-solution ou, si c'est le cas, signaler que le problème n'admet pas de solution.

La fonction recherche appliquée au graphe g1 depuis son état initial A sera appelée ainsi :

```
#recherche     estBut1
               opPoss1
               hEtat1
               "A";;
```

et donnera comme résultat le chemin `['b' ;'g' ;'h']` de coût `11`.

**4.** Modifiez votre implémentation afin qu'elle affiche, en fin de recherche, le nombre d'itérations et le nombre de nœuds créés. Pour conserver la même signature pour la fonction, ces statistiques ne sont pas à retourner, mais seulement à afficher à la fin de la recherche, avant de retourner le chemin solution.