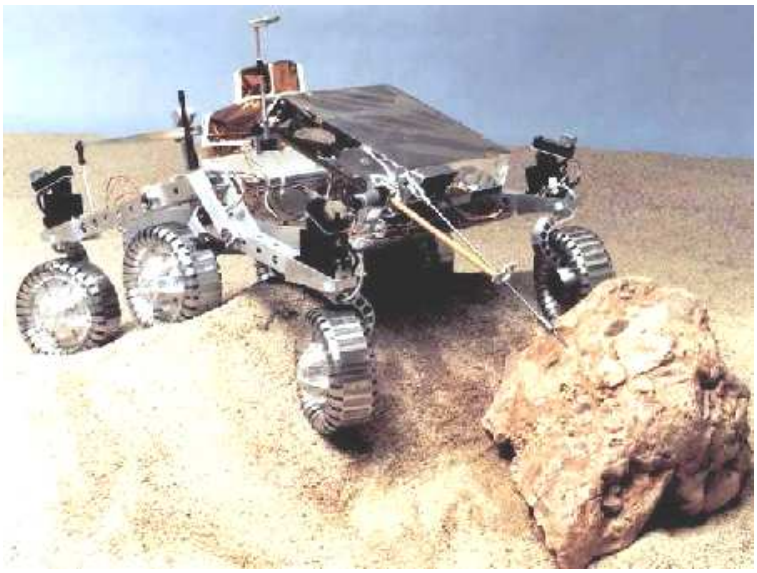
MÉRÉ Aurélien
FIIFO1

# AMC Pathfinder



# Sommaire

# Préambule

Bienvenue sur le manuel algorithmique d'*AMC PathFinder*. Ce logiciel permet de déplacer un robot dans un espace prédéfini, de manière manuelle avec des commandes basiques telles que « Tourner » ou « Avancer », mais également de manière automatique avec le calcul des différents chemins possibles pour notre robot.

Ce manuel présente tous les algorithmes de notre logiciel, à l'exception de ceux dépendant du langage de programmation qui sera employé lors de l'implémentation. Il s'agit notamment de certaines routines d'affichage.

Un exemple concret permet de voir progressivement les différentes capacités du logiciel et de mieux les exploiter par la suite. Les algorithmes sont écrits de la manière la plus concise et claire possible afin de faciliter le portage dans un langage de programmation, ainsi que sa relecture.

Les captures d'écran de labyrinthes que vous pourrez examiner durant votre lecture de ce manuel sont extraites d'un logiciel d'A. Méré, « Générateur de labyrinthes ». Il reprend la plupart des algorithmes présents ici et servira de base à l'implémentation de tous les algorithmes du logiciel.

# Modélisation de l'espace

Nous sommes en présence d'un labyrinthe constitué de M*N cases. Chaque case de ce labyrinthe possède quatre propriétés : il est possible de se déplacer ou non sur les cases annexes. Chaque case est donc spécifiée comme suit :

```
Type Case = Enreg( Haut : Booléen,
                   Bas : Booléen,
                   Droit : Booléen,
                   Gauche : Booléen)
```

Chaque booléen permet de savoir s'il est possible d'accéder à une case à partir de celle que l'on est en train d'adresser.

Le labyrinthe est constitué de M colonnes et N lignes. Le labyrinthe est donc adressé à l'aide d'un tableau de M*N cases :

```
Type Labyrinthe = Tableau (1 à M*N) de Case
```

**Dans la suite du rapport, nous utiliserons un labyrinthe LABY défini en variable globale, afin d'alléger le programme et faciliter la compréhension.**

# Modélisation des cases $\delta$, $\alpha$

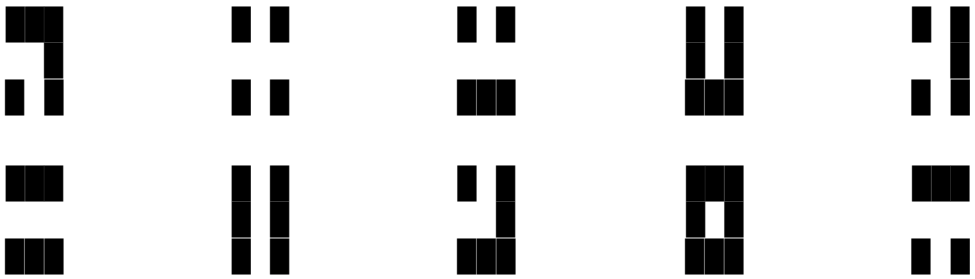
Ces différentes variables sont stockées globalement dans le programme. Les deux cases $\delta$, $\alpha$ ne sont caractérisées que par une position dans le labyrinthe. En conséquence, les variables Alpha et Delta ne contiennent qu'un entier caractérisant leur position dans le labyrinthe.

```
Type Position : Entier compris entre 1 et M*N ;

Variable Alpha : Position ;
         Delta : Position ;
```

L'utilisation d'un type Position permet d'avoir une vérification automatique de la validité des arguments passés dans une procédure, ceux-ci étant relatifs à un objet dans le labyrinthe. D'autres objets présents dans le labyrinthe peuvent être définis à l'aide de ce type personnalisé.

# Interface en mode texte

Une case du labyrinthe est représentée en mode texte par une matrice de 3 colonnes et 3 lignes. Suivant les différentes ouvertures d'une case, cette matrice sera légèrement différente. Voici quelques exemples :



Nous supposerons que le langage de programmation dans lequel le programme sera implanté est muni de routines portant sur le mode texte, et notamment la possibilité de choisir la ligne et la colonne en cours. Nous nommerons ses procédures SetLigne(l), SetCol(c) et SetPosition(l, c) dans la suite du rapport, chacune définissant respectivement la ligne, la colonne et le couple (ligne, colonne) en cours en mode texte. L'algorithme d'affichage du labyrinthe est défini comme suit :

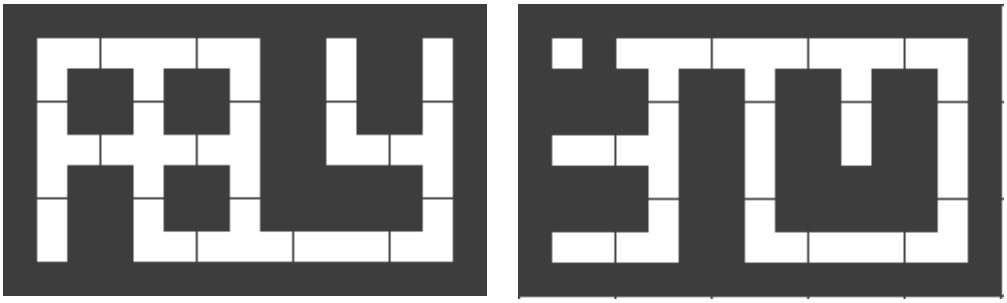
```
Procedure AfficheLaby (L: Labyrinthe {E])
Variable i : entier;
Début
        Pour i de 1 à (M*N) faire
              AfficheCase (i, L(i));
        Fin Pour;
Fin;
```

Cette procédure affiche toutes les cases du labyrinthe, une par une. Elle appelle pour ce faire la procédure AfficheCase qui affiche une unique case, en connaissant uniquement sa position et ses caractéristiques. La complexité de la procédure AfficheCase est unitaire ; en conséquence, la complexité de la procédure AfficheLaby est de l'ordre de (M*N).

Le code de la procédure AfficheCase est présenté ci-après. Il ne contient qu'un certain nombre de tests pour connaître les différents endroits où il est possible de se rendre. Si un mur est trouvé, on affiche le caractère associé au mur, sinon on laisse un blanc.

```
Procedure AfficheCase (numCase : position; tmpCase : case)

variable l, c: entier;

Début

        ■ On calcule la position sur l'écran de la case

        ■ En sachant qu'une matrice a une taille de 3*3

        l:= (numCase div M) * 3;
        c:= (numCase mod M) * 3;

        ■ Affichage de la première ligne (mur du haut)

        setPosition(l, c);

        Si (tmpCase(numCase).haut) alors
             ecrire ("# #");
        sinon
             ecrire ("###");
        fin si;

        ■ Affichage de la seconde ligne
          (murs gauches et droits)

        setPosition(l+1, c);

        Si (non(tmpCase(numCase).gauche))  alors
             ecrire ("#");
        fin si;

        SetCol(c+2);

        si (non(tmpCase(numCase).droit)) alors
             ecrire ("#");
        fin si;

        ■ Affichage de la troisième ligne

        setPosition(l+2, c);

        Si (tmpCase(numCase).bas) alors
             ecrire ("# #");
        sinon
             ecrire ("###");
        fin si;

fin;
```

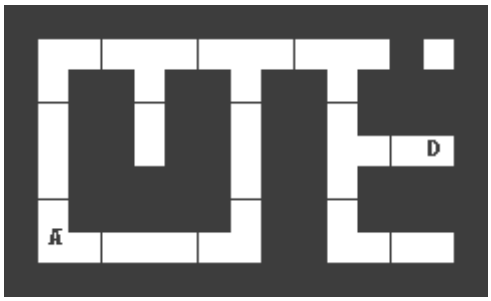
Considérons un labyrinthe de 5 cases sur 3. La combinaison des deux algorithmes précédents permet d'afficher le labyrinthe (les données sont sélectionnées au hasard) de cette manière :



Il est également nécessaire tout au long du programme d'afficher différents éléments dans le labyrinthe, comme par exemple le robot, le point de départ, d'arrivée, du carburant, etc… En mode texte, nous ne pouvons afficher qu'un unique caractère au milieu d'une case. L'algorithme AfficheObjet présenté ci-dessous prend en entrée la position de l'objet à afficher, et le caractère ASCII à afficher :

```
Procedure AfficheObjet (pos: Position; car: Caractere)
Variable l, c: entier;
Début

   ■  On calcule la position de l'objet à l'écran

      l:=(pos div M)*3 + 1;
      c:=(pos mod M)*3 + 1;

   ■  On définit la position et on écrit car

      SetPosition (l, c);
      Ecrire (car);
Fin;
```

Exemple avec les caractères D (départ) et A (arrivée) :



# Modélisation du robot

## *1 ) Le type Robot*

Notre robot est caractérisé par sa position dans le labyrinthe, mais également par sa direction. Nous mettrons les quatre directions possibles dans des constantes pour faciliter la compréhension. Par ailleurs, notre robot ne peut tourner qu'à gauche, aussi nous utiliserons le sens trigonométrique pour l'incrémentation des constantes :

```
Const DIR_DROIT    := 0 ;
Const DIR_HAUT     := 1 ;
Const DIR_GAUCHE   := 2 ;
Const DIR_BAS      := 3 ;
```

Nous définissons ensuite un type énuméré *Direction* à partir de ces différentes constantes :

```
Type Direction = {DIR_DROIT ; DIR_HAUT ; DIR_GAUCHE ; DIR_BAS}
```

Nous pouvons alors définir notre robot avec ses deux caractéristiques (direction et position) comme suit :

```
Type Robot = Enreg (pos : Position ; dir : Direction) ;
```

**Dans la suite de ce rapport, nous utiliserons un robot RBT défini comme variable globale, afin d'alléger le programme et faciliter la compréhension**.

## *2 ) Routines relatives au robot*

Différentes routines permettent d'utiliser le robot. Nous allons commencer par définir une fonction *Avance* et une procédure *Tourne*.

La fonction *Avance* permet de faire avancer le robot dans la direction courante, si cela est possible. Si l'action est réalisée, la fonction renvoie la valeur *Vraie*. Dans le cas où l'on se trouve face à un mur, la fonction renvoie la valeur *Faux*. La procédure *Tourne* a pour simple but de changer la direction du robot. Il suffit dans notre cas d'incrémenter la valeur de direction, tout en veillant à ce qu'elle ne dépasse pas les 4 directions possibles.

Le code de ces deux routines est présenté ci-après :

```
Fonction Avance : booléen
Variable tmp : booléen ;
Début

        Suivant RBT.Direction faire
        Cas DIR_DROITE :
              tmp := LABY(RBT.Position).droit ;
              Si tmp alors RBT.Position := RBT.Position + 1 ;

        Cas DIR_HAUT :
              tmp := LABY(RBT.Position).haut ;
              Si tmp alors RBT.Position := RBT.Position - M ;

        Cas DIR_GAUCHE :
              tmp := LABY(RBT.Position).gauche ;
              Si tmp alors RBT.Position := RBT.Position - 1 ;

        Cas DIR_BAS :
              tmp := LABY(RBT.Position).bas ;
              Si tmp alors RBT.Position := RBT.Position + M ;

        Fin ;

        Retourne tmp ;

Fin ;
```

En fonction de la direction du robot, on vérifie qu'il est possible d'avancer en testant la présence d'un mur dans le labyrinthe à la direction indiquée. Si le déplacement est possible, on change la position du robot dans le labyrinthe.

```
Procedure Tourne
Début

    RBT.Direction := Direction.Suivant'(RBT.Direction) ;

Fin ;
```

La procédure Tourne se contente de changer la direction du robot en demandant l'élément suivant du type énuméré Direction. Ainsi, si la direction actuelle est DIR_DROITE, l'élément suivant dans le type Direction étant DIR_HAUT, la nouvelle direction est DIR_HAUT. Lorsque l'on demande l'élément suivant de DIR_BAS, la circularité des types énumérés permet d'avoir une allocation automatique de DIR_DROITE, ce qui apporte un confort supplémentaire de programmation.

# Initialisation de l'espace

Notre espace est initialisé à l'aide de deux routines : la procédure InitEspace qui créé tous les murs et les chemins au hasard, et la fonction randBool qui renvoie un booléen au hasard. Etudions cette dernière fonction, plus simple :

```
Fonction RandBool : booléen
Début

       Retourne (Rand < 0,65)

Fin ;
```

RandBool exploite la fonction rand renvoyant un nombre entre 0 et 1. Si ce nombre est supérieur à 0.65, on retourne faux et vrai dans l'autre cas. Les dés sont légèrement pipés, mais ceci dans le but d'augmenter le nombre de chemins possibles dans le labyrinthe, pour tester plus efficacement les algorithmes de recherche et de parcours.

La fonction InitEspace se présente comme suit :

```
procedure InitEspace
variable i, j, tmpPos : entier

Début

      Pour j de 0 à N-1
            Pour i de 0 à M-1

                  ■ On calcule la position de la case en cours

                  tmpPos = (i + j*M);

                  ■ Si on est au bord supérieur du labyrinthe, on
                  place un mur. Sinon on place le chemin défini
                  par la case supérieure

                  Si (j=0) alors LABY (TmpPos).haut := faux;
                  sinon LABY (TmpPos).haut := LABY(TmpPos - M).bas;
                  fin si;

                  ■ Si on est au bord inférieur du labyrinthe, on
                  place un mur. Sinon on choisit un chemin au
                  hasard

                  Si (j=N-1) alors LABY(TmpPos).bas := faux;
                  sinon LABY(TmpPos).bas := RandBool;
                  Fin si;
```

- Si on est au bord droit du labyrinthe, on place un mur. Sinon on choisit un chemin au hasard

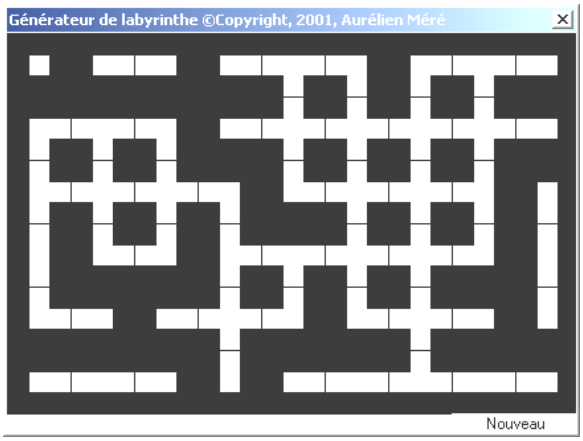
```
Si (i=M-1) alors LABY(TmpPos).droite := faux
sinon LABY(TmpPos).droite := RandBool;
Fin Si;
```

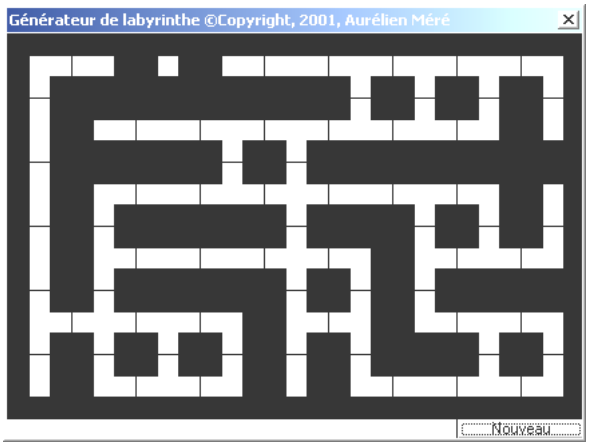
- Si on est au bord gauche du labyrinthe, on place un mur. Sinon on définit le chemin définit par la case précédente

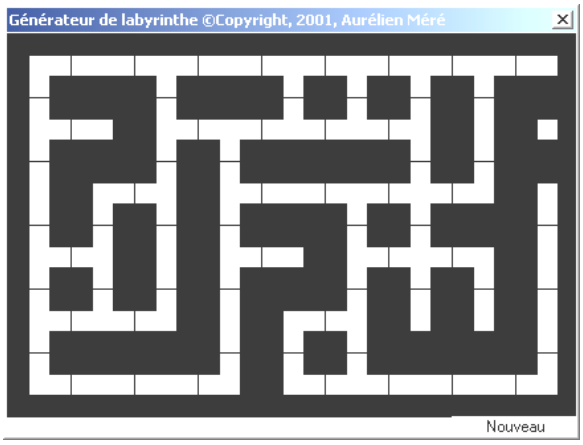
```
Si (i=0) alors LABY(TmpPos).gauche := faux;
sinon LABY(TmpPos).gauche := LABY(TmpPos -1).droite;
fin si;
```

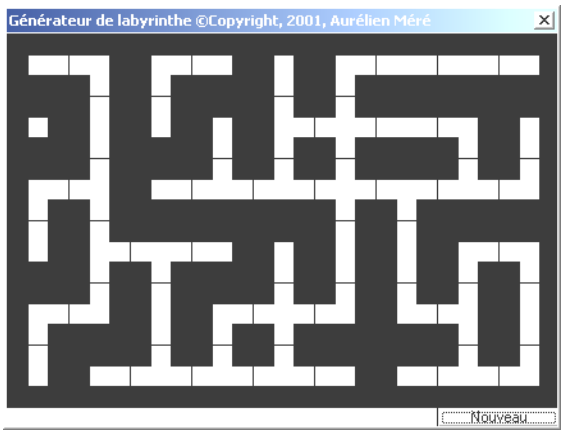
```
        fin Pour;
    fin Pour;
fin;
```

A l'aide de cet algorithme, nous pouvons obtenir des labyrinthes cohérents. Voici un certain nombre de labyrinthes obtenus à l'aide de cet algorithme :



# Arbre des chemins

L'arbre des chemins permet de trouver tous les chemins possibles à partir d'un point de départ pour atteindre une case. Cet algorithme utilise un tableau de booléens de la taille du labyrinthe indiquant si une case a déjà été visitée ou non. Ce tableau est défini de manière globale, de cette façon :

```
BoolTab : tableau (0 à M*N) de booléens
```

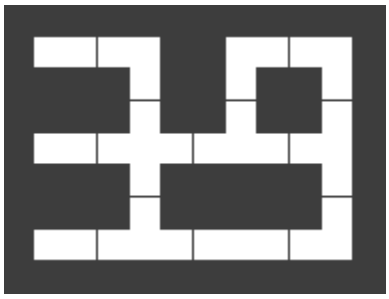
On considère que toutes les valeurs sont égales à FAUX lorsque le programme démarre. L'arbre des chemins est un arbre 4-aires défini de la façon suivante :

```
Type Arbre = Enreg(haut:ptr(Arbre),
                   bas:ptr(Arbre),
                   gauche:ptr(Arbre),
                   droite:ptr(Arbre)
                   position : entier);
```

On constate qu'il stocke la position du nœud dans le labyrinthe, ce qui facilite les recherches. Notre algorithme va donc créer un arbre de ce type. Voici son corps ; Sa complexité est au pire d'ordre n (nombre de nœuds)

```
Procedure CreeArbre (a : arbre {E/S}, pos: entier)
Début
Si BoolTab (Pos) = Faux alors

       BoolTab(Pos) := Vrai ;

       a :=allouer (arbre) ;
       a^.haut := NULL ; a^.bas := NULL ;
       a^.gauche := NULL ; a^.droite := NULL ;

       si LABY(pos).haut alors CreeArbre(a^.haut, pos – M) ;
       si LABY(pos).gauche alors CreeArbre(a^.gauche, pos – 1) ;
       si LABY(pos).bas alors CreeArbre(a^.bas, pos + M) ;
       si LABY(pos).droite alors CreeArbre(a^.droite, pos + 1) ;
       BoolTab(Pos) := Faux ;

Fin Si ;
Fin ;
```

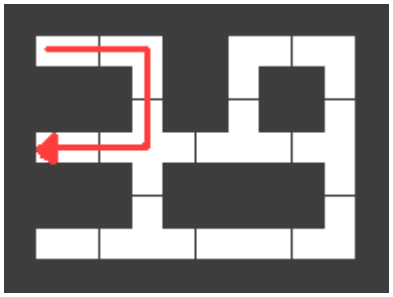
Voici ce que donne cet algorithme sur le labyrinthe suivant, en partant de la case en haut à gauche (position = 0).



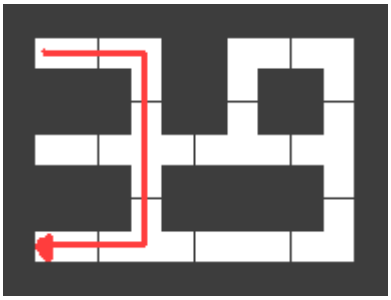
L'arbre présent en page suivante présente le résultat de l'appel à l'algorithme précédent.

On constate que 7 chemins peuvent être réalisés avec ce labyrinthe. Voici ces 7 chemins :

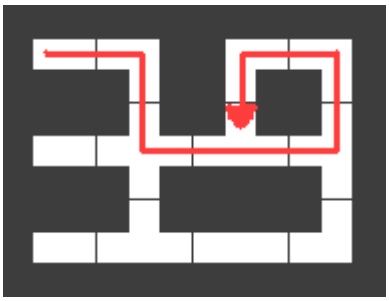
1er chemin :



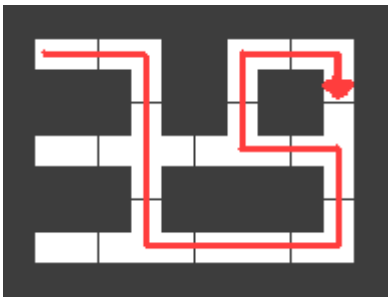
2ème chemin :



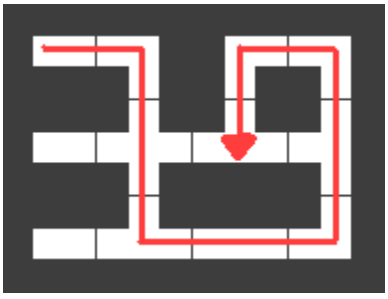
3ème chemin :



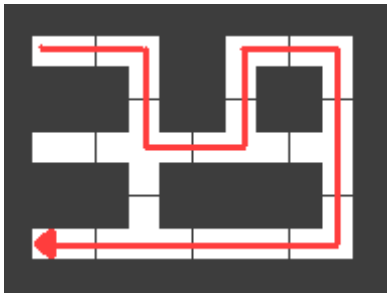
4ème chemin :



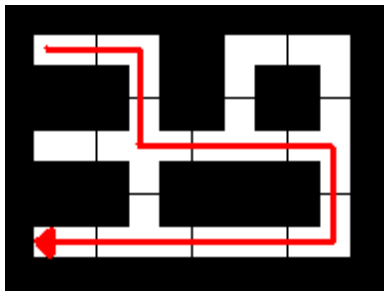
5ème chemin :



6ème chemin :



7ème chemin :




droit
bas
gauche
①
bas
droite
gauche
②
droite
droite
haut
droite
droite
bas
haut
gauche
③
haut
droite
bas
bas
bas
gauche
gauche
gauche
gauche
gauche
gauche
⑦
⑥
gauche
haut
haut
gauche
droite
bas
④
⑤

# Recherche d'un chemin allant de δ à α.

## *Recherche d'un chemin*

Il s'agit d'un algorithme de recherche dans un arbre. Le résultat est placé dans une pile, sous la forme d'une série de positions. On considère que cette pile est définie en variable globale et que les routines AjouterPile et RetirerPile ont été intégrées au langage de description.

L'algorithme prend en entrée l'arbre des chemins. On supposera que la racine de l'arbre des chemins correspond à la case de départ δ .

```
Fonction ChercheChemin (a : arbre {E}) : booléen
Début
Si (a=null) alors retourne faux ;
Si (pos = alpha) alors retourne vrai ;
Si ChercheChemin(a^.haut) alors AjouterPile (a^.haut^.pos);
Si ChercheChemin(a^.bas) alors AjouterPile (a^.bas^.pos);
Si ChercheChemin(a^.gauche) alors AjouterPile (a^.gauche^.pos) ;
Si ChercheChemin(a^.droite) alors AjouterPile (a^.droite^.pos) ;

Fin ;
```

La pile définie préalablement contient alors la liste de toutes les positions menant de la case delta à la case alpha. A noter que si l'arbre envoyé est vide, ou si aucun chemin n'a pu être trouvé, la fonction renvoie la valeur FAUX. En conséquence, la pile n'a pas été modifiée et il est donc inutile d'y mener des opérations. La complexité au pire de cet algorithme sera d'ordre n (nombre de nœuds ).

Le contenu de la pile sera le suivant avec l'exemple ci-contre (les nombres dans l'arbre correspondent à l'entier de position dans le labyrinthe) :

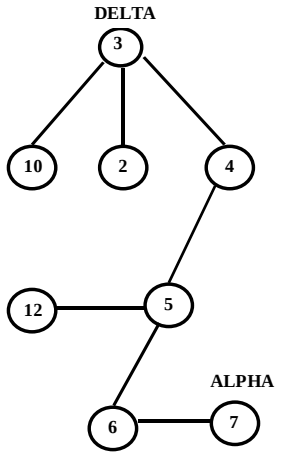
**SOMMET**

4

5

6

7



## *Recherche et affichage du chemin trouvé*

Voici une procédure qui permet de rechercher un chemin menant de delta à alpha, et au cas où cette recherche est satisfaisante, affiche le chemin qui y mène à l'aide des routines que nous avons définies précédemment. Cette procédure est notée Cherche_et_affiche. Elle prend en entrée l'arbre des chemins, la racine pointant sur delta. Les autres variables sont définies de manière globale.

```
Procedure Cherche_et_affiche (a : arbre {E})
Variables : pos : position ;
Début

      If ChercheChemin (a) alors
            -- On a trouvé un chemin valide

            Tant Que (pos <> delta) faire
                  pos := Depiler;

                  ■    On affiche la case où l'on se déplace

                  AfficheCase (pos, « * ») ;

             Fin Tant Que

     Sinon
            Ecrire (« Aucun chemin trouvé ») ;

     Fin Si ;
Fin ;
```

## *Interaction avec l'utilisateur*

L'utilisateur peut définir la position des cases alpha et delta, par simple lecture de nombre, mais également manipuler le robot par lui même comme nous l'avons vu précédemment. La combinaison des différentes routines précédentes permet d'obtenir une interaction maximale avec l'utilisateur. Cette routine appelle la fonction DirCar qui renvoie un caractère ASCII représentant une flèche dans la direction passée en argument. Elle est spécifique au langage de programmation, aussi elle ne sera pas détaillée ici.

```
Procédure Utilisateur
Variable    C :caractère ,
            TmpBool : booléen,
            termine : booléen,
            a : arbre ;
Début
      Termine := faux ;

      ■ On initialise l'espace
      ■ et on calcule l'arbre des chemins
      InitEspace ;
      CreeArbre (a, delta) ;

      Tant Que Non(Termine) faire
            ■ On affiche l'interface
            AfficheLaby ;
            AfficheObjet (alpha, « A ») ;
            AfficheObjet (delta, « D ») ;
            AfficheObjet (RBT.position, dirCar(RBT.direction)) ;

            ■ On attend la saisie de l'utilisateur
            ■ Et on la traite
            Lire (c) ;
            Si c= « T » alors tourne ;   -- Tourne
            Si c= « A »  alors           -- Avance
                  TmpBool := avance ;
                  Si non(TmpBool) alors ecrire (« Impossible ! ») ;
            Fin Si ;
            Si RBT.position = arrivee alors
                  Ecrire (« Vous avez réussi ! ») ;
                  Termine := vrai ;
            Fin Si ;
            Si c= « Q » alors termine := vrai ; -- Quitter
            Si c= « U » alors      -- Traitement automatique
                  Cherche_et_affiche(a) ;
            Fin Si
      Fin Tant Que
Fin ;
```

# Gestion du carburant

A chaque déplacement, le robot perd du carburant. La consommation est déterminée dans une fonction à part, car celle-ci peut ne pas être linéaire, ce qui facilite l'évolutivité du programme. Il est nécessaire de faire une unique modification sur la procédure de recherche des chemins. En effet, sur un grand labyrinthe, le carburant peut s'épuiser rapidement, ce qui réduit considérablement la taille de certaines branches. On ajoute donc un paramètre en entrée de la procédure, que l'on nommera c, de type Carburant. Rappelons que le type Carburant est défini par :

```
Type Carburant = Entier signé;
```

L'unique modification à porter est la présence d'une consommation à chaque déplacement entre différents nœuds, ce qui implique qu'à chaque mouvement dans une branche, le carburant décroît d'une certaine quantité, déterminée par la fonction consommation, celle-ci éventuellement dépendante du carburant c. On vérifie que le carburant est toujours positif, à savoir que le déplacement est réellement possible.

```
Procedure CreeArbre (a : arbre {E/S}, pos: entier, c :carburant)
tmpC : carburant ;
Début
Si (BoolTab (Pos) = Faux) et (c > 0) alors

      TmpC := TmpC – Consommation(c) ;

      BoolTab(Pos) := Vrai ;

      a :=allouer (arbre) ;
      a^.haut := NULL ; a^.bas := NULL ;
      a^.gauche := NULL ; a^.droite := NULL ;

      si LABY(pos).haut alors CreeArbre(a^.haut, pos – M, tmpC) ;
      si LABY(pos).gauche alors CreeArbre(a^.gauche, pos–1, tmpC) ;
      si LABY(pos).bas alors CreeArbre(a^.bas, pos + M, tmpC) ;
      si LABY(pos).droite alors CreeArbre(a^.droite, pos+1, tmpC) ;
      BoolTab(Pos) := Faux ;

Fin Si ;
Fin ;
```

## *Fonction Consommation*

La fonction Consommation peut être très simple : si l'on désire baisser le carburant de X litres, la fonction ressemble à ceci :

```
Fonction Consommation (c : carburant) : carburant ;
Début
      Retourne X ;
Fin ;
```

La fonction Consommation peut également renvoyer une valeur dépendante du carburant transporté, à savoir du poids du robot. En considérant qu'il s'agit d'une application linéaire A.C + B (où C est le carburant transporté), on obtient la fonction suivante :

```
Fonction Consommation (c : carburant) : carburant ;
Début
      Retourne (A * c + B) ;
Fin ;
```

## *Modifications dans le programme*

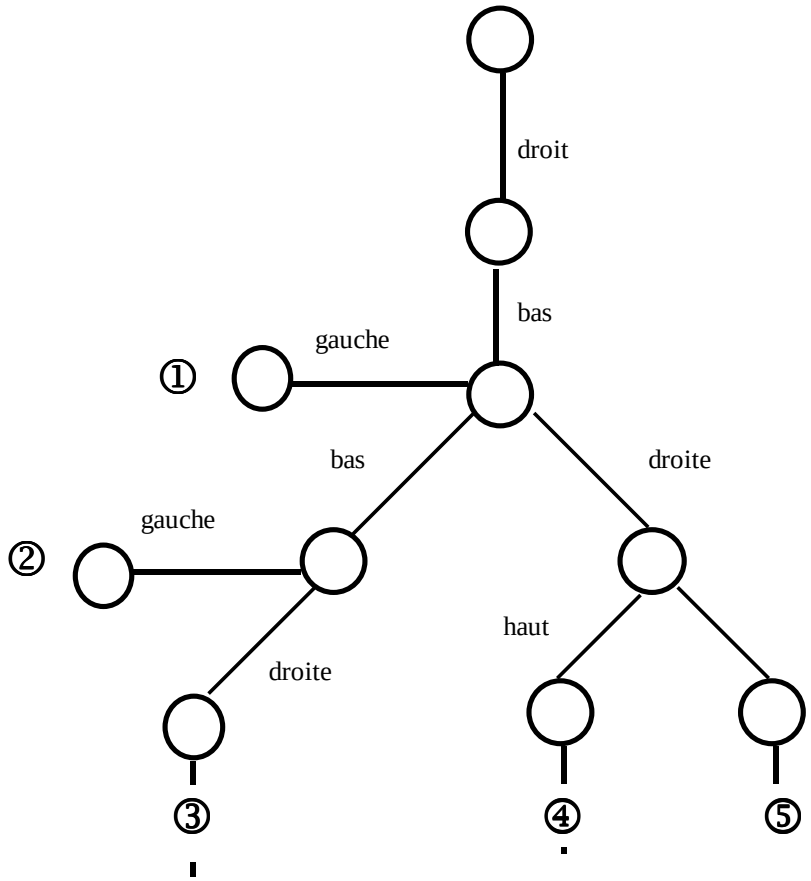
La seule modification à faire dans l'ensemble du programme se trouve dans l'appel à la procédure CreeArbre, étant donné qu'elle réclame un nouvel argument, qui est la quantité de carburant initial. Si l'on considère qu'il y a L litres dans le réservoir au départ, on appellera la procédure de cette manière :

```
CreeArbre (a, delta, L) ;
```

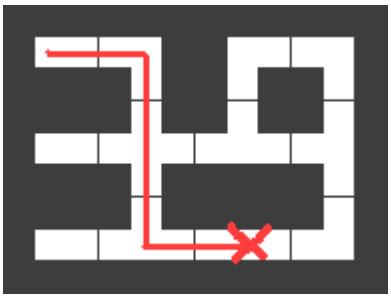
Il faut noter que l'utilisateur est parfaitement en mesure de rentrer, lors de l'initialisation du programme, la valeur du carburant L, ainsi que les paramètres de consommation.

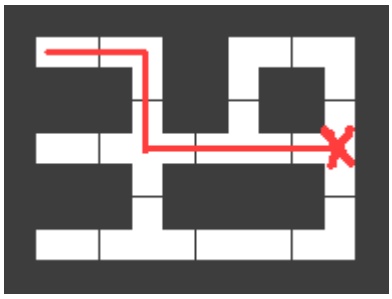
Pour notre test, nous considérerons que la consommation est de 1 à chaque déplacement, et que le carburant initial est de 4. Si l'on utilise la nouvelle procédure CreeArbre avec même labyrinthe que précédemment, notre arbre des chemins se trouve particulièrement rétréci. Ceci est dû au fait que la

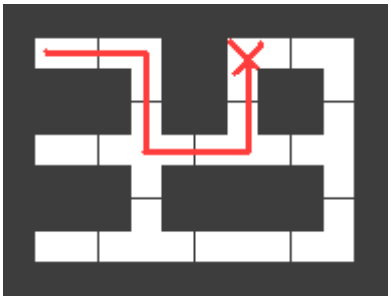
hauteur maximale de l'arbre sera la quantité initiale de carburant (dans notre exemple). Aussi, en reprenant le même exemple que précédemment, on trouve l'arbre suivant :




On constate qu'il n'y a plus que cinq chemins possibles. Les deux premiers chemins sont identiques.







La croix symbolise une panne d'essence.

Notre exemple montre que certaines cases ne sont plus accessibles désormais, et notamment celles des deux coins droits. Si l'on suppose que la case Alpha se trouve ici, il n'est plus possible de réaliser le parcours entier. La fonction ChercheChemin renverra alors la valeur Faux. On constate donc que les changements effectués ont un réel impact sur la création des chemins.

Il faut par ailleurs noter que si une valeur incorrecte de carburant est passée à la procédure (inférieure ou égale à 0), la procédure considère que le robot n'a plus de carburant et qu'il ne peut continuer. Celle-ci renvoie donc un arbre vide.

# Présence de carburant dans le labyrinthe

## *Modification des cases*

Nous désirons maintenant ajouter du carburant dans le labyrinthe. Cela nécessite d'ajouter une information dans chacune des cases du labyrinthe, à savoir la quantité de carburant qu'elle contient. Le nouveau type Case est alors défini comme ça :

```
Type Case = Enreg( Haut : Booléen,
                   Bas : Booléen,
                   Droit : Booléen,
                   Gauche : Booléen,
                   Carbu : Carburant)
```

Une  case ne portant pas de carburant a pour valeur Carbu = 0.

## *Initialisation de l'espace*

L'initialisation de l'espace doit donc prendre en compte ce nouvel élément. Dans l'initialisation de chaque case, il suffit de rajouter les lignes suivantes :

```
Si (rand > 0,95) alors LABY(tmpPos).Carbu := entier(Rand * L) ;
Sinon LABY(tmpPos).Carbu := 0 ;
Fin Si ;
```

On considère que la présence de carburant sur une case doit être assez exceptionnelle. Dans l'exemple précédent, il y a une chance sur 20 pour qu'une case contienne du carburant. Si c'est le cas, la quantité de carburant est déterminée au hasard, en fonction de la valeur initiale de carburant du robot.

## *Création de l'arbre des chemins*

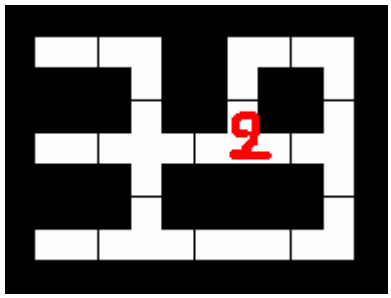
Il est ensuite nécessaire de prendre en compte ce détail dans la procédure de création de l'arbre des chemins, car la présence de carburant permet d'étendre l'arbre sur un maximum d'espace. Il est nécessaire de modifier une seule ligne de code, afin d'obtenir la procédure suivante :

```
Procedure CreeArbre (a : arbre {E/S}, pos: entier, c :carburant)
tmpC : carburant ;
Début
Si (BoolTab (Pos) = Faux) et (c > 0) alors
```

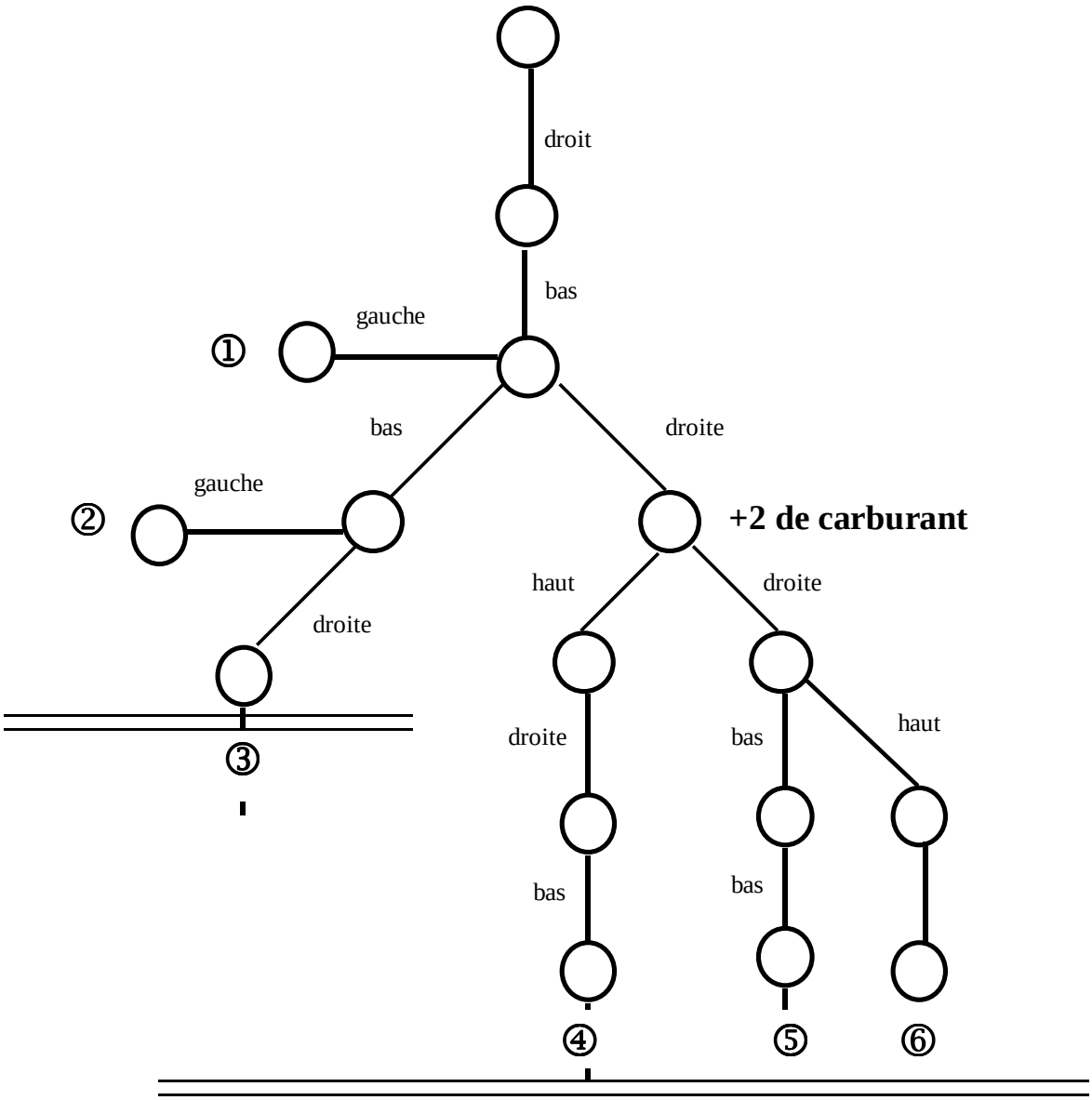
- **On élimine la consommation mais on ajoute également le carburant présent dans la case**

```
TmpC := TmpC – Consommation(c) + LABY(pos).carbu;

BoolTab(Pos) := Vrai ;

a :=allouer (arbre) ;
a^.haut := NULL ; a^.bas := NULL ;
a^.gauche := NULL ; a^.droite := NULL ;

si LABY(pos).haut alors CreeArbre(a^.haut, pos – M, tmpC) ;
si LABY(pos).gauche alors CreeArbre(a^.gauche, pos–1, tmpC) ;
si LABY(pos).bas alors CreeArbre(a^.bas, pos + M, tmpC) ;
si LABY(pos).droite alors CreeArbre(a^.droite, pos+1, tmpC) ;
BoolTab(Pos) := Faux ;

  Fin Si ;
Fin ;
```

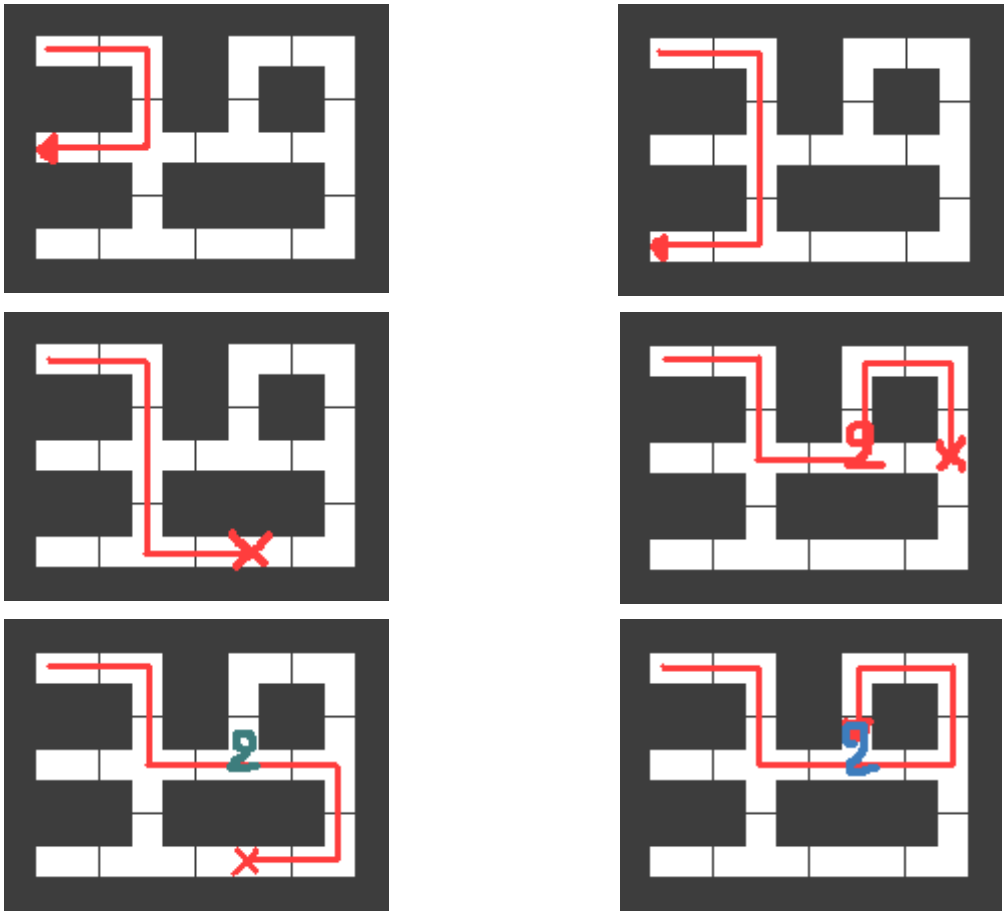
Prenons en compte ces nouvelles données dans notre exemple. Nous allons partir de l'exemple précédent en rajoutant une case contenant une extension de carburant de 2 unités :




La procédure CreeArbre nous donne un nouvel arbre des chemins :



Ce nouvel arbre fait apparaître un nouveau chemin et étend également deux chemins existants, ce qui permet de couvrir l'intégralité du labyrinthe. Il existe donc au moins un chemin permettant de relier delta et alpha. On le voit nettement avec les chemins représentés directement :



Toutes les cases de l'espace sont désormais accessibles. Ceci n'est qu'un exemple, et suivant les cas, les différents chemins peuvent varier. Mais le nombre de chemins possibles dans un labyrinthe de plus grande taille est particulièrement difficile à détailler dans un tel rapport, et nous préférons nous en tenir à des exemples simples mais qui montrent efficacement la puissance des algorithmes.