Compte-rendu du TD n°4

UV IA01

I – Ce qui a été fait en TD

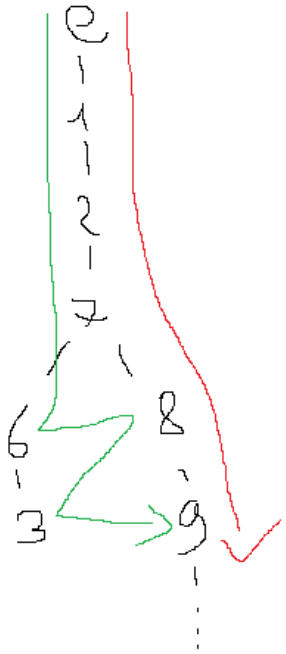
A – Définition du problème

Il y a donc 22 états possibles : *e, 1,2, … , 20,s*.

On peut représenter ce labyrinthe comme cela : (etat successeurspossible1 successeurpossible2 …).

Ce qui, pour notre labyrinthe, donne ceci :

```
(e 1)
(1 e 2)
(2 1 7)
(3 6)
(4 5)
(5 12 4)
(6 3 7)
(7 2 6 8)
(8 7 9)
(9 8 10)
(10 9 11 15)
(11 10 12 14)
(12 11 5)
(13 20)
(14 11)
(15 10 16)
(16 15 17)
(17 16 18)
(18 17 19)
(19 18 20)
(20 13 s)
(s 20)
```



Nous pouvons aussi représenter un labyrinthe sous forme d'arbre comme ci-dessus.

Nous pouvons le parcourir de plusieurs manières :

En rouge : parcours en profondeur

En vert : parcours en largeur

Dans notre cas, nous allons le parcourir en <u>profondeur</u>.

B – Fonctions de service

```
;; récupère les successeurs d'un état du labyrinthe
(defun next (laby state)
      (cdr (assoc state laby))
)
```

Cette fonction nous permet de récupérer les successeurs d'un état.

```
;; retourne les éléments suivants valides
(defun validSuccessors (laby state path)
      (let ((result ()))
      (dolist (x (next laby state) result)
            (if (not (member x path))
                  (push x result)))))
```

Quant à elle, cette fonction retourne les successeurs valides, c'est-à-dire ceux qui ne sont pas déjà présents dans le chemin fournit en paramètre.

Pour les tests de ces fonctions, se référer au test de la fonction explore.

II – Fonction explore

Nous sommes ici partis sur un algorithme complétement différent de celui qui nous a été proposé en cours.

En effet, nous avons fait le choix d'afficher tous les chemins possibles pour arriver à la sortie, plutôt qu'un seul.

```
(defun explore (labyrinth state goal currentPath)
      ; on récupère la liste des successeurs
      (setq successors (validSuccessors labyrinth state currentPath))
      ;on ajoute le noeud actuel au chemin parcouru
      (push state currentPath)
      ;;si on a trouvé la sortie
      (if (equal state goal)
      ; on est à la sortie : on affiche le chemin parcouru
            (print (reverse currentPath))
            ; sinon, pour tous les noeuds suivants valides
            (dolist (successor successors nil)
            ;on explore via récursivité
                  (explore labyrinth successor goal currentPath))))
```

Test de cette fonction avec le labyrinthe original (un chemin de sortie unique) :

```
(setq laby '(
      (e 1) (1 e 2) (2 1 7)(3 6)(4 5) (5 12 4)(6 3 7)(7 2 6 8)(8 7 9)
      (9 8 10) (10 9 11 15)(11 10 12 14)(12 11 5)(13 20)(14 11)
      (15 10 16)(16 15 17)(17 16 18)(18 17 19)(19 18 20)(20 13 s)(s 20)))
```

```
CG-USER(37): (explore laby 'e 's   ())

(E 1 2 7 8 9 10 15 16 17 18 19 20 S)
NIL
```

Test de la fonction avec un labyrinthe a deux chemins possibles. Pour cela, nous rajoutons la possibilité d'accéder à la sortie en passant de 14 vers 13, ce qui crée deux chemins possibles pour accéder à la sortie :

```
(setq laby '(
      (e 1)(1 e 2) (2 1 7)(3 6)(4 5)(5 12 4)(6 3 7)(7 2 6 8) (8 7 9)(9 8 10)
      (10 9 11 15)(11 10 12 14) (12 11 5)(13 20)(14 11 13)(15 10 16)
      (16 15 17)(17 16 18)(18 17 19)(19 18 20)(20 13 s)(s 20)))
```

```
CG-USER(39): (explore laby 'e 's   ())

(E 1 2 7 8 9 10 15 16 17 18 19 20 S)
(E 1 2 7 8 9 10 11 14 13 20 S)
NIL
```

Conclusion

Nous obtenons bien dans les deux cas les résultats escomptés au départ.