Automne 11

Rapport de TP

Marie Julien
Johan Medioni

Projet n°2 : Recherche dans un espace d'états.

Julien Marie GI01
Medioni Johan TC03

Dans ce TP, on effectue une recherche dans un espace d'états. Le problème est le suivant : on veut avoir 6 flèches dans le même sens. L'état initial est six flèches pointant vers le haut ou vers le bas. On associe au sens bas la valeur 0, au sens haut la valeur 1.

La représentation d'un état est une liste de 0 et de 1, un état étant alors la liste du sens de ses six flèches. Par exemple, l'état initial ici est (0 0 0 1 0 1) et l'état final (1 1 1 1 1 1).

Le coup permis pour retourner les flèches et le suivant : on doit toujours retourner simultanément deux flèches conjointes.

Table des matières

Julien Marie GI01
Medioni Johan TC03

I. Etablissement de fonctions de services

a) Réponse aux questions

1.

(defun newstate (etat pos) (let ((copie ())) (setf copie (copy-list etat)) (if (= 0 (nth pos copie)) (setf (nth pos copie) 1) (setf (nth pos copie) 0)) (if (= 0 (nth (+ 1 pos) copie)) (setf (nth (+ 1 pos) copie) 1) (setf (nth (+ 1 pos) copie) 0)) copie))	Fonction qui renvoie l'état obtenu en inversant les positions pos et pos+1 On crée une variable locale qui est une copie de l'état passé en paramètre On inverse pos et pos+1 On retourne la copie modifiée.

2.

(defun successors (state) (let ((liste '(O))) (setf liste (copy-list '(0))) (dolist (i compteur) (nconc liste (list (newstate state i)))) (rest liste)))	Fonction qui renvoie la liste des successeurs d'un état passé en paramètre. A l'aide de la variable globale compteur on crée une liste contenant (newstate i), i allant de 0 à 4. Ainsi on retourne une liste de sous-liste contenant chacune un état successeur possible.

Exemple d'exécution :

```
(successors '(0 0 0 1 0 1))
((1 1 0 1 0 1) (0 1 1 1 0 1) (0 0 1 0 0 1) (0 0 0 0 1 1) (0 0 0 1 1
0))
```

b) Autres fonctions de services et précisions

(defun appartient (sous-liste liste) (dolist (i liste) (if (equal i sous-liste) (return T))))	Fonction pour déterminer si un état appartiendra à la liste des états passés.

(defun getnewstate (state) (dolist (i (successors state)) (if (not (appartient i oldstates)) (return i))))	Fonction qui renvoie le premier successeur non visité d'un état. Le return arrête le dolist et renvoie le premier successeur non visité d'un état.

Il faut également initialiser/définir les variables globales utilisées par les fonctions. On procède comme suit dans le cas présent :

```
(setq etat-initial '(0 0 0 1 0 1))

(setq oldstates '( (0 0 0 1 0 1) ))

(setq compteur '(0 1 2 3 4))
```

II. Exploration

a) En profondeur d'abord

3.

(defun explore-prof (state) (cond ((null state) (print "échec")) ((equal '(1 1 1 1 1 1) state) (print state) (print "succès ! :) ")) ((eq NIL (getnewstate state)) ;; si etat bloqué (nconc oldstates (list state)) (explore-prof (backtrack state))) (T ;; sinon (nconc oldstates (list state)) (print state) (explore-prof (getnewstate state)))))	Fonction d'exploration en profondeur : Si l'état est nul, on s'arrête et on écrit échec. Si l'état est égal à l'état final recherché, on l'affiche et on écrit succès. Si on ne trouve aucun successeur non visité, on est bloqué. On met cet état dans oldstates, et on continue l'exploration à partir d'un noeud trouvé avec backtrack. Sinon, on se contente de mettre l'état dans oldstates, de l'afficher et d'explorer le premier état trouvé avec getnewstate.
(defun backtrack (state) (let ((candidats '(0))) (setf candidats (reverse oldstates)) (dolist (i candidats) (if (not (eq NIL (getnewstate i))) ;; si pas bloqué (return i)) ;; s'arrête quand trouve un non bloqué)))	Fonction qui permet de remonter jusqu'au dernier noeud qui a des successeurs non visités. Soit une variable locale candidats qui contient la liste des états déjà visités, du plus récent au plus vieux. On parcourt cette liste. S'il trouve un état non bloqué, il s'arrête et le renvoie. Sinon, dolist renverra NIL.

Exemple d'exécution :

```
> (explore-prof etat-initial)

(0 0 0 1 0 1)
(1 1 0 1 0 1)
(1 0 1 1 0 1)
(0 1 1 1 0 1)
(0 1 0 0 0 1)
(1 0 0 0 0 1)
(1 1 1 0 0 1)
(0 0 1 0 0 1)
(0 0 1 1 1 1)
(1 1 1 1 1 1)
"succès ! :) "
"succès ! :) "
```

b) En largeur d'abord

4.

(defun explore-breadth (state)	Exploration en largeur
(let ((f '())(gagne nil)) ;; création de deux variables locales, f et gagne	On crée deux variables locales, f une liste vide et gagne qui vaut NIL.
(push state oldstates) ;; on met l'état dans oldstates	L'état en paramètre est mis dans oldstates.
(setq f (append f (list state))) ;; f = (state) (ex ((0 0 0 1 0 1)))	On met dans f la liste contenant state.
(loop (if (or gagne (null f)) ;; si gagne=T ou f=NIL (return nil) ;; on retourne NIL (progn	On fait une boucle avec loop : Si gagne est vrai ou si f est nul, alors on retourne NIL.
(let ((x (pop f))) ;; x = premier élément de f (un état)	Sinon, on crée une variable locale x qui contient le premier élément de f (donc un état).
(dolist (succ (successors x)) ;; on parcourt les successeurs de cet état	On parcourt les successeurs de cet état :
(if (equal succ '(1 1 1 1 1 1)) ;; si on trouve l'état final (progn (print succ) (setq gagne T))	Si le successeur est l'état final, on l'affiche et gagne devient vrai.
(if (not (appartient succ oldstates)) ;; si le successeurs n'appartient pas à oldstates	Sinon, si le successeur n'appartient pas à oldstates ...
(progn (print succ) ;; on l'affiche (push succ oldstates) ;; on le met dans oldstates (setq f (append f (list succ))))	... on l'affiche ... ... on le met dans oldstates ... ... on concatène f et ce successeur.
))))))))))))	

c) Comparaison

Les explorations en profondeur d'abord et en largeur d'abord sont deux méthodes de recherche dans un espace d'états. Ici, on peut remarquer que les deux mènent presque aussi rapidement à la solution. Avec l'état initial considéré c'est la profondeur d'abord qui est plus efficace.

Néanmoins, l'une comme l'autre sont finalement assez peu efficaces. La suite du TP se propose d'essayer d'y remédier.

III. Des états proches de la solution ?

a) Définition d'un état proche – fonction déterminant la proximité.

Comment définir un état le plus proche ? Nous avons choisi d'opter pour la définition suivante : plus un état est similaire à un autre état, plus ils sont proches. Voici une fonction qui permet de déterminer un degré de proximité arbitrairement défini de 0 à 6 : il est incrémenté de 1 pour chaque flèche dans le même sens d'un état à un autre.

(defun proximite (state final) (let ((count 0) (pos 0)) (dolist (i state) (if (= i (nth pos final)) (setq count (+ 1 count))) (setq pos (+ 1 pos))) count))	On parcourt chaque élément de l'état (state). Si un élément de state est égal à un élément de final (l'état avec lequel on compare) pour la même position, alors on incrément de 1 count. Bien sûr on incrémente de 1 la position à étudier dans final avant de repasser dans la boucle du dolist. On retourne count.

Exemple d'utilisation :

```
CG-USER(6): (proximite  '(0 0 0 0 1 1) '(1 0 0 1 1 0))
3
```

b) Fonction de service utile à une nouvelle fonction d'exploration.

(defun getnewstate-bis (state) (let ((plus-proche '(0 0 0 0 0 0))) (dolist (i (successors state)) (if (and (not (appartient i oldstates)) (> (proximite i etat-final) (proximite plus-proche etat-final))) (setq plus-proche i))) plus-proche))	Cette fonction permet de récupérer l'état successeur de l'état courant le plus proche de l'état final ('(1 1 1 1 1 1) ici) et non parcouru.

Exemple d'utilisation :

```
CG-USER(18): (getnewstate-bis '(1 0 1 0 1 0))
(0 1 1 0 1 0)
```

c) Nouvelle fonction d'exploration

```
(defun explore-prof-bis (state)

        (cond
          ( (null state)
            (print "échec")
           )

          ( (equal etat-final state)
            (print state)
            (print "succès ! :) ")
           )

          ( (eq NIL (getnewstate-bis state))
           ;; si etat bloqué
            (nconc oldstates (list state))
            (explore-prof-bis (backtrack-bis state))
           )

          ( T ;; sinon
           (nconc oldstates (list state))
           (print state)
           (explore-prof-bis (getnewstate-bis
state))
           )
          )
)

(defun backtrack-bis (state)
        (let ((candidats '(0)))
          (setf candidats (reverse oldstates))

          (dolist (i candidats)
            ;; parcourt les anciens états du plus
récent au plus vieux
            (if (not (eq NIL (getnewstate-bis i)))
                ;; si pas bloqué
                (return i)
              ) ;; s'arrête quand trouve un non
bloqué
            )
          )
        )
```

Elle est semblable à la précédente (de la question 3). Néanmoins on utilise la nouvelle fonction du point précédent pour obtenir l'état à parcourir. Il s'agit donc d'une exploration en profondeur d'abord avec un choix mieux orienté pour la branche à parcourir.

Note : il ne faut pas oublier de définir etat-initial (en plus des trois autres variables globales déjà précisées plus haut) :

```
(setq etat-initial '(1 1 1 1 1 1))
(1 1 1 1 1 1)
```

Exemple d'application : comparaison entre la première et la nouvelle fonction d'exploration en profondeur d'abord.

(explore-prof etat-initial)	(explore-prof-bis etat-initial)
(0 0 0 1 0 1) (1 1 0 1 0 1) (1 0 1 1 0 1) (0 1 1 1 0 1) (0 1 0 0 0 1) (1 0 0 0 0 1) (1 1 1 0 0 1) (0 0 1 0 0 1) (0 0 1 1 1 1) (1 1 1 1 1 1) "succès ! :) " "succès ! :) "	(0 0 0 1 0 1) (1 1 0 1 0 1) (1 0 1 1 0 1) (0 1 1 1 0 1) (0 1 1 0 1 1) (1 0 1 0 1 1) (1 1 0 0 1 1) (1 1 1 1 1 1) "succès ! :) " "succès ! :) "

La nouvelle fonction permet sur cet exemple d'arriver un peu plus rapidement au succès. Peut-être qu'une autre définition plus complexe de la proximité de deux états permettrait d'arriver à un résultat différent. Par exemple, pourquoi ne pas essayer avec un degré de proximité qui augmenterait pour chaque paire de flèches (nth 0 et nth 1, nth 2 et nth 3, nth 4 et nth 5) dans le même sens ?

Conclusion

Ce projet nous a permis d'effectuer une recherche dans un espace d'états pas deux méthodes et d'en apprécier les avantages et les inconvénients. Il nous a également amener à affiner cette recherche. Il est intéressant de remarquer que le problème est insoluble s'il y a un nombre impair de flèches dans le même sens. On peut peut-être ajouter cette clause aux fonctions pour éviter d'avoir à parcourir tous les états avant de retourner « échec ».

Notions importantes utilisées :

- Quand return est rencontré, on sort de la boucle !
- Boucles conditionnelles : if, when, unless.
- Boucles itératives : dolist.
- Variables internes et globales : ne pas oublier de réinitialiser les variables globales modifiées avant de rappeler les fonctions d'exploration.
- nth et setf.
- Programmation fonctionnelle.