**Recherche largeur d’abord**

**I. Définition des opérateurs par des fonctions Lisp**

1-On considère l’ensemble des listes (a b c d) dont tous les éléments sont des 0 ou des 1 et les trois opérateurs *left*, *middle* et *right* qui consistent à retourner respectivement les 2 éléments de gauche, du milieu et de droite de la liste.

**Ecrire les trois opérateurs sous forme de fonction lisp. Ex : (left** ' **(0 1 1 0))= (1 0 1 0)**

2- On considère quatre pions alignés ABCD. Ils peuvent être rangés dans n’importe quel ordre, mais le pion A doit rester à gauche du pion D. ABCD et CBAD sont 2 états possibles, mais DCAB et CDBA ne le sont pas. Le 2ème pion en partant de la gauche peut changer de position avec le 1er, le 3ème ou le 4ème pion si l’état résultant est un état possible. On considèrera toujours les échanges possibles dans cet ordre.

**Ecrire les trois opérateurs *one, three* et *four* sous forme de fonction lisp. Ces opérateurs renvoient le nouvel état si c’est un état possible et *nil* sinon.**

Ex : (one ' (c a b d))= (a c b d) et (one ' (a d b c)) = NIL

3- On considère l’ensemble des listes de trois nombres **(a b c)** où a, b et c sont dans {1,2,3} et les opérateurs ***1a***, ***1b****,* ***1c***,***2a***, ***2b***, ***2c***, ***3a, 3b, 3c*** qui consistent à transformer en 1, 2 ou 3 l’élément a, b ou c de la liste. Les conditions d’application de ces opérateurs sont les suivantes :

Un opérateur ne s’applique à la liste (a b c) que dans le cas où son application renvoie une liste différente de (a b c).

Les opérateurs qui modifient **b** ne peuvent s’appliquer si **b=a**.

Les opérateurs qui modifient **c** ne peuvent s’appliquer si **c=a** ou **c=b**.

Un opérateur ne peut s’appliquer s’il donne à **b** ou **c** la valeur de **a**.

1. Ecrire sous forme de fonction lisp les 9 opérateurs. Un opérateur renvoie nil, quand il ne peut pas s’appliquer.

ex : (1a ' (2 1 3)) = (1 1 3)

(1a ' (1 2 3)) = nil

(2b ' (1 1 3)) = nil

(1c ' (1 1 3)) = nil

**II. Gestion des états de l’arbre de résolution**

Un état généré est un nœud défini par un identifiant *node*. Un état possède les propriétés *feature*, *father* et *operator*. La propriété *feature* a pour valeur la configuration du nœud, la propriété *father* l’identifiant du nœud dont *node* est issu et la propriété *operator* l’opérateur appliqué au père de *node* pour obtenir *node*.

Les fonctions suivantes renvoient respectivement la configuration d’un nœud, l’identifiant de son père et l’opérateur qui a été appliqué à son père pour le générer.

(defun feature(node)

(get node 'feature))

(defun father(node)

(get node 'father))

(defun operator(node)

(get node 'operator))

*Ex : Soit* g100 *le nœud de configuration* (1 0 1 0) *et* g101 *le nœud obtenu après application de l’opérateur* left *sur la configuration du nœud* g100*. L’on a*

(feature 'g100)= (1 0 1 0)

(father 'g100) = nil

(operator 'g100)= nil

(feature 'g101)= (0 1 1 0)

(father 'g101) = g100

(operator 'g101)= left

La fonction *set-node(node feat*\_*node father*\_*node* *op)* ci-dessous associe au nœud *node* sa configuration *feat*\_*node*, son père *father*\_*node* et son opérateur *op*, puis renvoie *node*.

(defun set-node(node feat-node father-node op)

(putprop node feat-node 'feature)

(putprop node father-node 'father)

(putprop node op 'operator)

node)

avec

(defun putprop (symb val prop)

(setf (get symb prop) val))

**1- Ecrire la fonction *new-node(node op father-node)* qui crée un nouveau nœud *node* à l’aide de la fonction *set-node*, si l’application de l’opérateur *op* à la configuration de *father-node* est possible et renvoie nil sinon.**

Pour tester cette fonction, créer par exemple un nœud père d’identifiant g100, de configuration (1 0 0 1), sans père ni opérateur avec

**(set-node 'g100 ' (1 0 0 1) nil nil)**.

Créer un fils g101 après application de l’opérateur *left* avec

**(new-node 'g101 'left 'g100).**

Vérifier que

**(feature 'g101)** = (0 1 0 1),

**(father 'g101)** =g1

**(operator 'g101)** =left.

**2- Ecrire la fonction *children(node lop)* qui crée tous les nœuds** **suivants de l’état *node* par application des opérateurs de la liste *lop*. Ne pas créer de nœud si un opérateur ne peut s’appliquer et renvoie nil. Pour créer automatiquement l’identifiant d’un nœud, on utilisera la fonction prédéfinie *gensym* avec *(gensym)*.**

**La fonction prédéfinie** gensym **est définie comme suit :**

gensym invents a print name and creates a new symbol with that print name. It returns the new, uninterned symbol. gensym is usually used to create a symbol that should not normally be seen by the user.

**Ex (gensym) = # :G3100**

Tester la fonction ***children*** avec la séquence

**(setq l1 (children 'g100 ' (left middle right))** ) = **(#:G150 #:G200 #:G300)**

La fonction ***setq*** crée une variable globale de nom **l1** et de valeur **(#:G150 #:G200 #:G300)**

En tapant **l1** on obtient **(#:G150 #:G200 #:G300)**

**(car l1)** = **#:G150**, etc…

Compléter :

**(feature (car l1))** = .. .

**(feature (cadr l1))** =…

**(feature (caddr l1))** =…

**(father (car l1))** = g100

**(father (cadr l1))** = g100

**(father (caddr l1))** = g100

**(operator (car l1))** = left

**(operator (cadr l1))** = …

**(operator (caddr l1))** = …

**3-** **Ecrire, à l’aide de la fonction prédéfinie *some*, la fonction *seen(node lnode)* qui teste si la liste *lnode* comporte au moins un nœud ayant la même configuration que le nœud node.**

**La fonction prédéfinie *some(predicate seq)* est définie comme suit :**

This function calls predicate on each element of seq in turn; if predicate returns a non-nil value, some returns that value, otherwise it returns nil. You can rely on the left-to-right order in which the elements are visited, and on the fact that mapping stops immediately as soon as predicate returns non-nil.

Ex: (some 'numberp ' (o 2 m)) = t

(some 'numberp ' (c v n)) = nil

Tester la fonction *seen* avec

(seen 'g100 ' (g101 g100)) = T

(seen (car l1) l1) = T

(seen 'g100 l1) = NIL

**III Algorithme de recherche en largeur d’abord**

On considère l’algorithme de recherche en largeur d’abord défini comme suit :

(defun run (initial-state goalfeature lop)

(bdf (list (create-first-node initial-state)) nil goalfeature lop))

;====================================================

(defun create-first-node (initial-state)

(set-node (gensym) initial-state nil nil))

(defun bdf(open closed goalfeature lop)

(cond ((null open) (nosolution closed))

((goal (car open) goalfeature) (onesolution open closed))

((bdf (newopen open closed lop) (newclosed open closed) goalfeature lop))

))

Où *initial*-*state* est la configuration de l’état initial (par exemple (1 0 0 1)), *goalfeature* celle de l’état final (par exemple (1 1 1 1)) et *lop* la liste d’opérateurs (par exemple (left middle right)).

**Ecrire les 2 fonctions *newopen (open closed lop)* et *newclosed (open closed)* de manière modulaire.**

**Faire tourner avec:**

**(run** ' **(1 0 0 1)** ' **(**1 1 1 1**)** ' **(left middle right)),**

**(run** ' **(1 0 0 1)** ' **(0 0 0 1)** ' **(left middle right))**

**(run** ' **(a b c d)** ' **(a d b c)** ' **(one three four))**

**(run** ' **(a b c d)** ' **(c d a b)** ' **(one three four))**

**(run '(1 3 2) '(3 2 1) '(1a 1b 1c 2a 2b 2c 3a 3b 3c))**

;====================================================

(defun nosolution(closed)

(print "pas de solution")(kill-all-node closed))

;====================================================

(defun onesolution(open closed)

(build-solution (car open))

(kill-all-node (append open closed)))

;====================================================

(defun goal(node goalfeature)

(equal (feature node) goalfeature))

(defun feature(node)

(get node 'feature))

(defun operator(node)

(get node 'operator))

(defun father(node)

(get node 'father))

;====================================================

(defun build-solution (node)

(mapc (lambda(n) (print-node n) (terpri)) (list-node node)))

;====================================================

(defun kill-node (node)

(remprop node 'feature)

(remprop node 'father)

(remprop node 'operator)

nil)

;====================================================

(defun kill-all-node (lnode)

(mapc 'kill-node lnode))

;====================================================

;imprime ce que l'on souhaite imprimer d'un noeud

(defun print-node (node)

(print-operator node)

(print-feature node)

)

;====================================================

;construit la liste inversee des noeuds conduisant a node

(defun list-node(node)

(append (ancestors node) (list node)))

;====================================================

(defun ancestors (node)

(ancestors-rec node ()))

(defun ancestors-rec (node res)

(let ((fathernode (get node 'father) ))

(if (null fathernode)

res

(ancestors-rec fathernode (cons fathernode res)))))

;====================================================

(defun print-feature (node)

(print (get node 'feature)))

;====================================================

(defun print-operator (node)

(if (operator node) (print (operator node))))