Programming Assignment for CSCI 5511

Shreyasi Pal
pal00007 5483657

December 6, 2018

Question 1

functionatom.lisp

Description

The lisp code takes as input a list of arbitrary structure and returns a list that consists of the same atoms in the same order but with all the atoms at the same level.

Pseudo-Code

```
concatenate_lists(list1,list2):
{
    if list1 is null:
        return list2
    else
    {
        var0=concatenate_lists( (cdr list1), list2))
        var1= concatenate_atom( (car list1), var0)
        return var1
    }
}
functionatom(list):
{
    if list is empty:
    {
        return null
    }
    else if (car list) is a list:
    {
        var1=functionatom( (car list) )
        var2=functionatom( (cdr list) )
        var3=join_lists(var1,var2)
        return var3
    }
    else
    {
        var1=(car list)
        var2=functionatom( (cdr list))
        var3=join_atom_list(var1,var2)
        return var3
    }
}
```

Code

```
(defun functionatom(x)
        (cond
                ((null x) nil)
                ((listp (car x))
                        (ap (functionatom (car x)) (functionatom (cdr x)) )
                )
                (t (cons (car x) (functionatom (cdr x) ) ))

        )
)

(defun ap(list1 list2)
        (cond
                ((null list1) list2)
                (t
                        (cons (car list1) (ap (cdr list1) list2))
                )
        )

)


(functionatom '( a ( v c) e ))
```

Terminal Output

Known bugs and deficits:

No known bug

REPLACEWORD.LISP

Description

The inputs are a symbol and a list. The function replaces in the list all the instances of the symbol with YYYY.

Pseudo-Code

```
replaceword(symbol,list):
{
    if list is null:
        return null
    else if (car list) is equal to symbol:
    {
        var1=replaceword( (cdr list))
        var2=join_atom_list(YYYY,var1)
        return var2
    }
    else
    {
        var1=replaceword( (cdr list))
        var2=(car list)
        var3=join_atom_list(var2, var1)
        return var3
    }
}
```

Code

```
(defun replaceword(word pattern)
        (cond
                ((null pattern) nil)
                ((eql word (car pattern) )
                        (cons 'YYYY (replaceword word (cdr pattern)))
                )

                (t (cons
                        (car pattern)
                        (replaceword word (cdr pattern) ) )
                )

        )
)

(replaceword 'hello '(hello hello world))
```

Terminal Output

Known bugs and deficits:

Does not handle nested lists contaning symbol.

Question 2

8 Puzzle Problem using A* Search Algorithm

Description

The 8 puzzle problem contains a 3x3 matrix containing the numbers from 1 to 8 and space in it. The problem is to move the tiles such that the goal pattern is reached in the minimum number of steps. This can be done using a informed search.

Checking feasibility: The feasibility is checked if the initial state can reach the final state. This is checked by counting the number of inversions that are there in the list. If the number of inversions is odd, then the state cannot reach the goal state.

Check the number of elements in the list where list(i)>list(j) where i<j.

Heuristic used: Sum of Manhattan distance of each tile in their corresponding locations in start state and goal state. The heuristic is admissible.

Search method: A* search

The A* Search method is a type of informed search where a heuristic function is used to estimate the nearness of the current state to the goal state. The method maintains two lists Open and Closed. Each state is assigned a f-Score which is the sum of g-score and h-score. g-score is the cost of reaching the current state from the start state. h-score is the estimate of the cost of reaching the goal state from the current state. The steps are as follows.

1) Add the initial state to the Open List

2) Choose the best state from the Open List with the minimum f-score

3) Check if it is goal state or not

4) If not goal state, remove it from open list and push it in Closed List

3) Generate the successors of the best state and add the successors to the open list which are not present in Open or Closed list already

4) continue until best state is equal to goal state

State Representation: The states are represented as 1-D lists. The choice of representation stems from the motive of reduction of complexity. During the implementation of A* algorithm, a special structure Node has been generated which contains the state's g-score and f-score along with the state representation. This helps in choosing the best node for the next step.

Pseudo-Code

```
a*search(openList, closedList):
{
    next=leastF-ScoreState (openList)
    if next is goalState:
        return the states from closedList
    else:
    {
        openList=remove(next,closedList)
        closedList=add(next,closedList)
        successors=generateSuccandF-score(next)
        if successors not in closedList and openList:
            openList=add(successors,openList)
        next=leastF-ScoreState(openList)
    }
}
```

Code

```
(defparameter *goalState*  '(1 2 3 4 5 6 7 8 0))
(defparameter *nExpanded* 0)
(defun solvablep (state)
        (let
                ( (count 0)
                )

                ( do
            ( ( i 0 ( 1+ i ) )
            )
            ( ( >= i ( length state ) ) count )
            ( unless ( = ( nth i state ) 0 )
                ( do
                    ( ( j i ( 1+ j ) )
                    )
                    ( ( >= j ( length state ) ) )
                    ( when
                        ( and
                            ( > ( nth i state ) ( nth j state ) )
                            ( /= ( nth j state ) 0 )
                        )
                        ( setf count ( 1+ count ) )
                    )
                )
            )
        )
```

```
            (cond

                        ((evenp count) T)
                        (t NIL)
                    )

        )

)


(defun isGoal (currState)
    ( let (stateCheck flag)
        (setf stateCheck (mapcar #'eq currState *goalState*))
        (setf flag (position NIL stateCheck))
        (null flag)
    )
)

( defun generateSuccessors ( state )
    ( let (
        ( location ( position 0 state ) )
        ( children '() )
        UP DOWN LEFT RIGHT
      )

        ( when ( > location 2 )
            ( setf UP ( copy-list state ) )
            ( rotatef ( nth location UP ) ( nth ( - location 3 ) UP ) )
            ( setf children ( cons UP children ) )
        )
        ( when ( < location 6 )
            ( setf DOWN ( copy-list state ) )
            ( rotatef ( nth location DOWN ) ( nth ( + location 3 ) DOWN ) )
            ( setf children ( cons DOWN children ) )
        )

        ( when ( > ( mod location 3 ) 0 )
            ( setf LEFT ( copy-list state ) )
            ( rotatef ( nth location LEFT ) ( nth ( - location 1 ) LEFT ) )
            ( setf children ( cons LEFT children ) )
        )

        ( when ( < ( mod location 3 ) 2 )
```

```
            ( setf RIGHT ( copy-list state ) )
            ( rotatef ( nth location RIGHT ) ( nth ( + location 1 ) RIGHT ) )
            ( setf children ( cons RIGHT children ) )
        )
    children
    )
)


( defun calculate_hScore (state)
    ( let
        (
            ( count 0 )
            correctPosition
        )

        ( do
            (
                        ( i 0 ( 1+ i ) )
            )
            ( ( >= i 9 ) count )

            ( when ( not ( eq ( nth i state ) ( nth i *goalState* ) ) )
                ( setf correctPosition
                ( position ( nth i state ) *goalState* ) )
                ( setf count
                ( + count ( abs ( - ( floor i 3 )
                ( floor correctPosition 3 ) ) ) ) )
                ( setf count
                ( + count ( abs ( - ( mod i 3 )
                ( mod correctPosition 3 ) ) ) ) )
            )

         )
    )
)
( defun generateNode ( state parent )
    ( list
        ( 1+ ( car parent ) )
        ( calculate_hScore state )
        state
    )
)
```

```
(defun leastScoreState(state)
        (setq children (generateSuccessors state))
        (mapcar #'calculate_fScore children)

)

(defun a*(state)
        (setf *nExpanded* 0)
        (setf openList (list (list '0 '0 state)))
        (setf closedList NIL)
        (setf solution (aHelper* openList closedList ))
        solution

)
(defun aHelper*(openList closedList)
    (let
            (
                (next (findLeastScoreNode openList))
                joinedLists
                successors
                solution
            )
            (
            do()
            ( (isGoal (caddr next)) (getSolution closedList))
            (setf joinedLists (addNextToClosedList next openList closedList))
            (setf openList (car joinedLists))
            (setf closedList (cadr joinedLists))

            (setf successorNodeList
                    ( map
                        'list
                        #'( lambda ( state )
                            ( generateNode state next)
                        )
                        ( generateSuccessors ( caddr next ) )
                    )
            )
            (setf *nExpanded* (1+ *nExpanded*))
            (setf joinedLists
            (addSuccToOpenList successorNodeList openList closedList))
            (setf openList (car joinedLists))
            (setf closedList (cadr joinedLists))
            (setf next (findLeastScoreNode openList))
```

```
                )

        )
)

 (defun getSolution(closedList)
        (setq solution (mapcar (lambda (e) ( caddr e)) closedList))
        (reverse (cons *goalState* solution))

)

( defun findLeastScoreNode( openList &optional ( best () ) )
    ( cond
        ( ( not ( car openList ) ) best )
        ( ( not best ) ( findLeastScoreNode ( cdr openList )( car openList ) ) )
        ( ( < ( calculate_fScore ( car openList ) ) ( calculate_fScore best ) )
            ( findLeastScoreNode ( cdr openList ) ( car openList ) )
        )
        ( t ( findLeastScoreNode ( cdr openList ) best ) )
    )
)

( defun calculate_fScore ( node )
    ( + ( car node ) ( cadr node ) )
)

(defun addNextToClosedList(next openList closedList)
        ( let ( joinedList )
        ( when ( member next openList :test #'equal )
            ( setf openList ( remove next openList :test #'equal ) )
            ( setf closedList ( cons next closedList ) )
            ( setf joinedList ( list openList closedList ) )
        )
        joinedList
        )
)

(defun addSuccToOpenList (succNodeList openList closedList)
    ( let
        (
            ( succNode ( car succNodeList ) )
            repeatNode
        )
        ( cond
            ( ( not succNode )
```

```
                ( list openList closedList )
            )
            ( t
                ( cond
                    ( ( setf repeatNode
                            ( searchNode ( caddr succNode ) closedList )
                        )

                    )
                    ( ( setf repeatNode
                            ( searchNode ( caddr succNode ) openList )
                      )
                    )
                    ( t
                        ( setf openList ( cons succNode openList ) )
                    )
                )
                ( addSuccToOpenList ( cdr succNodeList ) openList closedList )
            )
        )
    )
)
( defun searchNode ( state nodeList )
    ( car ( member
        state
        nodeList
        :test #'( lambda ( state node ) ( equal state ( caddr node ) ) )
    ))
)

(defun 8puzzle(puzzle)
    (let
        (
            solution
        )
                (cond
                        ( (not (solvablep puzzle))
                                        (format t "The puzzle is infeasible.~%")

                        )
                        (t
                                (setf solution (a* puzzle ))
                                solution
                        )
```

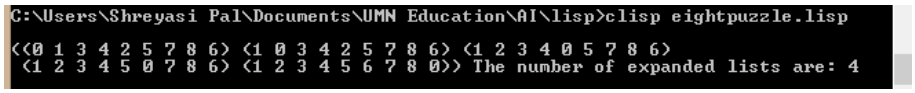
```
                    )


            )

)

(print (8puzzle '(0 1 3 4 2 5 7 8 6)))
(format t "The number of expanded lists are: ~S~%" *nExpanded*)
```

Terminal Output



The output shows the start state to goal state in one whole list. The state representation has been described before.

Known bugs and deficits:

No known bug