Lab-08

To show you how to solve problems using uninformed search

Objectives:

- To study uninformed search
- To study how agent can adapt uninformed search
- To implement the search Algorithms

Apparatus:

- Hardware Requirement
  Personal computer.
- Software Requirement
  Anaconda

Theory:

Often we encounter problems that do not have a trivial solution i.e. more than one step is required to solve the problem. To solve these kinds of problems we can search for solutions. We start with the given state of the problem and then apply the relevant actions that are allowed at that particular state. We repeat this process until either we get to the solution or we have no other actions to apply, in which case we declare that there is no solution exists.

To solve such problems, the first step is to model such problems.

The modeling requires the following elements:

- A way to represent the 'state' of the problem
- The complete state space of the problem
- The successor function. This function tells that at any given state which actions can be taken, what are the cost of those actions, and which new state our problem will be in, if we choose and particular action.
- Initial state, the initial configuration of the problem
- Goal test, a test to verify whether our current state is the goal state or not.

In this lab we will look at some reusable code to model the search problem. In addition, we will also look at some algorithms to solve these problems

The SearchProblem class defines the general structure that each problem needs to define. All problems should define how to model the state, how to generate successors and how to verify whether the goal has been achieved or not.

//SEARCH PROBLEM CLASS

The SearchProblem class defines the general structure that each problem needs to define. All problems should define how to model the state, how to generate successors and how to verify whether the goal has been achieved or not.

```
from abc import abstractmethod
 class SearchProblem(object):

   @abstractmethod
     def __init__(self, params):pass

   @abstractmethod
     def initialState(self): pass

   @abstractmethod
     def succesorFunction(self,currentState): pass

   @abstractmethod
     def isGoal(self,currentState): pass

   @abstractmethod
     def __str__(self) :
pass
```

//SEARCH STATE

The SearchState class is an abstraction of what every state should have. Every state should have a representation of itself, the action that got the search to this particular state, the cost of getting to this state and the string representation of the state. This string representation comes handy when maintaining a duplicate state set.

```
from abc import abstractmethod
 class SearchState(object):
     '''
   classdocs
   '''


   @abstractmethod
     def __init__(self, params): pass

   @abstractmethod
     def getCurrentState(self): pass

   @abstractmethod
     def getAction(self): pass

   @abstractmethod
     def getCost(self):pass

   @abstractmethod
     def stringRep(self) : pass
```

Modelling 8 Puzzle problem as a search problem

Now we will show how the searchProblem abstract class can be used to model any search problem. For an example we will model the 8 Puzzle problem. Actually the way we have modelled this problem, any valid N-Puzzle problem can be solved.

Below is the EightPuzzleProblem class

```
from com.search.searchProblem import SearchProblem import copy
from com.search.eightPuzzleState import EightPuzzleState

class EightPuzzleProblem(SearchProblem):
    '''
  classdocs
  '''

    def __init__(self, initialState,goalState):
    '''
    Constructor
    '''
    self._initialState = EightPuzzleState(initialState, '', 0)
    self._goalState = goalState
    self._numberOfRows = len(initialState)
    self._numberOfColumns = len(initialState[0])

    def initialState(self):
    return self._initialState

    def succesorFunction(self,cs):
    nextMoves = []
    emptyRow,emptyColumn = 0,0
    currentState = cs.currentState

    emptyFound = False

    for i in range(len(currentState)):
            for j in range(len(currentState[i])):
            if currentState[i][j] == 0:
            emptyRow,emptyColumn = i,j
            emptyFound = True
            break
            if emptyFound:
            break

    #check up move
    if emptyRow != 0:
```

```
            newState = copy.deepcopy(currentState)
            tempS = newState[emptyRow-1][emptyColumn]
            newState[emptyRow-1][emptyColumn] = 0
            newState[emptyRow][emptyColumn] = tempS
            ep = EightPuzzleState(newState, 'Move Up', 1.0)          nextMoves.append(ep)

    #check down move
    if emptyRow + 1 != self._numberOfRows:
            newState = copy.deepcopy(currentState)
            tempS = newState[emptyRow + 1][emptyColumn]
            newState[emptyRow + 1][emptyColumn] = 0
            newState[emptyRow][emptyColumn] = tempS
            ep = EightPuzzleState(newState, 'Move Down', 1.0)
            nextMoves.append(ep)

    #check left move
    if emptyColumn != 0:
            newState = copy.deepcopy(currentState)
            tempS = newState[emptyRow][emptyColumn-1]
            newState[emptyRow][emptyColumn-1] = 0
            newState[emptyRow][emptyColumn] = tempS
            ep = EightPuzzleState(newState, 'Move Left', 1.0)
            nextMoves.append(ep)

    #check right move
    if emptyColumn +1 != self._numberOfColumns:
            newState = copy.deepcopy(currentState)
            tempS = newState[emptyRow][emptyColumn+1]
            newState[emptyRow][emptyColumn+1] = 0
            newState[emptyRow][emptyColumn] = tempS
            ep = EightPuzzleState(newState, 'Move Right', 1.0)
            nextMoves.append(ep)

    return nextMoves

    def isGoal(self,currentState):
    cs = currentState.getCurrentState()          for  i  in
range(len(cs)):
            for j in range(len(cs[i])):
            if cs[i][j] != self._goalState[i][j]:
            return False        return True
```

The state of the Puzzle problem is represented as a two dimensional array.

The successor functions whether there is a move available in the up, down, left and right direction. If there is, then the next state, along with its action, cost is encapsulated in the EightPuzzle state class shown below.

```
from com.search.searchState import SearchState
 class EightPuzzleState(SearchState):
```

```
    '''
  classdocs
  '''
    def __init__(self, currentState,action,cost):
    '''
    Constructor
    '''
    self.currentState = currentState
    self.action = action
    self.cost = cost
    self.string = None

    def getCurrentState(self):
    return self.currentState

    def getAction(self):
    return self.action

    def getCost(self):
    return self.cost

    def stringRep(self) :
    if self.string is None:
            e = ''
            for i in range(len(self.currentState)):
            for j in range(len(self.currentState[i])):
            e += str(self.currentState[i][j])                    self.string = e
    return self.string
```

We also have a node class that represents the node of a search tree. The class is as follows:

```
class Node(object):
    '''
  classdocs
  '''
    def __init__(self, state, parentNode=None,depth=0,cost=0,action=''):
    '''
    Constructor
    '''
    self.state = state
    self.parentNode = parentNode
    self.depth =depth
    self.cost = cost
    self.action = action

    def __str__(self) :
    return self.state + ' -- '+self.action+' -- '+self.cost
```

Search Strategy

As we have discussed in class, the different search strategies like BFS, DFS and UCS only differ in the way they maintain the fringe list. Given below is the generic search strategy class. Every search strategy requires only three operations. 1) to check whether the fringe list is empty or nor, 2) to add a node, 3) to remove a node from the list.

```
from abc import abstractmethod
 class SearchStrategy(object):
    '''
   classdocs
   '''

   @abstractmethod
     def __init__(self, params):pass

   @abstractmethod
     def isEmpty(self):pass


   @abstractmethod
     def addNode(self,node):pass


   @abstractmethod
     def removeNode(self):pass
```

For this experiment, I will show you how to implement the breadthfirstsearch strategy.

```
from com.search.strategy.searchStrategy import SearchStrategy from queue import Queue
 class BreadthFirstSearchStrategy(SearchStrategy):
    '''
   classdocs
   '''
     def __init__(self):
     self.queue = Queue()

     def isEmpty(self):
     return self.queue.empty()

     def addNode(self,node):
     return self.queue.put(node)

     def removeNode(self):
     return self.queue.get()
```

Given these classes we can generically define the search class as follows:

```
 from    com.search.node    import
Node
from com.search.eightPuzzleProblem import EightPuzzleProblem
from com.search.strategy.breadthFirstSearchStrategy import BreadthFirstSearchStrategy
 class
Search(object):
    '''
  classdocs
  '''


    def __init__(self, searchProblem, searchStrategy):
    '''
    Constructor
    '''
    self.searchProblem = searchProblem
    self.searchStrategy        =        searchStrategy
    def solveProblem(self):
    node = Node(self.searchProblem.initialState(), None, 0, 0, '')
    self.searchStrategy.addNode(node)

    duplicateMap = {}
    duplicateMap[node.state.stringRep()] = node.state.stringRep()

    result = None

    while not  self.searchStrategy.isEmpty():
        currentNode        =        self.searchStrategy.removeNode()

            if self.searchProblem.isGoal(currentNode.state):
            result = currentNode
            break

            nextMoves = self.searchProblem.succesorFunction(currentNode.state)

            for nextState in nextMoves:
            if nextState.stringRep() not in duplicateMap:
            newNode = Node(nextState, currentNode, currentNode.depth + 1,
currentNode.cost        +        nextState.cost,        nextState.action)
    self.searchStrategy.addNode(newNode)
duplicateMap[newNode.state.stringRep()]                                =
newNode.state.stringRep()                    return result

    def printResult(self,result):
    if result.parentNode is None:
```

```
            print("Game Starts")
            print("Initial State : %s" % result.state.getCurrentState())          return
    self.printResult(result.parentNode)
    print("Perform the following action %s,  New State is  %s,  cost is
%d"%(result.action,result.state.getCurrentState(),result.cost))
```

To initialize the search, we need to pass it the search problem and a searching strategy. The search process if as follows: We have used a dictionary to maintain the duplicate list.

Lab Task

1. Modify the code in the search class to count the number of nodes expanded by the search
2. Understand the code and implement the following search strategies,
    - Depth first search, and
    - Uniform cost search
    - Breadth First Search

3. Model the Sudoku solving problem (Bonus marks)