

# DFS

def depthFirstSearch(problem):

start = problem.getStartState()

c = problem.getStartState()

exploredState = []

exploredState.append(start)

states = util.Stack()

stateTuple = (start, [])

states.push(stateTuple)

while not states.isEmpty() and not problem.isGoalState(c):

# pop one.. stack will result the state which was pushed in the end...

# hence the deeper node in the currenct branch will get an opporutunity.

state, actions = states.pop()

exploredState.append(state) # mark as visited

successor = problem.getSuccessors(state) # get childs

for i in successor:

coordinates = i[0]

if not coordinates in exploredState:

c = i[0]

direction = i[1]

states.push((coordinates, actions + [direction]))

return actions + [direction]

# BFS

def breadthFirstSearch(problem):

"""Search the shallowest nodes in the search tree first."""

start = problem.getStartState()

c = problem.getStartState()

visited = [] # just a array which tracks visited states

visited.append(start)

states = util.Queue() # a queue which helps in breadth first search

states.push((start,[]))

while not states.isEmpty():

state, action = states.pop() # pop one from the queue .. queue will return the one which was pushed first..

# if it's the goal return.. otherwise push the children to the queue

if problem.isGoalState(state):

return action

successor = problem.getSuccessors(state)

for i in successor:

coordinates = i[0]

if not coordinates in visited:

direction = i[1]

visited.append(coordinates)

states.push((coordinates, action + [direction]))

return action

# UCS

def uniformCostSearch(problem):

"""Search the node of least total cost first."""

print 'ullu ka pattha'

start = problem.getStartState()

visited = []

states = util.PriorityQueue()

states.push((start, []) ,0)

while not states.isEmpty():

state, actions = states.pop()

if problem.isGoalState(state):

return actions

if state not in visited:

successors = problem.getSuccessors(state)

for succ in successors:

coordinates = succ[0]

if coordinates not in visited:

directions = succ[1]

newCost = actions + [directions]

states.push((coordinates, actions + [directions]), problem.getCostOfActions(newCost))

visited.append(state)

return actions