ISLAMIC UNIVERSITY OF TECHNOLOGY

Organization of Islamic Cooperation

Board Bazar, Gazipur

Lab Report 1

CSE 4712

Md. Bakhtiar Hasan

11th September, 2022

In this report, the use of the Depth First Search, Breadth First Search and Uniform Cost Search algorithms with regards to Search Tree Problems is explored. All three algorithms are implemented using a priority queue. As such, the general structure of the algorithms is being explained first. The subtle differences to the structure based on the specific algorithm being used is explored afterwards.

## General Structure

The algorithm starts off with an empty action list and visited list. The action list is coupled with each state that is pushed into the fringe. Initially, it is coupled with the start state retrieved from the search problem. The visited list is used to keep track of nodes that have already been expanded.

action\_list = []  
visited = []  
start\_state = (problem.getStartState(), action\_list)

PYTHON

The start state coupled with the action list is then pushed onto the fringe.

fringe = util.PriorityQueue()  
fringe.push(start\_state, 0)

PYTHON

After this, we enter a loop that does not end until the fringe becomes empty. We pop states off the fringe one at a time. Since a priority queue is used, the order in which the states are popped off depends on their priority. The priority of the states in turn depend on the algorithm we use, so it is explored later.

while not fringe.isEmpty():  
 current\_state, action\_list = fringe.pop()

PYTHON

The next action is to check if the state that is being expanded now is the goal state. If it is, we return the associated action list.

if problem.isGoalState(current\_state):  
 return action\_list

PYTHON

Otherwise, we first ensure that the current state has not been visited before. If it is not, we add it to the visited list.

if current\_state not in visited:  
 visited.append(current\_state)

PYTHON

After that, we retrieve all the successors of the current state and push them onto the queue. Every time we push a state onto the fringe, the action associated with reaching that state from the current one is added to the action list being coupled with the state, so we can keep track of the path.

successors = problem.getSuccessors(current\_state)  
for successor, action, cost in successors:  
 next\_state = (successor, action\_list + [action])

PYTHON

Since a priority queue is being used, the state being pushed onto the queue will need to have a priority value associated with it. This priority depends on the algorithm we use, so that section is explored later.

Finally, if the entire loop completes without us having found the goal state, then there is no goal state. In this case, we return None.

return None

PYTHON

## Question 01

The first problem requires that we use a stack to implement the Depth First Search algorithm. To implement a stack using a priority queue, we must introduce a new variable, stack\_priority. This variable is decremented every time we push something onto the queue and incremented every time we pop something off of the queue. We also use the value of the variable as the priority for every new state we push onto the queue. Since consecutive pushes will have decreasing priority values, the last pushed state will always have the lowest priority value and will thus be the first state to be popped off by the queue.

Initially, the variable value is decremented when we first push the start state onto the queue.

stack\_priority = 0  
fringe = util.PriorityQueue()  
fringe.push(start\_state, 0)  
  
if fringe\_type == "stack":  
 stack\_priority = stack\_priority – 1

PYTHON

Inside the loop, for every state we pop off the queue, the variable value is incremented.

current\_state, action\_list = fringe.pop()  
  
if fringe\_type == "stack":  
 stack\_priority = stack\_priority + 1

PYTHON

When pushing states onto the queue, the current value of the variable is used as the priority. The value is then decremented.

next\_state = (successor, action\_list + [action])  
if fringe\_type == "stack":  
 fringe.push(next\_state, stack\_priority)  
 stack\_priority = stack\_priority – 1

PYTHON

## Question 02

The second problem requires us to use a First-In-First-Out (FIFO) queue to implement the Breadth First Search algorithm. A priority queue becomes a FIFO queue when the priority for every state is the same. As such, the only change required is to provide a constant priority when pushing states onto the priority queue.

elif fringe\_type == "queue":  
 fringe.push(next\_state, 0)

PYTHON

## Question 03

The final problem requires us to implement the Uniform Cost Search algorithm using a priority queue. As the basic structure already uses a priority queue, our only concern is the cost. The priority of states in the Uniform Cost Search algorithm is based on the total cost of reaching the states from the start node. Whenever a new node is expanded, the total cost of all actions required to reach that node is retrieved.

if fringe\_type == "priority\_queue":  
 total\_cost = problem.getCostOfActions(action\_list)

PYTHON

When pushing successors of the current node onto the queue, the priority is the sum of the total cost of reaching the current node and the cost of moving to the successor.

elif fringe\_type == "priority\_queue":  
 fringe.push(next\_state, total\_cost + cost)

PYTHON

## Complete Structure

The complete structure taking all three algorithms into consideration is provided below.

def uninformedSearch(problem, fringe\_type):  
 action\_list = []  
 visited = []  
 start\_state = (problem.getStartState(), action\_list)  
   
 stack\_priority = 0  
 fringe = util.PriorityQueue()  
 fringe.push(start\_state, 0)  
  
 if fringe\_type == "stack":  
 stack\_priority = stack\_priority - 1  
  
 while not fringe.isEmpty():  
 current\_state, action\_list = fringe.pop()  
  
 if fringe\_type == "stack":  
 stack\_priority = stack\_priority + 1  
   
 if fringe\_type == "priority\_queue":  
 total\_cost = problem.getCostOfActions(action\_list)  
  
 if problem.isGoalState(current\_state):  
 return action\_list

if current\_state not in visited:  
 visited.append(current\_state)  
 successors = problem.getSuccessors(current\_state)  
   
 for successor, action, cost in successors:  
 next\_state = (successor, action\_list + [action])  
 if fringe\_type == "stack":  
 fringe.push(next\_state, stack\_priority)  
 stack\_priority = stack\_priority - 1  
 elif fringe\_type == "queue":  
 fringe.push(next\_state, 0)  
 elif fringe\_type == "priority\_queue":  
 fringe.push(next\_state, total\_cost + cost)  
 return None

PYTHON