CS362 Artificial Intelligence

Course Instructor : Pratik Shah

Group Name : Code_Brigade
Group Members :

Patel Bhavik – 202051134
Suraj Poddar – 202051186
Tejas Gundale – 202051191
Tushar Maithani – 202051194

What is agent?

An agent is anything that perceives its environment using sensors, process it and respond the environment using actuators.

Uninformed Search Algorithms

In case of uninformed search algorithm we are not provided with information regarding our current node how much close to goal state/node.

Algorithm Name	Time Complexity	Space Complexity
Breadth-First search	$O(b^d)$	$O(b^d)$
Uniform Cost(Dijkstra)	$O(b^{1+C/\varepsilon})$	$O(b^{1+C/\varepsilon})$
Depth-First search	$O(b^m)$	O(bm)
Depth-Limited	$O(b^l)$	O(bl)
Iterative Deepening	$O(b^d)$	O(bm)
Bidirectional	$O(b^{d/2})$	$O(b^{d/2})$

Informed(Heuristic) Search Algorithms

It provides some information about goal state in form of heuristic function, which helps in finding solution more efficiently.

Algorithm Name	Time Complexity	Space Complexity
Greedy best-first search	O(bm)	O(|V|)
A^* Search	Depends on heuristic function	O(bm)

A .Write a pseudocode for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.

Pseudocode :

-we are provided with env, start state and goal state. And we have to tell user can we reach to goal state from start node.

Function :

def Graph_Search(env, start_state, goal_state)

-we have defined PriorityQueue as a frontier.

    frontier = PriorityQueue()

-dictionary to store explored nodes.

    explored = dict()

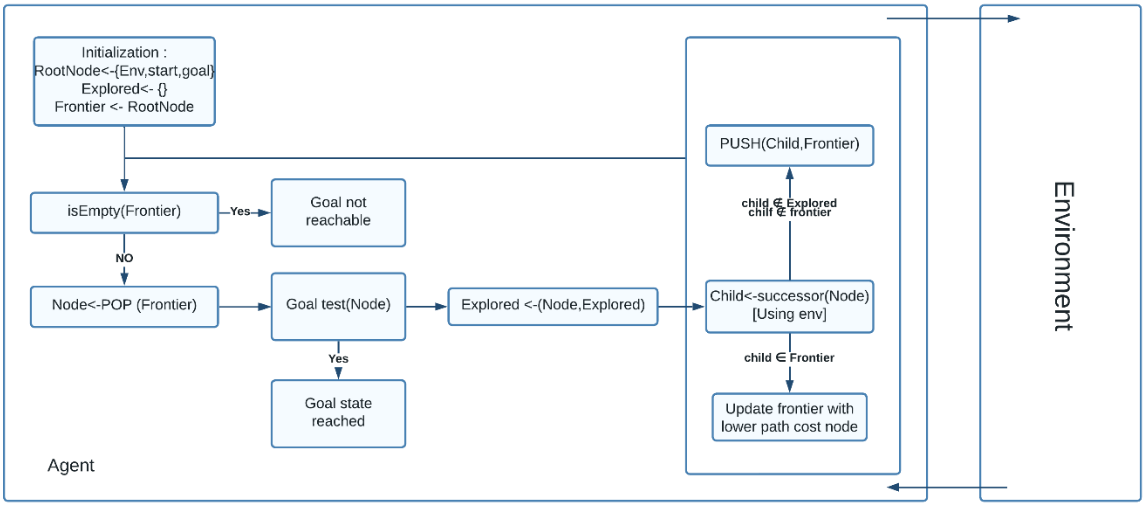
-First of all we will push our start_state in priority queue.

    frontier.push(start_state)

-Now we implement loop here till queue becomes empty
While not frontier,is_empty():
-Now we pop node from queue.
Current = frontier.pop()
if(current in explored)
continue
else
add it to explored
now compare it with goal_state if found
return true (.ie Here we are performing early testing)
else
using env find next all possible states and add it to frontier

if after completing all iteration goal state is not found then:
return false

Flowchart :



Reference : Artificial Intelligence A modern approach by Stuart Russell, Peter Norvig

```
class Agent:

    def __init__(self, env, heuristic):
        self.frontier = PriorityQueue()
        self.explored = dict()
        self.start_state = env.get_start_state()
        self.goal_state = env.get_goal_state()
        self.env = env
        self.goal_node = None
        self.heuristic = heuristic

    def run(self):
        init_node = Node(parent = None, state = self.start_state, pcost = 0, hcost=0)
        self.frontier.push(init_node)
        steps = 0
        while not self.frontier.is_empty():

            curr_node = self.frontier.pop()
```

```
            #print(curr_node.cost)
            next_states = self.env.get_next_states(curr_node.state)

            if hash(curr_node) in self.explored:
                continue

            self.explored[hash(curr_node)] = curr_node

            if self.env.reached_goal(curr_node.state):
                # print("We are done")
                self.goal_node = curr_node
                break
            goal_state = self.env.get_goal_state()

            for state in next_states:

                hcost = self.heuristic(state, goal_state)
                node = Node(parent=curr_node, state=state, pcost=curr_node.pcost+1,
hcost=hcost)
                self.frontier.push(node)
            steps += 1


        return steps, self.soln_depth()

    def soln_depth(self):
        node = self.goal_node
        count = 0
        while node is not None:
            node = node.parent
            count+=1

        return count

    def print_nodes(self):

        node = self.goal_node
        l = []
        while node is not None:
            l.append(node)
            node = node.parent

        step = 1
        for node in l[::-1]:
            print("Step: ",step)
            print(node)
            step+=1

    def get_memory(self):

        mem = len(self.frontier)*56 + len(self.explored)*56
        return mem
```

B.Write a collection of functions imitating the environment for Puzzle-8.

Initial State

1	2	3
8		4
7	6	5

Goal State

2	8	1
	4	3
7	6	5

It is 3*3 grid with 8 tiles numbered from 1 to 8 with one blank space.

-In function we have took one state ,depth as input.
-Then we are searching for blank space and storing it in tuple.
Space(0,0)
  for i in range(3):
    for j in range(3):
        if(state[i,j] == '_':
            space = (i,j)
-now on the basis of blank position we are applying all possible swapping functions as follows.(Basically we are swapping numbers)
If space[0] > 0 then we can move it up:
   new_state = copy(state)
   val = new_state[space[0], space[1]]
   new_state[space[0], space[1]] = new_state[space[0]-1, space[1]]
   new_state[space[0]-1, space[1]] = val
if space[0] < 2 then we can move it down:
   new_state = copy(state)
   val = new_state[space[0], space[1]]
   new_state[space[0], space[1]] = new_state[space[0]+1, space[1]]
   new_state[space[0]+1, space[1]] = val
if space[1] < 2 then we can move it right:
     new_state = copy(state)
     val = new_state[space[0], space[1]]
     new_state[space[0], space[1]] = new_state[space[0], space[1]+1]
     new_state[space[0], space[1]+1] = val
if space[1] > 0 then we can move it left:
     new state = copy(state) val = new_state[space[0], space[1]]
     new_state[space[0], space[1]] = new_state[space[0], space[1]-1]
     new_state[space[0], space[1]-1] = val

```
class Environment():

    def __init__(self, depth = None, goal_state = None,start_state=None):
        self.actions = [1,2,3,4] #1 - Up, 2 - Down, 3 - Right, 4 - Left
        self.goal_state = goal_state
        self.start_state = start_state
```

```python
    def get_start_state(self):
        return self.start_state

    def get_goal_state(self):
        return self.goal_state

    def get_next_states(self, state):

        space = (0,0)
        for i in range(3):
            for j in range(3):
                if state[i,j] == '_':
                    space = (i,j)
                    break

        new_states = []

        if space[0] > 0:# Move Up
            new_state = np.copy(state)

            val = new_state[space[0], space[1]]
            new_state[space[0], space[1]]  = new_state[space[0]-1, space[1]]
            new_state[space[0]-1, space[1]] = val

            new_states.append(new_state)

        if space[0] < 2: #Move down
            new_state = np.copy(state)

            val = new_state[space[0], space[1]]
            new_state[space[0], space[1]]  = new_state[space[0]+1, space[1]]
            new_state[space[0]+1, space[1]] = val

            new_states.append(new_state)

        if space[1]<2: #Move right
            new_state = np.copy(state)

            val = new_state[space[0], space[1]]
            new_state[space[0], space[1]] = new_state[space[0], space[1]+1]
            new_state[space[0], space[1]+1] = val

            new_states.append(new_state)

        if space[1] > 0: #Move Left
            new_state = np.copy(state)

            val = new_state[space[0], space[1]]
            new_state[space[0], space[1]] = new_state[space[0], space[1]-1]
            new_state[space[0], space[1]-1] = val

            new_states.append(new_state)

        return new_states
```

```python
def reached_goal(self, state):

    for i in range(3):
        for j in range(3):
            if state[i,j] != self.goal_state[i,j]:
                # print("Not able to reach goal state")
                return False
    # print("Reached successfully")
    return True
```

C. Describe what is Iterative Deepening Search.

BFS takes less time but more memory. And in case of DFS it consumes more time ,less memory, but it not always able to find goal state. Also DFS can stuck into infinite loop as it never keeps record of visited node.

In depth limited search we supply a depth limit ‘l’, and treat all nodes at depth ‘l’ as if they had no successors.

But choosing such ‘l’ such that we never miss desirable node is challenging, this problem is solved by iterative deepening search.

In Iterative deepening search, it solve this problem by trying all values for ‘l’ starting from 0,then 1,then 2, so on until either a solution is found or depth limited search returns the failure.

Thus we will get appropriate ‘l’ such that we get our goal state. First we perform DFS till ‘l’ ,then BFS at depth ‘l’ in this way it reduces space complexity a lot(.ie same as DFS) with assurity of getting solution(i.e. completeness).

Time Complexity : $O(b^d)$ when there is solution, or $O(b^m)$ when there is none.

Space Complexity : O(bd)

It is preferred uninformed search when state space is larger than provided memory and d is unknown.

Pseudocode:

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure

    for depth = 0 to ∞ do

        result←DEPTH-LIMITED-SEARCH(problem, depth)

        if result != cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, l) returns a node or failure or cutoff

    frontier←a LIFO queue (stack) with NODE(problem.INITIAL) as an element

    result←failure

    while not IS-EMPTY(frontier) do

        node←POP(frontier)

        if problem.IS-GOAL(node.STATE) then return node if DEPTH(node) > l then

result←cutoff

else if not IS-CYCLE(node) do

for each child in EXPAND(problem, node) do

add child to frontier

return result

Reference : Artificial Intelligence A modern approach by Stuart Russell, Peter Norvig

D. Considering the cost associated with every move to be the same (uniform cost), write a function which can backtrack and produce the path taken to reach the goal state from the source/ initial state.

Pseudocode :

Function:

```
def Path_to_goal(start,goal,graph):
    stack = []          //stack to store path(backtracking)
    set = {}            //to store visited node
    stack.push(start)
    set.add(start)
    while(set.size() ne number of node) :
        current = stack.pop()
        if(current eq goal)
            return stack content in reverse order
        else
            push such a child state to stack which is not in set
                if such state not exist then pop from stack
    return goal state not found
```

E. Generate Puzzle-8 instances with the goal state at depth “d”.

```python
def generate_start_state(self,depth,goal_state):

        past_state = goal_state
        i=0
        while i!= depth:
            new_states = self.get_next_states(past_state)
            choice = np.random.randint(low=0, high=len(new_states))

            if np.array_equal(new_states[choice], past_state):
                continue

            past_state = new_states[choice]
            i+=1

        return past_state
```

Testcase :

```python
depth = 500
goal_state = np.array([[1,2,3], [8,'_',4], [7,6,5]])
env = Environment(depth, goal_state)
print("Start State: ")
print(env.get_start_state())
print("Goal State: ")
print(env.get_goal_state())
# print(env.reached_goal()
```

```
... Start State:
    [['5' '4' '2']
     ['8' '3' '6']
     ['_' '7' '1']]
    Goal State:
    [['1' '2' '3']
     ['8' '_' '4']
     ['7' '6' '5']]
```

```python
depth = 500
goal_state = np.array([[1,2,3], [8,4,'_'], [7,6,5]])
env = Environment(depth, goal_state)
print("Start State: ")
print(env.get_start_state())
print("Goal State: ")
print(env.get_goal_state())
```

```
.. Start State:
   [['1' '_' '7']
    ['5' '2' '8']
    ['6' '3' '4']]
   Goal State:
   [['1' '2' '3']
    ['8' '4' '_']
    ['7' '6' '5']]
```

```python
depth = 200
goal_state = np.array([[1,'_',3], [8,4,2], [7,6,5]])
env = Environment(depth, goal_state)
print("Start State: ")
print(env.get_start_state())
print("Goal State: ")
print(env.get_goal_state())
# print(env.reached_goal())
```

```
Start State:
[['7' '4' '8']
 ['_' '3' '2']
 ['6' '5' '1']]
Goal State:
[['1' '_' '3']
 ['8' '4' '2']
 ['7' '6' '5']]
```

F. Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

```
depths = np.arange(0,501,50)
goal_state = np.array([[1,2,3], [8,'_',4], [7,6,5]])
times_taken = {}
mems = {}
for depth in depths:

    time_taken = 0
    mem = 0
    for i in range(50):
        env = Environment(depth=depth, goal_state=goal_state)
        agent = Agent(env = env, heuristic = heuristic)
        start_time = time()
        agent.run()
        end_time = time()
        time_taken+=end_time - start_time
        mem+=agent.get_memory()

    time_taken/=50
    mem = mem/50
    times_taken[depth] = time_taken
    mems[depth] = mem
    print(depth, time_taken, mem)
```

```
0 2.6659965515136717e-05 56.0
50 0.09459281444549561 25435.2
100 0.11656797885894775 52482.08
150 0.26750502109527585 92917.44
200 0.4435867404937744 119219.52
250 0.3287423896789551 104168.96
```

```
0 0.0001072216033935547 56.0
50 0.06930081367492676 23544.64
100 0.24551286697387695 67308.64
150 0.3341257619857788 87472.0
200 0.28524269580841066 87192.0
250 0.5877724361419677 123699.52
300 0.5580656909942627 123866.4
350 0.48133269786834715 132377.28
400 0.38779505252838137 116402.72
450 0.5758351135253906 137680.48
500 0.40150639057159426 116728.64
```

Time Complexity : $O(b^d)$
Space Complexity : $O(b^d)$
Where b = branching factor
d = depth

Reference:https://github.com/TanmayAmbadkar/CS302-AI/blob/master/Lab1/Graph%20Search%20Agent.ipynb