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)

Terms and some functions that we have used in our program:

➢ The graph search agent requires an environment to define the following

1. Start State
2. Goal State
3. Possible Actions

We have to make a generalised agent, which reaches the goal state using the functions of the environment. Our agent will use BFS/DFS to reach to the goal state

➢ The Node class generates the graph node. It has the following values

1. Parent Node
2. State
3. pcost - Path Cost

4. hcost - Heuristic Cost
5. cost - Total cost = pcost + hcost

It makes use of the following built in functions:
1. __hash__ : This provides the hash value for every node, which is required for the hashset
2. __eq__ : To check if 2 nodes are equal (Operator overload)
3. __ne__ : To check if 2 nodes are not equal (Operator overload)
4. __str__ : To get string representation of state in node

➢ Environment

The environment is what the agent plays in. It has the following entities:
1. actions : The actions defined in the environment
2. depth: the maximum depth of the solution
3. goal_state : The goal state of the environment
4. start_state : The start state generated at the depth

It has the following functions:
1. get_start_state : returns the start state
2. reached_goal : returns goal_state
3. get_next_states : Given current state, it returns all possible next states
4. generate_start_state : Given goal state and depth d, performs d moves to generate a start state.

➢ Agent

The agent is the player who plays the game against the environment to win. It has the following entities:
1. frontier : This is the priority queue used to store the nodes to be explored.
2. explored : This is the dictionary which stores the explored nodes
3. start_state : Stores the start state
4. goal_state : Stores the goal state
5. env : Stores the environment
6. goal_node : Stores the goal node if found
7. heuristic : Stores the heuristic function

The agent has the following functions:
1. run(): Is the function that explores the environment and finds the goal node. Uses the built in heuristic function to get the path costs
2. print_nodes(): To print the path from the start node to goal node

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)

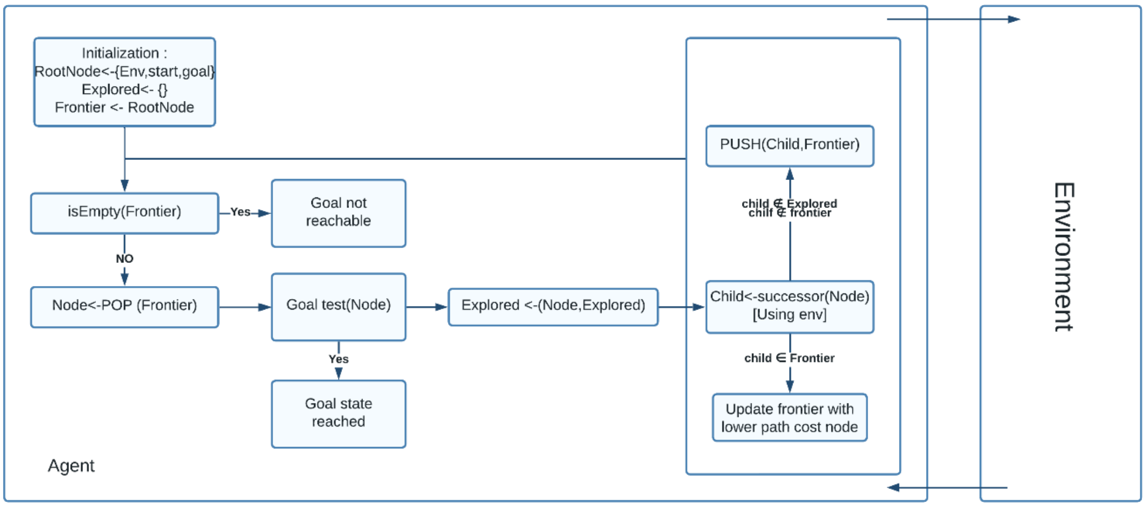
 else

   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

```python
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".

```
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 :

```
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']]
```

```
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']]
```

```
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(0,50,10)
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 = heuristic1)
        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 8.1634521484375e-05 56.0
10 0.0005667829513549805 803.04
20 0.0024642467498779295 2906.4
30 0.020881495475769042 13380.64
40 0.14380372524261475 46206.72
50 0.23339185237884522 72383.36
```

```
depths = np.arange(0,60,10)
goal_state = np.array([[1,2,3], [8,'_',4], [7,6,5]])
times_taken = {}
mems = {}
```

```python
for depth in depths:

    time_taken = 0
    mem = 0
    for i in range(10):
        env = Environment(depth=depth, goal_state=goal_state)
        agent = Agent(env = env, heuristic = heuristic1)
        start_time = time()
        agent.run()
        end_time = time()
        time_taken+=end_time - start_time
        mem+=agent.get_memory()

    time_taken/=10
    mem = mem/10
    times_taken[depth] = time_taken
    mems[depth] = mem
    print(depth, time_taken, mem)
```

```
0 0.0 56.0
10 0.00021033287048339845 918.4
20 0.0018488645553588867 1640.8
30 0.04061405658721924 19801.6
40 0.17886741161346437 60569.6
50 0.0682973861694336 42061.6
```

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