**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(bd) | O(bd) |
| Uniform Cost(Dijkstra) | O(b1+C/**ε**) | O(b1+C/**ε**) |
| Depth-First search | O(bm) | O(bm) |
| Depth-Limited | O(bl) | O(bl) |
| Iterative Deepening | O(bd) | O(bm) |
| Bidirectional | O(bd/2) | O(bd/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)

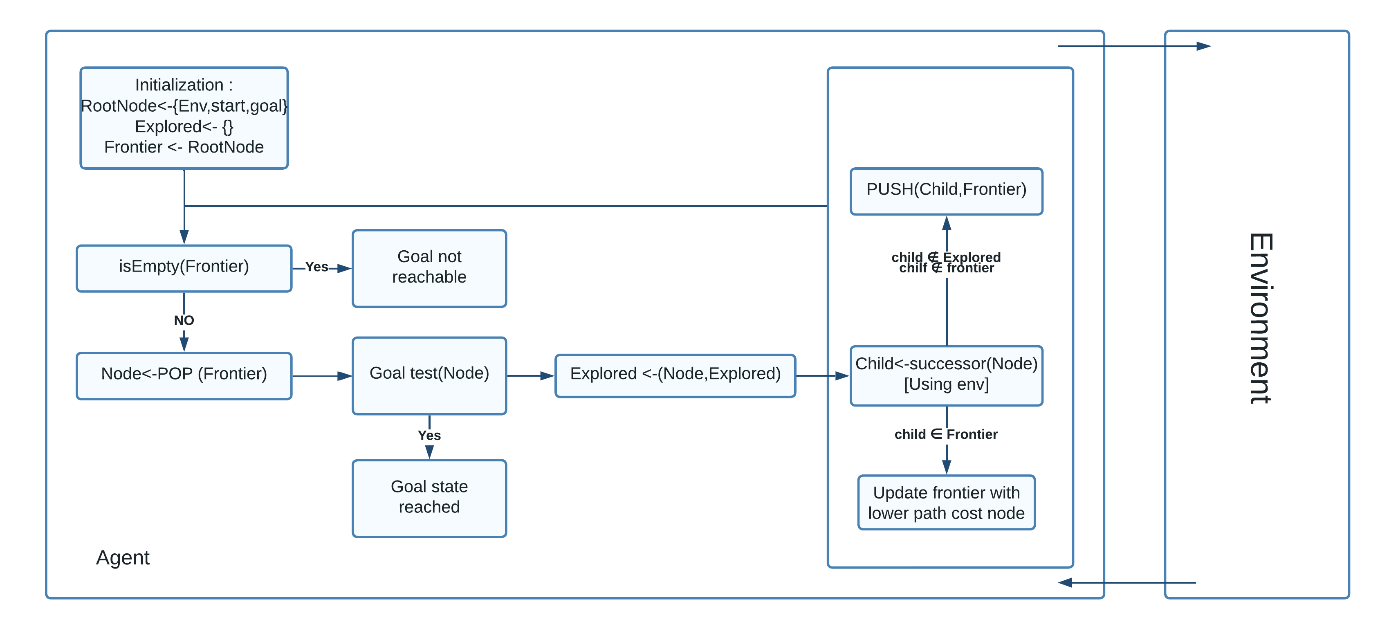
else

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

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)

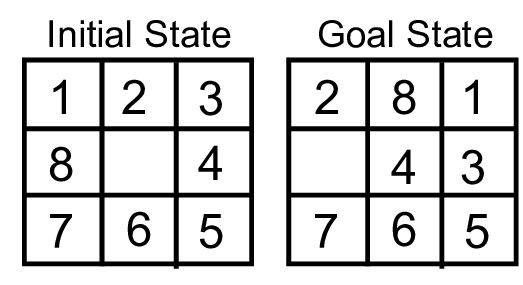
            step+=1

    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.**



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

    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

    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(bd) when there is solution, or O(bm) 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()



depth = 500

goal\_state = np.array([[1,2,3], [8,4,'\_'], [7,6,5]])

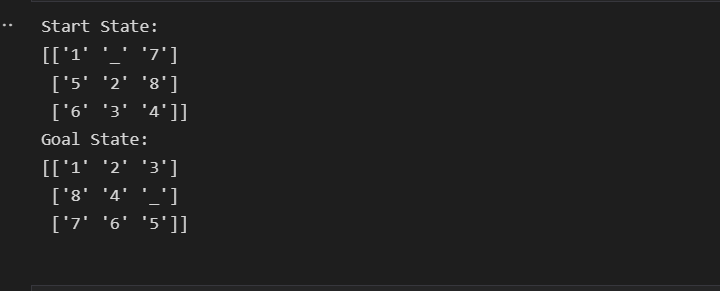
env = Environment(depth, goal\_state)

print("Start State: ")

print(env.get\_start\_state())

print("Goal State: ")

print(env.get\_goal\_state())



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())



**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()

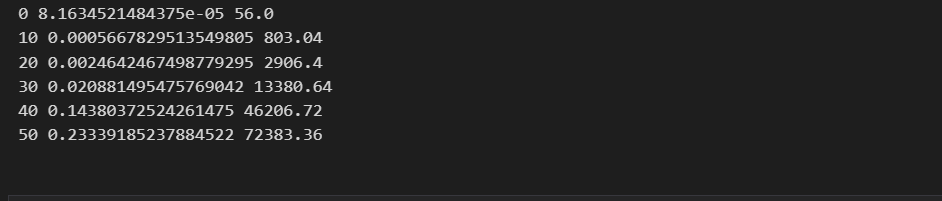
    time\_taken/=50

    mem = mem/50

    times\_taken[depth] = time\_taken

    mems[depth] = mem

    print(depth, time\_taken, mem)

****

depths = np.arange(0,60,10)

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(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()

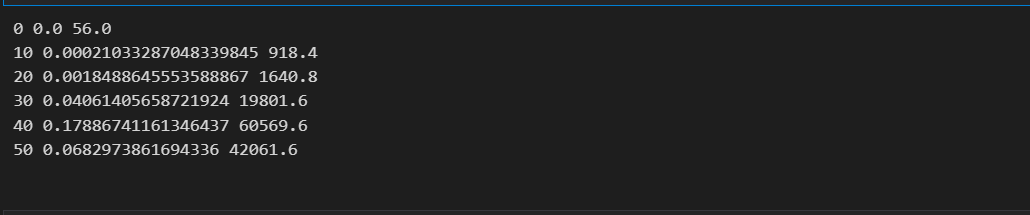
    time\_taken/=10

    mem = mem/10

    times\_taken[depth] = time\_taken

    mems[depth] = mem

    print(depth, time\_taken, mem)

****

Time Complexity : O(bd)

Space Complexity : O(bd)

Where b = branching factor

d = depth

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