Artificial Intelligence and Neural Network Lab Project

Project Title	Vacuum Cleanear Problem using Searching Algorithm	
Section	**Registration Number**	**Name**
Project by:	0147-BSCS-19	Hassaan Atif

Installation Requirement (if any)

import copy

Why you choose this algorithm (Advantage of Algorithm that you had use in your project)

(Maximum 200 words)

I chose these algorithms (Breadth First Search and Depth First Search) because they were the simplest ones of their kind. In fact, the problem scenario is quite simple, the properties of the environment make it particularly ideal for these fundamental searching algorithms. I am not penalizing my agent for anything! I don't care whether or not the agent performs optimally. I just have one goal and that is that the rooms are clean. I am not even accounting for non-determinism here. I feel as if it would be an "overkill" to opt for any other search strategy. Having said that, I have noticed several advantages of using DFS over BFS:
With DFS, I didn't have to explore any of the extra states that were not part of my solution path. With DFS, performance (number of steps) would be optimal if we ordered our action nodes **from worst to best.** Example: In state1, if we order actions as a FIFO list[L, R,S]. Then a FIFO list would retrieve the "S" option first and explore its successors and hence get us "optimally" to the goal state. So clearly, DFS is the better choice here.

Step by step visualization of Project.

(Add screenshot of your project with details about how to run project successfully)

You can run this just like any other python file by typing in:

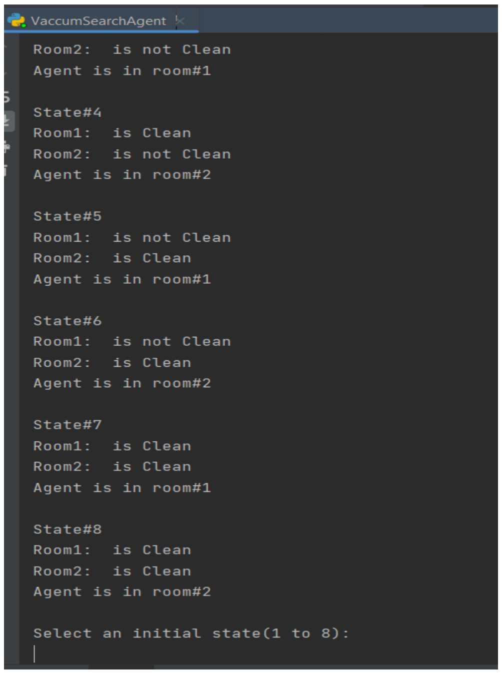
```
python VaccumSearchAgent.py
```

In my case, I am working with “PyCharm” IDE where I just toggle the run button that will run this python script for us.

Once run, it will create an instance of the environment class for us and an instance of the cleaner (which will be our agent). **Description for all these classes is given at the end.** Next, Icreate a state space and a transition table. The state-space consists of all the possible states of our environment and the transition table show which state to go upon any action in a particular state.

The transition table is represented as a key, value list/dictionary in python. The keys are a tuple of the current state and the action taken on the current state and the value is the state where we go to on that action.

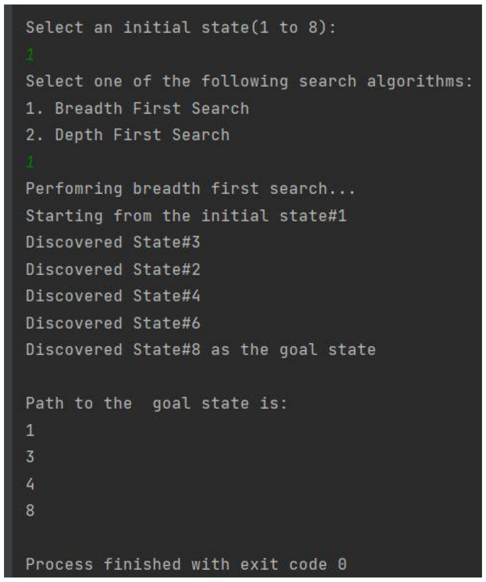
After creating the state space and the table, our program will then proceed to print out the entire state space. And then ask for the input with the aid of the function called "takeInput()". The user will first be asked to enter a valid initial state. The user may enter any of the numbers ranging from 1 to 8.



```
Room2:  is not Clean
Agent is in room#1

State#4
Room1:  is Clean
Room2:  is not Clean
Agent is in room#2

State#5
Room1:  is not Clean
Room2:  is Clean
Agent is in room#1

State#6
Room1:  is not Clean
Room2:  is Clean
Agent is in room#2

State#7
Room1:  is Clean
Room2:  is Clean
Agent is in room#1

State#8
Room1:  is Clean
Room2:  is Clean
Agent is in room#2

Select an initial state(1 to 8):
```

After you have entered a valid state number, the program will assume that that particular state is where your agent will start doing from, so the program will proceed to ask you to select a valid search algorithm. We have two options (Breadth First Search and Depth First Search). Let's say that I select state#1 as my initial state and I have decided to use breadth first search as my algorithm of choice:

Artificial Intelligence and Neural Network Lab Project



```
Select an initial state(1 to 8):
1
Select one of the following search algorithms:
1. Breadth First Search
2. Depth First Search
1
Perfomring breadth first search...
Starting from the initial state#1
Discovered State#3
Discovered State#2
Discovered State#4
Discovered State#6
Discovered State#8 as the goal state

Path to the  goal state is:
1
3
4
8

Process finished with exit code 0
```

The algorithm tells us the states that it had to explore while doing the search and then it returns a path to the main function where it is printing the entire path as: "1 \n3 \n4 \n8".

Let's get into the technicalities of our breadth-first search algorithm.

function: def breadthfirstsearch(initialState, transitions)

This function accepts an initial state and our transition model. It will then perform an "iterative breadth first search algorithm" and returns a path to the goal state.

It takes as the parameters the initial state and a transition table (dictionary). It first checks to see whether our initial state is the goal state (if that is true then it returns the initial state in the form of list. We can view this list as being a "**path**"). Otherwise, it will discover the immediate neighbors of the current node (except if the neighbor happens to the current node i.e., an action that takes us to the current state) and then see whether any of those neighbors are our goal state. It uses a **FIFO** queue to keep a track of the immediate neighbors. Then in the next iteration, it will pick out the first one of those neighbors and then discover its immediate neighbors and see whether any of those are our goal state. If that is not the case then it will check to see the immediate neighbors of **"the very first node whose neighbors we have not discovered yet"** (hence why I use a LIFO list).

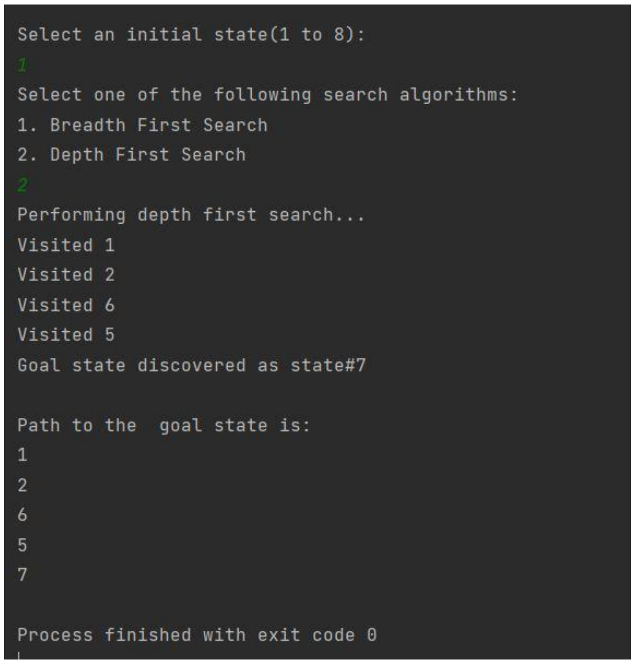
When we discover a goal state, we will retrieve the path to it through its **parent** field. We trace back to the initial state in reverse and put every state that we get into a stack-based list (**LIFO**). The stack is now filled with the reverse of our path. So we will empty the elements of the stack into another **FIFO** list (i.e., the **path** list). We then return the path to the **main** function where the path to the goal state is printed. It is also **important to note** that this function keeps track of all the nodes that we have discovered so far and makes sure that we don't revisit that same node/state again.

Artificial Intelligence and Neural Network Lab Project

function: def depthfirstSearch(initialState, transitions)

This function accepts an initial state and our transition model. It will then perform an "iterative depth first search algorithm" and returns a path to the goal state. Like our breadthfirstsearch algorithm, this function, too, keeps track of all the states that we have visited thus far to make sure that we don't revisit the same node again.
We initially check to see whether our initial state is the final state. If that is the case then we return our initial state as a list. If that is not the case then we perform the dfs algorithm onto it.
The depth first search algorithm will put all the immediate neighbors of the current node into a LIFO based list (which we'll call stack), at the same time checking whether any of those neighbors are our goal state. In the next iteration, it will pick the last of those neighbors that we put into the last during our last iteration and append all the immediate neighbors of that node into the stack and it will do the same checks as before and then keep performing the search in this fashion.
Just like our BFS algorithm, I are using the parent label to keep track of our path. And then using that I am getting the path from the initial state to our goal state using a combination of a LIFO list (for reverse path) and a FIFO list (for the actual, straight path).

Sample DFS Simulation:



```
Select an initial state(1 to 8):
1
Select one of the following search algorithms:
1. Breadth First Search
2. Depth First Search
2
Performing depth first search...
Visited 1
Visited 2
Visited 6
Visited 5
Goal state discovered as state#7

Path to the  goal state is:
1
2
6
5
7

Process finished with exit code 0
```

Other functions:

1. **def returnStateSpaceAndActions(state1)**

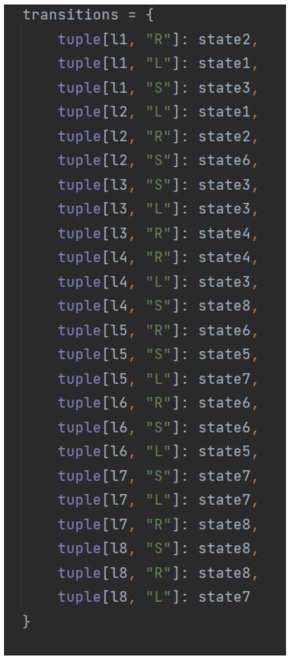
This creates the entire state space and transitions for different states on different actions. This method returns both the entire state space (in the form of a list) and a transition table. The value for the dictionary table is a state. So the general format for an entry in the table is as :

Artificial Intelligence and Neural Network Lab Project

tuple[arbitraryState.label, #action in the form of string (such as “L”, “R” and “S”)] : resultingState

So if we feed into our table an entry as:
Transitions = {
.
.
. entries till here,
tuple[state8.label,”L”] : state7
}

Now this tells our transition table that whenever we are in state8 and the action is go to left then it will transition to state7.
Our entire transition table:



```
transitions = {
    tuple[l1, "R"]: state2,
    tuple[l1, "L"]: state1,
    tuple[l1, "S"]: state3,
    tuple[l2, "L"]: state1,
    tuple[l2, "R"]: state2,
    tuple[l2, "S"]: state6,
    tuple[l3, "S"]: state3,
    tuple[l3, "L"]: state3,
    tuple[l3, "R"]: state4,
    tuple[l4, "R"]: state4,
    tuple[l4, "L"]: state3,
    tuple[l4, "S"]: state8,
    tuple[l5, "R"]: state6,
    tuple[l5, "S"]: state5,
    tuple[l5, "L"]: state7,
    tuple[l6, "R"]: state6,
    tuple[l6, "S"]: state6,
    tuple[l6, "L"]: state5,
    tuple[l7, "S"]: state7,
    tuple[l7, "L"]: state7,
    tuple[l7, "R"]: state8,
    tuple[l8, "S"]: state8,
    tuple[l8, "R"]: state8,
    tuple[l8, "L"]: state7
}
```

2. **returnNewState(room1Clean, room2Clean, agentPos, stateLabel)**

This method is pretty self explanatory. It asks for which of the rooms shuld be clean (in boolean values, agent’s position (which room the agent should be in) and then it asks for what label to give the state as. For convenience, I am using integer values as stateLabel. It then returns an instnace of the state class with the provided information.

3. **returnSuccessors(stateLabel, transitions)**
   This method returns to us the successor states of our current state. It will index each of the actions
4. **goalTest(state)**
   This accepts the current state as an input and returns true when both rooms are clean in the current state, otherwise, it returns false.

Artificial Intelligence and Neural Network Lab Project

The description of my classes:

1. **The cleaner class**
   This is the actual agent that will operate in the environment.
   getCurrentRoom(self): this gives us the current room for our agent
   setCurrentRoom(self,room): this allows us to travel from one room to the other
   suckDirt(self, roomNo, environment): this allows you to suck the dirt (if any) out of the current room.

2. **The environment class**
   This is the actual environment where the agent will perform its actions.

   isRoom1Clean(self): this tells us whether the first room is clean or not.
   isRoom2Clean(self): this tells us whether the second room is clean or not
   setRoom1Clean(self): clean room 1 if it is dirty
   setRoomt2Clean(self): clean room 2 if it is dirty

3. **State class**

   This represents a single state in the entire tree.. A state varies on the following factors:
   1. Presence/absense of dust in either of the rooms
   2. cleaner.getCurrentRoom()   //the location of the cleaner
   3. Environment //dust or no dust
   4. State label //this is just a number that we give to identify our states.
   5. The parent field is especially helpful in returning a path from our searching algorithms. This will depend on the searching algorithms AS WELL AS the initial state of our search algorithms

Artificial Intelligence and Neural Network Lab Project

Formalizing our entire model:

Start State: any state can be a start state
Actions: String labels “S” (meaning “‘suck”), “L” (indicating “go left”), and “R” (indicating “go right”)
Transition model: defined as a dictionary : Transitions(tuple[string, state], state)
Successor function: returnSuccessors(stateLabel, transitions)
Goal test: goalTest(state)

Write challenges of the project that you had faced while implementation.

Following were a few challenges that I had faced while implementing this:

- Programming Language barrier: perhaps this was the single most challenging thing during the implementation of this project. Getting across the language barrier seemed quite complex at times because it obviously makes us feel very uneasy. Since I have worked with Java before, my mind had been hardwired to translate the solutions in that particular way. Once this hurdle was out of the way, I was able to code this dream project of mine. And it made me feel very proud to see it come to life.
- Representation of a state: Having studied Artificial Intelligence from the book “Artificial Intelligence: A Modern Approach”, I understood these search problems quite well. The book talked about how in these elementary graph search algorithms (in the context of AI), I treat each state as an “**atomic**” blackbox. So translating that philosophy into code while still using the neat tricks of object-oriented principles and making sure that I was following good practices – essentially having “that balance” was quite a brainstormer for me.
- Data flow: Obviously, a lot of the data was going to be passed around. So I had to make sure that I had a well-structured way of communicating the data from one spot to another. As an example, I was assigning the responsibility for the state creation to a separate function which gets passed to the function that is responsible for making several other such states with difference in attriutes and references, at the same time, constructing a transition table based on these states. So, both the state space and the transition table get passed to the main method, and thus we have a well-structured way of dealing with this. This made my debugging process a lot simpler (but of course, debugging had its problems as well).
- Debugging of our search algorithms: It wasn’t that coding the actual algorithm itself was hard, but what made the process extra complicated was that I was utilizing and updating the “parent” attribute (that I discussed above) for each of the states in the search algorithm. I had to be very careful about when and where to update the parent reference, so that based on that data along with just the “final node itself” (i.e. the goal state), I would be able to track down my searching path.

What are the limitation of the Project?

Artificial Intelligence and Neural Network Lab Project

At the moment, the single biggest limitation is that the project doesn't give us an optimal path to the goal state. **Optimality**, here, implies the number of steps that it took for us to reach the goal state. Both of the algorithms that I have provided have this fatal flaw. A possible solution to this would be to use some sort of informed search algorithm that tries to maximize Also, if I do compare DFS algorithm with my BFS algorithm then I would like to highlight that there was a slighter advantage when the actions returned as a list were **placed from worst to best** because of the nature of a **LIFO** list.
Another limitation of the project is that I am not accounting for the possibility of **non-determinism** here. What if we suck dirt from room1, but room2 (which was previously clean) gets dirty as a result of that. I realize that the real world is not as "**deterministic**" as the program that I wrote (minimally, a solution to such a case would call for the need for an **AND/OR** representation of the state along with a search algorithm).

Submission Guideline:

You are required to submit Python files of project with Word Document (As per given template) in a single zip file (Attach .txt file or .csv file if there is any). Name of zip file should be the registration number. Registration Number should be as per university format.

For Viva, you are required to take the print of word document with fully charged laptop in which project should be run. In case, if you didn't bring your laptop or project will not run, the project will not be marked.