CISS450: Artificial Intelligence
Assignment 4

OBJECTIVES

1. The goal is to solve the several AI search problems using brute force or blind or uninformed search using BFS and DFS.

Note that you must implement the late graph search algorithm for this assignment (i.e., not the early graph search algorithm).

SUGGESTED PREPARATION

Note that the AIMA textbook does not give you all the details nor the code. (Obviously. Otherwise the AI class becomes an exercise in typing code.) It's up to you to study and analyze the search algorithms. Make sure you understand the algorithms completely before coding.

Before you write your program, I suggest you try to solve, for instance, the $n^2 - 1$ problem by hand to understand the algorithm. List down explicitly all the changes made to all the variables, objects, containers, etc. Here's an example problem for you:

0	1	2
3	4	
6	7	5

for you to try out using BFS and DFS. (For the above, what are the states you get when you expand? So what's in the fringe? What happens in the next iteration? Etc.)

If you still don't feel comfortable diving into code here's another problem for you to try for a couple of iterations:

0	1	2
3		5
6	4	7

You might want to develop a catalog of test cases completely solved by hand so that you can use it to test your program.

I definitely encourage you to collaborate and study the algorithms together.

However sharing of code is absolutely forbidden. Sharing code is the same as plagiarism – you will be dropped from the class with a grade of F if you plagiarize either from someone in the class or from some other resource such as the web.

Q1. MAZE PROBLEM

You will find a `maze` directory containing python code. It has a makefile. Go ahead and run make. (Make sure you copy all the file to you `a04q01` directory for this assignment.)

The goal is to place a bot in a maze and have the bot "think" and find a solution from its starting location to a given ending location based on different search algorithms. The visualization must show how the fringe and closed list changes in each iteration of the search process.

Watch the following videos:

1. BFS: https://photos.app.goo.gl/Mhee9sP7DQSJWx5j8
2. DFS: https://photos.app.goo.gl/4ZsnyhiPskTeM3mW8
3. BFS: https://photos.app.goo.gl/pSB6vkBxYTd5aJDB9
4. DFS: https://photos.app.goo.gl/ToDykYSojjzmuDgk7
5. BFS on 40-by-80 maze: https://photos.app.goo.gl/DirLhCC12whDmWxr8
6. DFS on 40-by-80 maze: https://photos.app.goo.gl/w6njwj4qtJ6AtgHo6

A state (a cell in the maze) is colored red if it is in the closed list, and blue if it is in the fringe. The initial state and goal state are colored green. The last two videos also draw edges between two states if they form a parent-child relation during the graph search process.

The following is the stdout output for a test case. This is meant to fix format – the output below is actually incorrect.

```
enter random seed: 5
0-random maze or 1-stored maze: 0
initial row: 1
initial column: 2
goal row: 3
goal column: 4
bfs or dfs: bfs
solution: ['S', 'N', 'E', 'N']
len(solution): 4
len(closed_list): 0
len(fringe): 0
```

(The above is meant to show you the format of the output. The solution is incorrect.)

There are 5 inputs:

- For the second input if the user enters 1, then a pre-created maze will be used.
- For the seventh input, selecting `bfs` will run the graph search algorithm using breadth first search strategy while selecting `dfs` will perform depth first search strategy.

Besides the above inputs, various parameters are stored in `config.py`, include the number of rows and columns of the maze, etc.

Beside the above, as mentioned in class, while the bot is thinking (i.e., performing a search to find the solution according to the choice of search strategy), your program must draw the fringe and closed list.

You will need to complete the code in several files that are provided. There are several parts to complete. You are advised to fully test each part before moving on.

- `main.py`: It's a good idea to first run `main.py` to get a feel for the problem. (A simple `makefile` is provided.) The code provided does not solve the maze problem – the bot simply start at $(0, 0)$ and walks about randomly. But you should trace `main.py` and see how the code works. In particular, you should look at `graph_search.py`.
- `pymaze.py`: This file contains the code to generate mazes. The code is complete and does not require changes. There's no need to study the code in details. The most important thing is to understand how to get the directions a bot can take at any (row, column) in the maze. Make sure you check the documentation at the top of `pymaze.py`. You will see how that's done once you trace `main.py` (see above). `main.py` will use the incomplete `graph_search` function in `graph_search.py` The incomplete `graph_search` function creates a random walk of the maze by querying the maze to see what are the available directions.
- `State.py`: It's up to you if you want to use a state class or simply use values (i.e., tuples) for states. For instance the state of $(2, 3)$ represents the fact that the bot is at row 2, column 3. The initial state and goal state (also tuples) are specified by the user. (This is a simple problem – using a class for states is probably an overkill.)
- `ClosedList.py`: The `ClosedList` class is the base class for the following two incomplete closed list classes that you must complete.
  - The `DummyClosedList` class is a dummy closed list class that maintains an empty closed list: if you add anything to the closed list, the value is ignored. If you ask if a value is in the closed list, it will reply with `False`.
  - The `SetClosedList` implements a closed list using the python sets.

  All the methods in the base class `ClosedList` must be implemented in the subclasses `DummyClosedList` and `SetClosedList`. Some test cases are included in `ClosedList`. Run `ClosedList.py` to run these test cases. Recall that closed lists contains states (well ... essentially, because recall I mentioned before that closed list tends to be huge and there are many ways to compress states.) Some test cases are in the `CloseList.py`. You are advised to include more test cases.
- `SearchNode.py`: Recall from the notes that a search node object contains a state (or rather a pointer or reference to state) and other information collected during the

search process (such as pointer or reference to the parent search node). You will need to complete the `solution()` method. Add more test cases.

- `Fringe.py`: The `Fringe` class is the base class for the following incomplete fringe classes that you must complete.
  - The `Stack` and `Queue` classes are the obviously subclasses of `Fringe` implemented using python's `deque` class.
  - The `FSStack` and `FSQueue` are the same as the `Stack` and `Queue` classes except that they also include the python `set` so that you can very quickly tell if a value is in the container or not. Memory aid `FS` = "fast search".

  Remember that a fringe holds search nodes.
  Your graph search algorithm must use the fast search versions. Note that the Fringe class contains an `__iter__` method. You don't have to worry too much about this since `__iter__` is already implemented in the `FSStack` and `FSQueue` classes. (See section on iterator.)
  A test for the stack fringe is included – study it carefully. You are advised to fully test all the fringe subclasses.
- Actions: This problem is simple enough that strings (i.e., `'N'`, `'S'`, `'E'`, `'W'`) are sufficient for actions. If you like, you can provide an `Action` and a subclass `MazeAction` class (but I think that's an overkill).
- `Problem.py`: You will need to complete the `MazeProblem` class which is a subclass of the `Problem` class. Note that a `MazeProblem` object contains a maze object create from the `DFSMazeWithCycles` class in `pymaze.py`.
- `graph_search.py`: The main function to implement is the `graph_search` function. The `graph_search` is called by `main.py`. The code in `main.py` calls `graph_search` with a suitable problem object, fringe object, closed list object, Of course you will need to provide a suitable fringe object and closed list object. An object, `view0`, for drawing a simulation of the search algorithm is also passed into `graph_search`.
  You are advised to create smaller test cases to test your graph search algorithm.

(See next few pages for general discussion on deques, sets, etc. Most of the information in found in the python notes.)

CLONING OBJECTS

Frequently you need to make a copy of objects. For basic types such as integers:

```
x = 5
```

the following works:

```
y = x
```

However in many cases, such as a list:

`x = [1,2,3]`

doing this

```
y = x
```

does not make a copy of `[1,2,3]` for `y`. `y` references the same object that `x` is refering to. This means

```
y[0] = 42
```

changes the same object that x is refering to. Try this:

```
x = [1,2,3]
y = x
y[0] = 42
print("%s %s" % (x, y))
print("%s %s" % (id(x), id(y)))
```

The above is already mentioned in the Python notes. One way to clone a list is the following:

```
x = [1,2,3]
y = x[:]
y[0] = 42
print("%s %s" % (x, y))
print("%s %s" % (id(x), id(y)))
```

For more complex objects the easiest way to clone an object is by using Python's copy module. The deepcopy function in copy clones objects. Try this:

```
import copy
class X:
    def __init__(self, x):
        self.__x = x
    def get_x(self):
        return self.__x
    def set_x(self, x):
        self.__x = x

a = X(0)
b = copy.deepcopy(a)
b.set_x(42)
print("%s %s" % (a.get_x(), b.get_x()))
```

ITERATOR

Briefly when you do

```
xs = [1,2,3,4]
for i in xs:
    print(i)
```

you are using an iterator object. The above is actually this:

```
xs = [1,2,3,4]
for i in iter(xs):
    print(i)
```

The above is actually `xs.__iter()__`.

If implemented, the `__iter()__` method allows iteration using a for-loop. For instance try this:

```
xs = [1,2,3,4]
it = iter(xs)
print(it.next())
print(it.next())
print(it.next())
print(it.next())
print(it.next())
```

Therefore the `__iter__` methods returns an iterator object whose purpose is to iterate over a sequence of values.

DEQUE

You may use the deque class (see python notes) in the collections module provided by Python. To try it out, go ahead and execute the following commands:

```
>>> import collections
>>> dir(collections)
>>> d = collections.deque()
>>> dir(d)
['__class__', '__copy__', '__delattr__', '__delitem__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__setattr__', '__setitem__', '__str__', 'append',
'appendleft', 'clear', 'extend', 'extendleft', 'pop', 'popleft', 'remove',
'rotate']
>>> d.append(1)
>>> print(d)
deque([1])
>>> d.append(5)
>>> print(d)
deque([1, 5])
>>> d.append(100)
>>> print(d)
deque([1, 5, 100])
>>> d.append(7)
>>> print(d)
deque([1, 5, 100, 7])
>>> len(d)
4
>>> d.appendleft(999)
>>> print(d)
deque([999, 1, 5, 100, 7])
>>> x = d.pop()
>>> print(d)
deque([999, 1, 5, 100])
>>> x = d.popleft()
>>> print(d)
deque([1, 5, 100])
>>>
```

Therefore the above deque class allows you to insert and remove from either end of the deque

object.

You can therefore build a stack and a queue class quickly.

USING DICTIONARIES TO RECORD THINGS

The closed list is a record of states already visited. If you use a list to record the visited states, then you might do this:

```
...
closed_list = []
...
while 1:
    ...
    closed_list.append(state0)
    ...
    if state1 in closed_list:
        ...
```

The problem is that as the closed list grows, the search time will grow too since the search in the closed list is $O(n)$ where n is the size of the list.

A faster way is to use a dictionary since the average runtime for insert, delete, and find is $O(1)$.

```
...
closed_list = {}
...
while 1:
    ...
    closed_list[state0] = 1
    ...
    if closed_list.has_key(state0):
        ...
```

The problem is that the keys of a dictionary must be immutable. The states might be lists or objects. You can convert the states (as lists) to strings:

```
...
closed_list = {}
...
while 1:
    ...
    closed_list["%s" % state0] = 1
    ...
    if closed_list.has_key("%s" % state0):
```

```
        ....
```

Note that we're only using the keys of the dictionary. The value (which in the above is always 1) is useless.

Also note that you might want to "compress" your states before putting them into a closed list. For instance in the case of the $n^2 - 1$ problem, the state is essentially a 2D array. You can convert that into a 1D array, then to a string, then strip away the `'['` and `']'`. This will save you a few characters. You can even compress it by converting the 2D array into a single integer. You can even convert your state into a bit pattern, something that is done a lot in AI. In the case of the $n^2 - 1$ problem, say the blank is replaced by a digit, you will get a 9-digit integer. If you convert the 2D array into a string, you get a string of length 33 which consumes 33 bytes. But a 9-digit integer can be just 4 bytes long on a 32-bit machine. For my machine, my python integer takes up 12 bytes and the string object of a 2D array of 3-by-3 integers takes up 54 bytes (although the length is 33). That's more than 4 times memory savings. Furthermore integer comparisons are faster than string comparison for strings of length 33.

USING SETS TO RECORD THINGS

Another way to record things would be to use a set. (Refer to Python notes.)

Note that just like dictionary keys, set values must be immutable. First try this:

```
closed = set()
state1 = [[1,2,3], [4,5,6]]
closed.add(state1)
```

You'll get an error. Read the error message. The point is that a set object in python can only store immutable objects.

Now try this:

```
closed = set()
state1 = [[1,2,3], [4,5,6]]
closed.add('%s' % state1)
print(closed)
```

So instead of storing the state as an object, you might want to store a string representation of the state.

For Python, the set is implemented as a hashtable. In fact it's a dictionary, i.e., the values are stored as keys in a dictionary. (For C++, the STL set class is a tree.)

HOW TO DECIDE: THE IMPORTANCE OF KNOWING YOUR DATA STRUCTURES

You have 3 ways to record states in the closed list: list, dictionary, and set. You have to decide which one to use. There are 2 things to consider: speed and memory.

I have already told you that python lists are like arrays. They are actually not linked lists. Search is by "linear search" (of course). This means that runtime is $O(n)$. This is case whether the python list is implemented as an array or a linked list anyway since the search is linear.

Briefly, the dictionary is a hashtable and the set is a basically a dictionary with no values. You can use the web to find out more. Knowing the general facts about a hashtable and a tree and also the list, you can decide which data structures to use. By the way, if you like you can also sort your python list and therefore improving the search performance. So here are your options:

List (array)
Sorted list (array)
Dictionary (hashtable)
Set (hashtable)

and remember that you only need to add and to search – you do not need to consider removing things from the above data structures. Therefore you need a data structure with good performance for these two operations. Now you check up the above data structures and see which one is the suitable one keeping in mind memory usage (if necessary).

If your search takes too long, it will be considered incorrect.

Q2. $n^2 - 1$ PROBLEM

Once you are done with the maze problem, the following is going to be easy because you already have most of the classes to solve the next problem, the $n^2 - 1$ problem.

The name of the program to be executed must be `main.py`. (Your solution to this problem can include other Python source files.)

Note that when expanding the state, the actions to be processed for the fringe MUST BE in the order N, S, E, W. For instance if the possible actions for a state is W, N, E, then your program must process the N action first, followed by E action, and finally by W action. Let me repeat and clarify what I just said.

Suppose `f` is the fringe and you are processing a node `n` with a reference to state `s`. On expanding `s`, you have three states `s1`, `s2`, `s3` with corresponding actions `'W','S','N'`:

`s` with `'W'` action gives `s1`
`s` with `'S'` action gives `s2`
`s` with `'N'` action gives `s3`

You must process the inserts to fringe `f` in this order:

put node containing `s3`, `'N'`, etc. into fringe `f`
put node containing `s2`, `'S'`, etc. into fringe `f`
put node containing `s1`, `'W'`, etc. into fringe `f`

Make sure you understand this. This has nothing to do with the search algorithm (tree or graph). This is just to ensure that your solution matches mine.

Again, this requirement is to ensure that your program provides the same solution as mine. If you do not follow the above ordering, your solution will differ from mine and will be considered INCORRECT.

To fix the input/output, here's an execution. Suppose you want to solve this:

0	1	2
3	4	5
6		7

with BFS search. Here an execution:

```
size: 3
```

```
initial: 0,1,2,3,4,5,6, ,7
bfs or dfs: bfs
solution: ['N', 'N', 'S', 'S', 'E', 'E', 'W', 'W']
len(solution): 8
len(closed_list): 9
len(fringe): 10
```

(The output above is incorrect. This is just to fix the output format.) The first input is the number of rows and number of columns of the puzzle, the second is the initial state of the puzzle, and the third input is bfs or dfs for BFS or DFS respectively. There are 4 outputs. Here's an explanation of the outputs:

- The first output is the solution:

  ```
  ['N', 'N', 'S', 'S', 'E', 'E', 'W', 'W']
  ```

  which is (of course) a list of actions. This means starting the initial state, if we apply action `N`, followed by `N`, followed by `S`, ... , we will arrive at the goal state. If no solution is found, the output must be

  ```
  None
  ```

In the `n2-1` directory of this download, you will find `animation.py` which contains a `draw` function that will give a graphical animation of solving the $n^2 - 1$ puzzle. Go ahead and run `animation.py`. You should see this: https://photos.app.goo.gl/JNca7kSZ97pQUAUs6 The `draw` function takes in the 2D array of the puzzle and the solution. Two examples of using `draw` is in `animation.py` (near the bottom). Once your program has computed the solution to an $n^2 - 1$ puzzle, after the above output to the console window, call `draw`. You should copy `animation.py` to your `a04q02` directory.

The following are some test cases. I am not claiming that these are the test cases I will use. I am not claiming that I will only test the case with $n = 3$.

```
Test 1: DFS
 0 1 2
 3 4 5
 6   7

Test 2: DFS
 0 1 2
 3   5
 6 4 7

Test 3: DFS
```

```
 0 1 2
   4 5
 3 6 7

Test 4: DFS
 0 1
 3 4 2
 6 7 5

Test 5: BFS
 0 1 2
 3 4
 6 7 5

Test 6: BFS
 0 1 2
 3 4 5
   6 7

Test 7: BFS
 0 1 2
 3 4
 6 7 5

Test 8: BFS
   0 2
 3 1 4
 6 7 5
```

(It's probably a good idea to work out some $n = 2$ cases complete by hand and use those for test cases too.)

DRAWING THE SEARCH [OPTIONAL]

Do the following only when you've completed the above.

It's actually quite illuminating to draw the search tree built from your search. There are many graph layout tools. The most famous one is graphviz. Google "graphviz" and you'll find information on how to install it and use it.

Modify your program so that it build a graphviz dot file. In the graph node, draw the state like this:

```
0,1,2
3,4,5
6, ,7
```

For $n = 4$, some entries can have 2 digits: you should use a window of size 2 for each entry in this case.

Use a box for the node boundary.

If a node is in the closed list, fill the node with light grey. Nodes in the fringe should be normal unfilled. Emphasize the goal state with a box with larger line width and a different color.

Save the graphviz dot file as graph.dot.

In the directory `n2-1` you will find an `images` directory. In the `images` directory you will find jpg files containing example trees of search nodes created during execution of some graph search.