Johns Hopkins University

665.645.8VL: Artificial Intelligence for Robotics

Homework and PA #1 [300 points]

PART A: Theory and Algorithms *[100 points]* ** See PART B Programming Assignment on Page 3*

Please - clearly write your **full name** on the first page. Submit a single PDF file as your answer for Part A. Please provide brief but complete explanations, using diagrams where necessary, and suitably using your own words. While presenting calculations and equations, explain the variables and answers in words.

Study Chapter 2 and Chapter 3 of Russel AI textbook Ed 3. up to page 91 only (DFS and BFS focused). Our focus is **Agents in AI** and **Search**. Answer the below questions in that order.

1. 2.4 only any four of the given activities list [4 Points]
2. 3.6 (a) and 3.6 (b) only [7 Points]
3. 3.9 (a) only [4 Points]

<u>Deepak Khemani AI Book</u> Chapter 2: State Space Search Answer the below questions

4. Question #1 (Page 50) [10 Points]

1. The *n-queens* problem is to place *n queens* on an n-by-n chessboard, such that no queen attacks another, as per chess rules. Pose the problem as a search problem.

5. Question #3 (Page 51) [10 Points]

3. In the *rabbit leap problem,* three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream with stones placed in the east west direction in a line. There is one empty stone between them.

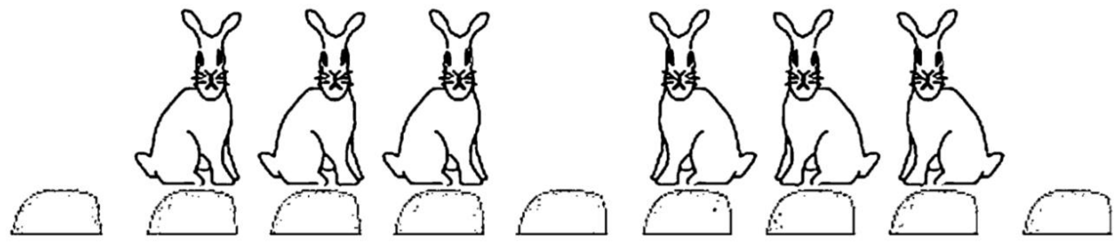


FIGURE 2.32 Rabbits waiting to cross. Each rabbit can jump over one, but not more than that. How can they avoid getting into a deadlock?

The rabbits can only move forward one step or two steps. They can jump over one rabbit if the need arises, but not more than that. Are they smart enough to cross each other without having to step into the water? Draw the state space for solving the problem, and find the solution path in the state space graph.

6. Question #10 (Page 52) [10 Points]

10. Given the *moveGen* function in the table below, and the corresponding state space graph, the task is to find a path from the start node *S* to the goal node *J*.

```
   moveGen
S → (D C B A)
A → (S B J E)
B → (S F A)
C → (S D H G)
D → (S I C)
E → (A J K)
F → (B K J)
G → (C L)
H → (C I M L)
I → (D H)
J → (A F E)
K → (E F)
L → (G H M)
M → (L H)
```

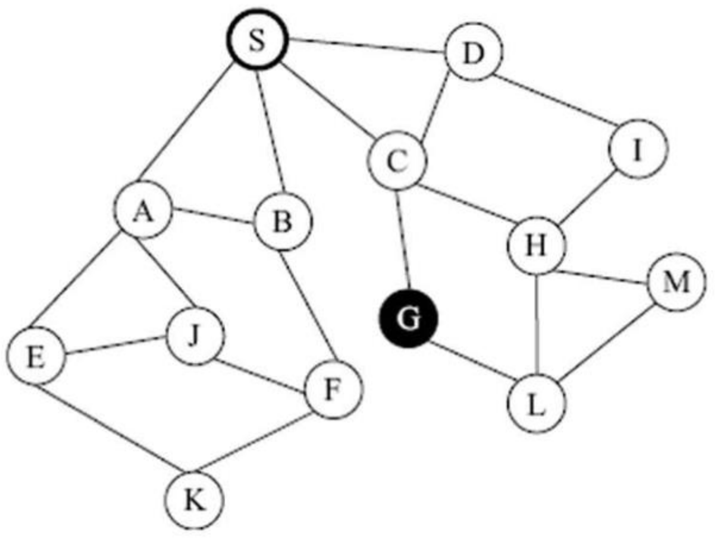


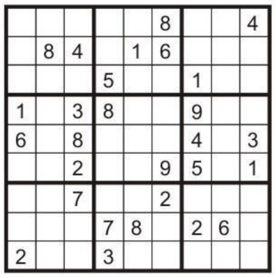
FIGURE 2.34 A small search problem. The task is to find a path from the start node *S* to the goal node *J*.

Show the *OPEN* and *CLOSED* lists for the *DFS* and the *BFS* algorithms. What is the path found by each of them?

7. **Sudoku**: see www.websudoku.com. Consider using Search to solve Sudoku puzzles: You are given a partially filled grid to start, already know there is an answer. *(credits CMU)* [25 Points]

   a. Define a state representation for Sudoku answer search. A state is a partially filled, valid grid in which no rows, column, or 3x3 square contains duplicated digits.

   b. Write a pseudocode for the successor function (generate a list of successor states from the current state) given your representation. You can assume you have functions like "duplicated" that returns true if there are two identical non-empty elements (or non-zero digits) in a collection.



					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8				4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

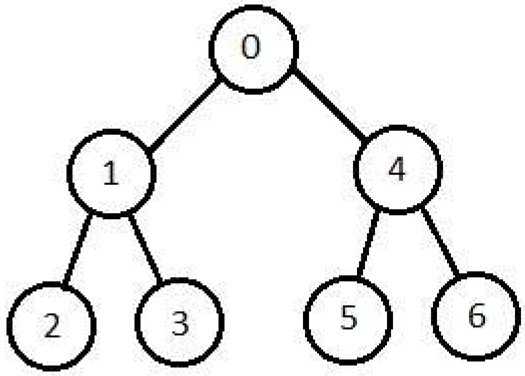
Figure 1: A sample Sudoku puzzle with 28 digits filled initially.

c. Write a pseudocode for goal function that returns true if a state is a goal. You can assume the state is reached by the successor function above so it is a valid state.

d. If the puzzle begins with 28 digits filled, what is *L*, the length of the shortest path to the goal using your representation?

8. Inspect the explanation in Python code for DFS and BFS at this tutorial. https://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/

Then, for the following basic search tree, define a state representation in detail and **First**, show by hand the paths traversed by DFS and BFS for this basic search tree. [10 Points]



Then, develop your own Python implementation, and show solution (answer) searching this tree using both DFS and BFS. It is OK to model your code after the above. [20 Points]
https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_popular_search_algorithms.htm

PART B: Implementation – Programming Assignment (PA #1) *[200 points]*

1. A* Search Algorithm in Pacman

We will be using the Pacman Project from UC Berkeley for this task. In this assignment, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build A star search algorithm and apply them to Pacman scenarios. http://ai.berkeley.edu/project_overview
Following is the link to the assignment details: http://ai.berkeley.edu/search.html

- Download code from archive zip archive.
- This project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

After downloading the code, unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in searchAgents.py is called the GoWestAgent, which always goes West (a trivial reflex agent). This agent can occasionally win:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Soon, your agent will solve not only tinyMaze, but any maze you want.

Note that pacman.py supports a number of options that can each be expressed in a long way (e.g., --layout) or a short way (e.g., -l). You can see the list of all options and their default values via:

```
python pacman.py -h
```

Also, all of the commands that appear in this project also appear in commands.txt, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with bash commands.txt.

- Attempt Questions 4,5,6,7 and 8.
- Submit the files search.py and searchAgents.py and a token generated by submission_autograder.py on BlackBoard.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your score.

2. Vacuum World Agent

In this part of the assignment, you will be writing a model-based reflex agent for the vacuum cleaner world. Implement in **Python** a performance-measuring environment simulator for the vacuum-cleaner world depicted in the figure(Figure 2.2 on page 36 of our textbook). Your implementation should be modular so that the sensors, actuators, and environment characteristics (size, shape, dirt placement, etc.) can be changed easily. You can use as a reference Russell and Norvig's AIMA home page. Feel free to start from anyone else's code but note that the continuum nature of your sensing will make for important changes in the data structure. if you are using any other code, make sure you cite the code you use. You can comment this in the code itself, showing which parts you have borrowed.

Continuum Vacuum World

Consider that the geography of the environment is not just two squares but a grid. The agent can go Up and Down as well as L and R. Consider also that the dirtiness of a grid square is not binary but is a continuum; thus the value is between 0 and 1. The objective is to pick up the maximum amount of dirt it can, within some time constraint, which is specified in terms of the number of moves. Your input includes a grid in the text file environ.txt, with a format like:

```
GRID: 8 5
 DIRT:
   0   0.5 0.8 0.1 0.1
   0.1 0   0.5 0.5 0.5
   0.3 0.5 0.4 0.3 0.2
   0.3 0.1 0.7 0.8 0.2
   0   0   0.2 0.8 0.3
   0   0   0   0.5 0.1
   0   0   0   0.5 0.1
   0   0   0   0.2 0
 MOVES: 30
 INITIAL: 3 5
```

The grid starts from (1,1) so the robot is initially on the last square of row 3 (value is 0.2).

Note that the performance measure in the book allows a 1000 moves, but we have far fewer moves here. "Suck" (S) empties all the dirt in a square, but constitutes a separate move from R, L, U, D.

You have to implement your robot in **Python**.

The input interface must read this text file from the program line:

```
vacuum_agent < environ.txt
```

At each step it should print out the move it makes and its performance score (e.g. S 0.2). Every 5 steps, it should print out a grid, with a "[ ]" on the square where the robot is now. Thus, with the above input, if the robot has done S U S L S, the output would be:

```
S 0.2
U 0.2
S 0.7
L 0.7
S 1.2

0   0.5 0.8 0.1 0.1
0.1 0   0.5 [0] 0
0.3 0.5 0.4 0.3 0
0.3 0.1 0.7 0.8 0.2
0   0   0.2 0.8 0.3
0   0   0   0.5 0.1
0   0   0   0.5 0.1
0   0   0   0.2 0
```

Please don't use any other interactions, so we can take a dump of all your output with:

```
vacuum_agent < environ.txt > output.txt
```

Performance Measure: Cleaning a dirty square = value of dirt for that square

To-do List

1. Write a reflex program as in the ordinary VacuumWorld - it has no memory, no state and can only sense the present square. It can sense the value of dirt present in and also if any of the boundary edges are walls.
2. Consider that your robot can see the four neighboring squares and use a greedy algorithm to decide which square to go to. It should move randomly when choices are equal.
3. Consider that the robot has a knowledge of its grid square (state) and also has a memory (squares visited earlier and their values etc.). It can see the four neighboring squares. Design the best-performing robot. Also provide a comparison table showing how your robots A, B and C perform.

Starting	Part A	Part B	Part C
NE(3,5)			
SW(7,1)			

Evaluation:
Your code will be evaluated on the basis of the correctness of the implementation and its performance measure.

Submission Checklist: **(see next page)**

1. search.py
2. searchAgents.py
3. submission_autograder.py
4. vacuum_agent.py
5. environ.txt
6. output_partA, output_partB, output_partC
7. Comparison table pdf