CS 534 Group Project Assignment 2 (100 Points)

Due: 11:59 p.m. on 03/19/2023

Project Objective: The goals of this assignment are to help you understand the concepts and the algorithms that we have discussed in Week 3 ~ Week 6.

Note:

*(1) This project is to be done by **EACH GROUP**. No help besides the textbook, materials, and the instructor/TA should be taken. Copying any answers or part of answers from other sources, including your classmate groups, will earn you a grade of zero.*

*(2) Your program must be developed and implemented in the PyCharm-like IDE, or 10% of the graded score is deducted. Please check and choose the one from here: https://realpython.com/python-ides-code-editors-guide/, as the suggestion. Note that we **DO NOT** use the Jupyter as the IDE. In this assignment, we just use the codes only provided by **games4e.ipynb**.*

(3) Assignments are accepted in their assigned Canvas drop box without penalty if they are received by 11:59PM EST on the due date, or 10% of the graded score is deducted for the late submission per day. Work submitted after one week of its original due date will not be accepted.

Project Deliverables: Submit a **zip** file that includes your answers for the below questions in the **pdf** file and the **.py files**, including (**Project 1**: SimpleProblemSolvingAgent.py, RomaniaCityApp.py, the map file, and other related files) and (**Project 2**: TicTacToeClass.py, TicTacToeGameApp.py, and other related files), to complete your group project assignment to Canvas.

A. Project Questions:

Question 1. Consider the map-coloring problem in Australia that we discussed in our lecture video. How many solutions are possible if two, three, and four different colors are allowed, respectively, in this problem.

Note: Each answer needs to be explained. Any reasonable assumptions can be made.

Question 2. Consider the problem of placing k queens on an $n \times n$ chessboard such that no two queens are attacking each other, where k is given and $k = n$. Formula a CSP $<\boldsymbol{V}, \boldsymbol{C}, \boldsymbol{D}>$ for this problem.

Note: Each answer needs to be explained. Any reasonable assumptions can be made.

B. Project Development

Project 1. In this project, we explore the use of local search methods to find an optimal path between any two cities in Romania shown in Figure 3.1 of your textbook, using the hill-climbing and simulated annealing.

(1). You need to extend your **SimpleProblemSolvingAgent.py** (SPSA) class in GP1 to include these two functions. Both of the functions, **hill_climbing(problem)** and **simulated_annealing(problem, schedule=exp_schedule())**, as the reference and the starting point, can be found in **search.py**. Any modifications to both functions to achieve the goal are allowed. Note that you can also develop your own functions for this agent based upon Figure 4.2 and Figure 4.5 of your textbook, respectively.

(2). The SPSA object will take a graph, i.e., the Romania map (any saved file type is fine, e.g., .txt, .json, etc., except for .py), as an input shown in Figure 3.1 of your textbook and the city coordinates, and any two cities in the map to find the best path between them using both hill-climbing and simulated annealing algorithms, respectively. The map file implementation should include the **romania_map** and **romania_map.locations** that I have shown you in my lecture slide and video as the reference and the starting point. Both **romania_map** and **romania_map.locations** have been included in the **search.py**. Note that you can develop your own graph for this map based upon Figure 3.1.

(3). You need to extend your **RomaniaCityApp.py** using the given construct and do the following:

```
def main():
    pass

if __name__ == "__main__":
    main()
```

(a) Your app will prompt and ask for a user to enter where the map file is located in your local directory and then read the **romania_map** and **romania_map.locations** in the map file. You can store your map file in any file type that you prefer only if your app can access the map file and read the **romania_map** and **romania_map.locations** in the map file.

(b) Your app will then prompt and ask for a user to enter any two cities from the **romania_map**. If these two cities are the same or either one of them or both of them cannot be found in the Romania map, please ask for the user to enter them again until the two cities are valid.

(c) Your app will then create a SPSA object from the **SimpleProblemSolvingAgent class** and search the best path between these two cities by using the **Best-First Search**, **A* Search**, **Hill Climbing** and **Simulated Annealing** algorithms, respectively. For each search algorithm used, the output should include (i) the search method name, (ii) the total cost of the path, and (iii) all the intermediate cities between them, including the start and the end cities so that you can see the difference among these four searching algorithms.

(d) At the end, your app will ask for the user if they would like to find the best path between any two cities again. If yes, repeat (b) and (c). If no, terminate the program and then display **"Thank You for Using Our App"**.

Project 2. In this project, we explore the use of the minimax search and alpha-beta pruning search algorithms to play the tic-tac-toe shown in Figure 5.1 of your textbook.

(1). Develop and implement a **TicTacToeClass.py** (TTT) class to instantiate a TTT object to play the game between the AI Player 'X' and Player 'O'. You can review and investigate the **"Game" class and TicTacToe(Game) in games4e.ipynb**, as the reference and the starting point, provided from **https://github.com/aimacode/aima-python**. Note that you can develop your own class based upon Figure 5.1 of your textbook.

(2). By default, the current game is now a 3 x 3 board. It means when a player has 3 'X' or 3 'O' in a row, in a column, or in a diagonal, that player is a winner. To make your game funnier and flexible, your version allows human users to determine the size of the game. For example, a human user can decide a game with a 4 x 4 board, 5 x 5 board, 6 x 6 board, etc. Each AI Player 'X' and 'O' should still have 3 'X' or 3 'O' in a row, in a column, or in a diagonal to win the game. The AI Player 'X' is always the first one to start the move of the game.

If you are interested (not required in this assignment), you can also allow a human user to decide the winning number of tokens as the same size of the board to win the game. For example, if it is a 4 x 4 game, each AI Player 'X' and 'O' must have 4 'X' or 4 'O' in a row, in a column, or in a diagonal to win the game. Note that it will take a longer time as usual to run and finish the game.

(3). For the TTT class, you need to develop and implement two searching algorithms: Minimax Search (i.e., **minmax_search(game, state)**) and Alpha-Beta Pruning Search (i.e., **alphabeta_search(game, state)**). Both of the functions, as the reference and the starting point, can be found in **games4e.ipynb**. Note that you can develop your own functions for the TTT class based upon Figure 5.3 and 5.7 of your textbook, respectively.

(4). Develop and implement a separate python program called **TicTacToeGameApp.py** using the given construct and do the following:

```
def main():
    pass

if __name__ == "__main__":
    main()
```

(a) Your app will prompt and ask for the size of the game from a human user.

(b) Your app will then prompt and ask for the human user to enter which searching strategy of each player 'X' and 'O' is used, respectively, in the game. The strategy could be randomly legal moves, alpha-beta legal moves, and minimax legal moves. You can find the corresponding codes in **games4e.ipynb**.

It is highly suggested to try different searching strategy combinations between the AI Player 'X' and 'O' (e.g., random vs random, random vs minimax, minimax vs alpha, and so on) so that you could see the difference among the game results.

(c) Your app will then create a TTT object from the **TicTacToeClass.py** based upon the human user's input from (a) and (b).

(d) Your game output should show each move of an AI Player 'X' or 'O' in its own turn, respectively, in sequence (see the below example as the reference). The move output can be on either a console or GUI as your own team's preference.

```
Player X move: (0, 0)
X . .
. . .
. . .

Player O move: (1, 1)
X . .
. O .
. . .

Player X move: (1, 2)
X . .
. O .
. X .

Player O move: (0, 1)
X . .
O O .
. X .

Player X move: (2, 1)
X . .
O O X
. X .

Player O move: (2, 0)
X . O
O O X
. X .

Player X move: (2, 2)
X . O
O O X
. X X

Player O move: (0, 2)
X . O
O O X
O X X


-1
```

(e) Ask for the human user if he/she would like to run the game again. If yes, repeat (a) ~ (d). If no, terminate the program and then display **"Thank You for Playing Our Game"**.

Grading Criteria: Your answers must be complete and clear.

Checkpoints	Points Possible
Project Question 1: ▪ Correct Solutions for Two, Three, and Four different colors, respectively.	10 Points
Project Question 2: ▪ Correct *CSP <V, C, D>* formulation for the *k*-queen problem.	10 Points
Project Development	80 Points
Project 1: 40 Points	
(1) Proper Naming Conventions and Program Documentation on Your Codes	5 Points
(2) Compliable Codes: ▪ SimpleProblemSolvingAgent.py ▪ RomaniaCityApp.py ▪ romania_map	6 Points
(3) Hill_Climbing() Implementation	4 Points
(4) Simulated_Annealing Implementation	4 Points
(5) The SPSA object can take any map file and prompt a user to read the Romania map	3 Points
(6) Prompt and ask for a user to enter any two cities	3 Points
(7) In case of an error, e.g., same cities or no such cities, ask the user to enter them again until the two cities are valid	3 Points
(8) The output should include the results for two cities ▪ Two Test Cities: Arad and Bucharest, Bucharest and Craiova, and Craiova and Arad ▪ Search Method Name ▪ Total Cost between Two Cities ▪ All the Intermediate Cities Between Them ▪ Ask the Users If They Would Like to Find the Optimal Path Between Any Two Cities Again ▪ Terminate the Program and Then Display "Thank You for Using Our App."	12 Points
Project 2: 40 Points	
(1). Proper Naming Conventions and Program Documentation on Your Codes	3 Points
(2). Compliable Codes: ▪ TicTacToeClass.py ▪ TicTacToeGameApp.py	10 Points
(3). Minimax Search Implementation	5 Points
(4). Alpha-Beta Pruning Search Implementation	5 Points
(5). Prompt and ask for the size of the game from a human user.	3 Points
(6). Prompt and ask for the human user to enter which searching strategy of each player 'X' and 'O' is used, respectively, in the game.	3 Points
(7). Create a TTT object from the TicTacToeClass.py based on the user's input above.	3 Points
(8). Show each move of an AI program player 'X' or 'O' in its own turn, by running their own searching strategy, respectively, in each individual game.	5 Points
(9). Ask the human user if he/she wants to run the game again. If yes, repeat the above steps. If no, terminate the program and then display "Thank You for Playing Our Game".	3 Points
Total	100 Points