CS440/ECE448 Fall 2020

Assignment 2: Planning, Games

In this assignment, you will solve a constraint satisfaction problem, and implement minimax and alpha-beta pruning algorithms for a two-player game. In Part 1, you will solve a classical CSP. In Part 2 you will practice coding minimax and alpha-beta pruning algorithms on ultimate tic-tac-toe games.

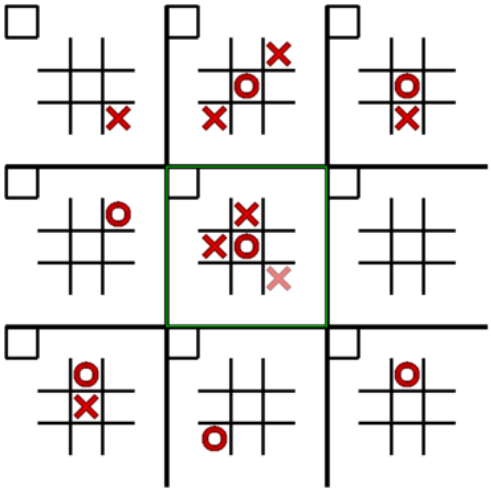

Image courtesy of the ECE448 course lecturer (Prof. Hasegawa-Johnson) at UIUC

Programming language

For Part 1, you may use modules from the Python standard library, and numpy. For Part 2, you may only use modules from the Python standard library.

Part 1: Constraint Satisfaction Problem

Your code will require the Numpy libary.

Many are familiar with the problem of pentomino tiling[1]. The problem statement is as follows: You are given a set of shapes each made up of 5 connected squares and a board of 0s and 1s. You are required to place the pentominos on the board such that

- Pentominos may be flipped or rotated
- No pentominos overlap
- All pentominos lie completely on the board
- All 1s on the board are covered by a pentomino

[1] https://en.wikipedia.org/wiki/Pentomino

- No 0s are covered by a pentomino

You will be tasked to solve this problem. To provide easier cases, you will also be tasked with tiling a board of the same size using dominos and triominos (shapes made of 2 and 3 squares respectively). The function you implement in solve.py will need to be able to tile a given board using a given set of tiles. That means that the function should work for arbitrary input tiles and boards. You may assume any instance we test on your code will have a solution. The interface of the solve function you will need to implement is described in the docstring in solve.py.

Part 2: Game of Ultimate Tic-Tac-Toe

For this part of the assignment, you will implement the game ultimate tic-tac-toe with minimax and alpha-beta pruning algorithms.

Rules

The rules of ultimate tic-tac-toe can be found on the wiki page[2]. ultimate tic-tac-toe is a board game composed of nine tic-tac-toe boards arranged in a 3-by-3 grid. Each smaller 3x3 grid is referred to as a local board, and the 9x9 grid is referred to as the global board. The player starting the game can place their move on any empty spot. After the first move, the opposing player must play in the board corresponding to the square where the first player played, as shown below:

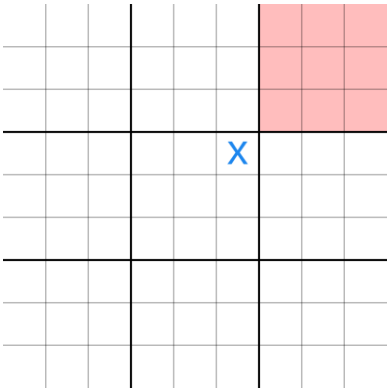

Image courtesy of the ECE448 course lecturer (Prof. Hasegawa-Johnson) at UIUC

In the above image, at starting of the game, since player "X" played at the top right corner at the central local board, player "O" is sent to the top right local board and only allowed to play there. O's next move determines the board in which X must next play, and so on.

The rules for the local board are the same as regular 3x3 tic-tac-toe. Ultimate tic-tac-toe has many variations of rules; for this part of the assignment, the winner will be the one who wins the first local board.

It's recommended that students use the online tool[3] to play the game to fully understand the rules. For this online tool, remember to select "First Tile Wins" at top left corner from the dropdown list.

[2] https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe

[3] http://ultimatetictactoe.creativitygames.net/

Requirements

The global game board has 9 local boards with 81 available spots. The coordinate of each empty spot is the row and the column number starting with 0. Each local board has an index starting at 0. The first row of local boards has index from 0 to 2, the second row of local boards have index from 3-5, and so on.

For this part of the assignment, you will first implement the predefined offensive agent and the predefined defensive agent to play against each other. In the provided skeleton code, the maxPlayer is the offensive agent, and the minPlayer is the defensive agent. The game always starts at the central local board (index 4). The evaluation function for each predefined agent is prioritized to three levels. If the higher level of evaluation rules is applied, then the lower level of evaluation rules won't be used. For example, if the second rule is applicable, then the third rule won't be used. Minimax or alpha-beta search always search the local board row by row from top left corner to bottom right corner. The search depth has maximum three levels, including current player, opposing player, and current player again. Then evaluate the resulting board position after the third level search. If the evaluation function returns two terminal states with the same value, choose whichever of the two moves comes first in this raster search order.

Predefined offensive agent:

- First rule: If the offensive agent wins (form three-in-a-row), set the utility score to be 10000.
- Second rule: For each local board, count the number of two-in-a-row without the third spot taken by the opposing player (unblocked two-in-a-row). For each unblocked two-in-a-row, increment the utility score by 500. For each local board, count the number of places in which you have prevented the opponent player forming two-in-a-row (two-in-a-row of opponent player but with the third spot taken by offensive agent). For each prevention, increment the utility score by 100.
- Third rule: For each corner taken by the offensive agent, increment the utility score by 30.

Predefined defensive agent:

- First rule: If the defensive agent wins (forms three-in-a-row), set the utility score to be -10000.
- Second rule: For each local board, count the number of two-in-a-row without the third spot taken by the opponent player. For each two-in-a-row, decrement the utility score by 100. For each local board, count the number of preventions of opponent player forming two-in-a-row (two-in-a-row of opponent player but with the third spot taken by defensive agent). For each prevention, decrement the utility score by 500.
- Third rule: For each corner taken by defensive agent, decrement the utility score by 30.

You tasks are the following:

- First implement the predefined offensive and defensive agents to play against each other for four games: offensive(minimax) vs defensive(minimax), offensive(minimax) vs defensive(alpha-beta), offensive(alpha-beta) vs defensive(minimax), and offensive(alpha-

beta) vs defensive(alpha-beta). In the first two games, maxPlayer goes first, while in the rest of the games, minPlayer goes first. Report the final game board positions, number of expanded nodes, and the final winner for each game.

- Then come up with a smarter evaluation function to beat the offensive agent at least 80% of the time. Play the game with alpha-beta pruning for both agents for at least 20 times, with both the starting local board index and the starting player set randomly. In your report, explain your formulation, and the advantages of your evaluation function. Report the percentage of winning time and number of expanded nodes for each game. Report 3-5 representative final game boards that show the advantage of your evaluation function vs predefined offensive evaluation function.
- Let a human play with your own defined evaluation function for at least 10 games. Report 3-5 representative final game boards that show the advantages or disadvantages of your evaluation function. Discuss the results and findings. Do human always win over the agent? If so, what does the evaluation function fails to do?

Tips

- Use the online tool to farmilarize yourself with the rules of the game, meanwhile gain some intuition of designing a good evaluation function.
- Make sure the predifined agents are implemented as described. Your implementation will be graded on percentage of correctness.
- Determine critical steps for coming up good evaluation function and transform your thoughts to math equations.

Provided Code Skeleton

We have provided (tar file or zip file) all the code to get you started on your MP. **Do not modify provided code.**

For part 2.1, you are provided the following:

- solve.py- This is the only python file you will need to modify.
- Pentomino.py- This contains useful verification algorithms.
- instances.py- This contains example pentomino sets and boards.

For part 2.2, you are provided the following: uttt.py

This is the only python file you need to modify to get your code running. The doc strings in the python file explain the purpose of each function.

- evaluatePredifined() : This function implements the evaluation function for ultimate tic tac toe for predifined agent.
- evaluateDesigned() : This function implements your own evalution function

- checkMovesLeft() : This function checks if there is any legal move remains on the board.
- checkWinner() : This function checks if any player wins the game.
- alphabeta(): This function implements alpha-beta pruning algorithm for ultimate tic-tac-toe.
- minimax(): This function implements minimax algorithm for ultimate tic-tac-toe.
- playGamePredifinedAgent(): This function implements the game of predefined agents play against each other.
- playGameYourAgent(): This function implements the game of your agent play against the predefined offensive agent.
- playGameHuman(): This function implements the game of human play against your agent.

Extra Credit Suggestion

Ultimate tic-tac-toe has multiple versions. Once each local board is won, the local board will be marked as victory to the corresponding player. The game terminates when three local boards form a row. If the local board has been won, no more move can be placed at that board. Implement this game with your own offensive and defensive agents. Play the game for at least 10 times. Explain your evaluation functions and report 3-5 representative final game board positions. Note that the extra credit will be capped to 10% of what the assignment is worth.

Deliverables

This MP will be submitted via Blackboard.

Please upload **only the following three files** to Blackboard.

1. solve.py - your solution python file to part 1
2. uttt.py - your solution python file to part 2
3. report.pdf - your project report in pdf format

Report Checklist

Your report should briefly describe your implemented solution and fully answer the questions for every part of the assignment. Your description should focus on the most "interesting" aspects of your solution, i.e., any non-obvious implementation choices and parameter settings, and what you have found to be especially important for getting good performance. Feel free to include pseudocode or figures if they are needed to clarify your approach. Your report should be self-contained, and it should (ideally) make it possible for us to understand your solution without having to run your source code.
Kindly structure the report as follows:

1. **Title Page:**
   List of all team members, course number and section for which each member is registered, date on which the report was written

2. **Section I:**
   CSP: We would like a description of your representation of the problem as well as an explanation of the algorithm (as well as heuristics) that you used to find a solution.

3. **Section II:**
   Ultimate Tic-Tac-Toe: for each of the four games of predefined agents, report the final game board positions, number of expanded nodes, and the final winner.

4. **Section III:**

   Ultimate Tic-Tac-Toe: for at least 20 games of offensive agent vs your agent, explain your formulation and advantages of evaluation function. Report the percentage of winning time and number of expanded nodes for each game. Report 3-5 representative final game boards that show the advantage of your evaluation function vs predefined offensive evaluation function. If your own defined agent fails to beat the predefined agent, explain why that happened.

5. **Section IV:**

   Ultimate Tic-Tac-Toe: for at least 10 games of human vs your agent, discuss your observations, including the percentage of winning time, the advantages or disadvantages of your defined evaluation functions. Report 3-5 representative final game boards that show the advantages or disadvantages of your evaluation function.

6. **Extra Credit:**
   If you have done any work which you think should get extra credit, describe it here

7. **Statement of Contribution:**
   Specify which team-member performed which task. You are encouraged to make this a many-to-many mapping, if applicable. e.g., You can say that "Rahul and Jason both implemented the BFS function, their results were compared for debugging and Rahul's code was submitted. Jason and Mark both implemented the DFS function, Mark's code never ran successfully, so Jason's code was submitted. Section I of the report was written by all 3 team members. Section II by Mark and Jason, Section III by Rahul and Jason. ".. and so on.

WARNING: You will not get credit for any solutions that you have obtained, but not included in your report!

Your report must be a formatted pdf document. Pictures and example outputs should be incorporated into the document. Exception: items which are very large or unsuitable for inclusion in a pdf document (e.g. videos or animated gifs) may be put on the web and a URL included in your report.

Extra credit:

We reserve the right to give **bonus points** for any advanced exploration or especially challenging or creative solutions that you implement. This includes, but is not restricted to, the extra credit suggestion given above.