CS 470/670 – Intro to Artificial Intelligence – Spring 2021
Instructor: Marc Pomplun

Assignment #4

Posted on April 15, Due by May 4 at 4:00pm

Question 1: Solving the 2x2x2 Rubik's Cube

Download the attached **rubik_lab_assignment_4.py** and **graphics.py** files and put them in the same directory on your computer. Running the first file should give you the Rubik Lab that allows you to enter cube configurations, make moves, and test your algorithm for solving the cube, including the h' scores that your h'-function returns for any shown cube state. Please check the Zoom recording from April 6 for a demo and further explanations.

Your only task is to improve the currently implemented h'-function, which simply returns 0 if the cube is in a solved state and 1 otherwise. As you know, this turns A* into breadth-first search, which is very inefficient. Please improve the h'-function to make it more efficient or use a different algorithm for solving the cube altogether. It is up to you. As long as you submit a modified code that can solve the cube more efficiently than the current code, you will receive full points.

However, please note that this is a great opportunity to think about and experiment with AI techniques to solve an interesting problem. You got the actual Rubik's Cubes to make this experience even more hands-on. Please give it a try and email me anytime if you have any questions, want to show me some code or ideas, or run into any problems.

Question 2: Hold Your Horses – the Game Tournament

Your task is to develop a Python game playing algorithm that will compete in the game tournament that we will hold (at least partially) during our May 11 class. The rules are pretty simple: Each player has a number of horses, which move like the knights in chess – jump two squares up, down, left, or right and then one square in perpendicular direction. They cannot land on squares occupied by their own team, but they can move to empty squares or squares occupied by the opponent. In the case of an opponent's horse, that horse is kicked out of the game. If it is the opponent's apple or if the opponent has no horses left on the board, the player on the move has won. If no player wins after a total of 40 moves (20 per player) have been played, the game is drawn and each player receives 50 points. Otherwise, the winning player's score is 100 plus the number of moves that could have been played before resulting in a draw. For example, if Player 1 wins after 25 moves have been played, Player 1 will receive 115 points, and Player 2 will receive 0 points.

Download the attached **HoldYourHorses.py**, **graphics.py**, **Knight_Rider,py**, **Brain_Fog.py**, and **Dark_Knight.py** files and put them in the same directory on your computer. You should take the **Knight_Rider.py** code, name it after yourself in the form **John_Doe.py** (but use your own name), and modify it to create your own game playing algorithm.

Knight_Rider uses iterative deepening to make the best use of the time available for thinking (currently set to 3 seconds per move), as we discussed in class. It applies the Minimax algorithm without alpha-beta pruning. It will always use the full time available unless you limit its maximum lookahead depth. Its evaluation function considers the number and positions of horses as well as the points won for a terminal game state. The second sample player, **Brain_Fog**, simply picks a random legal move and never thinks for more than a few milliseconds. In contrast, the third player, **Dark_Knight**, is quite strong using a secret evaluation function and other optimizations. Its code is obfuscated and precompiled so that it would be difficult to reverse engineer. Please spend your time thinking about how to optimize your own player instead of trying to figure out what **Dark_Knight** does.

You can use **Knight_Rider** and **Brain_Fog** and any of their variants you may create to test your own algorithm. At the bottom of the **HoldYourHorses.py** code, you can add the name of any new player files to **playerModuleList** and then include them in individual games or round-robin tournaments. In each case, refer to your player by its position in **playerModuleList** (e.g., the first player in the list is associated with number 0). Note that in round-robin tournaments, any two players meet twice so that each one has the first move once. Also notice that you, as a human, can play a game against any computer player as well. Just call **singleGame** with **-1** instead of a player index as the first or second argument. You can pick a horse with your mouse and then pick a target square to execute the move. If you chose a horse but want to pick a different one, just click on the same square again.

And here are your tasks:

(a) Add alpha-beta pruning to your own player program.

(b) Improve the scoring function **getScore** in your code so that your player will usually beat **Knight_Rider**.

(c) **Bonus:** Make further improvements to **getScore** or any other part of your player code to make is as competitive as possible for the tournament!

You can only use standard Python modules that are included when you install Python; the only non-standard module allowed is numpy. No ctypes allowed. Your code must consist of only one file and must run properly with the original HoldYourHorses.py file. Only modify the local copies of **getMoveOptions** or **makeMove** in your player code if you like, but leave them unchanged in HoldYourHorses.py. Make sure to check **timeOut()** in your code regularly to avoid time violations, in which case the interface will simply play the first legal move in the list.

Note: The current game parameters, such as number and initial placement of horses, board size, and time and move limits, will also be used in the tournament.

Please remove all **print** statements from your code. The **print** statements in **Knight_Rider** are intended only for your experimentation and debugging.

Here are some hints for writing a competitive player:

- In **getScore**, just measuring the Manhattan distance between a horse and its opponent's apple is not very useful; it may be better to consider how many actual moves the horse would need to eat that apple. Measuring distances between squares in this manner might make sense for other purposes as well.

- Protecting those squares from which the opponent could reach the player's apple is important. If an opponent's horse can get there without any of the player's horses being able to kick it out, the game is lost. More generally, if the player has fewer horses than the opponent that can jump to such a square within a single move, the opponent cannot be stopped from eating the apple.

- The point above obviously not only applies to protecting but also attacking. In general, players have to strike a balance between attack and defense. **Knight_Rider**'s assumption that being closer to the opponent's apple is better is not necessarily valid, especially if we forget to protect our own apple in this pursuit.

- The attack vs. defense balance should also take into account the maximum number of moves that can be played before the game is drawn. A win usually results in more than 100 points, whereas a draw only gives exactly 50 points. This means that winning 50% of all games and losing the other 50% will typically result in more points than drawing 100% of all games. Therefore, it seems that a slight bias favoring attack over defense would be beneficial to your player's performance.

- In the iterative deepening approach, when time runs out, Knight_Rider forgets everything about the current iteration and returns the best move that it found in the previous one. For example, it completes search at maximum search depths of 1, 2, and 3, and while it is working on maximum search depth 4, it gets the timeout signal. Then it simply picks the best move that it found for maximum search depth 3 and plays it. This is generally a good idea, because the result of an incomplete search can be misleading.

  However, there are situations when the partial results can still be useful. For example, assume that the result of maximum search depth 3 was a move, say, move 5 in the list of 30 legal moves, with a disadvantage for our player. Now in the next iteration (depth 4), we already checked the first 10 out of the 30 possible moves when we reached the time limit, so we have to stop the computation. However, move number 7 that we already analyzed now shows a huge advantage (maybe even a guaranteed victory) for our player. In such a case, our player should pick move 7, even though it resulted from an incomplete search of the game tree.

- Optimize your code in order to analyze more configurations per second.

- Remember that sorting the newly created nodes in the game tree using a heuristic, similar to A*, can improve the effectiveness of alpha-beta pruning.

- Test variants of your player against each other and the two players provided to find the strongest approaches and parameters for your final submission. You could also compute a feature vector from the game state and use machine evolution to optimize a weight vector, where your e(p) score is the dot product of the feature vector and the weight vector.

It should go without saying that you are not allowed to copy code from anyone or anywhere, except for the code provided for this assignment. You can always discuss ideas with other people, and you can do online research to study solutions for problems, but you have to write your own code. Anything else would be plagiarism (and prevent you from seeing your own code and ideas compete in the tournament).

Let us discuss any question you may have on Blackboard and during our online sessions.

Most importantly, have fun!