AI Final Exam Fall 2018

Name ___________________________________

Choose 1 problem per category below. Each completed task is worth 20 points out of 100. There is an exam download on Canvas that should be unpacked where it has access to your RubiksCubeSolver program. It contains scripts and data files for the Rubik's cube problems.

Rubik's Cube Problems (Choose 1)

[ ] 1. **About Face:** Modify your Rubik's cube solving program to allow rotating faces 180 degrees, as well as the 90 degree clock-wise and counter clock-wise rotations already supported. The rotate command must be modified to support the following syntax:

`rotate color direction`

`color` is any of the six colors: red, green, white, yellow, orange, blue
`direction` is any of the directions: cw, ccw, 180
This work should only require the Rubik's cube class and your driver program to be modified. Pass-off using the `pass_off_about_face.bash` script. This script needs to run where ./`RubiksCubeSolver` is available.

[ ] 2. **Middle Face:** Modify your Rubik's cube solving program to allow rotating one of the 3 middle axis, as well as the rotations already supported. The rotate command must be modified to support the following syntax:

`rotate color direction`

`color` is any of the six colors: red, green, white, yellow, orange, blue
`direction` is any of the directions: cw, ccw
Additionally if color is red, direction may be up, down, left or right to signal direction of travel of the center red piece. If the color is blue, up or down are also legal. This will produce a total of 6 additional new actions. When you implement this, don't actually move the middle pieces, instead move the two faces appropriately.
This work should only require the Rubik's cube class and your driver program to be modified. Pass-off using the `pass_off_middle_face.bash` script. This script needs to run where ./`RubiksCubeSolver` is available.

[ ] 3. **Any Color:** Modify your Rubik's cube solving program to allow faces to be specified as "any color", using the `*` character. The `isequal` command must allow face colors to be specified as any color. For equality comparison, the cube must consider the "any color" to match any other color. This work

should only require the Rubik’s cube class and maybe your driver program to be modified. Pass-off using the `pass_off_any_color.bash` script. This script needs to run where ./`RubiksCubeSolver` is available.

Rubik’s Cube Search Problems (Choose 1)

[ ] 4. **About Face Solve:** This work requires the **About Face** problem to be solved first. Modify your Rubik’s cube solving program to solve cubes using the 180 degree rotations as well as the 90 degree rotations. This work should only require work in your `Action` and `Problem` classes. Pass-off using the `pass_off_about_face_solve.bash` script. This script needs to run where ./`RubiksCubeSolver` is available, as well as `exam_about_face_cubes`. It uses graph astar search. I realize your heuristic may not be as accurate now. Don’t worry about it.

[ ] 5. **Middle Face Solve:** This work requires the **Middle Face** problem to be solved first. Modify your Rubik’s cube solving program to solve cubes using the additional 6 actions. This work should only require work in your `Action` and `Problem` classes. Pass-off using the `pass_off_middle_face_solve.bash` script. This script needs to run where ./`RubiksCubeSolver` is available, as well as `exam_middle_face_cubes`. It uses graph astar search. I realize your heuristic may not be as accurate now. Don’t worry about it.

[ ] 6. **Any Color Solve:** This work requires the **Any Color** problem to be solved first. Modify your Rubik’s cube solving program to allow for a goal state to be specified. The goal command has the same syntax as the `initial` command, except the keyword is `goal`, and the faces may be described with the any color syntax, `*`. This work should require work mostly in your `State` and `Problem` classes and the driver program. Pass-off using the `pass_off_any_color_solve.bash` script. This script needs to run where ./`RubiksCubeSolver` is available, as well as `exam_any_color_cubes`. It uses depth limited tree search.

Vertex-Cover Local Search Problems (Choose 1)

[ ] 7. **Good ‘nough:** Modify the LocalProblem::GoodEnough method to return true if all edges are covered and the cover set size is less than or equal to ¾ of the number of vertices in the graph. Complete enough runs with this change to show the difference in vertices generated compared to your studies during the homework project. Commit and push all code changes. Show the difference (either in graph or text form) to pass off.

[ ] 8. **Thin to win:** Modify the LocalProblem::RandomState method to only have a 10% chance of adding a vertex to the starting cover set. Complete enough runs with this change to show the difference in vertices generated compared to your studies during the homework project. Commit and push all code changes. Show the difference (either in graph or text form) to pass off.

[ ] 9. **Double-stuff:** Modify the Model::getValue method to add to v the number of double covered edges. This is true whether the number of uncovered edges is 0 or not. If your assignment already did this, then remove it from getValue. Complete enough runs with this change to show the difference in vertices generated compared to your studies during the homework project. Commit and push all code changes. Show the difference (either in graph or text form) to pass off.

Ultimate-Tic-Tac-Toe Adversarial Search Problems (Choose 1)

[ ] 10. **Northwest:** Make a new board evaluation function for your UTTT agent. The evaluation uses this formula: A winning board evaluates to 1.0, a losing board to -1.0, and a tie board to 0.0. If the board is not complete, then the board's value is the sum of the values of the subboards. A subboard's value is 1./9. if it's a win, -1./9. if it's a loss, and 0.0 if it's a tie. If the subboard is not complete, the value is the sum of the value of its squares. The squares are positive value if owned by the player, negative if owned by the other player and 0 if not owned. The magnitude of the value is 16./441. for the north-west cell, 8./441. for the two adjacent cells, 4./441. for the 3 adjacent to them, 2./441. for the 2 adjacent to them and finally 1./441. for the south-east corner. Allow this evaluation function to used and play your agent against itself, one with your evaluation function and one with this evaluation function. Commit and push all code changes. Demonstrate the game play to pass off.

16./441.	8./441.	4./441.
8./441.	4./441.	2./441.
4./441.	2./441.	1./441.

[ ] 11. **Other Side of the Mountain:** Modify your UTTT agent so that it spends more time working in the middle stages of the game than at the beginning and the end. Specifically, use this equation to decide how many play-outs or how much time to spend on a turn. Let t be the turn number for this player, t_1 be the maximum amount of work per turn, $t_0 = t_1/4$ be the minimum amount of work per turn, L be the expected number turns per player, $g = 3 * t_1 / (2 * L)$, then $f(t) = t_0 + g * t$ is the amount of work for turn t. For example you may use L = 20 and t1 = 40000 to configure the amount of work. Play this strategy against your normal strategy to observe the difference. Commit and push all code changes. Demonstrate the game play to pass off.

Wrap-up (Choose 1)

[ ] 12. **Pick-one:** Pick one of the uncompleted tasks from a previous group to complete.