<u>**The code and supporting files:**</u>

Download and extract **lab2_part1_starter.zip** from Schoology. There are a lot of files, most of which you don't need to read or edit (yet).

Files you'll <u>*edit and submit*</u> in this part of the lab:

- **game_algorithms.py**
- **eval_functions.py**

Files you'll want to *read and understand*:

- **gamesearch_problem.py**

Files you'll run to test your code:

- **test_game_gui.py**
- **play_game_gui.py**
- **play_game_console.py**

Other files you'll use:

- The folder **tictactoe_states** that has **tictactoe_[DESCR].txt** initial state files
- The folder **nim_states** that has **nim_[DESCR].txt** initial state files
- The folder **connectfour_states** that has **connectfour_[DESCR].txt** initial state files
- The folder **roomba_states** that has **roomba_[DESCR].txt** initial state files
    - Feel free to add some more of your own text files to any of these for testing!

When you submit your lab via Schoology, only **game_algorithms.py** and **eval_functions.py.** Be sure to include the names of both you and your partner at the top of the files. Only one partner needs to submit.

Part 0: Getting Comfortable, ReflexAgent, Evaluation Functions

1. Run the following to open the testing GUI:

   ```
   python test_game_gui.py tictactoe default
   ```

   To open a text file that defines a specific initial state, change out `default` for a filepath, like

   ```
   python test_game_gui.py tictactoe tictactoe_states/FILE.txt
   ```

   (You can also change out `tictactoe` for `nim`, `connectfour`, or `roomba`, but we'll focus on Tic Tac Toe for a little while)

   Get familiar with this GUI. You can click on the board to input actions and change the displayed state. From any state, you can test out different game search algorithms that you've written using the bottom panel of options.

   Random is the only working agent to start. You'll see that the algorithm simply picks a random action for that state, with no regard for the potential utility – value – of the resulting state.

2. Take a peek at **gamesearch_problem.py.** You'll notice the StateNode and Action objects are very similar to the last lab, with a few differences that are noted in the header comments.

3. Your first task is to implement **ReflexAgent** defined in **game_algorithms.py**

   ReflexAgent is the first of several GameAgent subclasses you will implement. All the GameAgent classes have the method pick_action() which must be defined so that, for a given state, it will return a chosen action, as well as as well as the estimated value of choosing that action, and the future state from which it expects to achieve that value. (This is a tuple of size 3)

   A ReflexAgent is not a search agent – it merely picks one action based on the value of the states that its actions immediately lead to.

   To compute the value of a state, we need to use an evaluation functions that, for terminal endgame states, it will return the true utility, and for non-terminal states it will make a (conservative) estimate.

   All the agents you will define have access to its given evaluation function with self.evaluation_fn, which takes two parameters – the state to evaluate, and the index of the player from whose perspective we want the utility.

   ReflexAgent should be short and simple – for every action, generate the resulting state and use self.evaluation_fn to evaluate that state. Return the best (action,value,state) tuple.

4. There are 3 predefined evaluation functions ("custom eval" not yet implemented) for Tic Tac Toe that you can test in the GUI. Test them with the "Print State Eval" button on various states.

The functions are defined towards the top of the file **eval_functions.py**, and listed in the dictionary tictactoe_functions. You don't really need to study them deeply at this point, but you can look if you want.

Then, try running ReflexAgent from various positions using each function. Both "endgame score" and "faster endgame score" only give 0 for non-endgame states, so you'll only really see smart choices at close-to-endgame positions, but "spaces weighted" tries to differentiate between the non-endgame states.

You've written an "intelligent" game agent! But, we can clearly do better.

Part 1: Game Tree Search

1. Your next task is to implement **MaximizingSearchAgent**, **MinimaxSearchAgent**, and **ExpectimaxSearchAgent** in **game_algorithms.py**

   All three are subclasses of **GameSearchAgent**. The only method you need to implement is pick_action(), similar to ReflexAgent; however, you're likely going to want to define additional helper methods for each. Take a look at the comment for GameSearchAgent's pick_action() for more important details.

   The code for each agent will be extremely similar – it is probably best to focus on getting one right first (MaximizingSearchAgent is easiest) before using it to start the next. There are ways to share some code between them, if you utilize inheritance.

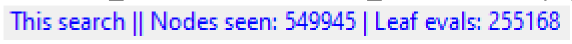
   NOTE: Do NOT utilize a transposition table or alpha beta pruning; ***these are full tree searches***.

2. Test each of your algorithms with the test GUI on TicTacToe on close-to-endgame situations with Depth Limit set to INF and the evaluation function **endgame score**. You can either click the board to input a few moves before searching, or try some of the provided initial state txt files like **tictactoe_winning.txt**, **tictactoe_losing.txt** or **tictactoe_slighthope.txt** to compare their behaviors.

   Each of your three algorithms should be modeling different opponent behavior:

   - Maximax should act optimistically, as if it assumes a benevolent opponent,
   - Minimax should act pessimistically, as if it assumes a purely adversarial opponent
   - Expectimax should act cautiously optimistic, as if it assumes a random acting opponent.

   Some Notes: After a search finishes, you can toggle between seeing the suggested move and the expected path. (Expectimax, however, will and should not return an expected path)

3. If you run a search from the default, empty board and use Depth Limit = INF, it will take a while, especially with state visualizations. If you use FLY BLIND SEARCH, it may take around 20-30 seconds (be patient, your GUI will temporarily appear to not respond). If you incremented self.total_counts and self.total_evals correctly, you should see
   
   This search || Nodes seen: 549945 | Leaf evals: 255168
   for each of the searches.

4. Compare evaluation functions **endgame score** and **faster endgame score**. Using **faster** should, in winning situations, try to win sooner and, in losing situations, try to lose slower.

   Try some of the provided initial state txt files like **tictactoe_winning_faster** or **tictactoe_losing_slower** or **tictactoe_slighthope** to compare the two utility functions.

5. Test setting Depth Limit to a shallower depth, like 4. We will need an evaluation function that actually estimates non-endgame evaluations. The **spaces weighted** evaluation function is already defined, which estimates value by "weighting" certain positions as more helpful.

   This heuristic is okay – they provide some information but are, of course, only an *estimate* of the expected utility of future endgame states. So, expect that the returned actions are different from

   Open **eval_functions.py** and find the method **space_weights_eval_tictactoe**. Try tweaking the space_weights dictionary to adjust the weight of each space, and see if you can produce different search behavior with each of the different search algorithms.

6. You now have some decent TicTacToe bots, which can play to a shallow depth of about 4 in reasonable time.
   Try playing a game against your bots – or pit them against one another! Run:

   ```
   python play_game_gui.py tictactoe default
   ```

   or

   ```
   python play_game_console.py tictactoe default
   ```

   Follow the prompts on the console to set up your agents, including the appropriate evaluation function and depth limit. For the gui, saying "no" for "Show search process?" will speed things up a lot (though you won't get to see the callback visualizations). The console version by default does not show callback visualizations, and will run faster than the gui, too.

7. At this point, you might try out the Nim or Roomba games; if you wrote your algorithms for the general StateNode and Action, they will also apply to the other games!

   You could try the ConnectFour game, but your current algorithms will be disappointingly slow and unable to search very deep. So, let's write some better ones!

   NOTE: One thing that we could do to reduce nodes/time and potentially increase our reasonable depth limit is to add a transposition table to our algorithms to avoid re-computing similar subtrees. This is helpful, but somewhat tricky to get right and memory expensive if not managed well. Ultimately, the value of transposition tables is mostly in repeated searches, which we will address later… Consider general transposition tables something you could add as an extension later; for now, we will set this idea aside.

Rough Checklist Rubric:

search_algorithms.py

- ✓ (10%) The method pick_action() in ReflexAgent works correctly, including:
    - Returns valid Action
    - Returns expected value of resulting state according to self.evaluation_fn
    - Returns expected resulting StateNode

- ✓ (30% each agent) The method pick_action() for **MaximizingSearchAgent**, **Minimax**, and **Expectimax** each works correctly, including:
    - Correctly searches to self.depth_limit
    - Returns valid and "optimal" Action according to each's model of the opponent.
    - Returns expected value of resulting state according to self.evaluation_fn
    - Returns expected resulting leaf StateNode (except for Expectimax)
    - Calls self.gui_callback_fn for each state in the search tree, passing its computed value
    - Terminates and returns None *promptly* when self.gui_callback_fn returning True
    - Updates self.total_nodes and self.total_evals correctly

This assignment (part) is worth 10 weighted points