The code and supporting files:

The files involved are the same files from Part 1 and Part 2, except for:

- A folder of some **roomba_[DESCR].txt** initial state files

Files you'll *edit and submit* in this part of the lab:

- **lab2_algorithms.py**
- **lab2_util_eval.py**

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

- **gamestatenode.py**
- **roomba_gamestate.py**

Files you'll run to test your code:

- **lab2_test_gui.py**
- **lab2_play_gui.py**
- **lab2_play_text.py**

When you submit your lab via Schoology, only submit **lab2_algorithms.py and lab2_util_eval.py.**

Be sure to include the names of both you and your partner. Submit individually.

Part 3: Progressive Deepening, Emergent Behaviours for Roomba

1. All of our algorithms so far must run to completion before usable results can be obtained. Searching to shallow depths is fast but usually not very accurate; searching to deeper levels of the game tree generally yields better policies but could take too long to finish. Guessing exactly how deep we can afford to search in the time available is impractical and risky; we need an approach that can return a usable result *anytime* we need it.

   The solution is to adapt an *iterative deepening* approach (aka "progressive deepening"). Perform any game-tree search with a cutoff depth of 1 and save those results. If time remains, repeat with a cutoff depth of 2, then 3, etc... Continue until you require an answer and then use the policy from the deepest search.

   Open **lab2_ algorithms.py**. You need to write **ProgressiveDeepening** so that it performs iteratively deeper minimax searches w/ alpha-beta pruning. It should stop searching as soon as possible when either A) the **time_limit** is up, or B) **state_callback_fn** returns True, and then return its results. What is returned differs from the other algorithms (namely, it returns lists of completed search results), and is detailed in the comments.

   Test each your ProgressiveDeepening with various games – make sure it terminates in the time limit and if the terminate button is hit. The algorithm will print a history of *all* its depth-increasing searches to the console if you return well-formed lists,

2. Another benefit of progressive deepening is that information gained in shallow searches can be used to direct the deeper searches. This is particularly useful for alpha-beta pruning, where action-ordering can make a big difference for pruning.

   Update **ProgressiveDeepening** so that, when the transposition table parameter is True, it uses results from previous searches to order moves in later searches.

   1. After each level's search completes, add the values in the transposition table to a persistent master transposition table (overwriting any old values, since values calculated from deeper searches are probably better).
   2. During the recursive part of the search, order the moves explored by these values, from most promising moves to least.
      Use **generate_next_states_and_actions**() to get a list of (action, child_state) tuples, then sort that list with the built-in **sorted()** method, which can sort by a function you provide.
      That function should return either
      i. If it is in the master transposition table, the saved value of the child-state.
      ii. If it is NOT in the master transposition table, the current alpha or beta value, depending on whether it is the maximizer or minimizer's move at the state. (Consider: why use alpha/beta instead of simply using 0 or +/- INF?)

   Test each your ProgressiveDeepening using a transposition table. Compare the results to doing a MinimaxAlphaBetaSearch at the same max-depth. Most likely, ProgressiveDeepening will have searched fewer nodes and done fewer evaluations on the final cutoff than MinimaxAlphaBetaSearch because of the improved move ordering. In fact, in some cases, the *total* number of searched nodes and evaluations will be less than the raw max-depth search!

If you wrote a decent heuristic for Connect Four, ProgressiveDeepening with a transposition table should play pretty close to average human level play. Given about 5 seconds to think, it can usually search to 6-8 levels deep in the early game, closer to 10-12 in the late game.

```
python lab2_play_gui.py connectfour default human progressive
```

3. With alpha-beta pruning and a transposition table, the default board for Roomba Race (5-by-7) can be solved to the endgame in a few seconds. However, for larger boards (say, 10x10) it becomes necessary to develop heuristics for shallower cutoffs. The true value of any position is difficult to estimate, but if we design relatively simple heuristics to encourage certain "strategies," we can produce decent (and somewhat human-like) emergent behavior that will hopefully get the agent to play well enough until it can search to the endgame.

   Open **lab2_util_eval.py**. You need to implement the heuristic evaluation functions **aggressive_eval_roomba** and **defensive_eval_roomba.**

   The former should give higher values to states where the two agents are closer, therefore encouraging an "aggressive" play style. This is simple to compute but surprisingly effective against random bots and even humans.

   The latter should produce a safer, more "defensive" play style. Exactly how this is done is up to you, but generally speaking the more movement options the agent has, the safer it is.

   Then, test your heuristics with the various algorithms at lower Cutoffs, including ProgressiveDeepening with short time limits. Use some of the different initial states, which are all larger boards (10x12) than the default 5x7, to confirm that the heuristics are producing the emergent behavior desired. The emergent behavior will eventually get eclipsed by optimal behavior in the late game when the algorithms are able to see the endgame states.

   Try pitting two bots against each other to see the results! You can use random move ordering to encourage some variation in the outcomes.

```
python lab2_play_gui.py roomba roomba_10x12 progressive
progressive
```

   If you want, you may OPTIONALLY implement the heuristic evaluation function **advanced_heuristic_eval_roomba** with a heuristic of your own design.

Rough Rubric: 35 points total

- (15 points) Basic **ProgressiveDeepening** implemented correctly – returns results from the deepest complete minimax w/ alpha-beta pruning search at any time when terminated.
- (10 points) When **transposition_table** is set to true, **ProgressiveDeepening** uses values from previous searches to help order moves during deeper searches. This should reduce the number of nodes searched at deeper cutoffs.
- (5 points) **aggressive_eval_roomba** is implemented, producing generally aggressive behavior in Roomba agents.
- (5 points) **defensive_eval_roomba** is implemented, producing generally defensive behavior in Roomba agents.