IMPORTANT NOTE:

You will need the latest version of Python 3 (3.10, at this time) which can be installed at www.python.org/downloads/. Check your python version by opening a cmd/terminal and running

```
python --version
```

The code and supporting files:

Download and extract **lab1_part1_starter.zip** from Schoology, or clone the starter code from Github:

There are a lot of files, most of which you don't need to read or edit.

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

- **search_algorithms.py** (Part 1a, Basic Search)
- **slidepuzzle_problem.py** (Part 1b, Problem Representation)

Files you'll want to read and understand:

- **search_problem.py** (deeply)
- **roomba_problem.py** (generally)

Files you'll *run to test* your code:

- **roomba_gui.py**
- **slidepuzzle_gui.py**

Other files you'll use:

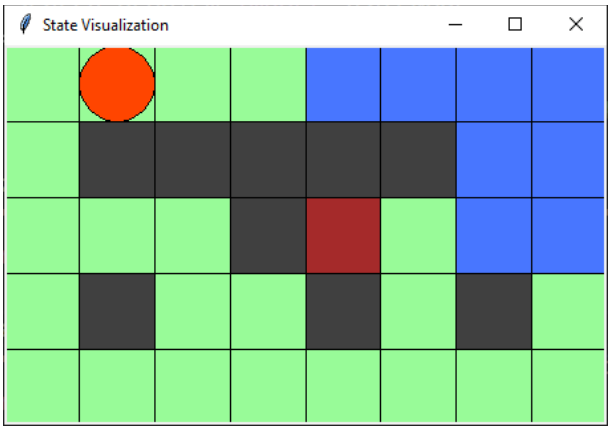
- The folder **roomba_files** that has **.roomba** text files
  - Open one the text files when running **slidepuzzle_gui.py**
  - The maze_[##] files have a single dirt spot (one goal state).
  - The maze_multi_[##] files have more than one dirt spot (multiple goal states).
  - Feel free to design your own!
- The folder **slidepuzzle_files** that has **.slidepuzzle** text files
  - Open one the text files when running **slidepuzzle_gui.py**
  - The [##] at the end of the file is the minimum number of moves to solve the board.
  - There are also some unsolvable test puzzles. (What happens when you search these…?)
  - Feel free to design your own!

When you submit your lab via Schoology, only submit **search_algorithms.py** and **slidepuzzle_problem.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 1a: Basic Search

1. Run the **roomba_gui.py** file, then open a file from the folder **roomba_files.** The Roomba (red circle) has the simple goal of finding any dirty tile (brown square). Click around on the visualized state to input actions. Hit "Reset to Init State" whenever you want to return to the beginning.

   A good example that demonstrates a lot of the differences between search strategies is **maze_01.roomba**, seen below:



2. Check out the StateNode class in **search_problem.py,** a template for our goal-based state search problem representations.

   StateNode objects maintain a representation of a state of the agent's environment. But by tracking things like the parent node, the last action, depth, and path costs, the StateNode ALSO represents a path as well as a state.

   Many of StateNode's methods are abstract and must be implemented by subclasses specific to certain problems. RoombaStateNode is a subclass of StateNode, and you'll implement another subclass in **Part 1b**. Don't worry about the details of either subclass now - since all our algorithms will work with abstract StateNode objects, it makes them usable across any problems that also subclass StateNode.

3. Open **search_algorithms.py**. Take a look at GoalSearchAgent. It has 3 abstract methods plus `__init__()`, which are to be implemented across subclasses.

   Instead of writing full subclasses of GoalSearchAgent, we will write partially implemented subclasses, then use multiple inheritance to "mix-and-match" search *algorithms* (tree and graph search) and search *strategies* (DFS, BFS, UCS).

   Note the RandomSearch subclass, which represents a random search *strategy* and implements all methods except for search. A random search **agent** will inherit from both RandomSearch and another subclass that implements the search *algorithm.*

4. You will first implement the partial class TreeSearchAlgorithm's `search()` method. In it, utilize the `self.frontier` variable and `self.enqueue()` and `self.dequeue()` methods as if they are implemented (as if from RandomSearch). See the comments for implementation details.

   NOTE that the algorithm has one major addition to the pseudocode described in lectures; you must call `gui_callback_fn()` on every **extended** state (not necessarily every dequeued state, as some dequeued states may not be extended).This allows the GUI to see the progress of your search, and to send a signal back for early termination.

   Test your search by running roomba_gui.py. Pick the random strategy, and the tree algorithm. (Don't worry about heuristics for now). Run or Step through the search process and watch the visualization and the text progress updates.

5. Now that you've written a search algorithm, its time to implement some better strategies than random! Implement the `__init__()`, `enqueue()`, and `dequeue()` methods of these subclasses:
   - DepthFirstSearch
   - BreadthFirstSearch
   - UniformCostSearch

   Be sure to read the comments about relevant data structures in the comments!

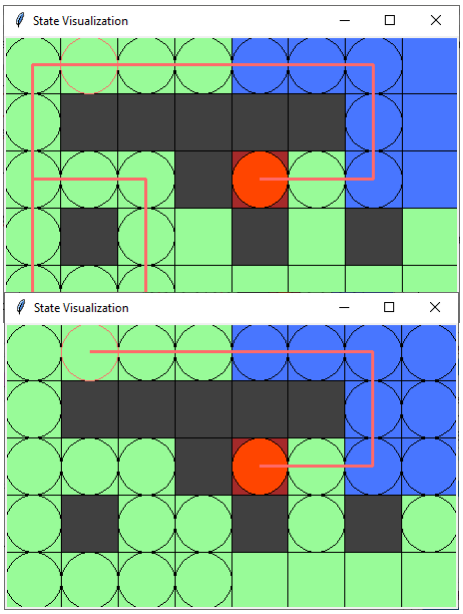
   Test each in the roomba gui (pick dfs, bfs, or ucs as the strategy). Note the difference in behavior. Try different maze files and compare! Some expected results on **maze_01.roomba** are shown below:

   - Tree DFS may differ depending on implementation, but here's one possible solution:

     ```
     Node Depth: 23
     Path Cost: 28.000
     Total Extends: 23
     Total Enqueues: 32
     ```

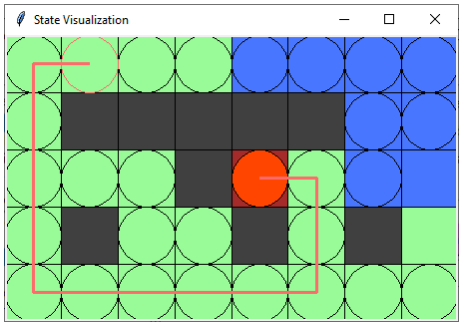
   - Tree BFS should find this solution. Exact extend/enqueue count may vary slightly, but should be very close:

     ```
     Node Depth: 9
     Path Cost: 14.000
     Total Extends: 44
     Total Enqueues: 69
     ```



- Tree UCS should find this solution, or go above/right the lower-left wall. Like BFS, extend/enqueue counts may vary slightly but should be pretty close.

```
Node Depth: 13
Path Cost: 13.000
Total Extends: 87
Total Enqueues: 122
```



NOTE: It is likely you overlooked or misunderstood the cutoff feature of these algorithms. Make sure that you pass the cutoff parameter to your `enqueue()` method. The `enqueue()` method should check that the state's depth or path cost (depending on the algorithm) does not exceed the cutoff before adding it to the frontier. Note that the cutoff does NOT terminate the search upon enqueueing/extending a state that exceeds the cutoff – it merely will not extend the search tree beyond that particular leaf node, but will continue to explore the remaining frontier.

Confirm that this cutoff feature is working correctly by trying your algorithms with a cutoff lower than the minimum depth/path cost for the solution; it should fail to find that solution. Also, try using a cutoff for DFS at *exactly* the minimum depth – suddenly, your DFS should be finding the same solution as BFS.

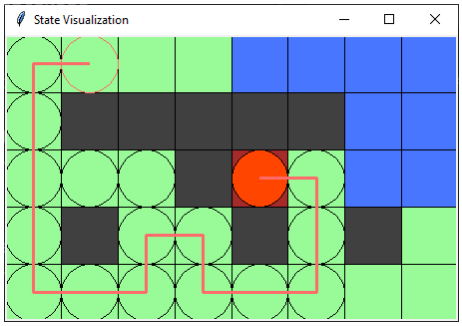
6. Finally, implement GraphSearchAlgorithm's run method. You can start by copying your TreeSearchAlgorithm code over, then making the necessary additions/edits (it will be very similar).

   Test the graph algorithm in the roomba gui with all 4 strategies so far. Compare its performance (time, number of extends/enqueues) to tree search.

   The performance of each strategy using graph search should be improved; expected results for **maze_01.roomba** are shown below:

- Like Tree, Graph DFS may differ depending on implementation, but search paths should no longer double back on themselves. Here's one possible solution:

```
Node Depth: 15
Path Cost: 15.000
Total Extends: 17
Total Enqueues: 24
```



- Graph BFS should find the same solution as Tree BFS, but in fewer extends/enqueues (again, results may vary by implementation):

```
Total Extends: 26
Total Enqueues: 37
```

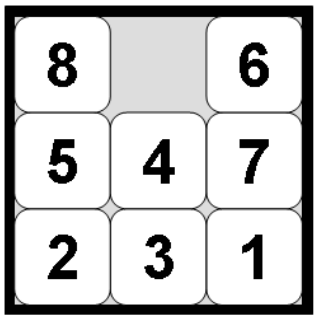
- Graph UCS should find the same solution as Tree UCS, but in fewer extends/enqueues (again, results may vary by implementation):

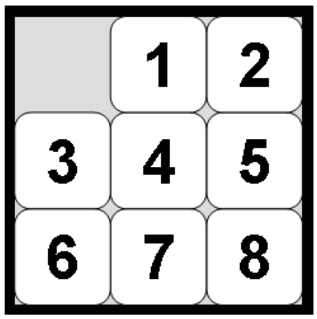
```
Total Extends: 28
Total Enqueues: 39
```

Part 1b: Problem Representation of 8-puzzle

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square tiles labeled 1 through 8 and a blank square. Your goal is to rearrange the tiles so that they are in order, with the empty space in the upper-left corner, ideally using as few moves as possible. You are permitted to slide tiles horizontally or vertically into the blank square.

Below is an example of an initial board (the worst-case, actually) and the goal board:





You can play with an 8-puzzle here to get the feel for how it works, although in this version (and most other others) the goal has the empty space in the bottom-right.

This game and its n-by-n variants are also a famous AI search problem. In this part of the lab, you will represent the general sliding puzzle problem as a StateNode subclass, then apply the various search algorithms you wrote to solve these puzzles.

1) Check out RoombaState in **roomba_problem.py**. You don't need to study it too hard, but you should recognize that it is also subclass of StateNode from **search_problem.py**. It implements all the unimplemented methods of StateNode, plus a few handy accessor methods that the GUI also uses.

   You can reference this file as a model for how you'll write SlidePuzzleState.

2) Finish implementing SlidePuzzleState in **slidepuzzle_state.py** by finishing every method with a **TODO** comment. You may, of course, add any additional methods that you find useful.You may wish to study similar methods in RoombaState to see how you might approach some of them.

   NOTE 1: While both use similar Coordinate objects, the way that RoombaAction and SlidePuzzleAction (which are subclasses of Coordinate) are interpreted are *different* – RoombaAction represents a *relative* amount of roomba movement (# of rows down, # cols right), while SlidePuzzleAction represents the *absolute* coordinate of whatever tile should be moved into the empty position.

NOTE 2: The instance variables and the __init__() method are already defined, but these could be changed if you want. If you do so, however, there are 3 accessor methods used by the GUI that must also be re-implemented: get_tile_at(), get_size(), get_empty_pos(). You would also need to update get_state_features().

NOTE 3: Try to avoid using more computation and memory than necessary in the methods – although it is not essential to optimize, efficient implementation is required for the algorithms to search quickly.

3) Test your implementation by running **slidepuzzle_gui.py** and opening a file in the **slidepuzzle_files** folder**.** You should be able to click on tiles to input actions, and all the search algorithms should work. As you test, you might reduce the step time, and also disable the checkboxes for visualization and/or various textual outputs for a major increase in speed.

   Expected performance:

   - Tree BFS and UCS should perform roughly equivalently, since each action has uniform cost. **They should be able to solve puzzles with solutions around 10 to 12 moves, but will start to be rather slow with more difficult puzzles**. Graph BFS/UCS will help reduce the number of extended nodes and speed up the process, but you're still not going to like waiting for test_puzzle20 to finish…
   - Tree DFS may not be able to solve even some of the smallest puzzles in a reasonable time, since the state space is very large – if it doesn't pick the right path in its initial choices, it's unlikely to find it for a long time! However, if you set the cutoff at optimal solution depth or just beyond it, tree DFS should find solutions in about the same time as tree BFS and UCS.
     - An interesting note: Graph DFS using a cutoff can cause some odd results where reachable solutions are missed. Can you explain why?

NOTE: For all StateNode subclasses, get_all_features() should return only IMMUTABLE types. This means no lists, only tuples. The **tiles** instance variable of SlidePuzzleState is a 2-D *tuple*, not list, for this reason. However, it may be difficult to work with, compared to the mutable list type.

You can easily create converted copies between lists and tuples.

```
...
1d_tuple = tuple(1d_list)
...
1d_list = list(1d_tuple)
```

Converting 2d grids of lists and tuples to one another can also be done pretty easily using the python list-comprehension features:

```
...
2d_list_grid = [list(row) for row in 2d_tuple_grid]
...
2d_tuple_grid = tuple(tuple(row) for row in 2d_list_grid)
```

Rough Checklist Rubric:

search_algorithms.py (Part 1a, Basic Search)

- ✓ TreeSearchAlgorithm is implemented correctly and works with random strategy (20%)
- ✓ DepthFirstSearch is implemented and works with tree search. *Includes depth cutoff* (10%)
- ✓ BreadthFirstSearch is implemented and works with tree search. *Includes depth cutoff* (10%)
- ✓ UniformCostSearch is implemented and works with tree search. *Includes path cost cutoff* (10%)
- ✓ GraphSearchAlgorithm is implemented correctly and works with all 4 strategies (20%)

slidepuzzle_problem.py (Part 1b, Problem Representation)

- ✓ SlidePuzzleState is fully and correctly implemented. The GUI correctly handles interactive button clicks, and all the above search strategies and algorithms work and solve short (~12 step) puzzles in a reasonable amount of time. (30%)

This assignment is worth 10 weighted points.