The code and supporting files:

Download and extract **lab1_part2_starter.zip** from Schoology; add the files into the same folder as the files from **lab1_part1_starter.zip**

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

- **search_algorithms.py** (Part 2a, Informed Search; Part 2d, Anytime Search)
- **slidepuzzle_heuristics.py** (Part 2b, Slide Puzzle Heuristics)
- **spotlessroomba_heuristics.py** (Part 2c, Spotless Roomba Heuristics)
- NEW FILE: **maze_multi_BRGNID[_##].roomba** (Part 2c, Spotless Roomba Heuristics)

Files you'll want to read and understand:

- **search_heuristics.py**
- **roomba_heuristics.py**
- **spotlessroomba_problem.py**

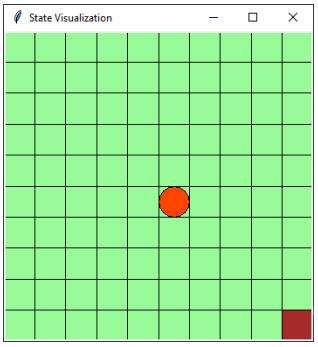
Files you'll *run to test* your code:

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

Other files you'll use (same as part 1):

- 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, and are suitable for the Spotless Roomba problem.
  - 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!

Part 2a: Informed Search



1. Run **python roomba_gui.py** and in the file dialog open **maze_06.roomba**.
   Alternatively, you can skip the file dialog by including the file path as an argument, like:
   `python roomba_gui.py Roomba_files/maze_06.roomba.`

   Run graph DFS, BFS, and UCS (though tree search is also amusing). This example reveals the shortcomings of these three strategies: they don't take into account the *future potential* of the states they choose to explore; that is, they don't utilize any sort of knowledge about how close a state is to reaching the goal.

   If we don't have any kind of state knowledge about goal proximity, then these uninformed search strategies are about as good as we get. But if we have a way of estimating goal proximity – aka, a *heuristic* - we should find a way to utilize it and improve our searching!

2. Open **roomba_heuristics.py** and note the 2 provided heuristic functions, `roomba_manhattan_onegoal()`and `roomba_manhattan_multigoal()`. Both heuristics take a RoombaState as input and return an *estimate* of the remaining cost to the goal from that state. The estimate is simply the Manhattan distance (sum of vertical and horizontal distance) from the roomba's position to the dirty tile. In the case of multiple dirty tiles, the latter function is more appropriate, as it will return the distance to the *closest* tile.

3. Open **search_heuristics.py** and note the 2 universal heuristic functions: `zero_heuristic(state)`, which consistently returns an estimate of 0 remaining cost to the goal, and `arbitrary_heuristic(state)`, which returns a pseudorandom value up to 100 as the estimated cost to goal, though always the same value for the same state. Obviously both are useless, or worse than useless – but they serve as interesting theoretical examples to compare useful ones against.

4. Open **search_algorithms.py**. Now that you know the 4 heuristics at our disposal for the roomba problem, its time to write search strategies that will utilize these heuristics – Greedy Best First search, and A* search.

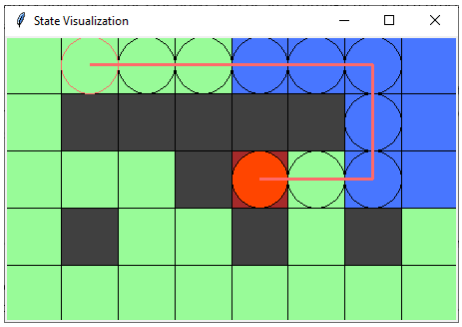
   These strategies will be subclasses of InformedSearchAgent. Note that InformedSearchAgent's `__init__()` has a parameter for a heuristic function, which is stored as `self.heuristic`. That heuristic can be called from `self.heuristic(state)` in any subclass.

5. Implement the `__init__()`, `enqueue()`, and `dequeue()` methods of these subclasses:
   - GreedyBestSearch
   - AStarSearch.

   Don't forget that both should also implement a path cost cutoff in enqueue.

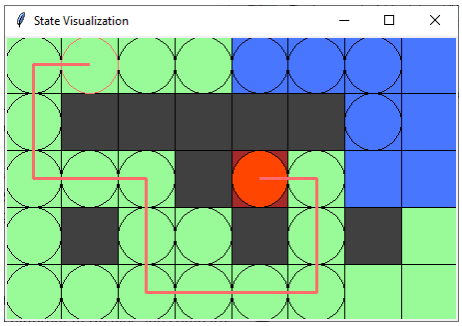
Test both search strategies on various Roomba mazes using the 4 different heuristics. Here are some expected patterns/results:

- Using either of the Manhattan heuristics on a single-goal maze such as **maze_01.roomba**. greedy search (either graph or tree) should yield a solution in very few extends/enqueues, but possibly not the optimal solution. Here's a likely result:




```
Node Depth: 9
Path Cost: 14.000
Total Extends: 9
Total Enqueues: 14
```

- Manhattan Dist. Heuristics with A* will find an optimal solution path like UCS – but generally in far fewer extends than UCS.



Using tree search:

```
Total Extends: 36
Total Enqueues: 50
```

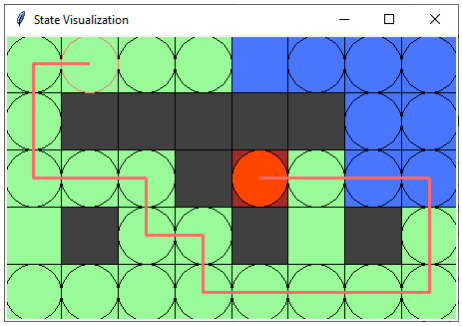
Using graph search:

```
Total Extends: 23
Total Enqueues: 32
```

- For maps with multiple goals (such as `maze_multi_01.roomba`), the choice of Manhattan distance will matter. Try both heuristics with both greedy and A* and see what happens!

- Greedy using the zero heuristic could be as good as random (why?), though in our implementation tends to behave like DFS.
  A* using the zero heuristic should behave like UCS. (why?)

- The arbitrary cost heuristic can have really wild results!
  - It likely makes tree greedy explore over the same few states repeatedly, and never reach the goal.
  - Graph greedy will eventually find *some* solution, but likely not a good one, like this:



  - A* in tree or graph form will also eventually find *some* solution too, but also not necessarily optimal. Also notice that the number of enqueues/extends is worse than UCS too – especially for the tree version, which seems almost as lost as a random search....

Part 2b: Slide Puzzle Heuristics

To apply our informed search algorithms to the 8-puzzle problem, we need a heuristic function that estimates how close a board is to the goal state.

You will implement two heuristic functions: `slidepuzzle_hamming()` and `slidepuzzle_manhattan()`, which take a SlidePuzzleState as an input and return a float.

- The Hamming heuristic function returns the number of blocks in the wrong position, not including the empty space. Intuitively, a search node with a small number of blocks in the wrong position is (probably) close to the goal.
- The Manhattan heuristic function returns the sum of the Manhattan distances ("city block distance", or sum of the vertical and horizontal distances) of each block to their goal positions (again, not including the empty space). The intuition is similar to the Hamming heuristic; the closer each block is to where it should finish, the closer to the goal.

For example, the Hamming and Manhattan heuristic functions of the initial 3x3 board below give 5 and 10, respectively.

```
 4  3  2          1  2     1  2  3  4  5  6  7  8    1  2  3  4  5  6  7  8
 7     5       3  4  5     ----------------------    ----------------------
 6  8  1       6  7  8     1  0  1  1  0  0  1  1    3  0  2  2  0  0  2  1

initial          goal         Hamming = 5              Manhattan = 10
```

1. Open **slidepuzzle_heuristics.py** and implement both `slidepuzzle_hamming()` and `slidepuzzle_manhattan()` as described above.

2. Run **slidepuzzle_gui.py** to test both of your heuristics on various slide puzzles. Either input actions with clicks, and/or pause at various points in the search, and read the printout "**Est. Rem. Cost to Goal:**" to verify that your heuristics are accurate for the displayed state.

   Once you're confident the heuristics are working correctly, try actually running both informed search algorithms on different puzzle files. The optimal solution length is the ## in the file name. Generally, higher numbers are harder, so **start low**.
   You can open a file quickly by adding the file path as an argument, like:

   `python slidepuzzle_gui.py slidepuzzle_files/test_puzzle[##].slidepuzzle`

   Expected performance:

   - Greedy will generally find solutions quickly but the paths might be (comically) long.
   - A* should find optimal solutions, and generally in far fewer extends than UCS/BFS.
   - Both informed algorithms are capable of finding solutions to nearly all the puzzles in reasonable amounts of time if you turn off the visualizer and progress outputs for speed, though some in the 40-50 range may take a minute or more….
   - Manhattan is usually best, but there are some rare puzzles where Hamming is better.

Part 2c: Spotless Roomba Heuristics

It's time to update our Roomba Route problem to a more interesting situation: now, when there are multiple dirty tiles, ALL of them must be cleaned up.

1. Open **spotlessroomba_problem.py** and skim the code for SpotlessRoombaState. Note that the state representation has changed somewhat from RoombaState in **roomba_problem.py**.

   In addition to the roomba's position, the state features also include which locations are still dirty. The coordinates of those spaces are kept in the tuple `self.dirty_locations`. Now, the goal test is whether that tuple is empty, and the transition function (`get_next_state()`) updates that tuple if the roomba moves onto one of those spaces.

2. Try running **python spotlessroomba_gui.py** using some of the smaller **maze_multi_##.roomba** files (such as **maze_multi_02.roomba** or **maze_multi_06. roomba**)! Your DFS, BFS, and UCS algorithms should be able to solve these in a minute if you reduce the step time (or disable visualization); you'll see numbers appear on dirty tiles in the order they are visited.

3. Open **spotlessroomba_heuristics.py**.

   Write 2 (or more) heuristics for RoombaMultiRouteState that should help greedy and A* converge more quickly on solutions. Note the two empty functions that are referenced in the SPOTLESSROOMBA_HEURISTICS dictionary, which you can rename as you like.

   You may choose start by copying relevant code from the heuristics in **roomba_heuristics.py.** *Note that the utilizing the RoombaMultiRouteState's property `dirty_locations` is more convenient and efficient for getting all dirty locations than manually searching all coordinates and checking get_terrain().*

   Some specifications:

   - Add comments at the top of each function explaining the idea behind each heuristic. If admissible/consistent, argue why.
   - Both heuristics should be nontrivial (more than merely returning zero), but also relatively quick and efficient to calculate: quick means stopping short of calculating the exact remaining distance. After all, doing as much work as another full search defeats the purpose of using a heuristic!
   - Both heuristics should generally help the informed algorithms look at fewer states than UCS, especially in bigger mazes. (The effect is more pronounced for tree than graph)
   - At least one heuristic must be ***admissible*** to guarantee an optimal solution from tree A* (though not necessarily an optimal solution for graph A*).
     - Admissibility means the heuristic never overestimates the distance to the goal. When running A* with an admissible heuristic, the sum of the path cost and heuristic value of extended states should *steadily* increase throughout search. You

might printing these out or write some assertions in code to confirm this behavior. If the sum is bobbing up and down, your heuristic is not admissible!

- It is not required, but a heuristic must be ***consistent*** to guarantee optimality from graph A*. Some extra credit is available if you create a consistent heuristic and convincingly argue for it.
  - Consistency is an extension on admissibility: the heuristic must never overestimate the distance to the goal, AND for any transition from *s* to *s'* that costs *c*, heuristic *h* must satisfy the equation *h(s) - h(s') <= c* (or, *h(s) <= h(s') + c*)
  - The intuition for consistency is difficult to grasp. One way to think about it is that along any transition, the heuristic cannot never MORE than the actual cost.
  - A consistent heuristic can be surprisingly hard to find for this problem!

You may certainly create more than two heuristics. If you do, add them to the SPOTLESSROOMBA_HEURISTICS so they can be seen when running **spotlessroomba_gui.py**.

Test your heuristics extensively! Either input actions with clicks, and/or pause at various points in the search. Read the printout "**Est. Rem. Cost to Goal:**" to verify that your heuristics are accurate for the displayed state.

4. Lastly, create one or more of your own multiple-dirt mazes to illustrate your heuristics' effectiveness. It should be challenging enough so that your heuristics make significant improvement over uninformed algorithms.

Your mazes should be in files named `maze_multi_BRGNID[_##].roomba`

BRGNID should be your bergen ID (usually 3 letters of first name, 3 letters of last name)
If you have multiple mazes, ## should be 01, then 02, etc.

The format of the files are as follows:

First row: 2 ints specifying the number of rows and cols in the maze
Second row: 2 ints specifying the initial row and col of the roomba (starting at zero)
Remaining rows: Terrain of the maze, drawn a row at a time. Be sure to not have extra spaces at the ends of lines.

'.' : Flooring (cost 1 to move onto)
'~' : Carpet (cost 2 to move onto)
'#' : Wall (can't move onto)
'?' : Dirty flooring (costs 1 to move onto and clean)
'+' : Dirty carpet (cost 2 to move onto and clean)

Test your maze using your heuristics. The maze should demonstrate how your heuristics are genuinely useful for Greedy/A*.

After submitting, feel free to share your maze with your classmates!

Part 2d: Anytime Search

Sometimes, a problem is too difficult to plan a full solution in a reasonable amount of time. A search algorithm may need to be terminated before finding the goal – maybe the frontier runs out due to cutoffs, or the user ends the search manually. Ideally, we would like to have some kind of partial solution, even in these early termination cases.

In the event of ending not yet finding a solution, our search algorithms so far simply return None for failure. Without any information about proximity to the goal, this makes sense – all incomplete paths are equally likely to be useful. But if we have a heuristic function, we can *always* select a path from the search tree, however incomplete, that seems closest to finding the goal at any time.

We call the class of algorithms that can always return a useful result, even if terminated early, "anytime algorithms." The final search algorithm you will implement is an "anytime" version of graph search.

1. Open **search_algorithms.py** again. Copy your code from GraphSearchAlgorithm into AnytimeSearchAlgorithm.

2. Update the code so that the algorithm never returns None, but instead always returns the **lowest-cost path to the lowest-heuristic state** found so far. *Do not merely return the most recently enqueued/extended node.*

   This should not take very much additional code.

3. Test out your anytime search algorithm with various search strategies on each of the different problems. *Make sure to pick a useful heuristic (Zero will not be useful).*

   Specify low cutoffs and/or manually terminate the search before a solution is found. The gui should display your returned partial solution after termination. You can continue running the search from the partial solution point.

Rough Checklist Rubric:

search_algorithms.py (Part 2a, Informed Search; Part2d, Anytime Search)

- ✓ GreedyBestSearch is implemented, works with both tree and graph search, and finds paths quickly with the roomba manhattan heuristics. *Includes path cost cutoff* (10%)
- ✓ AStarSearch is implemented, works with both tree and graph search, and finds optimal paths quickly with the roomba manhattan heuristics. *Includes path cost cutoff* (10%)
- ✓ AnytimeSearchAlgorithm is implemented correctly and works with all 6 strategies (20%)

slidepuzzle_heuristics.py (Part 2b, Slide Puzzle Heuristics)

- ✓ slidepuzzle_hamming() is implemented correctly. (10%)
- ✓ slidepuzzle_manhattan() is implemented correctly. (10%)

spotlessroomba_heuristics.py (Part 2c, Spotless Roomba Heuristics)

- ✓ At least two useful heuristics have been implemented (20%)
- ✓ At least one of the heuristics is admissible; admissibility is argued for sufficiently (10%)
- ✓ At least one of the heuristics is consistent; consistency is argued for sufficiently (+5%)
- ✓ At least one example maze has been designed (10%)