420-LCW-05 Programming Techniques and Applications - Assignment 2

February 12, 2019

Here's a reminder of the general requirements for all of the course assignments:

1. **Identification section** Do this for *every* Python file in every assignment in this course. This section must be either in a comment, with a '#' preceding each line, or enclosed within triple quotes ("'). The grader and I need this section for the *accurate processing of your assignment*. Assignments missing this may lose up to 5% of the total mark.

   Example:

   ```
   """
   Justin Trudeau, 1234567
   Sunday, February 31
   R. Vincent, instructor
   Assignment 2
   """
   ```

   Obviously substitute your name, Marianopolis ID, and the correct date for the appropriate fields!

2. Always include additional comments with your code. These need not explain every individual line of your program, but consider using comments for the following situations:

   - A brief explanation of a particular variable's purpose, included on the first line where the variable is defined, e.g.:

     ```
     hi = 100 # Define the upper limit of the range.
     ```

   - A note mentioning any website or person you may have consulted with to help with the assignment.
   - A comment describing any constant value that appears in your code.

   In addition, each `def` statement, whether for a global function or class method should include at least a brief docstring. *You should also provide docstrings for any new classes you create*.

3. Your submission for assignment will typically include multiple Python files, which have the extension `.py`. Before submission, these files must be combined into a single ZIP archive file (extension `.zip`). If you do not know how to create a ZIP file, I will demonstrate this in lab.

4. Be sure to respect other instructions specified in the assignment. Part of each assignment is to correctly follow the instructions as closely as possible.

Introduction

An "8-puzzle" or "15-puzzle" (see `https://en.wikipedia.org/wiki/15_puzzle`) is a simple puzzle that you solve by sliding interlocking tiles around inside a 3x3 or 4x4 frame. Of course, for the puzzle to work, at one tile has to be missing.

The tiles are often numbered, in which case the goal is to get all of the numbers in increasing order from the top left to tbe bottom right (other variations may have a picture on the tiles that you're trying to reassemble, but the idea is still the same).

In this assignment, you will finish a program that solves these puzzles.

To solve a puzzle like this, our approach is pretty simple. You'll be given an initial puzzle state, and you will need to try all possible future puzzle states until you find one that is in the correct, solved state.

One method to solve many puzzles like this is an algorithm called A* (pronounced "A star"). A* is an algorithm for searching for the best path through a maze, puzzle, map, or what computer scientists call a "graph" (We'll discuss graphs in much more detail soon). To guide the search process, you need to give the algorithm an idea of whether or not it is getting closer to the final or goal state. This means we need some measure of the "distance" from our current state to the goal state. The algorithm is always trying to find moves that *reduce* the distance to the goal.

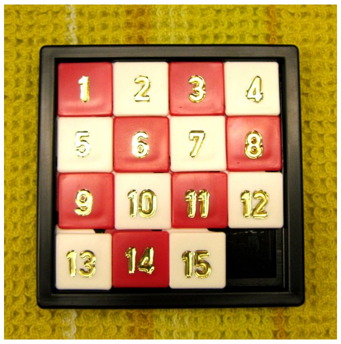
Figure 1: A solved 15-puzzle (from Wikipedia, photo: Micha L. Rieser)

While there are different ways of implementing A*, it turns out that a priority queue is a great tool for performing this search.

To get you started, I've provided a few basic pieces:

- `Board.py` implements the class that represents a puzzle configuration. It supports 2x2, 3x3, 4x4, or other square puzzles. I've already implemented the methods you will need, except for the `__repr__` method that you will provide.
  - `board = Board(string)` - construct a new board object, normally using the contents of a text file.
  - `board.neighbors()` - return a list of board objects that can be reached by the current board in one move.
  - `board.solved()` - returns `True` if the board is in a solved state.
  - `board.distance()` - returns the measure of how "good" the current state of the puzzle is.
- `Solver.py` is the file in which you will implement most of your work. I've already provided a key piece of the work, the class `Node`. The `Node` class represents each "puzzle state" that we are examining. This class defines the values that we put on our priority queue. It consists of the following attributes:
  - `board` - the associated `Board` object.
  - `moves` - the number of moves required to get to this `Board` configuration.
  - `cost` - the distance metric for this `Board` (e.g. the result of `board.distance()`.
  - `previous` - the `Node` immediately prior to this one in the search process. This is the `Node` that appears "one move back" in the puzzle solution.
- `MinPQ.py` is our implementation of a min-heap priority queue. This is our key to implementing A* properly.
- The files in the `puzzles` directory are sample puzzles. The filename gives the size and the minimum number of moves to complete the puzzle.
- The files `puzzle4x4-20-output.txt` and `puzzle4x4-12-output.txt` show sample output that your program should mimic *exactly*.

Exercise 1 - Solving a puzzle

1. Your first task is to implement the basic puzzle solving algorithm. You will do this by editing the provided file `Solver.py` and adding the algorithm to solve the puzzle to the `__init__()` method.

   Here is the basic idea of solving the puzzle: You are given a parameter `initial` that is the starting state of the puzzle. Since it is the only state you know about, the first thing you will do is create a `MinPQ`, the create a `Node` based on the initial `Board`, and insert the initial `Node` on the queue.

   Now you need to start searching for the solution. This is the essence of the A* algorithm: As long as the priority queue is not empty, you must remove the "best" `Node` from the priority queue. If this `Node` represents a solved puzzle, you've found the solution. Otherwise, you want to list all of the "neighbors" of the Node's board, and create new Nodes to represent each of these possible board states, then insert these newly-created Nodes onto the priority queue. **Each new Node needs to have the correct number of moves, and the correct previous Node specified when it is constructed.**

   Your algorithm should implement one important optimization: remember that we can generally return to a prior Board position in two moves. However, you don't want to put a Board back on the priority queue if you've already tried it. You should therefore "filter" the list of neighbors and insert only those that do not match the previous Board.

2. When you do find the solution, you need to convert it into a list of Board positions. To do this, you'll need to follow the `previous` links in the Nodes back to the original Board configuration. You'll want to set the attribute `__solvable` to `True` and `__trace` to the list of nodes in the solution.

3. Finally, you need to write a main program to exercise your `Solver` class. Your main program should be added at the end of the file `Solver.py`. It will prompt for a file name, open and read the file, and create an initial board position based on the contents of the file. Then your program should create a `Solver` object, and then print the solution found. See the examples given for the details of what you should print out.

   NOTE: as part of the exercise, you need to add a `__repr__` method to the `Board` class. *Your method should produce a format that exactly mimics the contents of the provided sample files.*

2		8
1	3	5
4	6	7

Figure 2: An example 3x3 board. For this configuration, the Hamming distance is 8, because every tile is in the incorrect location. The Manhattan distance is 13, because the 1 tile needs to move one position, 2 needs to move one, 3 -> 2, 4 -> 1, 5 -> 1, 6 -> 2, 7 -> 2, and 8 -> 3.

Exercise 2- Distance metrics and performance

There are two different distance calculations implemented in the `Board` class. One is based on the Manhattan or 'taxicab' distance, which is just the sum of the differences in both x and y coordinates. To compute this metric, we first compute the Manhattan distance for each tile from its current position to its correct position. Of course, this ignores the fact that the tiles can't move past one another. The overall distance metric is the sum of the Manhattan distances for every tile See Figure 2 for an example.

The second metric is much simpler, the Hamming distance (named for Richard Hamming). It simply computes the number of tiles that are out of place (Figure 2).

The default version of the `Board` class uses the Hamming distance, but I've provided the code to compute both. You can control which is used by changing the Boolean class variable `use_hamming`.

For this exercise, your task is as follows:

1. Modify your `Solver` class to keep track of the total number of board positions checked when computing a solution. This is essentially the number of times your main loop will execute.
2. Run tests with the current version of the `Board` class, using the files `puzzle4x4-01.txt`, `puzzle4x4-10.txt`, `puzzle4x4-16.txt`, `puzzle4x4-20.txt`, and `puzzle4x4-25.txt`.
3. Note the relationship between the number of moves required and the maximum number of items on the priority queue.
4. Modify the `Board` class to use the Manhattan distance rather than the Hamming distance. You can do this by changing the value of `use_hamming`.
5. Re-run the same five tests, and note the relationship between number of required moves and maximum priority queue size.
6. In a text file name `ex2.txt`, record your observations. Offer a brief explanation for the differences you observe.

Exercise 3- Detecting unsolvable puzzles (OPTIONAL)

It turns out that one interesting property of these puzzles is that it is impossible to solve some of them. What is more interesting is that if a puzzle cannot be solved, then its "twin" *can be solved*. The "twin" of a puzzle is the original configuration with *any two* tiles exchanged.

Extend your solver class to detect unsolvable puzzles by implementing the `twin()` method in the `Board` class. This extended version should return a new board object with exactly two tiles exchanged. Then create a new file, `Solver2.py` that runs *two* simultaneous versions of the A* algorithm, and see which of the two initial states leads to a solution. If the twin reaches the solved state, that means the original configuration cannot be solved.

If no solution exists for the original configuration, your program should now just print "No solution possible" and exit.

Submitting your work

When you have finished both sections, combine all of your modified files into a single ZIP file and upload that to Omnivox.