CSCI 360 – Project #1

Programming Part: 8-Puzzle - 6 Points

In this part of the project, you will implement Weighted A* to solve the 8-puzzle problem. We provide you with a skeleton implementation to help you with certain parts of the project (we refer to the provided code as necessary throughout the text). For further details, you should go over the code and read the comments. We have also included a small demo about how to use the provided code ('Demo' function in 'main.cpp'), which you can run by executing the program with no arguments: './8PuzzleSolver'.

Problem Description

In the 8-puzzle problem (similar to the 15-puzzle problem that you have seen in class), we are given a 3×3 board with 8 sliding tiles that can be moved horizontally or vertically. Given that only 8 of the 9 positions on the board can be occupied by the tiles, one position is always blank. We use the numbers 1-8 to denote the individual tiles, and use the number 0 to denote the blank position. Below is a possible starting configuration (that is, start state) of the puzzle:

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

A move in the 8-puzzle problem always costs 1, and can be performed by sliding a tile that is adjacent to the blank position onto the blank position (that is, swapping the 0 with one of its horizontal or vertical neighbors). The objective of the puzzle is to get to the following goal configuration with the minimum number of moves (for the example above, the minimum number of moves is 28):

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

Weighted A*

Although the 8-puzzle problem has a relatively small state space, many combinatorial search problems (such as the 15-puzzle or the Rubik's cube problem) can have large state spaces and solving them optimally using A* might require a lot of time and memory. One way to address this issue is to use Weighted A*, which is a simple variant of A* that inflates the heuristic. Rather than using the function $f(s) =$

$g(s) + h(s)$ to order the elements of the open list, Weighted A* uses the function $f(s) = g(s) + w^*h(s)$, for some value of w.

Consider the different best-first search algorithms that you have seen in class. Here is a quick list, along with the criteria that each algorithm uses for ordering the elements of the open list:

- Uniform-Cost Search: $f(s) = 1^*g(s) + 0^*h(s)$;
- A*: $f(s) = 1^*g(s) + 1^*h(s)$; and
- Pure heuristic search: $f(s) = 0^*g(s) + 1^*h(s)$ (or, equivalently, $f(s) = 1^*g(s) + C^*h(s)$, where $C \to \infty$).

In this context, Weighted A* can be seen as a general framework that unifies these algorithms (for Uniform-Cost Search, $w = 0$; for A*, $w = 1$; and for pure heuristic search, $w = C$ where $C \to \infty$).

For this part of the project, you will implement Weighted A* (by implementing A*, but using $f(s) = g(s) + w^*h(s)$ to order the elements in the open list). In your implementation, you can implement the g- and h-values of states as integers, but you should implement the f-values and w as floating point numbers.

Managing the State Space and Duplicate Detection

We can uniquely represent each state by concatenating the different rows of the puzzle (linearized representation). For instance, we can use "876543210" to represent the following state of the puzzle:

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

We can, in theory, store any search-related information (such as whether a state is in the closed list, its g-, h-, and f-values, etc.) in an array of length 876,543,210 and access information about a specific state by using its linearized representation. However, this approach is wasteful (for instance, the index 111,111,111 does not correspond to a valid state) and is infeasible for large state spaces, where we only want to allocate space for states that are relevant to the current search.

Instead, in this project, you will assign IDs to states as they are generated. The first state generated (the start state) will have an ID of 0, the next state will have an ID of 1, and so on. You can then maintain a vector of generated states (that grows as the search progresses) and use state IDs to access the relevant information in this vector.

Whenever you encounter a new state (generated as a successor of an expanded state), you should check if it has already been generated before in order to avoid duplicate states. If it has already been generated before, you should use its previous

ID. Otherwise, you should create a new ID for it. You can achieve this by using a hash table.

Our provided code contains the files ‘Puzzle8State.h’, which shows a possible definition of a state for the 8-puzzle problem, and ‘Puzzle8StateManager.h’, which contains a hash table implementation to detect duplicate states and assign IDs to newly generated states.

Heuristic

You will use the Manhattan Distance (as discussed in class) as heuristic. Basically, for each tile (excluding the blank tile), you will calculate the number of rows and columns that it is offset from its goal position and sum up these numbers.

Re-Expansions

You should not expand a state more than once. (Hint: Maintain a closed list, either as a hash-table or by using a ‘closed’ flag for each generated state.) You also shouldn’t add any state into the open list if it is already in the closed list.

Open List

As mentioned earlier, you will use $f(s) = g(s) + w^*h(s)$ to choose the next state to expand from your open list. You also need to implement your open list as a binary heap since maintaining an ordered list as priority queue can be very slow.

You can use ‘std::priority_queue’ as a binary heap, but you should be careful about the following issues:

- When defining a comparator, implementing a ‘greater’ operator (that is, lhs > rhs) results in std::priority_queue acting like a min heap (that is, top() returns the element with the minimum key), which is what we want for the open list. Implementing the comparator as a ‘less than’ operator would cause problems in your implementation.

- std::priority_queue does not allow for updating of heap elements. Therefore, if the g-value of a state decreases, you will have to insert a copy of it into the heap. You should be careful to not expand the same state twice (that is, you want to check before expanding if the state is already in the closed list).

Our provided code contains the file ‘Puzzle8PQ.h’ that shows a possible open list implementation using std::priority_queue.

Implementation

In ‘Puzzle8Solver.cpp’ in the provided code, you can find the following function:

```
void WeightedAStar(string puzzle, double w, int & cost, int & expansions)
```

This is the function that you should implement so that it takes as input a puzzle (represented in linearized form as a string) and a weight w, and performs a Weighted A* search with weight w. It should then update the values of 'cost' to the solution cost (that is, the length of the path) that it has found and 'expansions' to the number of expansions made by Weighted A*.

You are not allowed to change the 'main.cpp', 'CPUTimer.h', and 'CPUTimer.cpp' files. You are not allowed to change the declaration of the 'WeightedAStar' function. You are free to use, modify, or discard any other code that we provide.

You are not allowed to use any external libraries for your code, except for STL. We have tested and verified that the provided code (and a solution to the project that extends the provided code) compiles and runs on Ubuntu 16.04 with a g++ 4:5.3.1 compiler. You should check if our provided code works on your Linux setup and let us know if you have any problems (please do this until Friday this week). You are free to develop your code on any platform that you choose, although we have not tested if our code works on non-Linux operating systems (and we might not be able to help you to get your code work on your preferred platform). We will test your code on a Linux machine that supports C++11.

The provided code contains a makefile and can be compiled with the command 'make'. If you add any files to the project or discard any files, make sure that your code compiles with the 'make' command (by adding new .cpp files to and removing unnecessary .cpp files from the makefile).

Our provided code can be run in different ways, as follows:

- './8PuzzleSolver': Without any command line arguments, the 'Demo' function is called in 'main.cpp', which shows how some of the provided code can be used. You should try this in the beginning of the project and look at the 'Demo' function to become familiar with the provided code.
- './8PuzzleSolver P w': With two arguments, the 'WeightedAStar' function is called with puzzle P and weight w. For instance, './8PuzzleSolver 876543210 1.0' should report solution cost 28 after you finish implementing the 'WeightedAStar' function. We will check whether your implementation works correctly by using two arguments.
- './8PuzzleSolver id': With only one argument (which should be the last 3 digits of your student ID), the 'CreateTable' function is called, which performs the necessary experiments and prints the table that is required for the Theoretical part.

Theoretical Part: Experiments with Weighted A* - 4 points

In this part of the project, you will use your code to perform some experiments, report the results, and, finally, interpret them.

Performing the Experiments

Our provided archive contains the file 'instances', which is a list of 1000 8-puzzle instances numbered from 0 to 999. You will be working with 50 of these instances, based on your student ID, as follows: Find the instance that corresponds to the last three digits of your student ID and use that instance along with the following 49 instances (loop around if necessary: for instance, if your student ID ends with 990, use instances 990-999 and 0-39).

Run each instance with the following values for the weight w: 0, 0.25, 0.5, 0.75, 1, 1.5, 2, 3, 5, and 10. Then, for each weight, report the solution cost (that is, path length), runtime (in milliseconds), and number of expansions, averaged over your 50 instances. Your document should include a table like this:

```
     w       Cost    Time(ms)  Expansions
  0.00      21.34      374.96    83519.66
  0.25      21.34      154.20    34807.14
  0.50      21.34       45.75    11039.78
  0.75      21.34       10.87     2937.34
  1.00      21.34        3.16      914.54
  1.50      21.78        1.53      443.60
  2.00      22.78        1.21      357.20
  3.00      26.22        1.11      328.10
  5.00      30.30        0.96      283.86
 10.00      36.34        1.00      294.82
```

You can generate such a table (after implementing Weighted A* as detailed in the previous section), simply by using the command '8PuzzleSolver Last3DigitsOfYourStudentId'. The table above was generated with our Weighted A* implementation, using the command '8PuzzleSolver 000', although it is possible that your implementation results in a slightly different table if run with the same command (for example, due to differences in tie-breaking among states with the same f-value).

Interpreting the Results

Answer the following questions. If you were unable to do the programming part, say so and use the table provided in the previous section to answer the questions.

1. What trends do you observe in runtime, number of expansions, and the resulting path length as w varies?

2. For which values of w is Weighted A* guaranteed to find a shortest path (and why)? For which values of w is Weighted A* not guaranteed to find a shortest path (and why)?

3. What can you say about the solution cost as w increases? Why do you think that this is the case?

4. What can you say about the number of expansions as w increases? Why do you think that this is the case?

5. What can you say about the runtime as w increases? Why do you think that this is the case?

6. Suppose that we modify the Manhattan distance heuristic so that it also treats the blank position as a tile (that is, the summation also includes the distance of the blank position to its goal position). What are the advantages and disadvantages of using the modified heuristic in conjunction with a regular (unweighted) A* search? Is the modified heuristic admissible?

Submission

You need to submit your solutions through the blackboard system by Monday, Sep. 25, 11:59pm.

For the programming part, you should submit an archive named 'Part1.extension' (extension should be one of the following: .zip, .tar, or .tar.gz). As mentioned earlier, the archive should contain a makefile and the program should compile by invoking the 'make' command.

For the theoretical part, you should submit a PDF file named 'Part2.pdf' that contains both a table of experimental results and answers to the questions.

If you have any questions about the project, you can post them on Piazza (you can find the link on the course wiki) or come to the TAs' office hours.