Project 5: Sliding Tile Puzzle Solver

In this assignment you will implement one of two data structures that enable a state-space search algorithm called A* (pronounced A-star) that is applied to solve a simple sliding-tile puzzle. **The algorithm and puzzle solving code are provided**, you only have to implement and test the data structure.

Sliding Tile Puzzles

Sliding-tile puzzles are classical problems in state-space search. We will be considering an eight-tile puzzle. It consists of an 3x3 board of eight tiles labeled A-H and a single blank (missing) tile. The position of all tiles and the blank space is the *state* of the puzzle. A tile may slide into the blank position vertically or horizontally giving rise to other states. The goal of the puzzle is to find the sequence of tile moves necessary to reach a particular goal state. For example, consider the following puzzle state:

ABC
DEF
GH

There are two possible next states resulting from sliding H right into the blank space or sliding F down into the blank space:

ABC ABC
DEF DE
G H GHF

The state space search proceeds by searching through moves until a goal state is reached. This is similar to the maze problem from the previous project but the number of possible board states is much larger. Thus, we need a more efficient search algorithm that can use additional information, called a *heuristic*. The optimal algorithm (using the same information) is A*.

A* algorithm

Like the breadth-first search from the previous project, the A* algorithm can be described generically using a type **State** and operations on a **problem** instance:

- problem.initial() returns the initial state of the problem
- problem.goal(state) returns true if state is the goal state, else false
- problem.actions(state) returns a list of next states resulting from possible transitions from state

We only need to add two additional variables, traditionally called the **path-cost** and **f-cost**. The **path cost, *g***, is the number of state transitions from the initial state to the current state. The **f-cost** is the path cost plus a state-dependent value called the **heuristic, *h***, that estimates how far from the goal state the current state is; that is ***f = g+h***.

The supporting data structures for A* are a **priority queue** frontier modified to allow for inclusion tests and replacement, and a set explored. The frontier is a **min priority queue (min heap)** with s as the initial element, and explored starts as an **empty set**. Unlike in the previous project it is infeasible to implement the inclusion tests using an array structure because the number of states in general is very large.

Pseudocode for the A* algorithm is given below:

```
s = problem.initial()
if problem.goal(s) return s
while true
    if frontier is empty return failure
    s = pop next state from frontier
    add s to explored
    for each state s_next in problem.actions(s) do
        if s_next not in explored or frontier then
            if problem.goal(s_next) then return s_next
            else insert s_next into the frontier
        else if s_next is in frontier with a higher path-cost
            replace the state in frontier with s_next
```

We can apply this A* algorithm to our puzzle solver. The puzzle state is a specific arrangement of tiles. State transitions (actions) are determined by the location of the blank slot. We will simplify things somewhat by only keeping track of the path cost rather than the actual sequence of moves.

A* Implementation

An implementation of the A* algorithm is provided in the class PuzzleSolver (puzzle_solver.hpp and puzzle_solver.cpp) using the State class (the template in state.hpp and state.tpp). The puzzle board and supporting functionality is provided in the Puzzle class (puzzle.hpp and puzzle.cpp).

The explored_set in the provided code uses unordered_set, the dictionary/hash-table implementation in the C++ standard library. Start by reviewing the documentation of the methods/operations available for the unordered set (linked above), and **understand how they are used in the given PuzzleSolver class.**

Since the frontier is not a normal priority queue, **this is the data structure you will be implementing.** Review the method descriptions provided in frontier_queue.hpp as well as the description below.

There is a set of tests for the puzzle solver in test_solver.cpp. **When your frontier queue is implemented properly, these tests should pass**. Each test takes two strings in the form "012345678", where 0 = A, 1 = B, ... 7 = H, and 8 = BLANK, and converts them to Puzzle instances. It then checks that the solution path cost from one puzzle to another is correct, including the symmetric case (swap initial and goal states).

Frontier Queue

Your task is to implement the frontier queue in frontier_queue.tpp as a **min heap,** using dynamic allocation as necessary. **Your implementation should have the following complexity for each method** (see also the header file comments for details):

- push should add a state to the heap with time complexity **O(log n)**
- pop should remove and return the state in the heap with the **lowest f-cost**, and should have time complexity **O(log n)**. Utilize the State class **getFCost** method.
- contains should return true if the state is in the frontier, and should have time complexity **O(n) or better**
- replaceif should replace a state in the heap if the existing state has a higher path cost, and the resulting queue **must still be a valid min heap**. This should have a time complexity of **O(n) or better**.

Add tests to test_frontier_queue.cpp to test your implementation of each frontier queue method before testing the overall implementation with test_solver.

Note: You may use any combination of containers or algorithms from the **C++11 standard library** to implement the frontier queue. You will need to read and understand most of the code provided to you before starting.

Project Submission

Generate a zip file for submission by running the following commands in the top-level project directory:

- `cmake .`
- `make`
- `make submission`

Answer the questions in the project report template, and submit your code zip file and report pdf via Canvas.