**Sliding Tile Puzzles**

[Sliding-tile puzzles](https://en.wikipedia.org/wiki/Sliding_puzzle) are classical problems in state-space search. We will be considering the eight-tile puzzle. It consists of an 3x3 board of eight tiles labeled A-H and a single blank (missing) tile. This is called 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 tile moves need to reach a particular goal state. For example, consider the following puzzle state

ABC

DEF

GH

There are two possible 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. The optimal algorithm is called A\*.

In this project, you will implement 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.

**A\* algorithm**

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 states resulting from possible transitions from state

We only need add two additional variables, traditionally called the path-cost and f-cost. The path cost, , 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, , that estimates how far from the goal state the current state is; that is .

function astar\_search(problem) returns a solution or failure

s = problem.initial()

if problem.goal(s) return s

frontier is a min priority queue with s as the initial element

explores is an empty set

while true

if frontier is empty return failure

s = pop next state from frontier

add s to explored

for each state s' in problem.actions(s) do

if s' not in explored or frontier then

if problem.goal(s') then return s'

insert s' into the frontier

else if s' is in the frontier with a higher path-cost

replace the state in the frontier with the current s'

The supporting data structures for A\* are a priority queue modified to allow for inclusion tests and replacement, and a set.

We can apply this algorithm to our puzzle solver easily. The puzzle state is simply a given 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 module PuzzleSolver (puzzle\_solver.hpp and puzzle\_solver.cpp) using the State module (the template in state.hpp and state.tpp). The puzzle board and supporting functionality is provided in the Puzzle module (puzzle.hpp and puzzle.cpp).

The algorithm requires implementations of the data structures **frontier queue** and **explored set**. The explored set in the provided code uses [unordered\_set](http://en.cppreference.com/w/cpp/container/unordered_set), the hash-table implementation in the standard library. Since the frontier is not a normal priority queue, this is the data structure you will be implementing. The API of this module is defined in frontier\_queue.hpp and described 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 the solution path-cost from one puzzle to another is the correct one, including the symmetric case (swap intial and goal puzzle).

**Frontier Queue**

The frontier queue template should be implemented in frontier\_queue.hpp and frontier\_queue.tpp as a min heap using dynamic allocation as necessary and have the following complexity for each member (see the header file comments for details):

* push should add a state to heap with complexity
* push should remove and return the state in the heap with the lowest f-cost with complexity
* contains should return true if the state is in the frontier with complexity or better
* replace if should replace the given state in the heap if it has a higher path-cost with the resulting queue still being a valid heap. This should have a complexity of or better

**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.

**Testing**

You should implement tests for your frontier queue data module in the catch-based test test\_frontier\_queue.cpp. The CMakeLists.txt in the starter code is setup to run the provided puzzle solver tests as well as your test.

**Submission**

Once you are satisfied your code satisfies the project specification, upload the zip file containing your submission, **through Canvas** at the assignment link). The list of files to include is: frontier\_queue.hpp, frontier\_queue.tpp, and test\_frontier\_queue.cpp. Again, the build target called "submission" is configured by default to create this file with the correct contents in your build directory.

You should not submit the other files from the starter code, nor your build directory.