Project 4: Pathfinding

Motivation:

Computer aided design (CAD) tools in the area of electrical and computer engineering are important in the design of fabricated integrated circuits (ICs) which range from application specific ICs (ASICs) to processors. They also support the design for reconfigurable devices such as Field Programmable Gate Arrays (FPGAs). There are many other examples of CAD tools in other fields, from Mechanical Engineering, Civil Engineering, Aerospace Engineering, Architecture, and many others.

In terms of chip design CAD, one of the fundamental problems is the routing of signals from one place to another, be it to connect transistors, gates, or look-up tables found in full custom, standard cell, or FPGA-based hardware designs, respectively. An important fundamental technique for solving this problem is called pathfinding. In this project you will develop a pathfinding algorithm for a maze. The project is broken into two parts. In the first part you will build a single-ended queue (deque). In the second part you will use this deque to build a state search algorithm to find a path through a maze.

Part I: Deque

In this part you will define, implement and test a deque. The file `abstract_deque.hpp` in the starter code defines the templated interface, `AbstractDeque`. You should define a template `Deque` that publicly inherits from the interface in the header file `deque.hpp`. This template should override the following methods from the interface (see `abstract_deque.hpp` for details) and provide a constructor, copy constructor, copy-assignment operator, and destructor.

- isEmpty
- pushFront
- popFront
- front
- pushBack
- popBack
- back

`Deque` should use a singly-linked list for storage. The template method implementations should be placed in the file `deque.tpp` and be included in the `deque.hpp` header.

Define tests for your deque implementation in the file `test_deque.cpp`, which uses Catch as in previous assignments.

Part II: Pathfinder

Queues can be used in "state-space search" algorithms. Recall the bubble diagram concept from digital logic, building and traversing a state diagram is a good way to represent how a finite state machine (FSM) functions. When debugging a complex state machine, you may need to search the state space to ensure there are no bad states in your implementation. In some FSMs there is an initial state, intermediate states, and a goal state to be reached. To search the state space in software, a State type and operations on a problem instance such that:

- 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

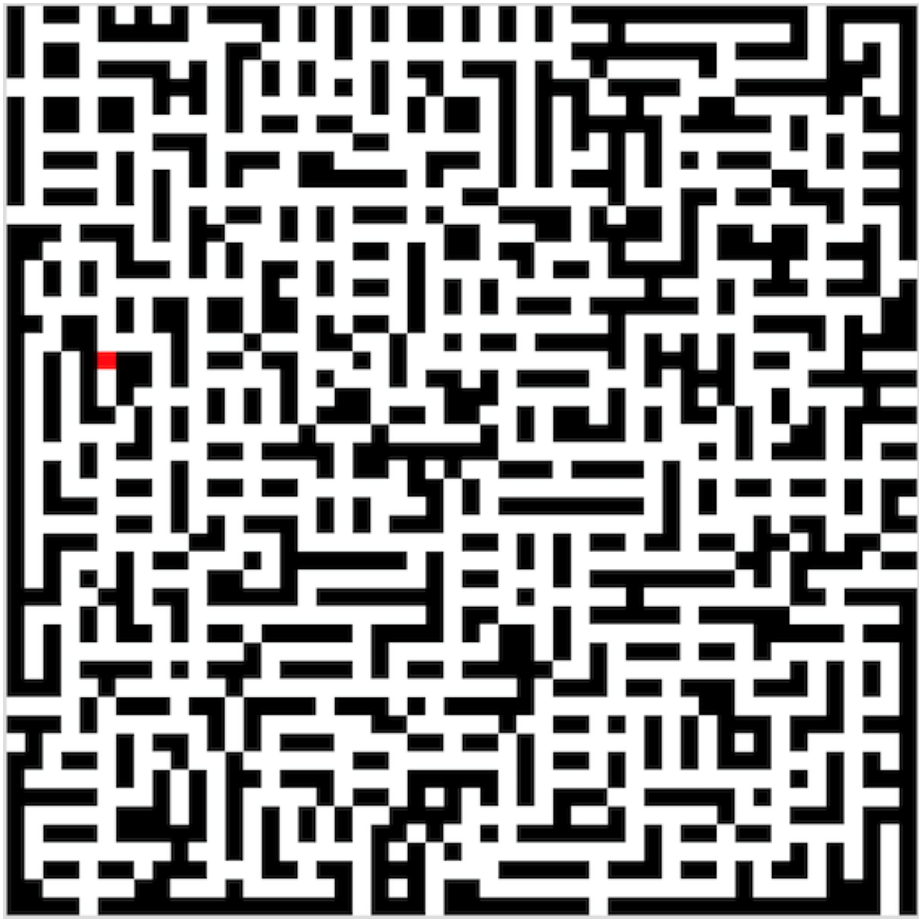
To traverse the state space we will use the breadth-first-search (BFS) algorithm.

```
function breadth_first_search(problem) returns a solution or failure

   s = problem.initial()
   if problem.goal(s) return s

   frontier is a FIFO 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
```

This algorithm can be used to solve a CAD routing problem. To illustrate this we will use a maze searching application which is similar in many respects to the routing problem.

Write a program called pathfinder that can solve mazes like the following of arbitrary size. Black squares are considered to be walls and cannot be occupied, white squares are open cells, and a single red square is the starting location. Allowed moves are left, right, up, and down only (no

diagonal moves) within the bounds of the maze if a cell is not occupied. Your program should find the nearest exit if a solution exists and mark it with the color green. The nearest exit is defined as a white square on the maze border requiring the smallest number of moves to reach it from the starting location.



In our case, the problem is defined by the maze structure, which is represented by a color image where the pixels represent the cells of the maze and whose color determines if the cell is open (WHITE) or a wall (BLACK). The *states* are pixel coordinates (a row and column index) in the maze and the number of moves from the initial state to those coordinates. To prevent confusion regrading coordinate systems, we define four image directions based on the pixel coordinates (row r and column c); that is, consider a pixel with coordinates (r,c) -- previousRow is defined as (r-1,c), nextRow as (r+1,c), previousColumn as (r,c-1), and nextColumn as (r,c+1). **This is the ordering the actions must be considered in the breadth-first search.**

The initial state is defined by a single RED pixel. The goal state is defined by being an image boundary pixel and having the smallest required moves to reach it and should be colored GREEN in the output. If the start pixel is a goal, then it should be recolored from RED to GREEN.

The frontier should use your deque implementation. The explored and frontier inclusion tests can be done efficiently using an image in our problem (and there is no inclusion test in the deque anyway).

Specification

The command-line usage of the pathfinder program is as follows (assuming a linux executable):

```
./pathfinder maze_input.png maze_output.png
```

The program should read the maze as the image file maze_input.png and write the solved maze as maze_output.png. A simple image library is provided to read, write, and manipulate the images. If a solution exists the output image should be the same as the input image with the correct exit colored green and the string "Solution Found" written to standard output. If no solution exists the output image should be the same as the input image and the string "No Solution Found" written to standard output.

On success your program should return EXIT_SUCCESS. On a user or application error your program should print an informative error message starting with the string "Error" to **standard error** and return EXIT_FAILURE. Examples of such errors are

- the program cannot read or write the specified image files,
- the input maze image contains colors other than black, white, or red, or
- the input maze has more than one red pixel.

See the file lib/image.h for documentation on how to use the image library. Note in particular it defines constants BLACK, WHITE, RED, and GREEN that should be used in your program when the appropriate color is needed. The program compare.cpp used in the tests, demonstrates how to use the library, including how to read files in PNG format, access pixels, and compare them.