Assignment 14: Solving the ice sliding puzzle - the rocky version

Course 'Imperative Programming' (IPC031)

Make sure to start working on the assignment **before the lab** session,
otherwise you will be too late for the deadline

1 Background

Last week we implemented the ice sliding puzzle, such that we could play it ourselves. Perhaps you are a puzzle god and managed to solve all the puzzles in the minimum number of steps, but for us mere mortals this is a hard task. Thankfully you have been learning about the breadth-first and depth-first algorithms, which can be used to find the puzzle solutions for us. In this grand finale assignment you will implement both algorithms to create an automatic puzzle solver, giving you some hands-on experience with these algorithms that are key exam subjects.

In our specific case finding a solution state alone isn't all that interesting. We could easily teleport the flamingo to the rescue location and call it "solved". For us it is much more interesting to find out which steps we should take to get to this solution state in as few steps as possible. As such we will construct a "solution path" using the search algorithms. Such a path is a sequence of puzzle configurations, from initial state to a solved state. Each move we perform creates a new puzzle state, which is added to the solution path. This means that for a puzzle that can be solved in n steps, the solution path contains $n + 1$ puzzle states. The first state is our puzzle as we began the search, and the last state will have the flamingo on the rescue location.

2 Learning objectives

After doing this assignment you are able to:

- implement breadth-first and depth-first problem solving techniques for new search problems.

3 Assignment

On Brightspace, you find the file "`assignment-14-mandatory-files.zip`". It contains a number of text files with names "`challenge.m.nsteps.txt`", where `m` is the number of the challenge and `n` is the least number of steps required to solve the challenge. In these text files a puzzle starting configuration is described by a matrix of cells that have the following representation:

1. an empty ice cell by `'.'`
2. a rock by `'r'`
3. the flamingo by `'f'`, and if it is on the rescue location by `'F'`
4. the empty rescue location by `'x'`

Part 1: Puzzle logic

Last week you designed your own `Puzzle` data structure, and functions to load, display, and play the puzzle. This should give you all the logic you need to implement the search algorithms of parts 2 and 3. Copy over the puzzle code from last week that you need for the search algorithms. If you did not manage to get the puzzle logic to work correctly, then use your feedback to improve it, or ask help from a TA if you are still unsure on how to fix it.

Part 2: Solve ice sliding, breadth first

Design and implement a breadth-first search algorithm `breadth_first` to solve a given starting configuration, which returns the number of steps needed to solve the puzzle, and stores the solution path in an output parameter if one was found. If the starting configuration is not valid, the integer constant `BAD_FORMAT` is returned instead. Likewise, if no solution was found, the integer constant `NO_SOLUTION` is returned. Test your implementation with the given challenges. Use the solutions to verify that the least number of moves to solve these challenges is correct.

Part 3: Solve ice sliding, depth first

Design and implement a recursive, depth-first search algorithm `depth_first` to solve a given starting configuration, which returns the number of steps needed to solve the puzzle, and stores the solution path in an output parameter if one was found. Bound the depth of the search by an integer parameter, which may have the special value `NO_DEPTH_LIMIT` that disables the depth bounding. The algorithm must keep track of the shortest-solution-so-far, and abandon an attempt if its length exceeds the length of the shortest-solution-so-far or the given bound, if enabled. Note that if the bound is too small, the algorithm will not find a solution. As in part 2 use the `BAD_FORMAT` and `NO_SOLUTION` constants as your return value when applicable, and test your implementation with the given challenges.

4 Products

As product-to-deliver you upload to Brightspace:

- "`main.cpp`" that you have created with solutions for each part of the assignment.
- "`main_test.cpp`" that has been extended with non-trivial unit tests for each new function that you have developed in "`main.cpp`".

Deadline

Lab assignment: Friday, December 22, 2023, 23:59h