CSCI-1200 Data Structures — Fall 2022
Homework 6 — Recursive Bridges

In this homework we will develop a solver for a 2D grid-based path finding puzzle game. The goal is to connect paths between the different colored endpoints, using provided "bridges" and optionally covering the entire board. The link below is a website that allows you to play through the bridges game, solving puzzles by hand. There are also versions of this puzzle game available for mobile devices.

You are encouraged to play with these games so you understand the rules. *As you play, think about how you as a human solve these puzzles.* As the puzzles grow larger (larger board, more colors, more bridges) how and why does it become more challenging? Can you generalize your method of finding a solution so you can teach a friend how to play and ultimately, "teach" the computer to play?

https://www.xpgameplus.com/games/flowbridges/index.html

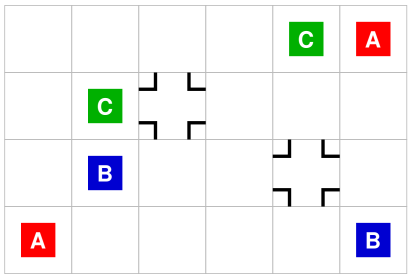
IMPORTANT NOTE: You may not search for, study, or use any code or techniques related to existing solvers for this puzzle game. *Please carefully read the entire assignment and study the examples before beginning your implementation.*

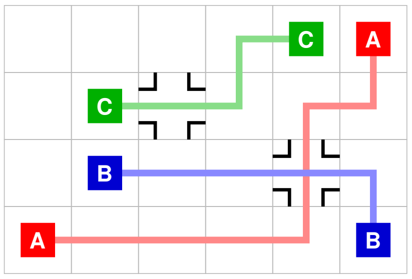
Bridges - How to Play

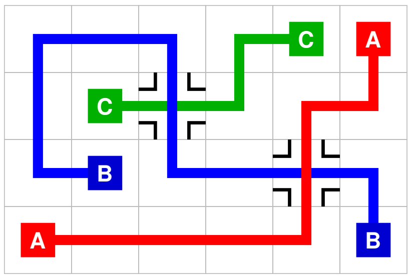
Bridges is a game played on a rectangular grid with *w* x *h* cells. A pair of endpoints for each of *c* colored paths are placed on the board. Additionally, a number (*b*) of the cells on the board are labeled as *bridges*, where two different colored paths will cross. Other than the bridge cells the paths should not cross. We can also optionally require that the collection of paths must fill the entire board.

An example puzzle board is shown below on the left. The three pairs of endpoints are successfully connected by paths snaking through the grid. In these diagrams the endpoints are matched both with color and a capital letter. For our implementation, they will just be labeled with capital letters.

The middle diagram below shows a solution to the puzzle. All three paths are completed and do not violate the basic rules which allow one path per cell, except bridge cells which allow two paths to cross. Paths can go straight or make a 90 degree turn in any cell, except bridge cells where the path must go straight.

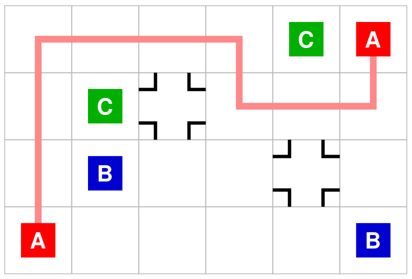


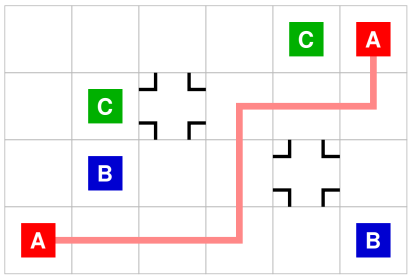


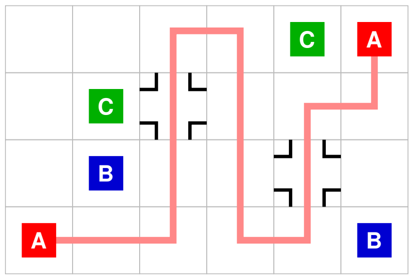


The right diagram above shows a solution to the puzzle if we enforce the optional requirement that the paths as a whole must *cover* the entire board. Every cell must be visited by a path, and every bridge must be crossed by two different paths. This particular puzzle has only 1 solution that covers the board, but it has 4 other solutions that do not cover the board (not including the one shown in the middle diagram). *Can you find them by hand?*

The first part of implementation for this homework is to enumerate all legal paths on the board for a single color. Let's consider the red path labeled A first. The diagrams below show three of the paths that correctly connect the the endpoints.







The paths must avoid the cells labeled with endpoints for other paths. The paths are allowed to use the bridge cells, but they must pass straight through those cells. Note: A path may not cross itself, even if it uses a bridge cell. How many paths can you find that correctly connect the red endpoints? *Hint: There are a total of 20 different paths!*

Command Line Arguments & Input/Output

```
....CA
.C#...
.B..#.
A....B
```

Your program will accept one or more command line arguments. The required argument is the name of a puzzle board file similar to the file shown on the right, which encodes the puzzle board discussed & illustrated above. The path endpoints are labeled with capital letters. We use '`.`' to represent an empty cell, and '`#`' to represent a bridge cell.

By default, your program should print to `std::cout` a single ASCII art solution (if one exists). We provide basic code to read in the puzzle board and produce the required ASCII art output format. You may use or modify any or all of this code, but your output artwork must match this format exactly in order to be graded correctly.

```
+---+---+---+---+---+---+
|                       |
| bbbbbbbbb   ccccC   A |
| b       b   c       a |
+ b +   + b + c +   + a +
| b       b   c       a |
| b   Cccc#cccc   aaaaa |
| b       b       a     |
+ b +   + b +   + a +   +
| b       b       a     |
| bbbbB   bbbbbbbb#bbbb |
|                 a   b |
+   +   +   +   + a + b +
|                 a   b |
| Aaaaaaaaaaaaaaaaa   B |
|                       |
+---+---+---+---+---+---+
```

By default your program should find a single solution to the basic puzzle (connecting all endpoints, but not necessarily covering the board). If the optional command line argument `--all_solutions` is specified, your program should output all valid solutions and also print a message at the bottom with the total number of solutions. If the optional command line argument `--covers_board` is specified (with or without `--all_solutions`), solutions that do not visit all cells or do not cross every bridge in both directions should be omitted. If there are no solutions, your program should print the empty grid followed by "`No Solutions`".

We *strongly recommend* that you get started by finding all legal paths on the board for a single color. We can test this by specifying the argument `--one_color` followed by the letter of the chosen color. Alternatively, the `--all_paths` argument will enumerate all paths connecting any pair of endpoints. Your program should print a message at the bottom with the total number of paths or the empty grid followed by "`No Paths`" if there are no legal paths for that color or for any color.

If there are multiple solutions and you are not asked to find all of them you may output any solution for full credit. If you are asked to output all paths or all solutions, they may be printed in any order for full credit.

Additional Requirements: Recursion & Big O Notation

You must use recursion in a non-trivial way in your solution for finding paths and putting the paths together to form a full solution to the puzzle. As always, we recommend you work on this program in logical steps. *IMPORTANT NOTE: This problem is computationally expensive, even for medium-sized puzzles! Be sure to create your own simple test cases as you debug your program.*

Once you have finished your implementation, analyze the performance of your program using big O notation. What important variables control the complexity of a particular problem? The dimensions of the board (w and h)? The number of colored path end points (c)? The number of bridges (b)? The length of the longest path in the solution (e)? In your `README.txt` file write a concise paragraph ($<$ 200 words) justifying your answer. Include a table summarizing the running time and number of solutions found by your program on each of the provided examples for the different command line arguments. *Note: It's ok if your program can't solve the biggest puzzles in a reasonable amount of time.*

You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.

Homework 6 Bridges Contest Rules

- All students are required to submit their program to the contest. Extra credit will be awarded for programs that have a strong performance in the contest.
- Members of the teaching staff will also be submitting to the contest... who will win???
- Contest submissions are a separate Submitty gradeable. Contest submissions are due Tuesday, November 1st at 11:59pm. You may not use late days for the contest. (The regular homework deadline is Thursday, Oct 27th at 11:59pm and late days are allowed for the regular homework submissions.)
- You may submit the same code for both the regular homework submission and the contest. Or you may make a small or significant change for the contest.
- Contest submissions *do not* need to use recursion.
- We will compile your code with optimizations enabled (`g++ -O3 *.cpp`) and run all submitted entries on the homework server.
- Programs must be single-threaded and single-process.
- We will run your program by *redirecting* `std::cout` to a file and measure performance with the UNIX `time` command. For example:

```
time bridges.out puzzle4.txt --all_solutions > out_puzzle4_all.txt
```

- You may want to use a *C++ code profiler* to measure the efficiency of your program and identify the portions of your code that consume most of the running time. A profiler can confirm your suspicions about what is slow, uncover unexpected problems, and focus your optimization efforts on the most inefficient portions of the code.
- We will be testing with and without the optional command line arguments `--one_color`, `--all_paths`, `--one_solution`, `--all_solutions`, and `--covers_board` and will highlight the most correct and the fastest programs.
- You may submit one or two new Bridges puzzle board file *with your regular homework submission* for possible inclusion in the contest. Name these test files: `puzzle_smithj_1.txt` `puzzle_smithj_2.txt` (where `smithj` is your RCS username). Extra credit will be awarded for interesting test cases that are used in the contest. Caution: Don't make the test cases so difficult that your program cannot solve them in a reasonable amount of time!
- In your `README_contest.txt` file, describe the optimizations you implemented for the contest and summarize the performance of your program on all test cases.