Carcassonne Recursion

Your task for this homework is to solve tile placement puzzles inspired by the board game "Carcassonne" using the technique of recursion. You can read more and see examples of the the full game here:

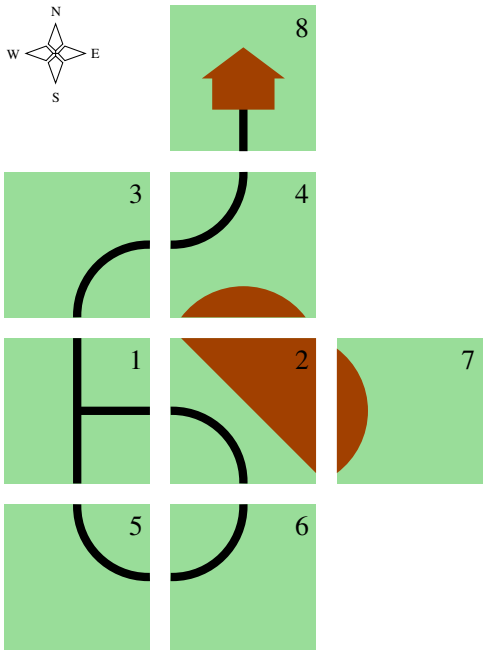
http://en.wikipedia.org/wiki/Carcassonne_(board_game)
http://norvig.com/carcassonne.html

Understanding the non-linear word search program from Lectures 13 & 14 will be helpful in thinking about how you will solve this type of problem. We strongly urge you to study and play with that program, including tracing through its behavior using a debugger or `cout` statements or both. *Please carefully read the entire assignment and study the provided code before beginning your implementation.*

You will be given a sequence of square tiles that must be played one at a time onto a two-dimensional grid board. Each of the four tile sides is labeled as "road", "city", or "pasture". The first tile may be played anywhere, but each following tile must touch a previously played tile along an edge. All touching tile edges must match. All tiles must be played in the order given – the sequence may not be rearranged. Finally, for our version of this game, a board layout is a "Solution" only if every "road" and "city" tile edge has a matching neighbor tile touching it. The only tile edges without neighbors should be labeled "pasture".

A tile with 0 or 1 road edges and no city edges is called an *abbey* tile and we draw a little building on the tile. Note that not all labelings of road/edge/pasture are legal tiles. (And, we use a simpler set than the original board game.) The provided code will check for illegal tiles in the input. You may assume that all input files are properly formatted and use only legal tiles.

Below is a sample input file (`puzzle5.txt`). The keyword `tile` is followed by four strings that specify the edges of the tile in the *north*, *east*, *south*, and *west* directions. On the right is a diagram showing these tiles arranged to form the only solution for this particular tile puzzle. Pastures are green, cities are brown, and roads are thick black lines. Note the numbers on each tile, which correspond to the line number from the input file. Every tile touches the edge of at least one tile with a smaller number that was played earlier in the game.




```
tile road road road pasture
tile city city road road
tile pasture road road pasture
tile road pasture city road
tile road road pasture pasture
tile road pasture pasture road
tile pasture pasture pasture city
tile pasture pasture road pasture
```

Your program will expect one or more command line arguments, e.g.:

```
./carcassonne.exe puzzle5.txt -board_dimensions 4 3
./carcassonne.exe puzzle5.txt -board_dimensions 4 3 -all_solutions
./carcassonne.exe puzzle5.txt -board_dimensions 4 3 -allow_rotations
./carcassonne.exe puzzle5.txt -allow_rotations -all_solutions
```

The first argument specifies the name of the input puzzle file. To begin, we recommend that you also use the optional argument `-board_dimensions <rows> <columns>`, which specifies maximum dimensions for the solution grid.

If the optional argument `-all_solutions` is *not* specified, your program should output *any* one solution. If `-all_solutions` is specified, your program should output *all* solutions, in any order, followed by the message "`Found XX Solutions(s).`" (where `XX` is the integer number of solutions). Note that if the input contains duplicate tiles, we do not consider swapping these tiles in the output grid to be a different solution. You should only output solutions that create a different map. If there are no solutions, your program should output "`No Solutions.`"

Finally, if the `-allow_rotations` argument is specified, each tile may be rotated *clockwise* 90, 180, or 270 degrees before it is inserted into the grid. In many cases this produces more solutions to the puzzle. However, we do not count (and you should not output) rotations of the whole board as a different unique solution.

All program output should be sent to `std::cout`. Each solution must be output with the keyword "`Solution:`" followed by the row and column coordinates and rotation (0°, 90°, 180°, or 270°) of each tile in the input (in order). For example, here are the 4 different solutions for puzzle 4 when rotations are allowed:

```
Solution: (1,1,0)(1,2,0)(2,1,0)(2,2,0)(3,2,0)(0,2,0)(1,0,0)(2,0,0)
Solution: (1,1,0)(1,2,0)(2,1,0)(2,2,90)(2,3,270)(0,2,0)(1,0,0)(2,0,0)
Solution: (1,1,0)(2,1,90)(1,0,90)(2,0,180)(3,0,0)(0,0,0)(1,2,90)(2,2,270)
Solution: (1,1,180)(0,1,270)(1,2,270)(0,2,270)(0,3,270)(2,2,180)(0,0,0)(1,0,0)
Found 4 Solution(s).
```

For human readability, we also print an ASCII art representation of each finished board. (The submission server will only be grading the lines that begin with "Solution:" and the final summary line with the total number of solutions.) Please study the sample output files provided on the webpage, and match the formatting exactly (except for choice of single output solution, choice among duplicate solutions, and order of all solutions).

Provided Code, Additional Requirements, and Homework Submission

We provide the `Tile`, `Board`, and `Location` classes, and code to parse the command line arguments, load the puzzle input file, and create the human-friendly ASCII art board output. You may use or modify any or all of the provided code in your solution.

You must use recursion in a non-trivial way in your solution to this homework. As always, we recommend you work on this program in logical steps. Partial credit will be awarded for each component of the assignment. Start by placing tiles onto the board that follow the game rules of matching edges and sequential playing (touching an edge of a previously placed tile). Then, work on creating boards with fewer or no unmatched road and city edges. Stopping here will earn the majority of points for this homework. The next step is to find all of the different solutions to the input puzzle. Move on to finding solutions that require rotating one or more pieces in the input collection. And finally, doing all this when no board dimensions are specified is worth full credit. *IMPORTANT NOTE: This problem is computationally expensive, even for medium-sized puzzles with less than a dozen tiles! 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 algorithm using order notation. What important variables control the complexity of a particular problem? The dimensions of the board (h and w)? The number of tiles (t)? The number of road (r) and city (c) edges? The number of duplicate tiles? Whether rotations are allowed? Etc. In your `README.txt` file write a concise paragraph (< 200 words) justifying your answer. Also include a simple table summarizing the running time and number of solutions found by your program on each of the provided examples. Indicate the command line arguments for each test (which puzzle, board dimensions, all solutions, and rotations allowed).

All students are required to submit their program to the Homework 6 contest (see below). Extra credit will be awarded for programs that have a strong performance in the contest.

Carcassonne Contest Rules

- Contest submissions are a separate homework submission. Contest submissions are due Saturday Apr 4th at 11:59pm. You may not use late days for the contest. (The regular homework deadline is Thursday Apr 2nd 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.
- Contest submissions must follow the output specifications and match the formatting of the examples posted on the course webpage.
- We will recompile (`g++ -O3 *.cpp`) and run all submitted entries on one of our machines. Programs that do not compile, or do not complete the basic tests in a reasonable amount of time with correct output, will not receive extra credit.
- 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 carcassonne.exe puzzle1.txt -all_solutions > output.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 `-board_dimensions`, `-all_solutions`, and `-allow_rotations` and will highlight the most correct and the fastest programs.
- You may submit up to two interesting new test cases for possible inclusion in the contest. Name these tests `smithj_1.txt` and `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, describe your new test cases, and summarize the performance of your program on all test cases.
- Extra credit will be awarded based on overall performance in the contest.