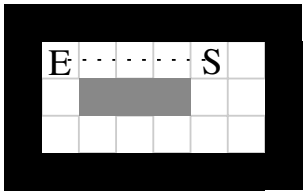
CSCI-1200 Data Structures — Fall 2009
Homework 6 — Maze Maker

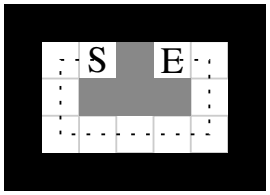
In this homework we will continue with the theme from Lecture 12 and Lab 8 of finding paths in rectilinear grids. For the first part you will write a recursive function to find the shortest path from start to end in a maze with obstacles. Unlike Lab 8, the path is not required to always decrease the distance to the end point or origin. In fact, often in walking around an obstacle you will first move away from the endpoint before you can move closer. For the second part you will write, design, and implement an algorithm (also using recursion) to automatically generate a maze with a specified minimum path length. *Please read the entire handout before beginning your implementation.*

Part 1: Finding the Shortest Path

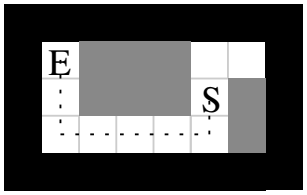
We have provided some starter code in `main.cpp`, `board.h`, and `board.cpp` that you are encouraged to use (but you may modify it as you see fit or not use it at all). This code includes several test cases for the first part, finding the shortest path in a maze. The maze is represented as a vector of strings. Your job for this part is to use recursion to complete the function `ShortestPath`, which should act as a driver function for your recursive implementation (you'll need to write one or more helper functions to do the real work). The `ShortestPath` function returns an integer indicating the fewest steps (up, down, left, or right) that are needed to move from the start (labeled 'S') to the end (labeled 'E') within the maze, avoiding all obstacles. If there is no path from start to end, the function should return -1. Study the examples below:



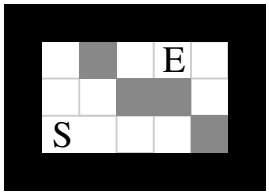
shortest path length = 4



shortest path length = 10



shortest path length = 7



no path

Make sure to add your own test cases to be sure your code is debugged. Once you have a working `ShortestPath` function, you can proceed on to the second part...

Part 2: Creating a Maze

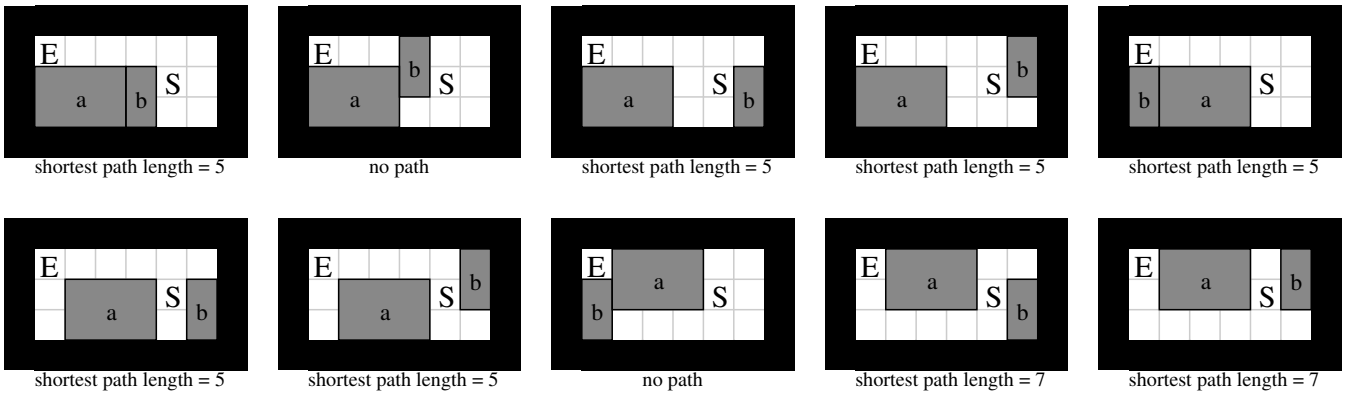
For this part you will load the requirements for the maze from an input file (we have provided the parse code) that is specified on the command line. And, as usual, you will write the output to a second file that is also specified on the command line. The third argument indicates whether all solutions should be generated, or if the algorithm can stop after the first solution is found.

```
./maze_maker puzzle0.txt output0.txt all_solutions
./maze_maker puzzle0.txt output0_one_solution.txt one_solution
```

Here's a sample input file:

```
board 6 3
start 4 1
end 0 0
desired_path_length 7
block 3 2 a
block 1 2 b
```

The input file contains the board dimensions (width & height) for the maze, the start and end locations (x,y coordinates) for the path within the maze, the desired shortest path length and finally, 0 or more rectangular blocks that will be placed in the board to form obstacles. In addition to the width & height, each block has a unique letter code, so we can label it appropriately when we visualize the packing of blocks into the board. The blocks are not allowed to rotate — they must be placed in the board in the specified orientation. First let's create all possible ways to put the blocks into the board:



For this example, there are 10 ways to place these blocks into the board. Your algorithm to generate these candidate boards should be recursive. Now for each of those candidates we can identify which boards have the desired shortest path length (2 solutions, in this case). Finally, we can also identify how many of these solutions are unique. In this example, both solutions with path length 7 are unique. But notice that 2 of the candidate boards (the top left and the top right, in the diagram above) contain equivalent obstacles, and if they were in fact solutions, they would represent only 1 unique solution. All of this information is sent to the output file. Please follow the example output file format below as closely as possible (more examples on the webpage).

```
10 candidate board(s)

########
#E  b  #
#aaabS #
#aaa   #
########

########
#E     #
#aaabS #
#aaab  #
########

**** 8 MORE CANDIDATE BOARDS OMITTED FOR SPACE ****

2 solution(s) with path length 7

########
#Eaaa b#
#.aaaSb#
#.....#
########

########
#Eaaa  #
#.aaaSb#
#.....b#
########

2 unique maze(s)

########
#E### ##
# ###S##
#      #
########

########
#E###  #
# ###S##
#     ##
########
```

Note that the order of the boards in the output file depends on your exact algorithm details, and you do not need to match the order in the examples. Also note that the solution boards all include an ascii art dotted line illustrating a shortest path solution on the board (there may be more than one, but you only need to draw one). Also note that in the unique mazes output, all block letters have been converted to '#' characters and the solution path is not drawn.

The sample output above is for the case when "`all_solutions`" is specified on the command line. When "`one_solution`" is specified on the command line, the output file should just contain a single solution (or the message "`no solutions`" if there are no solutions):

```
########
#E### ##
# ###S##
#     #
########
```

Algorithm Analysis

For larger, more complex examples, this is a really hard problem. Your program should be able to handle the small puzzles we provide in a reasonable about of time (less than a couple minutes). You should make up your own test cases as well to understand this complexity. Include these test cases and sample output with your submission. Summarize the results of your testing, which test cases completed successfully and the approximate "wall clock time" for completion of each test. The UNIX/cygwin `time` command can be prepended to your command line to estimate the running time:

```
time ./maze_maker puzzle0.txt output0.txt all_solutions
```

Once you have finished your implementation and testing, analyze the performance of your algorithm using order notation. What important variables control the complexity of a particular problem? The width & height of the grid, the number of blocks, the length of the path, the number of solutions, etc.? In your *plain text* `README.txt` file, write a concise paragraph (< 200 words) justifying your answer.

Additional Information

You must use recursion for this homework submission. You may use any of the data structures and techniques we have discussed so far in this course. Do all of your work in a new folder named `hw6` inside of your Data Structures homeworks directory. Use good coding style when you design and implement your program. Don't forget to comment your code! Use the template `README.txt` to list your collaborators and any notes you want the grader to read. When you are finished please zip up your `hw6` folder exactly as instructed for the previous assignments and submit it through the course webpage.

Extra Credit Contest!

For extra credit you may implement techniques to improve the performance (running time) of the basic algorithm and submit your homework to the HW6 contest. We will hold a contest to find the best maze maker program. You must use recursion for the homework assignment, but you are not required to do so for the contest. Entries must include a plaintext `README_contest.txt` file explaining their optimizations. Extra credit will be awarded for *all* contest entries that compile and run the basic test cases, so everyone should enter. Entries will be judged based on their performance (accuracy *and* speed) on a variety of test cases, including time to find one solution and time to find all solutions. The winners will be announced during lecture (date TBA) and the winners must be present to claim their prizes.

To enter the contest, prepare your entry in a folder named `hw6_contest` and submit it on the homework submission site. The usual homework late day policy applies to the non extra credit (non contest) homework submission. All contest entries are due on Saturday, October 30th by 11:59pm (this will not count against your late day total).