Project 4: marsh crossing

CPSC 335 - Algorithm Engineering

Spring 2019

Instructors: Krutik Amin ([kamin@fullerton.edu](mailto:kamin@fullerton.edu)), Doina Bein ([dbein@fullerton.edu](mailto:dbein@fullerton.edu))

# Abstract

In this project you will implement two algorithms that both solve the *marsh crossing problem.* The first algorithm uses exhaustive search, is similar to the exhaustive search algorithm from project 2, and takes exponential time. The second algorithm uses dynamic programming, and takes cubic time.

# The Hypothesis

This experiment will test the following hypothesis:

oy

# The Marsh Crossing Problem

The “marsh crossing problem” is a puzzle that comes from child tale, not sure about the title through. The person is chased by fiery animals and crossing a marsh will misdirect the followers. But the marsh is full of thickets that are impenetrable. The person starts in one corner of a matrix (location [0][0]) and needs to reach safety in the corner across the marsh (location [r-1][c-1]). The thickets are impenetrable and crossing the marsh can be done only moving right, from location [i][j] to location [i][j+1], or down, from location[i][j] to [i+1][j]. The person can move if there is no thicket at that location. We will represent the marsh area as a 2D grid, like the following:

.XX.X.......X..X.........

.....X.X........X........

................X.....X..

..........X....X......XX.

....X...X.X...X..........

...X.......X.X.........X.

.X...X......XX....X.X....

........X.X..............

.....X.......X.....X..X..

........X...X..X..X.X....

X.X....XX.....XXXX.......

.....X.....X.......X.....

The person starts at row 0 and column 0, i.e. coordinate (0, 0), at the top-left corner. Each . cell represents a passable spot and each X represents impenetrable thicket. The person’s goal is to plan an escape route to cross the marsh while avoiding the thicket. The objective of the problem is to compute the number of different paths to cross the marsh. Two paths are different is they differ by at least one spot.

For the previous grid, the optimal solution is

**17625**

We can define this puzzle as an algorithmic problem.

|  |
| --- |
| *Marsh crossing problem* |
| **input:** a r⨉c matrix G where each cell is one of . (passable) or X (impassable); and G[0][0]= .  **output:** the number of different paths starting at (0, 0); where each step is either a start, right move, or down move; that does not visit any X cell |

If the initial cell is blocked, there is no way of moving anywhere so output 0.

If the final cell is thicket, output 0.

## The Exhaustive Optimization Algorithm

Our first algorithm solving the marsh crossing problem is exhaustive. The output definition says that the number of different paths, so this is an exhaustive search algorithm that keeps a counter and does not return after such a path is found but increment the counter instead.

The following is a first draft of the exhaustive search algorithm.

marsh\_crossing\_exhaustive(G):

maxno = total number of different paths originating at (0,0) and ending at (n-1,n-1)

counter = 0 (number of valid paths in G

for len from 0 to maxno inclusive:

for each possible sequence S of {→, ↓} encoded as len:

candidate = [start] + S

if candidate is valid:

counter++

return counter

This is not quite clear, because the precise value of maxno, method of generating the sequences S, and verifying candidates, are all vague.

Since all paths start at (0, 0) and the only valid moves are right and down, valid paths are never backward or upward. So any valid path must reach the bottom-right corner of the grid. The grid has r rows and c columns, so this path involves (r-1) down moves and (c-1) right moves, for a total of r + c - 2 moves.

There are two kinds of move, right → and down ↓. Coincidentally there are two kinds of bits, 0 and 1. So we can generate move sequences by generating bit strings, using the same method that we used to generate subsets in section 6.5.4 of ADITA. We loop through all binary numbers from 0 through and interpret the bit at position *k* as the up/down step at index *k.*

A candidate path is valid when it follows the rules of the marsh crossing problem. That means that the path stays inside the grid, and never crosses a thicket (X) cell.

Combining these ideas gives us a complete and clear algorithm.

marsh\_crossing\_exhaustive(G):

len = r + c -2

counter = 0

for bits from 0 to 2len - 1 inclusive:

candidate = [start]

for k from 0 to len-1 inclusive:

bit = (bits >> k) & 1

if bit == 1:

candidate.add(→)

else:

candidate.add(↓)

if candidate stays inside the grid and never crosses an X cell:

counter++

**return counter**

Let Then the for loop over bits repeats times, and the inner for loops repeat times, and the total run time of this algorithm is This is a very slow algorithm.

## The Dynamic Programming Algorithm

This problem can also be solved by a dynamic programming algorithm. This dynamic programming matrix A stores partial solutions to the problem. In particular,

A[r][c] = the number of different valid paths that start at (0, 0) and end at (r, c); or 0 if (r, c) is unreachable

Recall that in this problem, some cells are filled with thickets and are therefore unreachable by a valid path.

The base case is the value for A[0][0], which is the trivial path that starts at (0,0) and takes no subsequent steps: A[0][0] = 1

We can build a solution for a general case based on pre-existing shorter paths. The person can only move right and down. So there are two ways the person can reach the cell at location (i, j).

1. The path reached the location above (i, j) and with an additional down step reaches (i,j)
2. The path reached the location to the left of (i, j) and with an additional right step reaches (i,j)

The algorithm should add both alternatives, which in this problem means adding different paths.

However, neither of these paths is guaranteed to exist. The from-above path (1) only exists when we are not on the top row (so when i>0), and when the cell above (i, j) is not thicket. Symmetrically, the from-left path (2) only exists when we are not on the leftmost column (so when j>0) and when the cell to the left of (i, j) is not thicket.

Finally, observe that A[i][j] must be None when G[i][j]==X, because a path to (i, j) is only possible when (i, j) is not a thicket.

Altogether, the general solution is:

A[i][j] = None and stays None if G[i][j]==X

A[i][j] = the sum of paths from\_above and from\_left where

from\_above = 0 if i=0 or G[i-1][j]==X; or A[i-1][j] otherwise (move is [↓])

from\_left = None if j=0 or G[i][j-1]==X; or A[i][j-1] otherwise (move is [→])

Putting the parts together yields a complete dynamic programming algorithm.

marsh\_crossing\_dyn\_prog(G):

A = new r⨉c matrix

# base case

A[0][0] = 1

# general cases

for i from 0 to r-1 inclusive:

for j from 0 to c-1 inclusive:

if G[i][j]==X:

A[i][j]=None

continue

from\_above = from\_left = 0

if i>0 and A[i-1][j] is not None:

from\_above = A[i-1][j]

if j>0 and A[i][j-1] is not None:

from\_left = A[i][j-1]

A[i][j] = sum of from\_above and from\_left; or None if both from\_above and from\_left are None

return A[r-1][c-1]

The time complexity of this algorithm is dominated by the general-case loops. The outer loop repeats times, the inner loop repeats times, **for a total of time**. While  **is** not the fastest time complexity out there, it is polynomial so considered tractible, and is drastically faster than the exhaustive algorithm.

# Obtaining and Submitting Code

This document explains how to obtain and submit your work:

[GitHub Education / Tuffix Instructions](https://docs.google.com/document/d/1XspE5hRtGfc3s66cx0Fhg66PEwub2kY7SlX4lWptVeI/edit?usp=sharing)

Here is the invitation link for this project:

<https://classroom.github.com/g/PjsAMWvN>

# Implementation

You are provided with the following files.

1. crossing\_algs.hpp is a C++ header that defines two functions, one for each of the algorithms defined above. These function bodies are marked TODO and your assignment is to fill them in with algorithm implementations.
2. crossing\_types.cpp is a C++ header that defines data types for the grids, paths, and related objects for the marsh crossing problem. This code is complete; you should not modify this file.
3. crossing\_timing.cpp is a C++ program with a main() function that measures one experimental data point for each of the algorithms. You can expand upon this code to obtain several data points for each of your algorithm implementations.
4. Makefile, crossing\_test.cpp, timer.hpp, rubrictest.hpp, and README.md work the same way as in prior projects.

# What to Do

Decide on who will be in your team, or decide to work alone; have one of your team members accept the GitHub assignment by following the invitation link; have any other team members join your team by following the invitation link; and add your group member names to README.md.

Then, implement each of the two algorithms in C++ using the provided skeleton code. Test your code using the provided unit tests.

Once you are confident that your algorithm implementations are correct, modify the timing code to gather numerous data points for the run-time of each of the two algorithms.

Finally, produce a brief written project report ***in PDF format***. Submit your PDF by committing it to your GitHub repository along with your code. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 4.
2. Three scatter plots:
   1. One showing the time complexity of the exhaustive algorithm.
   2. One showing the time complexity of the dynamic programming algorithm.
   3. One showing the performance of both algorithms together on the same plot. This will probably need to be zoomed-in since the exhaustive algorithm is so much slower than the dynamic one.
3. Answers to the following questions, using complete sentences.
   1. Are the fit lines on your scatter plots consistent with these efficiency classes? Justify your answer.
   2. Is this evidence consistent or inconsistent with the hypothesis stated on the first page? Justify your answer.
   3. Compare and contrast the difficulty you found in implementing the two algorithms. What was the most challenging part of implementing each algorithm. Overall, which implementation did you find harder, and why? Which algorithm implementation do you prefer?

# Grading Rubric

Your grade will be comprised of three parts: *Form,* *Function,* and *Analysis.*

*Function* refers to whether your code works properly as defined by the test program. We will use the score reported by the test program, when run inside the Tuffix environment, as your Function grade.

*Form* refers to the design, organization, and presentation of your code. A grader will read your code and evaluate these aspects of your submission.

*Analysis* refers to the correctness of your mathematical and empirical analyses, scatter plots, question answers, and the presentation of your report document.

The grading rubric is below.

1. Function = 13 points, scored by the unit test program
2. Form = 9 points, divided as follows:
   1. README.md completed clearly = 3 points
   2. Style (whitespace, variable names, comments, helper functions, etc.) = 3 points
   3. C++ Craftsmanship (appropriate handling of encapsulation, memory management, avoids gross inefficiency and taboo coding practices, etc.) = 3 points
3. Analysis = 12 points, divided as follows
   1. Report document presentation = 3 points
   2. Scatter plots = 3 points
   3. Question answers = 6 points

*Legibility standard:* As stated on the syllabus, submissions that cannot compile in the Tuffix environment are considered unacceptable and will be assigned an “F” (50%) grade.

# Deadline

The project deadline is Friday, May 10, 11:59 pm.

You will be graded based on what you have pushed to GitHub as of the deadline. Commits made after the deadline will not be considered. Late submissions will not be accepted.