Project 4: Dynamic vs. Exhaustive (40 points)

CPSC 335 - Algorithm Engineering
Fall 2021
Instructor: Doina Bein (dbein@fullerton.edu)
Last updated: Mon Nov 15 20:42:29 PST 2021

Abstract

In this project you will implement two algorithms that both solve the crane unloading problem. The first algorithm uses exhaustive optimization and takes exponential time. The second algorithm uses dynamic programming, and takes cubic time. Check back here regularly (and especially before your final submission), to make sure your submission complies with any changes or edits. You may check the top of the document, where the date of the last edit will be noted.

The Hypothesis

This experiment will test the following hypothesis:

> Polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem.

The Problem

Both algorithms will be used to solve another interesting problem related to containers and loading them into ships. Suppose that you have arrived at the seaport and need to navigate among buildings to reach various cranes that will take your containers and load them into various ships.

	
Rotterdam Port. Available at https://pixabay.com/photos/rotterdam-port-crane-heaven-clouds-1913741/	Available at https://www.pxfuel.com/en/free-photo-jncxe

Each truck is allowed to follow only one route and it can drive only south or east, but cannot drive north nor west. We will represent the seaport driveable area as a 2D grid, like the following.

```
.X..X.........X........
......c................
..................c.....
c............c....c.....
.Xc.........X.X...c.....
........c.......X.c.....
....X......X......cX....
c.c.........X..c.c......
.......................X..
...X..c..........c......
c..X......c.......c..X..
cXXX............X.....X.
```

A truck starts at row 0 and column 0, i.e. coordinate (0, 0), at the top-left corner. Each `c` cell represents a crane, each `X` represents a building, and each `.` cell represents a passable space. A truck's goal is to plan a route for driving that maximizes the number of cranes that can be reached, while avoiding buildings.

Here is an optimal solution for the previous grid:

```
+X..X.........X........
++++++C++++++++++.....
..................C.....
c............g....C.....
.Xc.........X.X...C.....
........c.......X.C.....
....X......X......CX....
c.c.........X..C.C+.....
...................+..X..
...X..c..........c+.....
c..X......c.......C..X..
cXXX............X.....X.
    steps=29 cranes=7
```

In this diagram, the `+` cells represent passable space that the truck drove through, and the capital `C` cells represent cranes that the truck reached and had cargo unloaded. A `.` still represents passable space that was never visited, and lower-case `c` still represents cranes that were neither reached nor used. Observe that the path starts at (0, 0), and moves east and south, but not in any other direction. Also note that there is no valid path in this grid that would reach more than 7 cranes.

We can define this puzzle as an algorithmic problem.

Crane unloading problem
input: a r×c matrix G where each cell is one of . (passable), c (crane), or X (building); and G[0][0]= . **output:** a path starting at (0, 0); where each step is either a start, move east, or move south; that does not visit any X cell; and the number of c cells along the path is maximized

The Exhaustive Optimization Algorithm

Our first algorithm solving the crane unloading problem is exhaustive. The output definition says that the number of cranes reached must be maximized, so this is an exhaustive optimization algorithm (not exhaustive search).

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

```
crane_unloading_exhaustive(G):
      maxlen = maximum number of steps (not counting the start step) in a valid
      truck path in G
      best = None
      for len from 0 to maxlen inclusive:
            for each possible sequence S of {→, ↓} of length exactly len:
                  candidate = [start] + S
                  if candidate is valid:
                        if best is None or candidate is better than best:
                              best = candidate
      return best
```

This is not quite clear, because the precise value of `maxlen`, method of generating the sequences `S`, and checking the validity of candidates, are all vague.

Since all paths start at (0, 0) and the only valid moves are east and south, paths are never west or north. So the longest possible path is one that reaches the bottom-right corner of the grid. The grid has r rows and c columns, so this longest path involves (r-1) down moves and (c-1) right moves, for a total of r + c - 2 moves.

There are two kinds of moves, east → and south ↓. 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 7.5 of the ADITA textbook. We loop through all binary numbers from 0 through $2^n - 1$, and interpret the bit at position k as the east/south step at index k.

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

Combining these ideas gives us a complete and clear algorithm.

```
crane_unloading_exhaustive(G):
   maxlen = r + c - 2
   best = None
   for len from 0 to maxlen inclusive:
      for bits from 0 to 2^len - 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:
                  if best is None or candidate reaches more cranes than best:
                        best = candidate
   return best
```

Let `n = max(r, c)`. Then `maxlen` $\in$ `O(n)`, the outermost for loop repeats `O(n)` times, the middle for loop over bits repeats $O(2^n)$ times, and the inner for loops repeat `O(n)` times, and the total run time of this algorithm is $O(n^2 2^n)$. This is a very slow algorithm.

The Dynamic Programming Algorithm

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

A[r][c] = the crane-maximizing path that starts at (0, 0) and ends at (r, c); or None if (r, c) is unreachable

Recall that in this problem, some cells are occupied by buildings and are therefore to be avoided by trucks.

The base case is the solution for A[0][0], which is the trivial path that starts and takes no subsequent steps.

A[0][0] = [start]

We can build a solution for a general case based on pre-existing shorter paths. Trucks can only drive east and south. So there are two ways a truck path could reach (i, j).

1. The truck path above (i, j) could add a southward step.
2. The truck path to the left of (i, j) could add an eastward step.

The algorithm should pick whichever of these two alternatives is optimal, which in this problem means whichever of the two candidate paths reach more cranes.

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 a building. Symmetrically, the from-west

path (2) only exists when we are not on the leftmost column (so when j>0) and when the cell left of (i, j) is not a building

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 building.

Altogether, the general solution is:

```
G[i][j] = None           if G[i][j]==X
G[i][j] = whichever of from_above and from_left is non-None and reaches
most cranes where
            from_above = None if i=0 or G[i-1][j]==X; or G[i-1][j] + [↓]
            otherwise
            from_left = None if j=0 or G[i][j-1]==X; or G[i][j-1] + [→]
            otherwise
```

The optimal solution is not required to end at (r-1, c-1). Indeed, the optimal path may end anywhere. So after the main dynamic programming loop, there is a post-processing step to find the path reaching most cranes.

Putting the parts together yields a complete dynamic programming algorithm.

```
crane_unloading_dyn_prog(G):
   A = new r×c matrix
   # base case
   A[0][0] = [start]
   # 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 = None
            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] = whichever of from_above and from_left is non-None and reaches
            more cranes; or None if both from_above and from_left are None
   # post-processing to find maximum-crane path
   best = A[0][0] # this path is always legal, but not necessarily optimal
   for i from 0 to r-1 inclusive:
      for j from 0 to c-1 inclusive:
            if A[i][j] is not None and A[i][j] reaches more cranes than best:
                  best = A[i][j]
   return best
```

The time complexity of this algorithm is dominated by the general-case loops. The outer loop repeats n times, the inner loop repeats n times, and creating each of from_above and from_left takes $O(n)$ time to copy paths, for a total of $O(n^3)$ time. While $O(n^3)$ 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 Instructions

Invitation links for the project:

https://classroom.github.com/a/2ElbGbzc

Implementation

You are provided with the following files.

1. `cranes_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. `cranes_types.cpp` is a C++ header that defines data types for the grids, paths, and related objects for the crane unloading problem. This code is complete; you should not modify this file.
3. `cranes_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`, `cranes_test.cpp`, `timer.hpp`, `rubrictest.hpp`, and `README.md` work the same way as in prior projects.

What to Do

First, add your group member names to `README.md`. Implement all the skeleton functions in the provided header file. Use the test program to check whether your code works.

When you go to commit files in git, make sure you are not checking any generated executable/object files into git. Use the *.gitignore* file to accomplish this.

Once you are confident that your algorithm implementations are correct, do the following:

1. Analyze your dynamic programming algorithm code mathematically to determine its big-O efficiency class, probably $O(n^2)$ or $O(n \log n)$.
2. Analyze your exhaustive optimization algorithm code mathematically to determine its big-O efficiency class, probably $O(2^n \cdot n)$.
3. Gather empirical timing data by running your implementations for various values of n.
4. Draw a scatter plot for each algorithm and fit line for your timing data. The instance size $n$should be on the horizontal axis and elapsed time should be on the vertical axis. Each plot should have a title; and each axis should have a label and units of measure.
5. Conclude whether or not your empirically-observed time efficiency data is consistent, or inconsistent, with your mathematically-derived big-O efficiency class for each algorithm.

Finally, produce a brief written project report ***in PDF format***. Your report should be submitted as a checked-in file in GitHub. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 2.
2. Two scatter plots meeting the requirements stated above.
3. Answers to the following questions, using complete sentences.
    a. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?
    b. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.
    c. Is this evidence consistent or inconsistent with **the** hypothesis ~~1~~? Justify your answer.
    ~~d. Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.~~

Grading Rubric

Your grade will consist 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 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: passes all the tests
2. Form - 11 points
    a. README.md complete: 3 points
    b. Style (whitespace, variable names, comments, etc.): 2 points

    c. Design (where appropriate, uses encapsulation, helper functions, data structures, etc.): 3 points
    d. Craftsmanship (no memory leaks, gross inefficiency, taboo coding practices, etc.): 3 points
3. Analysis - 16 points
        a. Report document presentation: 4 points
        b. Two scatter plots: 4 points each plot
        c. Question answers: 4 points

Deadline

The project deadline is Friday, December 17, 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.