HW 4
TCSS 343

Sean Hoyt, Winfield Brooks, Shenouda Massoud

March 8, 2016

1 Division of Labor

For our implementation of this assignment. We decided to split the labor as such:
Sean -recursive and bruteforce for first problem, and project write up.
Winfield - Dynamic solution for first problem, dynamic and brute force for challenge.
Shenouda - Building graphs, data files for testing.

2 Design for first problem

For our implementation, we thought to interpret the input cost array as a set of nodes and edges. Initially when the array is read in it creates n nodes, then for each entry in the cost array adds and edge between those particular nodes. This made for solving the minimum recursively cost much easier. Each of these nodes has a distance field and a visited Boolean flag, as well as a Node pointer, and an arrayList of edges.These elements help later when back traversal of the graph is necessary to find the cheapest cost.

2.1 Bruteforce approach

Our brute force generates all possible subsets of the available edges, or costs. It then checks to see if that particular path, begins at the source. In order to generate all possible subsets it will take exponential time and as such are algorithm is in $\theta\big((2^n\big)$. After generating all the values our algorithm then must compare each path's cost In our testing we only tested up to n=16, as the amount of time it took for large values, became slightly tedious to wait for.

2.2 Recursive approach

In our recursive approach, we thought of the different ways we could traverse the graph to find the shortest distance to the end.We initially set all nodes other than source at distance at ∞, to represent that there is no given path to current node. Since our graph is directed we thought using a recursive approach similar to Dijkstra's algorithm, would do the trick.It turns out upon writing this paper, I(Sean) had actually implemented the Bellman-Ford algorithm, without knowing it's existence. In this approach we visit each destination node in a given node's adjacency list, until we find the end, we then look at the cost of the current node, if the source (node's

cost) + (the edge weight to the destination) is less than the nodes current cost, then the cost of the node is updated. This is done for each level on the way back up until we have a minimal cost associated with each level. After this, a solution can be recovered in $\theta(n)$ time using our retrace node pointer. Testing on input sizes 100, 200, 400, 800 proved no problem for this algorithm, it was able to calculate the minimal distances in a much more timely more manner than the previous brute force approach.This algorithm runs in $\theta(|V|*|E|)$ where V is the number of vertices, and E is the number of edges, since V = n and E = $\sum_{i=1}^{n-1} i = \frac{n^2-n}{2}$
The overall time complexity is $\theta(n^3)$.

2.3 Dynamic approach

For the dynamic solution we initially wrote an algorithm that took in a 2D array that represented the posts and costs. In this implementation the dynamic programming table consisted of a single array that tracked the lowest cost up to each post and another array that kept track of the posts that were selected. This solution compared the cost of getting to each post to all possibilities and saved the lowest cost at cost[i] and the post used to get there in post[i]. When the final post is added the lowest cost to get to the nth post is saved in the last element of the cost array and the post used to get there was saved in the last element of the post array. After implementing the graph solution for the recursion and brute force we adapted the dynamic solution to work with a graph. Now instead of using an array as the dynamic table we used a graph composed of nodes and edges. As each node (post) is visited the cost of all paths to that node are compared. The node with the lowest path cost is then flagged to be traced. When the final node is reached we are able trace the lowest path cost by checking all nodes for those flagged to be traced taking O(n) time. To obtain the trace list at worst n comparisons are made n times. This gives the total runtime of $\theta(n^2 + n) \epsilon \theta(n^2)$

3 Challenge

3.1 Brute Force

3.1.1

F represents a Boolean matrix of size n x n such that F[i,j] is true if there is a stone at position (i,j)

SpecifiedSquare($F[1...n][1...n], intx, inty, ints$)
    For $i = x...x + s$:
        For $j = y...y + s$:
            If $i > n$ *or* $F[i][j] = true$:
                Return false.
    Return true.
End SpecifiedSquare

3.1.2

F represents a Boolean matrix of size n x n such that F[i,j] is true if there is a stone at position (i,j)

MaxSquareBruteForce(F[1...n][1...n])
    Let $max = 0$.
        For $i = 1...n$ do:
            For $j = 1...n$ do:
                For $k = 1...n$ do:
                    If $SpecifiedSquare(F[1...n][1...n], i, j, k) = trueandk > max$:
                        Let max = k.
                        Let Row = i.
                        Let Column = j.
    Print Max, Row, Column.
End MaxSquareBruteForce.

3.1.3 Asymptotic complexity

$SpecifiedSquare : \sum_{i=1}^{n-1} =1$ n·=d· $n^2 \in \theta\big((n^2)\big)$ where d is a constant

$$MaxSquareBruteForce : \sum_{i=1}^{n} \cdot \sum_{j=1}^{n} \cdot \sum_{k=1}^{n} dn^2 = dn^5 \epsilon\, \theta\big((n^5)\big)\big)$$

where d is a constant

3.2 Dynamic Programming

F represents a Boolean matrix of size n x n such that F[i,j] is true if there is a stone at position (i,j).

Max Square ($F[1...n][1...n]$)
Let $S[1...n][1..n]$ be matrix initialized to all zeros.
Let $max = 0$.
For $i = 1...n$ do:
$S[i][n] = F[i][n]$. //Set the right column in S to equal the right column in F
For $i = 1...n$ do:
S[n][i] = F[n][i]. //Set the bottom row in S to equal the bottom row in F
For $i = n - 1...1$ do:
For $j = n - 1...1$ do:
If F[i][j] is 1:
Let S[i][j] = $min(S[i + 1][j], S[i][j + 1], S[i + 1][j + 1]) + 1$.
If S[i][j]> max.
Let $max = S[i][j]$.
Let $Row = i$.
Let $Column = j$.
Else:
$LetS[i][j] = 0$.
Print Max, Row, Column.
End MaxSquare.

For this solution the dynamic table is represented by S. It is initialized as a table of n x n size with all values set to 0. The table will be filled from the bottom right corner to the top left corner. It is necessary to compare the values located in the bottom row and right column in the first iteration of the main loop. Therefore it is

necessary to set bottom row and right column of S to be equal to the bottom row and right column of F. To find the top right corner of the largest square possible each element of the matrix is first checked if it is a stone. If the element is not a stone it finds the minimum value of the elements right, below, and diagonal to the right and below and assigns itself that minimum value plus one. After the final element is assigned a value each element contains the largest square that can be represented with that element as the top right corner. To find the largest square compare all elements in the matrix and return the location and value the element holds.

This algorithm contains 2 loops of n iterations and one nested loop containing n x n iterations. Inside each loop a constant amount of work is done.

Function cost:

$$\sum_{i=1}^{n} c + \sum_{j=1}^{n} d + \sum_{k=1}^{n} \cdot \sum_{l=1}^{n} e = cn + dn + en^2 \epsilon \, \theta\big(n^2\big)$$

where c, d, e are constants.