CPSC 335-07 Project 2: Crane Problem

Group Members:

- Emma Jauregui (emmaj819@csu.fullerton.edu)
- Gregory Martinez (greg_2000@csu.fullerton.edu)
- Albert Nguyen (albertngn@csu.fullerton.edu)
- Matthew Tran (2020mtran@csu.fullerton.edu)

GitHub URL: https://github.com/2020mtran/Crane-Problem

Demo: https://youtu.be/4qvVvyvz2To

Exhaustive Pseudocode:

```
best;
for steps.size = 0 to steps <= max steps:
//iterates over the diff sizes of paths
    for i.size = 0 to i <= 2^(steps) - 1:
    //iterates over indices of path
        path candidate;
        boolean valid = true;

        for size.j = 0 to j <= steps - 1:
        //iterates over diff sizes inside path
            step direction;
            set size.k to (i shifted right by j bits) bitwise and 1.
            //make var only 1 or 0
            if k = 0:
                step direction is  east
            else if k = 1:
                step direction is south
            if path candidate is valid:
            //verification of direction
                add step
            else:
                valid = false
        If valid:
            if candidate total crane > best candidate total cranes:
```

```
                        best = candidate
        return best
```

Time Analysis:

The paths are traversed in an exhaustive manner in the way that all possible paths are generated in order to find the most efficient pathway. All possible sized paths are iterated through in order to fulfill the exhaustive method. The inner if statements can be considered as O(1).The outer loop will run m+1 times (0 to m, m being the max steps). The inner for loop will run 2^n - 1 (n being the number of steps) and the innermost loop will run N times (n -1). The dominant factor in the result of (m+1)(2^n)(n) is 2^n, so in Big O notation, the time complexity is O(2^n).

Dynamic Pseudocode:

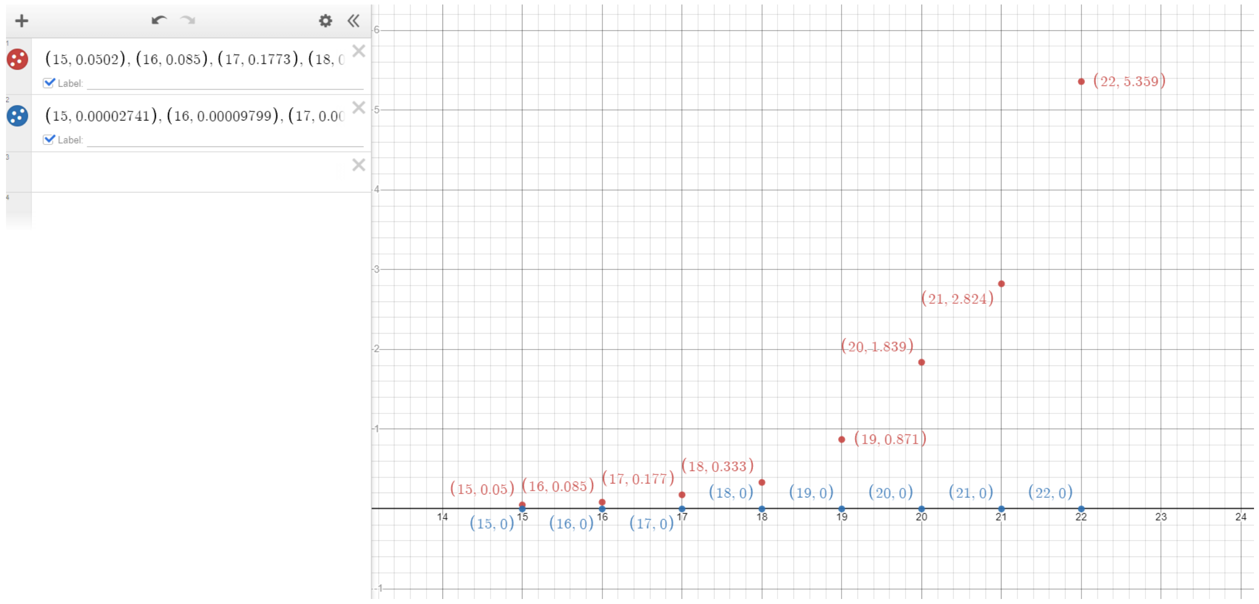
```
//iterate through dynamic 2d-matrix A[r][c]
for r=0 to max_cell_rows
    for c=0 to max_cell_collumns
        if cell == building
            reset to beginning;
        endif
        //check conditions for adding in matrix
    If a south move has a value
        if south cell value is valid
            add south move
    if a east move has a value
        if east move value is valid
            add east move
    //compare moves
    If south and east have a move
        If south > east
            Current cell -> south move
        else if east.move has value
                Current cell -> east move
        else if south.move has value
                Current cell -> south move
```

Time Analysis:

The for loops iterate through the dynamic matrix and insert the value of cells in the matrix to be compared and point to path with maximum cranes possible. As each if statement is a comparison or declaration, it can be said that the if statements would be compared to a O(1) time complexity and the outer double for loops for the 2D dynamic algorithm would make the pseudocode O(row x column) or O(n^2).

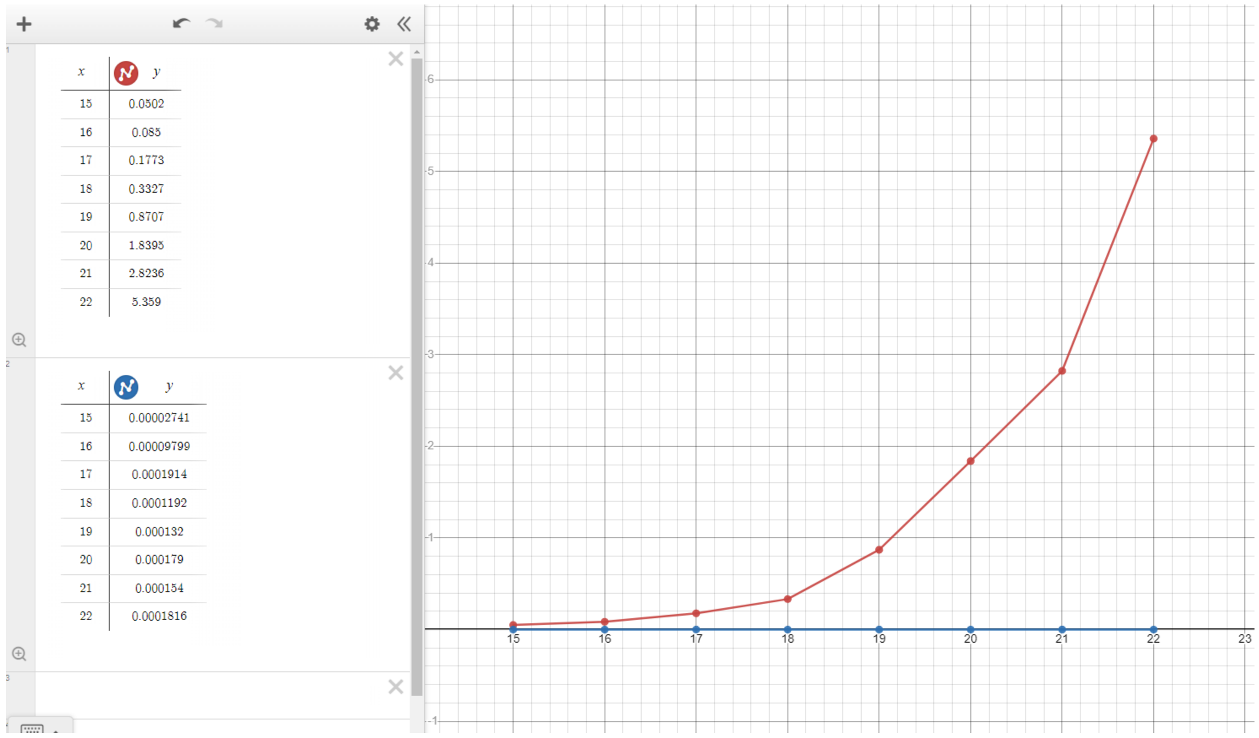
Graphs: input size vs runtime

Red is exhaustive optimization, blue is dynamic programming



(15, 0.0502), (16, 0.085), (17, 0.1773), (18, 0

(15, 0.00002741), (16, 0.00009799), (17, 0.00



x	y
15	0.0502
16	0.085
17	0.1773
18	0.3327
19	0.8707
20	1.8395
21	2.8236
22	5.359

x	y
15	0.00002741
16	0.00009799
17	0.0001914
18	0.0001192
19	0.000132
20	0.000179
21	0.000154
22	0.0001816

Questions:

1. There is a very significant difference in speed between the exhaustive optimization and dynamic programming algorithms. The dynamic programming algorithm is the faster algorithm by far, with time complexity of O(n^2) compared to the exhaustive optimization algorithm's much worse time complexity of O(2^n). This is unsurprising since the dynamic programming method caches the best possible path in an array which is faster to traverse than the exhaustive optimization algorithm's method of searching all possible paths to find the optimal solution.
2. The empirical analysis is consistent with the mathematical analysis. The exhaustive optimization algorithm can be seen in the graphs as growing in runtime at an exponential rate when compared to the runtime of the dynamic programming algorithm, which grows more like a polynomial. This remains consistent with the time complexity analyses that conclude that the exhaustive optimization algorithm runs in an exponential-time complexity of O(2^n) whereas the dynamic programming algorithm runs in a polynomial-time complexity of O(n^2)
3. The evidence is consistent with hypothesis 1: polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive optimization algorithms that solve the same problem. For example, the graph of the input size versus runtime for both algorithms reveals how slow the exhaustive optimization algorithm becomes at larger input sizes. Given the input 22, the exhaustive optimization algorithm has a runtime of 5.359 seconds whereas the dynamic programming algorithm has a significantly shorter runtime of 0.0001816 seconds for the same input size. This evidence shows that dynamic programming algorithms are more efficient than exhaustive optimization algorithms.
4. The evidence is inconsistent in consideration of hypothesis 2, the reverse of hypothesis 1. Hypothesis 2 would state that polynomial-time dynamic programming algorithms are less efficient when compared to exponential-time exhaustive search algorithms solving the same problem. Even when the lower-end of the graph is considered, the dynamic programming algorithm is still much faster than the exhaustive optimization algorithm is. For a given input of 15, the exhaustive optimization algorithm has a runtime of 0.0502 seconds while the dynamic programming algorithm has a runtime of 0.00002741 seconds. The evidence clearly shows that the dynamic programming algorithm is more efficient than the exhaustive optimization algorithm is.