Project 2: Dynamic vs. Exhaustive - Crane Unloading Problem

Authors:
Victor Nguyen - Victor326@csu.fullerton.edu - CWID: 885992636
Justin Nguyen - justindnguyen03@csu.fullerton.edu - CWID: 886407279

Exhaustive Algorithm Solution Pseudocode:

Pseudocode.txt

```
Function crane_unloading_exhaustive(setting):
    // Ensure the grid is not empty by checking if greater than 0
    assert that setting has at least one row and one column

    // Compute maximum path length
    max_steps = number of rows + number of columns - 2
    // Ensure the maximum steps are less than 64
    assert that max_steps < 64

    // Initialize the best path
    best = initial path using setting

    // Iterate through all possible step amounts
    for each possible number of steps from 0 to max_steps inclusive:

        // Iterate through all possible step patterns
        for each possible step pattern j from 0 to 2^steps - 1 inclusive:

            // Create a candidate path
            candidate = initial path using setting

            // Assume the path is valid until proven otherwise
            isValid = true

            // Iterate through all possible bit positions
            for each possible bit position k from 0 to steps - 1 inclusive: ---> step times

                // Get the current bit
                bits = get bit at position k in step pattern j

                // Determine direction based on the bit value
                if bits = 1:
                    direction = east
                else:
                    direction = south

                // Check if the candidate path is valid in the direction
                if candidate path is valid in direction:
                    // Add the direction to the candidate path
                    add direction to candidate path
                else:
                    // If the path is not valid, mark it as invalid and break the loop
                    isValid = false and break the loop

            // If the path is valid and has more cranes than the best path, update the best path
            if isValid and candidate has more cranes than best:
                best = candidate

    // Return the best path
    return best
```

Time Analysis for Exhaustive:

Pseudocode.txt

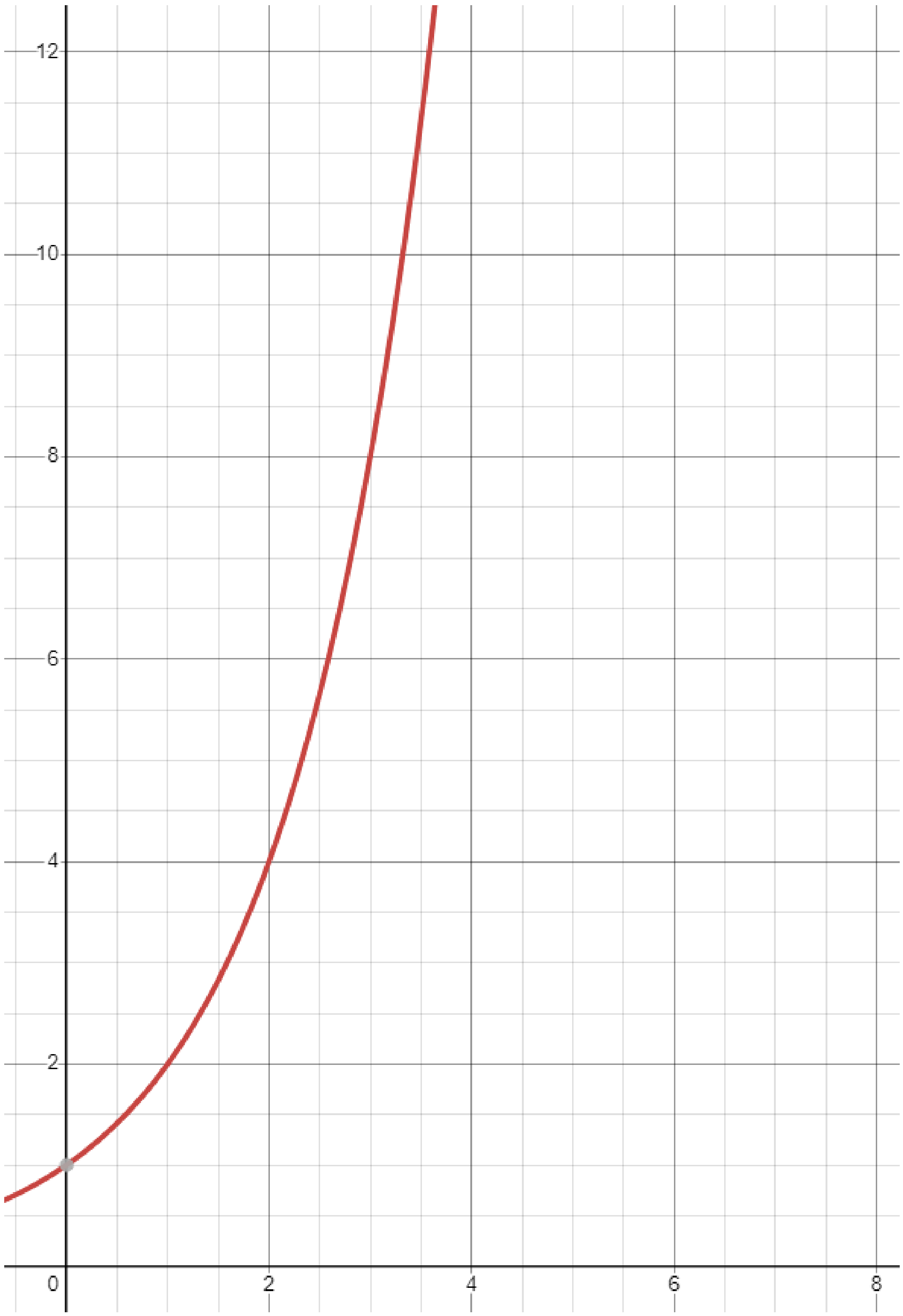
```
Function crane_unloading_exhaustive(setting):
    // Ensure the grid is not empty by checking if greater than 0
    assert that setting has at least one row and one column ---> 1 tu for rows and 1 tu for columns

    // Compute maximum path length
    max_steps = number of rows + number of columns - 2 ---> 3 tu

    // Ensure the maximum steps are less than 64
    assert that max_steps < 64 ---> 1 tu

    // Initialize the best path
    best = initial path using setting ---> 1 tu

    // Iterate through all possible step amounts
    for each possible number of steps from 0 to max_steps inclusive: ---> max_steps + 1 times

        // Iterate through all possible step patterns
        for each possible step pattern j from 0 to 2^steps - 1 inclusive: ---> 2^step times

            // Create a candidate path
            candidate = initial path using setting ---> 1 tu

            // Assume the path is valid until proven otherwise
            isValid = true ---> 1 tu

            // Iterate through all possible bit positions
            for each possible bit position k from 0 to steps - 1 inclusive: ---> step times

                // Get the current bit
                bits = get bit at position k in step pattern j ---> 1 tu

                // Determine direction based on the bit value
                if bits = 1: ---> 1 tu
                    direction = east ---> 1 tu
                else:
                    direction = south ---> 1 tu

                // Check if the candidate path is valid in the direction
                if candidate path is valid in direction: ---> 1 tu
                    // Add the direction to the candidate path
                    add direction to candidate path ---> 1 tu
                else:
                    // If the path is not valid, mark it as invalid and break the loop
                    isValid = false and break the loop ---> 1 tu

            // If the path is valid and has more cranes than the best path, update the best path
            if isValid and candidate has more cranes than best:
                best = candidate ---> 1 tu

    // Return the best path
    return best
```

Time Complexity:
Max_steps + 1 = n * m times
Steps = n + m
= 1 + 1 + 3 + 1 + 1 + (n*m) * (2^(n+m)) * (1 + 1 + ((n+m) * (3 + 1 + max(1,1) + 1 + max(1,1)) + 1))
= 7 + (n+m) * (2^(n+m)) * (2 + (n+m) * (8))
= 7 + (n+m) * (2^n+m) * (2 + 8n+8m)
= 7 + (n+m) * (2^n+m) * (8n+8m + 2)
= 7 + nm*2^(n+m+1)* + n^2m*2^(m+n+3) + nm^2*2^(n+m+3)

$$7 + 2^{m+n+1} \cdot mn + 2^{m+n+3} \cdot mn^2 + 2^{m+n+3} \cdot m^2 n$$

The time complexity of the code above is O(2^(n+m)).
Therefore, the time complexity of the Exhaustive Algorithm will belong to the **exponential time complexity of O(2^n).**

Graph for Time vs. Input Size for Exhaustive:



Dynamic Algorithm Solution Pseudocode:

Pseudocode.txt

```
Function crane_unloading_dyn_prog(setting):
    // Ensure the grid is not empty
    assert that setting has at least one row and one column

    // Initialize a 2D array for dynamic programming
    A = empty 2D array of optional paths with size rows x columns

    // Start from the first cell
    A[0][0] = initial path using setting

    // Ensure the first cell has a value
    assert that A[0][0] has a value

    // Iterate through each cell in the grid
    for each row and column in setting:

        // If the cell is a building, skip it
        if cell at row, column is a building:
            reset A[row][column] and continue

        // Initialize paths from above and from left as null
        from_above = null
        from_left = null

        // Get the path from the cell above, if it exists and is valid
        if row > 0 and A[row - 1][column] exists:
            from_above = clone of path at A[row - 1][column]
            if a step south from above is valid:
                add step south to from_above

        // Get the path from the cell to the left, if it exists and is valid
        if column > 0 and A[row][column - 1] exists:
            from_left = clone of path at A[row][column - 1]
            if a step east from left is valid:
                add step east to from_left

        // If both paths exist, choose the one with more cranes
        if both from_above and from_left exist:
            if from_above has more cranes than from_left:
                A[row][column] = from_above
            else:
                A[row][column] = from_left

        // If only the path from the left exists, choose it
        else if only from_left exists:
            A[row][column] = from_left

        // If only the path from above exists, choose it
        else if only from_above exists:
            A[row][column] = from_above

    // Initialize the best path as the first cell
    best = reference to A[0][0]

    // Iterate through each cell in the grid to find the path with the most cranes
    for each row and column in setting:
        if A[row][column] exists and has more cranes than best:
            best = reference to A[row][column]

    // Ensure the best path has a value
    assert that best has a value

    // Return the best path
    return dereference best
```

Time Analysis for Dynamic:

Pseudocode.txt

```
Function crane_unloading_dyn_prog(setting):
    // Ensure the grid is not empty by checking if greater 0
    assert that setting has at least one row and one column ---> 1 tu for row and 1 tu for column

    // Initialize a 2D array for dynamic programming
    A = empty 2D array of optional paths with size rows x columns ---> 1 tu

    // Start from the first cell
    A[0][0] = initial path using setting ---> 1 tu

    // Ensure the first cell has a value
    assert that A[0][0] has a value ---> 1 tu

    // Iterate through each cell in the grid
    for each row and column in setting: ---> n+1 tu for the row iteration and m+1 tu for column iteration

        // If the cell is a building, skip it
        if cell at row, column is a building: ---> 1 tu
            reset A[row][column] and continue ---> 1 tu

        // Initialize paths from above and from left as null
        from_above = null ---> 1 tu
        from_left = null ---> 1 tu

        // Get the path from the cell above, if it exists and is valid
        if row > 0 and A[row - 1][column] exists: ---> 3 tu
            from_above = clone of path at A[row - 1][column] ---> 2 tu
            if a step south from above is valid: ---> 1 tu
                add step south to from_above ---> 1 tu

        // Get the path from the cell to the left, if it exists and is valid
        if column > 0 and A[row][column - 1] exists: ---> 3 tu
            from_left = clone of path at A[row][column - 1] ---> 2 tu
            if a step east from left is valid: ---> 1 tu
                add step east to from_left ---> 1 tu

        // If both paths exist, choose the one with more cranes
        if both from_above and from_left exist: ---> 1 tu
            if from_above has more cranes than from_left: ---> 1 tu
                A[row][column] = from_above ---> 1 tu
            else:
                A[row][column] = from_left ---> 1 tu

        // If only the path from the left exists, choose it
        else if only from_left exists: ---> tu
            A[row][column] = from_left ---> 1 tu

        // If only the path from above exists, choose it
        else if only from_above exists: ---> 1 tu
            A[row][column] = from_above ---> 1 tu

    // Initialize the best path as the first cell
    best = reference to A[0][0] ---> 1 tu

    // Iterate through each cell in the grid to find the path with the most cranes
    for each row and column in setting: ---> n + 1 times for row and m + 1 for column
        if A[row][column] exists and has more cranes than best: ---> 2 tu
            best = reference to A[row][column] ---> 1 tu

    // Ensure the best path has a value
    assert that best has a value ---> 1 tu

    // Return the best path
    return dereference best
```

Time Complexity:

= 1 + 1 + 1 + 1 + (n+1)(m+1)(1 + 1 + 1 + 3 + 3 + 1 + max(1, (2 + 1 + 1), (2 + 1 + 1), (1 + max(1,1)), (1 + 1), (1 + 1)) + 1 + (n+1)(m+1)(2 + 1) + 1

= 4 + (n+1)(m+1)(10 + max(1, 4, 4, 2, 2, 2) + 1 + (n+1)(m+1)(3) + 1

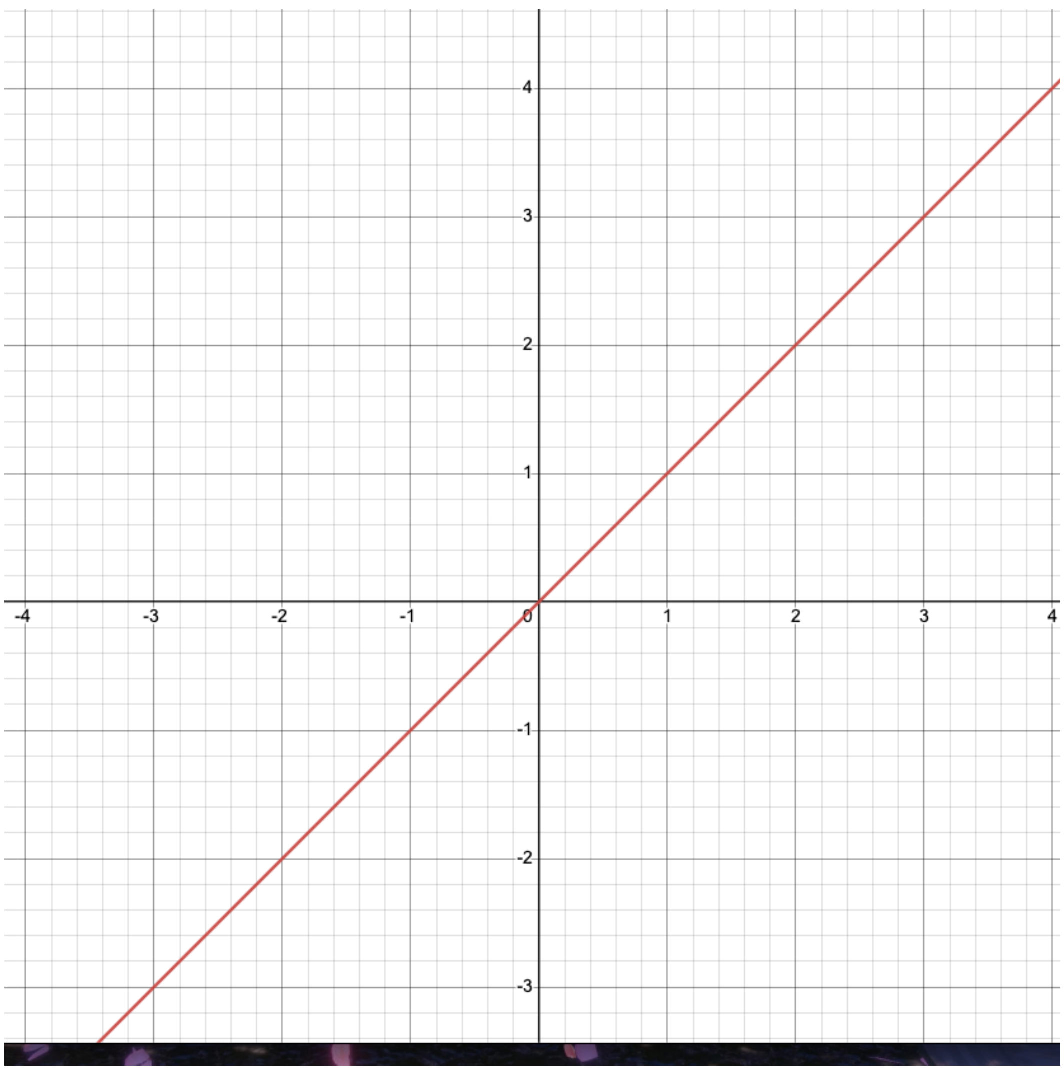
= 6 + (n+1)(m+1)(10 + 4) + (n+1)(m+1)(3)
= 6 + (nm + n + m + 1)(14) + (nm + n + m + 1)(3)
= 6 + 14nm + 14n + 14m + 14 + 3nm + 3n + 3m + 3
= 17nm + 17m + 17n + 23

$$17mn + 17m + 17n + 23$$

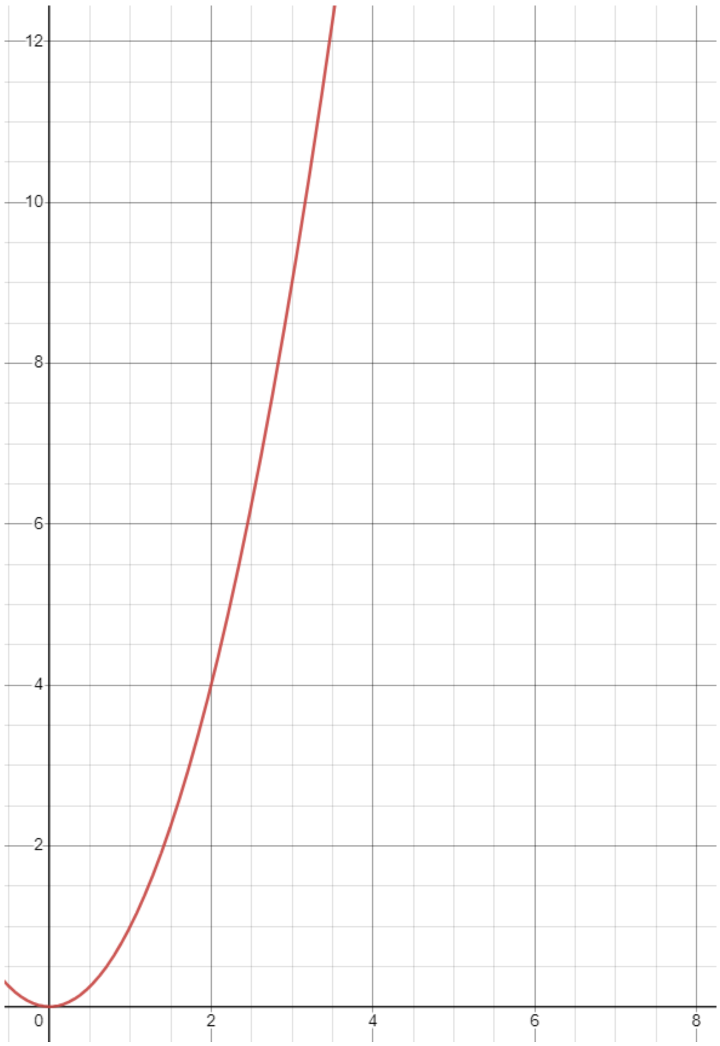
The time complexity of the code above is O(nm).
Therefore, the time complexity of the Exhaustive Algorithm will belong to the **polynomial time complexity of O(n^2).**

Graph for Time vs. Input Size for Dynamic:

(in general, notation is n*m for our dynamic like graph below or if grid is rectangular)



(If grid is square or n and m are same value/size, notation is n^2 like graph below)



Algorithm Run Time Observation Questions:

1. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?

There is a noticeable difference between the two algorithms above between exhaustive and dynamic programming. The faster algorithm is dynamic programming because exhaustive search has an exponential time complexity of O(2^(n+m)). Dynamic programming doesn't have an exponential time complexity but a polynomial time complexity that is of n*m instead. How much faster is dependent on the size, but generally, the greater the size or input, dynamic programming will be much faster. This result surprised us a bit but not much, as we knew that the exhaustive search would take longer. Still, we did not expect dynamic programming to be significantly faster, especially when the input size becomes larger. One thing to note, though is that if m and n are the same value or size, this will make the time complexity O(n^2) but generally is O(n*m) due to n and m having the possibility to be different value or size to be rectangular grid but even with n^2 time complexity, this will still make dynamic programming faster, especially on bigger grid or data/values but on smaller ones or smaller inputs, this gives exhaustive search algorithm a better chance at competing with dynamic programming.

2. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.

The empirical analysis IS consistent with our mathematical analyses. Base on the analysis for our exhaustive search algorithm, we can infer that it has an exponential time complexity of 2^n+m, which means that as the grid size increases (n and m), then the execution time will increase rapidly. This shows with our mathematical analysis in that it is an exponential time complexity as it belongs in the O(2^n+m) or basically belongs to the O(2^n) as you can see that our math comes out to the answer of 7 + 2^m+n+1 * mn+ 2^m+n+3 * mn^2 + 2^m+n+3 * m(^2) * n which reduces to 2^m+n or 2^n time complexity notation mathematically. For dynamic programming, we know that it has a polynomial time complexity base on the code or pseudocode, which means its execution time will increase but at a slower or more likely linear rate compared to the exhaustive search algorithm as the size of the grid (n and m) increases. From the mathematical part, we are able to infer that it does also belong in the polynomial time complexity of O(n*m), but we do want to point out as well that it COULD be O(n^2) as well if n and m are the same value or size making it a square grid. But in the end, or in general, we are able to see from the answers from the math equation that the time complexity of dynamic programming of 17mn + 17m + 17n + 23 reduces to O(n*m).

3. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.

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

The evidence is consistent with hypothesis 1 because polynomial-time algorithms are generally more efficient than exponential-time algorithms. Looking at the graphs, the dynamic algorithm can take more instances in less time. The exhaustive algorithm also takes more time with fewer instances. Therefore, we can conclude that polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive algorithms.