

# **CPSC 335 - Algorithm Engineering**

## **Project 4: Dynamic versus Exhaustive**

**Malka Ariel Lazerson**

**Fall 2021**

**Instructor: Doina Bein**

### ***Table of Contents***

Names, CSUF Email, and Intent	2
ReadMe.md Screenshot	3
Code Compilation and Successful Execution	3
Empirical Data	4
Scatter Plots	5
Mathematical Analysis	6
Question Answers	11

## **Names, CSUF Email, and Intent**

Name: Malka Ariel Lazerson

CSUF Email: mlazerson@csu.fullerton.edu

Intent: This document is intended to be one part of a submission for Project 4: Dynamic vs. Exhaustive. This document contains....

1. Names, CSUF-supplied email address, and an indication that the submission is for project 4
2. Proof of code compilation and successful execution
3. Empirical Data
4. Mathematical analysis and the big-O efficiency class for both algorithms
3. Two scatter plots
4. Answers to the following questions, using complete sentences.

Questions....

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

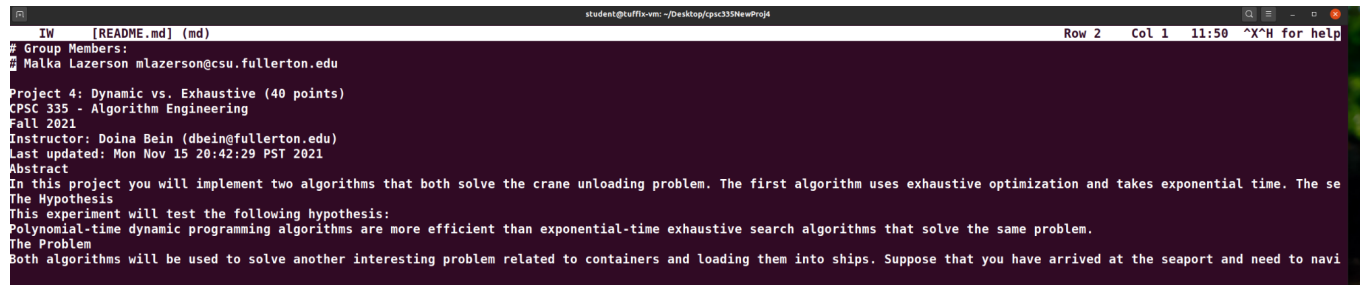
Are your empirical analyses consistent with your mathematical analyses? Justify your answer.

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

Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.

Code and more files will be placed in the github for the instructor to review.

## ReadMe.md Screenshot



```
student@tuffcm: ~/Desktop/cisc335NewProj4
IW [README.md] (md) Row 2 Col 1 11:50 ^X^H for help
# Group Members:
# Malka Lazerson mlazerson@csu.fullerton.edu

Project 4: Dynamic vs. Exhaustive (40 points)
CISC 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 se
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 navi
```

## Code Compilation and Execution

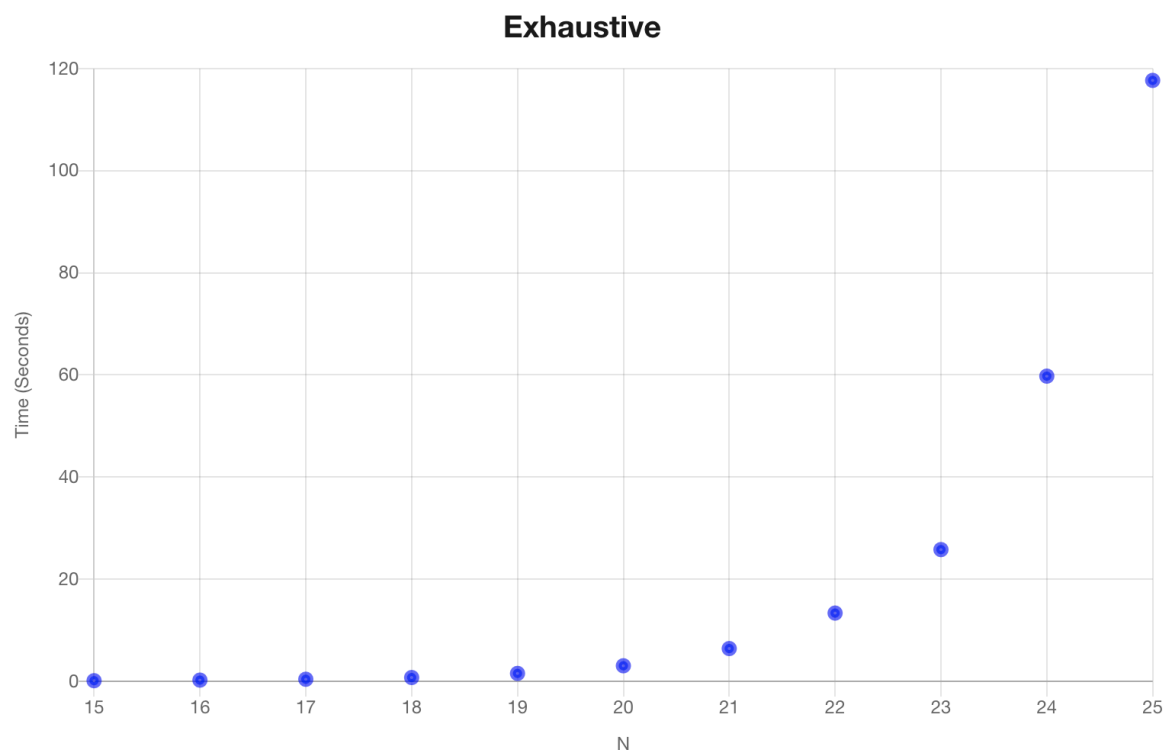
```
bash-3.2$ make
g++ -std=c++17 -Wall cranes_test.cpp -o cranes_test
./cranes_test
exhaustive optimization - simple cases: passed, score 4/4
exhaustive optimization - maze: passed, score 1/1
dynamic programming - simple cases: passed, score 4/4
dynamic programming - maze: passed, score 1/1
dynamic programming - random instances:
passed, score 1/1
stress test: passed, score 2/2
TOTAL SCORE = 13 / 13

g++ -std=c++17 -Wall cranes_timing.cpp -o cranes_timing
bash-3.2$
```

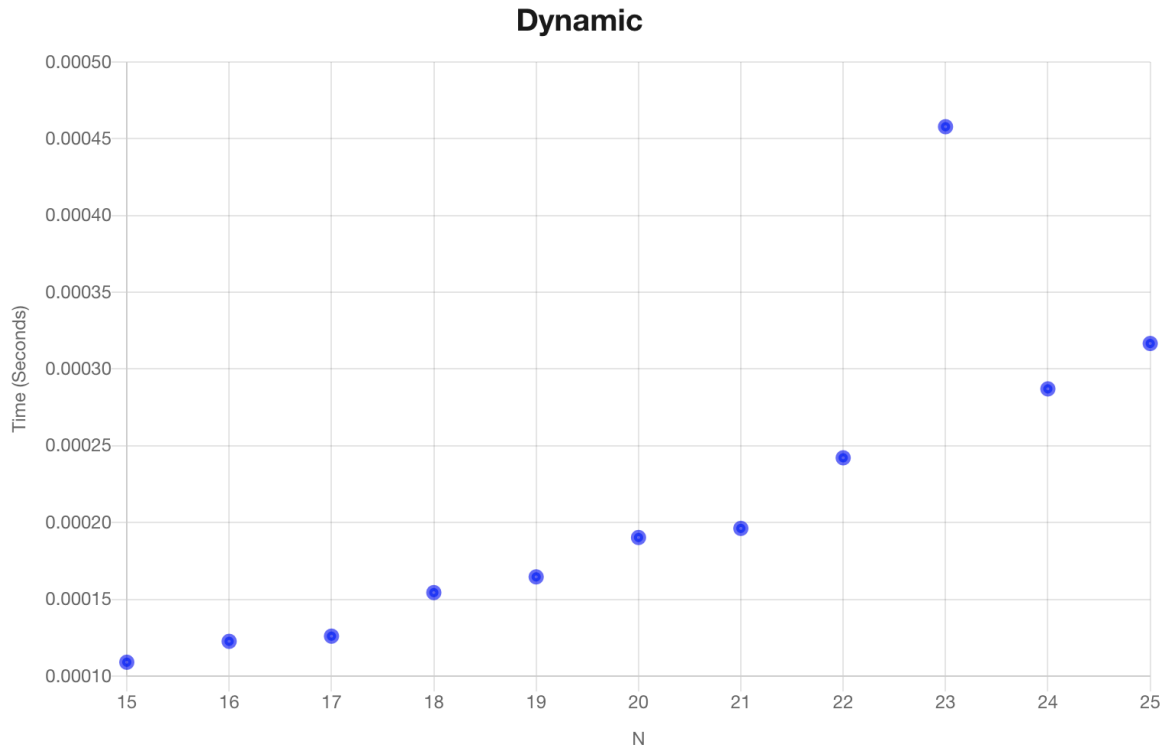
## **Empirical Data**

	Exhaustive	Dynamic
n	Time (seconds)	Time(seconds)
15	0.10114	0.00010899
16	0.214427	0.000122586
17	0.384004	0.000125968
18	0.722881	0.000154352
19	1.52655	0.000164576
20	3.02628	0.000190243
21	6.39913	0.000196174
22	13.3473	0.000242154
23	25.7838	0.000457848
24	59.7593	0.000287056
25	117.693	0.000316629

**Scatter Plots**



*Exhaustive Scatter Plot*



Dynamic Scatter Plot

## Mathematical Analysis

This section includes pseudocode, the step count, and Big O Time Complexity

### Exhaustive

Step Count.....

```
path crane_unloading_exhaustive(const grid& setting) {

    // grid must be non-empty.
    assert(setting.rows() > 0); // 1
    assert(setting.columns() > 0); // 1

    // Compute maximum path length, and check that it is legal.
    const size_t max_steps = setting.rows() + setting.columns() - 2; // 1
    assert(max_steps < 64); // 1

    path best(setting); // 1
```

```

for (size_t steps = 1; steps <= max_steps; ++steps) { // n
    uint64_t mask = uint64_t(1) << steps; // 1
    for (uint64_t bits = 0; bits < mask; ++bits) { // n^2

        path candidate(setting); // 1
        bool valid = true; // 1

        for (size_t i = 0; i < steps; ++i ) { // n

            int bit; // 1
            bit = (bits >> i) & 1; // 2

            if (bit == 1) { // 1

                if( candidate.is_step_valid(STEP_DIRECTION_EAST) ) { // 1
                    candidate.add_step(STEP_DIRECTION_EAST); // 1
                }

            }
            else {

                if (candidate.is_step_valid(STEP_DIRECTION_SOUTH)) { // 1
                    candidate.add_step(STEP_DIRECTION_SOUTH); // 1
                }

            }
        }

        if (valid && (candidate.total_cranes() > best.total_cranes())) { // 1
            best = candidate; // 1
        }
    }
}

return best; // 1

```

Calculation.....

$$5 + n * ( 2^n + ( 2 + (n + 10) ) ) + 1 =$$

$$20 + n * n * 2^n =$$

$$20 + n^2 * 2^n$$

***Exhaustive is in  $O(n^2 * 2^n)$  which is very slow***

Proof.....

Let  $t = 20 + n^2 * 2^n$  be in  $O(n^2 * 2^n)$

$$20 + 1 + 2 = 23, \text{ let } c = 23, \text{ let } n = 1$$

$$\text{Let } 20 + n^2 * 2^n \geq n * (n^2 * 2^n)$$

$$20 * 1 * 2 = 40$$

$$1 * 1 * 2 = 2$$

$$40 \geq 2 \text{ and } \geq 0$$

**Exhaustive algorithm is  $O(n^2 * 2^n)$  Time Complexity**

## Dynamic

Step Count.....

```
path crane_unloading_dyn_prog(const grid& setting) {

    // grid must be non-empty.
    assert(setting.rows() > 0); // 1
    assert(setting.columns() > 0); // 1

    using cell_type = std::optional<path>; // 1

    std::vector<std::vector<cell_type>>> A(setting.rows(),
                                         std::vector<cell_type>(setting.columns())); // 1

    A[0][0] = path(setting); // 1
    assert(A[0][0].has_value()); // 1
```



```

for (coordinate r = 0; r < setting.rows(); ++r) { // n
    for (coordinate c = 0; c < setting.columns(); ++c) { // n

        if (setting.get(r, c) == CELL_BUILDING) { // 1
            A[r][c].reset(); // 1
            Continue; // 1
        }

        //begin if (setting.get(r, c) != CELL_BUILDING) case.....
        //if (setting.get(r, c) != CELL_BUILDING) { // 1

            // from_above = from_left = None
            std::optional<path> from_above; // 1
            std::optional<path> from_left; // 1

            // if r greater than 0 and ! None
            if (r > 0 && A[r-1][c].has_value()) { // 1

                from_above = A[r-1][c].value(); // n

                if(from_above->is_step_valid(STEP_DIRECTION_SOUTH) ) { // 1
                    from_above->add_step(STEP_DIRECTION_SOUTH); // 1
                }
            }

            // if c greater than 0 and ! None
            if (c > 0 && A[r][c-1].has_value()) { // 1

                from_left = A[r][c-1].value(); // n

                if(from_left->is_step_valid(STEP_DIRECTION_EAST) ) { // 1
                    from_left->add_step(STEP_DIRECTION_EAST); // 1
                }
            }

            // if from_above and from_left ! None
            if(from_above.has_value() && from_left.has_value()) { // 1

```

```

        if(from_above->total_cranes() > from_left->total_cranes()) { // 1
            A[r][c] = from_above; // 1
        }
        else
        {
            A[r][c] = from_left; // 1
        }
    }
}

// if from_above is non-None
else if(from_above.has_value()) { // 1
    A[r][c] = from_above; // 1
}
// if from_left is non-None
else if (from_left.has_value()){ // 1
    A[r][c] = from_left; // 1
}

}
}

cell_type* best = &(A[0][0]); // 1
assert(best->has_value()); // 1
for (coordinate r = 0; r < setting.rows(); ++r) { // n
    for (coordinate c = 0; c < setting.columns(); ++c) { // n
        if (A[r][c].has_value() && A[r][c]->total_cranes() > (*best)->total_cranes()) { // 1
            best = &(A[r][c]); // 1
        }
    }
}

assert(best->has_value()); // 1
// std::cout << "total cranes" << (**best).total_cranes() << std::endl;

return **best; // 1
}

```

Calculation.....

$$6 + n * n + (7 + n + 3 + n + 2 + n + 2 + 8) + (2 + n * n + 2) + 2 =$$

$$6 + n^2 + 3n + 22 + n^2 + 4 + 2 =$$

$$34 + n^2 + n^2 + 3n =$$

$$34 + n^4 + 3n$$

***Dynamic is in  $O(n^4)$  which is slow, but polynomial, so much faster than exhaustive***

Proof.....

$$\text{Lim as } n \rightarrow \text{infinity } 34 + n^4 + 3n$$

$$\text{Lim as } n \rightarrow \text{infinity } 34 + n^4 + 3n / n^4$$

$$\text{Lim as } n \rightarrow \text{infinity } 34/n^4 + n^4 / n^4 + 3n / n^4$$

$$34/n^4 = 0$$

$$n^4 / n^4 = 1$$

$$3n / n^4 = 0$$

$$0 + 0 + 1 = 1$$

$1 \geq 0$ , so our proof is successful

***Dynamic is in  $O(n^4)$  Time Complexity***

## **Questions**

*Recall our hypotheses....*

*Hypothesis 1 from Project 2....*

1. *Exhaustive search and exhaustive optimization algorithms are feasible to implement, and produce correct outputs.*

2. *Algorithms with exponential running times are extremely slow, probably too slow to be of practical use.*

Hypothesis 2 for this project (Project 4)

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

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 large difference between the two, where exhaustive is much slower than dynamic. The mathematical analysis and time complexity back up this conclusion. Exhaustive is  $O(n^2 * 2^n)$  while dynamic is faster and polynomial at  $O(n^4)$ .

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

Yes, the timing data for exhaustive is much slower than dynamic. Exhaustive much slower than dynamic judging by the empirical data (time in seconds per run).

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

It has been proven that the exhaustive algorithm is feasible to implement and does produce the correct output. But given the empirical data and mathematical analysis, we also know exhaustive algorithms are exponential, very slow, and impractical. We have proven the dynamic algorithm is faster than the exhaustive one.

Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.

Mathematical analysis has proven the exhaustive algorithm is exponential, which is slower than the dynamic polynomial algorithm. Therefore, dynamic is more efficient, which is consistent with our hypothesis that dynamic is a faster way to solve the same problem.