Pseudocode for Exhaustive Algorithm:

```
Def Exhaustive_Optimization_Funct ( init grid by reference)
{
    assert the size of grid rows.
    assert the size of grid columns.
    compute the grid size and maximum path length.
    assert the grid size is legal (<64 units).
    init best path.
    for each element in step do:
        init Pos_Int //(0 for south, and 1 for east)
        int MyBit
        for each element in MyBit do:
            init a candidate path
            init bool valid = true // to check if a path is valid or not.
            let i < number of steps
            for each step in candidate:
                init size_t bit
                perform bitwise/ Bit-shift by a value of i
                if bit = 1
                    if stepping east is valid
                        step east
                    else valid = false
                else
                    if stepping south is valid
                        step south
                    else valid = false
        if candidate > best
            best = candidate
            return best.
}
```

Time Analysis:

```
path crane_unloading_exhaustive(const grid& setting)
{
  assert(setting.rows() > 0);
  assert(setting.columns() > 0);
  const size_t max_steps = setting.rows() + setting.columns() - 2;
  assert(max_steps < 64);
  path best(setting);
  for (size_t steps = 1; steps <= max_steps; ++steps)
  {
    uint64_t Pos_Int = uint64_t(1) << steps;
    for (uint64_t MyBit = 0; MyBit < Pos_Int; ++MyBit)
    {
      path candidate(setting);
      bool status = true;
      for (size_t i = 0; i < steps; ++i)
      {
        size_t bit = (MyBit >> i) & 1;
        if (bit == 1)
        {
          if (candidate.is_step_valid(STEP_DIRECTION_EAST))
          {
            candidate.add_step(STEP_DIRECTION_EAST);
          }
          else status = false;
        }
        else
        {
          if (candidate.is_step_valid(STEP_DIRECTION_SOUTH))
          {
            candidate.add_step(STEP_DIRECTION_SOUTH);
          }
          else status = false;
        }
      }
      if (status && (candidate.total_cranes() > best.total_cranes()))
      {
        best = candidate;
      }
    }
  }
  return best;
}
```

<u>Step Count</u>

1

1

3

1

SC_a

2

SC_b

1

SC_c

3

$1 + max(2,2) = 3$

2

$1 + max(1,1)$

1

1

2

$1 + max(1,1)$

1

1

$2 + max(1,0)$

$$SC_{Total} = 1 + 1 + 3 + 1 + SC_A \left[2 + SC_B \left(1 + SC_C \cdot SC_{for} \right) \right]$$

$$SC_{for\ loop} = 6$$

$$SC_{Total} = 6 + SC_A \left[2 + SC_B \left(1 + \sum_{C=0}^{n} 6 \right) \right]$$

$$= 6 + SC_A \left[2 + SC_B \left(1 + 6n \right) \right]$$

$$= 6 + SC_A \left[2 + \sum_{B=0}^{n} 1 + \sum_{B=0}^{n} 6n \right]$$

$$= 6 + \sum_{A=0}^{n} \left[2 + n + 6n^2 \right]$$

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

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

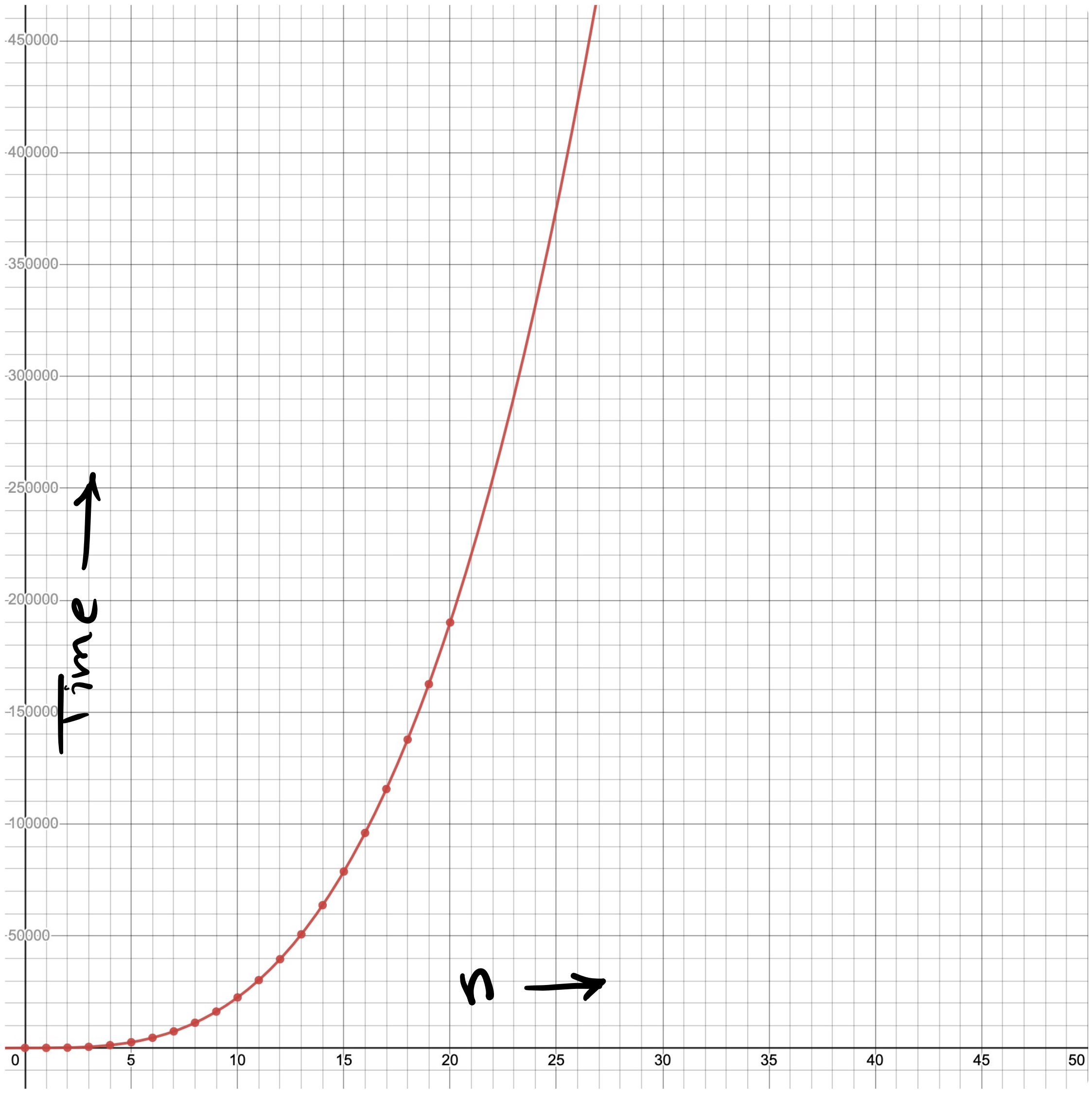
Time Complexity: $O(6n^3 + n^2 + 2n - 6)$

$= O(n^3)$

$\therefore$ Time Complexity $= O(25n^3 - 25n^2 + 2n)$

$= O(n^3)$

Graph :



Pseudocode for Dynamic Programming:

Def Dynamic_Crane (grid "Setting"):	Step Count
assert row_count > 0	**1**
assert col_count > 0	**1**
Cell_type = path	**1**
A[0][0]= path [setting]	**1**
for each row in rows() do:	**SC.a**
for each col in columns() do:	**SC.b**
If (not at building):	**SC (for block)** = 1 + 3 + 3 + 2 **= 9**
then	
if (No building above):	2+max (1,0) = 3
then:	
from_above = Above	
if (No building on left):	2+max (1,0) = 3
then:	
from_left= left	
if cranes::(from_Above > from_left):	1+max (1,1) = 2
then:	
A[row][col] = from_above	
else:	
A[row][col] = from_west	
Initialize best to origin	**1**
For each element in rows() do:	**SC.x (Outer loop)**
For each element in columns() do:	**SC.y (inner loop)**
If (candidate > best) then:	**SC (for block)** = 2+max(2,0) **= 4**
best = candidate	
Assert (best has value);	
Return best;	

$$SC^{(AB)}_{\text{for block}} = 1 + 2 + \max(1,0) + 2 + \max(1,0) + 1 + \max(1,1)$$

$$= 1 + 2 + 1 + 2 + 1 + 1 + 1$$

$$= 9$$

$$SC_{AB} = \sum_{r=0}^{n} \times \sum_{c=0}^{n} \cdot SC_{\text{for block}}$$

$$= \sum_{r=0}^{n} \times \sum_{c=0}^{n} \cdot 9$$

$$= \sum_{r=0}^{n} 9n$$

$$= 9n^2$$

$$SC_{\text{for block}}^{(xy)} = 2 + \max(2,0) = 4$$

$$SC_{xy} = \sum_{r=0}^{n} \times \sum_{c=0}^{n} \cdot SC_{\text{for block}}$$

$$= \sum_{r=0}^{n} \times \sum_{c=0}^{n} \cdot 4$$

$$= \sum_{r=0}^{n} 4n$$

$$= 4n^2$$

$$SC_{\text{Total}} = 1+1+1+1+SC_{(AB)} + 1 + SC_{(xy)}$$

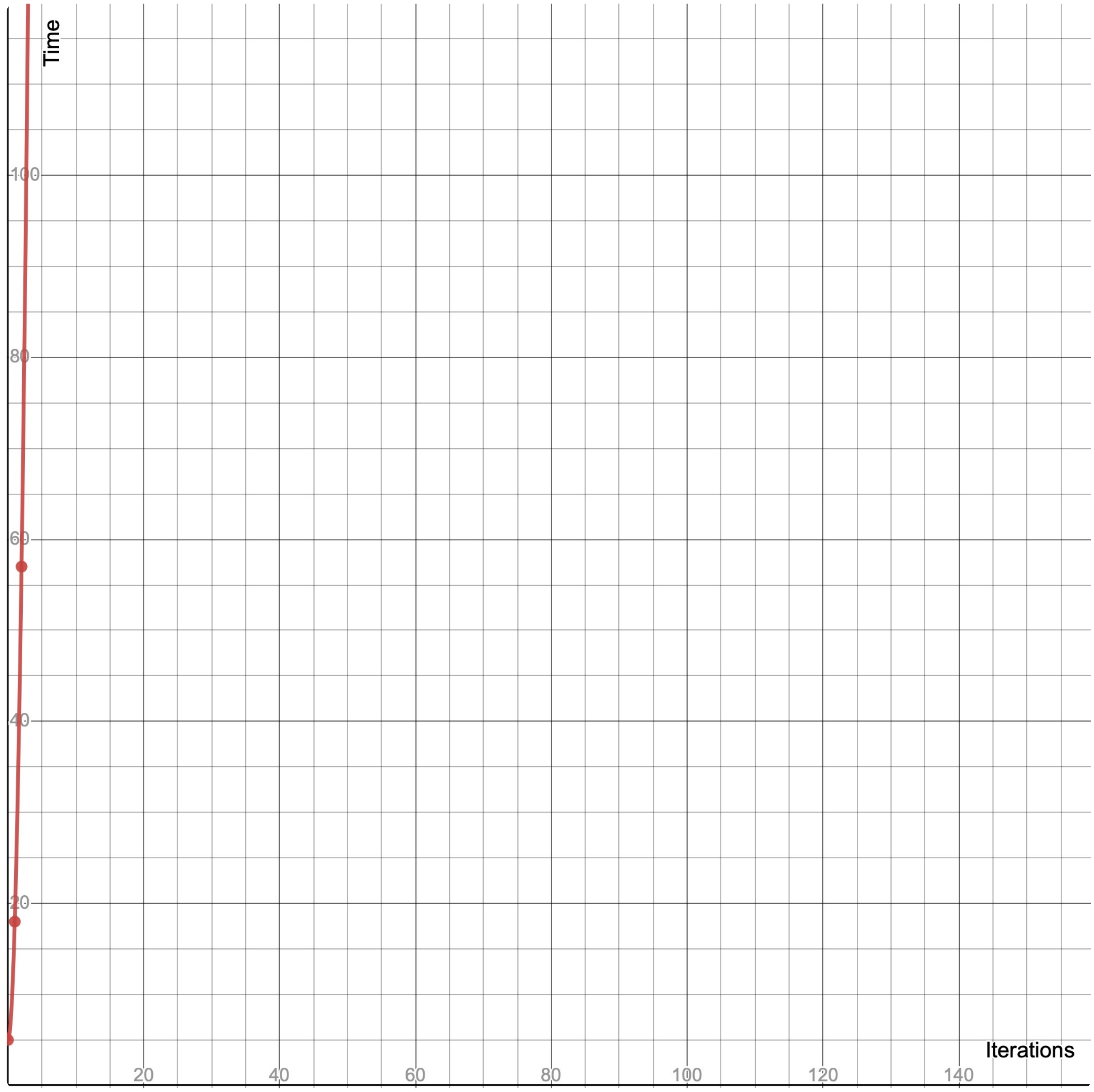
$$= 4 + 9n^2 + 1 + 4n^2$$

$$= 13n^2 + 5$$

Time Complexity $= O(13n^2 + 5)$

$= O(n^2)$

Graph:



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's a definite performance difference between the two algorithms. The Dynamic Programming Algorithm is a faster than the Exhaustive optimization as the latter uses brute force, and has to reevaluate its movements more often. The Dynamic Programming is faster by a factor of n (n^3/n^2).

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

Yes, both analyses are similar. I observed the amount of nested loops required for the Exhaustive Optimization to work properly, and noticed that it will be much slower. The mathematical analysis confirmed my discovery.

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

It is consistent with the hypothesis of the project. The Dynamic Programming Algorithm runs in quadratic time, while the Exhaustive Optimization Algorithm runs in cubic time.

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

I do not see any other hypothesis in the project papers or description.