Analysis

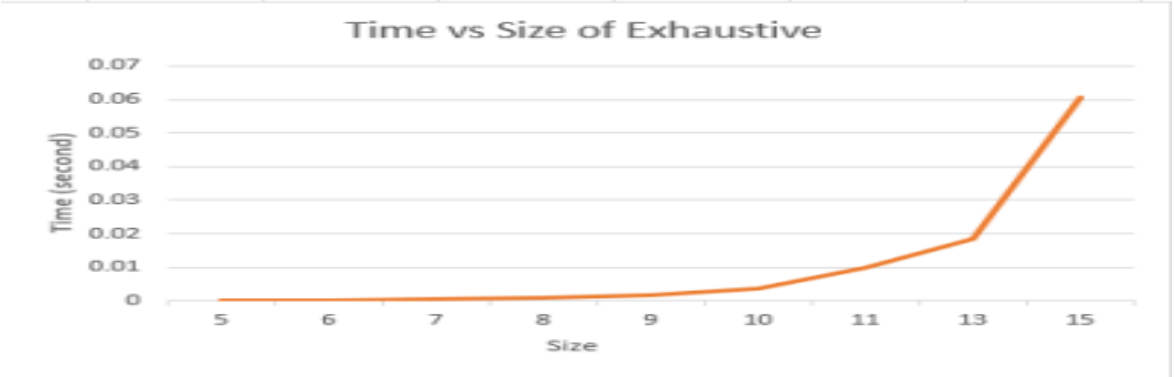
Exhaustive Search Algorithm

```
for (size_t steps = 0; steps <= max_steps; steps++) {                    //N
   for (uint64_t bits = 0; bits <= pow(2, steps) - 1; bits++){           //2^N
     path candidate(setting);                                            //      1
     bool valid = true;                                                  //      1
     for (uint64_t k = 0; k <= steps - 1; k++){                          //      n
       step_direction step_direction;                                    //            1
       int bit = (bits >> k) & 1;                                        //            2
       if (bit == 1){                                                    //            1
         step_direction = STEP_DIRECTION_EAST;                           //            1
       } else {
         step_direction = STEP_DIRECTION_SOUTH;
       }
       if (candidate.is_step_valid(step_direction)){                     //            1
         candidate.add_step(step_direction);                             //            1
       } else {
         valid = false;
         break;
       }
     }

     if (valid){                                                         //            1
       if (candidate.total_cranes() > best.total_cranes()){              //            3
         best = candidate;                                               //            1
       }
     }
   }
 }

 return best;                                                            //            1
```

Time Complexity:
= N * (2 ^ N) * (1 + 1 + 1 + 2 + 1 + 1 + 1 + 1 + 1 +3 + 1) + 1
= (2 ^ N) * N



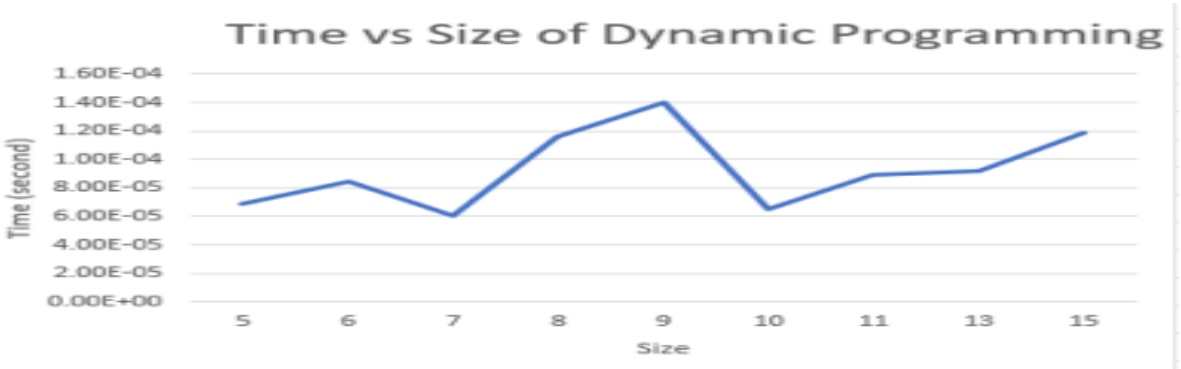
Dynamic Programming Algorithm

A = new r⨯c matrix
# base case
A[0][0] = [start]
# general cases
for r from 0 to n-1 inclusive: //n
    for c from 0 to n-1 inclusive: //n
        if G[r][c]==X:
            A[r][c]=None
            Continue
    from_above = *from_left* = None
    if r>0 and A[r-1][c] is not None:
        from_above = A[r-1][c] + [↓] // n
    if c>0 and A[r][c-1] is not None:
        from_left = A[r][c-1] + [→] // n

A[r][c] = whichever of *from_above* and *from_left* is non-None and has more cranes or None if both *from_above* and *from_left* are None

# post-processing to find maximum-gold path
best = A[0][0] # this path is always legal, but not necessarily optimal
for r from 0 to n-1 inclusive:
    for j from 0 to n-1 inclusive:
        if A[r][c] is not None and [r][c] collects more crane than best:
            best = A[r][c]
return best

Time Complexity:
= N * (N) * (N)
= N* N* N



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?**

We observe that the running speed for Exhaustive Optimization Algorithm is much slower compared with the Dynamic Programming Algorithm. This observation does not surprise us because after mathematically analyzing, we already know that the efficiency class of Exhaustive Optimization Algorithm is O(n*2^n) time (exponential time), which is very slow compared to O(n3) time (cubic time) of the Dynamic Programming Algorithm.

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

Yes, Dynamic Programming Algorithm is much faster than Exhaustive Search Algorithm and from the Mathematical side, N^3 is much smaller than 2^N * N.

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

Based on our evidence and observations in figure 1,2, we conclude that the initial hypotheses for this project is consistent with our experimental data: Polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem.

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

The fit line on Figure1 is a very slow exponential curve, which is consistent with the efficiency class of Exhaustive Optimization Algorithm O(n*2^n) time, exponential time. - The fit line on Figure2 is likely close to an cubic curve, which is consistent with the efficiency class of Dynamic Programming Algorithm as O(n^3) time, cubic time.

```
student@tuffix-vm:~/Desktop/new-project-4-zhaolin-main$ 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:

    TEST FAILED:
    line 104 of file cranes_test.cpp, message: small
    score 0/1
stress test: passed, score 2/2
TOTAL SCORE = 12 / 13
```