Project 4 Report

1. Jacob Nguyen, jn42@csu.fullerton.edu
2. Sam Sandoval, sjsandoval@csu.fullerton.edu

Exhaustive Search Algorithm Analysis

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

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

   // Compute maximum path length, and check that it is legal.
   const size_t max_steps = setting.rows() + setting.columns() - 2; //3 TU
   assert(max_steps < 64);

   path best(setting); //5 TU

   for (size_t steps = 1; steps <= max_steps; ++steps) //n times
    {
     uint64_t mask = uint64_t(1) << steps; //2 TU
     for (uint64_t bits = 0; bits < mask; ++bits) //2^n
     {
       path candidate(setting); //5 TU
       bool valid = true; //1 TU
       // add to candidate a path not exceedings <steps> binary values
       for (size_t k = 0; k < steps; ++k) //n times
       {
         int bit = (bits >> k) & 1; //3 TU
         if (bit == 1) // 1 TU
         {
           if (candidate.is_step_valid(STEP_DIRECTION_EAST)) //6 TU
           { //max(6,1)
             candidate.add_step(STEP_DIRECTION_EAST); // 6 TU
           }
           else
           {
```

```
            valid = false; // 1 TU
          }
        }
        else
        {
          if (candidate.is_step_valid(STEP_DIRECTION_SOUTH)) //6 TU
          {
            candidate.add_step(STEP_DIRECTION_SOUTH); //6 TU
          }
          else
          {
            valid = false; //1 TU
          }
        }
      }
      if (valid && (candidate.total_cranes() > best.total_cranes()))//4 TU
      {
        best = candidate; //1 TU
      }
    }
  }
  return best;
}
```

3 + 5 + n(2 + 2^n(5 + 1 +n(3 + 1 + max(12,12)) + 5)))
8 + n(2 + 2^n(6+n(4+12) + 5))
8 + n(2 + 2^n(11 + 16n)
8 + n(2 + (2^n)11 + 2^(n+4)n)
2^(n+4)n^2 + (2^n)11n +2n + 8

Big O Efficiency Class: 2^(n+4)n^2 + (2^n)11n +2n + 8 ∈ O(2^n(n^2))

Proof by definition: Given f(n), show that f(n) ∈ O(g(n)).

f(n) = 2^(n+4)n^2 + (2^n)11n +2n + 8 , g(n) = 2^n(n^2)

Assume c = 32, n_o = 1

2^(n+4)n^2 + (2^n)11n +2n + 8 ≤ 32 * 2^n(n^2) ∀ n ≥ 1

2^(1+4)(1)^2 + (2^(1))11(1)+ 2(1) + 8 ≤ 32 * 2^1(1)^2

32 + 22 + 2 + 8 ≤ 32 * 2

64 ≤ 64

Therefore 2^(n+4)n^2 + (2^n)11n +2n + 8 $\in$ O(2^n(n^2))

O(2^n(n^2)) = { 2^(n+4)n^2 + (2^n)11n +2n + 8 | $\exists$ c, n_o such that 2^(n+4)n^2 + (2^n)11n +2n + 8 $\leq$ c * 2^n(n^2), $\forall$ n $\geq$ n_o}

Dynamic Programming Algorithm Analysis

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

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

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

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

   A[0][0] = path(setting); //6 TU
   assert(A[0][0].has_value());

   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) //2 TU
       {
         A[r][c] = std::nullopt; //1 TU
         continue;
       }
       cell_type from_above, from_left = std::nullopt; //2 TU
       // set the value for A[r][c] as a path collecting most cranes
       if (r > 0 && A[r - 1][c].has_value()) //4 TU
       {
         from_above = A[r - 1][c]; //n TU
         from_above->add_step(STEP_DIRECTION_SOUTH); //6 TU
       }
       if (c > 0 && A[r][c - 1].has_value()) //4 TU
       {
         from_left = A[r][c - 1]; //n TU
         from_left->add_step(STEP_DIRECTION_EAST); //6 TU
       }
       if (from_left.has_value() && from_above.has_value()) //1 TU
       {
         if (from_left->total_cranes() > from_above->total_cranes()) //1 TU
```

```
        {
          A[r][c] = from_left; //1 TU
        }
        else
        {
          A[r][c] = from_above; //1 TU
        }
      }
      else
      {
        if (from_left.has_value()) //1 TU
        {
          A[r][c] = from_left; //1 TU
        }
        else if (from_above.has_value()) //1 TU
        {
          A[r][c] = from_above; //1 TU
        }
      }
    }
  }

  cell_type *best = &(A[0][0]); //1 TU
  assert(best->has_value());

  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())
      {// 3 TU
        best = &(A[r][c]); //1 TU
      }
    }
  }

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

  return **best; //1 TU
}
```

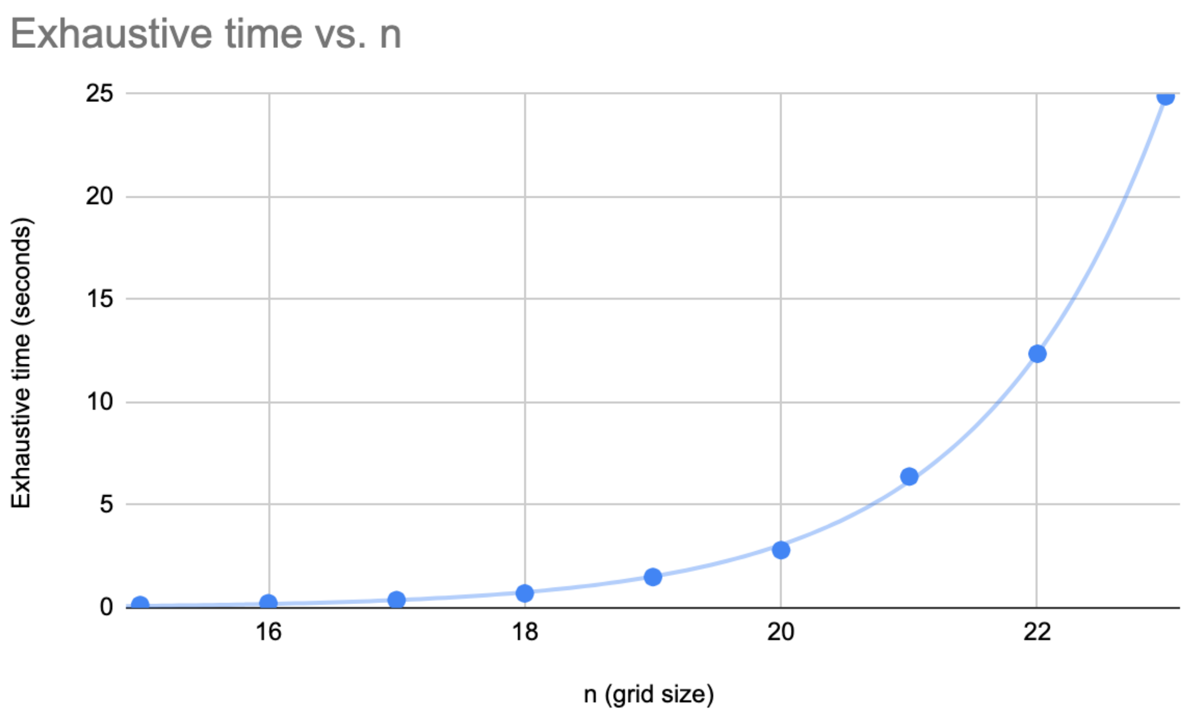
1 + 1 + 6 + n * n(2 + 1 + 2 + 4 + n + 6 + 4 + n + 6 + 1 + max(2, 2)) + 1 + n * n(3 + 1) + 1
8 + n * n(28 + 2n) + 1 + n * 4n + 1
8 + n * (28n + 2n^2) + 1 + n * 4n + 1
8 + 28n^2 + 2n^3 + 1 + 4n^2 +1
2n^3 + 32n^2 + 10
Add + 1 to loops

Big O Efficiency Class: 2n^3 + 32n^2 + 10∈ O(n^3)

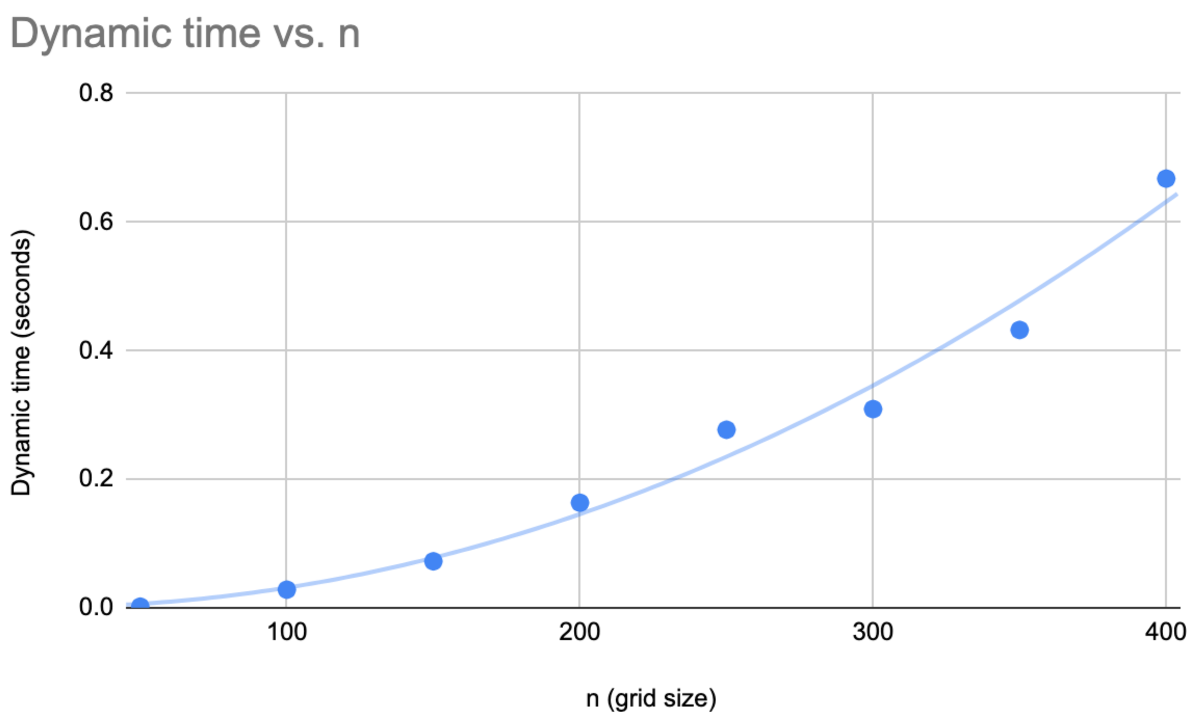
Proof by definition: Given f(n), show that f(n) ∈ O(g(n)).

f(n) = 2n^3 + 32n^2 + 10 , g(n) = n^3
Assume c = 44, n_o = 1
2n^3 + 32n^2 + 10 ≤ 44 * n^3 ∀ n ≥ 1
2(1)^3 + 32(1)^2 + 10 ≤ 44 * (1)^3
2 + 32 + 10 ≤ 44 * 1
44 ≤ 44
Therefore 2n^3 + 32n^2 + 10 ∈ O(n^3)
O(n^3) = { 2n^3 + 32n^2 + 10 | ∃ c, n_o such that 2n^3 + 32n^2 + 10 ≤ c * n^3, ∀ n ≥ n_o}

Scatterplots



Exhaustive Graph (**above**)



Dynamic Graph(**above**)

Responses to Questions

3a. The difference between exhaustive performance and dynamic performance is drastic. Dynamic is much faster even being able to solve a 400 x 400 maze faster than exhaustive can solve a 15 x 15 maze. This is surprising in how fast and efficient dynamic time outscales the exhaustive time.

3b. The empirical analysis is consistent with the scatter plots given that the exhaustive is exponential and we see exponential growth on the time it takes. Also, for dynamic the time scales polynomially which aligns with it being a O(n^2) efficiency class.

3c. Yes, the evidence is consistent with the hypothesis, as we see from the mathematical analysis and empirical evidence of the graphs, we can conclude that the dynamic algorithm is faster than the exhaustive search algorithm.