Table of Contents

Group members

Eduardo Gaxiola ardo1488@csu.fullerton.edu

Zhihuang Chen zhchen@csu.fullerton.edu

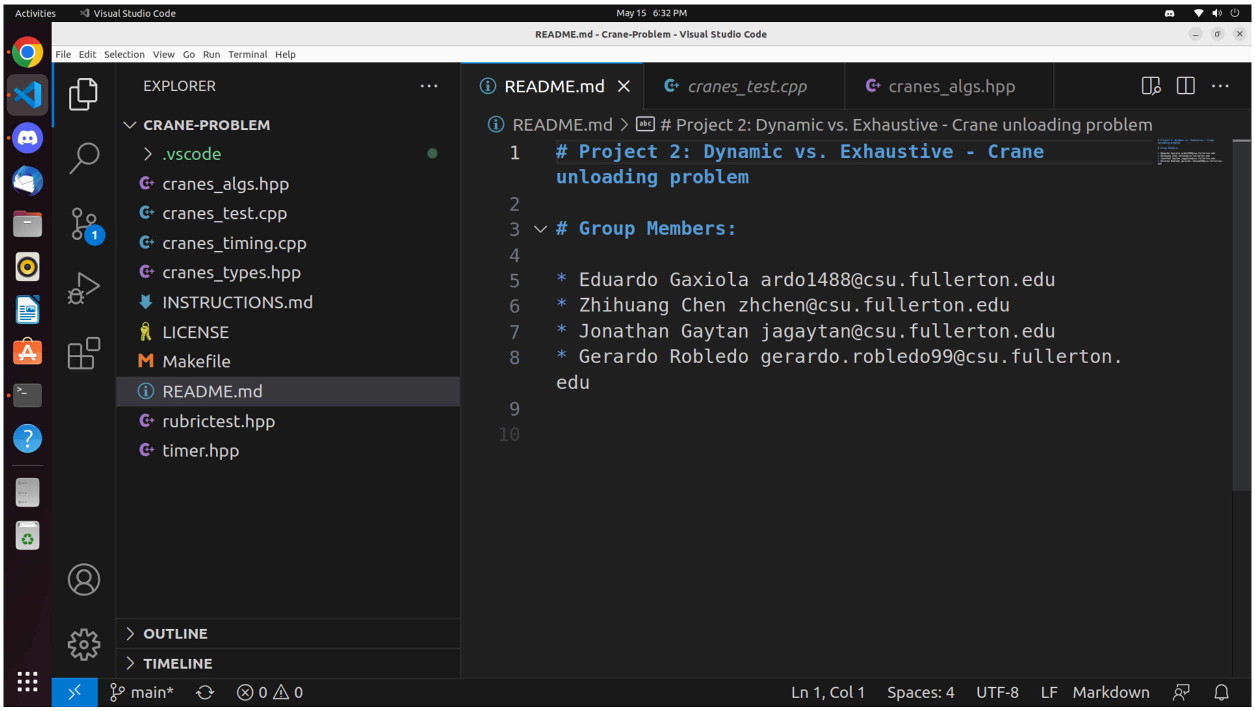
Jonathan Gaytan jagaytan@csu.fullerton.edu

Gerardo Robledo gerardo.robledo99@csu.fullerton.edu

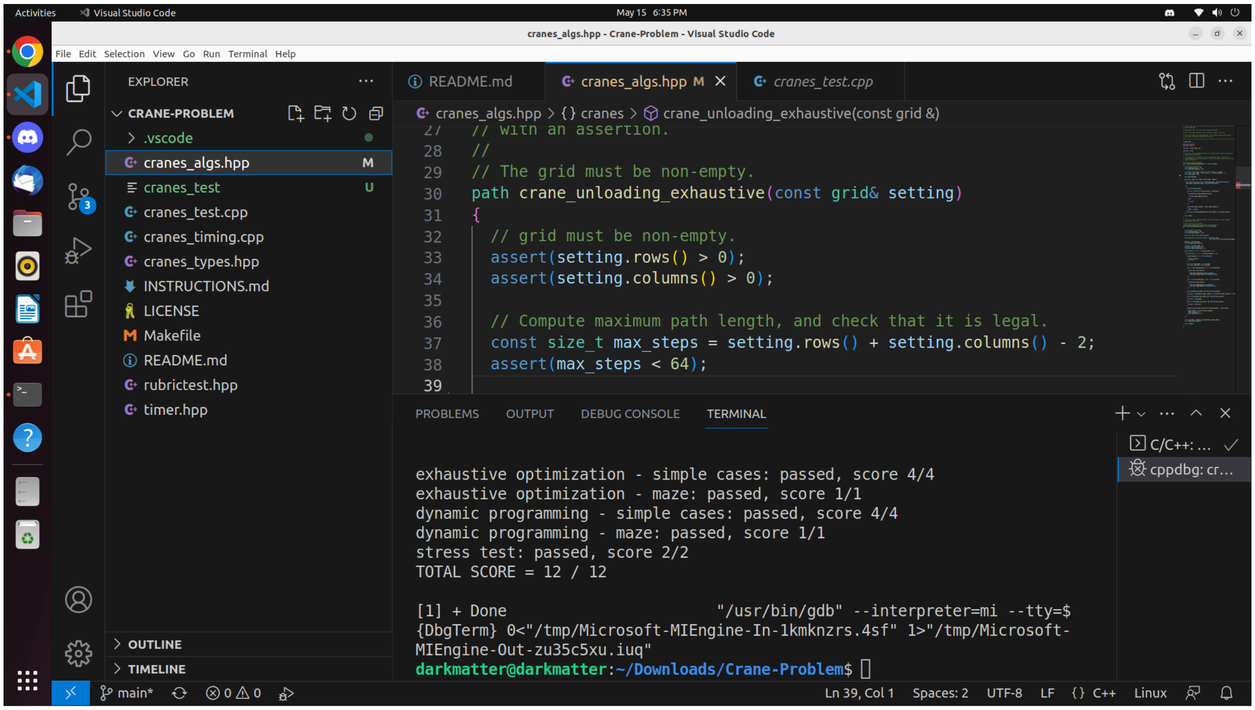
Submission for Project 2

Screenshots of Editor and Code Compiling/Executing

Editor



Code Compiling and Executing



Exhaustive Optimization Algorithm

Pseudocode

```
path crane_unloading_exhaustive(const grid& setting)
{
  // grid must be non-empty.
  assert(setting.rows() > 0); 1 tu
  assert(setting.columns() > 0); 1 tu

  // 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); 1 tu

  path best(setting);

  for(size_t steps = 0; steps <= max_steps; steps++)
  {
        std::vector<step_direction> directions(steps, STEP_DIRECTION_EAST);
        directions.resize(max_steps, STEP_DIRECTION_SOUTH);

        do
        {
        path current(setting);
```

```
 for(const step_direction& direction : directions){

if(current.is_step_valid(direction)){ 1 tu
current.add_step(direction); 1 tu
}
else
{
break;
}
}
if(current.total_cranes() > best.total_cranes())  1 tu
{
best = current; 1 tu
}
} while(std::next_permutation(directions.begin(), directions.end()));
}
return best;
}
```

Time Analysis

Outer loop = max_steps * 2 tu

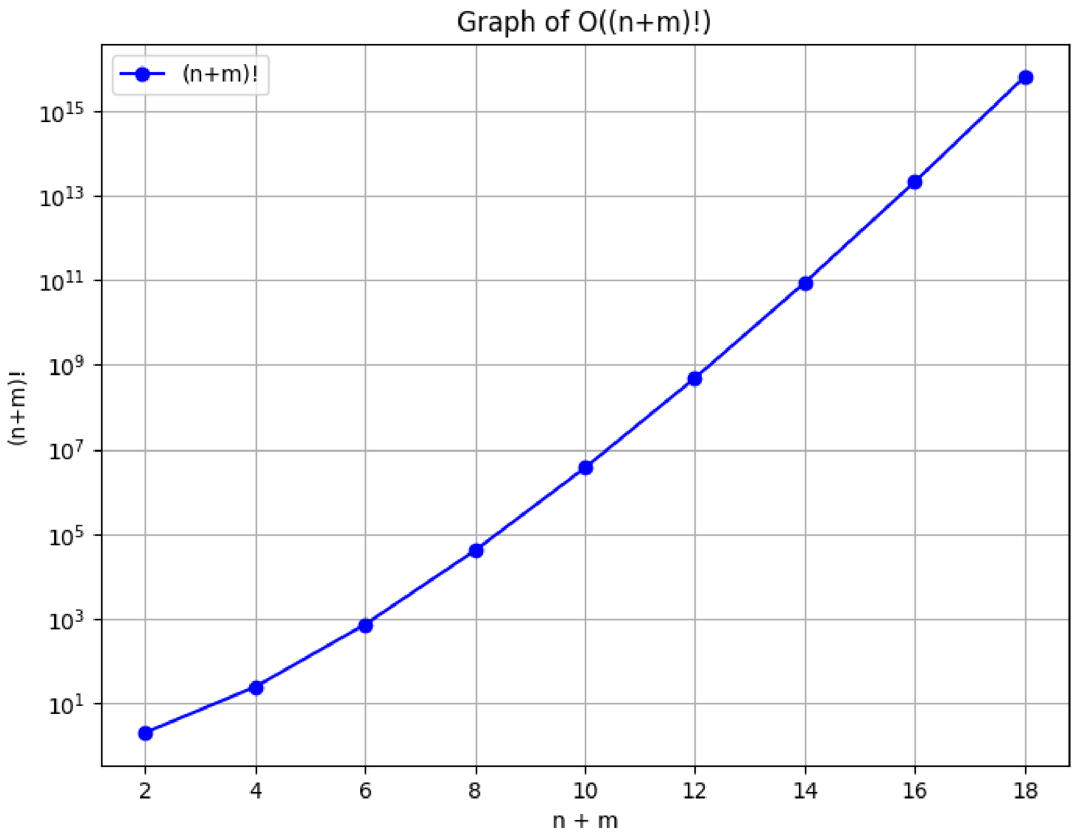
Inner loop = (n + m) * 3 tu

max_steps = setting.rows() + setting.columns() – 2 = n + m - 2

max_steps * 2 + (n + m)! * ((n + m) * 3 + 2) = 2 * max_steps + (n + m)! * (3n + 3m + 2) = 2 * (n + m - 2) + (n + m)! * (3n + 3m + 2)

Thus, the result = O((n+m)!)

Graph


Graph of O((n+m)!)
(n+m)!
10^{15}
10^{13}
10^{11}
10^{9}
10^{7}
10^{5}
10^{3}
10^{1}
(n+m)!
2
4
6
8
10
12
14
16
18
n + m

Dynamic Programming Algorithm

Pseudocode

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

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

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

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

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

  unsigned int most_cranes = 0; 1 tu
  coordinate best_row_path = 0;  1 tu
  coordinate best_column_path = 0;  1 tu

  for(coordinate r = 0; r < setting.rows(); ++r)
  {
    for(coordinate c = 0; c < setting.columns(); ++c)
    {
      if(setting.get(r, c) == CELL_BUILDING) 1 tu
      {
        A[r][c].reset(); 1 tu
```

```
  continue;
 }
 cell_type from_above = std::nullopt;  1 tu
 cell_type from_left = std::nullopt;  1 tu
 if(r != 0 && setting.get(r-1, c) != CELL_BUILDING) 3 tu
 {
  if(A[r-1][c].has_value()) 1 tu
  {
   from_above.emplace(A[r-1][c].value()); 1 tu
   from_above->add_step(STEP_DIRECTION_SOUTH); 1 tu
  }
 }
 if(c != 0 && setting.get(r, c-1) != CELL_BUILDING)  3 tu
 {
  if(A[r][c-1].has_value())  1 tu
  {
   from_left.emplace(A[r][c-1].value()); 1 tu
   from_left->add_step(STEP_DIRECTION_EAST); 1 tu
  }
 }

 if(from_above.has_value() && from_left.has_value()) 1 tu
 {
  A[r][c] = from_above->total_cranes() >= from_left->total_cranes() ? from_above :
from_left; 3 tu
 }
 else if(from_above.has_value() && !from_left.has_value()) 2 tu
 {
  A[r][c] = from_above; 1 tu
 }
 else if(!from_above.has_value() && from_left.has_value()) 2 tu
```

```
        {
          A[r][c] = from_left; 1 tu
        }

        if(A[r][c].has_value() && A[r][c]->total_cranes() > most_cranes)  2 tu
        {
          most_cranes = A[r][c]->total_cranes(); 1 tu
          best_row_path = r;  1 tu
          best_column_path = c;  1 tu
        }
      }
    }

    cell_type *best = &A[best_row_path][best_column_path];  1 tu
    assert(best->has_value()); 1 tu

    return **best;
  }
  }
```

Time Analysis

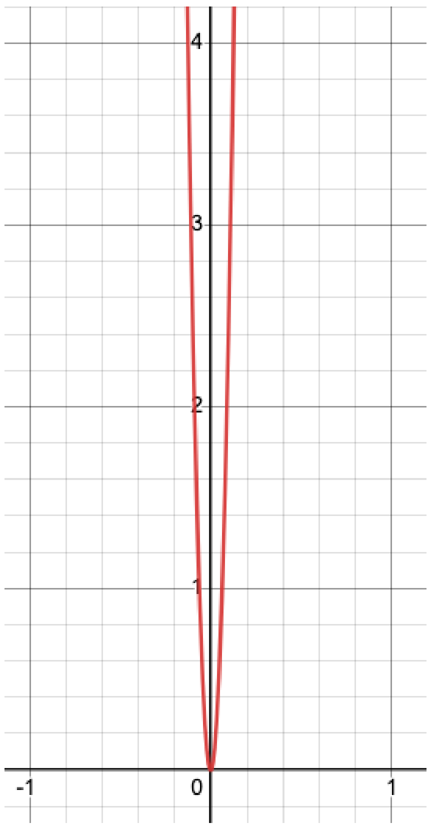
$$10\sum_{0}^{n}\sum_{0}^{m}(1 + 1 + 2 + 3 + 1 + 2 + 3 + 1 + 2 + 4 + 5)$$

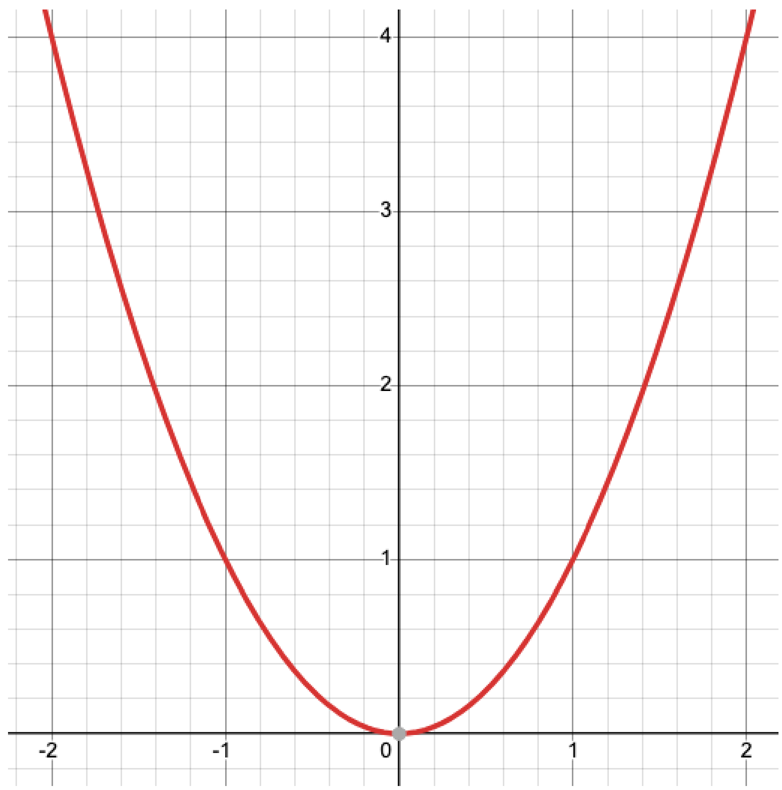
$$\rightarrow 10\sum_{0}^{n}\sum_{0}^{m} 25 \rightarrow 10\sum_{0}^{n} 25m \rightarrow 250\sum_{0}^{n} m \rightarrow 250(n * m)$$

This results in big $O(n^2)$.

Graph

O($250n^2$) O(n^2)





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. The dynamic programming algorithm takes exponentially less time than the exhaustive search algorithm. This didn't surprise us, though, due to what we had learned from the lectures. Exhaustive search is a brute force method of implementing a function; the design is meant to have a much larger run time than that of dynamic programming. Dynamic Programming eliminates the brute force approach and, with that, lowers run time.

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

   Our empirical analyses are consistent with our mathematical analyses and go hand-in-hand. Our mathematical analyses show that our exhaustive search algorithm takes O(n!) time complexity, while the dynamic algorithm only takes polynomial time complexity, O(n^2). Our time analyses mathematically proves our empirical analyses since O(n^2) is more efficient than O(n!).

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

   Hypothesis 1 states that polynomial-time dynamic programming algorithms are more efficient exponential-time exhaustive search algorithms that solve the same problem. Our evidence proves exactly this. Through time analyses we proved the difference between the run times of our dynamic programming algorithm and exhaustive search algorithm. As these algorithms both solve the crane unloading problem, the hypothesis is indeed correct.