Project 2 Crane-Problem Report

Team Members:

Ricardo Granados Macias: ricardog2002@csu.fullerton.edu

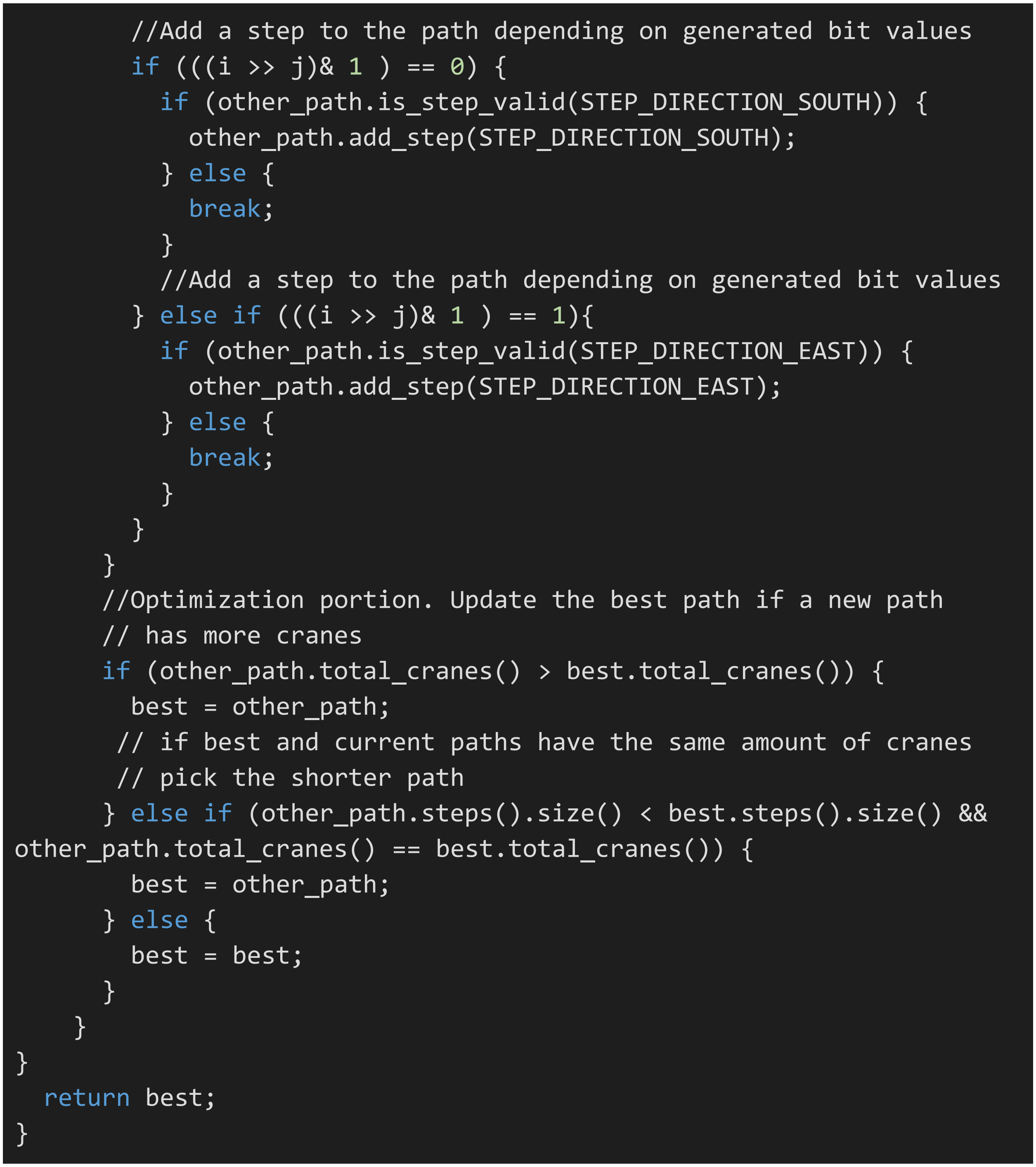
Tommy Ly: lytommy321@csu.fullerton.edu

Project Video Link:

https://drive.google.com/file/d/1u2lWE8wVglWLn5bqpV4Oy924T3DObsoH/view?usp=sharing

Exhaustive Search Optimization Pseudocode

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

  path best(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;
  assert(max_steps < 64);

  for (size_t steps = 0; steps <= max_steps; steps++) {
    for (unsigned i = 0; i < (1 << steps); i++) {
       //Generate path_canidate that will be used to keep track of
       // the best path
        path other_path(setting);
      for (size_t j = 0; j < steps ; j++) {
```

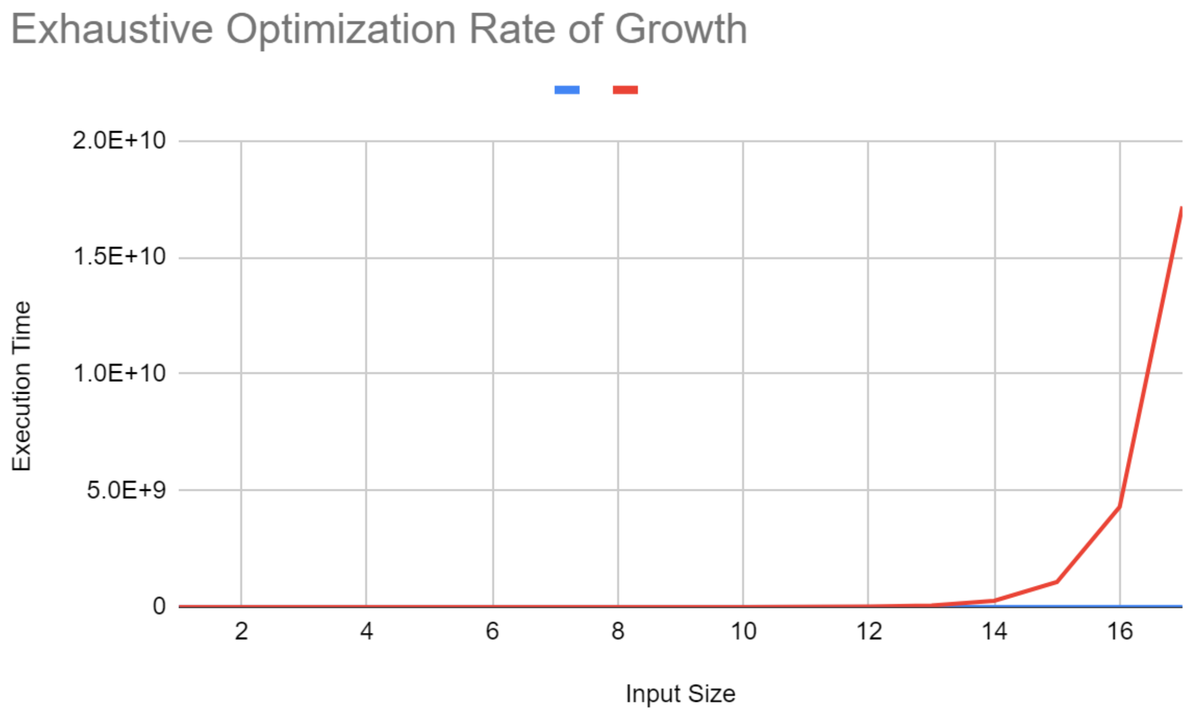


```
        //Add a step to the path depending on generated bit values
        if (((i >> j)& 1 ) == 0) {
          if (other_path.is_step_valid(STEP_DIRECTION_SOUTH)) {
            other_path.add_step(STEP_DIRECTION_SOUTH);
          } else {
            break;
          }
          //Add a step to the path depending on generated bit values
        } else if (((i >> j)& 1 ) == 1){
          if (other_path.is_step_valid(STEP_DIRECTION_EAST)) {
            other_path.add_step(STEP_DIRECTION_EAST);
          } else {
            break;
          }
        }
      }
      //Optimization portion. Update the best path if a new path
      // has more cranes
      if (other_path.total_cranes() > best.total_cranes()) {
        best = other_path;
       // if best and current paths have the same amount of cranes
       // pick the shorter path
      } else if (other_path.steps().size() < best.steps().size() &&
other_path.total_cranes() == best.total_cranes()) {
        best = other_path;
      } else {
        best = best;
      }
    }
}
  return best;
}
```

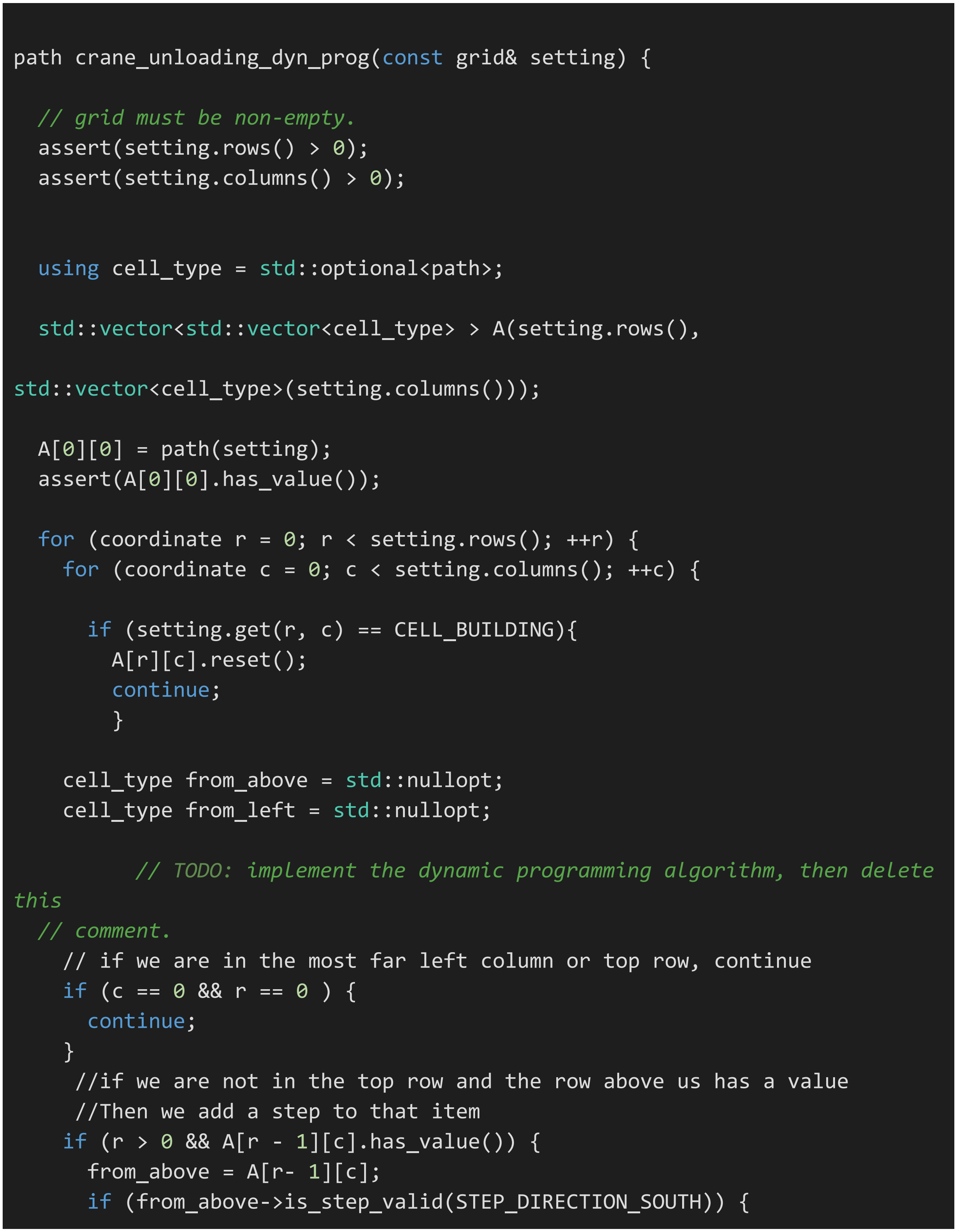
Exhaustive Optimization Search Time Analysis

Exhaustive optimization is a sequential search algorithm that looks for a specific value. More specifically, the exhaustive optimization tries to find a candidate value that best satisfies some condition or problem that you’re trying to solve. First the algorithm must generate the candidate results from the input that it will then attempt to test against other generated values to find the best one. This process of generating candidate results, and then testing each individual one to find the best result. This is usually a lengthy process. In our case, we had a total input size of 15, with 7 rows and 8 columns. We can calculate the time complexity of our exhaustive optimization algorithm with the formula 2^(n + m) + n where n is the number of rows and m is the number of columns. With our input size, our expected time complexity isO(2^(15) + 7) or O(32775). This exponential time complexity becomes very large, very quickly, so our algorithm is an extremely slow one.

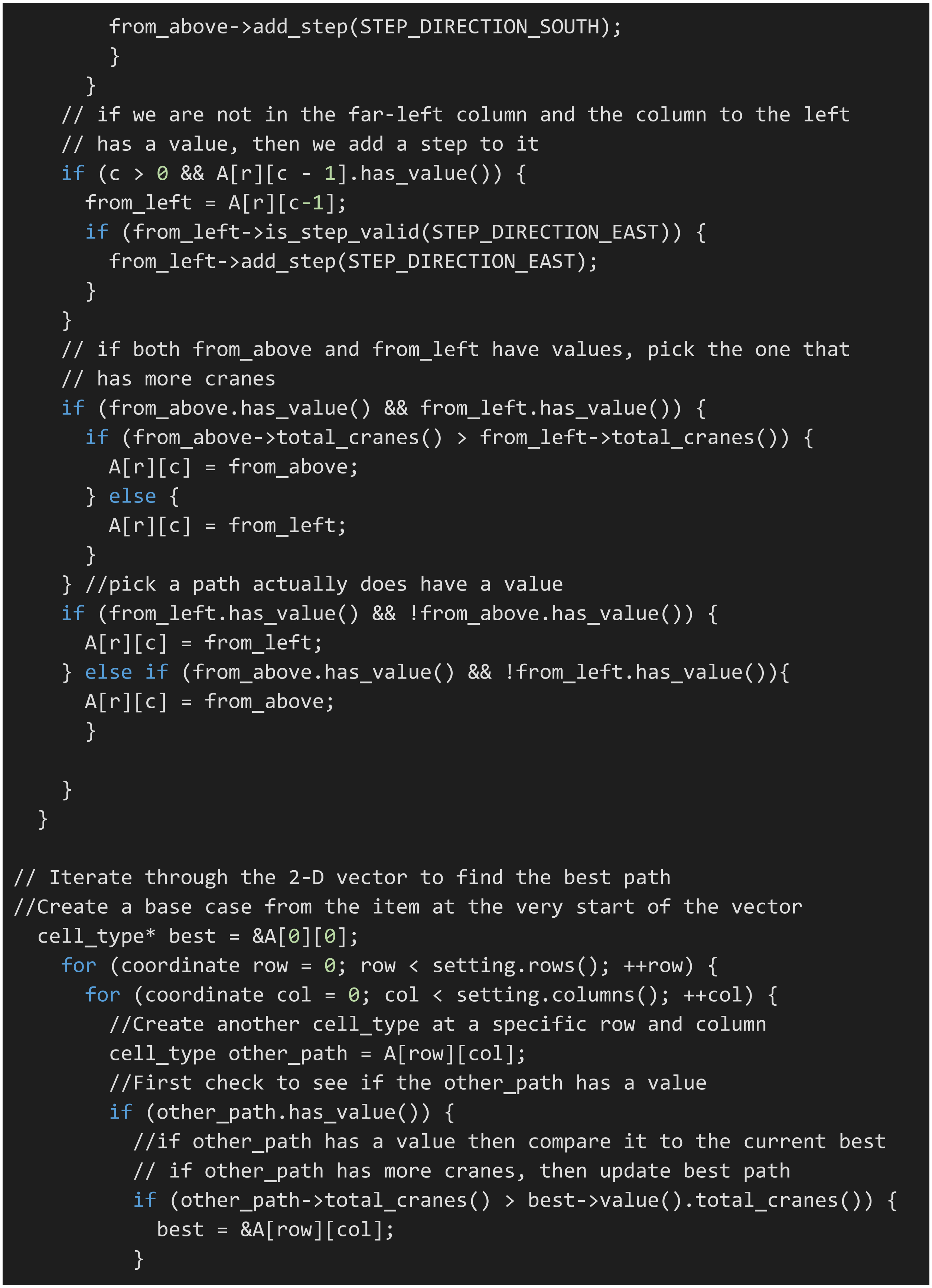
Exhaustive Search Optimization Graph



Dynamic Programming Pseudocode



```
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>;

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

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

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

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

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

    cell_type from_above = std::nullopt;
    cell_type from_left = std::nullopt;

          // TODO: implement the dynamic programming algorithm, then delete
this
  // comment.
    // if we are in the most far left column or top row, continue
    if (c == 0 && r == 0 ) {
      continue;
    }
     //if we are not in the top row and the row above us has a value
     //Then we add a step to that item
    if (r > 0 && A[r - 1][c].has_value()) {
      from_above = A[r- 1][c];
      if (from_above->is_step_valid(STEP_DIRECTION_SOUTH)) {
```



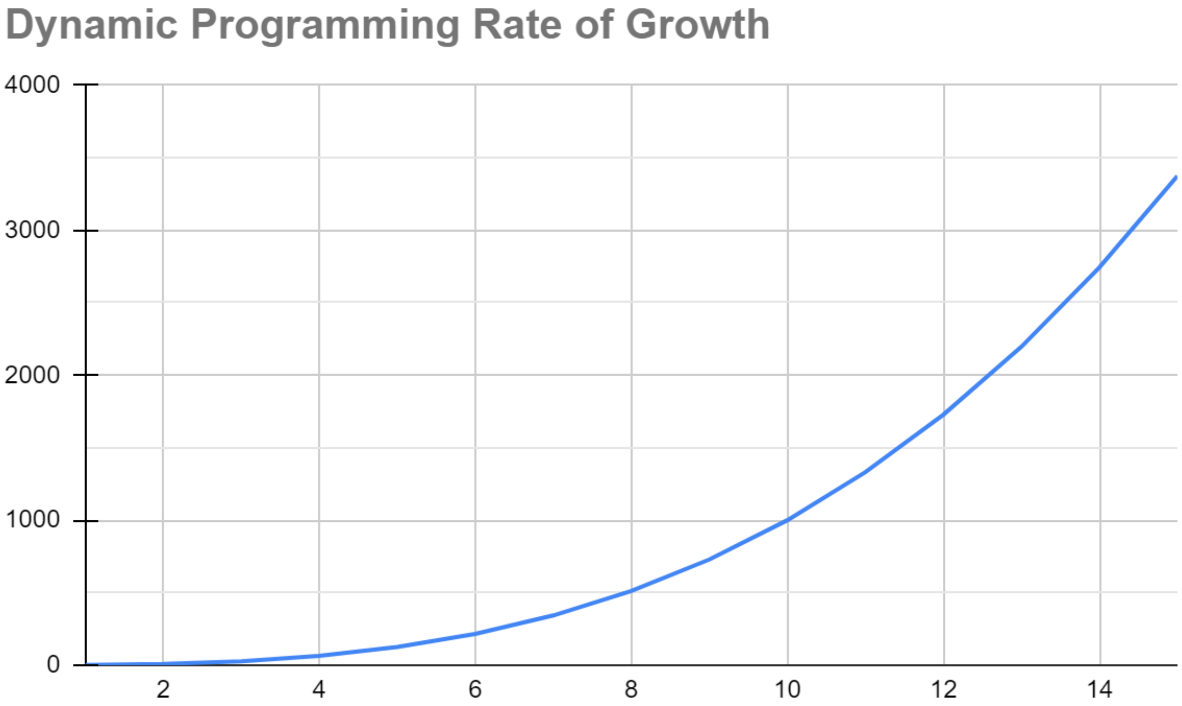
```
        from_above->add_step(STEP_DIRECTION_SOUTH);
        }
      }
    // if we are not in the far-left column and the column to the left
    // has a value, then we add a step to it
    if (c > 0 && A[r][c - 1].has_value()) {
      from_left = A[r][c-1];
      if (from_left->is_step_valid(STEP_DIRECTION_EAST)) {
        from_left->add_step(STEP_DIRECTION_EAST);
      }
    }
    // if both from_above and from_left have values, pick the one that
    // has more cranes
    if (from_above.has_value() && from_left.has_value()) {
      if (from_above->total_cranes() > from_left->total_cranes()) {
        A[r][c] = from_above;
      } else {
        A[r][c] = from_left;
      }
    } //pick a path actually does have a value
    if (from_left.has_value() && !from_above.has_value()) {
      A[r][c] = from_left;
    } else if (from_above.has_value() && !from_left.has_value()){
      A[r][c] = from_above;
      }

    }
  }

// Iterate through the 2-D vector to find the best path
//Create a base case from the item at the very start of the vector
  cell_type* best = &A[0][0];
    for (coordinate row = 0; row < setting.rows(); ++row) {
      for (coordinate col = 0; col < setting.columns(); ++col) {
        //Create another cell_type at a specific row and column
        cell_type other_path = A[row][col];
        //First check to see if the other_path has a value
        if (other_path.has_value()) {
          //if other_path has a value then compare it to the current best
          // if other_path has more cranes, then update best path
          if (other_path->total_cranes() > best->value().total_cranes()) {
            best = &A[row][col];
          }
```

```
//Return the best path
   return **best;
  }
```

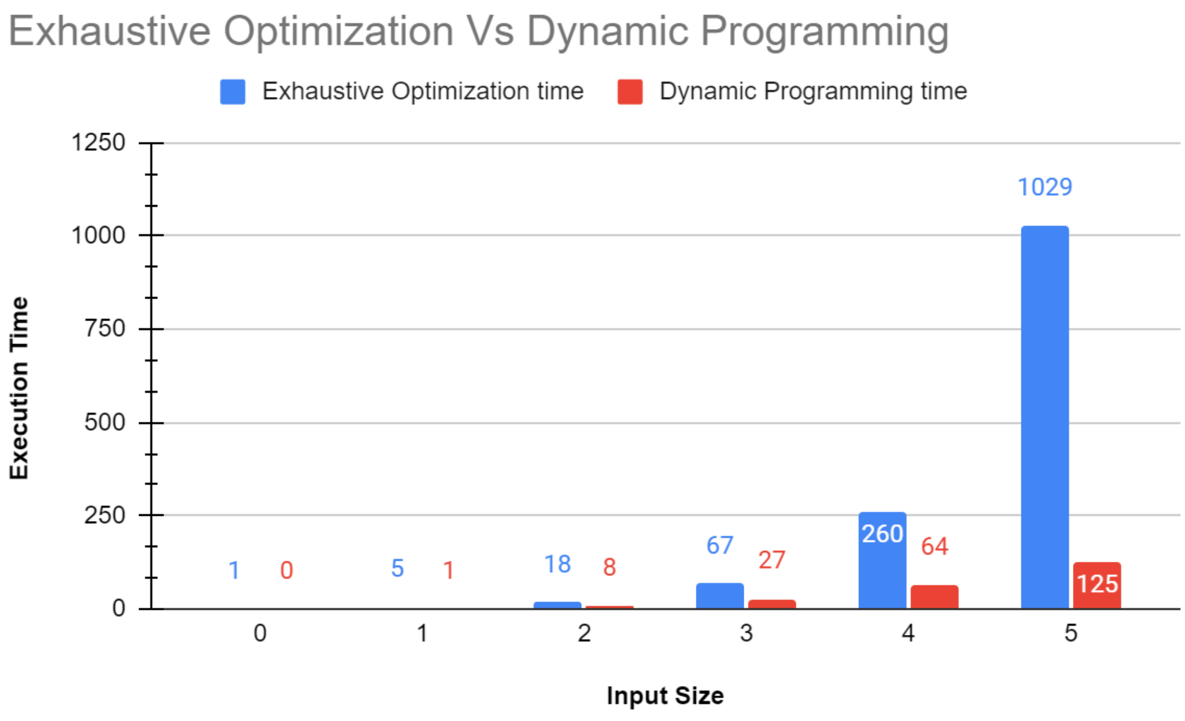
Dynamic Programming Time Analysis

Dynamic programming is an algorithm that can be used to eliminate recursive calls for divide and conquer problems. It works by populating an array with values, and using the values stored in the array to calculate future values. This eliminates the need of recalculating any values that may have been previously calculated as they can be pulled from our cache vector to be used for later. In this case we had a 2-D vector. Normally, this would mean that this dynamic programming problem would be a dp-2 problem. However, because of the final post processing loop to find the best path, this specific dynamic programming problem is O(n^3) time complexity. In this case the input size n, was 15. Calculating our time complexity using the formula of O(n^3), the estimated time complexity for this specific algorithm is O(15^3) or O(3375) time units.

Dynamic Programming Graph

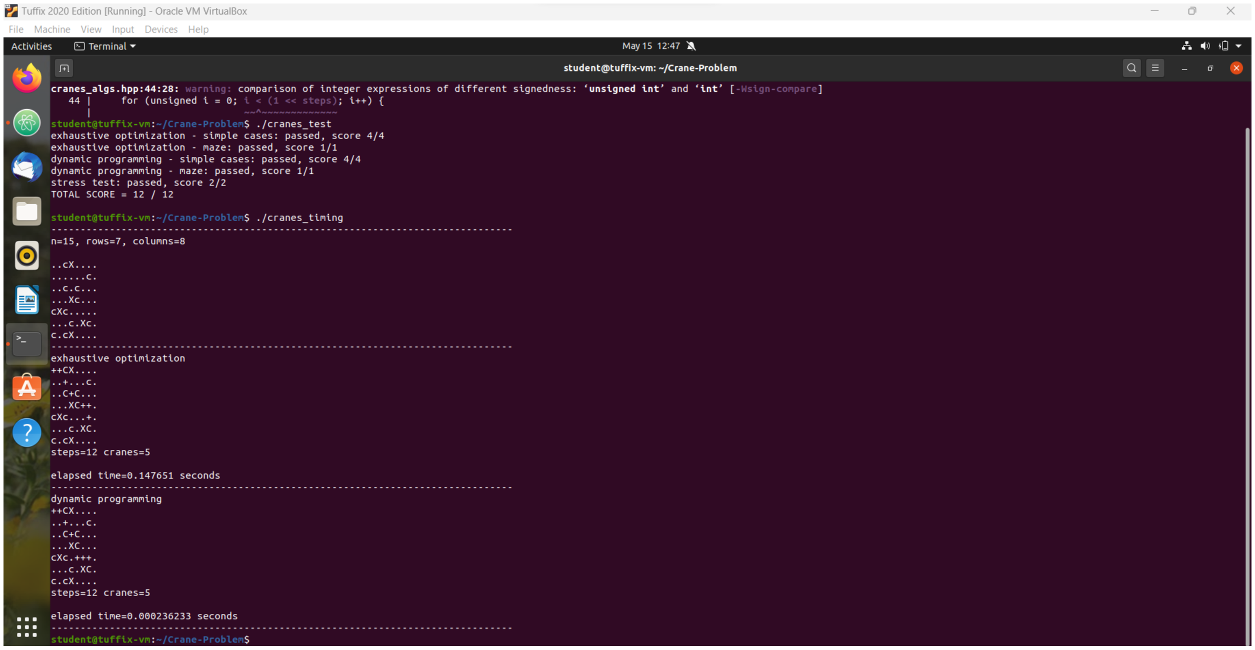


Dynamic Programming Graph Vs Exhaustive Optimization Search Graph



Project 2 Questions

Cranes_timing.cpp Time Measurement and Results


Tuffix 2020 Edition [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
May 15 12:47
student@tuffix-vm: ~/Crane-Problem

```
cranes_algs.hpp:44:28: warning: comparison of integer expressions of different signedness: 'unsigned int' and 'int' [-Wsign-compare]
   44 |     for (unsigned i = 0; i < (1 << steps); i++) {
      |                          ~~^~~~~~~~~~~~~~
student@tuffix-vm:~/Crane-Problem$ ./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
stress test: passed, score 2/2
TOTAL SCORE = 12 / 12

student@tuffix-vm:~/Crane-Problem$ ./cranes_timing
-------------------------------------------------------------------------------
n=15, rows=7, columns=8

..cX....
......c.
..c.c...
...Xc...
cXc.....
...c.Xc.
c.cX....
-------------------------------------------------------------------------------
exhaustive optimization
++CX....
..+...c.
..C+C...
...XC++.
cXc...+.
...c.XC.
c.cX....
steps=12 cranes=5

elapsed time=0.147651 seconds
-------------------------------------------------------------------------------
dynamic programming
++CX....
..+...c.
..C+C...
...XC...
cXc.+++.
...c.XC.
c.cX....
steps=12 cranes=5

elapsed time=0.000236233 seconds
-------------------------------------------------------------------------------
student@tuffix-vm:~/Crane-Problem$
```

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

Yes there is a noticeable difference in the performance of the two algorithms. Dynamic programming is faster than the exhaustive programming approach by 625 Times.
0.147651 seconds / 0.000236233 seconds. = 625.02275296 difference in performance.
No this does not surprise us, as exhaustive optimization typically takes a long time to execute in order to ensure that the candidate it returns is the best one to solve a problem. As the input size increases, the benefits of storing subproblem solutions increase as well because we do not have to recompute the solutions to variations of problems.

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

Yes, our observations are consistent with our mathematical analyses. Our mathematical analysis concluded that with a total input size of 15 (7 rows and 8 columns) that our exhaustive optimization algorithm would take approximately take O(32775) time units, while our dynamic programming algorithm would take around O(3375) time units, which is much shorter than the optimization algorithms time complexity. Additionally, when we initially timed our code, we noticed that the exhaustive program took 0.147651 seconds. On the other hand, the optimization problem took a fraction of that time, taking a total of 0.000236233 seconds to solve the same problem. Furthermore, when we graphed the time complexity for the two programs, the exhaustive optimization rate of growth grew extremely quickly, and spiked with an input size of only 15. On the other hand, dynamic programming increased with a gradual curve, which can be seen above.

3. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer. is dynamic faster than exhaustive

Yes, polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem. One reason why this is consistent is that exponential equations grow significantly faster than polynomial equations. As can be seen from the Exhaustive Optimization vs Dynamic Programming chart, the exponential equation grows significantly faster than the polynomial function even at smaller input sizes. At input size 1 the difference in time efficiency is 5x in favor of the polynomial function. At input size 5, the difference in time efficiency is 1029/125 = 8.232. We can see the trend that as the input size increases, dynamic programming becomes more efficient than exhaustive programming.

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

The evidence is inconsistent with hypothesis 2. The evidence collected from graphing the two functions and our mathematical analysis concluded that exhaustive optimization takes much longer than dynamic programming which contradicts the second hypothesis. We graphed data onto three graphs, one for the execution time of exhaustive optimization, another for dynamic programming, and a third one that directly compares the two graphs. In the final graph that compares the run times for the two algorithms. This graph illustrates that an exhaustive optimization algorithm has a much faster rate of growth than the dynamic programming algorithm. With a total input size of 4, it takes exhaustive optimization approximately 260 time units to execute, but it takes the dynamic programming solution only 64 time units to execute. Therefore, exhaustive optimization has a much bigger rate of growth than dynamic programming. As the input increases, the run time for the exhaustive optimization problem is going to take increasingly longer than the dynamic programming algorithm. Therefore, the second hypothesis is not supported by our data.