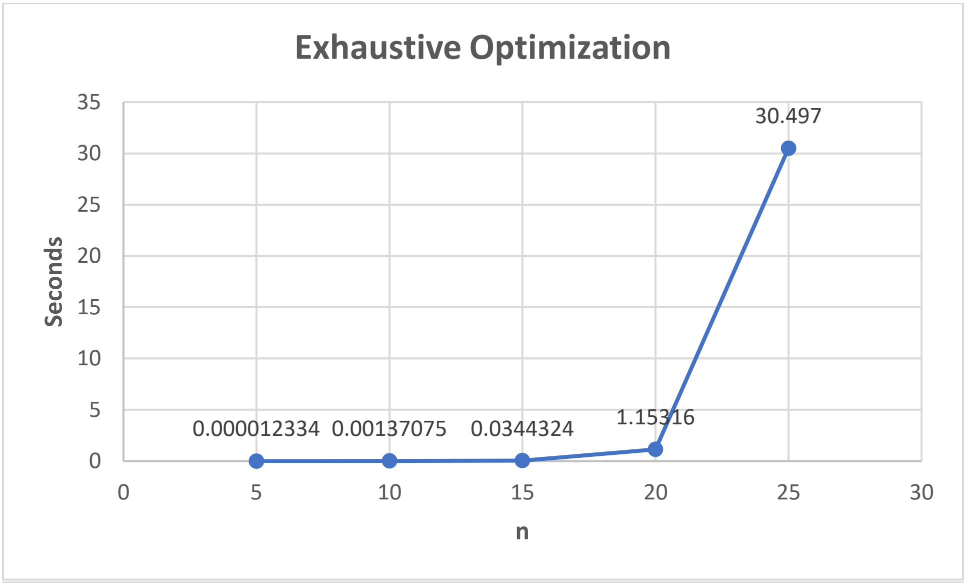
Moses Merugu

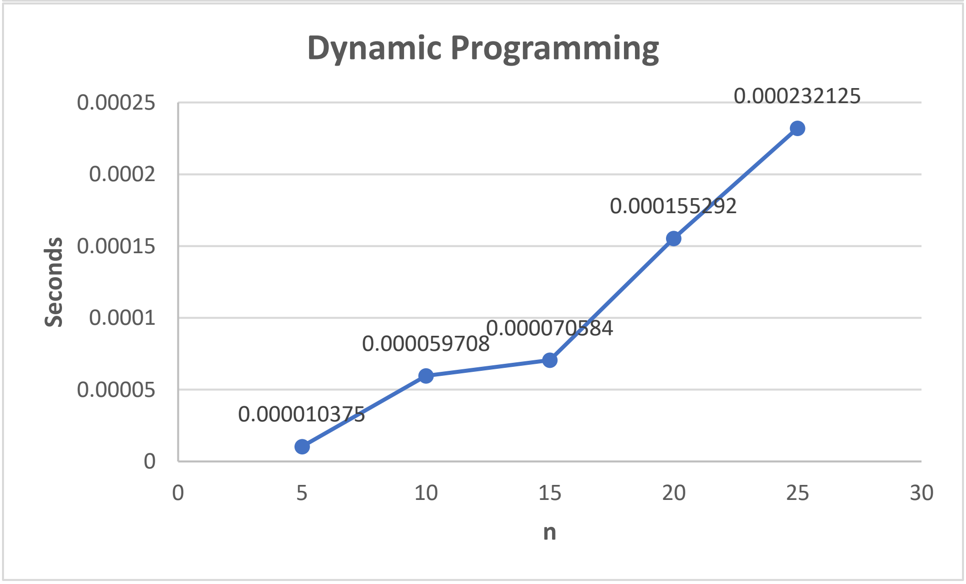
mmeru@csu.fullerton.edu

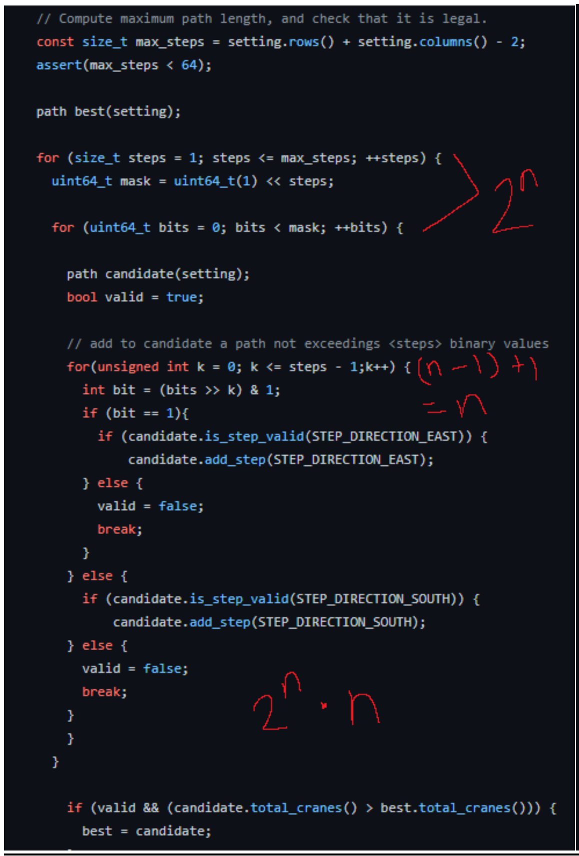
CPSC 335-01

Project 4 Dynamic vs. Exhaustive

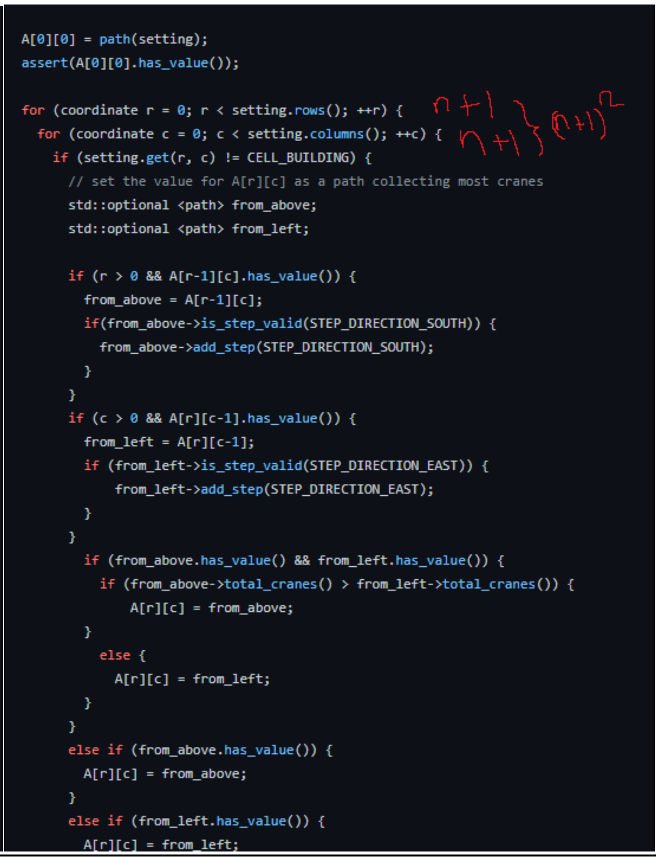
n	Exhaustive Optimization	Dynamic Programming
5	0.000012334	0.000010375
10	0.00137075	0.000059708
15	0.0344324	0.000070584
20	1.15316	0.000155292
25	30.497	0.000232125







```
// Compute maximum path length, and check that it is legal.
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 mask = uint64_t(1) << steps;

  for (uint64_t bits = 0; bits < mask; ++bits) {

    path candidate(setting);
    bool valid = true;

    // add to candidate a path not exceedings <steps> binary values
    for(unsigned int k = 0; k <= steps - 1;k++) {
      int bit = (bits >> k) & 1;
      if (bit == 1){
        if (candidate.is_step_valid(STEP_DIRECTION_EAST)) {
            candidate.add_step(STEP_DIRECTION_EAST);
      } else {
        valid = false;
        break;
      }
    } else {
      if (candidate.is_step_valid(STEP_DIRECTION_SOUTH)) {
          candidate.add_step(STEP_DIRECTION_SOUTH);
    } else {
      valid = false;
      break;
    }
    }
  }

    if (valid && (candidate.total_cranes() > best.total_cranes())) {
      best = candidate;
```

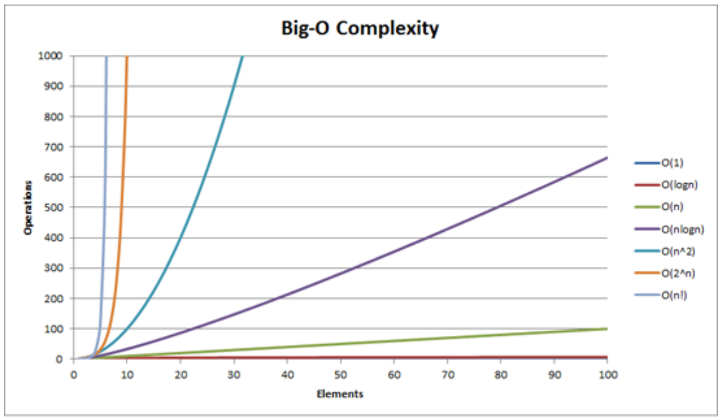


```
A[0][0] = path(setting);
assert(A[0][0].has_value());

for (coordinate r = 0; r < setting.rows(); ++r) {
  for (coordinate c = 0; c < setting.columns(); ++c) {
    if (setting.get(r, c) != CELL_BUILDING) {
      // set the value for A[r][c] as a path collecting most cranes
      std::optional <path> from_above;
      std::optional <path> from_left;

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

Questions

3a) There is a substantial difference between the performance of the two algorithms. The dynamic programming algorithm is much faster than the exhaustive by a large margin. The exhaustive method struggles as n gets larger and can take up to ~30 seconds. The dynamic method on the other hand takes milliseconds to run with any value of n. This does not surprise me as the exhaustive method uses brute-forcing principles while dynamic does not.

3b)



I do believe my empirical analyses are consistent with my mathematical analyses. When compared to the Big-O chart, both of my exhaustive and dynamic lines correlate with the O(2^n) and O(n^2) lines.

3c) The evidence I have gathered is in fact consistent with the hypothesis of polynomial-time dynamic programming algorithms having greater efficiency than exponential-time exhaustive search algorithms that solve the same problem. This can clearly be seen in the chart and scatter plots made for each algorithm that clearly reflect the greater efficiency of the dynamic algorithm given different values of n.