David Nguyen

Vivian Truong

CPSC 335

6 May 2023

CPSC 335 Project 2 Submission Document

Github Link: https://github.com/Pzychopomp/CPSC-335-Proj2

Pseudo Code and Time Efficiency(in red)

The Exhaustive Optimization Algorithm Pseudocode:

```
def crane_unloading_exhasutive (setting):
      assert(setting.rows() > 0)  (3)
      assert(setting.columns() > 0)  (3)

      max_steps = setting.rows() + setting.columns() - 2 (5)
      assert(max_steps < 64)  (2)
      best = None  (1)

      for steps = 1 to max_steps inclusive:
          for bits = 0 to (2^steps) - 1 inclusive:
              candidate = [start]  (1)
              valid = true  (1)
              for k = 0 to steps - 1 inclusive:
```

```
                    bit = (bit >> k) & 1 (3)


                    if (bit == 1): (1)

                        if (candidate.is_step_valid(STEP_DIRECTION_EAST): (1)

                            candidate.add_step(STEP_DIRECTION_EAST): (1)

                        else valid = false (1)

                    else:

                        if (candidate.is_step_valid(STEP_DIRECTION_SOUTH): (1)

                            candidate.add_step(STEP_DIRECTION_SOUTH) (1)

                        else valid = false (1)


                endfor

            if (valid && (candidate.total_cranes() > best.total_cranes())): (4)

                best = candidate (1)

            endfor

        endfor
```

The Exhaustive Algorithm Step Count:

$$sc = 3 + 3 + 5 + 2 + 1 + \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} [(1 + 1 + \sum_{k=0}^{s-1} (3 + 3)) + 5]$$

$$= 14 + \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} [2 + \sum_{k=0}^{s-1} (6) + 5]$$

$$= 14 + \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} [2 + 6(s - 1 - 0 + 1) + 5]$$

$$= 14 + \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} [6s + 7]$$

$$= 14 + \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} (6s) + \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} (7)$$

$$= 14 + \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} (6s) + \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} (7)$$

$$= \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} (6s) = \sum_{s=1}^{n} 6s(2^s - 1 + 1) = \sum_{s=1}^{n} 6s(2^s) = 6 \sum_{s=1}^{n} s(2^s)$$

$$= 6(1(2^1) + 2(2^2) + ... + n(2^n)) = 12(1 - 2^n + 2^n n)$$

$$= \sum_{s=1}^{n} \sum_{b=0}^{2^s-1} (7) = \sum_{s=1}^{n} 7(2^s - 1 - 0 + 1) = \sum_{s=1}^{n} 7(2^s) = 7(2^1 + 2^2 + 2^3 + ... + 2^n)$$

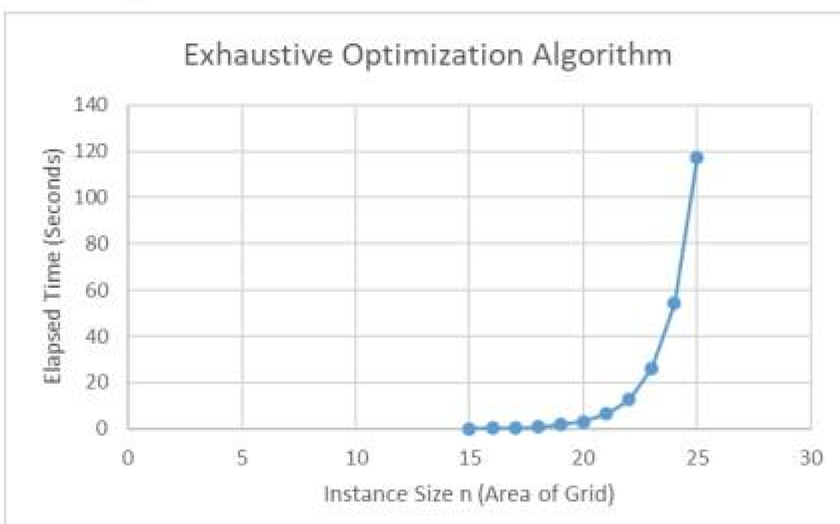
$$= 7[2(2^n - 1) = 14(2^n - 1)$$

$$= 14(12(1 - 2^n + 2^n n)) + 14(2^n - 1)$$

$$= 14 + 12 - 12(2^n) + 12(2^n)(n) + 14(2^n) - 14$$

Final Step Count = $12(2^n)(n) + 2(2^n) + 12$

Graph for Exhaustive Search



② Exhaustive

Limit

$$12(2n^2)(n) + 2(2^n) + 12 \in O(2^n * n)$$

$$\lim_{n \to \infty} \frac{12(2n^2)(n) + 2(2^n) + 12}{2^n * n} = 0$$

$$\lim_{n \to \infty} 12(2n) + 2^{-n} + 12 / (n * 2^n)$$

$$\lim_{n \to \infty} \frac{(24n + 2^{(1-n)})}{2^n} + 12 / (n * 2^n)$$

$$0 = 0$$

By L.T. $12(2n^2)(n) + 2(2n) + 12 \in O(2^n * n)$

L'Hopital

$$\frac{12(2n^2)(n) + 2(2^n) + 12}{2^n * n}$$

Factor: $\frac{2^n(6n^3+1)+6)}{2^n * n}$

$\lim_{n \to \infty} \frac{6n^3+1+6}{2^n * n}$ Derivative $\Rightarrow \lim_{n \to \infty} \frac{18n^2}{2^n * \ln(2) + n + 2^n}$

$\Rightarrow \lim_{n \to \infty} \frac{18n^2}{2^n * \ln(2) + n + 2n} = 0$

The Dynamic Programming Algorithm Pseudocode:

```
crane_unloading_dyn_prog(setting):

    assert(setting.rows() > 0) (3)
    assert (setting.columns > 0) (3)

    A = (setting.rows(), vector<cell_type>(setting.columns())) (3)
    A[0][0] = path(setting) (2)
    assert(A[0][0].hash_value()) (2)

    for r = 0 to setting.rows() - 1: (n - 1 - 0 + 1) = n
        for c = 0 to setting.columns() - 1: (n - 1 - 0 + 1) = n
          if (setting.get(r, c) != CELL_BUILDING): (2)
              from_above = None (1)
              from_left = None (1)

              if (r > 0 && A[r - 1][c].has_value()): (4)
```

```
                from_above = A[r -1][c] (2)
                if (from_above->is_step_valid(STEP_DIRECTION_SOUTH): (1)
                    from_above->add_step(STEP_DIRECTION_SOUTH) (1)
                     (4 + 2 + 1 + 1 = 8)

            if (c > 0 && A[r][c - 1].has_value()): (4)
                from_left = A[r][c - 1] (2)
                if (from_left->is_step_valid(STEP_DIRECTION_EAST)): (1)
                    from_left->add_step(STEP_DIRECTION_EAST) (1)
                     (4 + 2 + 1 + 1 = 8)

            if (from_above.has_value() && from_left.has_value()): (3)
                if (from_above->total_cranes() > from_left->total_cranes()): (3)
                    A[r][c] = from_above (1)
                else: A[r][c] = from_left (1)
                     (3 + max(3 + max(1, 1), 0) = 3 + max(4, 0) = 3 + 4 = 7)

            if (from_above.has_value() && !(from_left.has_value())): (4)
                A[r][c] = from_above (1)

            if (from_left.has_value() && !(from_above.has_value())): (4)
                A[r][c] = from_left (1)
            endif
        end for
    end for

    // Post-processing to find maximum-crane path
    best = A[0][0] (1)
    assert(best->has_value()) (2)
    for r = 0 to setting.rows() - 1: (n - 1 - 0 + 1) = n
    for c = 0 to setting.columns() - 1: (n - 1 - 0 + 1) = n
    if (A[r][c].has_value() && A[r][c]->total_cranes() > (*best)->total_cranes()):
(4)
    best = &(A[r][c])(1)
    assert(best->has_value()) (2)

return best (0)
```

The Dynamic Programming Algorithm Step Count:

sc = $3 + 3 + 3 + 2 + 2 + n[n * (2 + max(1 + 1 + 8 + 8 + 7 + 5 + 5, 0))] + 3 + 5n^2$

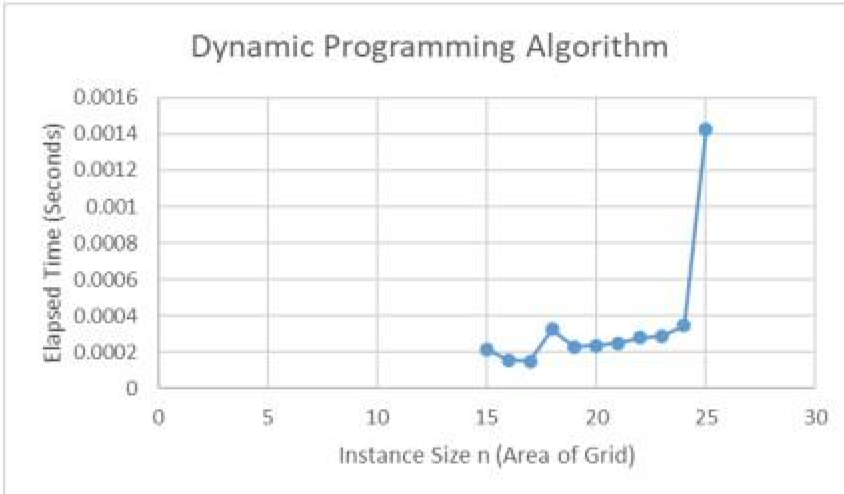
sc = $16 + n[n * (2 + 35)] + 5n^2$

sc = $16 + 37n^2 = 5n^2$

sc = $16 + 42n^2$

Final Step Count = $42n^2 + 16$

Graph for Dynamic Search



① Dynamic Programming: $O(n^2)$

Limit: $42n^2+16 \in O(n^2$

$\lim\limits_{n \to \infty} \dfrac{42n+16}{n^2} = \lim\limits_{n \to 100} \dfrac{42n}{n^2} + \lim\limits_{n \to 100} \dfrac{16n}{n^2} \neq 0$

By L.T. $42n^2+16 \in O(n^2)$

L'Hopital

$42n^2+16 \in O(n^2)$

①

$\lim\limits_{n \to \infty} \dfrac{42n^2+16}{n^2}$

②

$\lim\limits_{n \to \infty} \dfrac{84n}{2n} = \lim\limits_{n \to \infty} 42 \leq 42$ ✓

Question Responses

1. Yes there is a significant difference in the performance of the two algorithms given. We see that dynamic programming is significantly faster than the exhaustive algorithm. The exhaustive algorithm is in exponential time complexity while the dynamic algorithm is in polynomial time complexity. In all, the dynamic algorithm can finish within less than a second while the exhaustive algorithm takes a couple minutes to finish. When using large inputs, it is logical that the exhaustive algorithm becomes unsuitable while the dynamic algorithm is appropriate.

2. Given the time unit analysis and the graph of the two algorithms, we can see that the dynamic programming algorithm is much faster than the exhaustive algorithm. The exhaustive algorithm is in exponential time so therefore as the input size increases so does the time. In all, this idea supports my empirical analysis with my mathematical analysis. The dynamic programming time complexity is O(n^2). When the input size increases for this algorithm, the time increases polynomially due to its being O(n^2). Therefore, It is faster than exhaustive optimization.

3. This evidence is consistent with hypothesis 1 stating that polynomial time dynamic program programming algorithms are more efficient than exponential- time exhaustive search algorithms that solves the same problem. We know that the dynamic programming algorithm has a polynomial time complexity of O(n^2) while the other algorithm has an exponential time complexity of O(2^n*n)). Looking at the time complexity as the input size gradually increases, we see that the dynamic algorithm will be faster than the exhaustive algorithm.