Project 2: Crane Problem

Andy Ly – Andy_ly01@csu.fullerton.edu

Sadia khan – Sadia.khan@csu.fullerton.edu

Cynthia Zepeda-Rodriguez – cynthia_zr@csu.fullerton.edu

Department of Computer Science, California State University, Fullerton

CPSC 335 Section 08: Algorithms Engineering

May 14, 2023

Exhaustive Algorithm Pseudocode & Time Analysis

```
  assert(setting.rows() > 0) // 3tu
  asser(setting.columns() > 0) // 3tu

  max_steps = setting.rows() + setting.columns() - 2 // 5tu
  assert(max_steps < 64) // 2tu

  path best(setting) // 1tu

  for steps = 0 to max_steps do // ntu
  for path_bits = 0 to pow(2, steps) - 1 do // 2^n tu
    path candidate(setting) // 1tu
    bool valid_path = true // 1tu

    for i = 0 to steps do // mtu
      bool east = (path_bits >> i) & 1 // 3tu
      step_direction dir
    |
      // Block A:
      if east do // 1tu
        dir = STEP_DIRECTION_EAST // 1tu
      else
        dir = STEP_DIRECTION_SOUTH // 1tu
      END IF

      // Block B:
      if candidate.is_step_valid(dir) do // 1tu
        candidate.add_step(dir) // 1tu
      else
        valid_path = false // 1tu
        break
    ENDFOR

    if valid_path and candidate.total_cranes() > best.total_cranes() do // 4tu
      best = candidate // 1tu
    ENDIF
  ENDFOR
  ENDFOR
return best
```

$$Block\ A \ = \ 1 \ + \ max(1, 1) \ = \ 2$$
$$Block\ B \ = \ 1 \ + \ max(1, 1) \ = \ 2$$
$$Block\ C \ = \ 4 \ + \ max(1, 0) \ = \ 5$$

$$S.C = 3 + 3 + 5 + 2 + 1 * n * (1 * 2^n * 1 * 1 * m + 3 + Block\ A + Block\ B) + Block\ C$$

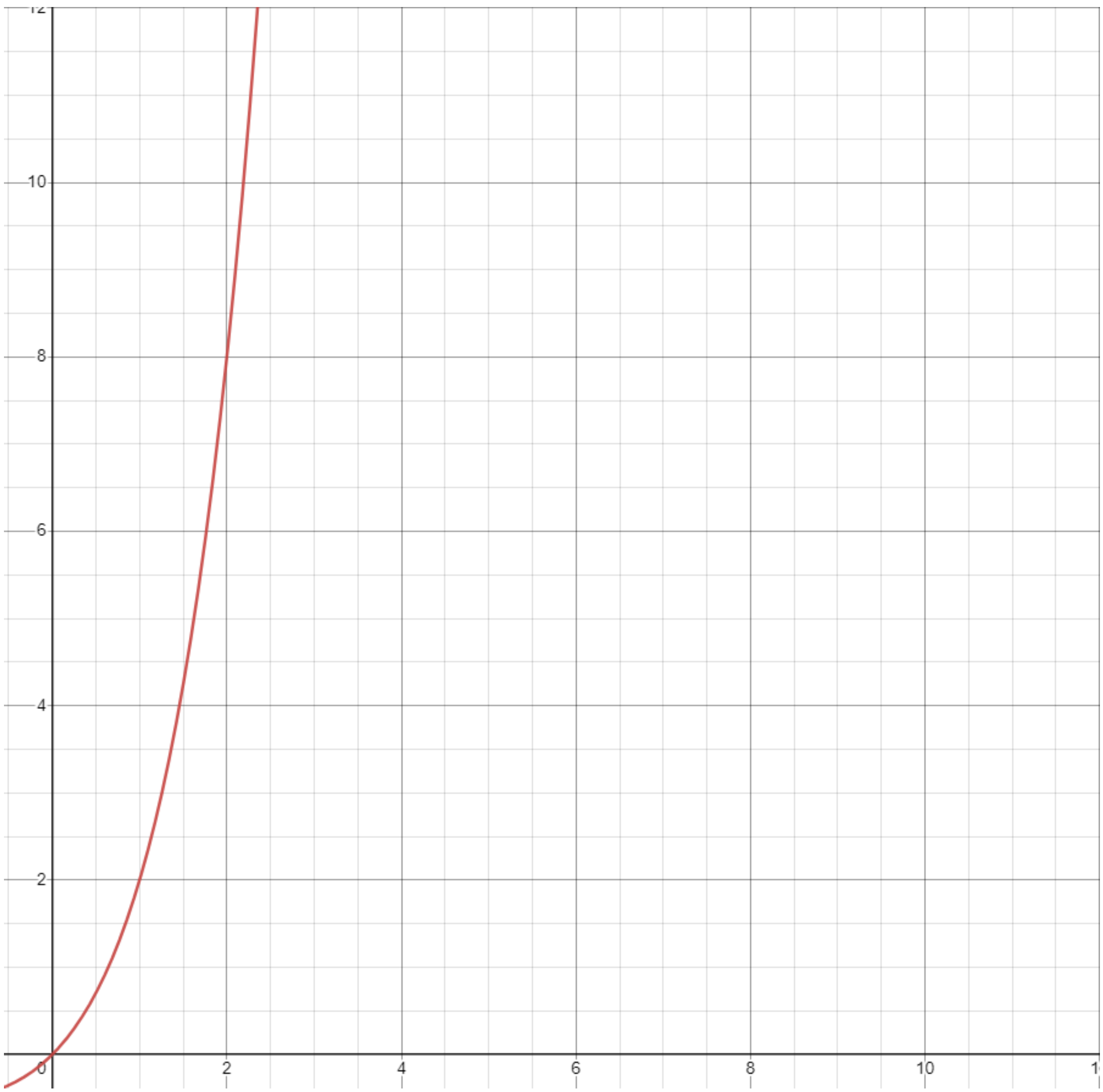
$$S.C = 14 * n * (1 * 2^n * 1 * 1 * m + 3 + 2 + 2) + 5$$

$$S.C = 14n * (2^n * m + 8) + 5$$

Exhaustive Algorithm Graph

Time Vs Input Size

T(n) = n*2^n



Dynamic Algorithm Pseudocode & Time Analysis

```
assert(setting.rows() > 0) // 3tu
assert(setting.columns() > 0) // 3tu

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

for r - 0 to setting.rows() - 1 do // ((n-1)-0+1) = n
  for c = 0 to setting.columns() -1 do // (n-1-0=1) = n

    if setting.get(r,c) == CELL_BUILDING do // 2tu
      A[r][c].reset(); // 2tu
      continue
    ENDIF

    from_above = null // 1tu
    from_left = null // 1tu

    if r != 0 and setting.get(r-1, c) != CELL_BUILDING && A[r-1][c].has_value() do // 8tu
      from_above.emplace(A[r-1][c].value()) // 3 tu
      from_above->add_step(STEP_DIRECTION_SOUTH) // 1tu
    ENDIF // Block A = 8 + 3 + 1 = 12tu

    if c != 0 && setting.get(r, c-1) != CELL_BUILDING && A[r][c-1].has_value() do // 8tu
      from_left.emplace(A[r][c-1].has_value() // 3tu
      from_left->add_step(STEP_DIRECTION_EASTH) // 1tu
    ENDIF // Block B = 8 + 3 + 1 = 12tu

    if from_above.has_value() && from_left.has_value() do // 3tu
      if from_above->total_cranes() >= from_left->total_creanes() do // 3tu
        A[r][c] = from_above //3tu
      else
        A[r][c] = from_left // 3tu
      ENDIF // Block C = 3 + max(3 + max(1, 1), ) = 7tu
    else if from_above.has_value() && !from_left.has_value() do // 4tu
      A[r][c] = from_above // 3tu
    else if !from_above.has_value() && from_left.has_value() do // 4tu
      A[r][c] = from_left // 3tu
    ENDIF
  ENDFOR
ENDFOR
```

```
// post-processing step to find the best path
max_rows = 0 // 1tu
max_columns = 0 // 1tu
max_cranes = 0 // 1tu

for rows = 0 to setting.rows() do // (n - 1 - 0 + 1) = n
  for columns = 0 to setting,columns() do // (n - 1 - 0 + 1) = n
    if A[rows][columns].has_value() && A[rows][columns]->total_cranes > mac_cranes do // 5tu
      max_rows = rows // 1tu
      max_columns = columns // 1tu
      max_cranes = A[rows][columns]->total_cranes() // 3tu
    ENDIF
  ENDFOR
ENDFOR

*best = &A[max_rows][max_columns] // 3tu

assert(best->has_value()) // 2tu
return **best
```

$S.C\ = 3 + 3 + 3 + 2 + 2 + n[n * (2 + max(2 + 1 + 1 + 12 + 12 + 7 + 7 + 7, 0))] + 3 + 10n^2 + 3 + 2$

$S.C = 13 + n[n * (2 + max(49, 0))] + 10n^2 + 8$

$S.C = 13 + n[n * (2 + 49)] + 10n^2 + 8$

$S.C = 13 + n * 51n + 10n^2 + 8$

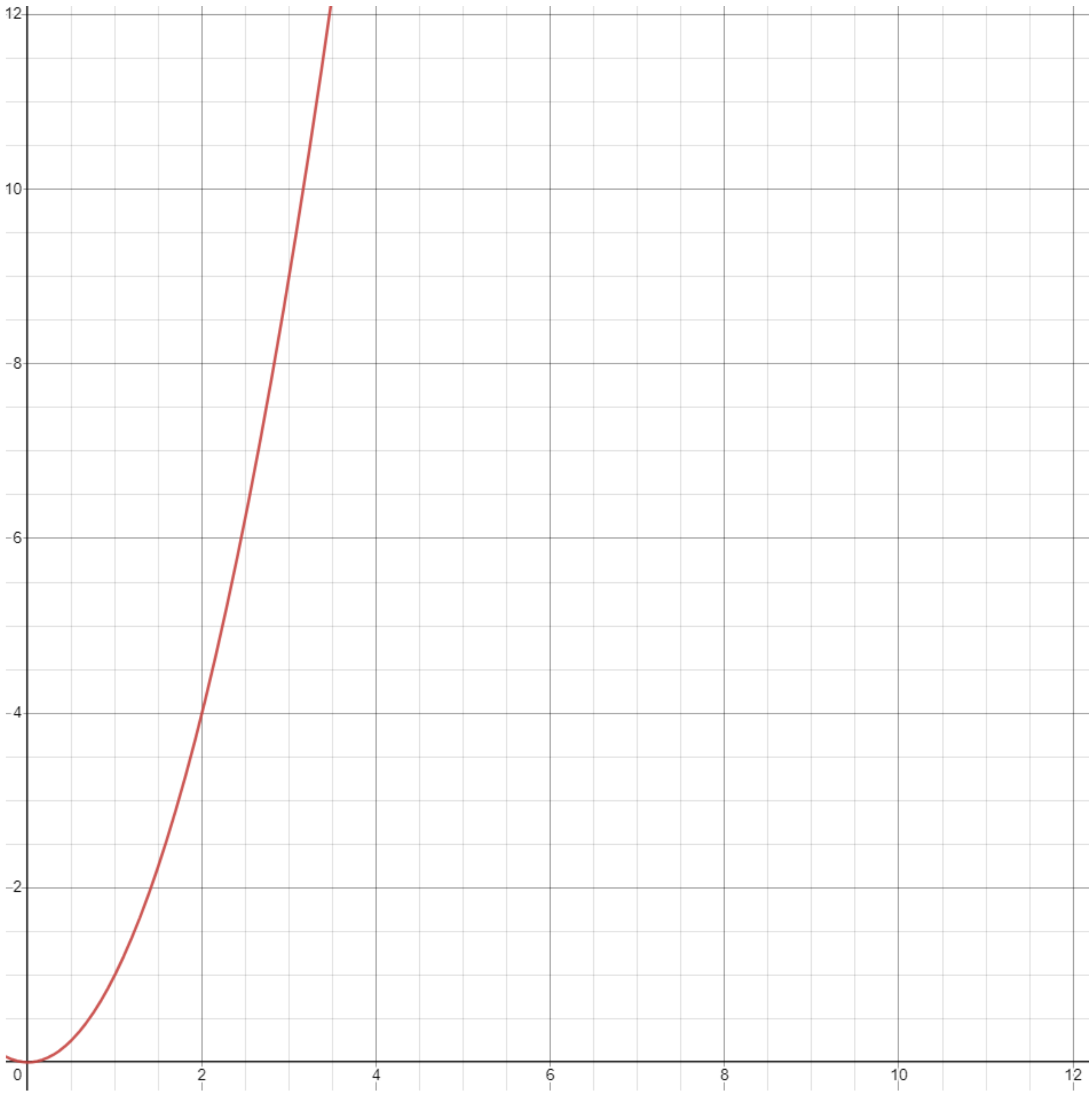
$S.C = 21 + 51n^2 + 10n^2$

$S.C = 61n^2 + 21$

Dynamic Algorithm Graph

Time Vs Input Size

T(n) = n^2



Questions

1. **Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much?**
   Comparing the exhaustive optimization algorithm(crane_unloading_exhaustive) with the dynamic programming algorithm(crane_unloading_dyn_prog), the results show that the dynamic programming algorithm is the faster of the two. The step count for exhaustive optimization resulted in an exponential time complexity O(2^n). With increasing grid size or input, it will cause the runtime of the algorithm to grow exponentially. Whereas for the dynamic algorithm, the step count resulted in a quadratic time complexity O($n^{2,}$) which is the more efficient of the two. As the input size increases the exhaustive optimization algorithm will take exponentially longer than the dynamic programming.
2. **Are your empirical analyses consistent with your mathematical analyses? Justify your answer.**
   Our mathematical analyses are consistent with our empirical analysis. For the exhaustive algorithm, the time complexity of our solution is O(2^n), which is exponential time. Therefore it makes the exhaustive optimization algorithm slow. However, the time complexity of our dynamic programming algorithm is O(n^2), which is much faster, and makes it more efficient.
3. **Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.**
   The evidence is consistent with the hypothesis. Looking at the graphs, we notice that the dynamic programming algorithm can take more instances in less time. On the other hand, the exhaustive optimization algorithm takes longer when it takes fewer instances than the dynamic programming algorithm. We can conclude that the dynamic algorithm is more efficient than the exhaustive algorithm.

4. **Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.**
   N/A