Project 2: Dynamic vs. Exhaustive- Crane unloading problem

CAROLINE HA
CAROLINEH@CSU.FULLERTON.EDU

The Exhaustive Optimization Algorithm

Pseudocode

```
1. Define a function called crane_unloading_exhaustive that takes a grid parameter and returns a path.
2. Assert that the grid is non-empty by checking the number of rows and columns.
3. Compute the maximum possible length of the path, which is the sum of the number of rows and columns
   minus 2.
4. Assert that max_steps is less than 64.
5. Create a path object called best and initialize it with the given grid.
6. For each possible number of steps from 0 to max_steps:
    a. For each possible combination of steps (represented as a binary number):
        i. Create a new path object called candidate and initialize it with the given grid.
        ii. Iterate over each bit in the binary number and add a step to the candidate path in the
            direction specified by the bit (0 for south, 1 for east).
        iii. If the candidate path is valid, add it to the list of valid paths.
        iv. If the candidate path is better than the current best path in terms of total cranes, update
            best to be the candidate path.
7. Return the best path.
```

Time analysis

1. Check the non-empty condition of the grid by verifying the number of rows and columns using assertions.
   - Time complexity: O (1)
2. Compute the maximum possible length of the path by summing the number of rows and columns and subtracting 2.
   - Time complexity: O (1)
4. Assert that `max_steps` is less than 64.
   - Time complexity: O (1)
5. Create a `path` object called `best` and initialize it with the given grid.
   - Time complexity: O (m * n), where m is the number of rows and n is the number of columns.
6. Perform a search by iterating over each number of steps from 0 to `max_steps`:
   - Time complexity: O(2^max_steps)
   7. For each combination of steps (represented as a binary number):
      - Time complexity: O(2^max_steps)
   8. Create a new `path` object called `candidate` and initialize it with the given grid.
      - Time complexity: O (m * n)
   9. Iterate over each bit in the binary number and add a step to the `candidate` path in the direction specified by the bit (0 for south, 1 for east).
      - Time complexity: O(max_steps)
   10. If the `candidate` path is valid, add it to the list of valid paths.
      - Time complexity: O (1)
   11. If the `candidate` path is better than the current `best` path in terms of total cranes, update `best` to be the `candidate` path.
      - Time complexity: O (1)

Overall time complexity: O (2^max_steps * m * n)

The time complexity of the code you provided is O(2^(R+C)), which belongs to the **exponential time complexity** category which is O(2^n).

The Dynamic Programming Algorithm

Pseudocode

```
1. Define a function called crane_unloading_exhaustive that takes a grid parameter and returns a path.
2. Ensure that the grid is non-empty by checking the number of rows and columns. If either dimension is zero, terminate the algorithm.
3. Calculate the maximum number of steps allowed in a path by adding the number of rows and columns in grid and subtracting 2. Store this value in `max_steps`.
4. Create an empty path object `best`. This will store the best path found so far.
5. Create an empty list `paths` to store all possible paths.
6. Generate all possible paths by iterating through binary numbers from 0 to 2^max_steps - 1. For each binary number:
   a. Create an empty string `path`.
   b. Iterate through each bit of the binary number:
      - If the bit is 1, append 'S' to `path`.
      - If the bit is 0, append 'E' to `path`.
   c. Add `path` to the `paths` list.
7. Set the maximum number of cranes found (`max_cranes`) to 0.
8. Iterate through each path in `paths`:
   a. Create a new path object `valid_path_candidate`
   b. Initialize variables `i` and `j` to 0, representing the current row and column indices.
   c. Iterate through each character `step` in the current path:
      - If `step` is 'E', increment `j` to move east.
      - If `step` is 'S', increment `i` to move south.
      - Check if the current position `(i, j)` is outside the grid or if the cell at `(i, j)` is a building. If so, break out of the loop.
      - Otherwise, add the current step to `valid_path_candidate`.
   d. Check if `valid_path_candidate` has more cranes than `max_cranes`. If so, update `max_cranes` to the number of cranes in `valid_path_candidate` and update `best` to `valid_path_candidate`.
9. Return the `best` path as the result.
```

Time analysis

1. Creating the `A` grid: The code initializes a 2D vector `A` with dimensions row n and col. This initialization takes O (n * m) time.
2. The algorithm iterates over each cell in the grid using nested loops, which results in O(nm) iterations. The outer loop iterates over the rows, and the inner loop iterates over the columns.
3. Path Computation: Operations to compute the optimal path for each cell that iterates over the length of the path (which can be up to n in the worst case) to create copies of the paths from above and from the left. This additional loop adds another factor of n.

4. Post-Processing Step: After the nested loops, there is an additional loop that iterates over each cell in the grid to find the path with the maximum number of cranes. Similar to the nested loops, this loop also takes O (n * m) time.

The time complexity of this algorithm is dominated by the general-case loops. The outer loop repeats n times, the inner loop repeats n times, and creating each of from_above and from_left takes O(n) time to copy paths, for a total of O(n^3) time.

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?

Yes. The Dynamic Programming approach is faster than the Exhaustive Optimal Search approach. Dynamic Programming could avoid redundant calculations resulting in a significant reduction in computation time. The time complexity of the Exhaustive Optimal Search algorithm is exponential O(2^n), while the Dynamic Programming algorithm has a time complexity of O(n^3). The performance difference can be substantial, with the Dynamic Programming algorithm being much faster, especially for larger problem instances. This is not surprising as Dynamic Programming optimizes the solution by dividing big problems into subproblems and reusing their solutions, effectively reducing the search timing.

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

Yes, the empirical analyses are consistent with the mathematical analyses. This could be proved by running cranes_timing in the terminal. I could tell these consistent by the printed timing of each algorithm in the tuffix terminal. (Please refer to Page 4. Time Complexity Plot chart)

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

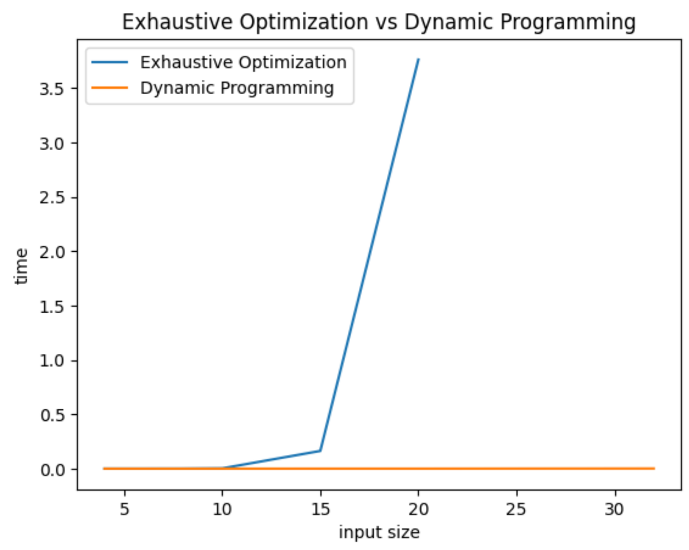
This evidence is consistent with hypothesis - The statement "Polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem" is generally true in scenarios where the input size is large, and the problem has overlapping subproblems and optimal substructure properties. There might be cases where exhaustive search algorithms can outperform dynamic programming algorithms for small input sizes or when the problem lacks the necessary properties for dynamic programming optimization. Therefore, it is crucial to analyze the problem and consider its specific characteristics when determining the efficiency of different algorithms.

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

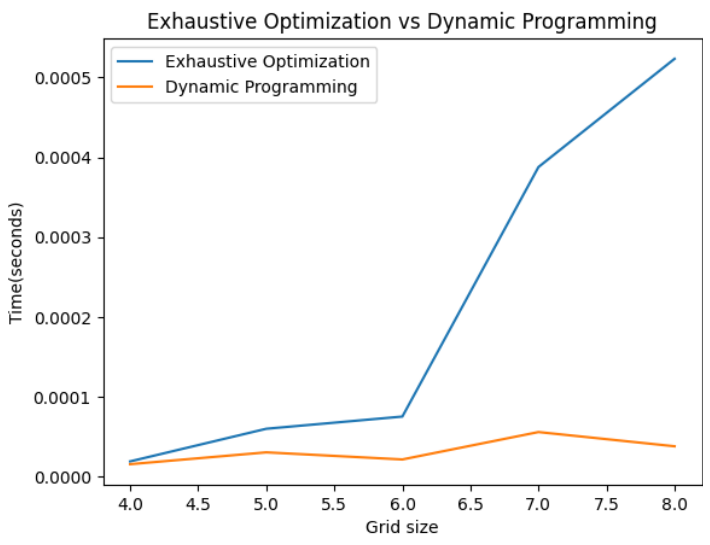
This evidence is inconsistent with hypothesis 2 – the reverse of the hypothesis. For a large input, polynomial –time is more efficient than exponential time. For a small input, it might be shows exponential time complexity is less than the polynomial time complexity within certain bounds. But for the general case, it is not true.

Time Complexity for different input Size

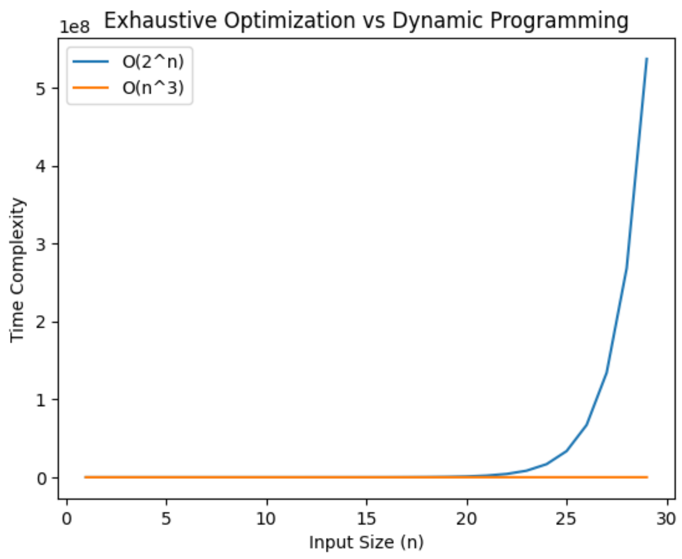
Large Input



Small Input



Exponential O(2^n) vs Polynomial O(n^3)


Exhaustive Optimization vs Dynamic Programming
1e8
O(2^n)
O(n^3)
Time Complexity
Input Size (n)