Ashton Kirkman

12/6/2024

**Project 5 TSP**

**Complexity Analysis:**

Greedy Algorithm: Time Complexity

1. Start at an initial city (city = 0)
2. Initialize an empty set of visited cities and an empty tour O(1)
3. While the tour is incomplete: O(n)
   1. Mark the current city as visited O(1)
   2. Find the nearest unvisited city (looping over all cities) O(n)
   3. Add the nearest city to the tour O(1)

Complexity of Step 3: O(n) \* (O(1) + O(n) + O(1)) = O(n2)

1. Return the tour and its cost O(1)

**Overall time complexity:** O(n2)

**Space complexity:**

Stack – The stack stores partial paths which at worst case can grow to O(n!)

Visited Cities – Each path requires O(n) space to store visited cities

Input Matrix – The input edge cost matrix is O(n2)

**Overall space compexity:** O(n!)

DFS:

1. Initialize a stack with the starting city as the initial path O(1)
2. While the stack is not empty: O(n!)
   1. Pop the top path from the stack O(1)
   2. If the path forms a complete tour: O(1)
      1. Calculate its cost O(n)
      2. If it is better than the best solution so far (BSSF), update the BSSF

O(1)

* 1. If the path is incomplete: O(1)
     1. Expand the current path by adding all unvisited cities O(n)
     2. Push each expanded path onto the stack O(1)

1. Return the BSSF and its cost

**Overall time complexity:** O(n\*n!)

**Space complexity:**

Branch-and-bound:

1. Priority Queue : **Time:** Adding a state to the priority queue takes O(log k ) time where k is the current number of elements in the queue. Extracting from the queue also takes O(log k) time. **Space:** Stores all active states, in the worst case it could be O(n!) for n cities.
2. Reduced Cost Matrix: **Time:** *Initial reduction –* Finds the minimum of each row (O(n)) and subtracts it from all elements (O(n)) and repeat for columns or total of O(n2). *Updating Matrix* – To update a matrix when expanding a state requires marking invalid entries (O(n)) and reducing rows/columns O(n2). This gives total time complexity of O(n2). **Space:** Each matrix has O(n2) space.
3. BSSF Initialization: **Time:** The greedy algorithm iterates over all cities, finding the nearest unvisited city (O(n2)). **Space:** O(n2) for distance matrix and O(n) for the greedy tour.
4. Expanding One Search State: **Time:** For each child you must update the matrix (O(n2)) and calculate the new lower bound (O(n2)). There are O(n – path length) children. Total per state: O(n3) for n – path length children. **Space:** Each child state includes a partial path (O(n)) and a reduced cost matrix (O(n2)) for a total of O(n2)
5. Data Structures for Partial States: **Path:** List of visited cities (O(n)). **Lower Bound:** A single number (O(1)). **Reduced Matrix:** A 2D array(O(n2)).
6. Full Algorithm: **Time:** Priority queue operations - At most O(n!) states are put in the queue and taken out and each operation would have O(log k) time. Total for queue operations: O(n! \* log n!). Reduced cost matrix updates – O(n2) per state expansion. O(n) children per state). Total: **O(n3 \* n!)**. This is the dominant term in the equation. **Space:** Stores up to O(n!) states. Each state has a matrix (O(n2)), path (O(n)), and lower bound (O(1)). Total: **O(n2 \* n!)**.

Branch and Bound Smart:

The “smart” branch and bound algorithm builds upon the standard branch and bound algorithm by employing an optimization strategy to prune states more effectively.

Differences:

1. Priority Queue Optimization:
   1. Instead of being implemented with a simple stack, smart branch and bound uses a priority queue (heapq), prioritizing states based on a composite score (-len(path) and lower\_bound). This encourages deeper paths with promising bounds to be processed earlier. The advantage of this is that it is a more balanced search, exploring deeper paths that could form complete tours sooner. This difference technically does not affect the time and space complexity.

Priority queue data structure: The priority queue is implemented with a **binary min-heap** (heapq) and stores tuples (-len(path), lower\_bound, path, reduced\_matrix).

The priority queue first prioritizes longer lengths which promotes deep searches first, and then prioritizes paths with a lower bound which will have a higher change of finding a tour that is better than BSSF. I was hoping that with this, it would be able to quickly find viable tours without spending so much time looking at paths that are shallow. I think this strategy did work because it was consistently giving me a better cost in roughly the same amount of time as the Branch and Bound algorithm.

**Empirical Results:**

|  |
| --- |
|  |
| **Seed** | **N** | **Random** | **Greedy** | **DFS** | **B&B** | **Smart B&B** |
| 312 | 10 | 3.376 | 3.783 | 3.376 | 3.376 | 3.376 |
| 1 | 15 | 5.134 | 5.131 | inf | 4.993 | 4.93 |
| 2 | 20 | 6.968 | 4.419 | inf | 4.419 | 4.419 |
| 3 | 30 | 12.091 | 7.015 | inf | 7.015 | 6.721 |
| 4 | 50 | 25.85 | 8.715 | inf | 8.715 | 8.715 |
| 5 | 12 | 3.922 | 4.064 | 3.922 | 3.727 | 3.727 |
| 6 | 18 | 5.453 | 3.735 | inf | 3.735 | 3.711 |
| 7 | 22 | 6.954 | 5.221 | inf | 5.221 | 5.121 |