Reflection

Greedy algorithms are a fundamental problem-solving paradigm in computer science, characterized by making locally optimal decisions at each step in the hope of reaching a globally optimal solution.

1. Activity Selection

The classic activity selection problem involves selecting the maximum number of non-overlapping activities, each with a start and finish time. By ordering activities by completion time and always selecting the one that finishes first, the algorithm is simple (due to the ordering, the complexity is O(n log n)) and provably optimal. Proof by contradiction confirms that this greedy strategy does not sacrifice global optimality.

2. Job Scheduling

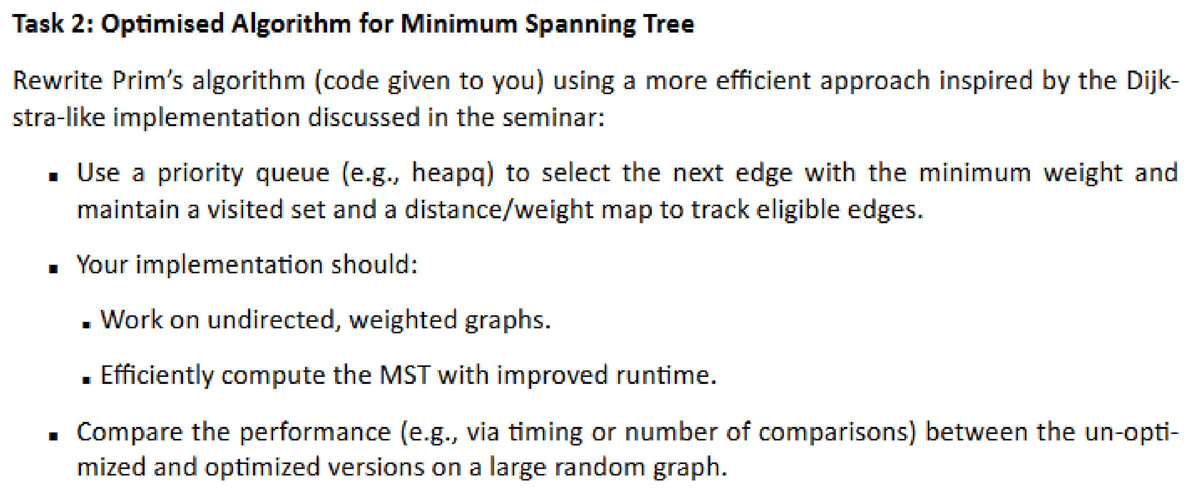
While the image includes "job scheduling," a general greedy approach to solving these problems typically involves ordering jobs by deadline or profit. The system selects the most profitable task, with the goal of maximizing throughput or minimizing latency. This illustrates how greedy techniques can be effective for real-world scheduling problems, although the formal correctness may be more subtle.

3. Huffman Coding

Huffman coding is a greedy algorithm used for lossless data compression. At each step, the two least frequent characters are merged to construct a prefix-free binary tree, thus minimizing the total encoding length. This method ensures optimality for variable-length encoding and is the basis of compression techniques.

4. Minimum Spanning Tree (Prim's Algorithm and Kruskal's Algorithm)

The greedy approach is also the basis of Prim's and Kruskal's algorithms. Both algorithms construct minimum spanning trees on weighted undirected graphs by carefully selecting the edges with the lowest weights. Each step makes the best local choice, either starting from a single node or adding the minimum number of edges between components, and both algorithms obtain the best overall result.



Task 2: Optimised Algorithm for Minimum Spanning Tree

Rewrite Prim's algorithm (code given to you) using a more efficient approach inspired by the Dijk-stra-like implementation discussed in the seminar:

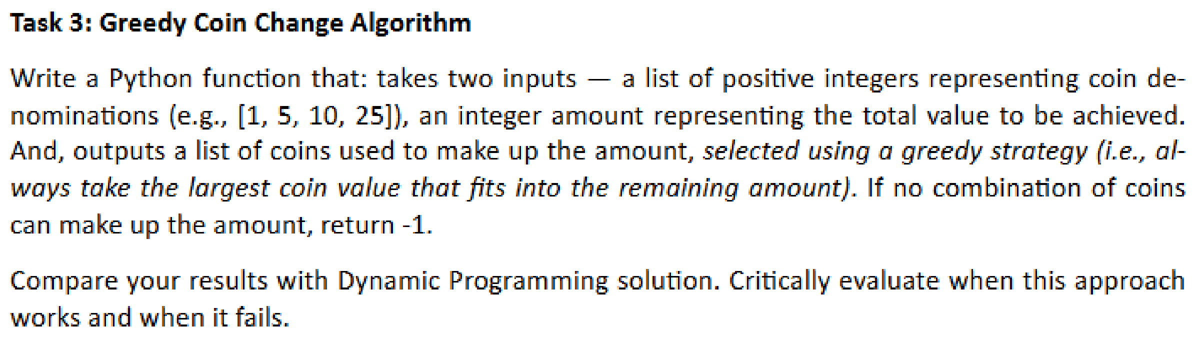
- Use a priority queue (e.g., heapq) to select the next edge with the minimum weight and maintain a visited set and a distance/weight map to track eligible edges.
- Your implementation should:
  - Work on undirected, weighted graphs.
  - Efficiently compute the MST with improved runtime.
- Compare the performance (e.g., via timing or number of comparisons) between the un-opti-mized and optimized versions on a large random graph.

The original Prim's algorithm has a time complexity of $O(V^2)$, where V represents the number of vertices. This quadratic complexity means that the runtime increases quadratically as the graph size increases.

The optimized version reduces the time complexity to $O(E \log V)$, where E represents the number of edges, by introducing a priority queue (min heap). The main cost of the optimized algorithm is increased space complexity, as a priority queue must be maintained.

The main reason for this performance difference is that the original algorithm must traverse all edges of all visited vertices each time it selects the minimum edge. The optimized algorithm, using a priority queue, can retrieve the minimum-weight edge in logarithmic time.

As the graph size increases, the cost of the original algorithm's inner loop increases dramatically, but the optimized algorithm effectively controls this cost through its choice of data structure.



Task 3: Greedy Coin Change Algorithm

Write a Python function that: takes two inputs — a list of positive integers representing coin denominations (e.g., [1, 5, 10, 25]), an integer amount representing the total value to be achieved. And, outputs a list of coins used to make up the amount, *selected using a greedy strategy (i.e., always take the largest coin value that fits into the remaining amount)*. If no combination of coins can make up the amount, return -1.

Compare your results with Dynamic Programming solution. Critically evaluate when this approach works and when it fails.

The greedy algorithm uses an intuitive strategy, always choosing the coin with the largest denomination, to produce the optimal solution. The greedy algorithm has a linear time complexity of O(n) and a constant space complexity of O(1), making it extremely efficient.

However, the greedy algorithm does not always yield the optimal solution. When the coin system does not meet the "standard coin system" condition, that is, when there is no integer multiple relationship between the coin denominations, the greedy algorithm may fail.

For example, in the coin system [1, 5, 10, 21, 25], to make a change of 63 yuan, the greedy algorithm will produce a solution with 5 coins, while the dynamic programming algorithm will find the optimal solution with only 3 21 yuan coins.

The dynamic programming algorithm systematically considers all possible combinations to ensure that the solution with the least number of coins is found. Although dynamic programming can always yield the optimal solution, the computational cost is relatively high, especially for large amounts.