Algorithms and Data structures
Practical 1

Radboud University

Pumping stations

Team-366

Dylan Elens - s1051438

26 October 2023

Introduction

The project involves creating a strategic route to control the water flow in and out of polders, under time constraints. This report outlines our approach to solving this problem using Python and some Dynamic programming techniques, detailing the development and analysis of our algorithm. The focus is on maximizing the efficiency of water removal while adhering to the given time limits, thereby mitigating the risks of flooding.

Algorithm Explanation

Parsing

The ‘parse_input‘ function plays an important role in our algorithm. It begins by reading the number of intersections (‘num_intersections‘), pumping stations (‘num_pumping_stations‘), roads (‘num_roads‘), and the time limit (‘time_limit‘) from the input. These values are critical for constructing the graph that represents our problem space.

Next, we initialize the ‘graph‘ as a dictionary, where each intersection is a key, and its value is another dictionary representing connected nodes (other intersections) and their respective travel times. This structure is vital for representing the bidirectional roads between intersections.

The function then reads the IDs of the pumping stations and stores them in ‘pumping_station_ids‘. We adjust these IDs by subtracting one to align with Python’s zero-indexing. The starting point (ID 0) is also added to this list to ensure the algorithm considers the starting location.

For each road, the function reads its start and end points (‘start‘, ‘end‘) and the travel time (‘travel_time‘). These roads are added to the ‘graph‘, considering that they are bidirectional. Each road’s travel time is set as the value connecting the two intersections in the graph.

Lastly, the function utilizes the ‘dijkstra‘ function to determine the shortest paths from each pumping station to every other intersection. This is crucial for identifying the most efficient routes in our water pumping station management problem. The ‘dijkstra‘ function itself implements Dijkstra’s algorithm, a well-known algorithm for finding the shortest paths between nodes in a graph. It uses a priority queue (‘heapq‘) to efficiently find the node with the smallest distance, then iteratively updates the distances to each neighbor of this node.

The result of the ‘parse_input‘ function is a graph that fully represents our problem’s landscape, a time limit for the operations, and a list of pumping stations, all of which are

essential for the subsequent steps of the algorithm.

The ‘graph_to_distances_array‘ converts the graph dictionary into a 2D list (‘graph_list‘) where each list element represents the distances between intersections. The size of this list is determined by the largest key in the graph dictionary, ensuring coverage of all intersections.

We initialize ‘graph_list‘ with high values, symbolizing no direct path between intersections initially. Then, we iterate over each entry in the graph dictionary, updating ‘graph_list‘ with actual travel times. This process effectively transforms our graph into a matrix format, which is more suitable for certain types of algorithmic processing.

Solving

The held_karp_modified function in our project is a strategic variation of the Held-Karp algorithm, specifically designed to tackle a Set Traveling Salesman Problem (Set TSP) in the context of efficient water management. Unlike traditional algorithms that prioritize finding the fastest route, our algorithm aims to maximize the volume of water pumped during a flood. By analyzing the network of pumping stations, our algorithm calculates not just the shortest path, but the most effective route in terms of water management, focusing on the stations that contribute most significantly to alleviating flood conditions. This approach ensures that our solution addresses the primary objective of flood mitigation through optimal water pumping.

Initially, it sets up a dictionary ‘C‘ to store the minimum cost to reach each node from a given subset of nodes. The algorithm iteratively increases the subset size, calculating the cost of reaching each node within the subset by considering all possible paths that lead to it. This is done by looking at the cost of reaching the previous node in the subset and adding the cost of traveling from there to the current node.

To ensure the solution adheres to the time constraint (‘time_limit‘), the function incorporates a check for each path, only considering those that fall within the time limit. The ‘calculate_edge_value‘ function is used to calculate the cost of each edge based on the arrival time and time limit.

Finally, the function reconstructs the optimal path by starting from the full set of nodes (excluding the starting node) and repeatedly selecting the node that can be reached at the minimum additional cost. The best path will be returned. However this might not always the the best order to travel them in.

In the `water_pumped` function, we calculate the total volume of water pumped (W)

using the following formula:

$$W = \sum_{i=0}^{n-2} \left[\max(0, (T - t_i - d_{a_i,a_{i+1}} - 10)) \times R\right]$$

Where:

- n is the number of nodes in the path.
- a_i and a_{i+1} are consecutive nodes in the path.
- $d_{a_i,a_{i+1}}$ is the distance between nodes a_i and a_{i+1}.
- T is the time limit.
- t_i is the cumulative time spent up to node a_i.
- R is the rate of pumping (200 units per time unit).
- The term $\max(0, (T - t_i - d_{a_i,a_{i+1}} - 10))$ calculates the remaining time available for pumping at node a_{i+1}, ensuring it's not negative.

Correctness of the Algorithm

The held_karp_modified is not always correct. This is because the best path found path might be fully exist in our timelimit. However thankfully we do get the minimum amount of nodes required to correct this I created a function that for the possible paths tries each possibility to unill it finds the best one. Thankfully we are somewhat guarenteed to have a small end path since there a maximum amount pumping stations which is relatively small. And it will mostly get decreased by the amount of by main solver.

Running Time Complexity

The runtime complexity of the `dijkstra` function can be defined formally as $O((V + E) \log V)$, where V is the number of vertices (nodes) in the graph, and E is the number of edges (connections between nodes).

The runtime complexity of the `parse_input` function can be assessed by considering the operations involved:

- Reading input parameters and initializing the graph requires $O(V)$ time, where V is the number of intersections (nodes), represented by `num_intersections` in the code.

- Constructing the list of pumping stations, `pumping_station_ids`, takes $O(P)$ time, with P being the number of pumping stations, distinguishing them from regular intersections.
- The loop for reading road data and updating the graph executes E times, where E is the number of roads (`num_roads`). Each iteration takes constant time, leading to $O(E)$ complexity for this step.
- The loop where Dijkstra's algorithm is run for each pumping station has a complexity of $O((V + E) \log V)$ per run. Executed P times, this step results in $O(P \cdot (V + E) \log V)$ complexity.

Thus, the overall complexity of `parse_input` is $O(V + P + E + P \cdot (V + E) \log V)$.

held-karp analysis

The `held_karp_modified` function implements a dynamic programming approach, leading to a significant computational complexity. Specifically:

- The first loop initializes the cost to reach each node from the start node, running $O(P)$ times, where P is the number of nodes to visit.
- The second loop, generating all combinations of subsets and finding the minimum cost path, is the most computationally intensive. For each subset of size s, it iterates over all combinations, which is $O(2^P \cdot P^2)$ in the worst case.
- Within this loop, each combination requires updating costs based on previously computed values, contributing to the overall complexity.

Thus, the total runtime complexity of `held_karp_modified` is $O(P + 2^P \cdot P^2)$.

water calculations

The `water_pumped` function has a linear runtime complexity, denoted as $O(N)$, where N is the number of nodes in the path. The function iterates over the path once, performing constant-time operations at each step.

The `find_max_water_pumped` function involves finding permutations of the path, leading to a factorial runtime complexity, denoted as $O(N!)$. It iterates over all permutations of the path (excluding the first node) and calls the `water_pumped` function for each permutation. The complexity of this function is significantly higher due to the factorial nature of permutations.

Main Execution Block Analysis

The main execution block, encapsulated within `if __name__ == "__main__":`, orchestrates the workflow of the algorithm:

- `parse_input()` is called to construct the graph and retrieve parameters like `time_limit` and `pumping_station_ids`. This function's complexity is $O(V + P + E + P \cdot (V + E) \log V)$.
- `graph_to_distances_array(matrix)` converts the graph into a distance matrix, with a complexity of $O(V^2)$.
- `held_karp_modified(matrix, pumping_station_ids)` computes the optimal path, with a complexity of $O(P + 2^P \cdot P^2)$.
- `find_max_water_pumped(path, time_limit, matrix)` identifies the most effective water pumping path, with a factorial complexity of $O(N!)$.