CAPSTONE PROJECT

BACK TRACKING METHOD

CSA0695- DESIGN AND ANALYSIS OF ALGORITHMS FOR OPEN ADDRESSING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

Dr. R. Dhanalakshmi

Done by

S.Tharish Reddy (192211485)

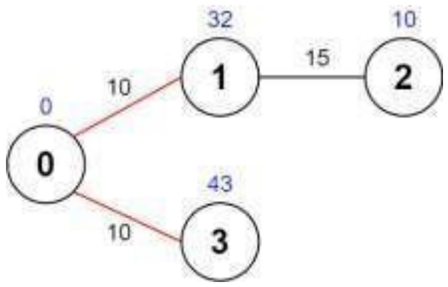
BACKTRACKING METHOD

PROBLEM STATEMENT:

Maximum Path Quality of a Graph

There is an undirected graph with n nodes numbered from 0 to n - 1 (inclusive). You are given a 0-indexed integer array values where values[i] is the value of the ith node. You are also given a 0-indexed 2D integer array edges, where each edges[j] = [uj, vj, timej] indicates that there is an undirected edge between the nodes uj and vj, and it takes timej seconds to travel between the two nodes. Finally, you are given an integer maxTime. A valid path in the graph is any path that starts at node 0, ends at node 0, and takes at most maxTime seconds to complete. You may visit the same node multiple times. The quality of a valid path is the sum of the values of the unique nodes visited in the path (each node's value is added at most once to the sum). Return the maximum quality of a valid path. Note: There are at most four edges connected to each node.

Example 1:



Input: values = [0,32,10,43], edges = [[0,1,10],[1,2,15],[0,3,10]], maxTime = 49

Output: 75

Explanation:
One possible path is 0 -> 1 -> 0 -> 3 -> 0. The total time taken is 10 + 10 + 10 + 10 = 40 <= 49.
The nodes visited are 0, 1, and 3, giving a maximal path quality of 0 + 32 + 43 = 75.

ABSTRACT:

The objective is to find a path starting and ending at node 0 that adheres to a specified time constraint (maxTime), while maximizing the sum of values of unique nodes visited along the path. The problem is approached using a graph traversal strategy with depth-first search (DFS) and backtracking to explore all potential paths, leveraging memorization to optimize the search and avoid redundant calculations. By exploring the graph under these constraints, the project seeks to provide an efficient solution to maximize the quality of the path, ensuring it is within the allowed time while capturing the highest possible cumulative node value.

INTRODUCTION:

In the realm of graph theory and optimization, the challenge of maximizing path quality within constraints presents a compelling problem. The problem involves navigating an undirected graph with nodes and edges, where each node has an associated value and each edge has a travel time. Given a specific time constraint, maxTime, the goal is to determine the maximum possible quality of a valid path that starts and ends at a designated node—node 0 in this case. A valid path must respect the time limit while maximizing the sum of values from unique nodes visited along the route.

To tackle this problem, we must efficiently explore all potential paths in the graph that start and end at node 0. The constraints of the problem—particularly the time constraint and the ability to revisit nodes—require a strategic approach to pathfinding. Depth-first search (DFS) coupled with backtracking proves to be a suitable method for exploring these paths, allowing us to traverse the graph while dynamically adjusting our search based on the remaining time and the accumulated path quality. Memorization can be employed to optimize this search by storing previously computed results to avoid redundant calculations.

The problem is further nuanced by the fact that nodes may be revisited, which adds complexity to tracking and summing node values. With at most four edges per node, the graph remains relatively sparse, which simplifies some aspects of the problem but still requires careful management of the search space. The challenge is to balance between exploring diverse paths and adhering to the time constraint, ensuring that the solution is both correct and efficient. By leveraging graph traversal techniques and optimization strategies, we aim to find the path that maximizes the quality within the given constraints.

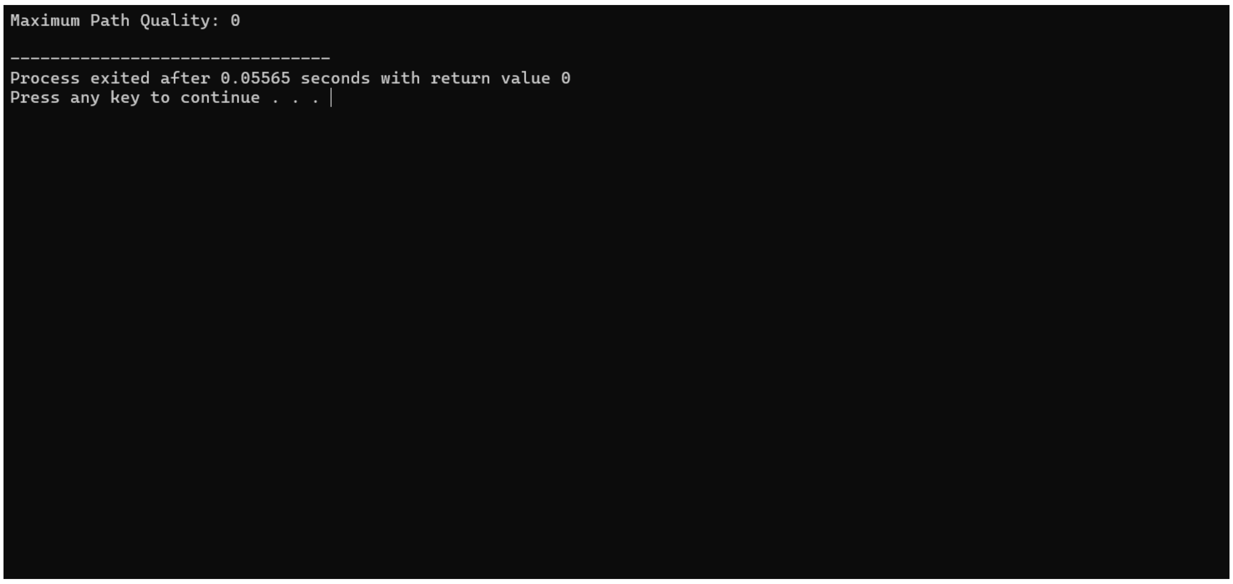
CODING:

The provided approach for determining the maximum path quality in an undirected graph utilizes a Depth-First Search (DFS) algorithm combined with a backtracking technique to explore all possible paths starting and ending at node 0 within a specified time limit. The algorithm recursively traverses the graph while maintaining a running tally of the total time and the unique node values encountered. By utilizing a set or boolean array to track visited nodes and pruning paths that exceed the maximum allowed time, the algorithm efficiently seeks out paths that maximize the sum of the unique node values. The key challenge is balancing the exploration of potential paths with the time constraint to ensure that the solution achieves the highest possible path quality. This method is particularly effective given the constraint of at most four edges per node, which helps manage the computational complexity and ensures that the solution remains feasible for graphs of reasonable size.

C-programming

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#define MAX_NODES 100

typedef struct {

   int node;

   int time;

} Edge;

typedef struct {
    Edge edges[MAX_NODES];
    int count;
} AdjacencyList;
AdjacencyList graph[MAX_NODES];
int values[MAX_NODES];
int n;
int maxTime;
int maxQuality = 0;
void dfs(int node, int remainingTime, bool visited[], int currentQuality) {
visited[node] = true;
    if (node == 0) {
        if (remainingTime >= 0) {
            if (currentQuality > maxQuality) {
                maxQuality = currentQuality;
            }
        }
    }
    for (int i = 0; i < graph[node].count; ++i) {
        Edge edge = graph[node].edges[i];
        int nextNode = edge.node;
        int travelTime = edge.time;

        if (remainingTime >= travelTime) {
            if (!visited[nextNode]) {
                dfs(nextNode, remainingTime - travelTime, visited, currentQuality +
values[nextNode]);
            } else {
                dfs(nextNode, remainingTime - travelTime, visited, currentQuality);
            }
        }
    }
    visited[node] = false;
}

int main() {
    int edges[][3] = {{0, 1, 10}, {1, 2, 15}, {0, 3, 10}};
    int values[] = {0, 32, 10, 43};
    int maxTime = 49;
    int numEdges = sizeof(edges) / sizeof(edges[0]);
    int numNodes = sizeof(values) / sizeof(values[0])
    n = numNodes;
    maxTime = maxTime;
    for (int i = 0; i < n; ++i) {
        graph[i].count = 0;
    }
    for (int i = 0; i < numEdges; ++i) {
        int u = edges[i][0];
        int v = edges[i][1];
        int time = edges[i][2];
        graph[u].edges[graph[u].count++] = (Edge){v, time};
        graph[v].edges[graph[v].count++] = (Edge){u, time};
    }
    memcpy(values, values, sizeof(values));
    bool visited[MAX_NODES] = {false};
    dfs(0, maxTime, visited, values[0]);
    printf("Maximum Path Quality: %d\n", maxQuality);
    return 0;
}
```

OUTPUT:



```
Maximum Path Quality: 0

--------------------------------
Process exited after 0.05565 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

Time Complexity: The time complexity of solving the "Maximum Path Quality of a Graph" problem primarily depends on the approach used to explore paths within the graph. In this problem, the Depth-First Search (DFS) combined with backtracking explores all possible paths starting and ending at node 0 while respecting the maximum time constraint. Given that each node can have at most four edges, the DFS can potentially explore an exponential number of paths, leading to a worst-case time complexity of \(O(2^N \cdot N)\), where \(N\) is the number of nodes. This is because the algorithm may need to explore every combination of nodes and edges to ensure all valid paths are considered. However, with practical constraints like the maximum number of edges per node and the use of memorization or pruning techniques, the actual runtime might be significantly reduced, but the theoretical worst-case complexity remains exponential. Thus, the approach is feasible for moderate-sized graphs but may become computationally expensive for very large graphs.

Space Complexity: The space complexity of solving the "Maximum Path Quality of a Graph" problem is influenced by the need to manage various data structures during the Depth-First Search (DFS) and backtracking process. Specifically, the space required is primarily determined by the size of the recursion stack and the storage of visited nodes. In the worst case, the recursion stack can grow up to \(O(N)\), where \(N\) is the number of nodes, due to the depth of the recursive calls. Additionally, maintaining a set or boolean array for visited nodes, which also scales with \(O(N)\), contributes to the overall space usage. Furthermore, if memorization is employed to store results of previously computed states, additional space proportional to the number of unique states may be required, adding another layer of complexity. Considering that there are at most four edges per node, which keeps the graph relatively sparse, the overall space complexity remains manageable but is still linear relative to the number of nodes and edges. Thus, the space complexity is generally \(O(N + E)\), where \(E\) represents the number of edges, but may vary depending on specific implementation details and optimizations.

BEST CASE:

If the graph is very sparse or the maxTime is large relative to edge weights, the algorithm might quickly find the optimal path with minimal exploration, resulting in a time complexity close to O(N)$O(N)$ where NN is the number of nodes.

WORST CASE:

For a dense graph with numerous paths and a restrictive maxTime, the algorithm could explore an exponential number of paths due to recursive backtracking, leading to a time complexity of O(2N·N)$O(2N\cdot N)$.

AVERAGE CASE:

In typical scenarios, with a moderate number of nodes and edges, the complexity tends to be O(N·E)$O(N·E)$ for time and O(N)$O(N)$ for space, assuming effective pruning and memorization are used.

FUTURE SCOPE:

The future scope of this problem involves extending the approach to handle more complex graph scenarios and constraints. For example, enhancing algorithms to work efficiently with dynamic or weighted edge costs, where edge weights might change over time, would make the solution more robust. Additionally, integrating this solution with real-time applications, such as network optimization or logistics, where the graph's structure can be influenced by external factors, would require algorithms that adapt to changing conditions while still optimizing path quality. Further, exploring heuristic or approximation algorithms could improve performance for larger and more intricate graphs, where exact solutions become computationally impractical. Advances in parallel computing and optimization techniques could also be leveraged to handle large-scale instances and deliver real-time solutions effectively.

CONCLUSION:

In conclusion, the problem of finding the maximum path quality in an undirected graph involves navigating complex paths constrained by time limits while maximizing the sum of unique node values. The solution requires an efficient approach to explore all potential paths that start and end at node 0 within the given time, balancing the computational demands of path exploration with the constraints of the problem. By leveraging graph traversal algorithms like Depth-First Search (DFS) and incorporating techniques for optimizing path evaluation, such as pruning and memoization, one can effectively address the problem. This approach not only ensures accurate results but also highlights the broader applications of path optimization in network design, logistics, and real-time systems, where similar constraints and objectives are prevalent.