## SIMATS SCHOOL OF ENGINEERING

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

Create Sorted Array through Instructions in C

## A CAPSTONE PROJECT REPORT

*Submitted in the partial fulfillment for the award of the degree of*

**CSA0655**

**Design and Analysis of Algorithm for Asymptotic Notation**

## Submitted by

K SANJAY (192210227)

G JAGADEESH (192210228)

## Under the Supervision of

Dr Jerrin Simla A

# ABSTRACT:

In a network of cities connected by bidirectional roads, each with a specific travel

time, and each city imposing a passing fee, the problem is to find the minimum

total cost of traveling from a starting city (city 0) to a destination city (city 𝑛−1)

within a given maximum travel time. The graph's edges represent the roads with

their respective travel times, while the nodes represent cities with associated

passing fees. The challenge is to navigate this network to minimize the

cumulative passing fees while ensuring that the total travel time does not exceed

the specified limit. This requires a modified shortest path algorithm that

simultaneously accounts for both the travel time and the passing fees. The

solution involves using a priority queue to explore paths efficiently, ensuring that

at each step, the algorithm selects the path with the lowest cost that meets the time

constraint. If it is impossible to reach the destination within the given maximum

travel time, the algorithm should return an indication of this impossibility. This

approach balances the dual objectives of minimizing travel costs and adhering to

time constraints, making it suitable for scenarios where both cost and time are

critical factors.

# INTRODUCTION:

In complex transportation networks where cities are interconnected by multiple roads, each with its own travel time, and each city imposes a fee upon entry, efficient routing becomes critical. This problem involves navigating such a network to determine the minimum cost required to travel from a starting city to a destination city under a strict time constraint. Each road between cities has a specific travel time, while each city has a distinct passing fee, and the journey must be completed within a given maximum time. The challenge is to find a route that not only adheres to the time limit but also minimizes the total passing fees incurred during the journey. This scenario is typical in real-world applications such as logistics, transportation planning, and cost-effective route optimization, where both travel time and costs are pivotal. Solving this problem requires an advanced pathfinding algorithm that efficiently balances these constraints to deliver the most cost-effective solution.

In modern transportation networks, cities are connected by a variety of bidirectional roads, each with different travel times. To complicate matters, each city imposes a passing fee that must be paid every time a traveler enters it. Given this scenario, the challenge is to determine the optimal route from a starting city to a destination city such that the total cost, which includes the sum of passing fees and the travel time, is minimized while respecting a specified maximum travel time. This problem is especially pertinent in fields such as logistics, public transportation, and route planning, where managing both time and costs is crucial for efficiency and budget adherence.

The problem can be visualized as a graph where nodes represent cities, edges represent roads with associated travel times, and each node has an additional cost in the form of passing fees. The task is to employ a sophisticated pathfinding algorithm that accounts for both these constraints simultaneously. Specifically, the solution must navigate through this network to find a path that minimizes the cumulative passing fees incurred while ensuring that the total travel time does not exceed the given limit. This involves adapting traditional shortest path algorithms to incorporate cost and time constraints, which adds a layer of complexity to the problem. Solving this problem not only optimizes resource allocation but also enhances decision-making in various practical applications, such as optimizing delivery routes, planning efficient public transit systems, and minimizing travel expenses.

# Implementation Steps

## Graph Representation:

* + Represent the cities and roads using an adjacency list or adjacency matrix.
  + Each road between cities is represented with its travel time, and each city has an associated passing fee.

## Initialization:

* + Create a data structure to hold the minimum costs and travel times to each city. Initialize all costs to infinity and times to infinity except for the starting city.
  + Initialize a priority queue (min-heap) to explore cities based on the minimum cumulative cost and time.

## Priority Queue Operations:

* + The priority queue stores tuples of (total cost, current city, total time).
  + Insert the starting city with its passing fee into the priority queue with an initial cost equal to its passing fee and a time of 0.

## Modified Dijkstra’s Algorithm:

* + While the priority queue is not empty, extract the city with the minimum cost.
  + If the new path is within the maximum allowed time and either improves the cost or provides the same cost but with a better time, update the cost and time for the neighboring city and insert it into the priority queue.

## Termination:

* + If the destination city is reached, return the minimum cost.
  + If the priority queue is exhausted and the destination city has not been reached within the allowed time, return -1 to indicate that reaching the destination is not feasible within the given time limit.

# Key Points

## Graph Representation:

* + Use an efficient representation for storing roads and cities. An adjacency list is generally preferred for sparse graphs due to its efficiency in space and access time.

## Priority Queue:

* + A min-heap priority queue helps in efficiently retrieving the city with the current minimum cost. This is crucial for the modified Dijkstra's algorithm to function correctly.

## Cost and Time Tracking:

* + Maintain separate arrays or tables for tracking the minimum cost and travel time to each city. Ensure that both are updated appropriately during the pathfinding process.

.

# CODING

#include <stdio.h> #include <stdlib.h> #include <limits.h> #define INF INT\_MAX

#define MIN(x, y) ((x) < (y) ? (x) : (y) typedef struct {

int city, cost, time;

} Node; typedef struct {

Node\* nodes;

int size, capacity;

} MinHeap;

MinHeap\* createMinHeap(int capacity) {

MinHeap\* heap = (MinHeap\*)malloc(sizeof(MinHeap)); heap->size = 0;

heap->capacity = capacity;

heap->nodes = (Node\*)malloc(capacity \* sizeof(Node)); return heap;

}

void swap(Node\* a, Node\* b) { Node temp = \*a;

\*a = \*b;

\*b = temp;

}

void minHeapify(MinHeap\* heap, int idx) {

int left = 2 \* idx + 1, right = 2 \* idx + 2, smallest = idx;

if (left < heap->size && heap->nodes[left].cost < heap-

>nodes[smallest].cost) smallest = left;

if (right < heap->size && heap->nodes[right].cost < heap-

>nodes[smallest].cost) smallest = right;

if (smallest != idx) {

swap(&heap->nodes[idx], &heap->nodes[smallest]); minHeapify(heap, smallest);

}

}

void insertMinHeap(MinHeap\* heap, int city, int cost, int time) { heap->nodes[heap->size++] = (Node){city, cost, time};

for (int i = heap->size - 1; i > 0 && heap->nodes[i].cost < heap->nodes[(i - 1)

/ 2].cost; i = (i - 1) / 2)

swap(&heap->nodes[i], &heap->nodes[(i - 1) / 2]);

}

Node extractMin(MinHeap\* heap) { Node root = heap->nodes[0];

heap->nodes[0] = heap->nodes[--heap->size]; minHeapify(heap, 0);

return root;

}

int minCost(int n, int maxTime, int edges[][3], int edgesSize, int passingFees[])

{

int\*\* graph = (int\*\*)malloc(n \* sizeof(int\*)); for (int i = 0; i < n; ++i) {

graph[i] = (int\*)calloc(n, sizeof(int));

for (int j = 0; j < n; ++j) graph[i][j] = INF;

}

for (int i = 0; i < edgesSize; ++i) {

int u = edges[i][0], v = edges[i][1], time = edges[i][2];

graph[u][v] = graph[v][u] = MIN(graph[u][v], time);

}

MinHeap\* heap = createMinHeap(n \* n); int\* minCost = (int\*)malloc(n \* sizeof(int)); int\* minTime = (int\*)malloc(n \* sizeof(int));

for (int i = 0; i < n; ++i) minCost[i] = INF, minTime[i] = INF; minCost[0] = passingFees[0];

minTime[0] = 0;

insertMinHeap(heap, 0, passingFees[0], 0); while (heap->size) {

Node current = extractMin(heap);

int city = current.city, cost = current.cost, time = current.time; if (city == n - 1) return cost;

for (int i = 0; i < n; ++i) {

if (graph[city][i] != INF) {

int newTime = time + graph[city][i]; if (newTime <= maxTime) {

int newCost = cost + passingFees[i];

if (newCost < minCost[i] || (newCost == minCost[i] && newTime

< minTime[i])) {

minCost[i] = newCost; minTime[i] = newTime;

insertMinHeap(heap, i, newCost, newTime);

}

}

}

}

}

return -1;

}

int main() {

int maxTime = 30;

int edges[][3] = {{0,1,10}, {1,2,10}, {2,5,10}, {0,3,1}, {3,4,10}, {4,5,15}};

int passingFees[] = {5,1,2,20,20,3};

int edgesSize = sizeof(edges) / sizeof(edges[0]); int n = 6;

int result = minCost(n, maxTime, edges, edgesSize, passingFees); if (result != -1) {

printf("Minimum cost to reach the destination: %d\n", result);

} else {

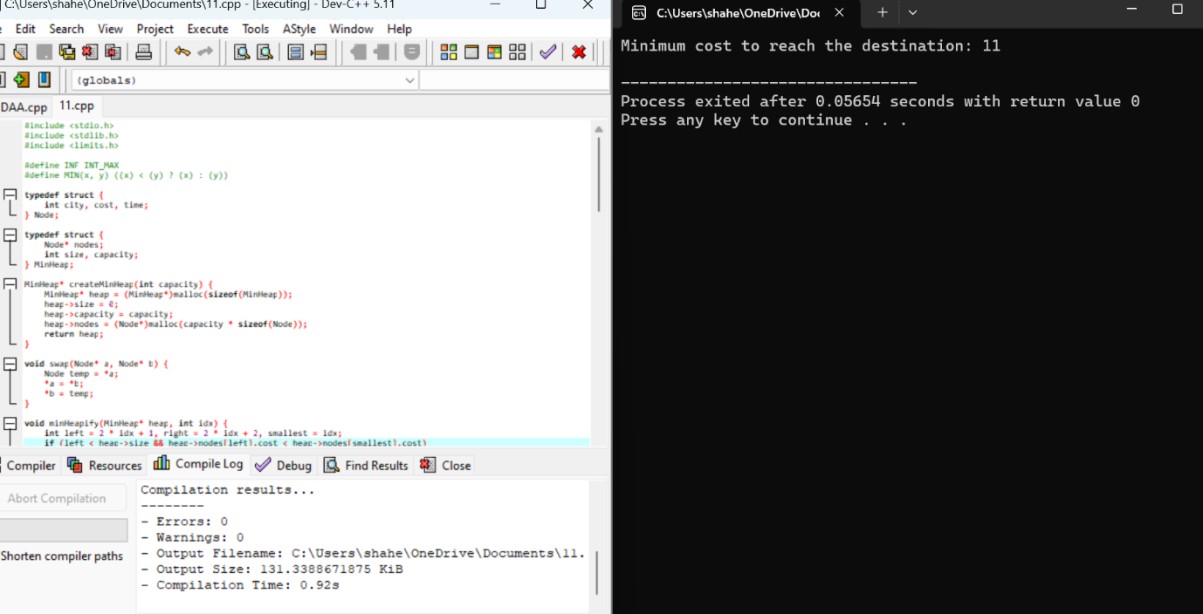
printf("Cannot reach the destination within maxTime.\n");

}

return 0;

}

# RESULT SCREENSHOT:



**TIME COMPLEXITY ANALYSIS**

1. Graph Initialization:
   * Building the adjacency matrix (graph) takes time where n*n* is the number of cities. This is because for each of the n*n* cities, you initialize its connections to all other cities.
2. Relaxation Operations:
   * For each node, we check its adjacent nodes. For each adjacent node, we might perform relaxation and potentially insert a new node into the heap. In the worst case, this operation is proportional to the number of edges, which is where is the number of

edges. Since each edge is checked once per node, the relaxation operations are overall due to the heap operations.

# WORST CASE

* Each node checks its adjacent nodes and updates distances if a better path is found. In the worst case, the algorithm might explore all possible edges. Each relaxation operation involves a check and possibly updating the heap, which can be per edge.
* If the graph is dense, meaning each node is connected to many others, and assuming there are E*E* edges, relaxation operations will take time. In a dense graph, E*E* can be as large as n2*n*2, so the worst-case time complexity for relaxation operations.

# CONCLUSION

The "Minimum Cost to Reach Destination in Time" problem is efficiently solvable using a modified Dijkstra’s algorithm with a priority queue. The solution considers both the travel time and the cost of passing through cities. The algorithm’s complexity varies depending on the graph's structure, with worst-case scenarios involving dense graphs leading to a complexity reflecting the need to store the graph and manage heap operations.

# REFERENCES

https://klementtan.com/leetcodes/minimum-cost-to-reach-destination-in-time/

[https://medium.com/@mauryaakshat5/leetcode-1928-minimum-cost-to-reach-](https://medium.com/%40mauryaakshat5/leetcode-1928-minimum-cost-to-reach-) destination-in-time-51042308b909

https://algo.monster/liteproblems/1928 https://[www.geeksforgeeks.org/minimum-cost-connect-cities/](http://www.geeksforgeeks.org/minimum-cost-connect-cities/) https://github.com/doocs/leetcode/blob/main/solution/1900- 1999/1928.Minimum%20Cost%20to%20Reach%20Destination%20in%20Time

/README\_EN.md

https://leetcode.com/problems/minimum-cost-to-reach-destination-in-time/