School of Engineering and Applied Science (SEAS), Ahmedabad University

CSE400: Fundamentals of Probability in Computing

Group Name: ITS-S1
Team Members:

- Dev Kansara (AU2340222)
- Param Shah (AU2340192)
- Hir Gaglani (AU2340136)
- Priscilla R (AU2340001)
- Hir Vora (AU2340156)

I. Background and Motivation

..1 Background

Our project is to explore the randomized route considering the travel time. There are several algorithms such as A* and Dijkstra's which aims to find the shortest path in terms of the spatial distance. But, the concern arises when travel time depends on the certain parameters like signal delays, traffic congestion, weather conditions and etc. During this uncertainties we need optimized algorithms which computes the route that minimize the actual time spent on the road which further adapts to uncertain and dynamic environments.

..2 Motivation

- The goal is to develop a model that reflects real-time constraints and provides more accurate results.This is obtained by introducing the randomness factor.

- This project help us built the probabilistic models that help us build the system which is more flexible to variability, offering more robust route recommendation in uncertain environment.

- This project bridges between the theory and the real-time systems. It combines graphs, algorithms, and probabilistic modeling into one application.

II. Application

1. Adaptive Traffic Signal Control

   - Adjusts interpretations according to live data sourced by sensors at intersections while changing signals in real time.
   - Helps to minimize waiting time and queues based on the current traffic flows, regardless to centralized coordination.

2. Decentralized Routing Algorithms

   - Routs vehicles or local infrastructure nodes using local information to decide the routing.
   - Examples: Ant colony optimization, distributed shortest-path algorithms.
   - Balances traffic across different routes available, which have been found and reduces bottlenecks.

3. Vehicle-to-Infrastructure (V2I) Communication

   - Vehicles communicate with nearby traffic signals or road units.
   - Quick Response to Traffic Change,s E.g., Green Wave Coordination.

4. Real-Time Traffic Monitoring and Feedback Systems

   - The roadside units or cameras monitor traffic flow, providing feedback to drivers or local controllers.
   - Route shifts during congestion peaks are encouraged more frequently.

5. Decentralized Incident Detection and Response

   - The local system detects a local anomaly that may be an accident or bottleneck and redirects traffic immediately around the event.
   - Reduces the effects of secondary congestion.

6. Platooning Support and Cooperative Driving

   - Very close vehicles operate in platoons to effectively pass through intersections.
   - Locally managed to ensure minimum disruption to other traffic flows.

7. Intelligent Intersection Management

   - Each intersection is completely independent and operates on sensor data (such as data from cameras or induction loops).
   - Logic or AI is used to transition phases and prioritize busy lanes.

III. T1 - Mathematical Modelling and Mathematical Analysis

The goal of this project is to optimize route selection in decentralized traffic networks by minimizing travel time (Key Performance Indicator) and congestion. The selected path is modeled using a random variable and its associated probability mass function (PMF).

A. Random Variable: Selected Path (P)

Let $P = \{P_1, P_2, \ldots, P_n\}$ represent the set of all possible paths in the network, where each path P_i is a sequence of nodes and edges. The random variable P denotes the selected path.

B. Probability Mass Function (PMF)

The Probability Mass Function (PMF) for the selected path P_i is given by the formula:

$$P(P_i) = \frac{e^{-\beta f(P_i)}}{\sum_{j=1}^{n} e^{-\beta f(P_j)}}$$

- $f(P_i)$ is the cost function of path P_i
- β is the randomness control parameter, which adjusts the exploration versus exploitation tradeoff.
- Lower the cost for taking a Path(P_i), higher is its probability.

When β is large, the algorithm behaves more deterministically, favoring paths with lower costs. Conversely, when β is small, the algorithm introduces more randomness in the selection of paths. Hence, distribution of β is chosen Gaussian as below:

$$\beta \sim N(1, 0.1^2)$$

C. Cost Function

The total cost $C(P_i)$ of a selected path P_i consists of an actual cost term and a heuristic cost term:

$$C(P_i) = \text{Actual Cost}(P_i) + \beta \times \text{Heuristic Cost}(P_i)$$

D. Expected Travel Time

The expected travel time $\mathbb{E}[\text{Travel Time}]$ is computed as the weighted sum of travel times for all possible paths, where the weights are the probabilities of selecting each path:

$$\mathbb{E}[\text{Travel Time}] = \sum_{i=1}^{n} P(P_i) \times \text{Travel Time}(P_i)$$

E. Pathfinding Strategies

We consider three algorithms for path selection, each with a different approach to minimizing travel time and congestion.

E..1 Pure Dijkstra($\beta = 1$)

The Pure Dijkstra algorithm guarantees the shortest path by exhaustively searching all possible paths. The optimal path P^* is found as:

$$P^* = \min_{P_i} C(P_i)$$

E..2 Standard A*($\beta = 0$)

The A* algorithm uses a heuristic function $h(P_i)$ to estimate the remaining cost, improving search efficiency. The total cost function is:

$$C(P_i) = \text{Actual Cost}(P_i) + h(P_i)$$

E..3 Randomized Approach($\beta > 1$)

In the Randomized approach, we introduce controlled randomness into the path selection, balancing exploration and exploitation. The probability $P(P_i)$ of selecting path P_i is computed using the PMF as described above.

F. Heuristic Costs

The heuristic costs are computed using three types of distance functions:

- **Euclidean Distance**:

$$D_{\text{Euclidean}} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- **Manhattan Distance**:

$$D_{\text{Manhattan}} = |x_2 - x_1| + |y_2 - y_1|$$

- **Diagonal Distance**:

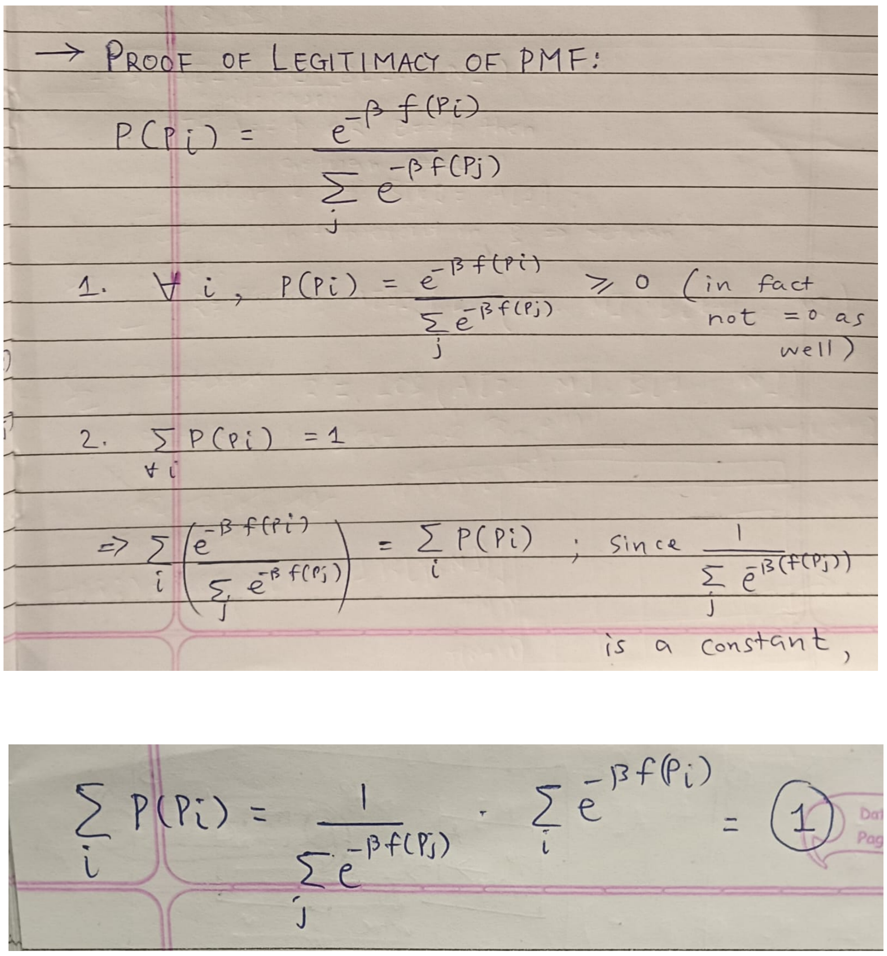
$$D_{\text{Diagonal}} = \max(|x_2 - x_1|, |y_2 - y_1|)$$

These heuristic functions are used to guide the A* and Randomized algorithms in selecting the most promising paths.

G. Exploration vs Exploitation

In the Randomized approach, the balance between exploration (searching for new paths) and exploitation (favoring known good paths) is controlled by the parameter β. A higher value of β makes the algorithm focus more on exploitation, while a smaller value encourages exploration:

$$P(P_i) = \frac{e^{-\beta f(P_i)}}{\sum_{j=1}^{n} e^{-\beta f(P_j)}}$$



→ PROOF OF LEGITIMACY OF PMF:

$$P(P_i) = \frac{e^{-\beta f(P_i)}}{\sum_j e^{-\beta f(P_j)}}$$

1. $\forall i$, $P(P_i) = \frac{e^{-\beta f(P_i)}}{\sum_j e^{-\beta f(P_j)}} \geq 0$ (in fact not $= 0$ as well)

2. $\sum_{\forall i} P(P_i) = 1$

$\Rightarrow \sum_i \left(\frac{e^{-\beta f(P_i)}}{\sum_j e^{-\beta f(P_j)}} \right) = \sum_i P(P_i)$; since $\frac{1}{\sum_j e^{-\beta(f(P_j))}}$ is a constant,

$$\sum_i P(P_i) = \frac{1}{\sum_j e^{-\beta f(P_j)}} \cdot \sum_i e^{-\beta f(P_i)} = \textcircled{1}$$

Figure 1: Proof of Legitimacy of PMF

IV. T2 - Code (with description of each line)

1. Importing Required Libraries

The following Python libraries were used:

- random – for simulating traffic by generating random weights.
- heapq – for priority queue implementation.
- matplotlib.pyplot and networkx – for visualizing the grid and shortest path.
- json – to store and load traffic weights.
- time – for optional performance evaluation.

```python
import random
import heapq
import matplotlib.pyplot as plt
import networkx as nx
import json
```

2. Loading and Generating the Graph

A 5x5 grid is generated where each node initially has a weight of 1. Random traffic weights between 1 and 5 are applied and stored in a JSON file for reuse.

- Each node is a tuple (i, j) representing grid coordinates.
- Weights vary randomly between 1 and 5, simulating light to heavy traffic.
- Traffic data is saved to and loaded from a JSON file for consistency and reusability.

```python
base_graph = {(i, j): 1 for i in range(5) for j in range(5)}

def update_traffic(graph):
    traffic_weights = {}
    for edge in graph:
        traffic_weights[edge] = random.randint(1, 5)
        graph[edge] = traffic_weights[edge]
    return traffic_weights

def apply_traffic_weights(graph, traffic_weights):
    for node in graph:
        graph[node] = traffic_weights[node]
```

3. Heuristic Function

The Manhattan distance is used as the heuristic function for grid-based movement.

$$h(n) = |x_1 - x_2| + |y_1 - y_2| \tag{1}$$

```python
def heuristic(position, goal):
    return abs(position[0] - goal[0]) + abs(position[1] - goal[1])
```

4. A* Algorithm

A* searches for the lowest cost path from start to goal, adjusting dynamically with traffic weights.

$$f(n) = g(n) + h(n)$$

Where:

- $g(n)$ is the actual cost from the start to node n.
- $h(n)$ is the estimated cost from n to the goal.
- $f(n)$ is the total estimated cost of the cheapest path through n.

In this implementation, nodes with the lowest $f(n)$ are expanded first. If traffic increases weight on a path, A* dynamically avoids it in favor of cheaper alternatives.

```python
def a_star_fastest_path(start, goal, graph):
    open_list = []
    closed_list = {}
    start_node = Node(start)
    start_node.g = 0
    start_node.f = start_node.h
    heapq.heappush(open_list, start_node)
    closed_list[start] = start_node.g

    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.position == goal:
            return reconstruct_path(current_node), current_node.g

        neighbors = get_neighbors(current_node.position, graph)
        for next_position, cost in neighbors:
            tentative_g = current_node.g + cost
            if next_position not in closed_list or tentative_g <
                closed_list[next_position]:
                closed_list[next_position] = tentative_g
                neighbor_node = Node(next_position, current_node, cost)
                neighbor_node.g = tentative_g
                neighbor_node.h = heuristic(neighbor_node.position, goal)
                neighbor_node.f = neighbor_node.g
                heapq.heappush(open_list, neighbor_node)
    return None, float('inf')
```

5. Initializing Nodes and Utility Functions

Each node stores path cost, heuristic, and total score. Utility functions manage neighbors and path reconstruction.

Utility functions:

- get_neighbors — Gets adjacent nodes (up, down, left, right).
- reconstruct_path — Backtracks from goal to start using parent links.

```python
class Node:
    def __init__(self, position, parent=None, cost=1):
        self.position = position
        self.parent = parent
        self.g = float('inf')
        self.h = 0
        self.f = float('inf')
        self.cost = cost

    def __lt__(self, other):
        return self.f < other.f

def get_neighbors(position, graph):
    directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]
    neighbors = []
    for d in directions:
        neighbor = (position[0] + d[0], position[1] + d[1])
        if neighbor in graph and graph[neighbor] != float('inf'):
            neighbors.append((neighbor, graph[neighbor]))
    return neighbors

def reconstruct_path(current_node):
    path = []
    while current_node is not None:
        path.append(current_node.position)
        current_node = current_node.parent
    return path[::-1]
```

6. Visualizing the Graph

The graph is displayed using `networkx`, with the shortest path shown in red.

```python
def draw_graph(graph, path=None):
    G = nx.Graph()
    for node in graph:
        G.add_node(node)
        for neighbor, cost in get_neighbors(node, graph):
            G.add_edge(node, neighbor, weight=cost)

    pos = {node: (node[1], -node[0]) for node in G.nodes()}
    weights = nx.get_edge_attributes(G, 'weight')

    plt.figure(figsize=(6, 6))
    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray', node_size=500)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=weights)

    if path:
        path_edges = list(zip(path, path[1:]))
        nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='red', width=2)

    plt.show()
```

V. T3 - Results and Inferences (Domain + CS perspective)

Inferences and results form the integral part of this project, as this domain helps evaluate and make a comparison on existing algorithms and the algorithm proposed in this project. This task further explains two different domains. The first domain is the CS domain, which is purely based on the analysis of the algorithm and the computational efficiency of the code provided by us. On the other hand, the second domain, which is the project domain, focuses on the idea of route optimization considering the travel time , which purely depends on the results drawn from the graph.

This further can be explained by having a detailed comparison of **Randomized A* Algorithm and Deterministic A* Algorithm**.

A. CS Domain

A..1 Time Complexity

Randomized A* Algorithm	Deterministic Algorithm
$O(b^d + b)$	$O(b^d)$

Table 1: Time Complexity of both the algorithms

Explanation:

- For A* Algorithm (Deterministic):
  This performs Exhaustive Search (where all possible paths are explored until an optimal solution is found).
  It prioritize the path using the heuristic function:

  $$f(n) = g(n) + h(n)$$

  where,
  1) $f(n)$ = Total time estimated cost of the path through node n.
  2) $g(n)$ = Cost from the start node to the current node n.
  3) $h(n)$ =Estimated cost from node n to the goal.

  But, during the worst case, heuristic function is not helpful which leads to exponential time complexity $O(b^d)$.
  Where,
  b= branching factor
  d= depth of optimal path or solution.

- For Randomized A* Algorithm:
  This algorithm introduces to probabilistic path selection, which adapts to dynamic traffic conditions.

$$P(P_i) = \frac{e^{-\beta f(P_i)}}{\sum e^{-\beta f(P_i)}}$$

  This adds extra computation per node as:
  1) It computes exponential function $e^{-\beta f(P_i)}$
  2) Calculates the total sum over all the possible paths

  Thus, the total time complexity is $O(b^d + b)$ where, $O(b)$ is the additional cost per node.
  Hence, the time complexity of both the algorithms is nearly same.

A..2 Floating Point Operation Per Second (FLOPS)

Randomized A* Algorithm	Deterministic Algorithm
Total FLOPs per node= $O(b^{+}logb + 10) \approx O(b)$	Total FLOPs per node= $O(logb + 4) \approx O(logb)$

Table 2: FLOPS per node of both the algorithms

Calculation of FLOPs per node:

- A* Algorithm:
  1. Computing heuristic h(n):

     a) 2 FLOPs = Euclidean
     b) 1 FLOP = Manhattan
  2. Computing g(n): 1 FLOP
  3. Computing f(n): 1 FLOP
  4. Heap Insertion($logb$):$\sim O(logb)$ FLOPs
- Randomized A*:
  1. Exponential Function:$\sim O(10)$FLOPs
  2. Summation over all paths: $O(b)$ FLOPs
  3. Sampling from probability distribution: $O(logb)$FLOPs

Thus, Randomized A* requires more FLOPs due to probabilistic calculations at each step.

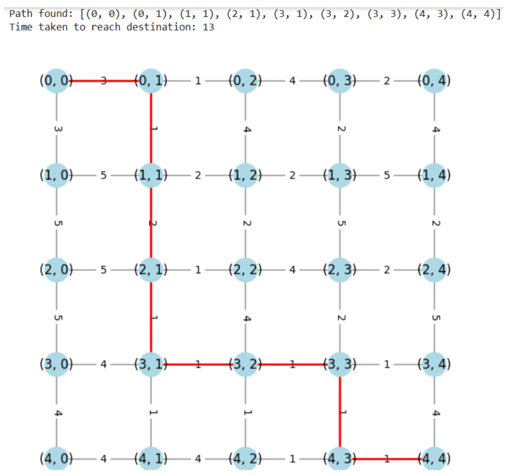
A..3 Convergence Rate

Parameters	Randomized A* Algorithm	Deterministic Algorithm
Optimality vs Speed	Finds optimal path faster, but lacks in accuracy	Always find optimal path, but takes longer time
Handling Large Graphs	Scales better as it skips unnecessary node expansions	It is slower in large graphs due to exhaustive search

Table 3: Comparison of Convergence Rate

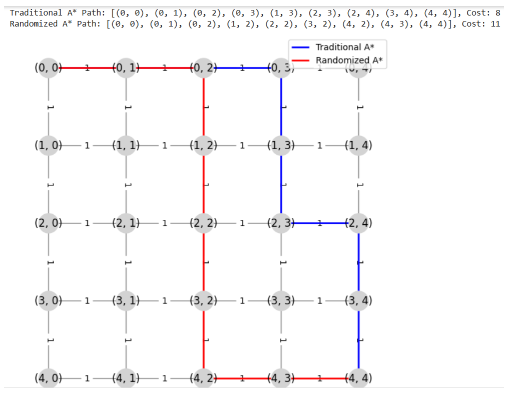
B. Project Domain

This domain specifically helps define the real-world context and relevance to our project. This basically further explains why the problem is important, where it can be applied and who benefits from the solution.



```
Path found: [(0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (4, 3), (4, 4)]
Time taken to reach destination: 13
```

Figure 2: Path finding with Randomized A*



```
Traditional A* Path: [(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 4), (3, 4), (4, 4)], Cost: 8
Randomized A* Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4)], Cost: 11
```

Figure 3: Path finding through both the algorithms havingn uniform edge weight

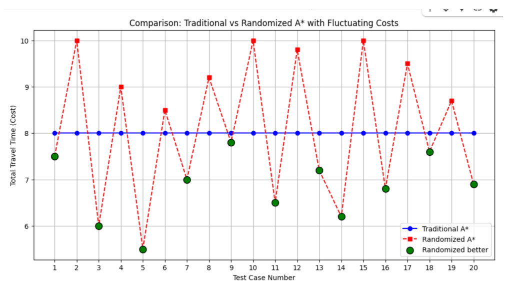


Figure 4: Total time vs No. of simulations

1. For figure-2: This is using Randomized A*, where it calculates the minimum travel time having different weights (represents time) of each edge.This graph despite not being the shortest in terms of node count, the chosen path minimizes cumulative weight (time).
   This graph demonstrates the algorithm's ability to optimize based on real-world travel time, not just spatial distance.

2. For figure-3: This graphs shows both the algorithm, where the Traditional A* chooses the shortest possible path, whereas the Randomized A* explores more diverse paths, adding to more cost.
   But, this infers that traditional algorithm is optimal for static environments, whereas the randomized A* is more optimal for real-world variability.

Figure-4 is the representation of 2 axis graph for both the Aalgorithms to find out the different travel time based on the total number of simulations. This graph provide an overall insight on the above two graphs. Based on this, the inferences are:

– **Scalability in Large-scale networks**: For example, in urban traffic routing there are some factors that are dynamic in nature, such as congestion, weather, or road closure. In such cases, Randomized A* is prioritized where exploring alternative path can sometimes give shorter path as compared to the fixed heuristics.

– **Algorithmic Efficiency based on travel time**: This graph shows that the traditional A* algorithm has a consistent travel time across test cases, further depicting higher reliability in stable environments. Whereas, Randomizes A* algorithm is volatile leading to better performance in about half the test cases.

– **Balancing Exploration and Exploitation in Path finding**: Traditional A * follows exploitation (following the best optimal path) and Randomized A* follows exploration(testing alternative route). Thus, here randomized A* introduces the degree of uncertainty, which is beneficial when dealing with unknown fluctuations.

VI. T4 - Algorithm(Deterministic/Baseline and Randomized)

The search for the most efficient path and routing strategy remains a fundamental challenge across traffic networks, robotics, AI navigation systems, and distributed computing. This paper addresses the need to alleviate traffic congestion and minimize average travel time from the source to the destination, chiefly in decentralized traffic networks. Various algorithms developed over time to address these challenges can be generically classified into two major categories: deterministic algorithms and randomized algorithms.

- Deterministic algorithms use strictly systematic techniques to ensure optimal solutions through systematic node expansions.
- Randomized algorithms, in contrast, typically add some elements of controlled randomness to maximize adaptability in dynamic, uncertain environments.

A. Deterministic Algorithim

Algorithms like Dijkstra's Algorithm and A* Search are based on a well-defined heuristic search strategy that guarantees optimal solutions. They proceed to expand nodes systematically based on cost functions that minimize total path cost. This systematic and predictable behavior classifies them as deterministic algorithms.

A..1 Key Characteristics

- Employs a priority queue (Open List) to select the most promising nodes.
- Uses a predefined cost function f(v) = g(v)+h(v):
    - g(v): Cost from the start node to v
    - h(v): Estimated heuristic cost from v to the goal.
- Guarantees an optimal path if it exists.
- Stops upon termination: Either purpose is achieved or exhausts all possibilities.

A..2 Advantages

- Guaranteed optimality: Always finds the shortest path.
- Reproducible results: Same input equals same output.
- Good efficiency in clearly defined environments: Operates well when the graph structure and heuristics are well understood.

Algorithm 1 A* Search Algorithm

```
   Inputs: Graph $G(V, E)$, Start node $s$, Goal node $t$, Heuristic function $h$
   Output: Shortest path from $s$ to $t$ (if exists)
 1: Initialize open list (priority queue) and closed list
 2: Insert $s$ into open list with $f(s) = 0$
 3: while open list is not empty do
 4:     Remove node $q$ with lowest $f$ from open list
 5:     if $q = t$ then return path
 6:     end if
 7:     for each successor of $q$ do
 8:         Compute $g$, $h$, and $f = g + h$
 9:         if better path found or node not in open/closed list then
10:             Update and insert successor into open list
11:         end if
12:     end for
13:     Add $q$ to closed list
14: end while
15: return "No path found"
```

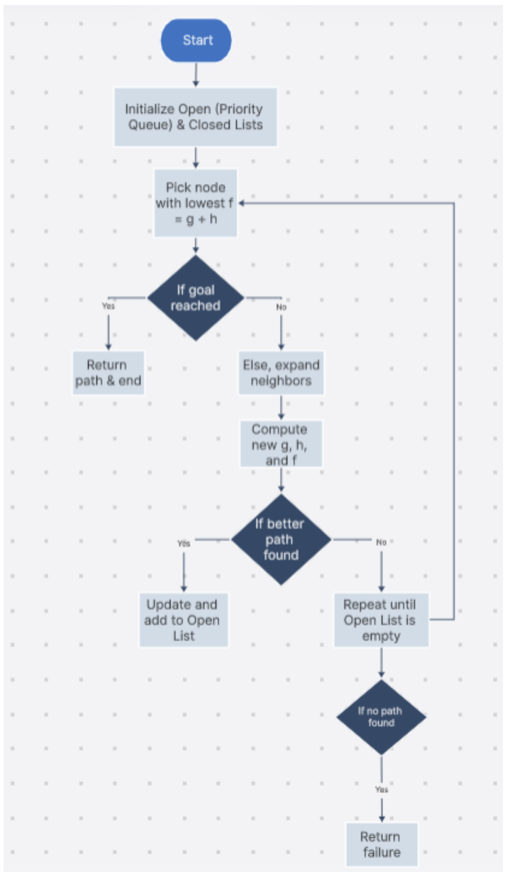


Figure 5: A* Search Algorithm

A..3 Disadvantages

- Heavy in computations: Makes computation more formidable in larger and more complex graph situations.
- Not adaptive: Works poorly in dynamic and uncertain environments.

B. Randomized Algorithms

Randomized algorithms are added with stochastic components to allow for adaptability and performance when faced with uncertain environments. One particular algorithm is Randomized A*, an A* variation in which probabilistic perturbations are applied to the selection of nodes.

B..1 Key Characteristics

- Cost function modified: $f(v) = g(v) + kh(v)$, where k with some probability is picked randomly from an interval (1,k_{max}).
- Controlled randomness, broadening the search.
- Helps to exit local minima via the exploration of other paths.
- The balance is struck between deterministic and random search by means of a beta function (β).

B..2 Advantages

- Increased Exploration: Keeps from being stuck in arbitrary, bad solutions.
- Adaptability: Works well in dynamically changing environments in real-time.

Algorithm 2 Random A* Scaling Algorithm

```
Require: Road network $G_{V,E,W}$, source $s$, destination $t$, $k_{\max}$
Ensure: A route from $s$ to $t$
 1: $Q \leftarrow \emptyset$, $Q' \leftarrow \emptyset$
 2: for each vertex $v \in V$ do
 3:     $d_s(v) \leftarrow \infty$, $h_t(v) \leftarrow \infty$, $f(v) \leftarrow \infty$
 4: end for
 5: $d_s(s) \leftarrow 0$, $h_t(s) \leftarrow 0$, $f(s) \leftarrow 0$
 6: $Q$.Insert($s$)
 7: while $Q$ is not empty do
 8:     $u \leftarrow Q$.ExtractMin()
 9:     $Q'$.Insert($u$)
10:     if $u = t$ then
11:         return path from $s$ to $t$
12:     end if
13:     $k \leftarrow$ random(1, $k_{\max}$)
14:     for each vertex $v \in Q$ do
15:         $f(v) \leftarrow d_s(v) + k \times h_t(v)$
16:     end for
17:     for each edge $e_{u,v} \in E$ do
18:         if $v \in Q'$ then
19:             continue
20:         end if
21:         if $v \notin Q$ then
22:             $Q$.Insert($v$)
23:         end if
24:         $d_s(v)' \leftarrow d_s(u) + weight_e$
25:         $h_t(v)' \leftarrow$ EstimateCost($v, t$)
26:         $f(v)' \leftarrow d_s(v)' + k \times h_t(v)'$
27:         if $f(v)' < f(v)$ then
28:             $d_s(v) \leftarrow d_s(v)'$
29:             $h_t(v) \leftarrow h_t(v)'$
30:             $f(v) \leftarrow f(v)'$
31:             $prev(v) \leftarrow u$
32:         end if
33:     end for
34: end while
```

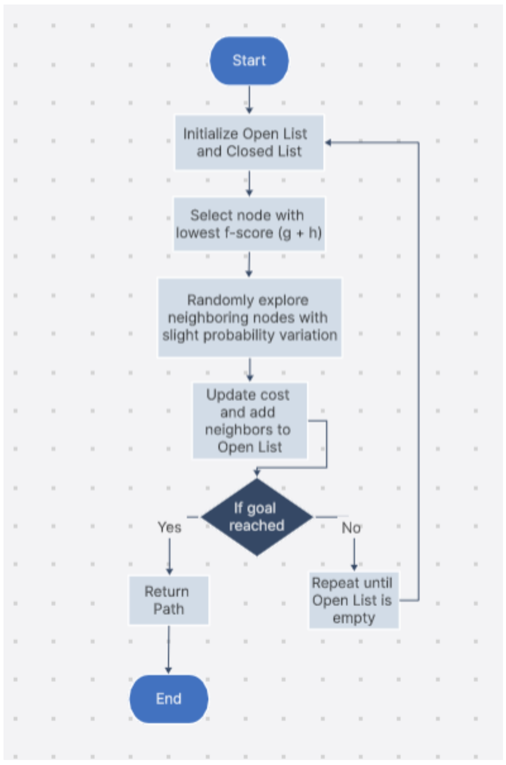

Start
Initialize Open List and Closed List
Select node with lowest f-score (g + h)
Randomly explore neighboring nodes with slight probability variation
Update cost and add neighbors to Open List
If goal reached
Yes
No
Return Path
Repeat until Open List is empty
End

Figure 6: Random A* Scaling Algorithm

B..3 Disadvantages

- No guaranteed optimality: Not always give the result-catalog as the shortest path.
- Larger computation cost: It spreads over unnecessary nodes.
- Non-deterministic results: Different runs usually lead to different paths.

C. Importance of Randomization

1. Avoiding Local Minima
   - Randomization provides alternative ways of exploration, thereby preventing the algorithm from getting trapped in local optima.
   - Example: In pathfinding, when heuristic approaches are adopted, they may fail due to a highly complex terrain, while randomization will permit diverse explorations.
2. Balancing Exploration with Exploitation
   - The deterministic algorithms exploit known pathways, whilst the randomized methods explore alternatives.

   - Example: Randomness in Simulated Annealing prevents premature convergence.

3. Handling Uncertainty

   - Randomized algorithms adapt very well in dynamic and unpredictable environments.
   - Example: By incorporating a random approach to path selection, an autonomous robot can deal with suddenly occurring obstacles.

4. Load Balancing and Fairness

   - Within network systems, randomized algorithms lend themselves to efficient load distribution.
   - Example: Randomized task assignment prevents bottlenecks in distributed computing.

VII. T5 - Derivation of Bounds and Results (new inferences)

The idea is to derive probabilistic bound for travel time by integrating path selection probability and integrating Chernoff bound with min-cut algorithm. The main goal is to identify probability of deviation in travel time due to traffic congestion and apply Chernoff bound to obtain an exponential decay bound. And also integrating the Min-Cut Algorithm, which helps in figuring the minimal capacity required to disconnect the source and destination, giving us a refined bound.

A. Deriving Chernoff Bound

– To integrate probability of selecting each path,We use PMF:

$$P(Pi) = \frac{e^{-\beta f(Pi)}}{\sum_j e^{-\beta f(Pj)}}$$

– Also, the Expected Travel Time is given by:

$$E[T] = \sum_i P(Pi)T(Pi)$$

Now, we use Standard Chernoff bound technique, which depends on moment generating function of T. For any $s > 0$:

$$P(T \geq (1+\delta)E[T]) = P(e^{sT} \geq e^{s(1+\delta)E[T]})$$

Applying Markov's inequality:

$$P(T \geq (1+\delta)E[T]) \leq \frac{E[e^{sT}]}{e^{s(1+\delta)E[T]}}$$

Now, we derive $E[e^{sT}]$. Since T is a weighted sum over selected paths:

$$E[e^{sT}] = \sum_i P(P_i)e^{sT(P_i)}$$

Substituting $P(P_i)$:

$$E[e^{sT}] = \sum_i \frac{e^{-\beta f(P_i)}}{\sum_j e^{-\beta f(P_j)}} e^{sT(P_i)}$$

We choose s optimally as $s = \frac{\delta}{3E[T]}$, following standard Chernoff analysis, to get the bound:

$$P(T \geq (1+\delta)E[T]) \leq e^{-\frac{\delta^2 E[T]}{3}}$$

B. Applying Chernoff Bound

On Applying Chernoff Bound for travel time deviation:

$$P(T \geq (1+\delta)E[T]) \leq e^{\frac{-\delta^2 E[T]}{3}}$$

We get:

- The probability of travel time exceeding a certain threshold is upper-bounded.
- An exponential decay in probability as the deviation increases.

C. Integrating Min-Cut with Chernoff Bound

The min-cut algorithm is basic result in network theory that finds out the smallest set of edges which can be removed to separate source from destination.
To define min-cut in our problem:

- Set A: Contains the source S.
- Set B: Contains the Destination D.
- **Min-Cut Capacity** $C_{min-cut}$: The minimum sum of capacities over all edges crossing from A to B:
  $$C_{min-cut} = \sum_{e \in cut} c(e)$$
  where $c(e)$ is capacity of edge e in the cut.

Now,Instead of bounding deviations relative to expected travel time, we now bound them relative to $C_{\text{min-cut}}$. The modified bound is:

$$P(T \geq (1+\delta)C_{\text{min-cut}}) \leq e^{-\frac{\delta^2 C_{\text{min-cut}}}{2(1+\delta)}}$$

Also, importance of denominator different from 3 to $2(1+\delta)$ is:

- When bounding travel time via expected value $E[T]$, the variance of the sum of independent random variables is used, leading to the denominator 3.
- When bounding deviations in terms of capacity, we use a different Chernoff bound variant that considers capacity constraints, leading to the denominator $2(1+\delta)$.
- This modified form ensures a tighter bound when dealing with flow constraints.

VIII. References

[1] GeeksforGeeks. (2022). *A* search algorithm* [Accessed: 2025-04-06]. https://www.geeksforgeeks.org/a-search-algorithm/

[2] Kivimäki, I., Lebichot, B., Saramäki, J., & Saerens, M. (2016). Two betweenness centrality measures based on randomized shortest paths. *Scientific Reports*, *6*(1), 19668. https://doi.org/10.1038/srep19668

[3] Nguyen, U. T. V., Karunasekera, S., Kulik, L., Tanin, E., Zhang, R., Zhang, H., Xie, H., & Ramamohanarao, K. (2015). A randomized path routing algorithm for decentralized route allocation in transportation networks. *Proceedings of the 8th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, 15–20. https://doi.org/10.1145/2834882.2834886