AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

AMITY UNIVERSITY CAMPUS,
SECTOR-125, NOIDA-201303

Analysis and Design of Algorithms

CASE STUDY
CODE: CSE 303

Submitted to:
Dr. Sumita Gupta

Submitted by:
Jigyasa Kumari
(A2305221529)
Shambhavi Mishra
(A2305221660)
5CSE6-Y

RouteViz: Optimizing Delivery Routes for a Food Delivery Service

Background:

A popular food delivery service is facing challenges in optimizing their delivery routes. With a growing customer base and expanding service area, they need an efficient way to ensure timely and cost-effective deliveries.

Objective:

The objective of this case study is to implement and analyze various pathfinding algorithms, including Dijkstra's Algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS), to optimize the delivery routes for the food delivery service. The goal is to reduce delivery times, minimize fuel costs, and improve overall customer satisfaction.

Details:

1. **Data Collection:**
   Collect data on customer locations, restaurant locations, and road networks in the service area.

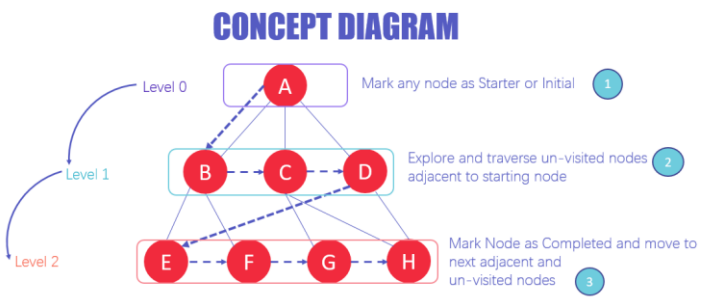
2. **Algorithms:**

Dijkstra's Algorithm:

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

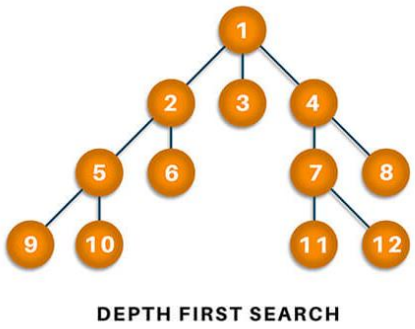
Breadth-First Search (BFS):

Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.



Depth-First Search (DFS):

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure.



DEPTH FIRST SEARCH

3. **Considerations:**
   Take into account factors such as traffic conditions, time of day, and one-way streets in the pathfinding process.

4. **Custom Scenarios:**
   Create scenarios with varying numbers of orders, different restaurant locations, and time constraints to test the algorithms' adaptability.

5. **Performance Metrics:**
   Measure and compare the total distance travelled, average delivery time, and fuel costs for each algorithm.

6. **Optimization Strategies:**
   Explore additional strategies, such as clustering nearby orders, to further optimize routes.

7. **Simulation and Analysis:**
   Simulate deliveries over a specified time period and analyze the results to identify the most efficient algorithm and strategies.

8. **Recommendations:**
   Based on the analysis, provide recommendations for route optimization strategies that the food delivery service can implement.

Benefits:

By optimizing delivery routes, the food delivery service can reduce operational costs, improve delivery times, and enhance customer satisfaction. This case study provides valuable insights into the application of pathfinding algorithms in real-world logistics scenarios.

Constraints:

- The service area is represented as a connected graph, where each node represents a location (customer or restaurant) and each edge represents a road between locations.
- Each road has an associated distance, representing the travel distance between two locations.
- The number of customers and restaurants may vary, but the graph is assumed to be connected, ensuring there is a path from any customer to any restaurant.

- Time of day and traffic conditions may affect travel times, but for the purpose of this study, these factors are not explicitly considered.
- The algorithms assume that the road network does not change during the delivery process.

Algorithm Implementation in C++:

BFS:

```
#include <iostream>
#include <list>

using namespace std;

class Graph {
  int numVertices;
  list<int>* adjLists;
  bool* visited;

   public:
  Graph(int vertices);
  void addEdge(int src, int dest);
  void BFS(int startVertex);
};

// Create a graph with given vertices,
// and maintain an adjacency list
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
}

// Add edges to the graph
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_back(dest);
  adjLists[dest].push_back(src);
}

// BFS algorithm
void Graph::BFS(int startVertex) {
  visited = new bool[numVertices];
  for (int i = 0; i < numVertices; i++)
    visited[i] = false;

  list<int> queue;

  visited[startVertex] = true;
  queue.push_back(startVertex);

  list<int>::iterator i;

  while (!queue.empty()) {
    int currVertex = queue.front();
```

```
    cout << "Visited " << currVertex << " ";
    queue.pop_front();

    for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {
      int adjVertex = *i;
      if (!visited[adjVertex]) {
        visited[adjVertex] = true;
        queue.push_back(adjVertex);
      }
    }
  }
}

int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 0);
  g.addEdge(2, 3);
  g.addEdge(3, 3);

  g.BFS(2);

  return 0;
}
```

DFS:

```
#include <iostream>
#include <list>
using namespace std;

class Graph {
  int numVertices;
  list<int> *adjLists;
  bool *visited;

   public:
  Graph(int V);
  void addEdge(int src, int dest);
  void DFS(int vertex);
};

// Initialize graph
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
  visited = new bool[vertices];
}
```

```
// Add edges
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_front(dest);
}

// DFS algorithm
void Graph::DFS(int vertex) {
  visited[vertex] = true;
  list<int> adjList = adjLists[vertex];

  cout << vertex << " ";

  list<int>::iterator i;
  for (i = adjList.begin(); i != adjList.end(); ++i)
    if (!visited[*i])
      DFS(*i);
}

int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 3);

  g.DFS(2);

  return 0;
}
```

Dijkstra's Algorithm:

```
#include <iostream>
#include <vector>

#define INT_MAX 10000000

using namespace std;

void DijkstrasTest();

int main() {
  DijkstrasTest();
  return 0;
}

class Node;
class Edge;

void Dijkstras();
vector<Node*>* AdjacentRemainingNodes(Node* node);
```

```
Node* ExtractSmallest(vector<Node*>& nodes);
int Distance(Node* node1, Node* node2);
bool Contains(vector<Node*>& nodes, Node* node);
void PrintShortestRouteTo(Node* destination);

vector<Node*> nodes;
vector<Edge*> edges;

class Node {
  public:
 Node(char id)
   : id(id), previous(NULL), distanceFromStart(INT_MAX) {
   nodes.push_back(this);
 }

  public:
 char id;
 Node* previous;
 int distanceFromStart;
};

class Edge {
  public:
 Edge(Node* node1, Node* node2, int distance)
   : node1(node1), node2(node2), distance(distance) {
   edges.push_back(this);
 }
 bool Connects(Node* node1, Node* node2) {
   return (
     (node1 == this->node1 &&
      node2 == this->node2) ||
     (node1 == this->node2 &&
      node2 == this->node1));
 }

  public:
 Node* node1;
 Node* node2;
 int distance;
};

//////////////////
void DijkstrasTest() {
 Node* a = new Node('a');
 Node* b = new Node('b');
 Node* c = new Node('c');
 Node* d = new Node('d');
 Node* e = new Node('e');
 Node* f = new Node('f');
 Node* g = new Node('g');
```

```
  Edge* e1 = new Edge(a, c, 1);
  Edge* e2 = new Edge(a, d, 2);
  Edge* e3 = new Edge(b, c, 2);
  Edge* e4 = new Edge(c, d, 1);
  Edge* e5 = new Edge(b, f, 3);
  Edge* e6 = new Edge(c, e, 3);
  Edge* e7 = new Edge(e, f, 2);
  Edge* e8 = new Edge(d, g, 1);
  Edge* e9 = new Edge(g, f, 1);

  a->distanceFromStart = 0;  // set start node
  Dijkstras();
  PrintShortestRouteTo(f);
}

//////////////////

void Dijkstras() {
  while (nodes.size() > 0) {
    Node* smallest = ExtractSmallest(nodes);
    vector<Node*>* adjacentNodes =
      AdjacentRemainingNodes(smallest);

    const int size = adjacentNodes->size();
    for (int i = 0; i < size; ++i) {
      Node* adjacent = adjacentNodes->at(i);
      int distance = Distance(smallest, adjacent) +
            smallest->distanceFromStart;

      if (distance < adjacent->distanceFromStart) {
        adjacent->distanceFromStart = distance;
        adjacent->previous = smallest;
      }
    }
    delete adjacentNodes;
  }
}

// Find the node with the smallest distance,
// remove it, and return it.
Node* ExtractSmallest(vector<Node*>& nodes) {
  int size = nodes.size();
  if (size == 0) return NULL;
  int smallestPosition = 0;
  Node* smallest = nodes.at(0);
  for (int i = 1; i < size; ++i) {
    Node* current = nodes.at(i);
    if (current->distanceFromStart <
      smallest->distanceFromStart) {
      smallest = current;
      smallestPosition = i;
```

```
    }
  }
  nodes.erase(nodes.begin() + smallestPosition);
  return smallest;
}

// Return all nodes adjacent to 'node' which are still
// in the 'nodes' collection.
vector<Node*>* AdjacentRemainingNodes(Node* node) {
  vector<Node*>* adjacentNodes = new vector<Node*>();
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    Node* adjacent = NULL;
    if (edge->node1 == node) {
      adjacent = edge->node2;
    } else if (edge->node2 == node) {
      adjacent = edge->node1;
    }
    if (adjacent && Contains(nodes, adjacent)) {
      adjacentNodes->push_back(adjacent);
    }
  }
  return adjacentNodes;
}

// Return distance between two connected nodes
int Distance(Node* node1, Node* node2) {
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    if (edge->Connects(node1, node2)) {
      return edge->distance;
    }
  }
  return -1;  // should never happen
}

// Does the 'nodes' vector contain 'node'
bool Contains(vector<Node*>& nodes, Node* node) {
  const int size = nodes.size();
  for (int i = 0; i < size; ++i) {
    if (node == nodes.at(i)) {
      return true;
    }
  }
  return false;
}

//////////////////
```

```
void PrintShortestRouteTo(Node* destination) {
  Node* previous = destination;
  cout << "Distance from start: "
     << destination->distanceFromStart << endl;
  while (previous) {
    cout << previous->id << " ";
    previous = previous->previous;
  }
  cout << endl;
}

// these two not needed
vector<Edge*>* AdjacentEdges(vector<Edge*>& Edges, Node* node);
void RemoveEdge(vector<Edge*>& Edges, Edge* edge);

vector<Edge*>* AdjacentEdges(vector<Edge*>& edges, Node* node) {
  vector<Edge*>* adjacentEdges = new vector<Edge*>();

  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    if (edge->node1 == node) {
      cout << "adjacent: " << edge->node2->id << endl;
      adjacentEdges->push_back(edge);
    } else if (edge->node2 == node) {
      cout << "adjacent: " << edge->node1->id << endl;
      adjacentEdges->push_back(edge);
    }
  }
  return adjacentEdges;
}

void RemoveEdge(vector<Edge*>& edges, Edge* edge) {
  vector<Edge*>::iterator it;
  for (it = edges.begin(); it < edges.end(); ++it) {
    if (*it == edge) {
      edges.erase(it);
      return;
    }
  }
}
```

Comparison of DFS, BFS, and Dijkstra's Algorithm for Routing:

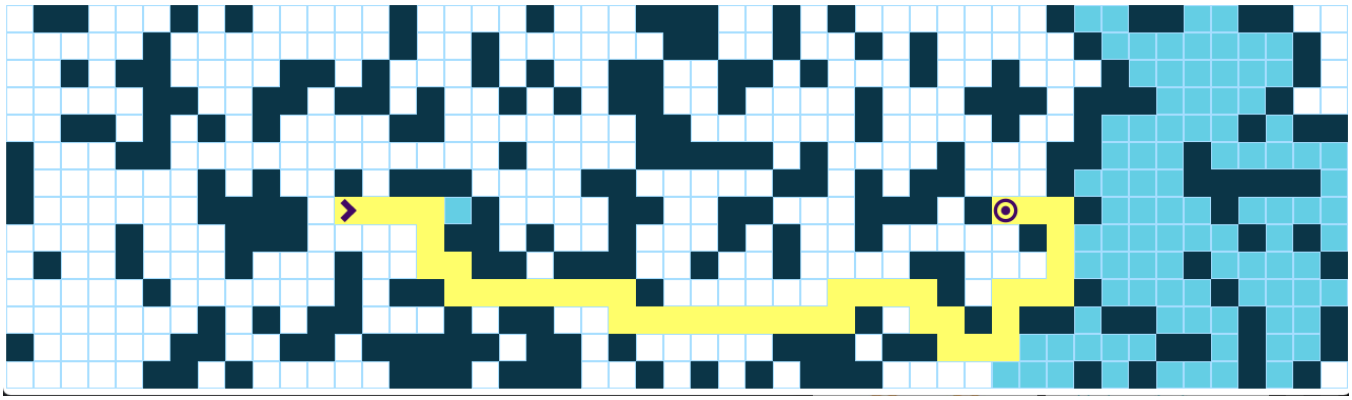
1. **Depth-First Search (DFS):**
    - **Complexity:**
        - Time Complexity: O(V + E)
        - Space Complexity: O(V)

- **Strengths:**
    - Memory efficient as it only requires space for the visited nodes and the call stack.
    - Well-suited for finding paths in unweighted graphs or traversing deep into a graph.
- **Weaknesses:**
    - Not suitable for finding shortest paths in weighted graphs as it may explore longer paths first.

2. **Breadth-First Search (BFS):**
    - **Complexity:**
        - Time Complexity: O(V + E)
        - Space Complexity: O(V)
    - **Strengths:**
        - Guarantees the shortest path in unweighted graphs.
        - Well-suited for finding shortest paths and level-based traversal.
    - **Weaknesses:**
        - Requires more memory compared to DFS.
3. **Dijkstra's Algorithm:**
    - **Complexity:**
        - Time Complexity: O((V + E) * log(V)) with a min-priority queue (using Fibonacci heap, this can be reduced to O(V^2 + E))
        - Space Complexity: O(V)
    - **Strengths:**
        - Guarantees the shortest path in weighted graphs.
        - Highly efficient for finding shortest paths, especially in scenarios where edge weights vary.
    - **Weaknesses:**
        - Requires more computational resources, especially in dense graphs.

Test Cases:

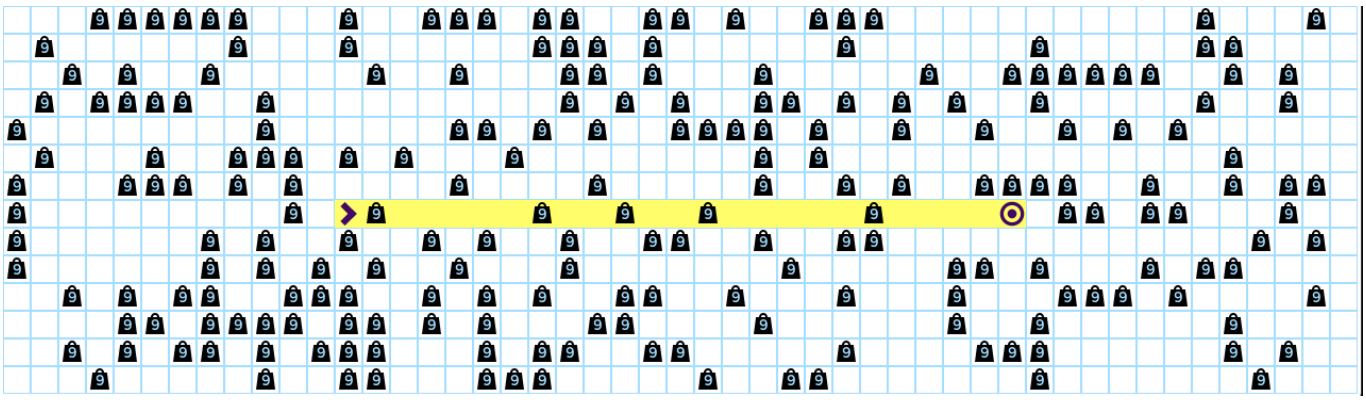
- **Unweighted Graph**



Output:

Algorithm	Steps Explored	Path Length	Path Cost	Time Taken
Depth First Search	136	41	40	1.19ms
Breadth First Search	303	33	32	1.82ms
Dijkstra's Search	301	33	32	4.30ms

- **Weighted Graph**



Output:

Algorithm	Steps Explored	Path Length	Path Cost	Time Taken
Depth First Search	25	25	64	0.84ms
Breadth First Search	469	25	64	3.98ms
Dijkstra's Search	451	35	34	6.05ms

Summary:

- **DFS** is memory efficient and is suitable for unweighted graphs. However, it may not find the shortest path in weighted graphs.
- **BFS** is well-suited for finding the shortest path in unweighted graphs. It requires more memory compared to DFS but guarantees optimal results.
- **Dijkstra's Algorithm** is the most efficient for finding shortest paths in weighted graphs, but it comes at the cost of higher computational complexity. It is ideal when edge weights vary and an accurate shortest path is crucial.
- In routing scenarios where accuracy and efficiency are critical, **Dijkstra's Algorithm** is the preferred choice. **BFS** can be suitable for unweighted graphs, while **DFS** may be used in scenarios where memory efficiency is a priority and an exact shortest path is not necessary.