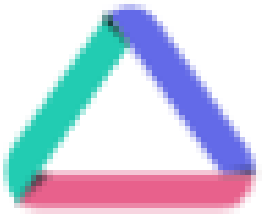

Route Mapper Pro

- Algorithms Project
- Team 28
- Supervised By:
    - Dr. Ahmed Salah
    - T.A.: Hossam Sherif

Name	DEP	ID
Mohamed Hussein Ragab	**SC**	**2022170581**
Ehab Mohamed Salah	**SC**	**2022170087**
Jupiter Mousa Zakria	**SC**	**2022170114**
Kerolos Tharwat Fawzy	**SC**	**2022170317**
Mazen Mahmoud Mostafa	**SC**	**2022170339**
Moataz Mohamed Mansour	**SC**	**2022170625**

1. Graph Construction

Description

The graph construction process is designed to represent the geographical road network for pathfinding algorithms efficiently. First, we define a lightweight Point struct to make coordinate handling more fluent and intuitive. Then, we create an array to store the coordinates of all intersections for fast access. Finally, we construct adjacency lists using tuples containing both time-based costs and physical distances, effectively maintaining two parallel graph representations in one data structure: one weighted by road lengths and another by travel times. This dual representation allows the algorithm to optimize routes based on travel time while also tracking the actual distances traveled.

```
public struct Point
{
    public double X;
    public double Y;

    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }
}

const double walkSpeed = 5.0;
const double eps = 1e-12;
// Taking Input
int n = mapData.Intersections.Count;
Point[] intersections = new Point[n];
for (int i = 0; i < n; i++) // O(N)
{
    var intersection = mapData.Intersections[i];
    intersections[i] = new Point(intersection.x, intersection.y);
}

List<(int Node, double Cost)>[] roads = new List<(int, double)>[n];
for (int i = 0; i < n; i++) // O(N)
{
    roads[i] = new List<(int, double)>();
}

// Dictionary to Save the Length of Each Road with a Key pair of (Starting
Intersection, Ending Intersection) and value is the length of the road
Dictionary<(int, int), double> roadLengths = new Dictionary<(int, int), double>();

foreach (var road in mapData.Roads) // O(N)
{
    int u = road.fromId;
    int v = road.toId;
    double len = road.length;
    double speed = road.speed;
    double cost = len / speed;

    roadLengths[(u, v)] = len;
```

```
    roadLengths[(v, u)] = len;

    roads[u].Add((v, cost));
    roads[v].Add((u, cost));
}
```

Graph Construction

The graph is constructed using adjacency lists. Then, we use two Lists of dictionaries, one for the roads' length and one for the roads' time.

---

2. Shortest Path Algorithm

Description

First, we implement a multi-source shortest path version of Dijkstra's algorithm. This approach allows us to have multiple starting points (all intersections within walking distance of the source) and find the optimal path to the destination. For each query, we identify all intersections that are within the specified walking radius from both the start and end points, then use these as potential entry and exit points for the road network.

Algorithm Flow

1. **Initialization**:

```
private static double SquaredDistance(in Point a, in Point b)
{
    double dx = a.X - b.X;
    double dy = a.Y - b.Y;
    return dx * dx + dy * dy;
}

private static double EuclideanDistance(in Point a, in Point b)
{
   return Math.Sqrt(SquaredDistance(a, b));
}
```

   - For each intersection, calculate its Euclidean distance to both start and end points:-

```
double[] distToStart = new double[n]; // Dist from Starting Point to Each
intersection Pre-Computed
double[] distToEndSq = new double[n]; // Dist from Each intersection to
Ending Point, but I saved it Squared to Compare it with R^2 instead of R to
Avoid The Additional Log
bool[] withinRangeFromStart = new bool[n]; // Cache the comparison to use it
again instead of re-calc it with a high constant factor
bool[] withinRangeFromEnd = new bool[n]; // Cache the comparison to use it
again instead of re-calc it with a high constant factor
```

```
for (int i = 0; i < n; i++)
{
    double distSqStart = SquaredDistance(intersections[i], start);
    double distSqEnd = SquaredDistance(intersections[i], end);
    // Caching Code
    withinRangeFromStart[i] = distSqStart <= rSquared + eps;
    withinRangeFromEnd[i] = distSqEnd <= rSquared + eps;

    if (withinRangeFromStart[i])
    {
        distToStart[i] = Math.Sqrt(distSqStart);
    }

    distToEndSq[i] = distSqEnd;
}
```

- Mark intersections as valid starting points if they're within walking radius of the source. Initialize the priority queue with these valid starting points, setting the initial cost as walking time to reach them

```
for (int i = 0; i < n; i++)
{
    if (withinRangeFromStart[i])
    {
        dist[i] = distToStart[i] / walkSpeed;
        pq.Enqueue(i, dist[i]);
    }
}

double directDistance = EuclideanDistance(start, end);
double bestTime = directDistance <= r + eps ? directDistance /
walkSpeed : double.PositiveInfinity;
int bestNode = -1;
```

2. **Multi-Source Dijkstra**:

```
// Dijkstra's algorithm
while (pq.Count > 0)
{
    int cur = pq.Dequeue();
    double cost = dist[cur];

    if (cost > dist[cur] + eps)
        continue;

    if (withinRangeFromEnd[cur])
    {
        double endDistance = Math.Sqrt(distToEndSq[cur]);
        double totalCost = cost + (endDistance / walkSpeed);
        if (totalCost < bestTime)
        {
            bestTime = totalCost;
            bestNode = cur;
        }
    }
```

```
    if (cost > bestTime)
        continue;

    foreach (var (neighbor, edgeCost) in roads[cur])
    {
        double newCost = cost + edgeCost;
        if (newCost < dist[neighbor] - eps)
        {
            dist[neighbor] = newCost;
            parent[neighbor] = cur;

            pq.Enqueue(neighbor, newCost);
        }
    }
}

List<int> path = new List<int>();
if (bestNode != -1)
{
    int pathLength = 0;
    int current = bestNode;
    while (current != -1)
    {
        pathLength++;
        current = parent[current];
    }
    path = new List<int>(pathLength);
    current = bestNode;
    while (current != -1)
    {
        path.Add(current);
        current = parent[current];
    }
    path.Reverse();
}
```

- Execute Dijkstra's algorithm starting from all valid entry points simultaneously
- Use travel time as the cost metric (distance/speed)
- Maintain priority queue with SortedSet<QueueItem> for efficient node selection

3. **Destination Checking During Exploration**:
    - For each processed node, check if it's within walking distance of the destination
    - If yes, calculate total journey time (driving time + final walking segment)
    - Keep track of the best exit node that minimizes total travel time
    - Implement early termination by skipping exploration when the current cost exceeds the best found solution

Path Construction:

```
        // Calculate distances
```

```
double drivingDistance = 0;
double walkingDistanceStart = 0;
double walkingDistanceEnd = 0;

if (path.Count > 0)
{
    walkingDistanceStart = distToStart[path[0]];
    walkingDistanceEnd = Math.Sqrt(distToEndSq[path[path.Count - 1]]);

    for (int i = 1; i < path.Count; i++)
    {
        int u = path[i - 1];
        int v = path[i];
        drivingDistance += roadLengths[(u, v)];
    }
}
else if (directDistance <= r + eps)
{
    walkingDistanceStart = directDistance;
}

double totalWalk = walkingDistanceStart + walkingDistanceEnd;
double totalDriving = drivingDistance;

// Add query result
mapData.QueryResults.Add(new QueryResult
{
    PathIds = path,
    ShortestTime = bestTime * 60, // Convert to minutes
    TotalDistance = totalWalk + totalDriving,
    WalkingDistance = totalWalk,
    VehicleDistance = totalDriving
});
```

- After finding the optimal exit node, trace back using parent pointers to reconstruct the path
- Calculate walking distance at start, walking distance at end, and driving distance between
- Handle edge case of direct walking path when start and end are within walking distance

Priority Queue: -

Used in graph traversal algorithms where nodes need to be processed in order of their priority value (e.g., shortest distance first). Implements natural ordering through IComparable<QueueItem> with stable tie-breaking based on node ID.

Source: .NET Core's Internal PriorityQueue (now .NET 5+):

https://github.com/dotnet/runtime/blob/main/src/libraries/System.Collections/src/System/Collections/Generic/PriorityQueue.cs

```
public class PriorityQueue<T>
{
    private List<(T item, double priority)> heap = new List<(T, double)>();
    public int Count => heap.Count;

    public void Enqueue(T item, double priority)
    {
        heap.Add((item, priority));
        HeapifyUp(heap.Count - 1);
    }

    public T Dequeue()
    {
        if (heap.Count == 0)
            throw new InvalidOperationException("The queue is empty.");
        T result = heap[0].item;
        heap[0] = heap[heap.Count - 1];
        heap.RemoveAt(heap.Count - 1);
        if (heap.Count > 0)
            HeapifyDown(0);
        return result;
    }

    public T Peek()
    {
        if (heap.Count == 0)
            throw new InvalidOperationException("The queue is empty.");
        return heap[0].item;
    }

    public bool Contains(T item)
    {
        return heap.Exists(pair => EqualityComparer<T>.Default.Equals(pair.item,
item));
    }

    public void Remove(T item)
    {
        int index = heap.FindIndex(pair =>
EqualityComparer<T>.Default.Equals(pair.item, item));
        if (index < 0)
            return;

        heap[index] = heap[heap.Count - 1];
        heap.RemoveAt(heap.Count - 1);

        if (index < heap.Count)
        {
            HeapifyDown(index);
            HeapifyUp(index);
        }
    }

    public void UpdatePriority(T item, double newPriority)
    {
        int index = heap.FindIndex(pair =>
EqualityComparer<T>.Default.Equals(pair.item, item));
        if (index < 0)
```

```
            return;

        double oldPriority = heap[index].priority;
        heap[index] = (item, newPriority);

        if (newPriority < oldPriority)
            HeapifyUp(index);
        else
            HeapifyDown(index);
    }

    private void HeapifyUp(int index)
    {
        while (index > 0)
        {
            int parent = (index - 1) / 2;
            if (heap[index].priority >= heap[parent].priority)
                break;
            (heap[index], heap[parent]) = (heap[parent], heap[index]);
            index = parent;
        }
    }

    private void HeapifyDown(int index)
    {
        int lastIndex = heap.Count - 1;
        while (index < heap.Count)
        {
            int left = index * 2 + 1;
            int right = index * 2 + 2;
            int smallest = index;
            if (left <= lastIndex && heap[left].priority < heap[smallest].priority)
                smallest = left;
            if (right <= lastIndex && heap[right].priority <
heap[smallest].priority)
                smallest = right;
            if (smallest == index)
                break;
            (heap[index], heap[smallest]) = (heap[smallest], heap[index]);
            index = smallest;
        }
    }
}
```

---

3. Multi-Source Shortest Path Algorithm:

Our algorithm implements a modified version of Dijkstra's algorithm to solve a constrained shortest path problem with multiple source points. The approach optimizes travel time while ensuring distance constraints are satisfied.

Algorithm Steps

1. **Initialization:**

- Identify all road intersections within a Euclidean distance radius r*r* from the starting position as potential source points.
- For each valid intersection, compute the walking time from the origin and initialize the priority queue with these entries.

2. **Execution & Path Optimization:**
   - Execute a multi-source Dijkstra's algorithm, propagating shortest-path distances from all source intersections simultaneously**.**
   - Maintain a priority queue to efficiently explore the most promising paths first.
   - For each intersection encountered during traversal, check if it lies within walking distance *R* of the destination.
   - If valid, compute the total travel time as:
     - Total Time=Path Time (walk + drive) +Final Walk Time (to destination)
   - Track the minimum-time path while pruning suboptimal branches.
3. **Solution Selection:**
   - Compare all candidate paths that satisfy the distance constraints.
   - Select the path with the minimal total travel time, which may include:
     - Walking directly to the destination (if faster).
     - A combination of walking and driving through the optimal road intersection**.**

Optimizations:

- Early Termination: Halt exploration if remaining paths cannot improve the current best solution.
- Distance Precomputation: Avoid repeated square root calculations by working with squared distances during feasibility checks.
- Direct Walk Check: Evaluate if walking the entire route is faster than any driving path before running the full algorithm.

Component Analysis

Pathfinder Algorithm

1. **Helper Functions**

- **leq(double a, double b**): O (1) (simple floating-point comparison)
- **SquaredDistance(Point a, Point b):** O (1) (fixed arithmetic operations)
- **EuclideanDistance(Point a, point b):** O (1) (includes square root)

2. **Input Processing**
    - **Intersections (Nodes):** O(|V|) (storing all intersections)
    - **Roads (Edges):** O(|E|) (building adjacency lists)
3. **Per-Query Operations**
    - **Initialization:**
        - **dist and parent arrays:** O(|V|)
    - **Precomputing Euclidean Distances:**
        - **For all intersections:** O(|V| log |V|) (if sorted for proximity checks)
    - **Priority Queue Setup:**
        - **Adding valid start intersections:** O(|V| log |V|) (worst-case, if all are within walking distance)
    - **Multi-Source Dijkstra's Algorithm:**
        - O((|V| + |E|) log |V|) (standard Dijkstra with a priority queue)
        - Each node processed once: O(|V| log |V|)
        - Each edge relaxed once: O(|E| log |V|)
    - **Brute-Force Best Path Selection:**
        - Checking all intersections for minimal total time: O(|V|) (since Euclidean distances are precomputed)
    - **Path Reconstruction:**
        - Backtracking from destination: O(|V|) (worst-case, linear path)
    - **Walking Distance Calculation:**
        - Binary search on precomputed distances: O(log |V|)
    - **Output Path:**

        - Printing the path: O(|V|) (worst-case, linear path)

4. **Overall Complexity for Q Queries**
    - **Best Case (Sparse Graph, $|E| \approx |V|$):**
        - $O(Q\ |E|\ \log\ |V|)$
    - **Worst Case (Dense Graph, $|E| \approx |V|^2$):**
        - $O(Q\ |V|^2\ \log\ |V|)$

Space Complexity Analysis

1. **Graph Representation**
    - **Intersections (Nodes):** O(|V|) (storing coordinates)
    - **Roads (Edges):** O(|V| + |E|) (adjacency lists)
2. **Algorithm State**
    - **dist array**: O(|V|)
    - **parent array**: O(|V|)
    - **Priority queue:** O(|V|) (worst-case)
3. **Path Storage**
    - **Reconstructed path:** O(|V|) (worst-case, linear path)
4. **Precomputed Distances**
    - **Walking distances for intersections:** O(|V|)

Total Space Complexity: O(|V| + |E|)

Priority Queue:

Time Complexities:

1. **Enqueue(item, priority)**
    - **O(1)** for adding to the end of the list

    - **O(log n)** for heapify-up (bubble up through tree height)
    - **Total: O(log n)**
2. **Dequeue()**
    - **O(1)** for accessing the root and swapping the last element
    - **O(log n)** for heapify-down (bubble down through tree height)
    - **Total: O(log n)**
3. **Peek()**
    - **O(1)** - Just accesses the first element
4. **Contains(item)**
    - **O(n)** - Linear search through all elements (no indexing)
5. **Remove(item)**
    - **O(n)** to find the item (linear search)
    - **O(log n)** for heapify-up or down after removal
    - **Total: O(n)** (dominated by the search)
6. **UpdatePriority(item, newPriority)**
    - **O(n)** to find the item
    - **O(log n)** for heapify-up or down
    - **Total: O(n)** (dominated by the search)
7. **HeapifyUp/HeapifyDown**
    - Both **O(log n)** as they traverse the tree height

Space Complexity:

- **O(n)** - Linear space for storing all elements in a list