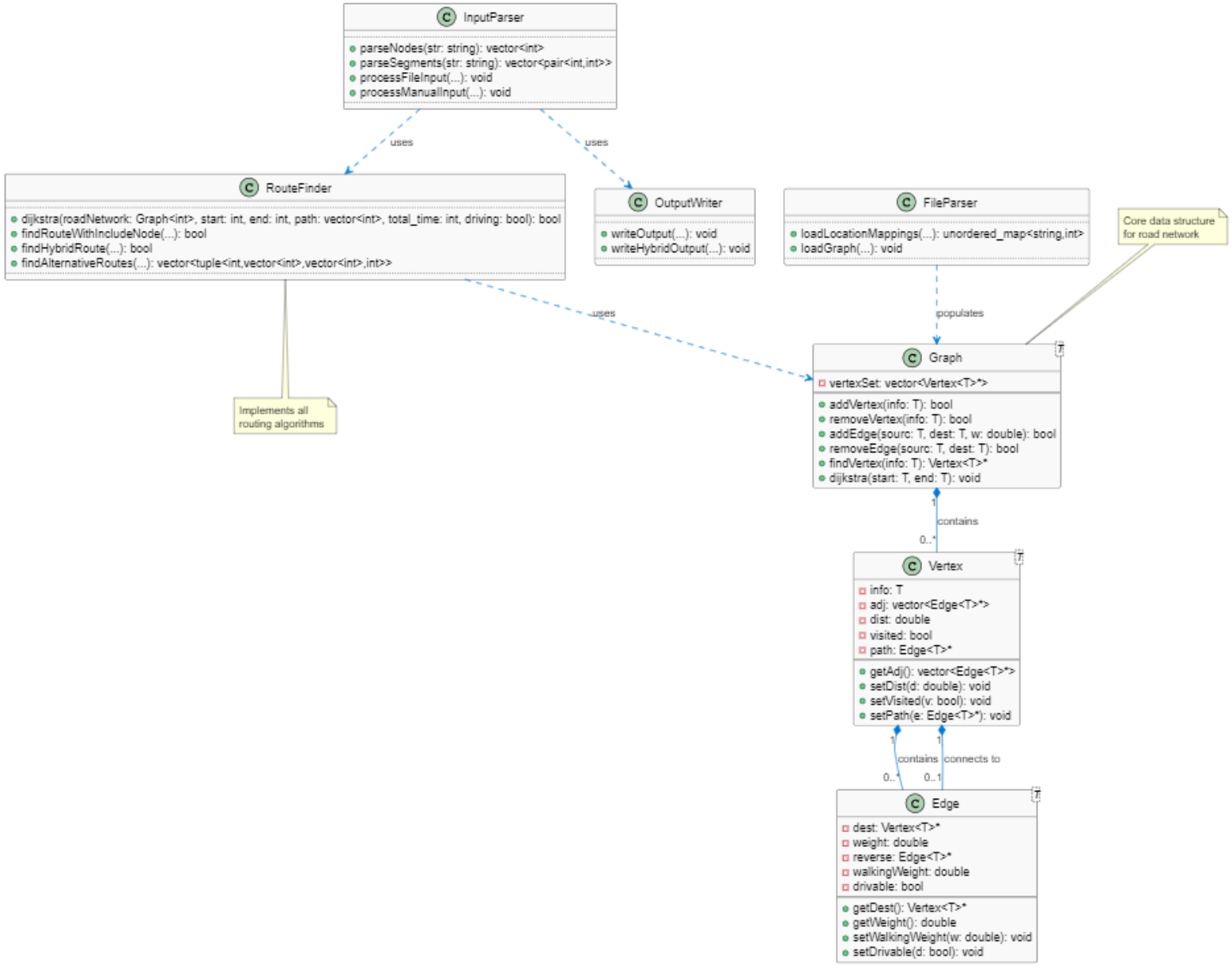
Class Diagram and Involved Files



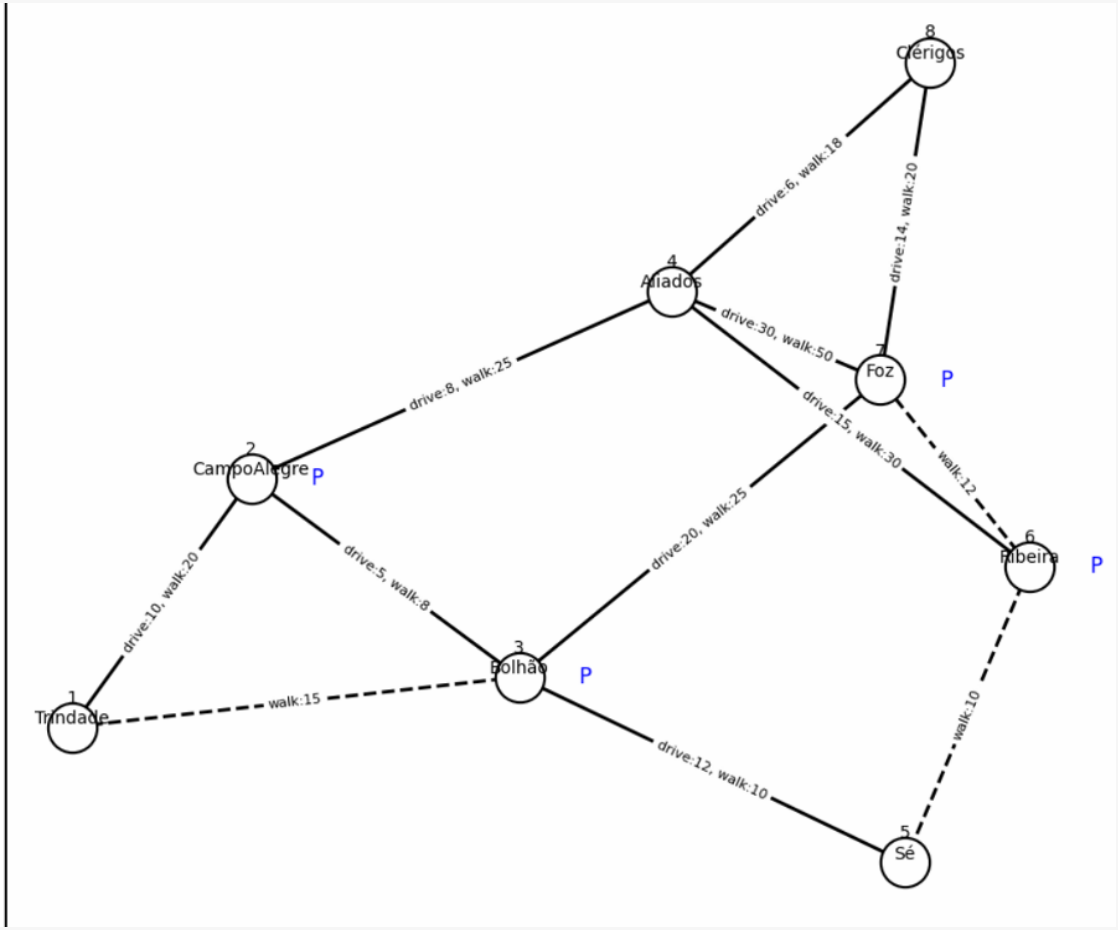
Dataset Reading Description

- **Input Files**
- **`locations.csv`**
  - Columns: `Description, ID, Code, Is_Parking`
  - Maps location codes to unique IDs (e.g., `"Lisbon", 1, "LIS", 1`)
- **Distances.csv**
  - Columns: `From_Code, To_Code, Driving_Time, Walking_Time`
  - Defines connections and travel times (e.g., `"LIS", "POR", 30, 45`)

Dataset Reading Description

```cpp
unordered_map<string, int> FileParser::loadLocationMappings(
    const string& filename, unordered_map<int, bool>& parkingData) {

    unordered_map<string, int> codeToId;
    ifstream file(filename);
    if (!file.is_open()) throw runtime_error( arg: "ERRO: Não foi possivel abrir " + filename);
    string line;
    getline( [&] file, [&] line); // Skip header

    while (getline( [&] file, [&] line)) {
        stringstream ss(line);
        vector<string> tokens;
        string token;

        while (getline( [&] ss, [&] token, delim: ',')) tokens.push_back(token);

        if (tokens.size() >= 4) {
            try {
                int id = stoi(tokens[1]);
                string code = tokens[2];
                bool isParking = (tokens[3] == "1");
                codeToId[code] = id;
                parkingData[id] = isParking;
            } catch (...) {
                throw runtime_error( arg: "ERRO: Formato inválido no arquivo de localizações: " + line
            }
        }
    }
    return codeToId;
}
```

- codeToId: {"LIS" → 1, "POR" → 2, ...}
- parkingData: {1 → true, 2 → false, ...}



Graph Representation of the Dataset

- Edge:
  - Double walkingWeight

    ```
    getWalkingWeight()
    SetWalkingWeight()
    ```

  - Bool drivable;

    ```
    setDrivable();
    IsDrivable();
    ```

What differs?

- Enables **hybrid routing** by tracking separate costs.
- Models **restricted roads** using `drivable=false`.

Implemented Functionalities

1. Transport Mode Selection

- **Driving-Only Routes**

- **Mode:driving** in input.txt or CLI selection
  - Uses road network with `drivable=true` edges only
  - Optimizes for shortest driving time
- **Hybrid Driving+Walking**
  Mode:driving-walking
  - Finds optimal parking spot near destination
  - Ensures walking segment ≤ specified MaxWalkTime

2. Custom Route Restrictions

- **Node Avoidance**
  AvoidNodes: 3,7,12
  - Automatically excludes specified locations
  - Uses `removeVertex()` internally
- **Road Segment Blocks**
  AvoidSegments: (1,2)(5,8)
  - Closes specific roads in both directions
  - Updates graph with `removeEdge()`

- **3. Alternative Path Suggestions**
- **When Primary Route Fails**
  - Shows up to 2 fallback options
  - Displays:

```
Option 1: Drive 12min + Walk 8min (Total: 20min)
Option 2: Drive 15min + Walk 3min (Total: 18min)
```

Interactive Choice:

```
cout << "View alternatives? (y/n): ";
cin >> choice;  // User-controlled output
```

- **4. Special Requirements**
- **Mandatory Stops**
  IncludeNode:5
  - Forces route through specified point
  - Uses two-pass Dijkstra's algorithm

Implemented Functions (Time Complexities)

Algorithm Name	Purpose	Time Complexity	Notes
loadLocationMappings	Loads location mappings from CSV and identifies parking spots	O(n)	n = number of lines in CSV file
loadGraph	Builds graph from road network CSV data	O(n + e)	n = nodes, e = edges. Handles both driving and walking weights
parseNodes	Parses comma-separated node list from string input	O(k)	k = number of nodes in input string
parseSegments	Parses segment pairs from formatted string input	O(k)	k = number of segments in input string
OutputWriter::writeOutput	Writes driving route results to file	O(n)	n = number of nodes in the path
OutputWriter::writeHybridOutput	Writes hybrid (driving+walking) route results to file	O(n + m)	n = driving nodes m = walking nodes

Implemented Functionalities (Time Complexities)

Algorithm Name	Purpose	Time Complexity	Notes
dijkstra	Finds shortest path using Dijkstra's algorithm	O((V+E)logV)	V = vertices, E = edges. Uses priority queue. Handles driving/walking modes
dijkstra + avoidNodes	Find Shortes path with Dijkstra's algorithm after removing nodes or segments	O(k*E + (V'+E')logV')	V' = V - k, E' ≈ E - arestas removidas, k avoided nodes
findRouteWithIncludeNode	Finds path that must include specific node via two Dijkstra runs	O((V+E)logV)	Combines paths from start→node and node→end
findHybridRoute	Finds optimal driving+walking route with parking constraints	O(P*(V+E)logV)	P = parking nodes. Worst case checks all parking spots
findAlternativeRoutes	Finds backup routes when primary fails	O(P*(V+E)logV)	Returns top 2 alternatives by total time

User Interface and Usage Example

- 1: Select mode
- 2: Enter Nodes
- 3: Select Avoidances (Optional)
- 4: OutputResult

```
=== BATCH MODE ===

=== MENU PRINCIPAL ===
1. Executar com arquivo input.txt (gera output.txt)
2. Inserir parametros manualmente
3. Sair
Escolha:2


=== ENTRADA MANUAL ===
Insira os parametros (deixe em branco para usar o padrão):
Modo (driving/driving-walking):driving-walking

No de origem:8

No de destino:5

Tempo maximo a pe (minutos):15

Nos a evitar (separados por virgula, ou deixe em branco):1

Segmentos a evitar (formato (a,b)(c,d), ou deixe em branco):


=== RESULTADO ===
Rota encontrada!
Trajeto de carro: 8 4 2 3
Estacionamento: 3
Trajeto a pe: 3 5
Tempo total: 29
```

```
=== VALORES ===
Modo: driving-walking
Origem: 8 -> Destino: 5

=== RESULTADO ===
Minimo tempo necessario (10) maximo tempo permitido (3)

Existem rotas alternativas disponiveis.
Deseja visualizar as rotas alternativas? (s/n):s


Rotas alternativas disponiveis:
Opcao 1:
  Estacionamento: 3
  Carro (19min): 8 4 2 3
  A pe (10min): 3 5
  Total: 29min
Opcao 2:
  Estacionamento: 6
  Carro (21min): 8 4 6
  A pe (10min): 6 5
  Total: 31min
```

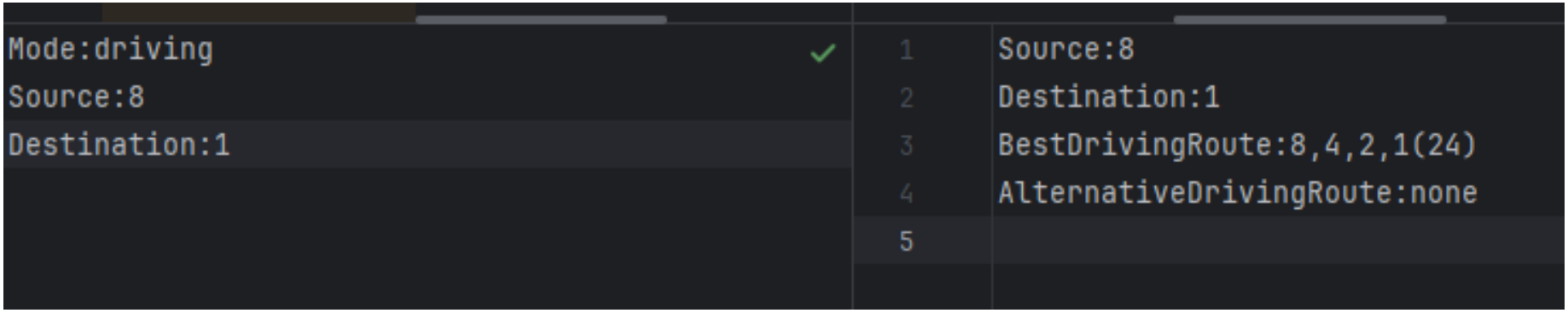
```
=== ENTRADA MANUAL ===
Insira os parametros (deixe em branco para usar o padrão):
Modo (driving/driving-walking):driving

No de origem:2

No de destino:4

No a incluir (ou deixe em branco):7

Nos a evitar (separados por virgula, ou deixe em branco):

Segmentos a evitar (formato (a,b)(c,d), ou deixe em branco):


=== RESULTADOS ===
Melhor rota (45 min): 2 -> 3 -> 7 -> 8 -> 4
```

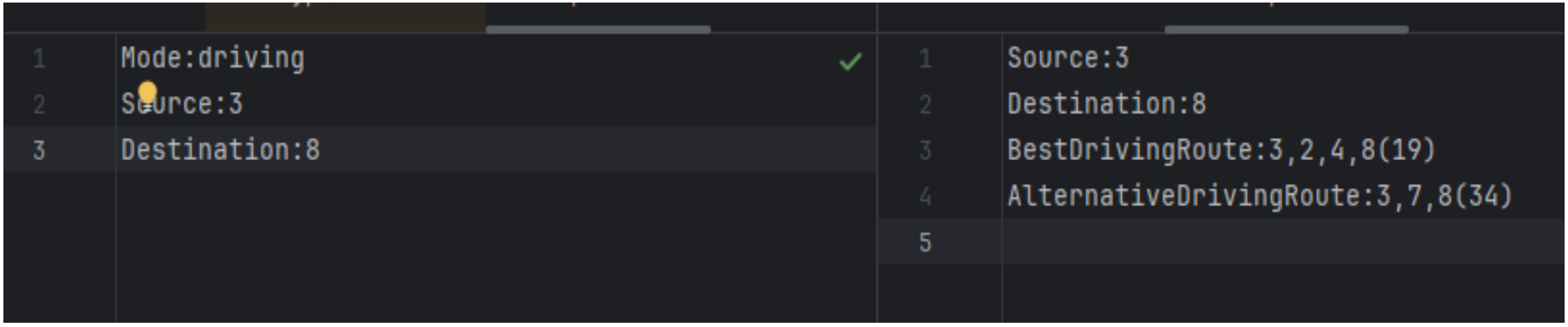
Highlighted Functionalities

- **`FindRouteWithIncludeNode algorithm:`**

Took me a while to figure out a way to make it. It Runs Dijkstra from start the node to be included and from there to the end. After that combines both path's. Decomposition of the problem made it, but I had a hard time with it.

Independent Route Tests





Restricted Route Planning

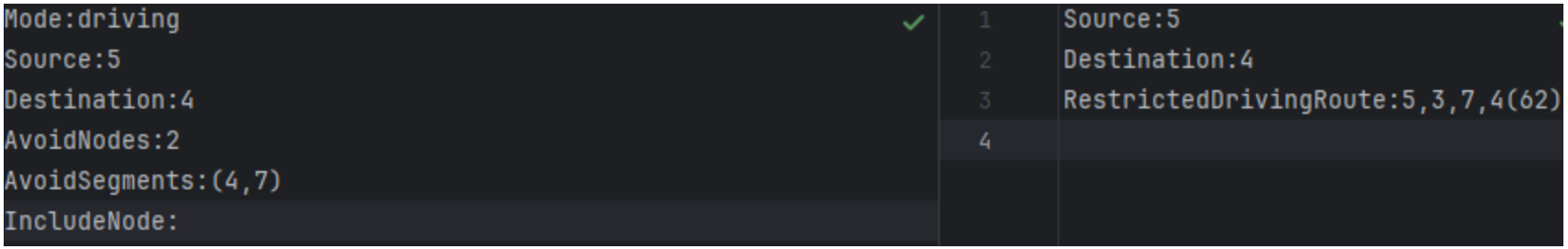
```
Mode:driving
Source:5
Destination:4
AvoidNodes:2
AvoidSegments:
IncludeNode:
```

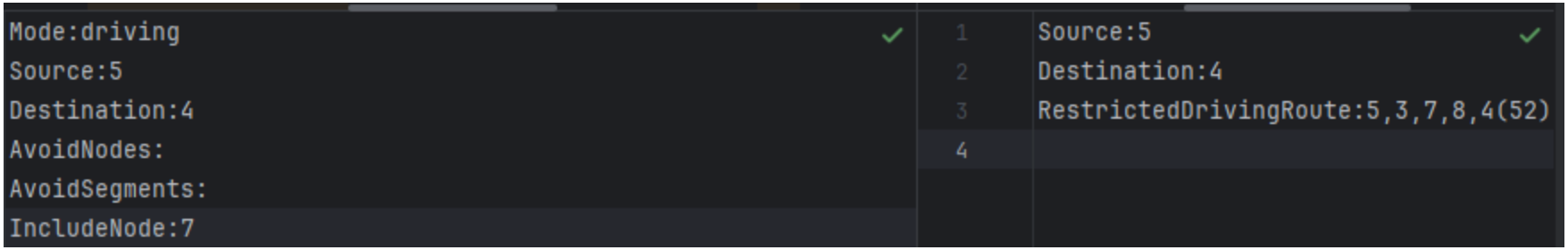
```
1  Source:5
2  Destination:4
3  RestrictedDrivingRoute:5,3,7,8,4(52)
4
```

```
1  Mode:driving
2  Source:5
3  Destination:4
4  AvoidNodes:
5  AvoidSegments:(3,2),(7,8)
6  IncludeNode:
```

```
1  Source:5
2  Destination:4
3  RestrictedDrivingRoute:5,3,7,4(62)
4
```

Restricted Route Planning Cont.





Environmentally-Friendly Route Planning (driving and walking)