Maze Search Algorithm

The **Maze Search Algorithm** presented in the classes **MazeSearchAlgo**, **MazeListElement**, **MazeSearchElement**, and **MazeShoveTraceAlgo** is an implementation of a routing algorithm that seeks to establish connections between incomplete nets on a printed circuit board (PCB) using a maze-expansion-based approach.

This algorithm deals with complex situations, such as routing around obstacles (traces, vias, components) and optimizing path lengths and vias for efficiency in the final route.

This documentation outlines the following aspects:

1. **Overview of the Algorithm**
2. **Key Classes**
3. **How the Classes Interact**
4. **Flow of the Algorithm**
5. **Key Decisions and Optimizations**
6. **Use of the Algorithm in the Application**

**1. Overview of the Algorithm**

The **Maze Search Algorithm** is used to route incomplete connections on a PCB. The primary goals of the algorithm are to:

* Find valid paths (routes) from a set of **starting points** (pins or existing traces) to a set of **destination points** while considering PCB design rules (trace widths, clearances, and via constraints).
* Avoid obstacles such as other traces, vias, and components.
* Optimize the path to minimize the number of vias (layer changes) and overall trace length.
* Handle complex routing scenarios like narrow passages, corners, and fanout from surface-mounted devices (SMDs).
* The algorithm expands routes incrementally by evaluating each possible connection point (called **Expansion Rooms**) and adding them to a priority queue.

**2. Key Classes**

1. **MazeSearchAlgo**:
   * The main class responsible for controlling the maze search.
   * Holds the state of the search, including the list of expanded elements, search tree, and search distance calculation.
   * Implements the core algorithm to find a connection route between start and destination points.
2. **MazeListElement**:
   * Represents an element in the maze expansion list.
   * Each element corresponds to a **door** or **drill** (i.e., a point on the route where an expansion can occur).
   * Tracks the current expansion state, including the expansion value, the backtrack path, and any rooms or traces that have been "ripped up" (i.e., removed for re-routing).
3. **MazeSearchElement**:
   * Represents the state of a specific section or "door" in the expansion tree.
   * Tracks whether the door is occupied and whether any adjustment (e.g., shoving the trace to make space) has been applied.
   * Supports operations like backtracking and adjusting the expansion path.
4. **MazeShoveTraceAlgo**:
   * An auxiliary class that handles "shoving" traces—moving traces slightly to make space for routing new connections.
   * Used when a trace room (an area around a trace) needs to be pushed to accommodate a new route.
   * Contains logic to check if a trace line can be shoved and calculates which segments or sections of the trace can be adjusted.

**3. How the Classes Interact**

* **Initialization**:
  + The algorithm is initialized using **MazeSearchAlgo.get\_instance()**, which sets up the starting conditions (i.e., sets of start and destination points) and reduces the trace shapes around pins that might have multiple nets.
  + The **MazeSearchAlgo** instance builds the search tree, adds start rooms to the maze expansion list, and begins the expansion process.
* **Maze Expansion**:
  + **MazeSearchAlgo.occupy\_next\_element()** expands elements from the priority queue, one by one.
  + Each **MazeListElement** represents a possible expansion point (door or drill). The search algorithm evaluates whether the expansion improves the current route and checks if the destination has been reached.
  + If a **MazeListElement** corresponds to a **destination door**, the algorithm finishes, and the route is found.
  + If not, the algorithm continues to explore new paths, possibly expanding through other layers (vias) or shoving traces using **MazeShoveTraceAlgo**.
* **Shoving and Ripup**:
  + If a trace or via is in the way, the algorithm may attempt to "shove" the trace using **MazeShoveTraceAlgo**. This means adjusting the trace slightly to make room for the new route.
  + If shoving fails, the algorithm may resort to **ripping up** the trace (removing it temporarily) to find a better route.

**4. Flow of the Algorithm**

**4.1 Initialization**

1. **Initialization**:
   * The algorithm starts with the **get\_instance()** method, which takes in the sets of **start items** and **destination items**.
   * The trace shapes are reduced at tie pins (pins with multiple nets) to ensure the pin is not blocked.
2. **Building the Expansion List**:
   * **Expansion Rooms** are created for each start item. These rooms represent areas on the PCB where expansion (routing) can occur.
   * Each expansion room has **Expansion Doors**, which are potential entry points for new traces.
   * **Completed Expansion Rooms** are also added to the expansion list.

**4.2 Maze Expansion**

1. **Expansion Process**:
   * The algorithm works by iteratively expanding from **start items**. It evaluates each **Expansion Door** to see if expanding through it brings the route closer to the **destination**.
   * Each **MazeListElement** in the expansion list is processed in order of **sorting value**, which is a combination of the current expansion distance and an estimate of the remaining distance to the destination.
2. **Handling Obstacles**:
   * As the maze expansion proceeds, obstacles (e.g., traces, vias, components) are encountered. The algorithm attempts to **shove** or **rip up** traces that block the path, using the **MazeShoveTraceAlgo** to adjust the traces.
3. **Vias and Layer Changes**:
   * When routing needs to change layers (for example, moving from a top-layer trace to a bottom-layer trace), the algorithm inserts vias.
   * The algorithm tries to minimize the number of vias, but if necessary, it will route through multiple layers using the **expand\_to\_other\_layers()** method.

**4.3 Completing the Route**

1. **Destination Found**:
   * Once an expansion reaches a **destination door**, the algorithm stops, and the complete route is backtracked using the **backtrack\_door** property in each **MazeListElement**.
2. **Finalizing the Route**:
   * The algorithm completes the routing process by ensuring that all traces, vias, and components are correctly connected.

**5. Key Decisions and Optimizations**

* **Ripup Costs and Shoving**: The algorithm balances **shoving traces** and **ripping up traces**. Shoving is preferred when slight adjustments can be made, but ripping up may be necessary when no valid route exists.
* **Randomization for Avoiding Loops**: When the algorithm encounters repetitive loops, it introduces randomization to prevent the same failed paths from being explored multiple times.
* **Layer Transitions via Vias**: The algorithm uses vias efficiently, aiming to minimize the number of vias used while ensuring the routing stays within design rules.
* **Destination Distance Calculation**: The **DestinationDistance** object calculates the estimated remaining distance to the destination, helping to prioritize expansion directions.

**6. Use of the Algorithm in the Application**

The **Maze Search Algorithm** is a core part of the **FreeRouting** PCB routing engine. It is used to:

* **Automatically route incomplete connections** after the initial placement of components.
* **Optimize existing routes** by minimizing trace lengths and the number of vias, thus improving PCB design quality.
* **Handle complex routing scenarios**, such as navigating tight spaces between components, traces, and vias.

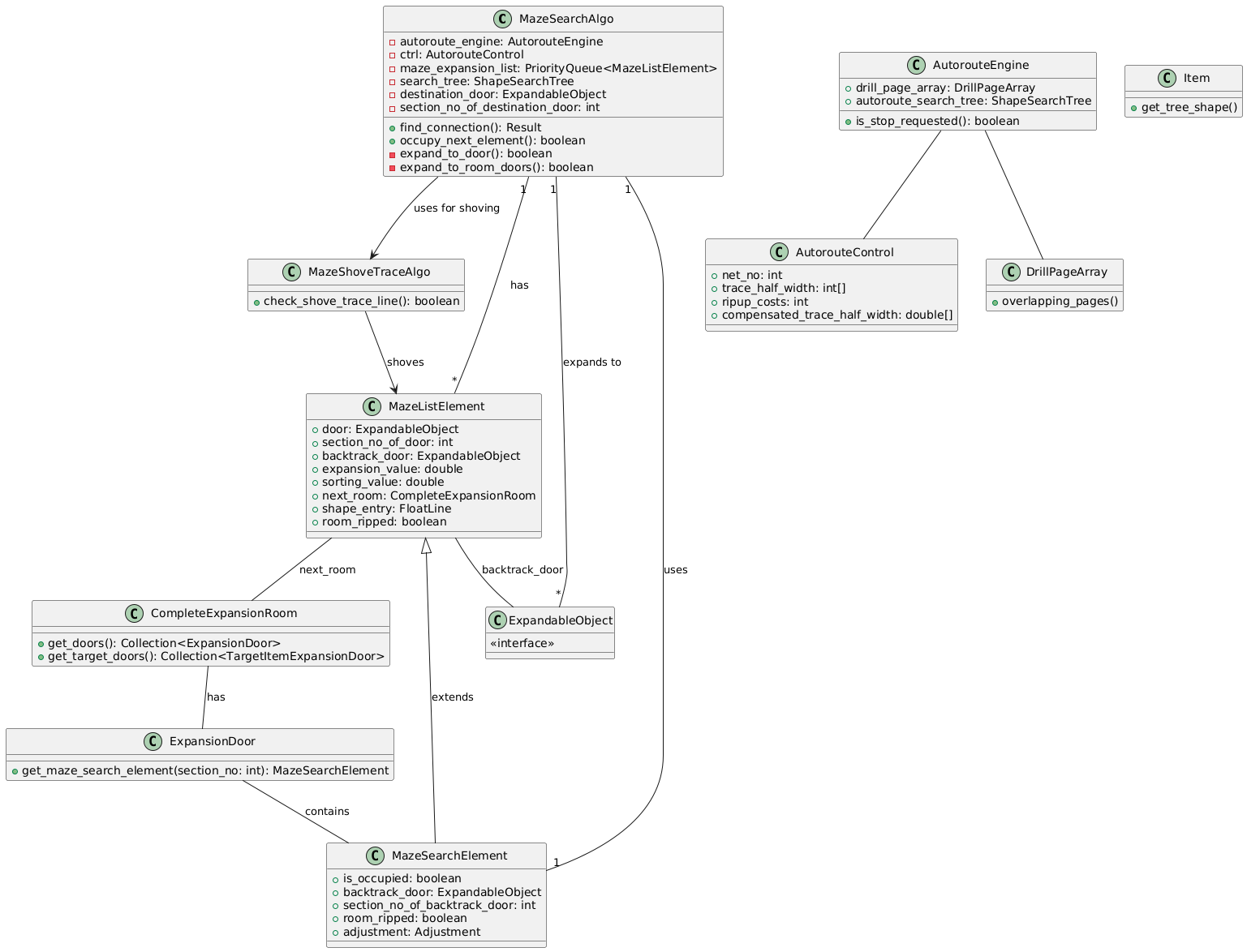
This algorithm allows the FreeRouting tool to handle large and complex PCBs, providing high-quality routing results even in challenging design environments. It operates in conjunction with other routing strategies in the FreeRouting engine to ensure comprehensive and efficient auto-routing capabilities.

**UML Diagrams:**

**1. Class Diagram**

The class diagram showcases the relationships between the primary components:

* **MazeSearchAlgo** is the central controller of the algorithm. It interacts with various expansion rooms, doors, and search elements.
* **MazeListElement** is used to represent each possible expansion point during the routing.
* **MazeSearchElement** tracks the status of each door during the expansion.
* **MazeShoveTraceAlgo** is used when traces need to be adjusted to accommodate new routes.
* Various helper classes such as **ExpandableObject**, **CompleteExpansionRoom**, and **ExpansionDoor** are part of the expansion process.



**2. Sequence Diagram**

The sequence diagram shows the step-by-step process of routing an incomplete net:

1. The user initiates the routing process by calling **find\_connection()** on the **MazeSearchAlgo**.
2. The algorithm iterates through the expansion list (**MazeListElement**), attempting to expand each element.
3. If an obstacle is encountered, it calls **MazeShoveTraceAlgo** to attempt to shove the trace or handle rip up if needed.
4. The algorithm continues expanding until the destination is reached or all possibilities are exhausted.

