AutoRoute Folder

## AutoRouteControl.java:

The AutorouteControl.java class manages the configuration and parameters for controlling the **autoroute algorithm** in a PCB (Printed Circuit Board) routing tool. The auto router algorithm automatically calculates how traces (wires) are routed between various components, following constraints and optimizations.

## Fields (Variables) Defined in the Class:

**Trace Costs and Layer Configuration**

* **trace\_costs**: An array of **ExpansionCostFactor** objects representing the cost of expanding a trace horizontally or vertically on each layer of the PCB. The algorithm tries to minimize these costs when routing traces.
* **with\_neckdown**: A flag indicating whether **neckdown** is allowed. Neckdown refers to reducing the width of traces in narrow or constrained areas.
* **layer\_active**: A boolean array indicating whether each layer on the PCB is **active** for routing. If false for a layer, the autorouter won’t use it.
* **layer\_count**: The total number of layers in the PCB (top, bottom, internal layers, etc.).
* **trace\_half\_width**: The half-width of traces used on each layer during routing.
* **compensated\_trace\_half\_width**: The half-width of traces **after compensating** for clearance (distance from other traces or objects).

**Via Configuration and Cost Calculation**

* **via\_radius\_arr**: Stores the **via radius** (size) on each layer. Vias are small holes that connect traces between layers of the PCB.
* **add\_via\_costs**: Represents the additional costs associated with inserting a via between two layers. The autorouter tries to minimize these costs.
* **trace\_clearance\_class\_no**: Identifies the **clearance class** (spacing rules) used by the traces on the PCB. Different nets may have different clearance requirements based on electrical rules.
* **vias\_allowed**: If **true**, vias are allowed to be inserted to change layers during routing.
* **attach\_smd\_allowed**: If **true**, vias can be inserted and drill into the **Surface-Mount Device (SMD)** pins.
* **min\_normal\_via\_cost**: The **minimum cost** of inserting a normal via (determined based on radius and layer count).

**Ripup and Retry Settings**

* **ripup\_allowed:** If true, the autorouter is allowed to ripup (remove) previously routed traces if it needs to find a better route.
* **ripup\_costs:** The cost of ripping up a trace during rerouting.
* **ripup\_pass\_no:** The number of passes allowed for ripup and retry attempts.

**Routing Parameters**

* **is\_fanout:** If true, the autorouter completes the process after the first fanout pass. Fanout refers to spreading traces out from a common point, often in BGA (Ball Grid Array) routing.
* **remove\_unconnected\_vias:** If true, the autorouter removes unconnected vias at the end of the process.
* **net\_no:** The current net number that the autorouter is working on. Each net represents a set of connected electrical pins.
* **via\_clearance\_class:** The clearance class specifically for vias, determining how far vias must be spaced from other traces or components.
* **via\_rule:** Defines the rules for using vias (e.g., how they are inserted, allowed layers).
* **via\_info\_arr:** An array of ViaMask objects that hold information about the via’s range and layer.

**Additional Parameters for Trace Shoving and Optimization**

* **tidy\_region\_width:** Defines the width of the region around traces where the autorouter will attempt to tidy (optimize) trace placement**.**
* **pull\_tight\_accuracy:** A value that determines how accurately traces are pulled tight around obstacles.
* **max\_shove\_trace\_recursion\_depth:** The maximum recursion depth for shoving traces (i.e., pushing traces to make room for new routes).
* **max\_shove\_via\_recursion\_depth:** Similar to the above, but for vias.
* **max\_spring\_over\_recursion\_depth**: The maximum recursion depth for traces that are allowed to spring over obstacles.
* **min\_cheap\_via\_cost:** The minimum cost of cheap vias, which are lower-cost alternatives to normal vias.

## Constructor Methods:

**Main Constructor (AutorouteControl) :**

**public AutorouteControl(RoutingBoard p\_board, int p\_net\_no, Settings p\_settings)**

* This constructor initializes the autorouter control for a given net on the RoutingBoard.
* It uses the board settings and autoroute settings from the provided Settings object to set up the autorouter's behavior.

**Secondary Constructor:**

**public AutorouteControl(RoutingBoard p\_board, int p\_net\_no, Settings p\_settings, int p\_via\_costs, ExpansionCostFactor[] p\_trace\_cost\_arr)**

* A more advanced constructor that allows specification of **trace costs** and **via costs** explicitly when initializing.

**Private Helper Method:**

**private void init\_net(int p\_net\_no, RoutingBoard p\_board, int p\_via\_costs)**

* The constructors call this method to initialize the control for a specific **net**
* It retrieves information like **trace width**, **clearance classes**, and **via rules** from the board’s rules and configures the autorouter accordingly
* It also calculates the costs for each layer via radii.

## Helper Classes:

**ExpansionCostFactor:**

* This class holds the **horizontal** and **vertical cost factors** for routing on a specific board layer.
* The autorouter uses these values to minimize the cost of trace expansion in different directions.

**ViaCost:**

* Holds an array of via costs for transitioning from one layer to another.
* Each layer has a different cost associated with inserting a via.

**ViaMask:**

* Describes a via's range in terms of layers, and whether the via is allowed to drill into SMD pins**.**

**Summary:**

* The AutorouteControl class is responsible for managing all aspects of the autoroute algorithm in a PCB routing program.
* It handles the configuration of trace widths, via rules, layer activation, and cost calculations to control how the autorouter lays out traces and vias.
* It interacts with the RoutingBoard and Settings objects to dynamically adapt its behavior for different nets and board configurations.

## AutorouteEngine.java

The AutorouteEngine.java class is a core part of the PCB autorouting system, responsible for executing the autorouting algorithm, maintaining the search trees and expansion areas, and managing routing paths and costs.

It leverages data from the **RoutingBoard**, controls the expansion of routes, and coordinates the process of finding connections between electrical components on a PCB.

### Fields (Variables) Defined in the Class:

**Static Constant:**

* **TRACE\_WIDTH\_TOLERANCE**: This is a static constant that defines how much tolerance is allowed in the width of traces (electrical paths) when routing.

**Autoroute Search Tree and Drill Page Array:**

* **autoroute\_search\_tree:** This holds the search tree that is used by the autorouter to explore possible routing paths on the PCB. This data structure is critical for efficiently managing the areas being searched during routing.
* **drill\_page\_array:** Represents a two-dimensional array of drill pages. A drill page is a section of the board used to track where holes (vias) and other drilling actions will take place.

**Routing Board and Control:**

* **board:** Refers to the PCB board currently being routed. The autorouter interacts with the board’s layers, nets, and components to create the routing paths.
* **stoppable\_thread:** This thread can stop the autorouting algorithm prematurely if needed.
* **net\_no**: The current net number being routed. A net refers to a group of connected electrical nodes (e.g., pins, pads) that traces should connect.
* **time\_limit**: A limit set on how long the autorouting process should take before being stopped.

**Expansion Rooms and Performance Management:**

* **incomplete\_expansion\_rooms:** A list of incomplete expansion rooms. These rooms represent areas where traces are currently being expanded but the routing isn’t complete yet.
* **complete\_expansion\_rooms:** A list of complete expansion rooms, which represent areas where routing has finished successfully.
* **expansion\_room\_instance\_count:** Tracks the number of expansion rooms created so far. This helps the autorouter keep track of its progress and manage different expansion attempts.

### Constructors:

**Main Constructor:**

**public AutorouteEngine(RoutingBoard p\_board, int p\_trace\_clearance\_class\_no, boolean p\_maintain\_database)**

* This constructor initializes the **AutorouteEngine** with the given **RoutingBoard** and trace clearance class.
* **maintain\_database**: A boolean indicating whether the autorouter’s database should be retained after a connection is made for better performance.
* It creates an autoroute search tree and sets up the drill page array based on the PCB board’s properties.

**init\_connection Method:**

**public void init\_connection(int p\_net\_no, Stoppable p\_stoppable\_thread, TimeLimit p\_time\_limit)**

* Initializes a routing connection for the given **net\_no**.
* It invalidates existing incomplete rooms for the net, removes any old connections, and sets the time limit for autorouting. The thread can be stopped prematurely if required by using **p\_stoppable\_thread**.

### Core Autorouting Functionality:

**autoroute\_connection Method:**

**public AutorouteResult autoroute\_connection(Set<Item> p\_start\_set, Set<Item> p\_dest\_set, AutorouteControl p\_ctrl, SortedSet<Item> p\_ripped\_item\_list)**

* This core function is responsible for **autorouting** a connection between the source (**p\_start\_set**) and destination (**p\_dest\_set**).
* **Routing Process:**
  + It starts by initializing the maze search algorithm, which attempts to find a connection between the start and destination sets.
  + If the connection is found, **LocateFoundConnectionAlgo** is used to insert the connection into the routing board.
  + The function also handles removing any ripped-up traces and notifies observers of changes to the board
* The method returns an enum value (**ROUTED**, **NOT\_ROUTED**, **ALREADY\_CONNECTED**, or **INSERT\_ERROR**), which indicates the status of the autorouting operation.

**MazeSearchAlgo:**

* This is an algorithm used to find the path (trace) between two points. The result of this search is then passed to the next stage where the routing result is processed.

**InsertFoundConnectionAlgo**:

* Once the search has found a valid path, this algorithm inserts the newly routed connection into the PCB board.

**Handling Ripped Connections:**

**for (Item curr\_ripped\_item : p\_ripped\_item\_list)**

* When a trace is rerouted, previous connections might be ripped up or removed to make space for new routes. The algorithm keeps track of these removed traces and updates the board accordingly.

## Managing Expansion Rooms:

**add\_incomplete\_expansion\_room Method:**

**public IncompleteFreeSpaceExpansionRoom add\_incomplete\_expansion\_room(TileShape p\_shape, int p\_layer, TileShape p\_contained\_shape)**

* This method creates a new incomplete expansion room in which the autorouter is currently working. The room’s shape is usually unbounded when it’s first created, and it’s expanded as the router progresses.

**complete\_expansion\_room Method:**

**public Collection<CompleteFreeSpaceExpansionRoom> complete\_expansion\_room(IncompleteFreeSpaceExpansionRoom p\_room)**

* Completes an incomplete expansion room and returns the final set of expansion rooms that were successfully routed. This method ensures that the completed rooms don't overlap with obstacles or traces.

**remove\_complete\_expansion\_room Method:**

**public void remove\_complete\_expansion\_room(CompleteFreeSpaceExpansionRoom p\_room)**

* This method removes a complete expansion room from the list and invalidates the drill pages associated with that room (so they are recalculated in future runs).

## Managing Autorouter Database:

**clear Method:**

**public void clear()**

* Clears all the temporary autorouting data stored during the routing process. This method is useful when switching between different nets or when restarting the routing process.

**reset\_all\_doors Method:**

**private void reset\_all\_doors()**

* Resets all the doors used in autorouting. Doors represent entry and exit points of expansion rooms and resetting them is important when switching between connections.

### Validation and Drawing:

**validate Method**

**public boolean validate()**

* Validates the internal data structures used by the autorouter to ensure that everything is consistent and ready for routing.

**draw Method**

**public void draw(Graphics p\_graphics, GraphicsContext p\_graphics\_context, double p\_intensity)**

* This method draws the current expansion rooms on the graphics context. It is typically used to visualize the routing process, showing the areas being routed and their progress.

### Expansion Room and Door Calculations:

**calculate\_doors Method:**

**private CompleteExpansionRoom calculate\_doors(ExpansionRoom p\_room)**

* Calculates the **doors** of an expansion room. A **door** is a boundary through which the autorouter can expand the routing area. Calculating doors helps the autorouter decide where to expand the routing process next.

**complete\_neighbour\_rooms Method:**

**public void complete\_neighbour\_rooms(CompleteExpansionRoom p\_room)**

* This method ensures that the **neighbor rooms** of the specified expansion room are completed before routing continues. Completing the neighbor rooms ensures that the routing paths don’t overlap or interfere with one another.

### Autoroute Result Enum:

**public enum AutorouteResult**

**{**

**ALREADY\_CONNECTED, ROUTED, NOT\_ROUTED, INSERT\_ERROR**

**}**

* **AutorouteResult**: Represents the possible outcomes of the autorouting process. It can be:
  + **ALREADY\_CONNECTED**: The connection is already made.
  + **ROUTED**: The routing was successful.
  + **NOT\_ROUTED**: The router failed to find a valid path.
  + **INSERT\_ERROR**: There was an error inserting the routed path into the board.

**Summary:**

* The AutorouteEngine class is responsible for managing and executing the autorouting algorithm on a PCB layout.
* It interacts with the board, expansion rooms, and search algorithms to find optimal routes for electrical connections, handle ripped connections, manage obstacles, and draw routing results.
* It provides the foundation for autorouting processes in a PCB design tool like **FreeRouting**.

## BatchAutorouter.java:

* The **BatchAutorouter.java** class is responsible for managing the **batch autorouting** process in the FreeRouting application.
* This class handles the sequencing of autorouting passes, where it continuously reroutes parts of the board until all connections are routed or the process is interrupted (by the user or some condition).

### Fields (Variables) Defined in the Class:

**Routing and Thread Control:**

* **thread:** An instance of InteractiveActionThread that represents the thread in which autorouting actions are performed. This handles user interactions and autorouting logic concurrently.
* **hdlg:** A BoardHandling object used to interact with the PCB board. It helps retrieve settings and manage the board state.
* **routing\_board:** A reference to the RoutingBoard, which holds all the information about the current PCB, including layers, traces, and components.
* **is\_interrupted:** A flag indicating whether the autorouting process has been interrupted, either by the user or due to some internal condition (e.g., loop detection).

**Autorouting Settings:**

* **trace\_cost\_arr**: An array that holds the cost factors (horizontal and vertical) for routing traces on each layer of the PCB. This defines how expensive it is for the autorouter to place a trace in a given direction.
* **remove\_unconnected\_vias**: A flag indicating whether unconnected vias (vias not linked to traces) should be removed at the end of the routing process.
* **start\_ripup\_costs**: Defines the initial cost used in **ripup** operations. **Ripup** means removing previously routed traces to make room for new routes.

**Tracking of Routing Progress:**

* **already\_checked\_board\_hashes**: Keeps track of board states that have already been routed (using board hash values). This prevents the autorouter from entering an infinite loop by re-checking the same board configuration.
* **traceLengthDifferenceBetweenPasses**: Stores the differences in trace lengths between autorouting passes. This is used to monitor whether routing is making progress. If the trace length doesn’t change significantly over a few passes, the autorouter may stop.

### Constructors**:**

**Primary Constructor**

**public BatchAutorouter(InteractiveActionThread p\_thread, boolean p\_remove\_unconnected\_vias, boolean p\_with\_preferred\_directions, int p\_start\_ripup\_costs)**

* This constructor initializes the autorouter with the thread handling user interactions, board settings, and routing configuration.
* **p\_remove\_unconnected\_vias**: Specifies whether the autorouter should remove unconnected vias at the end of routing.
* **p\_with\_preferred\_directions**: If true, the autorouter will prefer routing traces in specific directions on different layers (e.g., horizontal on one layer and vertical on another).
* **p\_start\_ripup\_costs**: The initial ripup cost for the first autorouting pass.

**Alternative Constructor**

**public BatchAutorouter(InteractiveActionThread p\_thread, boolean p\_remove\_unconnected\_vias, boolean p\_with\_preferred\_directions, int p\_start\_ripup\_costs, RoutingBoard updated\_routing\_board)**

This version allows for an updated routing board to be passed in, if available. If not, it uses the default routing board from the BoardHandling (hdlg) object.

### Core Autorouting Functionality:

**autoroute\_passes\_for\_optimizing\_item**

* This static method performs multiple autorouting passes to optimize the PCB until the board is fully routed, the maximum number of passes is reached, or the user interrupts the process.
* p\_max\_pass\_count: Limits the number of autorouting passes.
* p\_ripup\_costs: Sets the cost for ripping up traces during each pass.
* p\_with\_preferred\_directions: Whether to prefer routing in specific directions on different layers.

**autoroute\_passes**

* autoroute\_passes is responsible for performing autorouting in batch mode. It performs multiple passes until the board is fully routed or the process is interrupted.
* Progress Tracking: It checks how much progress is being made in each pass by comparing the lengths of traces between passes.
* Saving Intermediate Stages: If save\_intermediate\_stages is true, the autorouter saves intermediate stages of the board to a file.
* Detecting Stagnation: If the trace length does not significantly change after a few passes, the autorouter assumes it cannot make further improvements and stops.

**autoroute\_pass**

* Performs a **single autorouting pass**, attempting to route all unrouted items on the board.
* **p\_pass\_no**: Indicates the current pass number (useful for tracking progress and for display purposes).
* **p\_with\_screen\_message**: If true, displays status messages in the UI to indicate progress.
* It iterates over each **item** on the board, routing those that are not yet fully connected. For each item:

1. It checks the **net** associated with the item and determines whether it needs routing.
2. If routing is needed, it calculates the shortest distance between the item’s unconnected points and attempts to route the trace.
3. If the user requests to stop the autorouter, it interrupts the process.

**autoroute\_item**

* This method attempts to autoroute a specific item on a given net.
* **p\_item**: The item to be routed (could be a trace, pin, or via).
* **p\_route\_net\_no:** The net number associated with this item.
* **p\_ripped\_item\_list:** A list of items that have been ripped up (removed) to make space for new traces.
* **p\_ripup\_pass\_no**: The current ripup pass number. Each pass increases the ripup cost to encourage better optimization in later passes.

### Utility Methods:

**remove\_tails:**

* Removes unnecessary trace tails (segments of traces that don't connect to anything) after routing. If fanout vias are used, it keeps them; otherwise, it removes them.

**calc\_airline:**

* Calculates the shortest distance between two sets of items (usually pins and vias) and draws a virtual connection line (called an airline). This line is used to help guide the autorouter.

### Handling User Interaction and Autorouting Control:

**is\_stop\_auto\_router\_requested:**

* Throughout the code, there are checks to see if the user has **requested to stop the autorouter**. This ensures that the process can be safely interrupted by the user when necessary.

**get\_air\_line:**

* Returns the current **airline** (a line between the start and destination points of the currently routed connection). This is useful for visualization purposes in the user interface.

**Summary:**

* The **BatchAutorouter** class is responsible for managing multiple passes of the autorouting algorithm.
* It handles routing all unrouted items on the PCB by iterating over the items, calculating distances, and routing connections.
* It supports user interruption, checks for routing stagnation, and manages trace tails after routing.
* This class interacts heavily with the board handling (**hdlg**) and routing board (**RoutingBoard**) classes to modify the board’s layout during routing.

## BatchFanout.java:

The BatchFanout class is part of the autorouting process in a PCB design tool. It handles the **fanout operation**, which involves creating connections from **surface-mount device (SMD) pins** to the rest of the board using vias. This step is essential for autorouting, especially when dealing with dense SMD components. The class is designed to manage the sequencing of fanout operations on the board in a **batch processing** fashion, ensuring that each component's SMD pins are connected efficiently.

**Key Concepts:**

1. **Fanout**: This operation places vias to connect SMD pins to other layers, providing access for routing on multi-layer boards.
2. **Batch Processing**: The fanout process is performed in multiple passes. Each pass attempts to route the SMD pins of components, and the process continues until no more pins can be routed or a maximum number of passes is reached.

**Class Structure and Key Methods**

**1. Attributes**

* **thread**: An instance of **InteractiveActionThread**, which handles user interaction and the overall control flow. This is the main thread that manages the interaction and autorouting process.
* **routing\_board**: An instance of **RoutingBoard**, representing the PCB where the fanout operation is taking place. This includes all the components, nets, and pins on the board.
* **sorted\_components**: A **SortedSet** of Component objects. This holds all the components on the board that contain SMD pins. The components are sorted based on the number of SMD pins, with components that have more pins prioritized for fanout.

**2. BatchFanout() Constructor**

The constructor initializes the class by:

* Storing the **InteractiveActionThread** and retrieving the **RoutingBoard** from it.
* Retrieving all the **SMD pins** from the board via **routing\_board.get\_smd\_pins().**
* Initializing the sorted set of **components** (instances of the inner Component class), which are sorted based on the number of SMD pins they contain. Only components with SMD pins are included in the **sorted\_components** set.

**3. fanout\_board() Static Method**

This is the main entry point for the fanout process:

* It creates an instance of BatchFanout.
* Then it iteratively performs **fanout passes** using the **fanout\_pass()** method.
* The process stops either when no new SMD pins can be routed in a pass or when a maximum of 20 passes (MAX\_PASS\_COUNT) is reached.

**4. fanout\_pass() Method**

This method performs a single fanout pass:

* For each component in **sorted\_components**, it attempts to fanout its SMD pins.
* The method tracks the number of components left to fanout (**components\_to\_go**) and the number of SMD pins that were successfully routed (**routed\_count**).
* It also tracks **failures** (pins that couldn't be routed) and **errors** (issues inserting routes).

**Fanout Process for Each Pin**:

* The board starts marking the area that will be changed via **routing\_board.start\_marking\_changed\_area()**.
* The actual fanout operation is performed using **routing\_board.fanout(),** which returns one of three results:
  + **ROUTED**: The pin was successfully routed.
  + **NOT\_ROUTED**: The pin could not be routed.
  + **INSERT\_ERROR**: An error occurred when trying to insert the route.

**Time Limit**:

* The method enforces a **time limit** (TimeLimit) for the fanout operation, which increases with each pass (i.e., more time is allocated for later passes).

**Stopping Condition**:

* The method stops early if the user requests it (**thread.is\_stop\_requested()).**
* Once all pins have been processed, the number of routed pins is returned.

**5. Component Inner Class**

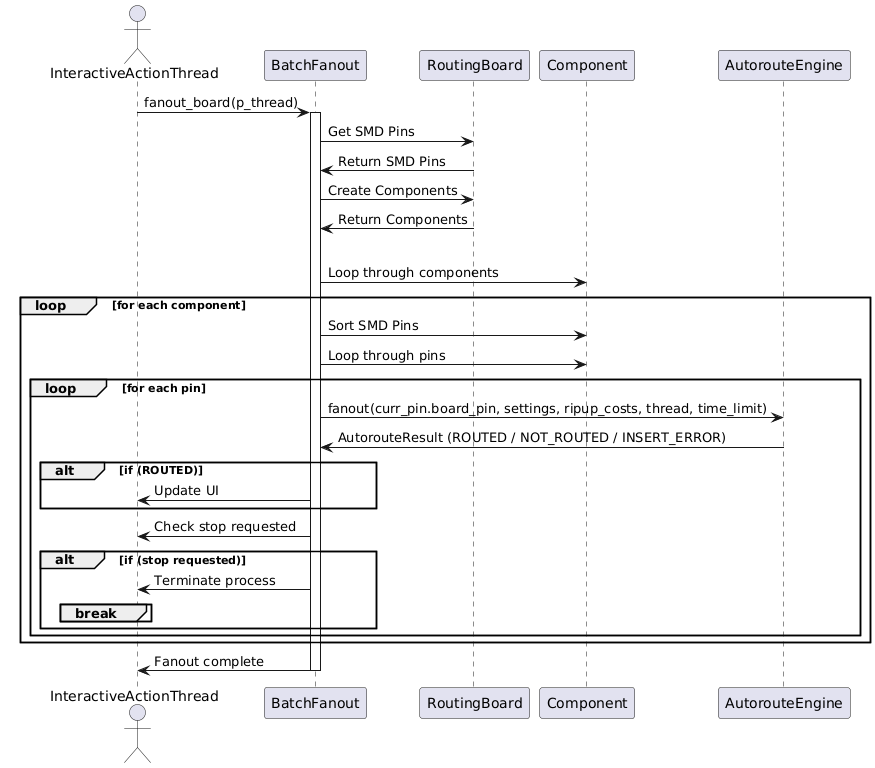
The Component class represents a PCB component that contains SMD pins.

* **Attributes**:
  + **board\_component**: A reference to the original component on the board.
  + **smd\_pin\_count**: The number of SMD pins in this component.
  + **smd\_pins**: A sorted set of **Pin** objects, which represent the SMD pins of the component.
  + **gravity\_center\_of\_smd\_pins**: This is the center of gravity of all SMD pins. It is used to prioritize pin routing based on proximity to the center of the component.
* **Constructor**:
  + This constructor calculates the **center of gravity** for the component’s SMD pins.
  + It also initializes the sorted set of SMD pins (**smd\_pins**), which are sorted by their distance from the center of gravity.
* **compareTo() Method**:
  + The components are sorted by their number of SMD pins (components with more SMD pins are processed first). If two components have the same number of SMD pins, they are sorted by component number.

**6. Pin Inner Class**

The Pin class represents a single SMD pin of a component.

* **Attributes**:
  + **board\_pin**: The reference to the pin on the board.
  + **distance\_to\_component\_center**: The distance from this pin to the **center of gravity** of the component’s SMD pins. This is used to prioritize the fanout of pins closest to the center.
* **compareTo() Method**:
  + Pins are sorted by their distance to the component’s center of gravity (pins closer to the center are processed first). If two pins are at the same distance, they are sorted by their pin number.



**BatchFanOut.java Execution Sequence Diagram**

## BatchOptRoute.java:

The **BatchOptRoute** class is designed to **optimize the routing of a PCB** after the autorouting process has completed. This class works on a single-threaded optimization process and tries to improve the routing quality by reducing the number of **vias**, minimizing the **trace length**, and resolving potential issues like **unrouted nets**. The class makes multiple optimization passes over the routed board and applies incremental improvements during each pass.

**Key Concepts in the BatchOptRoute Class:**

1. **Routing Optimization**:
   * After the autorouter has done its job, the routes may not be fully optimized. The goal of BatchOptRoute is to take a routed board and improve its routing by reducing the number of vias and minimizing the trace length while maintaining or improving the connectivity of the board.
2. **Multiple Optimization Passes**:
   * The optimization is done in **multiple passes**, each one trying to reroute connections to achieve a more optimal layout. Each pass either reroutes traces, removes unnecessary vias, or shortens trace lengths.
3. **Rip-Up and Re-Route Strategy**:
   * The class uses a **rip-up and re-route** strategy, where existing traces are removed and rerouted in a more optimal manner. This is done while keeping track of rip-up costs, and the strategy may vary based on the pass number.
4. **Snapshot and Undo**:
   * The class takes snapshots of the board's state and uses undo/redo mechanisms to revert changes if a re-route results in a worse configuration than the original.

**Attributes of the BatchOptRoute Class**

1. **MAX\_AUTOROUTE\_PASSES**:
   * A constant that defines the maximum number of optimization passes that can be made over the board. This ensures that the optimization doesn't run indefinitely.
2. **ADDITIONAL\_RIPUP\_COST\_FACTOR\_AT\_START**:
   * A multiplier used to increase rip-up costs during the initial optimization passes. This makes the algorithm more aggressive in early passes.
   * **thread**:
     1. An instance of InteractiveActionThread that manages user interactions and the current optimization process. It allows the optimizer to communicate with the user interface.
   * **clone\_board**:
     1. A boolean flag that indicates whether a cloned copy of the board is used during optimization. Cloning is useful when optimizing a board without affecting the original until the optimization is completed.
   * **routing\_board**:
     1. The RoutingBoard instance representing the current PCB being optimized.
   * **sorted\_route\_items**:
     1. A list of route items (e.g., traces, vias) that are sorted and processed for optimization.

**Key Methods in the BatchOptRoute Class**

**1. Constructor: BatchOptRoute(InteractiveActionThread p\_thread, boolean p\_clone\_board)**

* Initializes the optimizer with an optional flag to clone the board. If **p\_clone\_board** is true, a deep copy of the **RoutingBoard** is made, allowing optimization without altering the original board until the optimization is finished.

**2. optimize\_board(boolean save\_intermediate\_stages, float optimization\_improvement\_threshold, InteractiveActionThread isStopRequested)**

* This method orchestrates the entire optimization process.
* It performs multiple optimization passes, aiming to improve the routing. The process continues until the improvement is below the threshold (**optimization\_improvement\_threshold**) or the user requests the process to stop (**isStopRequested**).
* The **save\_intermediate\_stages** flag determines if intermediate results should be saved after each pass, which can be useful in case the optimization is interrupted.

**3. opt\_route\_pass(int p\_pass\_no, boolean p\_with\_preferred\_directions)**

* This method performs a **single optimization pass**. It tries to reduce the number of vias and shorten the trace lengths for the entire board.
* It returns a float value representing how much the routing has improved during this pass (in percentage). A value of -1 indicates that the process should continue, regardless of the improvement.
* The **p\_with\_preferred\_directions** flag toggles whether the preferred routing directions should be enforced during this pass. This helps create variations in routing strategies.

**4. opt\_route\_item(Item p\_item, int p\_pass\_no, boolean p\_with\_preferred\_directions)**

* Attempts to optimize a specific item (e.g., a trace or via) by rerouting it.
* The method rips up the current item, reroutes it, and compares the results (via count, trace length, etc.). If the new routing is better, the changes are kept; otherwise, the board is restored to its previous state using the undo mechanism.
* The method checks if the rerouting improves the connectivity, reduces vias, and shortens trace lengths.

**5. calc\_weighted\_trace\_length(RoutingBoard p\_board)**

* Computes a weighted sum of the trace lengths on the board. This method factors in the trace width and clearance when calculating the total length.
* The function helps evaluate the quality of the routing by measuring the trace length and allowing the optimizer to compare improvements across passes.

**6. ReadSortedRouteItems Class**

* A nested class responsible for reading the vias and traces on the board in ascending order (sorted by coordinates). This ensures that the optimizer processes items in a consistent order.
* It prioritizes vias over traces when both are present at the same location. This helps the optimizer focus on via reduction first, as vias tend to increase the complexity of the routing.
* It maintains the current position (min\_item\_coor, min\_item\_layer) and advances to the next route item in a systematic manner.

**Optimization Process Workflow:**

1. **Initialization**:
   * When BatchOptRoute is initialized, it can optionally create a cloned version of the RoutingBoard to perform optimizations on a copy of the board.
2. **Multiple Passes**:
   * The optimize\_board method is the main entry point. It runs multiple optimization passes (up to MAX\_AUTOROUTE\_PASSES), and for each pass, it:
     1. **Calculates the Current Trace Lengths** and number of vias on the board.
     2. **Performs a rip-up and re-route** of traces and vias to reduce overall complexity and trace lengths.
     3. **Compares the results**: If the optimization pass results in fewer vias or shorter trace lengths, the changes are kept; otherwise, the board is reverted to the previous state.
3. **Stopping Conditions**:
   * The process stops if the improvement in the routing falls below the optimization\_improvement\_threshold or if the user explicitly requests it.
4. **Saving Intermediate Stages**:
   * If save\_intermediate\_stages is set to true, the optimizer saves the current state of the board after each pass. This is helpful for long-running optimizations where unexpected interruptions might occur.

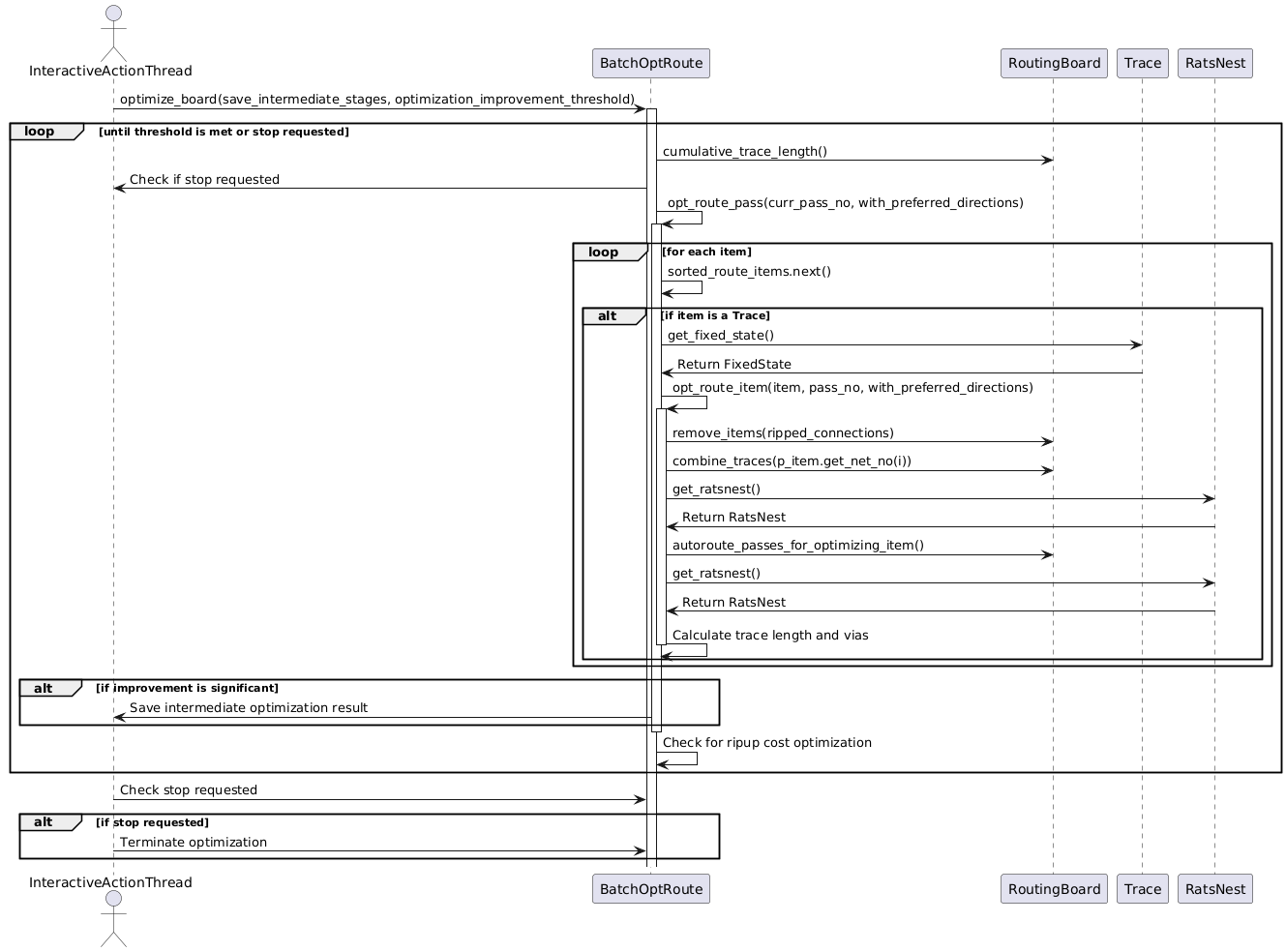
**Key Optimizations in the Code:**

1. **Weighted Trace Lengths**:
   * The calc\_weighted\_trace\_length() method calculates the total trace length, taking into account both the trace width and clearance. This ensures that wider traces, which have a bigger impact on the routing, are given more weight during the optimization process.
2. **Rip-Up and Re-Route**:
   * The opt\_route\_item() method implements a rip-up and re-route strategy, where an item (such as a trace) is temporarily removed and rerouted to see if a better route can be achieved. If no improvement is made, the item is restored to its original state.
3. **Incremental Improvements**:
   * The optimizer runs in multiple passes, gradually improving the routing. It tracks the number of vias and trace lengths after each pass, and if no improvement is made, it adjusts the rip-up costs and continues the process.

**Conclusion:**

The BatchOptRoute class is a **post-autorouting optimizer** that seeks to improve the routing on a PCB by making incremental improvements across multiple passes. It minimizes vias, shortens traces, and ensures better connectivity. The class uses a rip-up and re-route strategy, and it can be configured to run on a cloned board or the main routing board. The multiple optimization passes allow for gradual improvements, and the use of snapshots and undo mechanisms ensures that changes are only kept if they result in better routing.

This class is essential in the PCB design process as it ensures that the initial results of the autorouter are further refined for **optimal performance** and **manufacturability**.



**BatchOptRoute.java Sequence Diagram**

## BatchOptRouteMT.java:

The **BatchOptRouteMT** class is a **multi-threaded version** of the BatchOptRoute class, designed to optimize the routing on a PCB after the autorouting has completed. This class leverages multiple threads to **improve the performance and efficiency** of the routing optimization process. It achieves this by splitting the work into multiple tasks and distributing them across a thread pool, allowing for parallel optimization.

The class handles two main tasks:

1. **Item selection and optimization**: Items (such as traces and vias) are selected for rerouting, and then rerouted to achieve shorter paths or fewer vias.
2. **Board update strategy**: This governs how the routing board is updated after rerouting tasks complete, either globally (after all tasks) or incrementally (after each task).

**Key Concepts**

1. **Multi-threading**:
   * The class uses **Java's ThreadPoolExecutor** to run multiple optimization tasks in parallel. Each task optimizes a different part of the routing board.
2. **Routing Board Update Strategies**:
   * **GLOBAL\_OPTIMAL**: All tasks complete their optimization, and then the board is updated globally at the end.
   * **GREEDY**: The board is updated after each task if that task produces a better result. This allows new tasks to work on the updated board immediately.
   * **HYBRID**: A combination of both strategies, alternating between global and greedy updates.
3. **Item Selection Strategies**:
   * **SEQUENTIAL**: Items are processed in a fixed, sequential order.
   * **PRIORITIZED**: Items are prioritized based on their previous improvement results.
   * **RANDOM**: Items are processed in a random order.
4. **Optimization Tasks**:
   * Each optimization task reroutes an individual item (such as a trace or via) and compares the new routing result to the original. If the new routing is better, the task may update the board.

**Attributes of BatchOptRouteMT**

1. **board\_update\_strategy**:
   * The strategy to use for updating the board. Can be GLOBAL\_OPTIMAL, GREEDY, or HYBRID. It defines how and when the board is updated with the best routing results.
2. **item\_selection\_strategy**:
   * Defines how items are selected for optimization. Can be SEQUENTIAL, PRIORITIZED, or RANDOM.
3. **thread\_pool\_size**:
   * The number of threads in the thread pool, which defines how many tasks can run in parallel.
4. **item\_ids**:
   * A list of item IDs representing the items that need to be optimized.
5. **result\_map**:
   * A map storing the optimization results for each item, keyed by item ID. This allows for prioritizing or reprocessing items based on their previous results.
6. **hybrid\_list**:
   * A list used when the HYBRID update strategy is active. It alternates between GLOBAL\_OPTIMAL and GREEDY strategies based on the provided ratio.
7. **pool**:
   * The thread pool executor responsible for managing and executing the optimization tasks.
8. **best\_route\_result**:
   * Stores the best routing result found during the optimization process.
9. **winning\_candidate**:
   * The task that produces the best routing result during a pass.
10. **task\_completion\_signal**:

* A CountDownLatch used to signal the completion of all tasks in a pass.

**Key Methods in BatchOptRouteMT**

**1. opt\_route\_pass(int p\_pass\_no, boolean p\_with\_preferred\_directions)**

* **Purpose**: This is the main method responsible for executing a single pass of the optimization process. It splits the work into tasks and executes them in parallel across multiple threads.
* **Workflow**:
  1. **Prepare items**: The method calls prepare\_next\_round\_of\_route\_items() to gather and sort the items that need to be optimized in this pass.
  2. **Task Execution**: It creates a thread pool with the specified number of threads (thread\_pool\_size) and schedules the tasks to optimize each item.
  3. **Wait for Completion**: The method waits for all tasks to complete using pool.awaitTermination().
  4. **Update Board**: Depending on the update strategy, the board is either updated incrementally (for each task) or globally (after all tasks complete).
  5. **Log Results**: Logs the results of the pass, including the number of vias, trace length, and time taken.
* **Logging**: Extensive logging is used to track the progress of the optimization, including task counts, updates, and results.

**2. prepare\_next\_round\_of\_route\_items()**

* **Purpose**: Prepares the list of items that will be optimized in the current pass.
* **Workflow**:
  1. **Sort Items**: Depending on the ItemSelectionStrategy, items are sorted sequentially, prioritized based on previous results, or shuffled randomly.
  2. **Clear Previous Results**: Clears the result map and the list of items (item\_ids) to prepare for the new pass.

**3. update\_master\_routing\_board()**

* **Purpose**: Updates the routing board with the results of the winning candidate task.
* **When Used**: This method is called when the GREEDY update strategy is used, and a task produces a better result. It applies the best routing result found so far to the main routing board.
* **Workflow**:
  1. **Update Board**: Copies the winning candidate's board state to the master routing board.
  2. **Recalculate Metrics**: Recalculates metrics like the cumulative trace length and the number of vias.
  3. **Log Update**: Logs the update with details about the new board state.

**4. is\_winning\_candidate(OptimizeRouteTask task)**

* **Purpose**: Determines if the result from a given optimization task is better than the current best result.
* **Workflow**:
  1. **Compare Results**: Compares the task's result with the current best result (best\_route\_result). If the task's result is better, it becomes the new best result.
  2. **Update Board (if GREEDY)**: If the GREEDY update strategy is being used and the task's result is better, the board is immediately updated.

**How Multi-threading Works in BatchOptRouteMT**

1. **Thread Pool Creation**:
   * The thread pool is created using ThreadPoolExecutor with a fixed number of threads (thread\_pool\_size). This defines the level of parallelism in the optimization process.
2. **Task Scheduling**:
   * For each item to be optimized, a new task (OptimizeRouteTask) is created and submitted to the thread pool for execution. Each task reroutes a specific item and compares the new routing result with the current best result.
3. **Task Completion**:
   * The method waits for all tasks to finish using pool.awaitTermination(). If the optimization is interrupted, the pool is shut down immediately.
   * As tasks finish, they report their results using is\_winning\_candidate(), and if a task produces a better result, the board may be updated (depending on the update strategy).

**Optimization Workflow:**

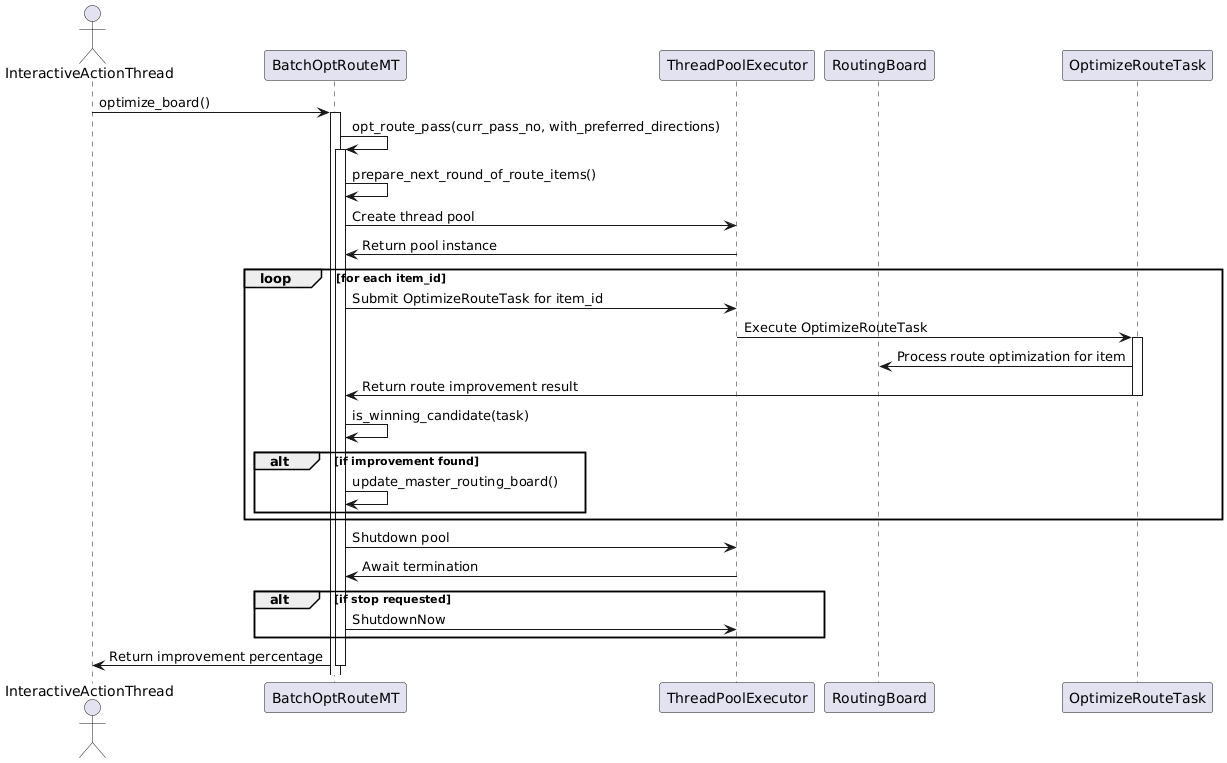
1. **Item Preparation**:
   * Items are selected and sorted based on the current strategy (sequential, prioritized, or random).
2. **Parallel Task Execution**:
   * Each item is optimized in parallel by separate threads. The tasks reroute the items and report their results back to the main process.
3. **Winning Candidate Selection**:
   * The best result (winning candidate) is tracked, and if it improves the routing, it may update the board.
4. **Board Updates**:
   * In GREEDY mode, the board is updated after each successful task. In GLOBAL\_OPTIMAL mode, the board is updated only after all tasks complete.
5. **Next Pass**:
   * After each pass, the process repeats until all items are optimized or no further improvements are possible.

**Conclusion**

The **BatchOptRouteMT** class introduces **multi-threading** to the routing optimization process, significantly improving the performance and efficiency compared to a single-threaded approach. It distributes the workload across multiple threads, with each thread optimizing a separate part of the routing.

Depending on the **board update strategy**, the board can be updated globally after all tasks complete, or incrementally after each task, allowing for more flexible and efficient optimization.

This approach is especially useful for **large, complex PCBs** where the optimization process would otherwise take a long time in a single-threaded environment. By parallelizing the optimization tasks, BatchOptRouteMT ensures that routing improvements can be achieved much faster, making it ideal for complex designs.



BatchOptRouteMT.java file sequence diagram