Tomasz Włostowski

Interactive PCB Routing

Ideas & algorithms we implemented in Kicad

FOSDEM 2015

Brussels, February 1st 2015

Outline

- Introduction
- Geometry & storage model
- Shove/hug algorithm
- Optimization algorithms
- Status & future improvements

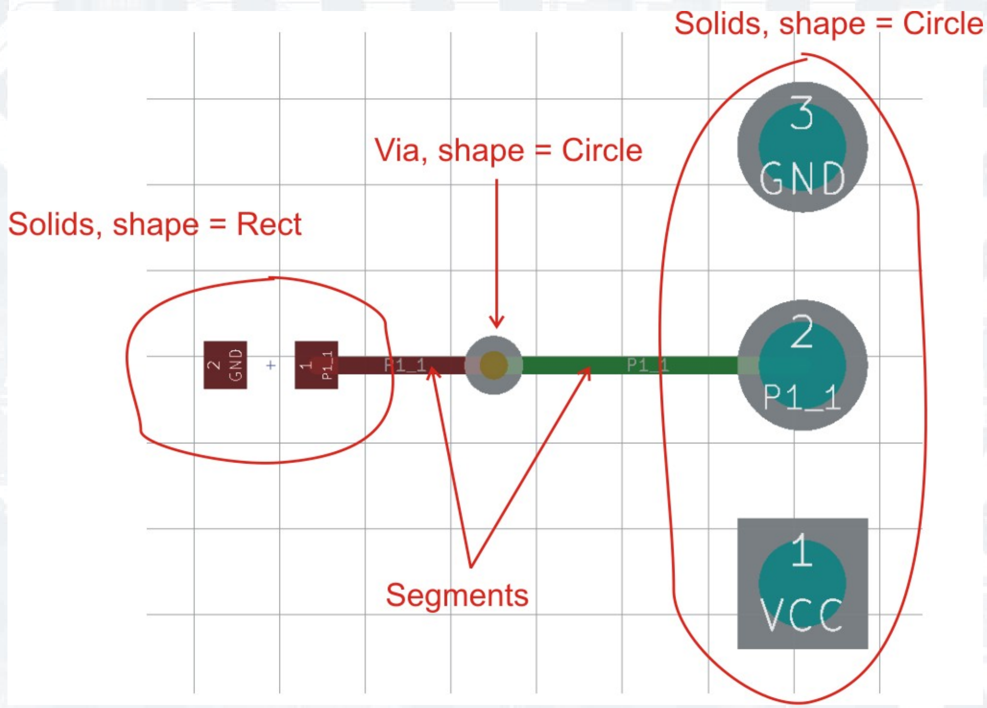
Interactive routing – what is it?

- Standard in high-end and mid-range proprietary software, now available in F/OSS tools.
- Routed trace follows mouse cursor.
- Hug fixed obstacles (such as component pads).
- Push/shove away colliding traces and vias, making space for the newly routed trace.
- Retreating the cursor causes modified traces/vias to spring back to their former shapes.

Why route interactively?

- Full control of the layout.
- Assist the user instead of trying to outsmart him.
- No post-routing cleanup and complex configuration required.
- Rapid layout modification.

Geometry



- Operates on **solids**, **segments** and **vias**.
- Shape based: each PCB feature associated with a generic geometric shape (circle, rectangle, convex polygon, etc.).

Geometry

- **Solids** are fixed PCB items (pads, keepout regions, board outline).
- **Vias** are …. vias ;-)
- **Segments** are straight line segments with non-zero thickness.
- A chain of **Segments** on the same layer and of same width (with/without a via on end) is a **Line.**
- Adding new PCB features → just add a new shape.
- No floating point. Guaranteed 1 LSB error in all calculations.

Storage Model - requirements

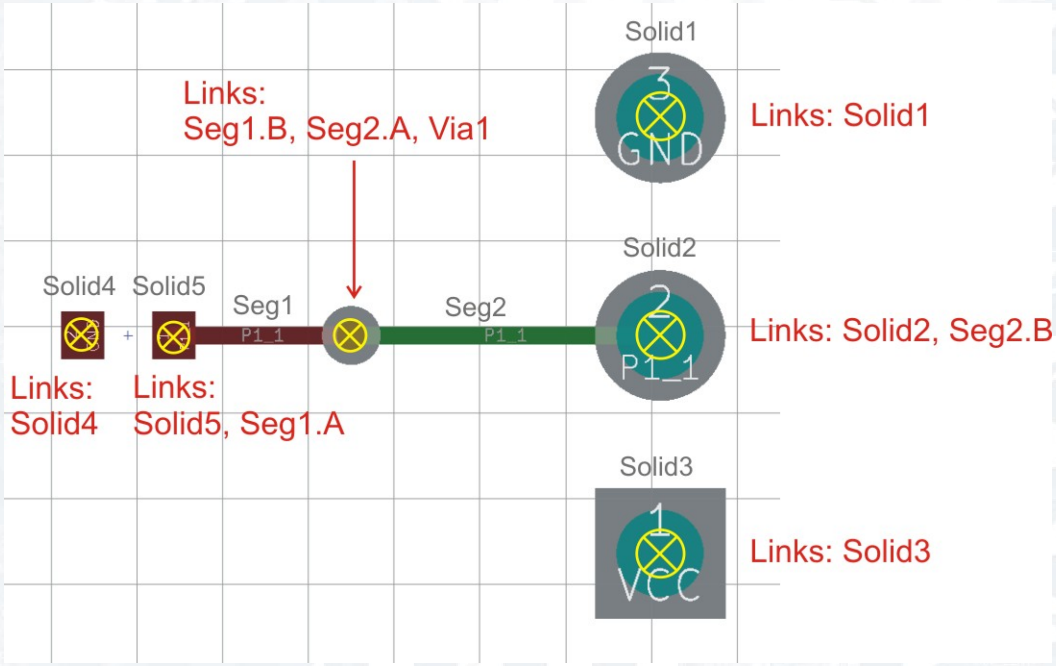
- **Fast collision search.**
- **Resolving connectivity** between items (connection graph).
- **Flat storage**: no hierarchy and fixed links between items.
- Items owned by the database are **non-mutable**.
- Lightweight **copy-on-write** cloning (for trace springback and trying out different optimization strategies).

Storage Model – collision search

- Shapes are stored in **R-Trees**.
- One R-tree per board layer.
- Separate R-trees for pads & vias (reduce overlap).
- Clearance is provided by an external class (can apply complex DRC rules without making the router more complex)

Storage Model – connectivity

- Flat model: stored items do not know about each other.
- Connectivity between items is resolved through **joints.**
- **A joint** binds together items at the same location, overlapping set of layers and same net.



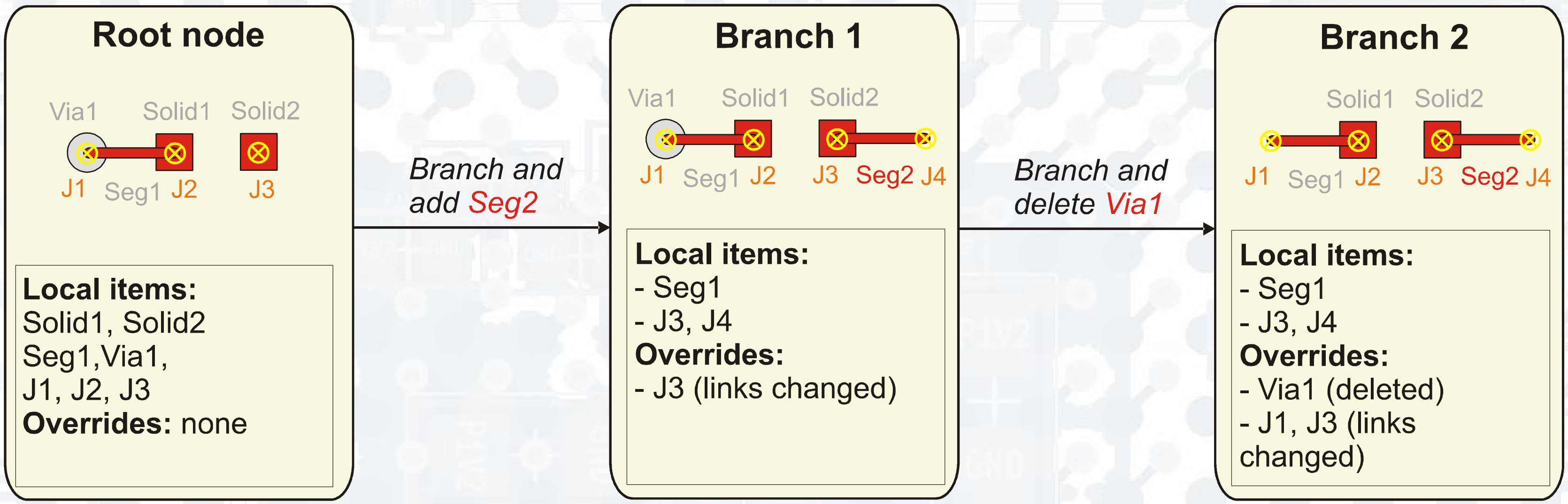
Storage Model – connectivity

- Joints are stored in a hash table.
- Hash key: `(x, y, net ID).`
- Each joint keeps a list of linked items → simple connection graph traversal (segments = edges, joints = vertices).
- Joints are updated when items are added/removed.
- Split/merge joints when layer sets change (e.g. two joints on separate layers become one when a via is placed)

Storage Model – cloning

- Need to keep several branches of the database for springback, loop removal & parallel optimization strategies
- Tree-like structure
- Root node holds the entire PCB geometry
- Leafs keep items and joints modified wrs to the root node
- Overridden items are stored in an unordered set

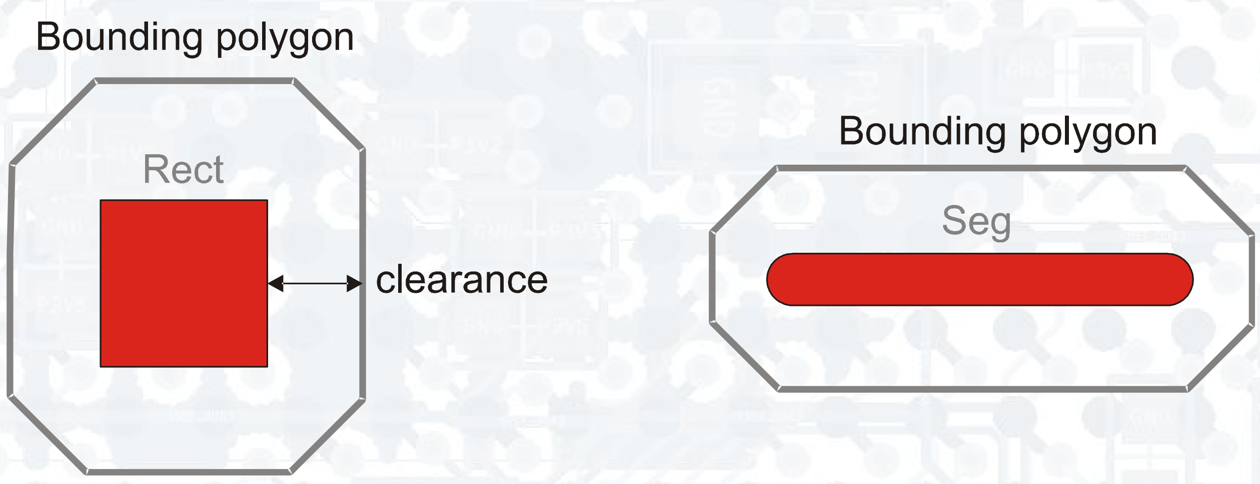
Storage Model – cloning



- Branch → copy override set and clone modified items only.
- Modifications are usually local, so little stuff to copy.
- Easy to track modifications and update the host tool PCB model.

Shove & Hug – bounding polygons

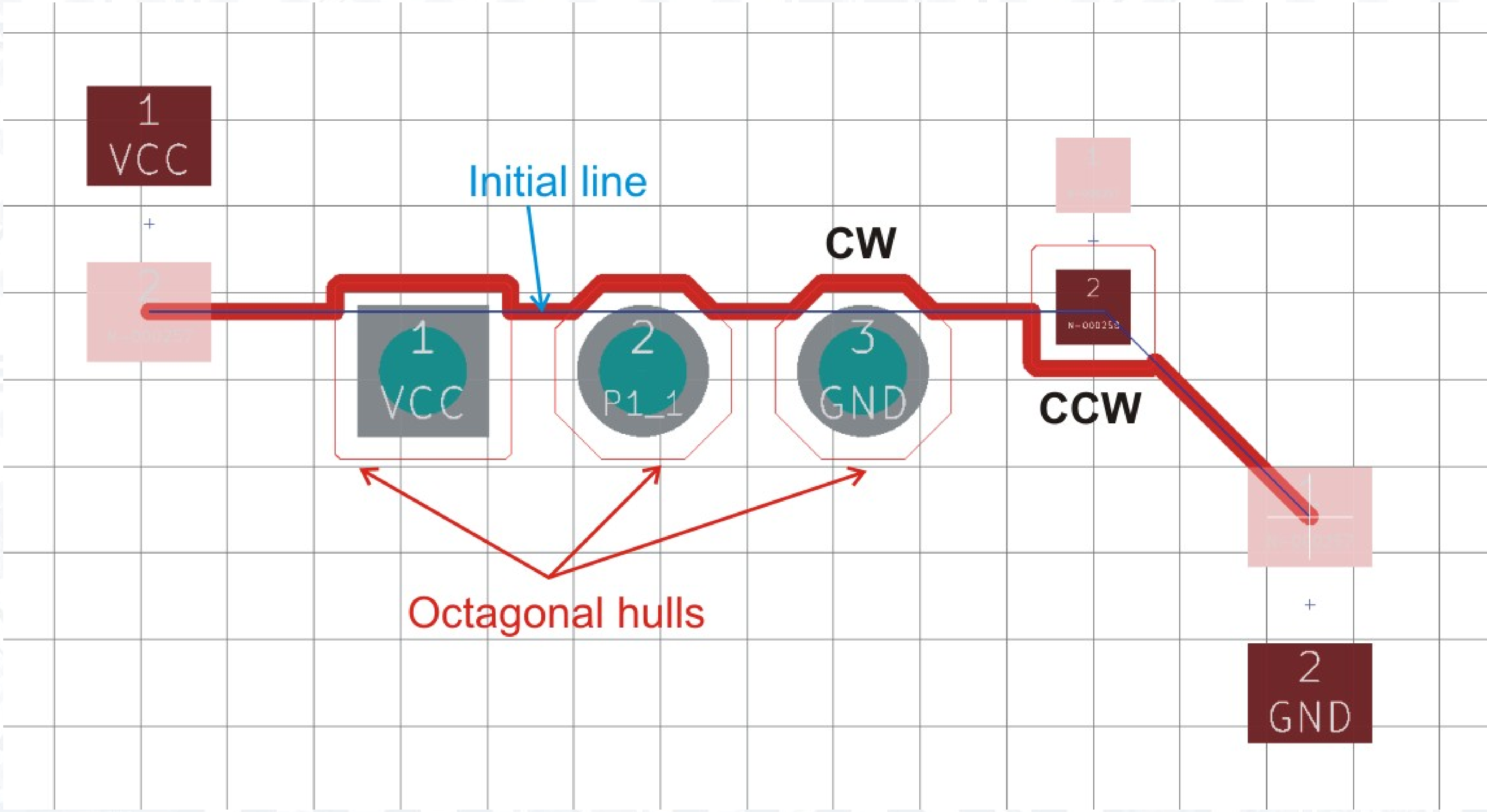
- Build octagonal hulls around colliding items
- Distance between the bounding polygon and its owner defined by the clearance rule
- Use the hulls to compute shoved & hugged traces



Hug & walk around

- Build initial 45-degree **Line.**
- Find colliding items.
- Compute their hulls.
- Merge hulls with the original line to get the walkaround path.
- Choose shortest CW/CCW path (locally and globally)

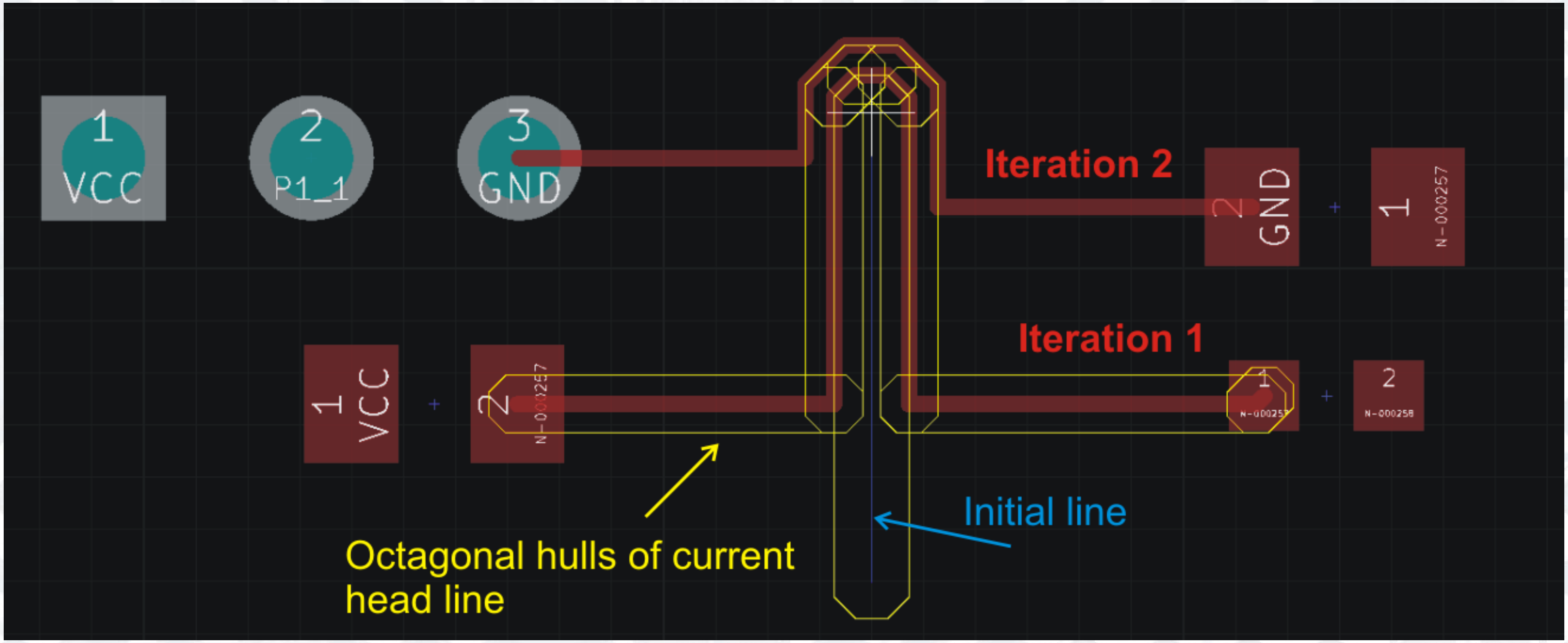
Hug & walk around



Shove algorithm

- Take the initial 45-degree **head line**, push on the stack.
- Find colliding lines, sort according to the distance from the head line, process one by one, push on the stack.
- Use octagonal hulls to shove the colliding lines away.
- Use heuristics to determine which side to push (open curve has no orientation...)
- Keep iterating until stack is empty.

Shove algorithm – line shoving

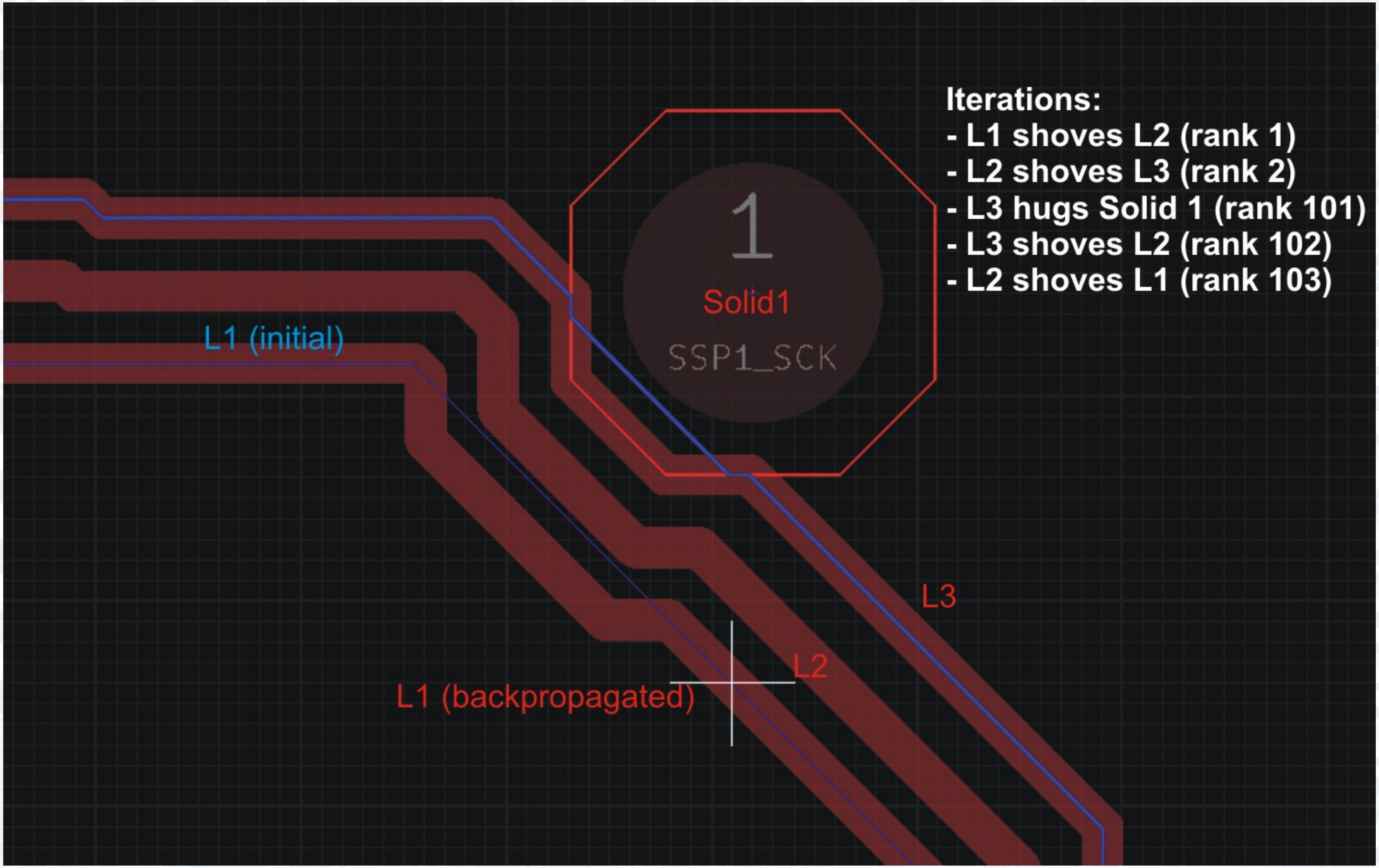


T. Włostowski
Interactive PCB routing

Shove algorithm – solid collisions

- Decide whether to backpropagate the collision or jump around the obstacle (heuristics again)
- Add ranks to the lines: higher rank pushes lower rank
- Fresh lines → rank = 0
- Shoved line → rank + 1
- Backpropagated line → rank + 100 (triggering shove in opposite direction)

Shove algorithm – backpropagation



Shove algorithm – vias

- Via shoves line → just compute octagonal hull
- Via shoves via or line shoves via → compute minimum resolving force (a trivial physics simulation)
- Move the colliding via away
- Drag traces connected to the via being pushed and put on the stack

Shove algorithm – springback

- Small step approach: while moving the cursor, the result of previous step is the input to the current step
- **Branch stack** → each successful result is pushed on top of the stack in a new DB branch
- If the head line is not colliding anymore with the top of the branch stack, pop the stack and repeat until collision found.

Putting it all together

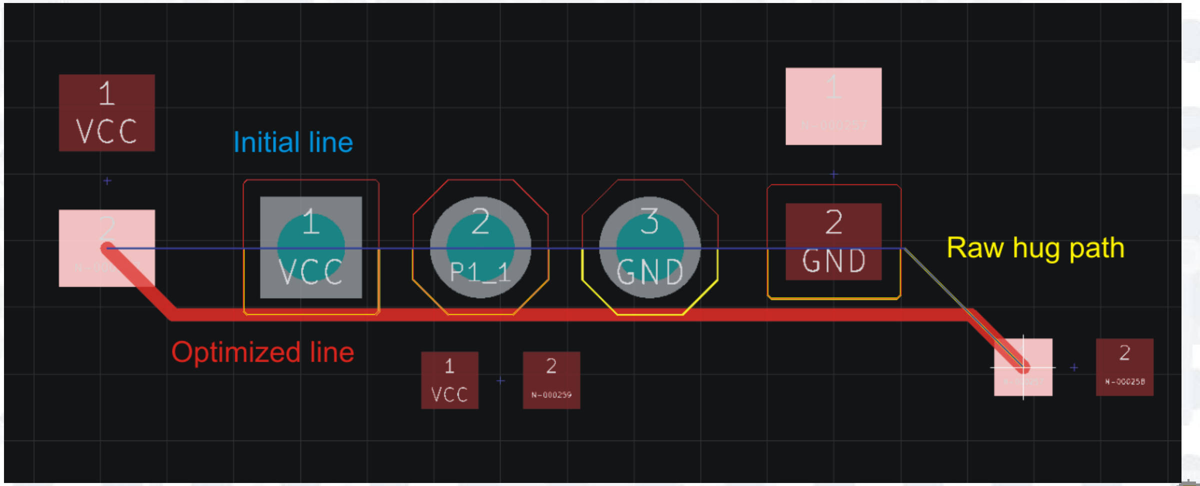
- User moves cursor.
- Compute initial head line.
- Hug solids.
- Check springback stack collisions (and rewind if possible).
- Iteratively shove lines, force-propagate vias, backpropagate solids.
- Optimize all affected lines/vias.
- Store optimized snapshot on springback stack.

Optimization algorithms

- Reduce length.
- Reduce corner count.
- Remove acute/right-angled corners.
- Tighten traces.
- Align traces/vias.
- Pad auto-neckdown.
- All of the above use a lot of collision checking → efficient storage is a must!

Example: merging segments

- For each line, take all vertex pairs separated by more than one vertex.
- Try to find a 2-segment 45-degree bypass that doesn't collide with anything on board.
- If found, connect the vertices with the bypass and remove the previous path



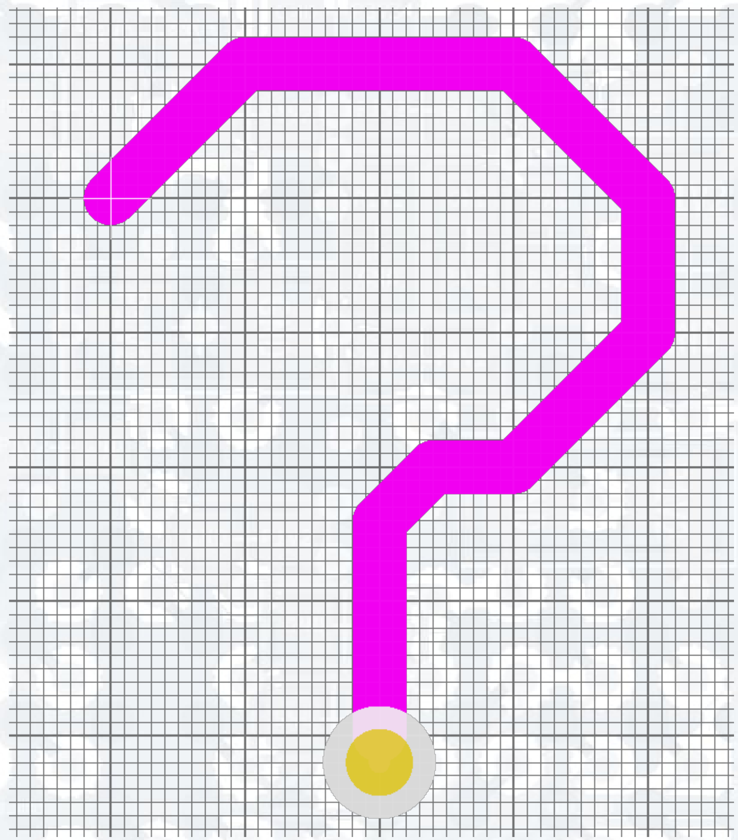
Status

- Single trace interactive routing available in Kicad since September 2013.
- Already used for routing complex projects.
- Currently working on differential pairs and length matched traces (meandering).
- Release coming soon.

Future outlook

- Hugging keepout zones & board outline.
- Improved optimizer: area restrictions, differential pair awareness.
- Dynamic joints computed with R-Tree instead of hash table.
- Auto-finish trace mode.
- … and more.

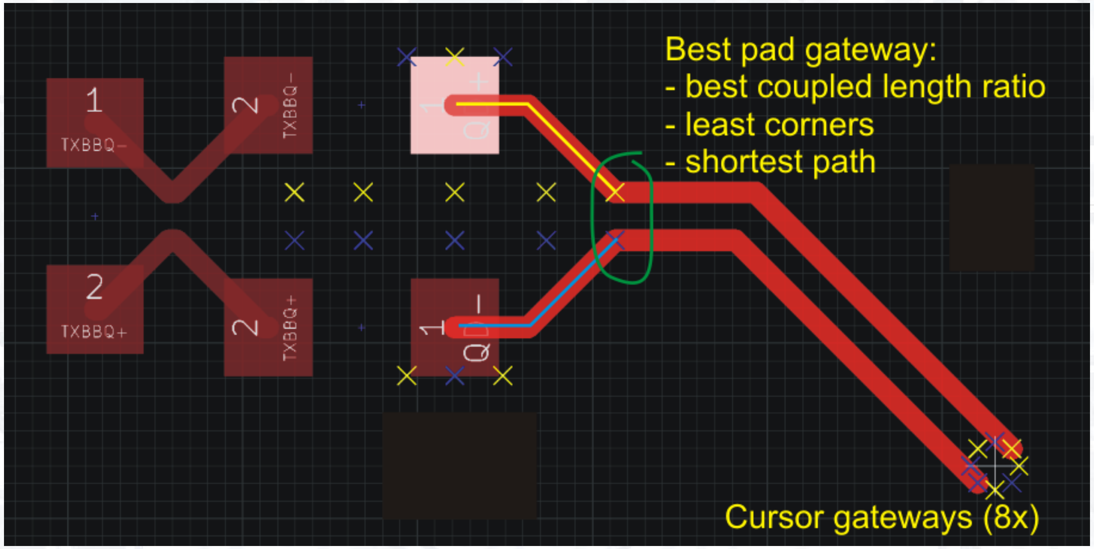
Questions



Backup slides

Differential pair routing

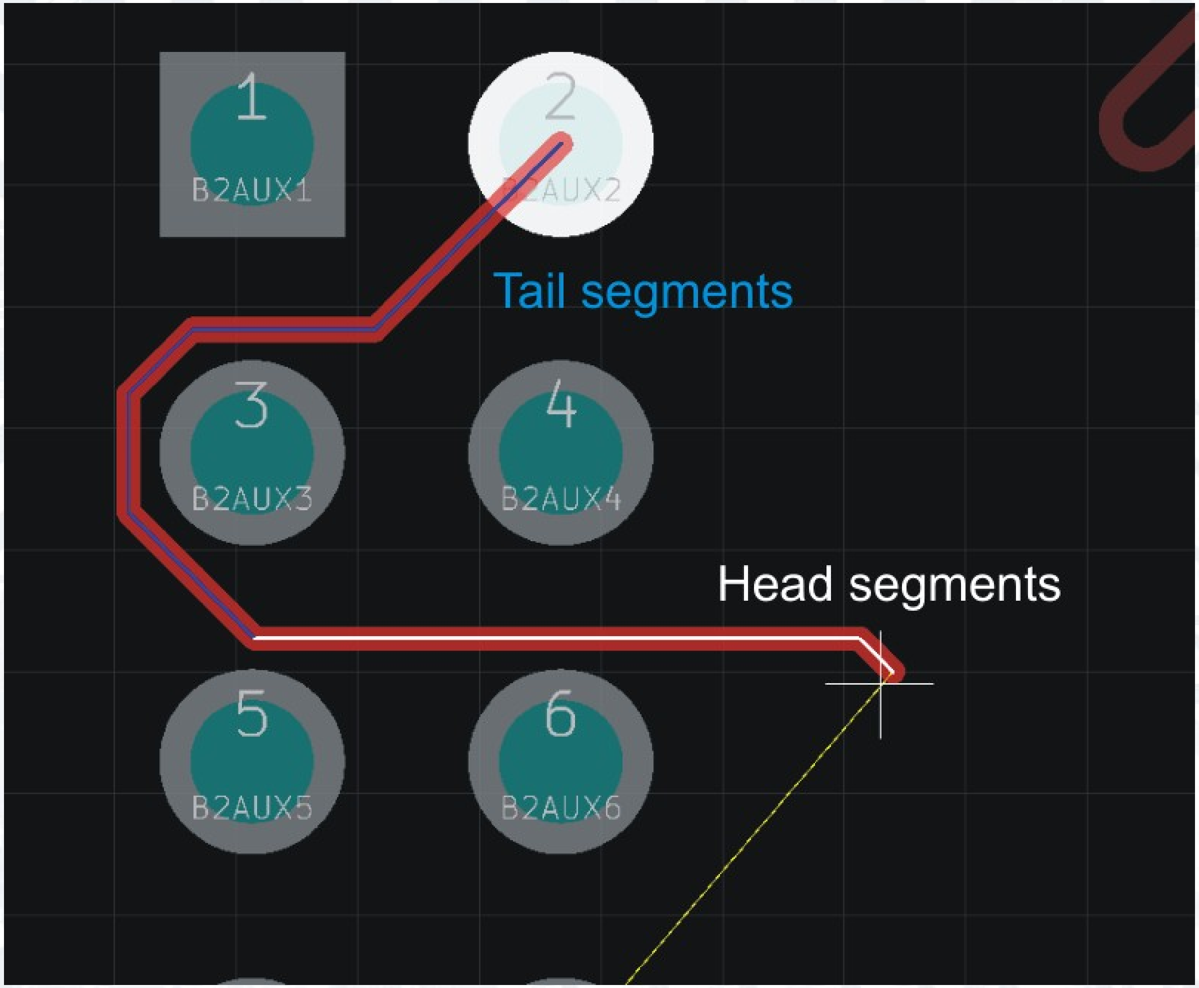
- Gateway matching algorithm.
- Generate a set of gateways for pad/via pairs or the cursor.
- Match them to maximize coupling and minimize corners and length. Currently brute-force.



Follow mouse algorithm

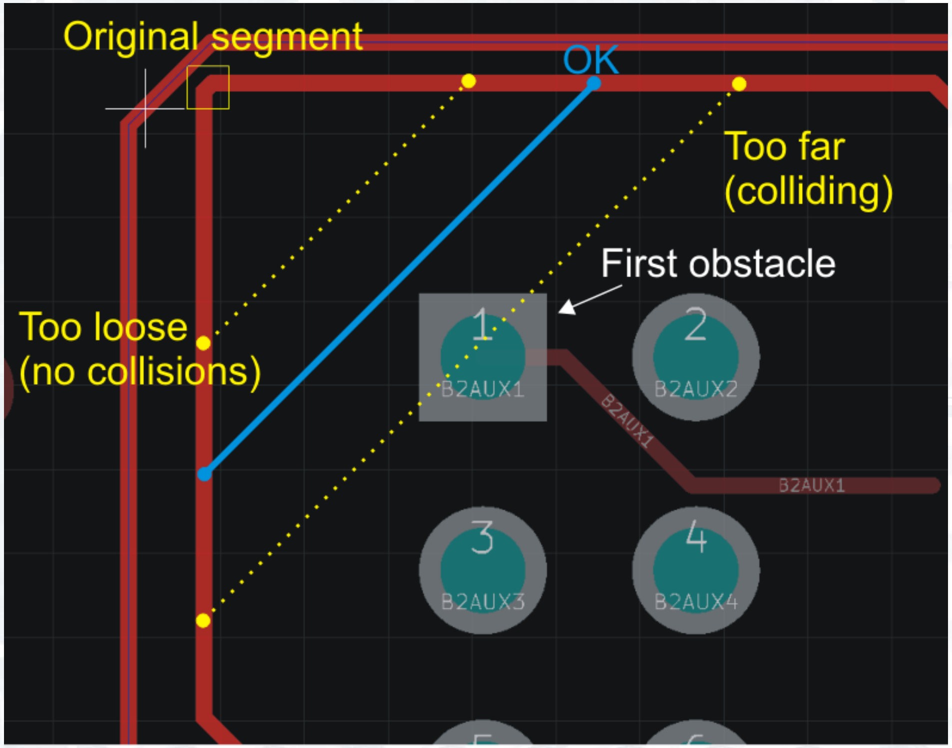
- Routed trace has a tail and a head.
- Tail is formed by the segments that have already hugged some solid obstacles.
- Head is the result of hugging the most recent obstacle (if any).
- Clip the tail upon intersection with the head.
- Remove some tail segments when head forms a non-obtuse angle with the tail.
- Run optimizer on the segments near head-tail transition to make routing smoother.

Follow mouse algorithm



Optimization - tightening

- Simple divide-and-conquer algorithm.
- Can be also used to remove right/acute corners or tighter shove springback.



Technicalities

- Written in C++ / Boost (optional, unordered)
- Separate data model.
- No UI library dependencies.
- Can be integrated into other F/OSS EDA tools without much hassle.