Construction of All Rectilinear Steiner Minimum Trees on the Hanan Grid

Sheng-En David Lin and Dae Hyun Kim

Presenter: Dae Hyun Kim (Assistant Professor)
daehyun@eecs.wsu.edu

School of Electrical Engineering and Computer Science
Washington State University

ISPD'18, Seaside, CA (3/26/2018)

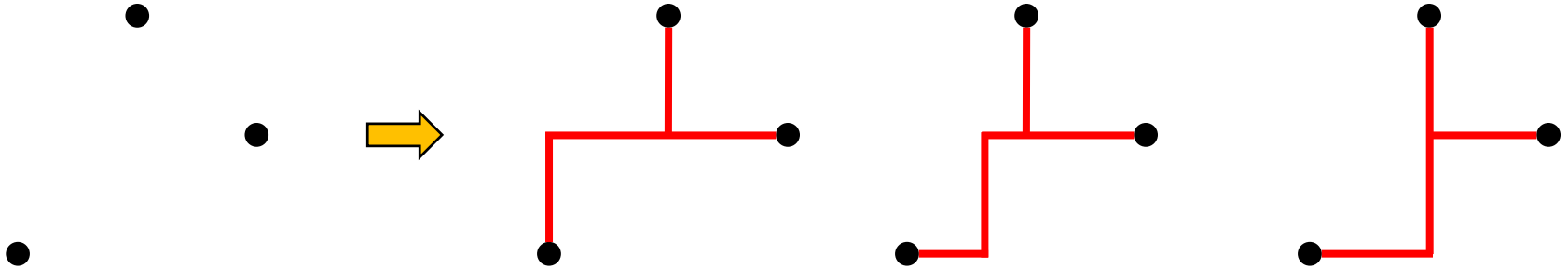
Sponsors

- DARPA YFA D16AP00119

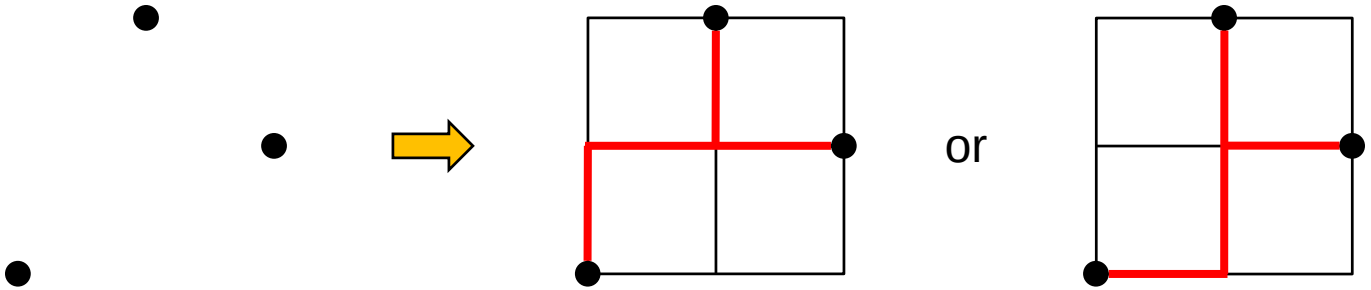
- Washington State University 125679-002

Overview and Motivation

- Rectilinear Steiner Minimum Tree (RSMT)
  - Shortest wire length



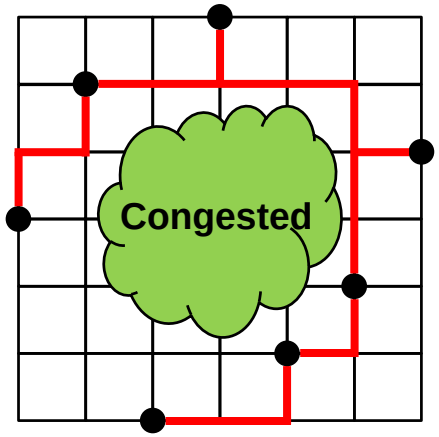
  - On the Hanan Grid



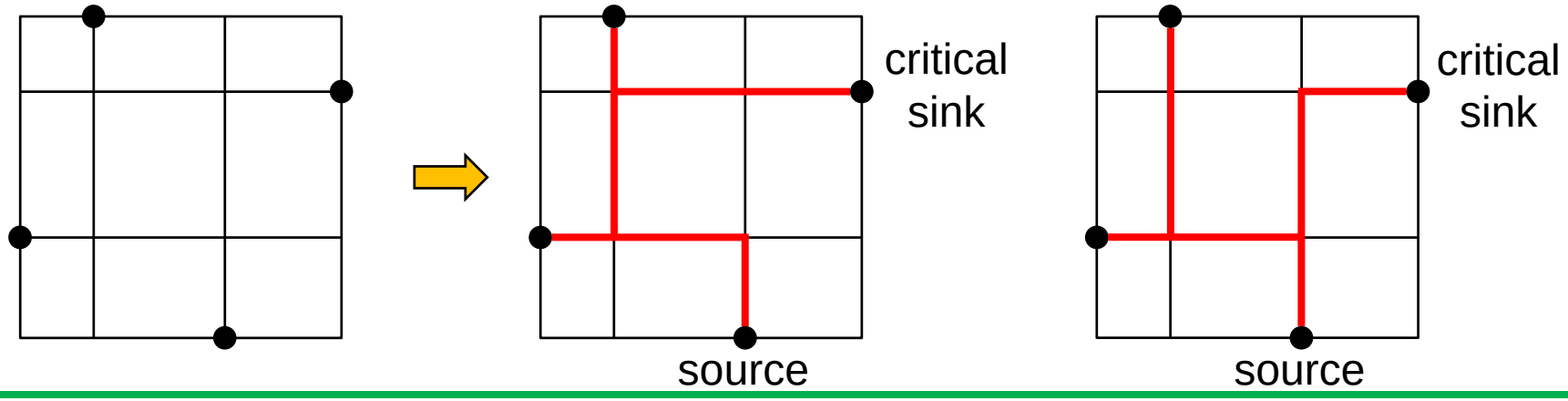
  - NP-complete

Overview and Motivation

- All Rectilinear Steiner Minimum Trees on the Hanan Grid
  - Applications
    - Congestion-aware routing
    - Congestion estimation

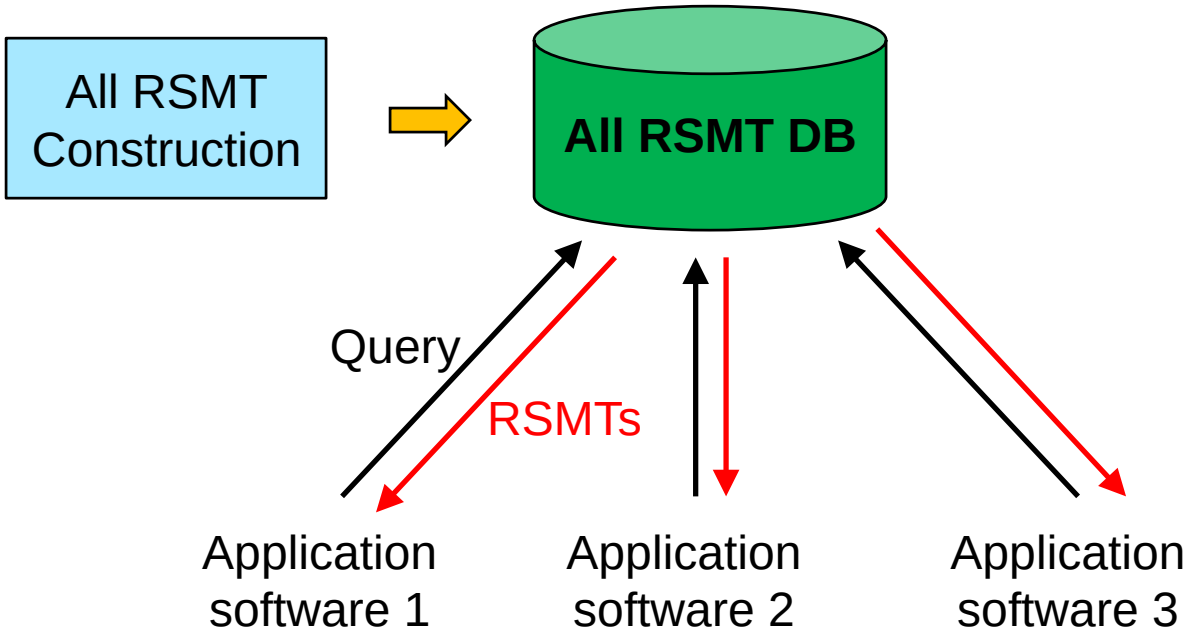


    - Minimization of the source-to-critical-sink length and the total wire length
    - …



Overview and Motivation

- Goal

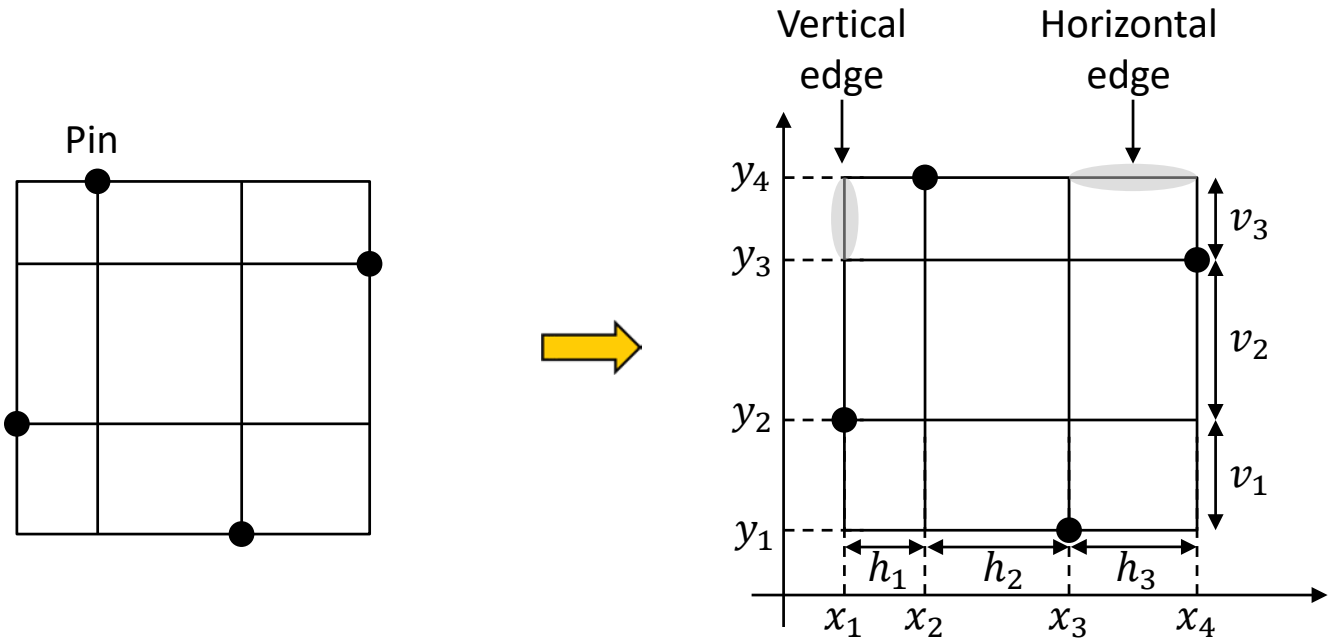


Outline

- Review
  - Chris Chu, “FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design,” TCAD’08.
- Goal
- Construction of All Rectilinear Steiner Minimum Trees on the Hanan Grid
- Example
- Simulation Results
- Conclusion

FLUTE – Lookup-Table-Based RSMT Construction

- Edge decomposition



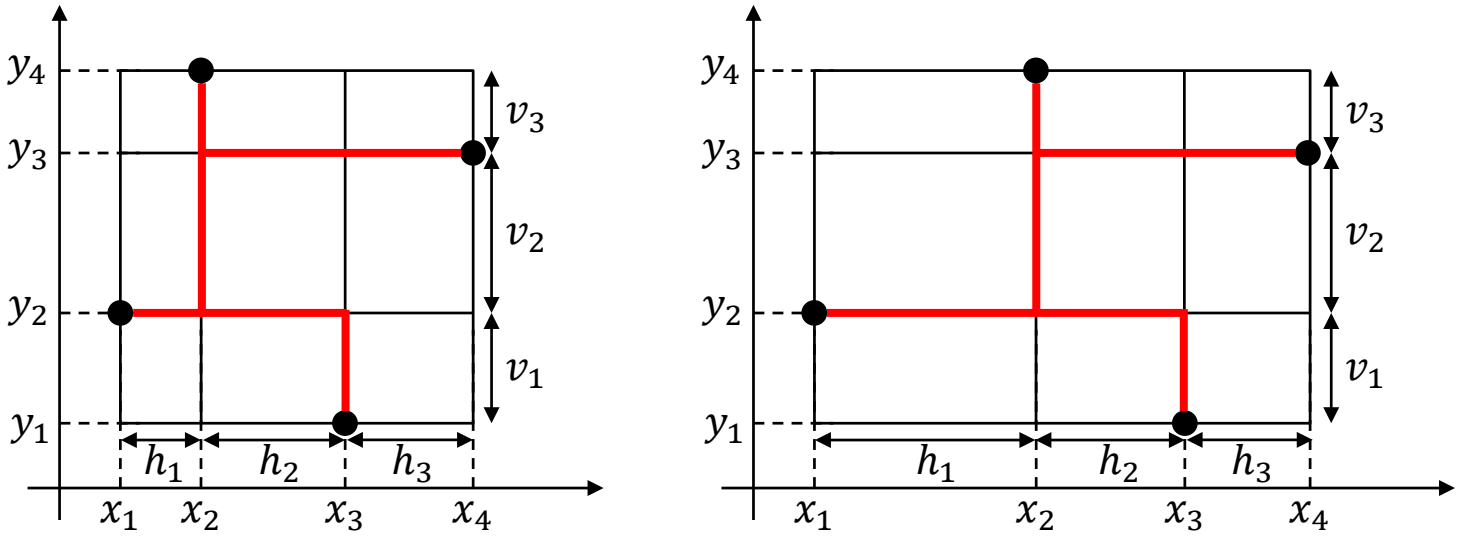
FLUTE

- Wire length computation

$$L = 1 \cdot h_1 + 2 \cdot h_2 + 1 \cdot h_3 + 1 \cdot v_1 + 1 \cdot v_2 + 1 \cdot v_3$$
$$= (1,2,1,1,1,1) \cdot (h_1, h_2, h_3, v_1, v_2, v_3)$$

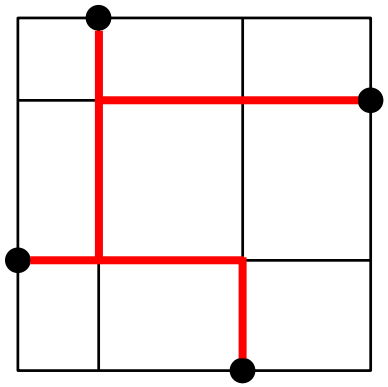
Wirelength Vector (Topology-dependent, constant)

Coordinate-dependent (variable)



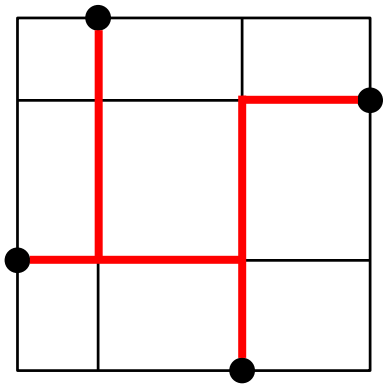
FLUTE

- Observation
  - The topology-dependent vectors of some topologies cannot generate RSMTs.
  - Example
    - (a, b, c, d, e, f): $L_1 = a \cdot h_1 + b \cdot h_2 + c \cdot h_3 + d \cdot v_1 + e \cdot v_2 + f \cdot v_3$
    - (a, b, c+1, d, e, f): $L_2 = a \cdot h_1 + b \cdot h_2 + (c+1) \cdot h_3 + d \cdot v_1 + e \cdot v_2 + f \cdot v_3$
    - $L_2 - L_1 = h_3 > 0$



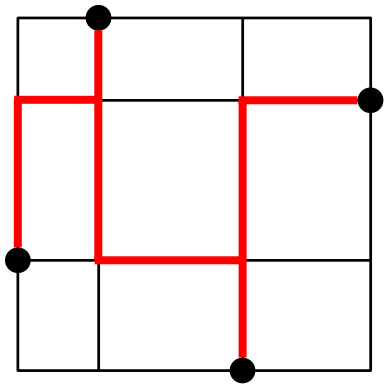
$(1,2,1,1,1,1)$

Potentially optimal? YES



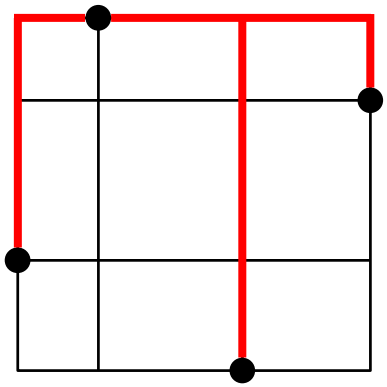
$(1,1,1,1,2,1)$

YES



$(1,1,1,1,\mathbf{3},1)$

NO

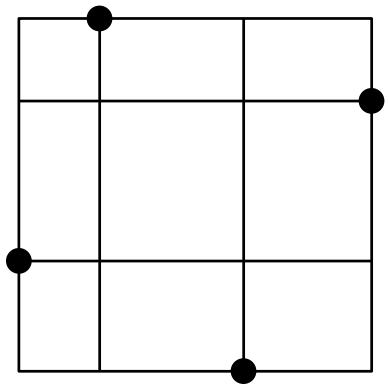


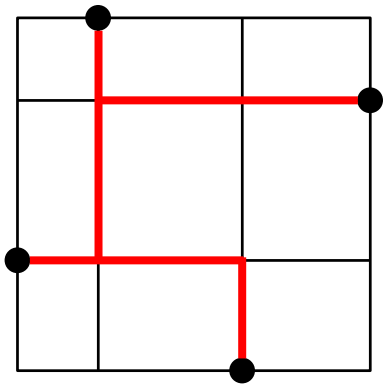
$(1,1,1,1,2,\mathbf{3})$

NO

FLUTE

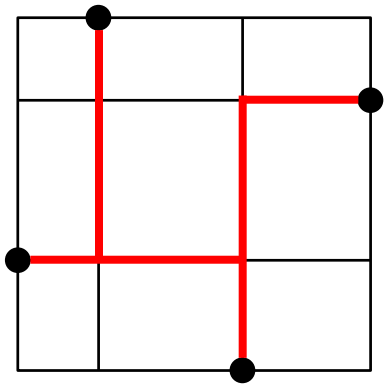
- Optimal topology





$(1,2,1,1,1,1) \cdot (h_1, h_2, h_3, v_1, v_2, v_3)$

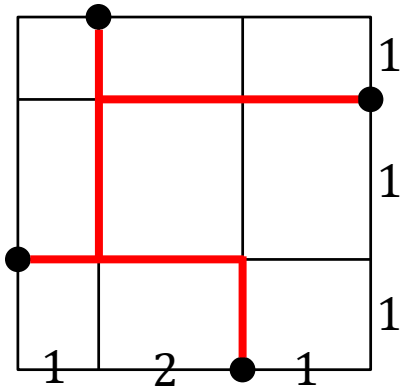
OR



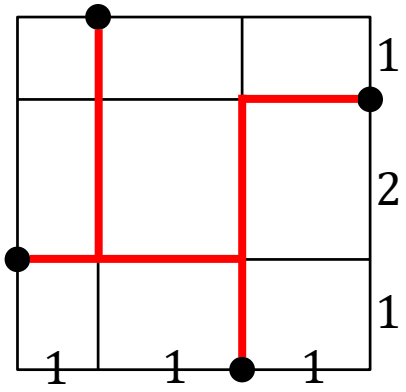
$(1,1,1,1,2,1) \cdot (h_1, h_2, h_3, v_1, v_2, v_3)$

FLUTE

- Find all potentially optimal wirelength vectors for each set of relative pin locations.

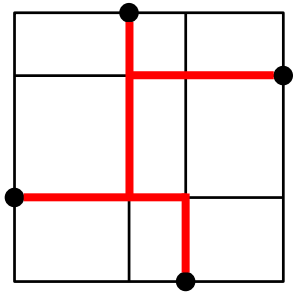


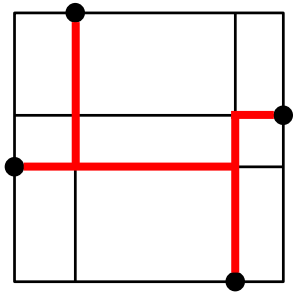
(1,2,1,1,1,1)



(1,1,1,1,2,1)

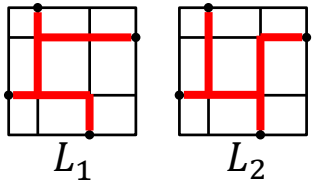
- For given pin locations, compute L and obtain a shortest-length topology.

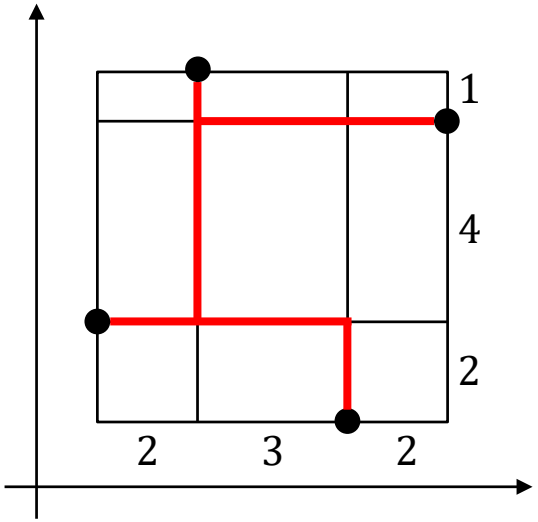




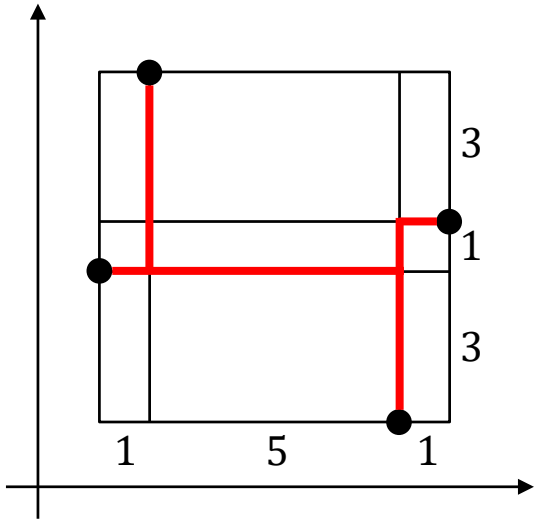
FLUTE

- Example
  - $L_1 = (1,2,1,1,1,1) \cdot (h_1, h_2, h_3, v_1, v_2, v_3)$
  - $L_2 = (1,1,1,1,2,1) \cdot (h_1, h_2, h_3, v_1, v_2, v_3)$

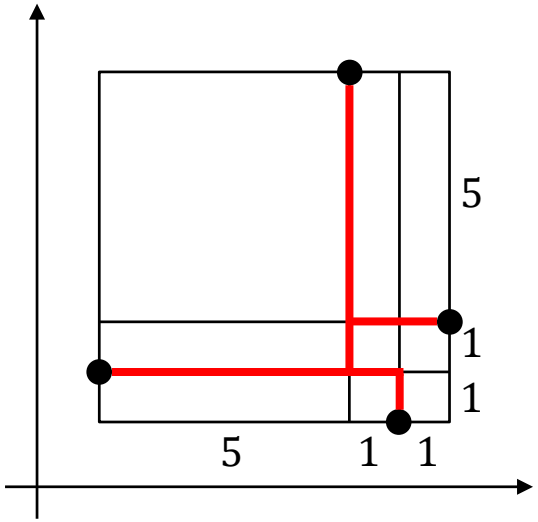




$L_1 = (1,2,1,1,1,1) \cdot (2,3,2,2,4,1) = 17$
$L_2 = (1,1,1,1,2,1) \cdot (2,3,2,2,4,1) = 18$



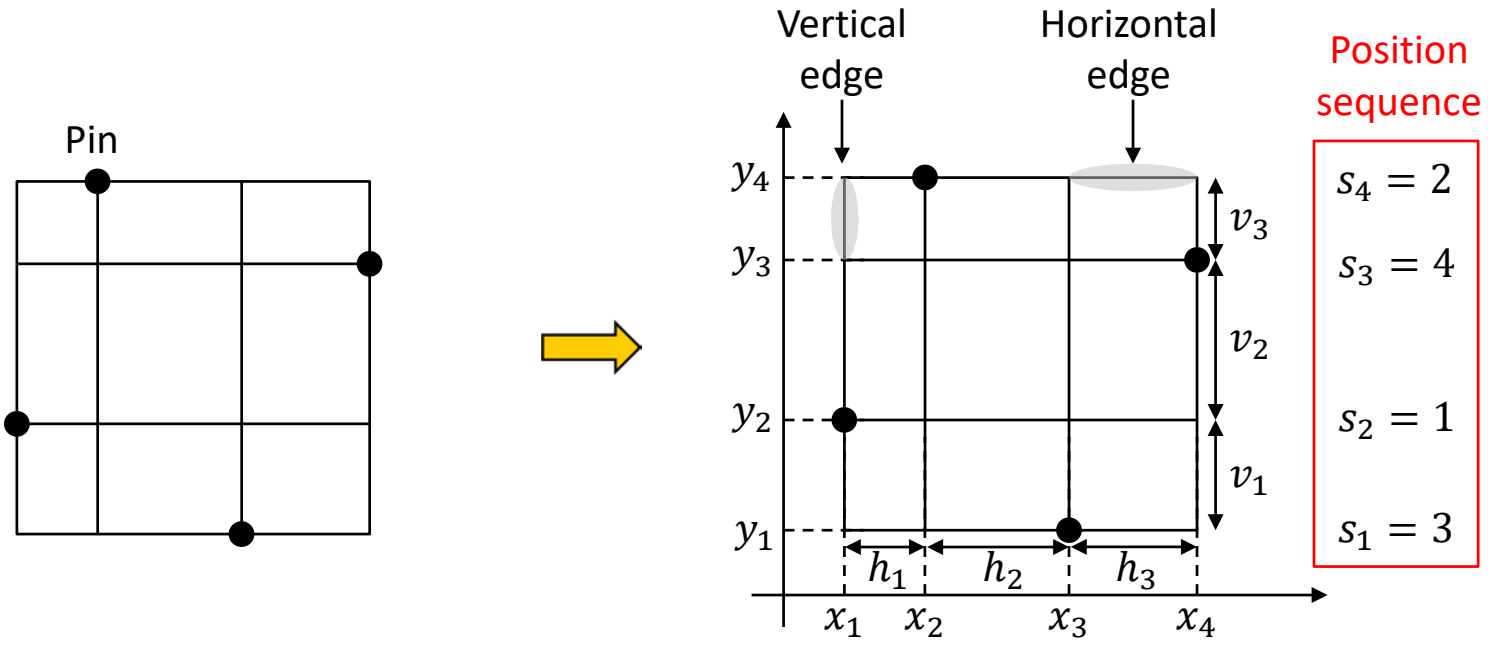
$L_1 = (1,2,1,1,1,1) \cdot (1,5,1,3,1,3) = 19$
$L_2 = (1,1,1,1,2,1) \cdot (1,5,1,3,1,3) = 15$



$L_1 = (1,2,1,1,1,1) \cdot (5,1,1,1,1,5) = 15$
$L_2 = (1,1,1,1,2,1) \cdot (5,1,1,1,1,5) = 15$

FLUTE

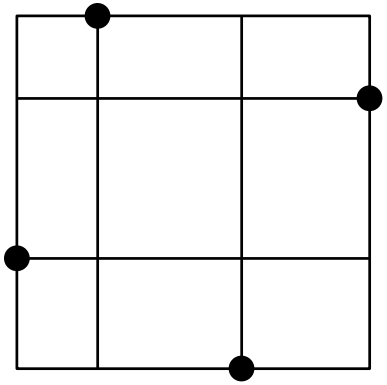
- Position sequence (pin group)



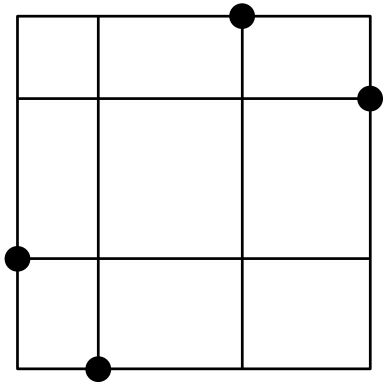
  - Sort the pins in the increasing order of the y-coordinates (y_1, y_2, y_3, y_4).
  - Obtain their x-coordinates (x_3, x_1, x_4, x_2).
  - Obtain the indices (3,1,4,2).

FLUTE

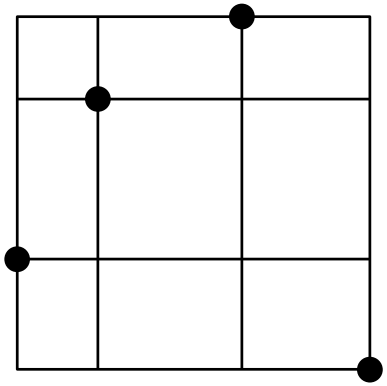
- Example (position sequence)



3 1 4 2



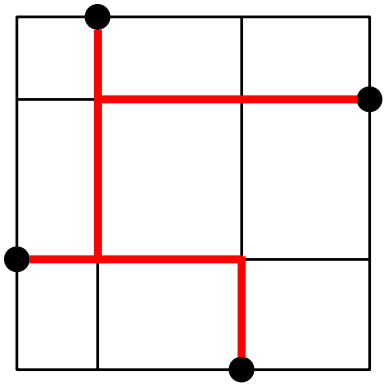
2 1 4 3



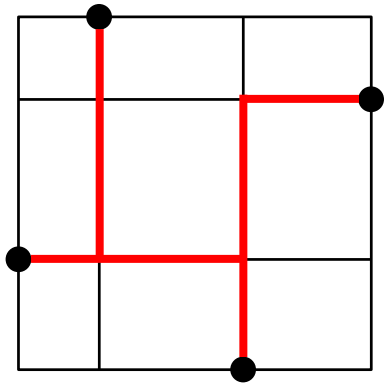
4 1 2 3

FLUTE

- Potentially Optimal Wirelength Vector (POWV)
  - For each position sequence
  - Example for position sequence (3142)
    - (1 2 1 1 1 1)
    - (1 1 1 1 2 1)
- Potentially Optimal Steiner Tree (POST)
  - For each POWV



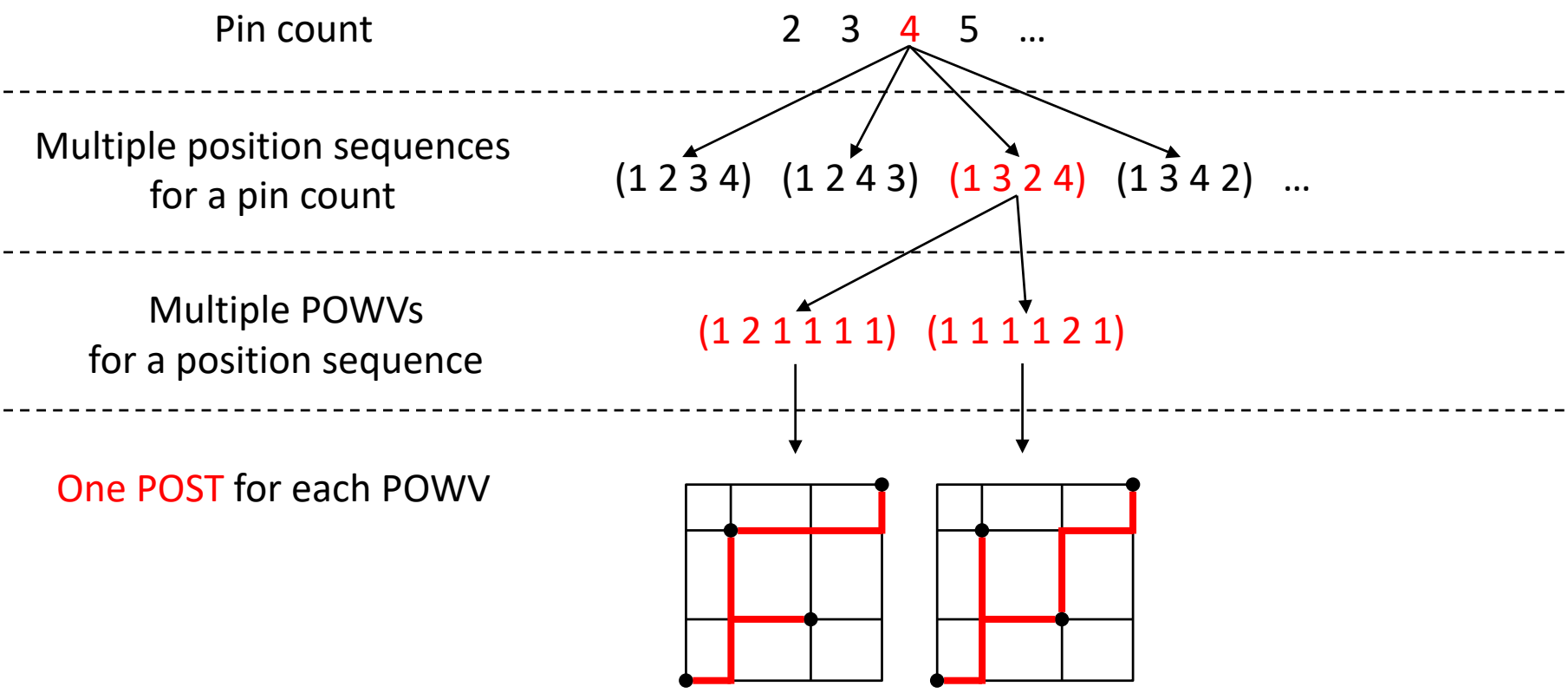
(1,2,1,1,1,1)



(1,1,1,1,2,1)

FLUTE

- Database structure



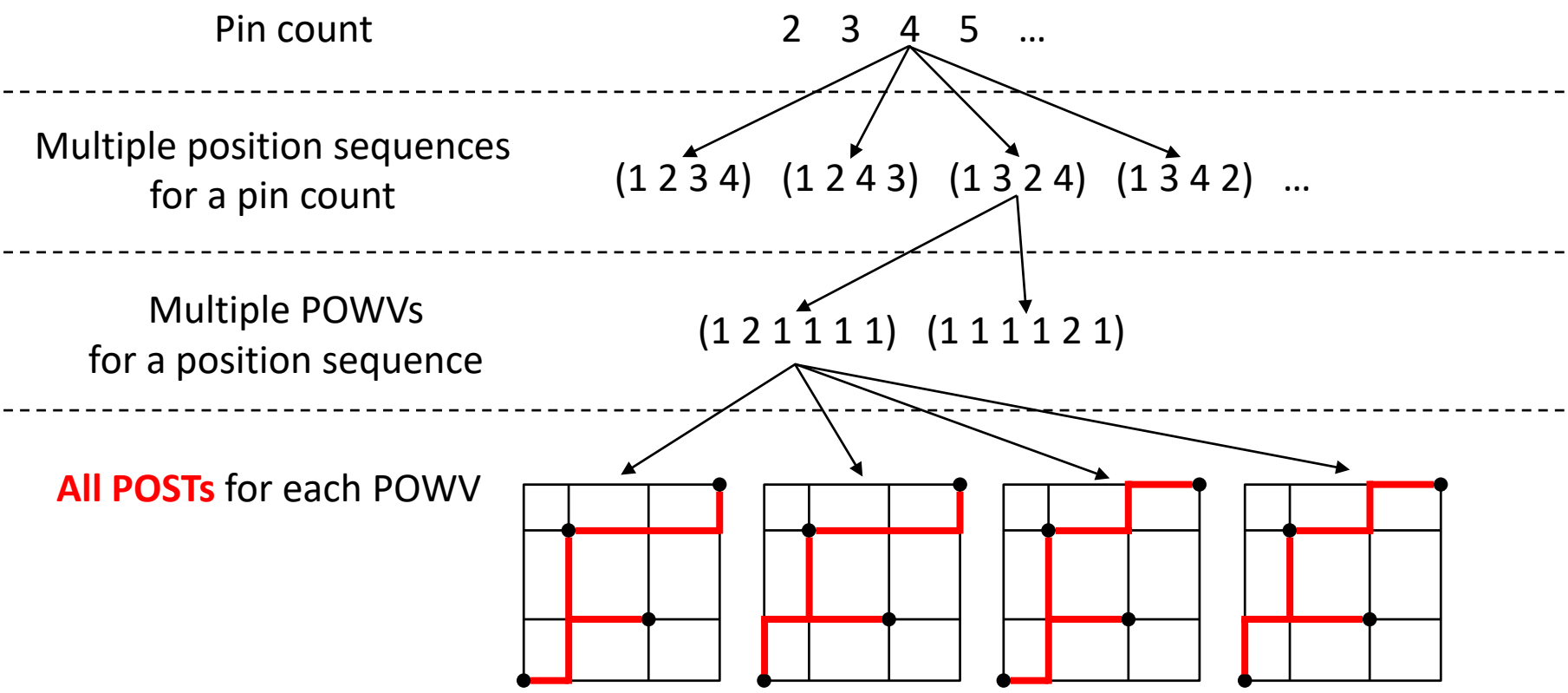
FLUTE

- How to find POWVs and POSTs
  - Please see the FLUTE paper.

  - Chris Chu and Yiu-Chung Wong, “FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, Issue 1, Jan. 2008, pp. 70–83.

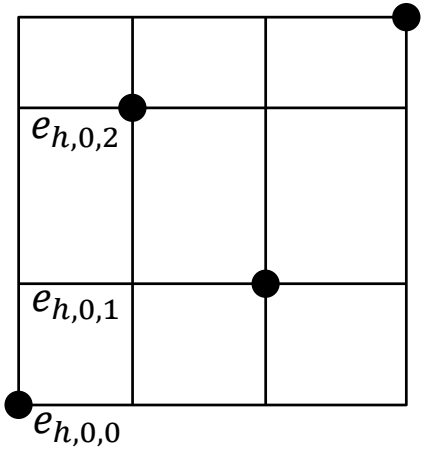
Goal

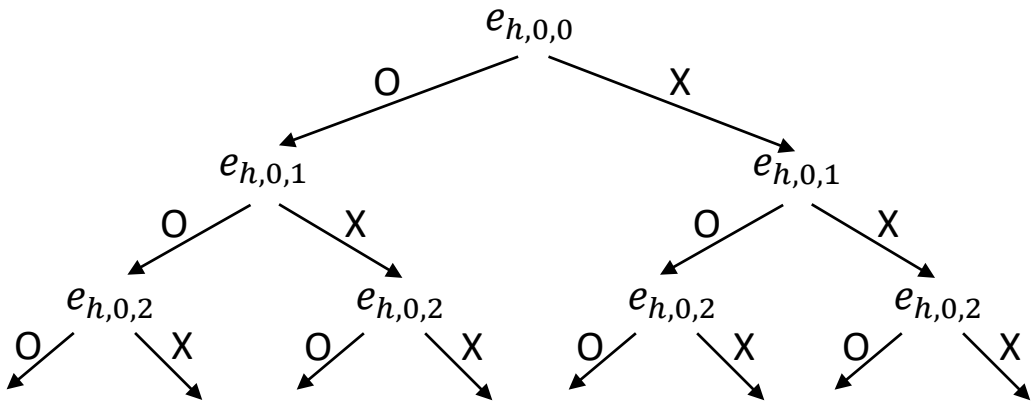
- Database structure



Brute-Force Algorithm

- Binary-tree-based (enumeration)



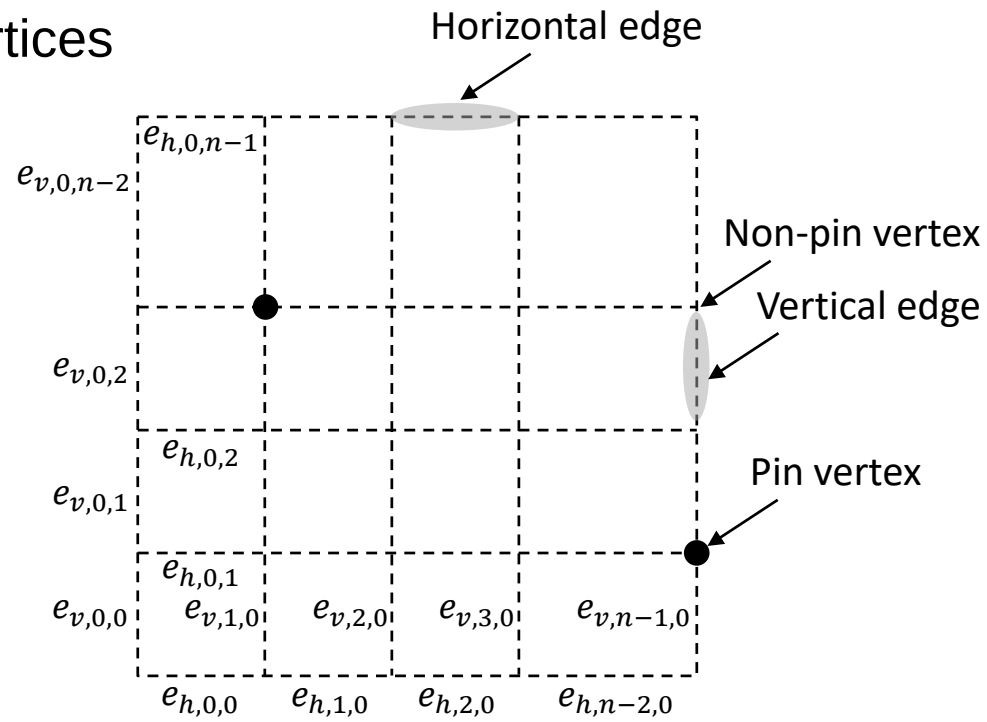


  - Whenever a leaf node is reached, evaluate the branch to check
    - All the pins are connected.
    - Shortest
  - # Leaf nodes to check (n: # pins)
    - $2^{2n(n-1)}$
  - We develop a faster algorithm.

Terminologies

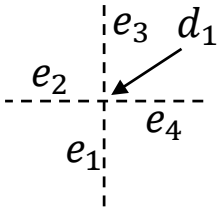
- Assume n distinct pins
  - For any two pins $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, $x_i \neq x_j$ and $y_i \neq y_j$.

- Edges, vertices

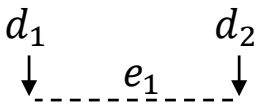


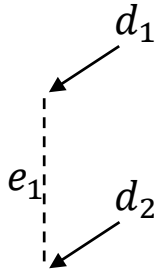
Terminologies

- Neighboring edges of a vertex
  - $NE(d_1) = \{e_1, e_2, e_3, e_4\}$

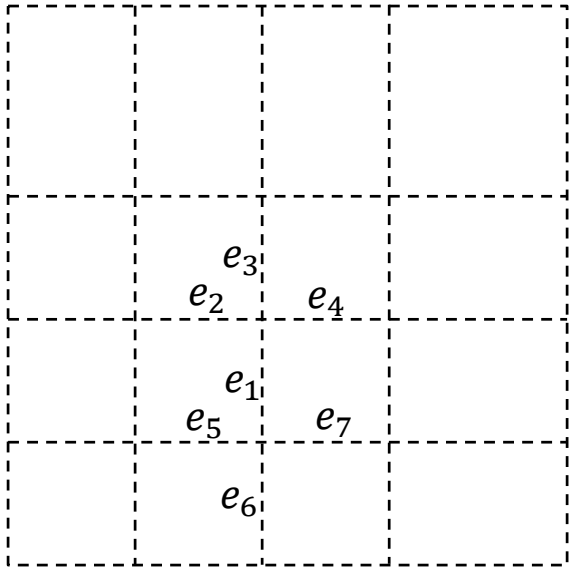


- Neighboring vertices of an edge
  - $NV(e_1) = \{d_1, d_2\}$



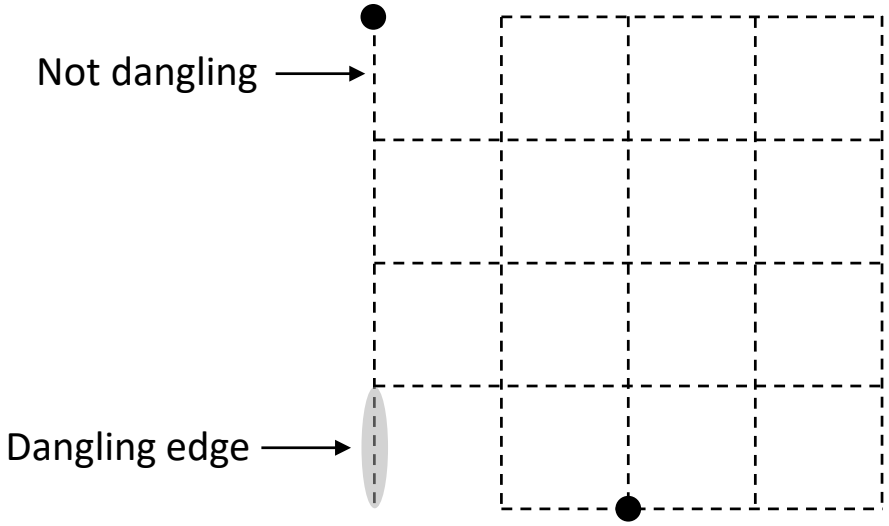


- Neighboring edges of an edge
  - $NE(e_1) = NE(NV(e_1)) = \{e_2, e_3, e_4, e_5, e_6, e_7\}$



Terminologies

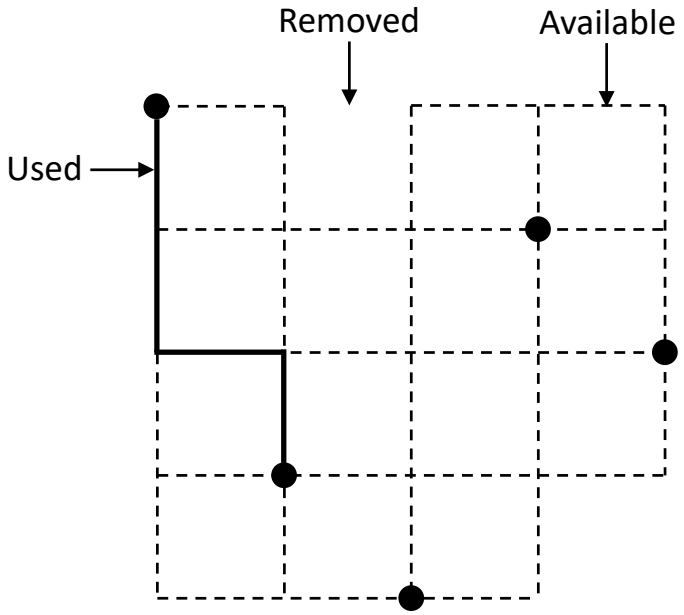
- Dangling edges
  - If a non-pin vertex is connected to only one edge, the edge is dangling.



- Theorem
  - A POST cannot have dangling edges.

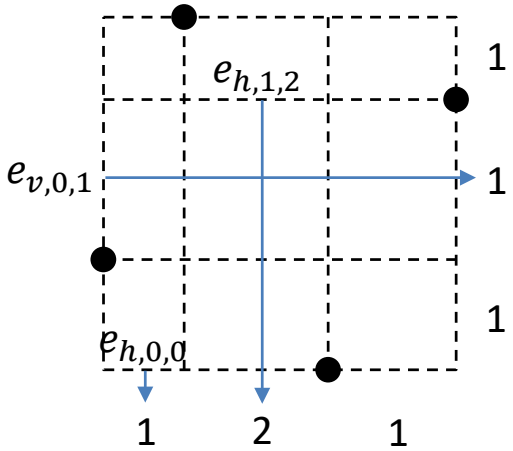
Terminologies

- Status of an edge
  - Used
  - Removed
  - Available (not used nor removed)



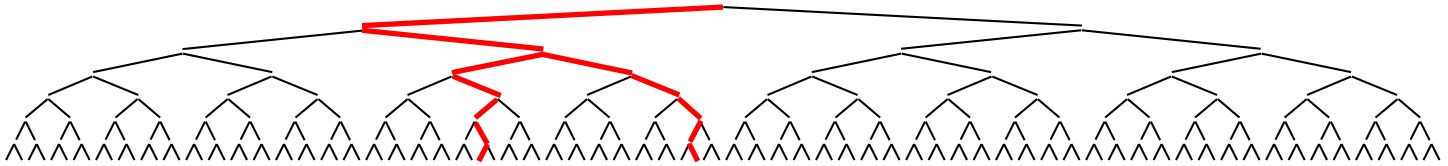
Terminologies

- POWV of an edge
  - POWV element corresponding to the edge
  - Example
    - POWV = (1 2 1 1 1 1)
    - $powv(e_{h,0,0})$ = 1
    - $powv(e_{h,1,2})$ = 2
    - $powv(e_{v,0,1})$ = 1



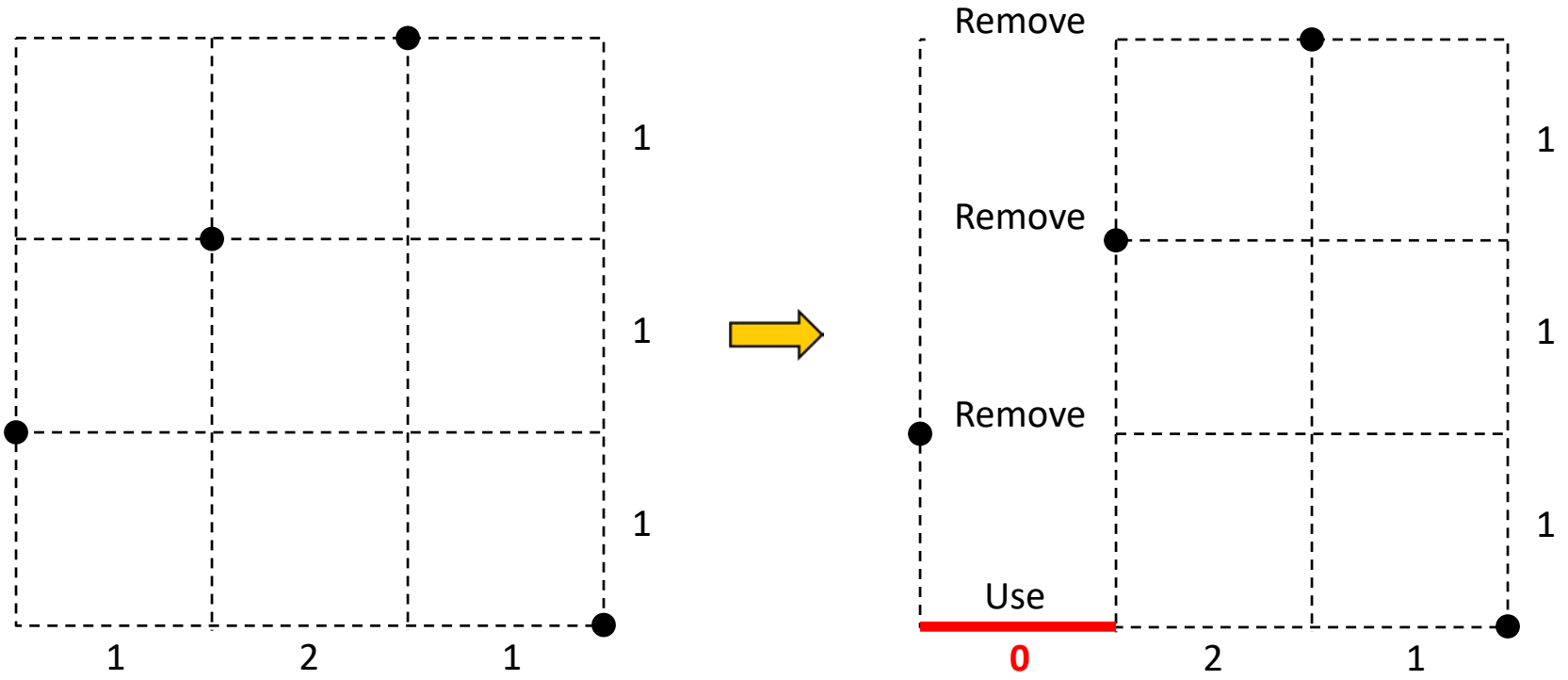
Construction of All POSTs for a Given POWV

- We apply speed-up techniques to reduce the search space size.
  - Pruning by Zero POWV elements
  - Pruning by must-use and must-remove edges
  - POST evaluation
  - Intermediate connectivity check



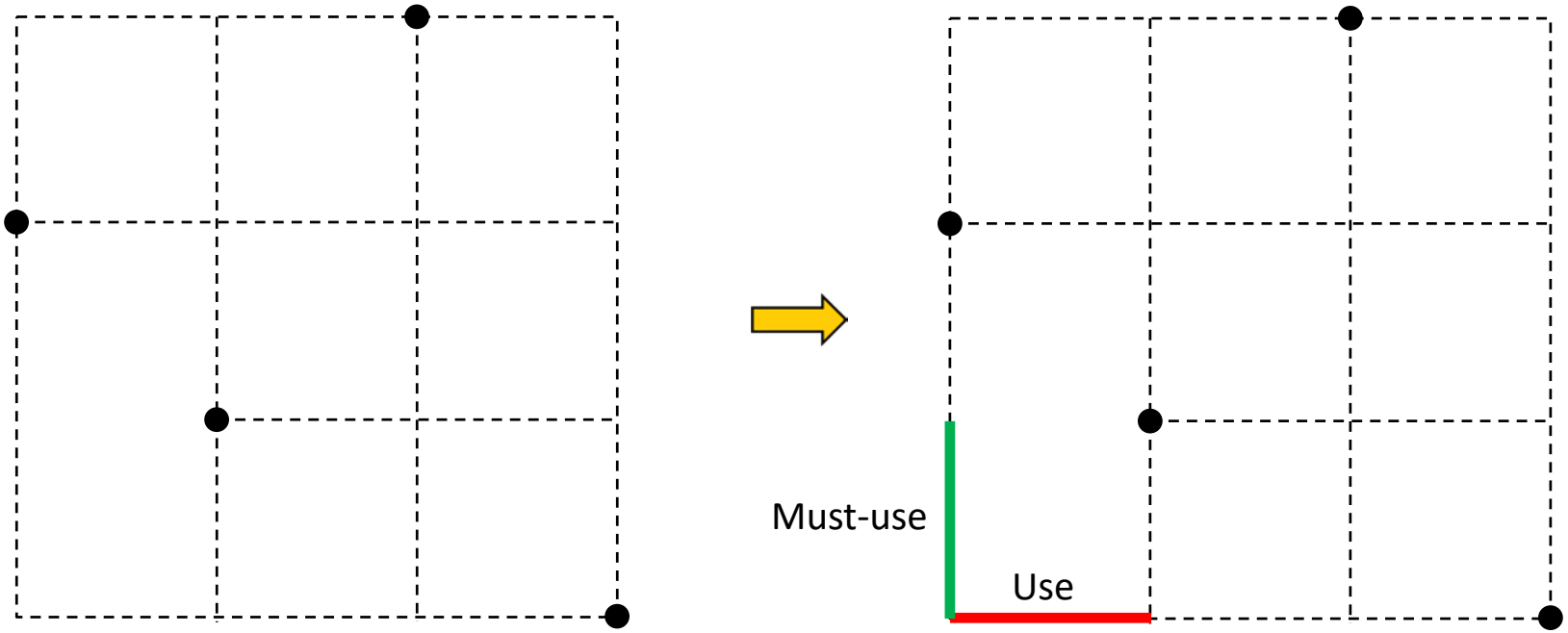
Construction of All POSTs for a Given POWV

- Pruning by zero POWV elements
  - If we decide to use an edge e, reduce $powv(e)$ by 1.
  - If $powv(e)$ becomes 0, remove all the available edges in the column/row.



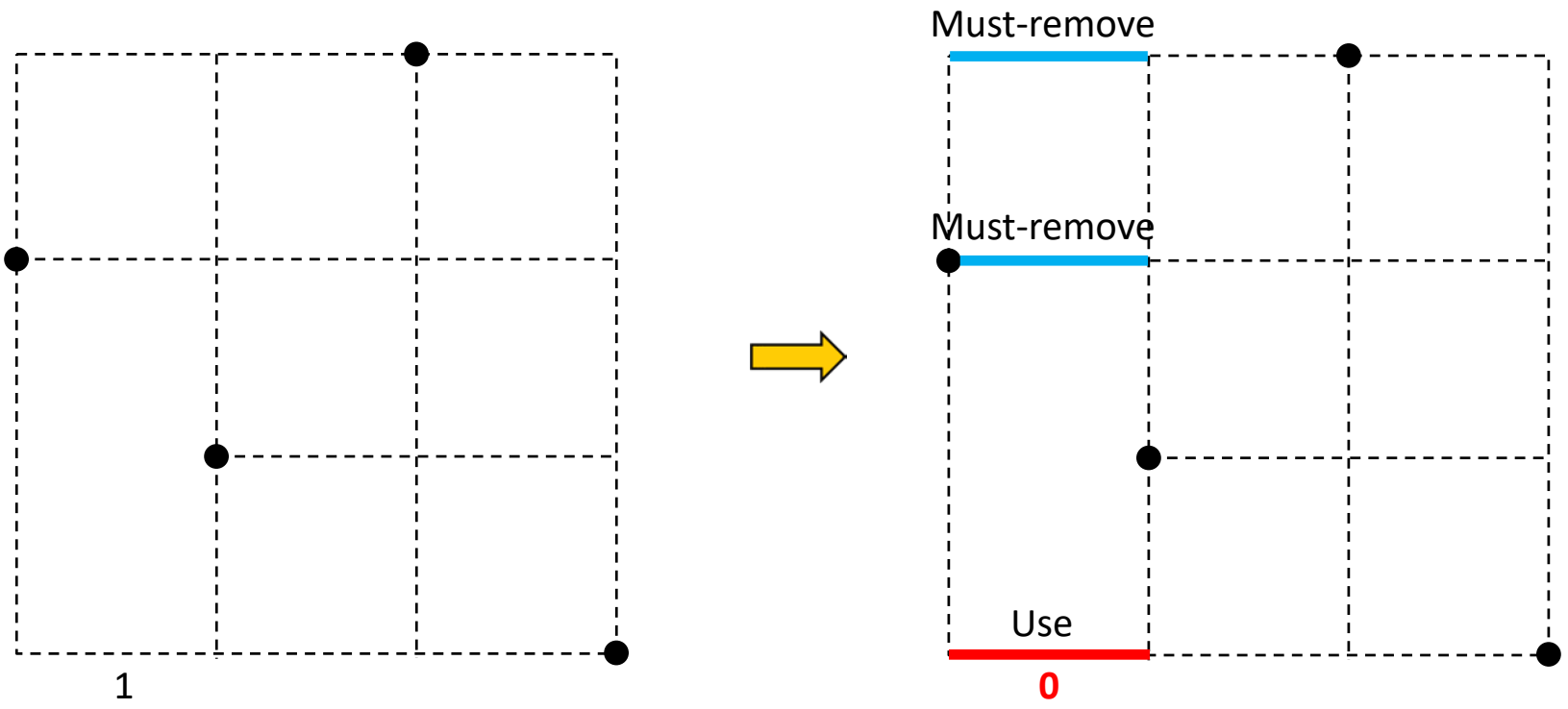
Construction of All POSTs for a Given POWV

- Pruning by must-use and must-remove edges
  - Must-use (remove) edges: Edges that must be used (removed)
  - Using or removing an edge forces us to use or remove some of its neighboring edges.
  - Use → Must-use (to avoid dangling edges)



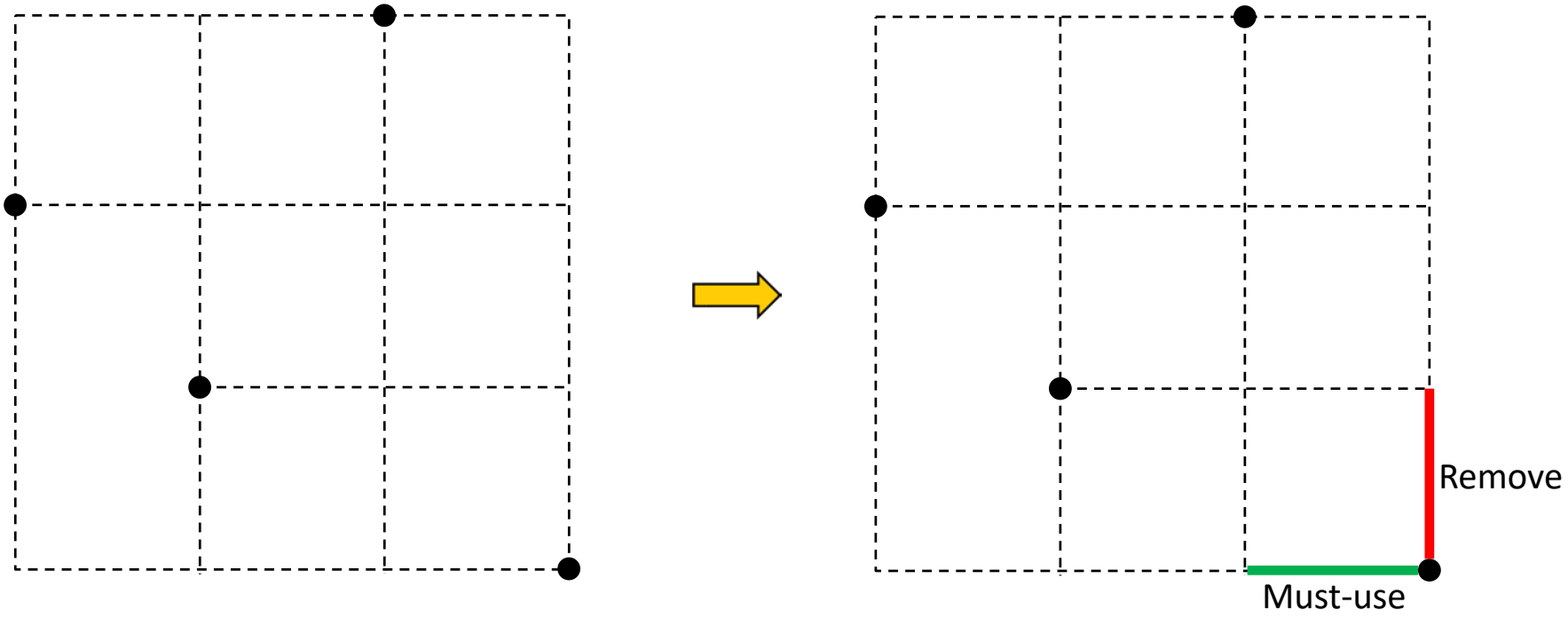
Construction of All POSTs for a Given POWV

- Pruning by must-use and must-remove edges
  - Use → Must-remove (zero POWV)



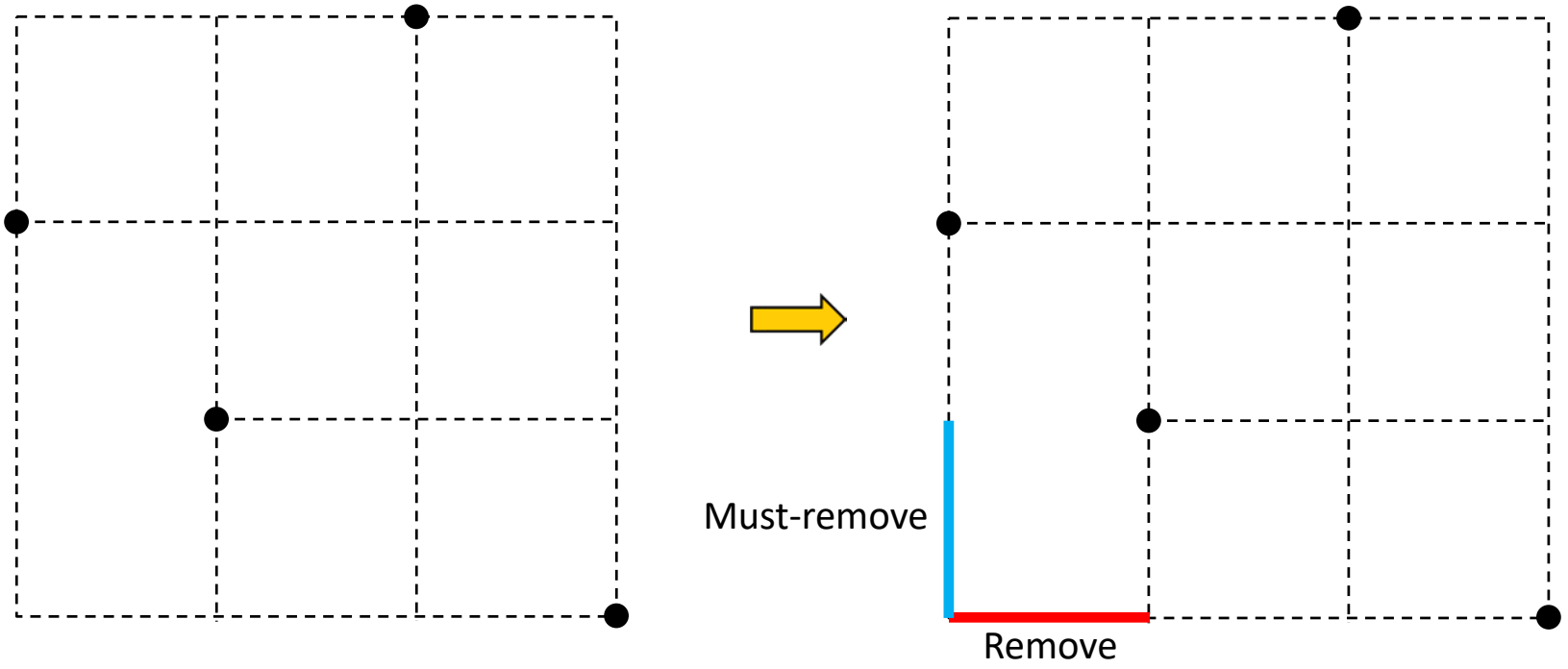
Construction of All POSTs for a Given POWV

- Pruning by must-use and must-remove edges
  - Remove → Must-use (to connect the pin at a dead end)



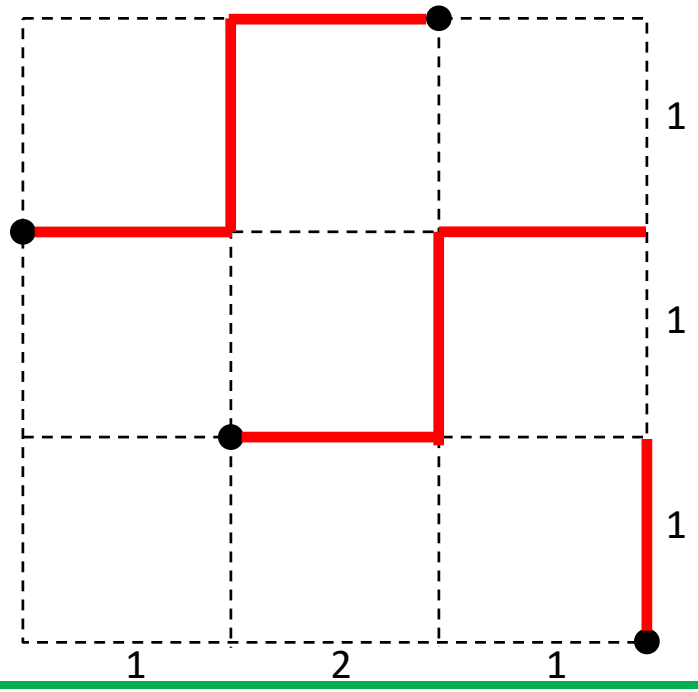
Construction of All POSTs for a Given POWV

- Pruning by must-use and must-remove edges
  - Remove → Must-remove (to remove dangling edges)



Construction of All POSTs for a Given POWV

- POST evaluation
  - Evaluation of a topology checks whether the topology connects all the pins.
  - We evaluate a topology only when all the POWV elements become zero.
    - We can evaluate a topology even if we do not reach a leaf node in the binary tree.

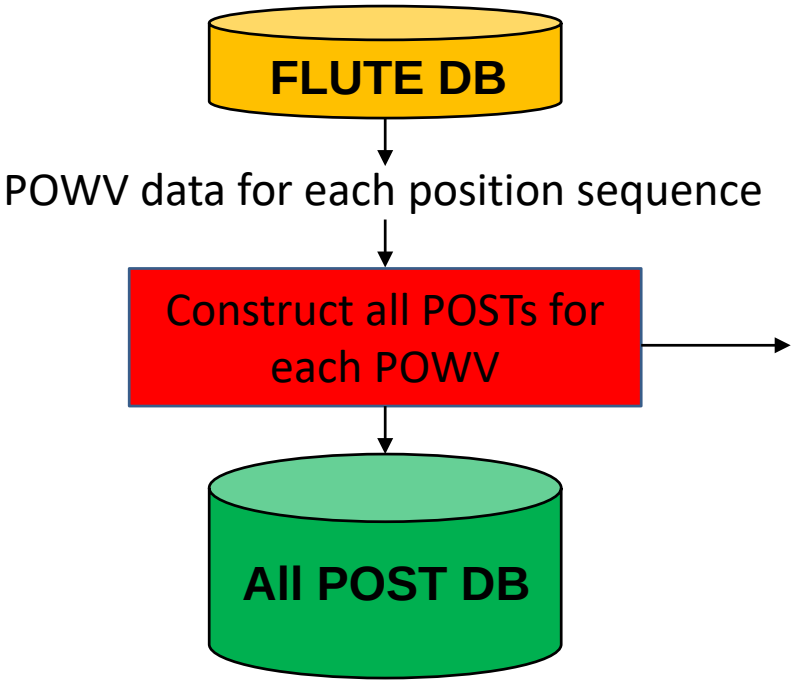


Construction of All POSTs for a Given POWV

- Intermediate connectivity check
  - In some cases, a topology is disconnected, especially after using and/or removing edges consecutively.
  - In this case, it is meaningless to proceed further.
  - Thus, we sometimes check whether a topology still connects all the pins through used and available edges.
    - Connected: Proceed further
    - Disconnected: Roll-back
  - Criterion
    - # consecutively used and removed edges $\geq$ threshold
  - Breadth-first search

Construction of All POSTs for a Given POWV

- Algorithm



Input: Pin locations and a POWV (powv).

1: Ordered set $E = (e_h(0, 0), \ldots, e_v(n-1, n-2))$;
2: $R = \{\}$;
3: Call **recursive_construction** (powv, E, R, 0);
4: Return R;

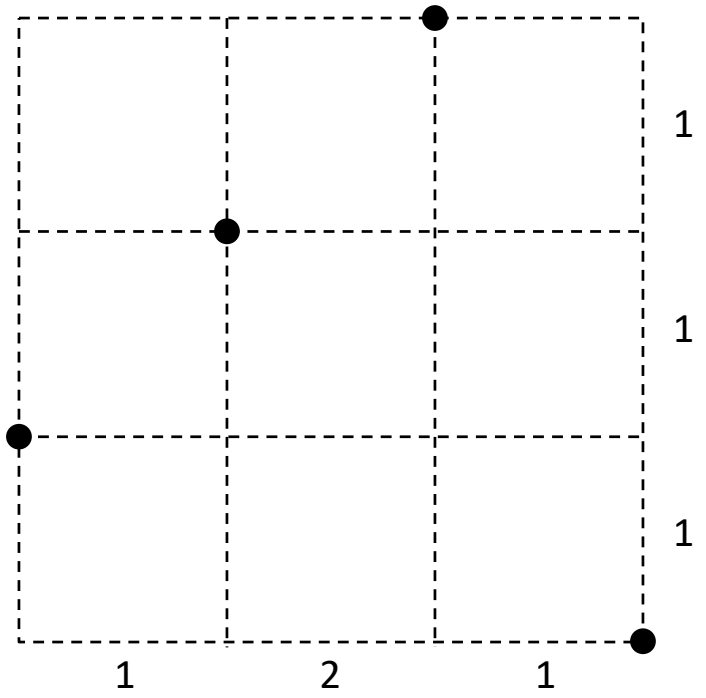
Construction of All POSTs for a Given POWV

```
Function: recursive_construction (powv, E, R, index)
 5: if powv == 0 or index == E.size then
 6:     if Current graph G connects all the pins then
 7:         Insert G into R;
 8:     end if
 9:     return;
10: end if
11: e = E[index];
12: if e is a used or removed edge then
13:     Call recursive_construction (powv, E, R, index+1);
14:     return;
15: end if
16: if powv(e) > 0 then
17:     Call use_or_remove_and_prune (e, NULL, powv);
18:     if # must-use and must-remove edges ≥ threshold then
19:         if Current graph G connects all the pins then
20:             recursive_construction (powv, E, R, index+1);
21:         end if
22:     else
23:         recursive_construction (powv, E, R, index+1);
24:     end if
25:     Roll back the must-use and must-remove edges.
26: end if
27: Call use_or_remove_and_prune (NULL, e, powv);
28: if # must-use and must-remove edges ≥ threshold then
29:     if Current graph G connects all the pins then
30:         recursive_construction (powv, E, R, index+1);
31:     end if
32: else
33:     recursive_construction (powv, E, R, index+1);
34: end if
35: Roll back the must-use and must-remove edges.
```

```
Function: Use_or_remove_and_prune (u, m, powv)
Input: (Edge u to use, Edge m to remove, a POWV (powv)).
 1: U = {u};
 2: M = {m};
 3: while U.size + M.size > 0 do
 4:     while U.size > 0 do
 5:         for each e ∈ U do
 6:             if e is a removed edge or powv(e) == 0 then
 7:                 return invalid_topology;
 8:             end if
 9:             Use e in G;
10:             powv(e) = powv(e) - 1;
11:             if powv(e) == 0 then
12:                 Insert all available edges in PE(powv(e)) into M;
13:             end if
14:             Insert all must-use edges in NE(e) into U.
15:         end for
16:     end while
17:     while M.size > 0 do
18:         for each e ∈ M do
19:             if e is a used edge then
20:                 return invalid_topology;
21:             end if
22:             Remove e from G;
23:             Insert all dangling edges in NE(e) into M;
24:             Insert all must-use edges in NE(e) into U;
25:         end for
26:     end while
27: end while
```

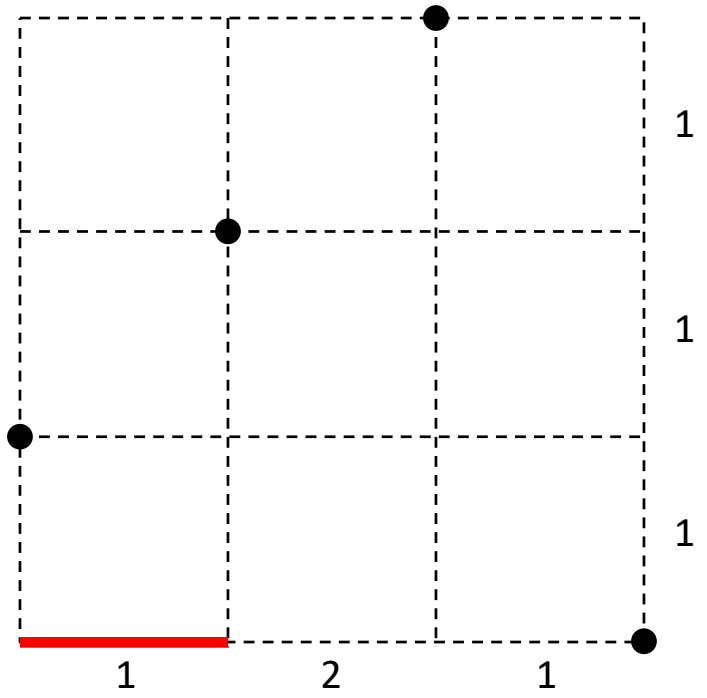
Example

- # Pins = 4
  - Position sequence = (4 1 2 3)
    - POWV = (1 2 1 1 1 1)



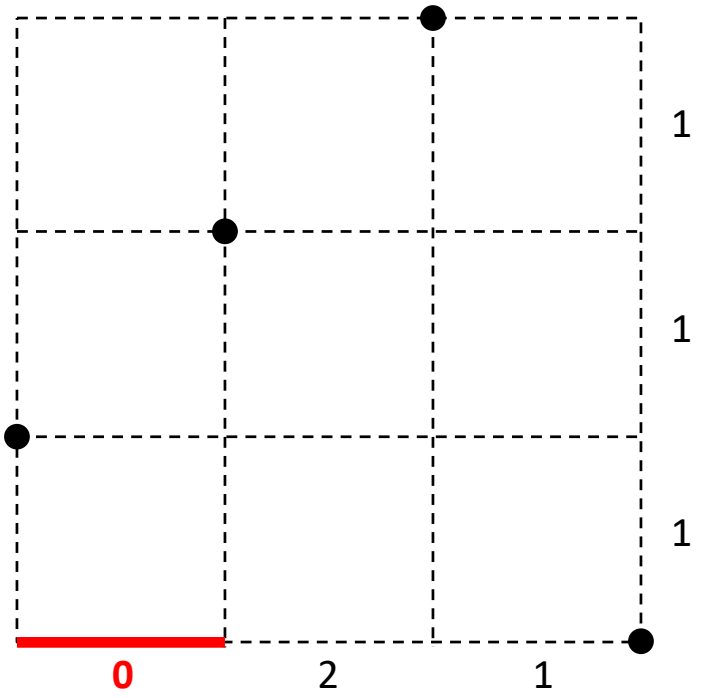
Example

- Use $e_{h,0,0}$ (later, we will also try removing $e_{h,0,0}$)



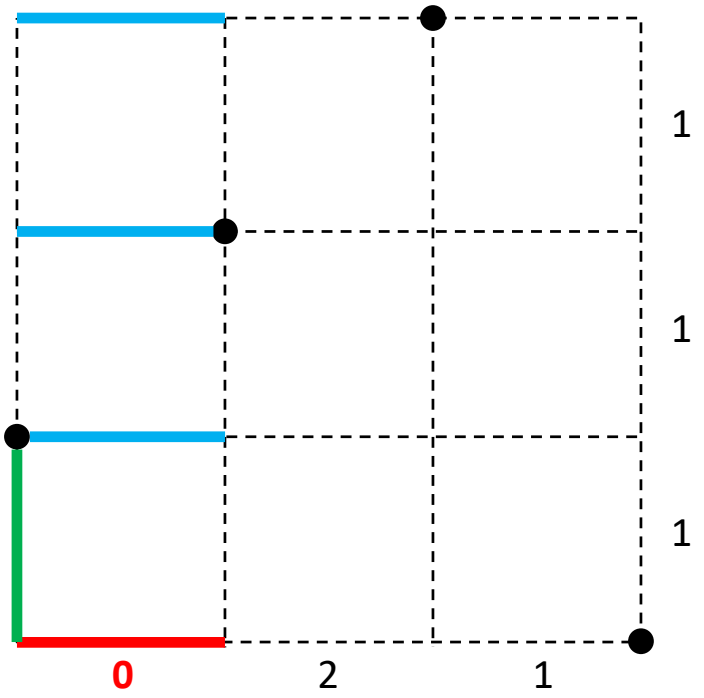
Example

- Decrease $powv(e_{h,0,0})$ by 1.



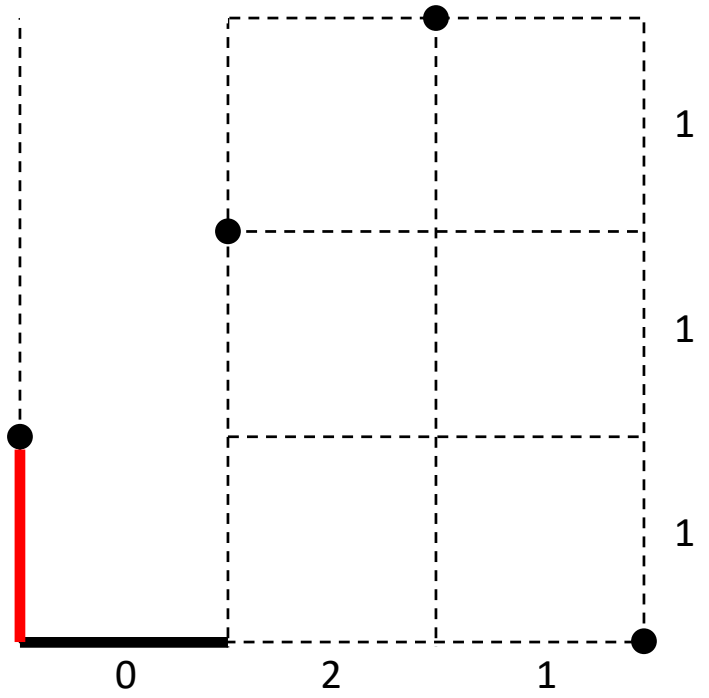
Example

- Must-use and must-remove edges



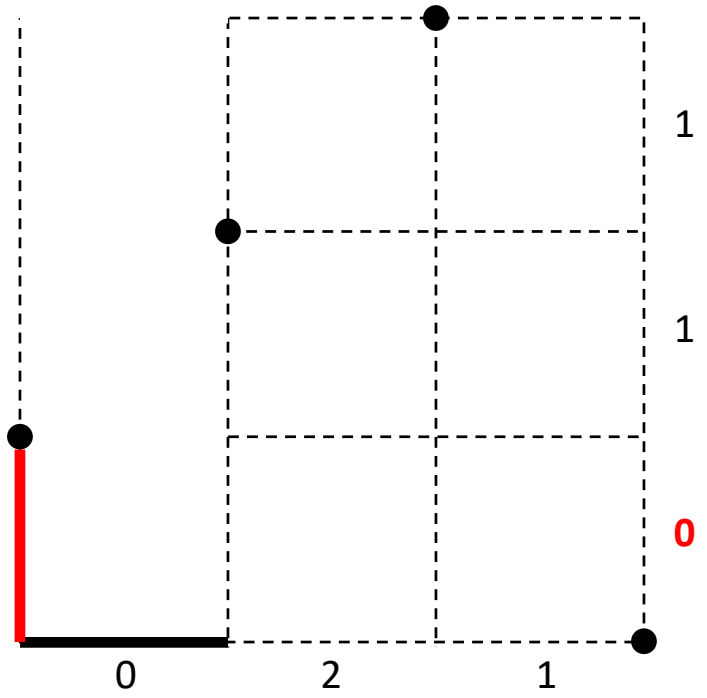
Example

- Use the must-use edges and remove the must-remove edges.



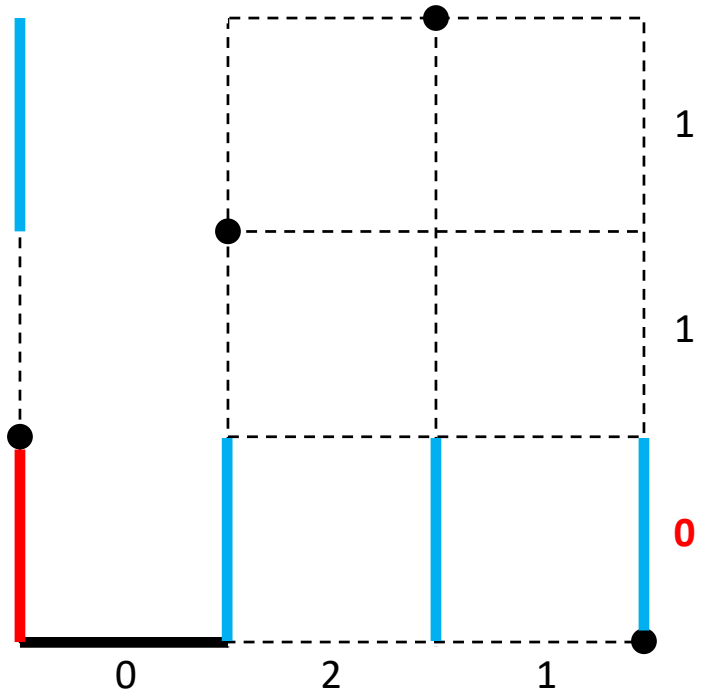
Example

- Decrease $powv(e_{v,0,1})$ by 1.



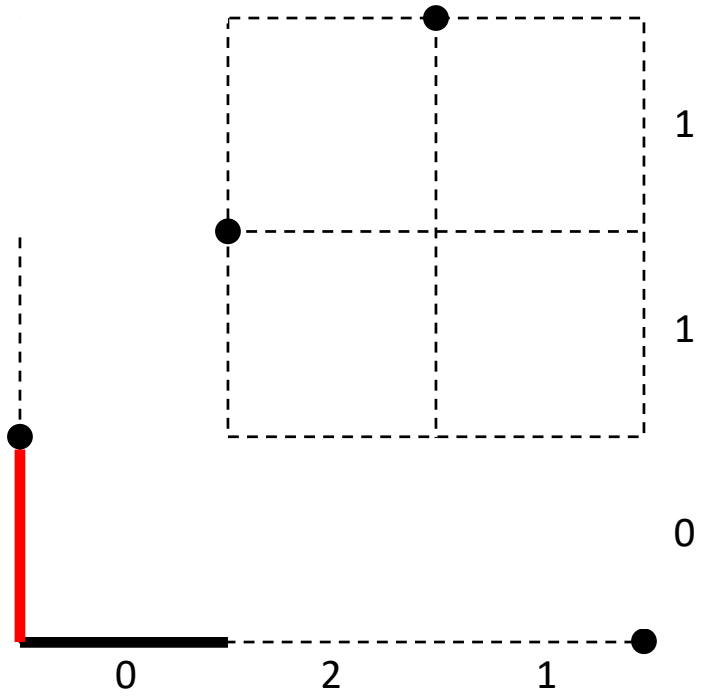
Example

- Must-remove edges



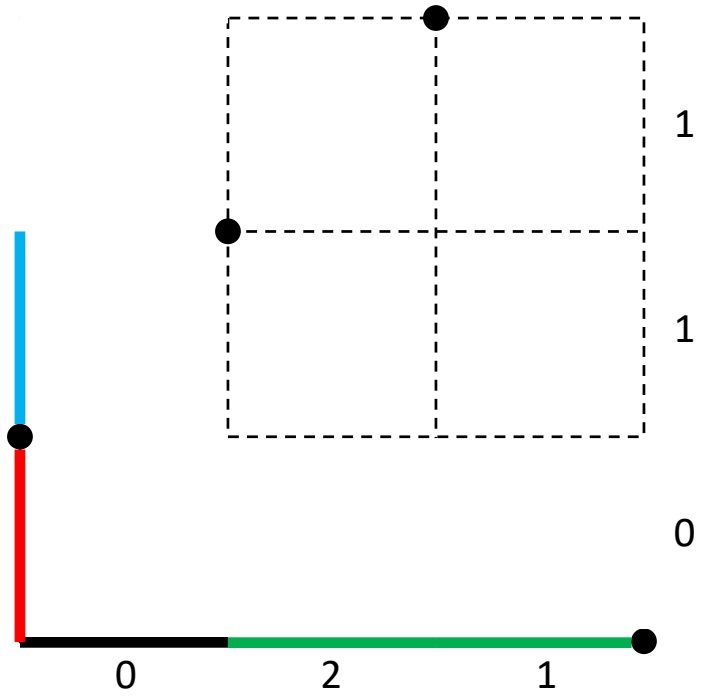
Example

- Remove the must-remove edges



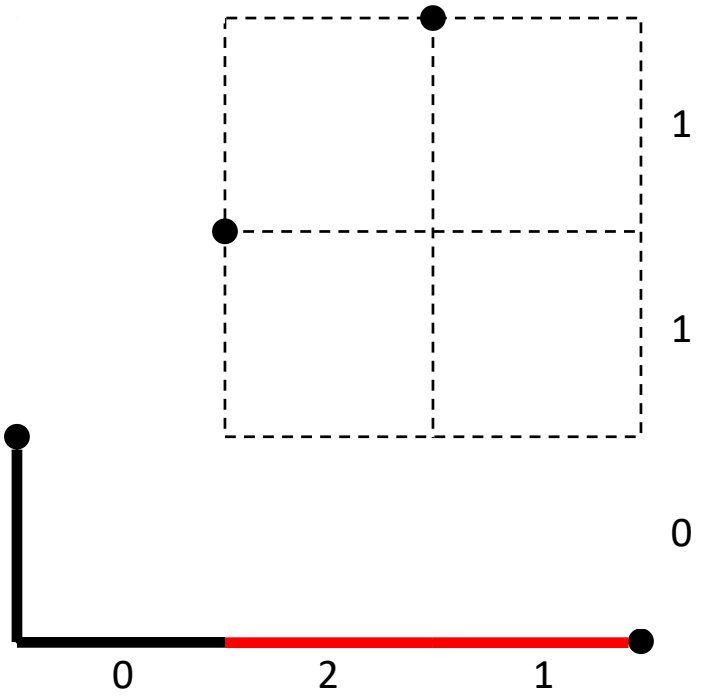
Example

- Must-use and must-remove edges



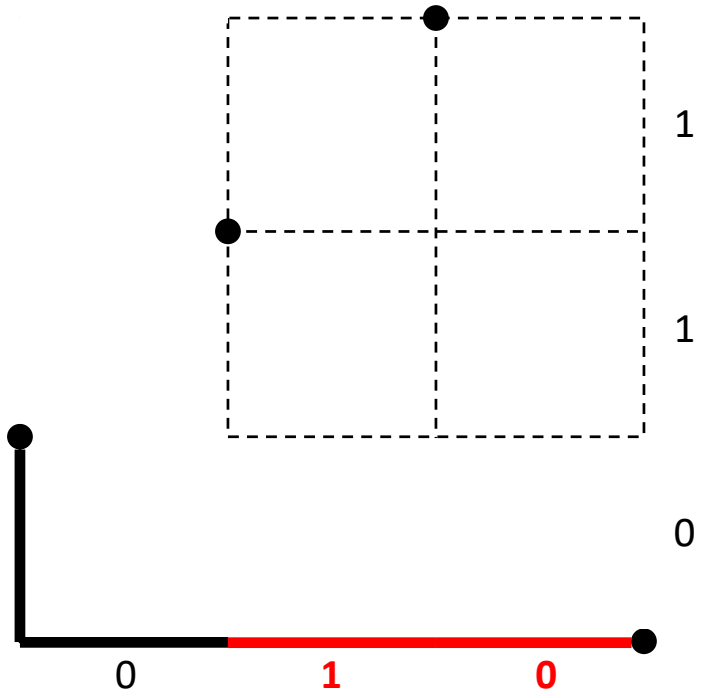
Example

- Use the must-use edges and remove the must-remove edges.



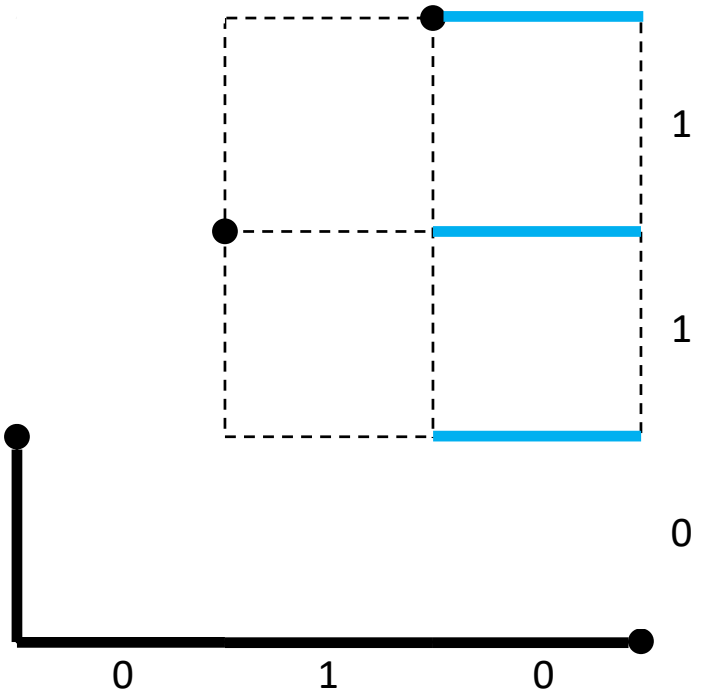
Example

- Decrease $powv(e_{h,1,0})$ and $powv(e_{h,2,0})$ by 1.



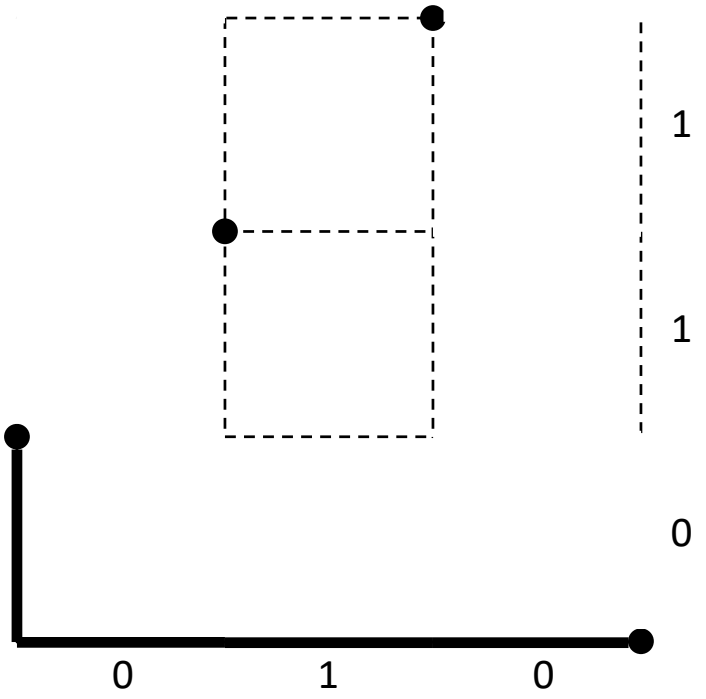
Example

- Must-remove edges



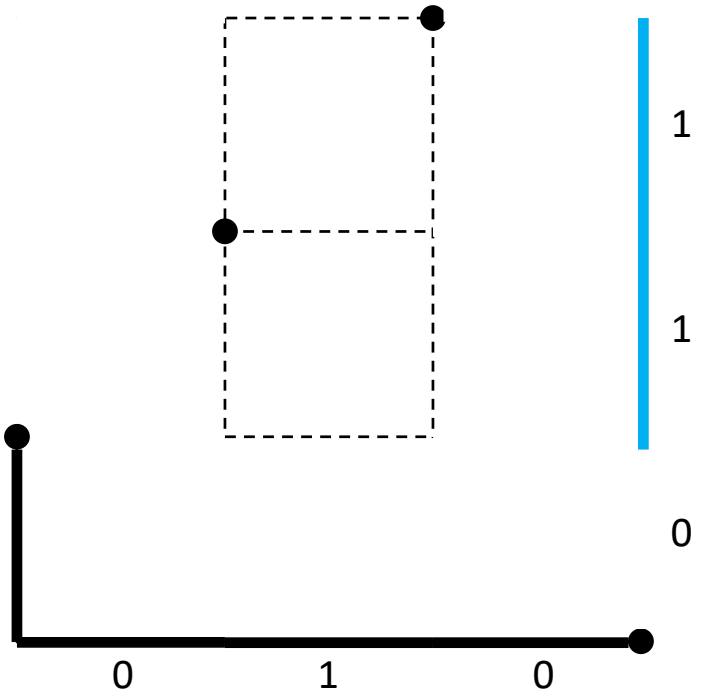
Example

- Remove the must-remove edges



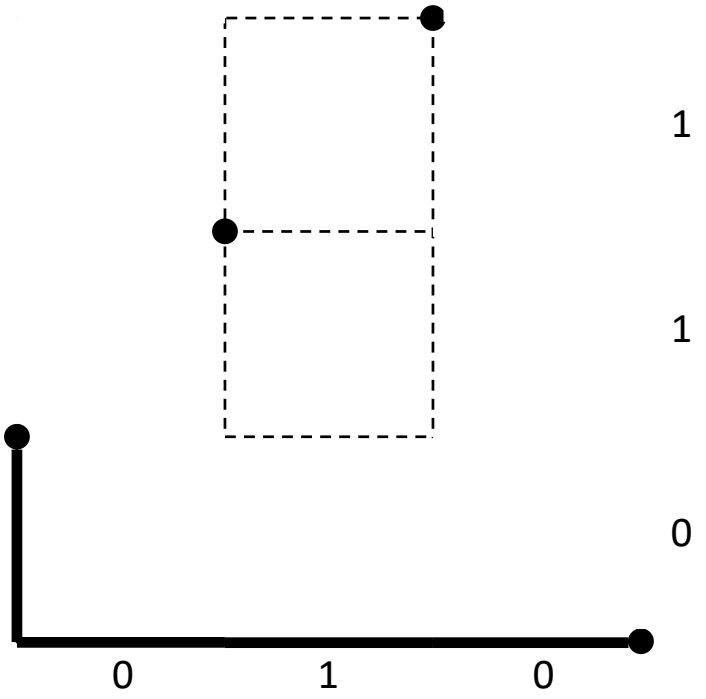
Example

- Must-remove edges



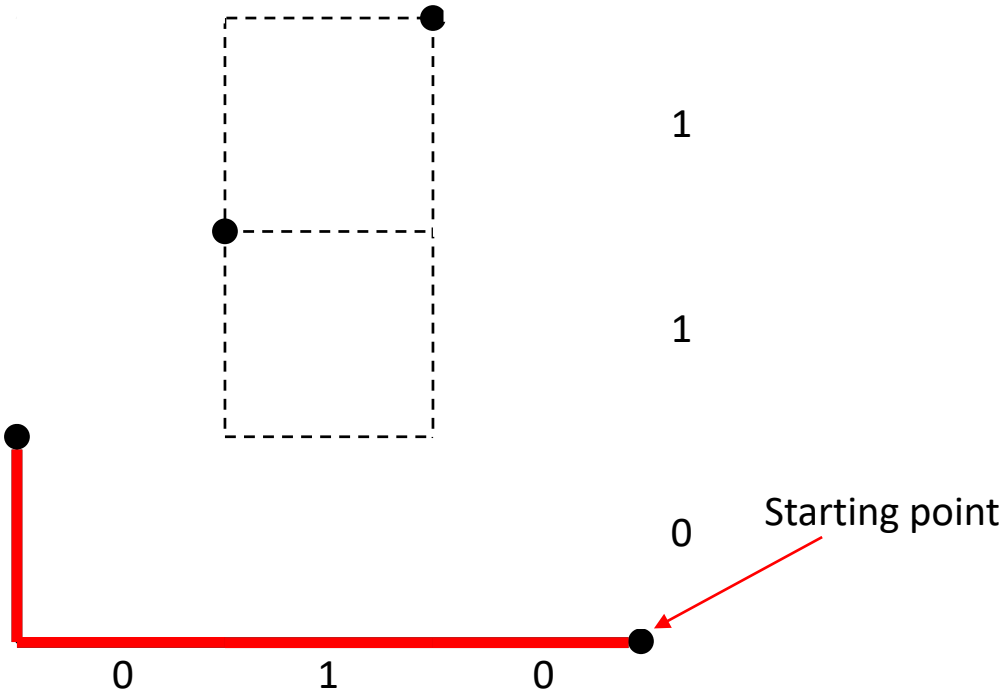
Example

- Remove the must-remove edges



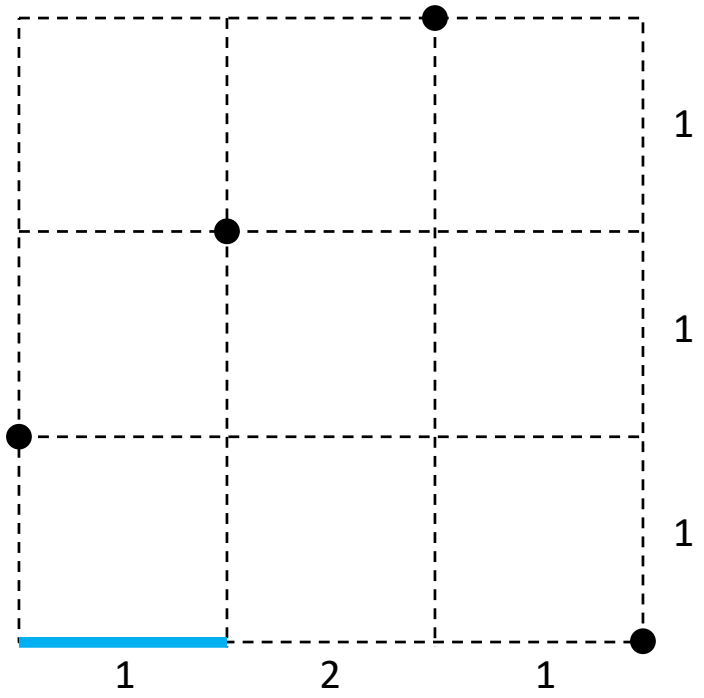
Example

- Intermediate connectivity check.
  - Check the connectivity through the used and available edges.
  - # consecutively used and removed edges = 8
  - BFS



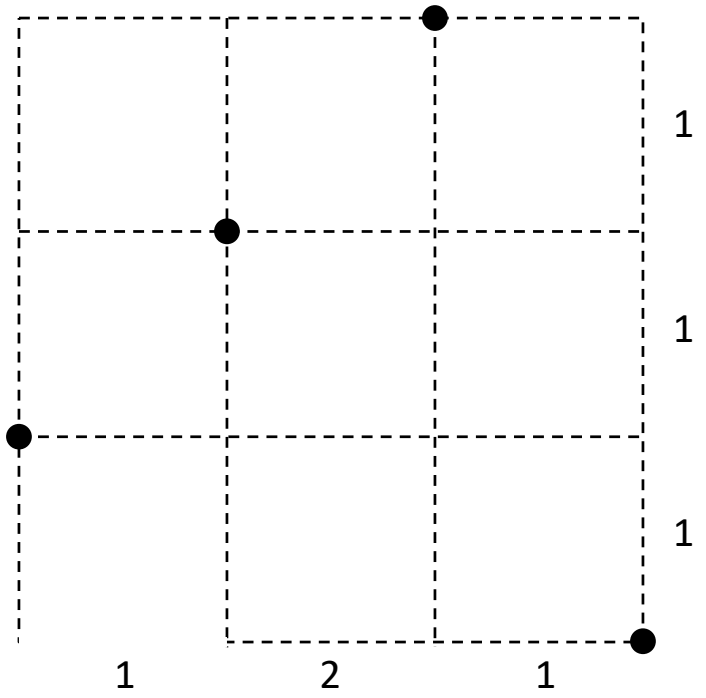
Example

- Roll-back
  - Remove $e_{h,0,0}$



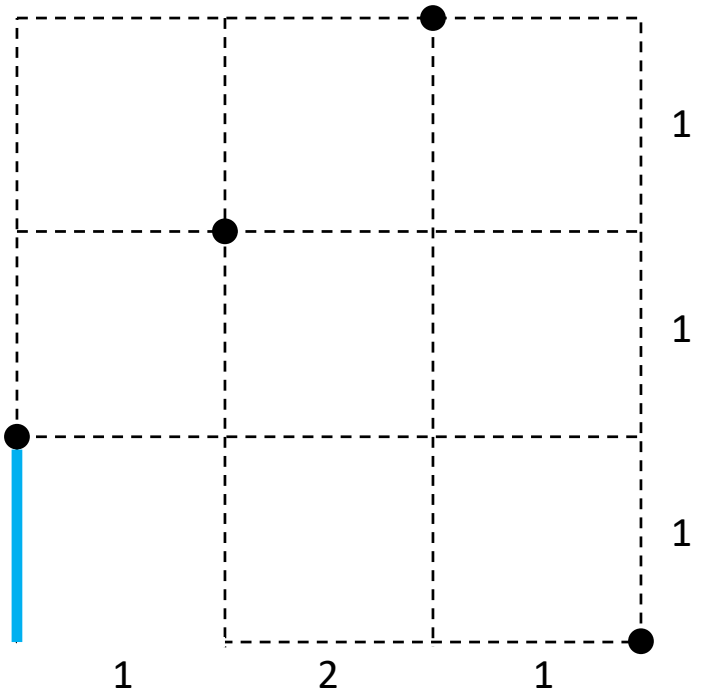
Example

- Roll-back
  - Remove $e_{h,0,0}$



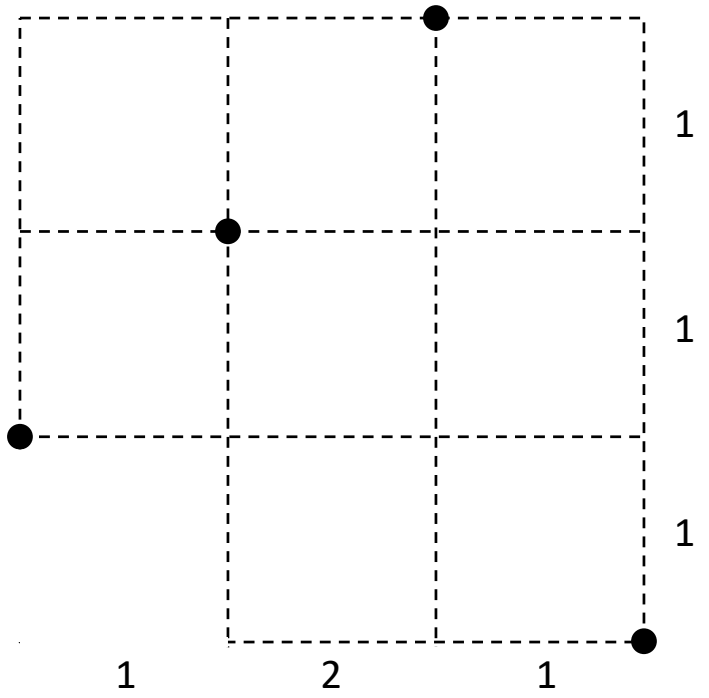
Example

- Must-remove edges



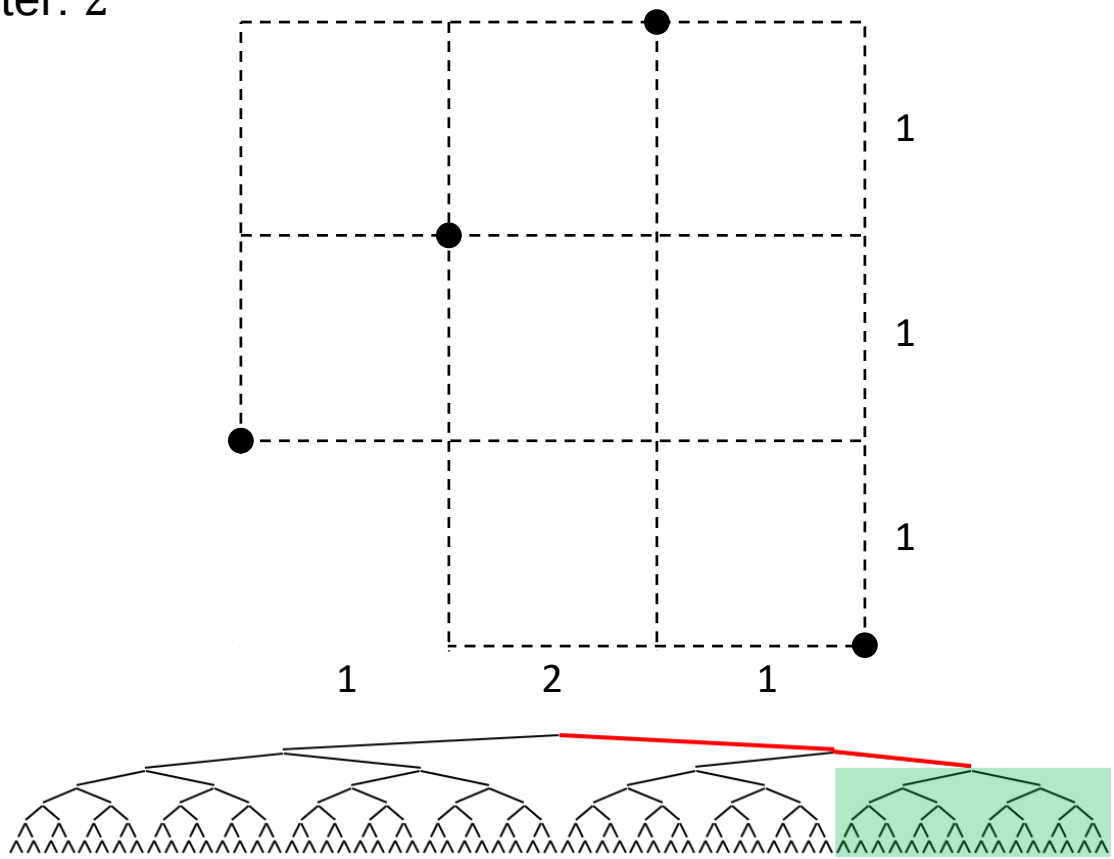
Example

- Remove the must-remove edge.



Example

- Solution space size is reduced by 4X.
  - Before: 2^{24}
  - After: 2^{22}



Simulation Results – POST DB Construction

1. Time: Construction time
2. Eff.: Construction efficiency (# POSTs found/second)
3. Size: Table size (file size)

# pins	# pin groups (n!)	# POWVs in a group			# POSTs for a POWV			Total # POSTs	Time[1]	Eff. [2]	Size[3]
		Min.	Avg.	Max.	Min.	Avg.	Max.				
2	2	1	1	1	2	2	2	4	0s	-	0
3	6	1	1	1	2	2.667	4	16	0s	80K	0
4	24	1	1.667	2	2	7.100	12	284	0.004s	81K	0
5	120	1	2.467	3	4	14.392	38	4,260	0.1s	54K	0.1MB
6	720	1	4.433	8	4	37.661	216	120,212	3.7s	32K	3.5MB
7	5,040	1	7.932	15	4	98.080	852	3,920,832	254s	15K	141MB
8	40,320	1	15.251	33	6	289.972	6,558	178,313,916	9hr	5.5K	7.7GB
9	362,880	1	30.039	79	8	929.600	52,020	10,133,050,012	1,700hr	1.7K	525GB

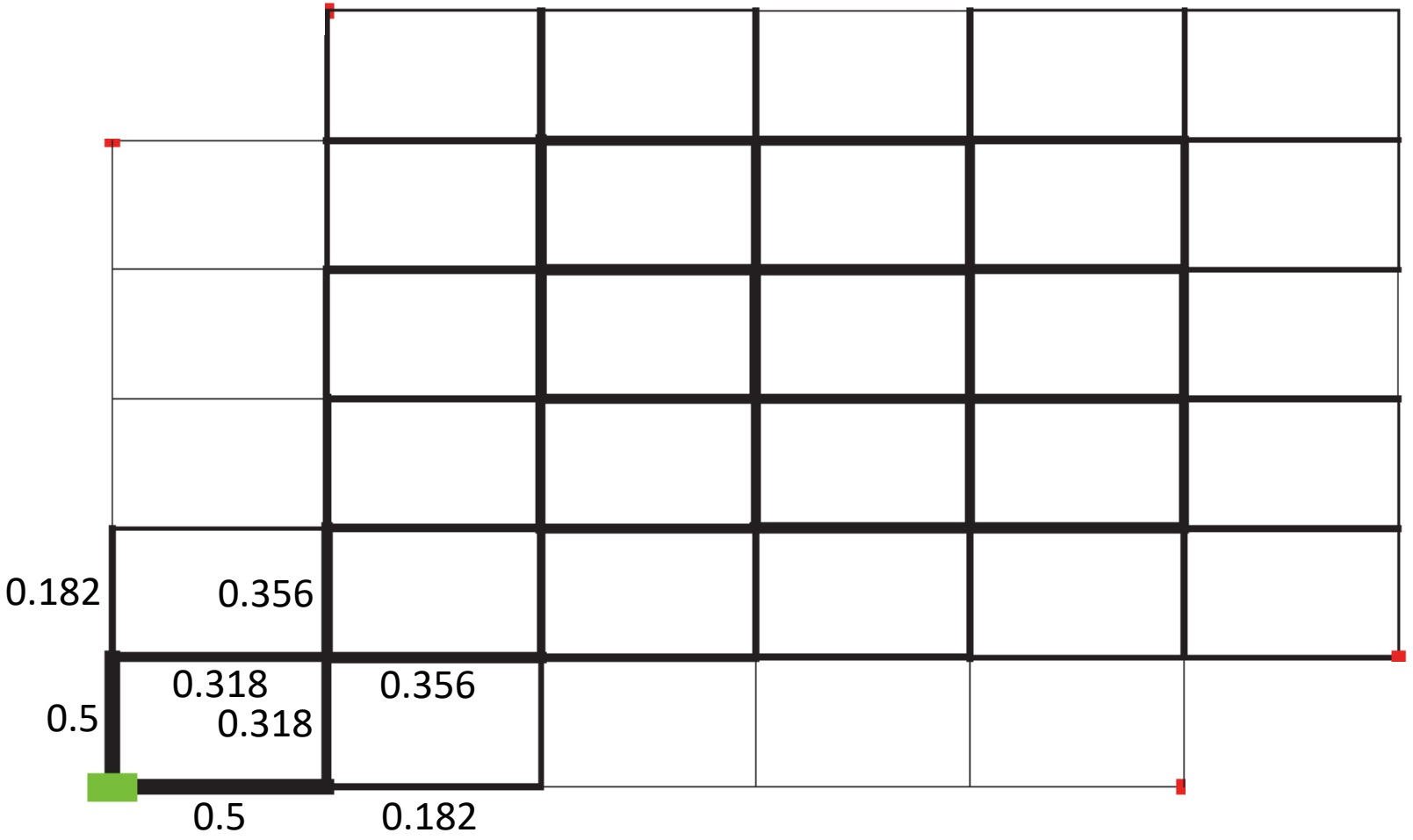
Effectiveness of the Speed-Up Techniques

- Runtime comparison (in seconds)

	with All	w/o "Zero POWV elements"	w/o must-use edges	w/o must-remove edges	w/o intermediate connectivity check
6 pins	3.72	14	3,850	22	9
	Ratio (1.00)	3.65×	1,035×	6.02×	2.46×
7 pins	254	2,837	∞	6,973	964
	Ratio (1.00)	11.17×	-	27.45×	3.80×

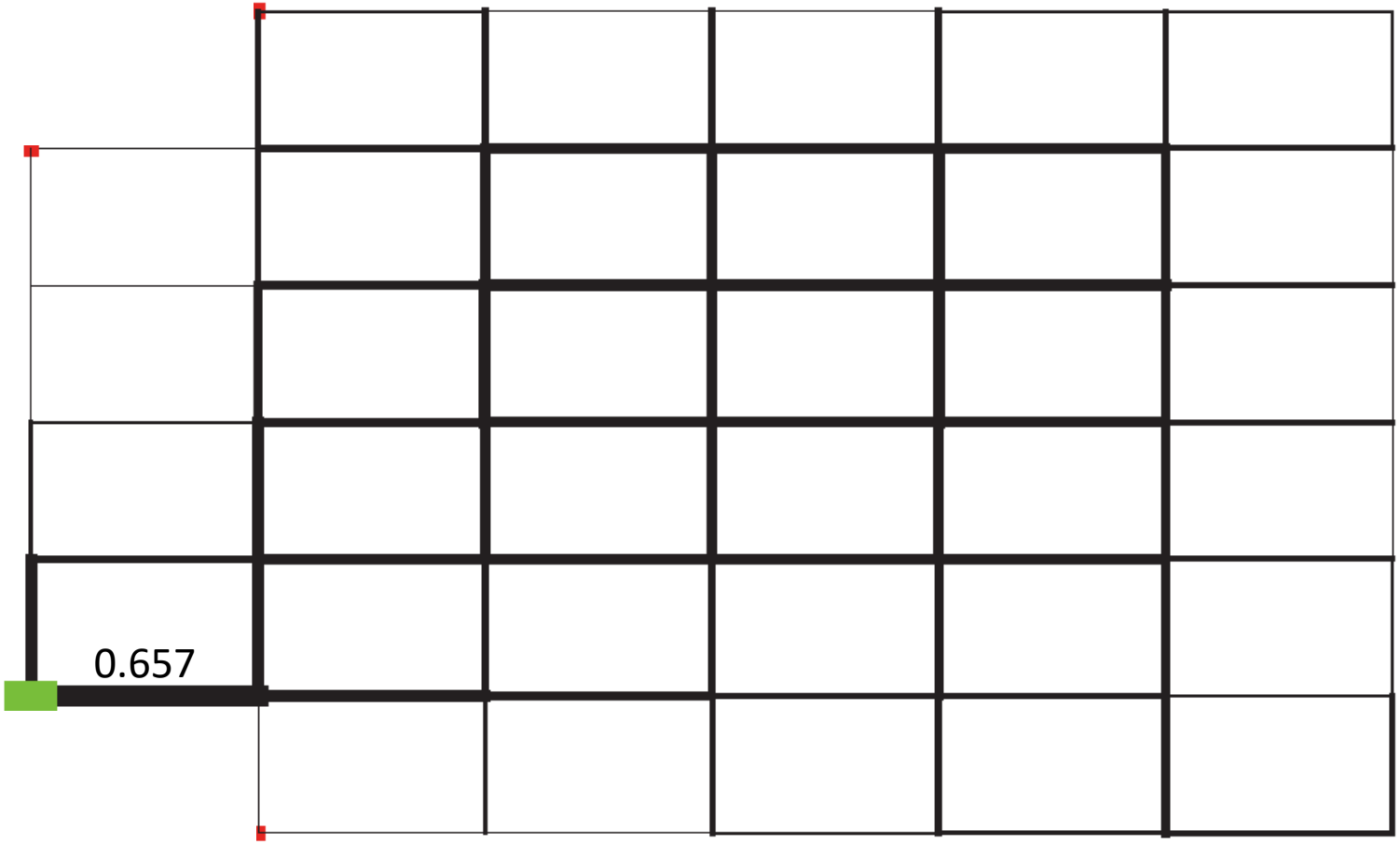
Statistics – # Edges Used in POSTs

- 1XXXXXX



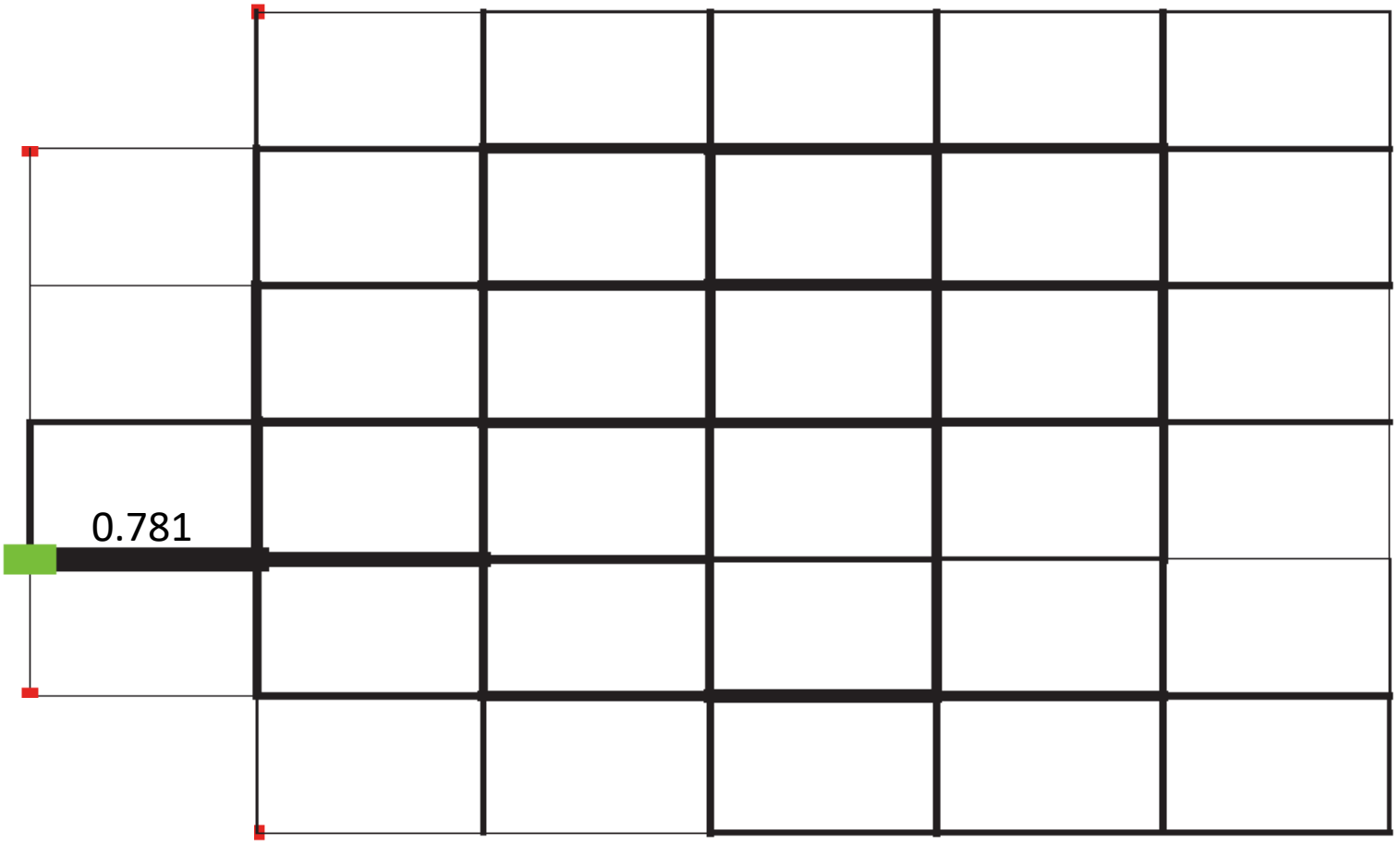
Statistics – # Edges Used in POSTs

- X1XXXXX



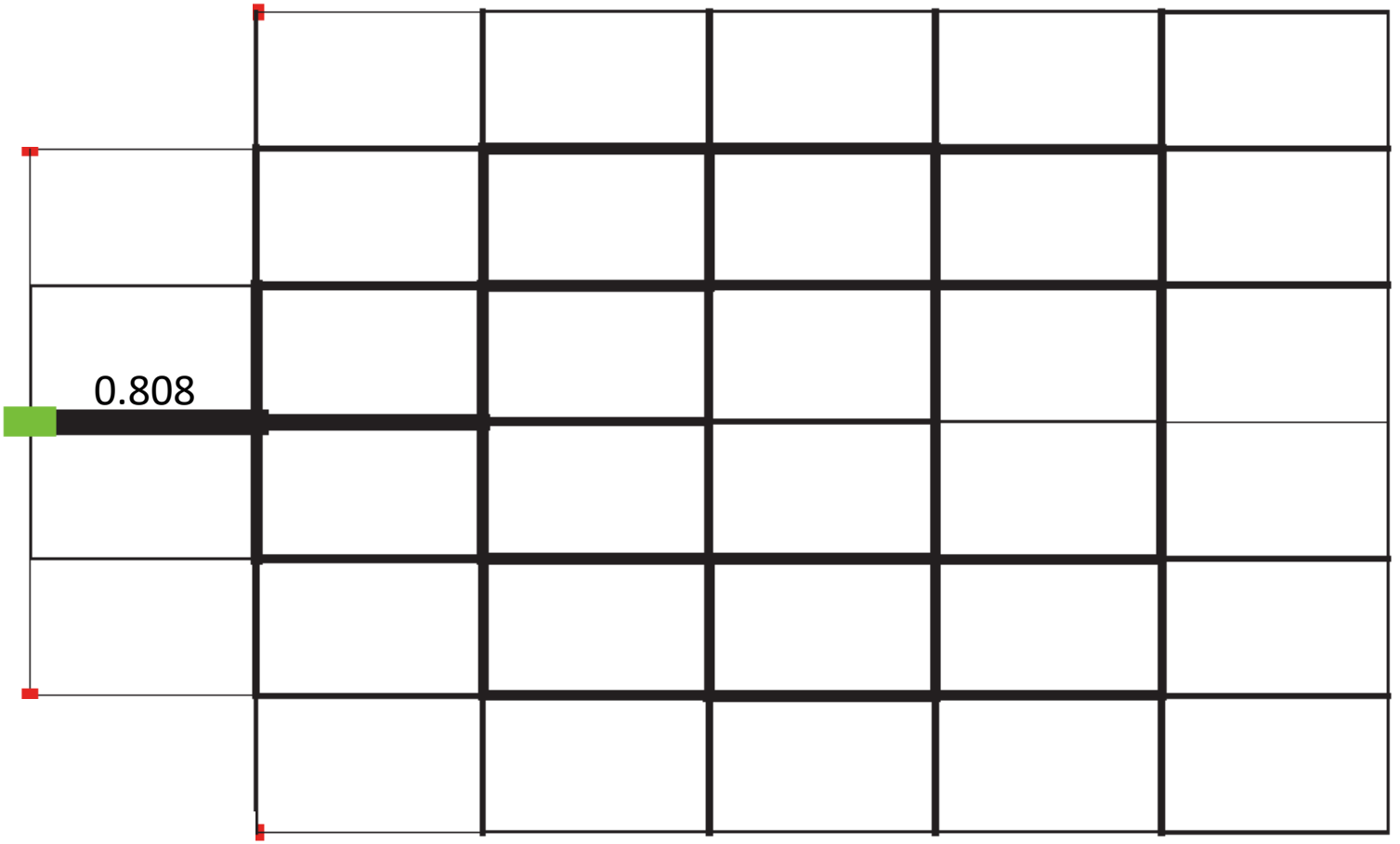
Statistics – # Edges Used in POSTs

- XX1XXXX



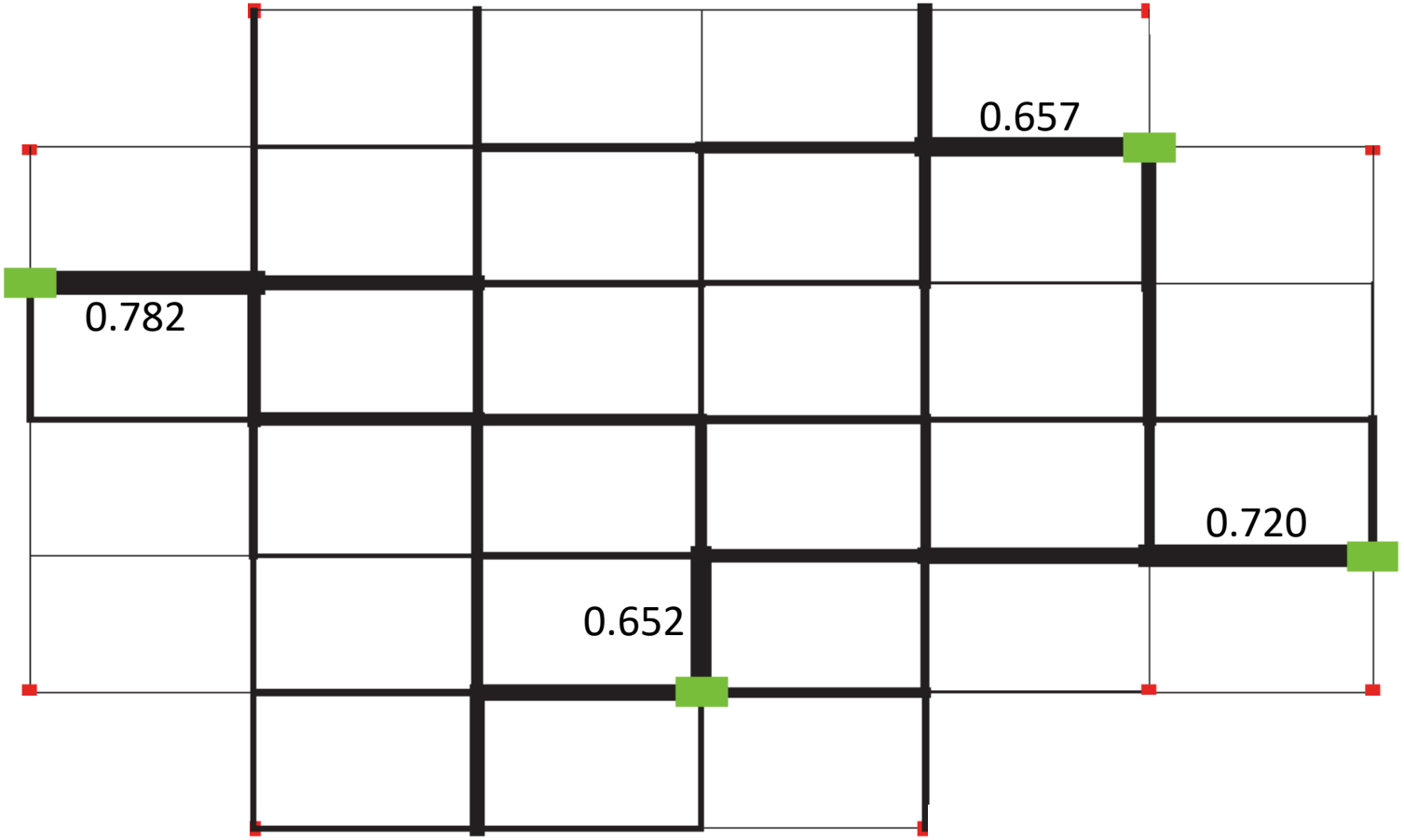
Statistics – # Edges Used in POSTs

- XXX1XXX



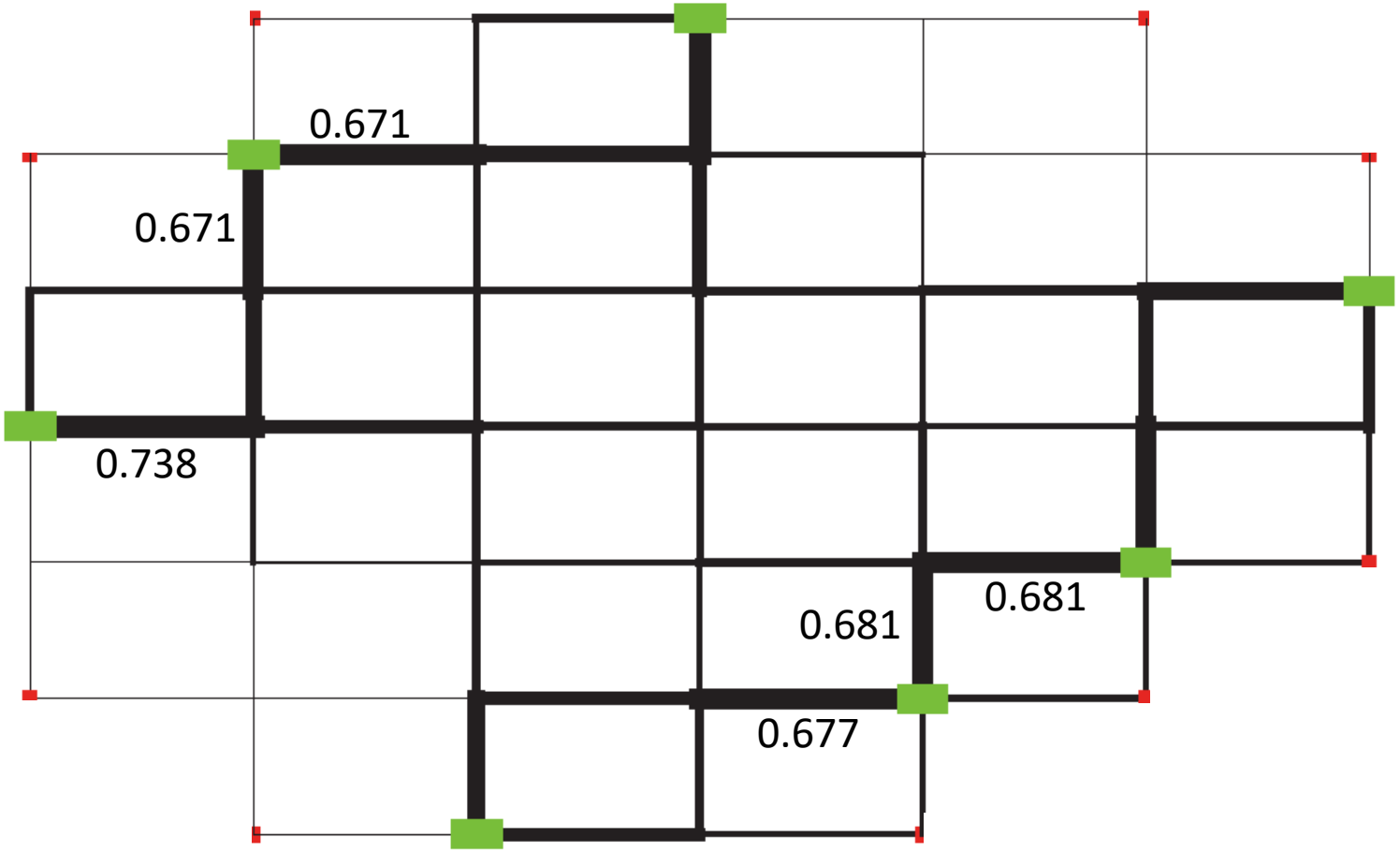
Statistics – # Edges Used in POSTs

- X47X16X



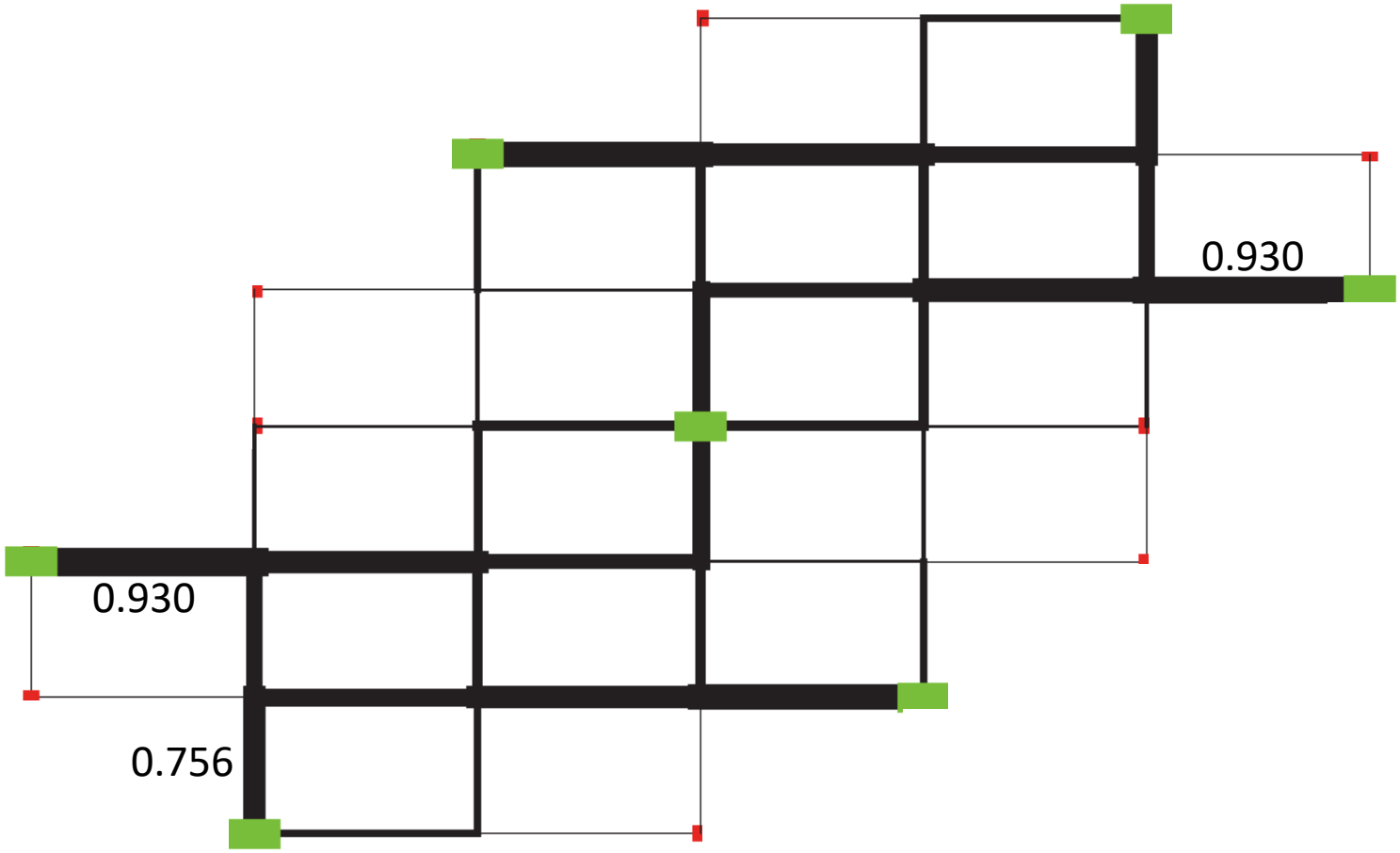
Statistics – # Edges Used in POSTs

- 3561724 (randomly chosen)



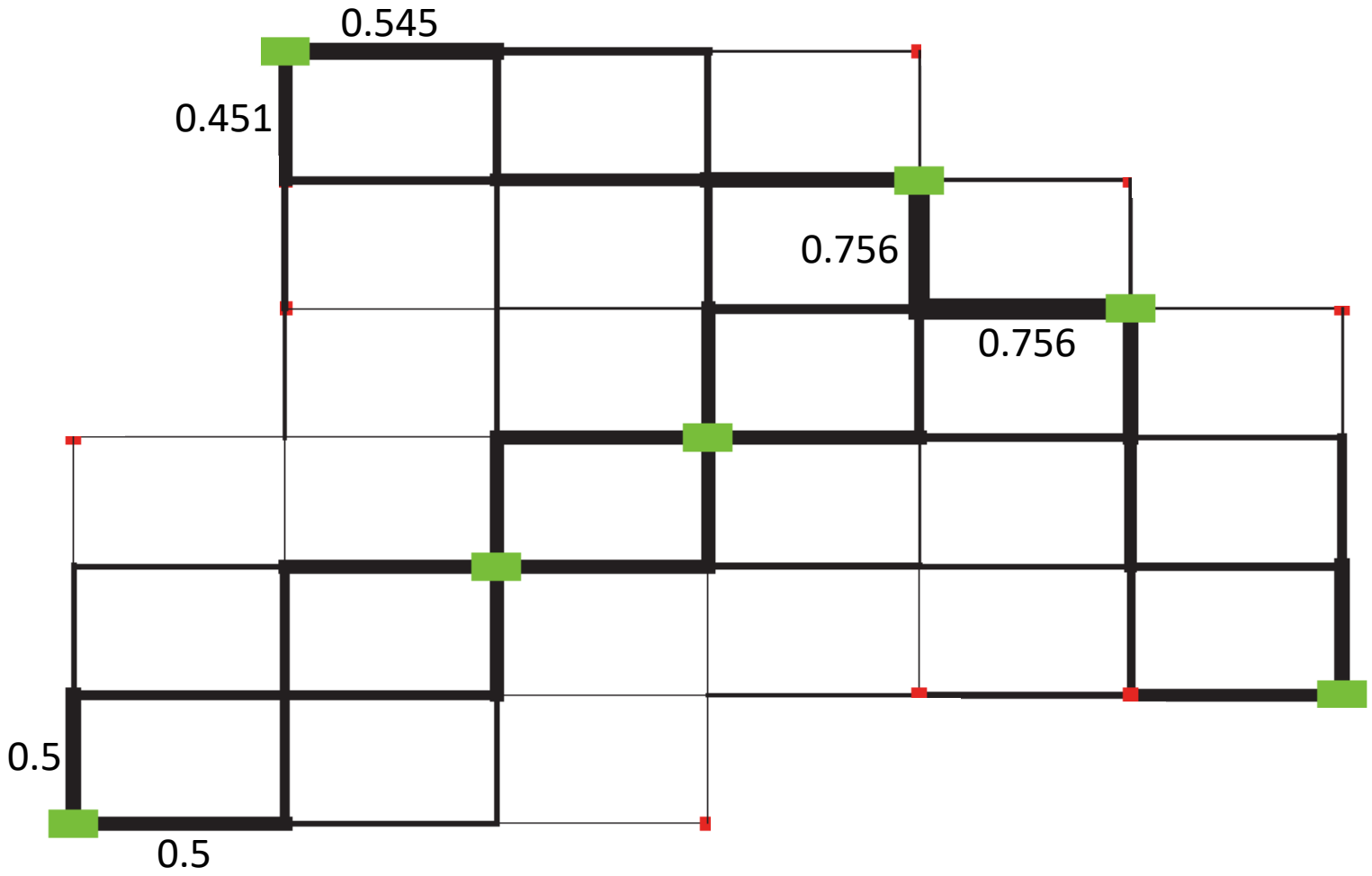
Statistics – # Edges Used in POSTs

- 2514736 (POWV having the fewest POSTs)



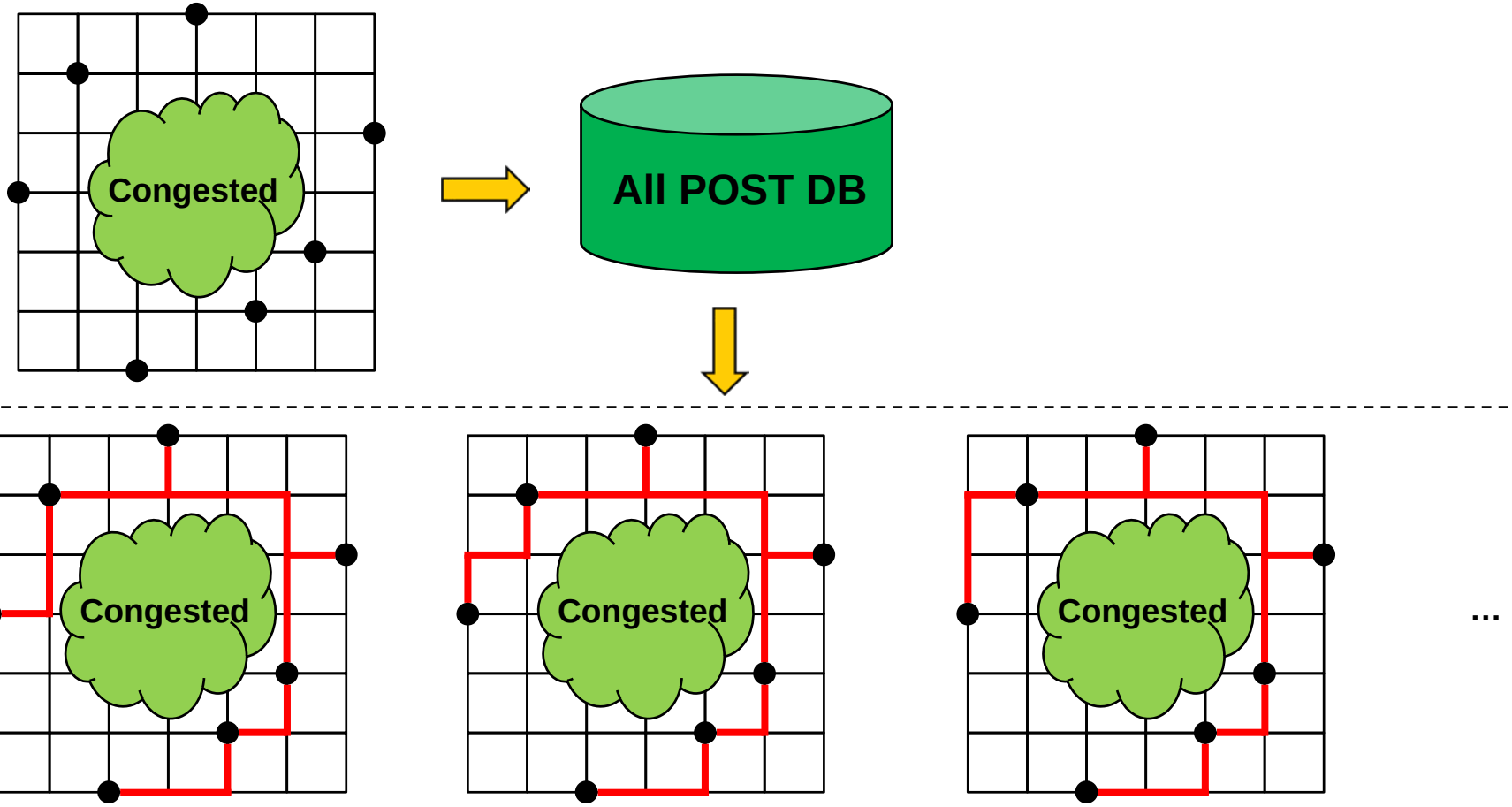
Statistics – # Edges Used in POSTs

- 1734652 (POWV having the most POSTs)



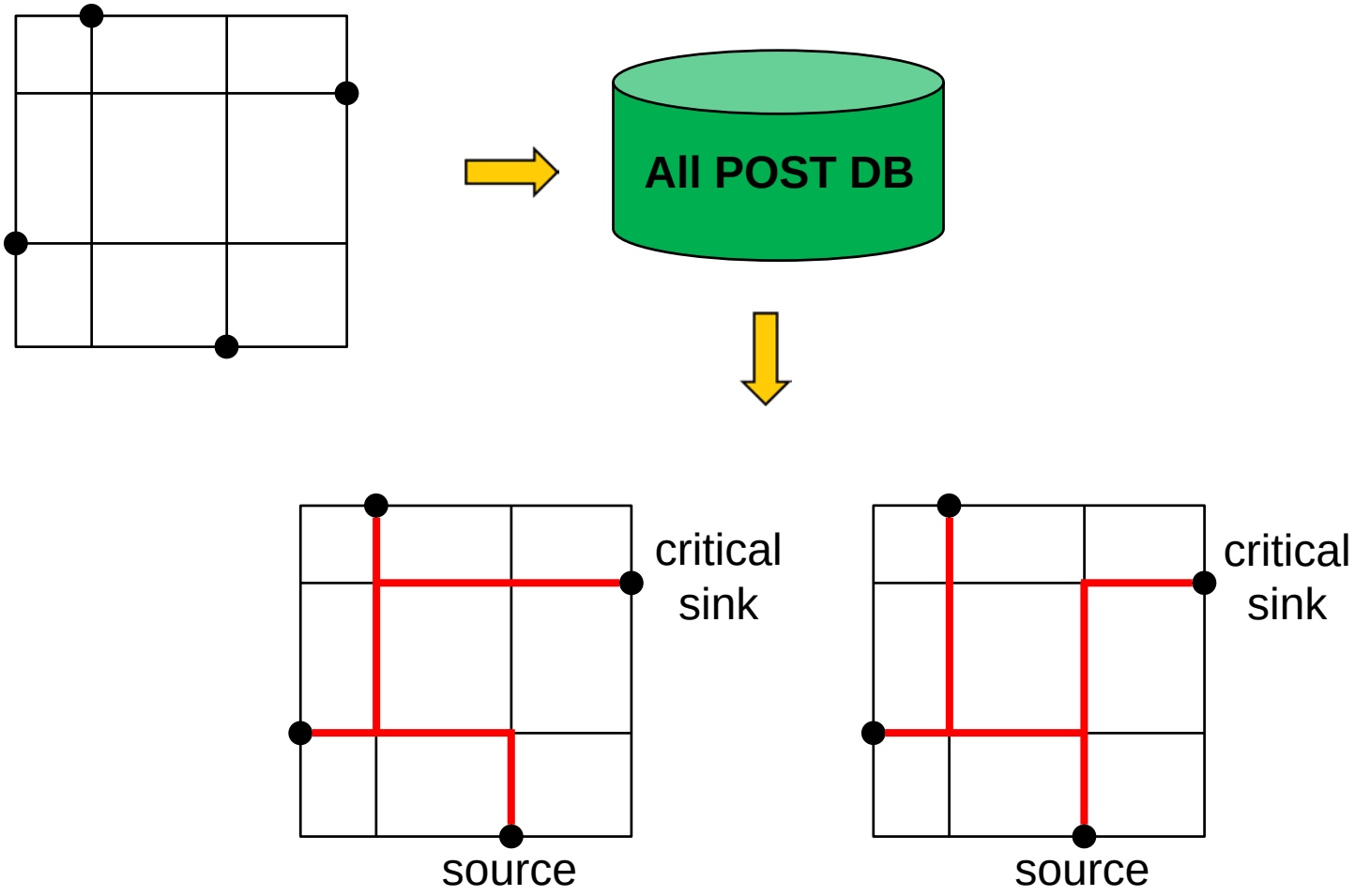
Application

- Congestion-aware RSMT generation (finding RSMTs avoiding specific edges)



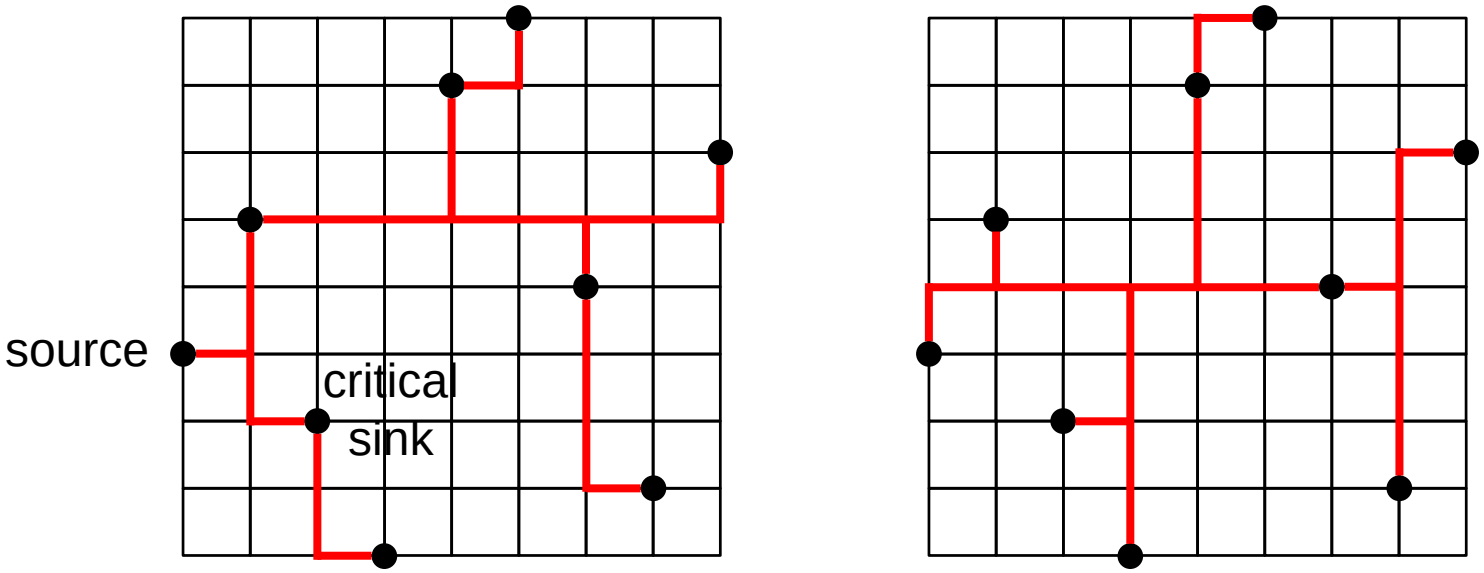
Application

- Source-to-critical-sink length minimization



Application

- Source-to-critical-sink length minimization
  - Position sequence: 483172956

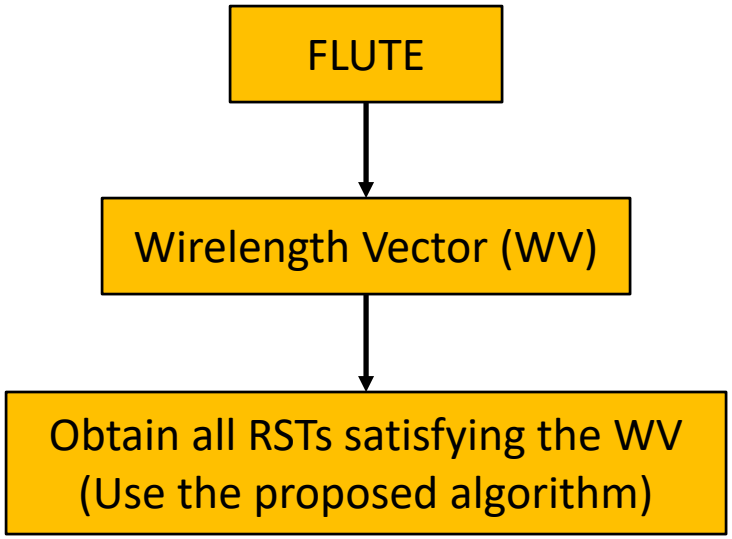


Application

- Non-preferred routing path (finding RSMTs avoiding specific edges)
- Preferred routing path (finding RSMTs using specific edges)
- Source-to-critical-sink length optimization
- Fast routing estimation (placement, routing, …)
- …

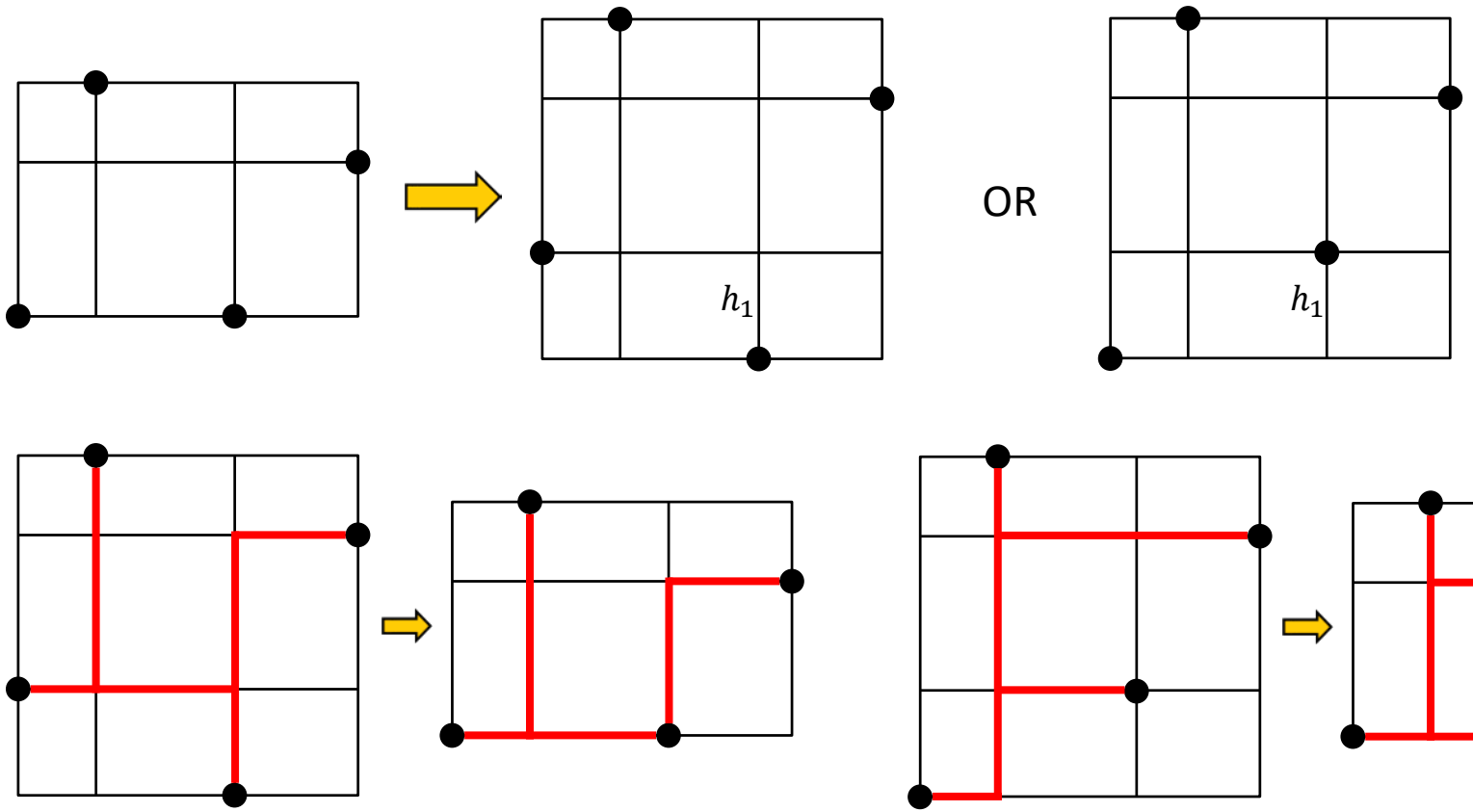
For High-Degree Nets

- FLUTE generates an RST.



For Non-Distinct Pins

- $\lim_{h_1 \to 0} L$



Conclusion

- We proposed an efficient algorithm to construct a database of all POSTs for up to nine pins.

- The DB can be used for various purposes.
  - Placement
  - Routing
  - …

- The algorithm can also be used to find RSTs for high-degree nets (more than nine pins).

Thank you

daehyun@eecs.wsu.edu

(Please email me for the POST DB)