Obstacle-Avoiding Rectilinear Steiner Minimal Tree Construction

Pei-Ci Wu, Jhih-Rong Gao and Ting-Chi Wang

Department of Computer Science

National Tsing Hua University

Outline

- Introduction
- Problem formulation
- Algorithm
- Experimental results
- Conclustion

Introduction

- Routing plays an important role in VLSI/ULSI physical design.
- Today's design often contains rectilinear obstacles, like macro cells, IP blocks, and pre-routed nets.
- By taking obstacles into consideration, obstacle-avoiding rectilinear Steiner minimal tree (OARSMT) construction becomes a very practical problem.

Previous Work

- An-OARSMan [Hu et al, ASP-DAC 2005]
- Spanning graph based method [Shen et al, ICCD 2005]
- CDCTree [Shi et al, ASP-DAC 2006]
- $O(n\log n)$ algorithm: 2-OASMT [Feng et al, ISPD 2006]

- Good wirelength performance, but long runtime
- Very efficient even in large cases but get worse wirelength

Outline

- Introduction
- Problem formulation
- Algorithm
- Experimental results
- Conclustion

Problem formulation

- Given: a set of terminals and a set of rectangular obstacles
- Goal: a rectilinear Steiner minimum tree which connects all terminals together but does not intersect any obstacle
  - wirelength $\rightarrow$ as small as possible
  - running time $\rightarrow$ efficient

Outline

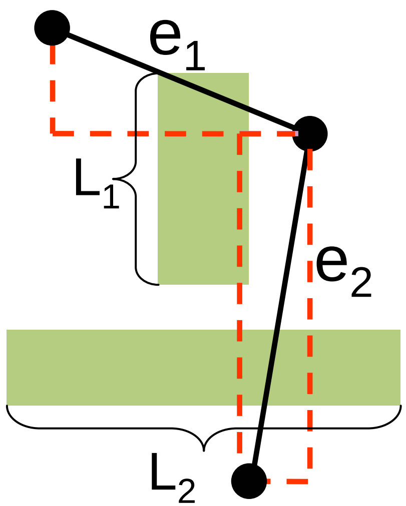
- Introduction
- Problem formulation
- Algorithm
- Experimental results
- Conclustion

Algorithm

- Step 1: Partition terminals into a set of sub-trees
- Step 2: Construct the spanning graph
- Step 3: Merge the sub-trees using the ant colony optimization (ACO) based algorithm
- Step 4: Rectilinearization and refinement

Step 1: Partitioning (1)

- In a complete graph, construct a minimal spanning tree (MST) to connect all the terminals
  - dist(a, b)=Manhattan_distance(a, b)+ obstacle_penalty(a, b)
  - obstacle_penalty=length of the side of intersected obstacle

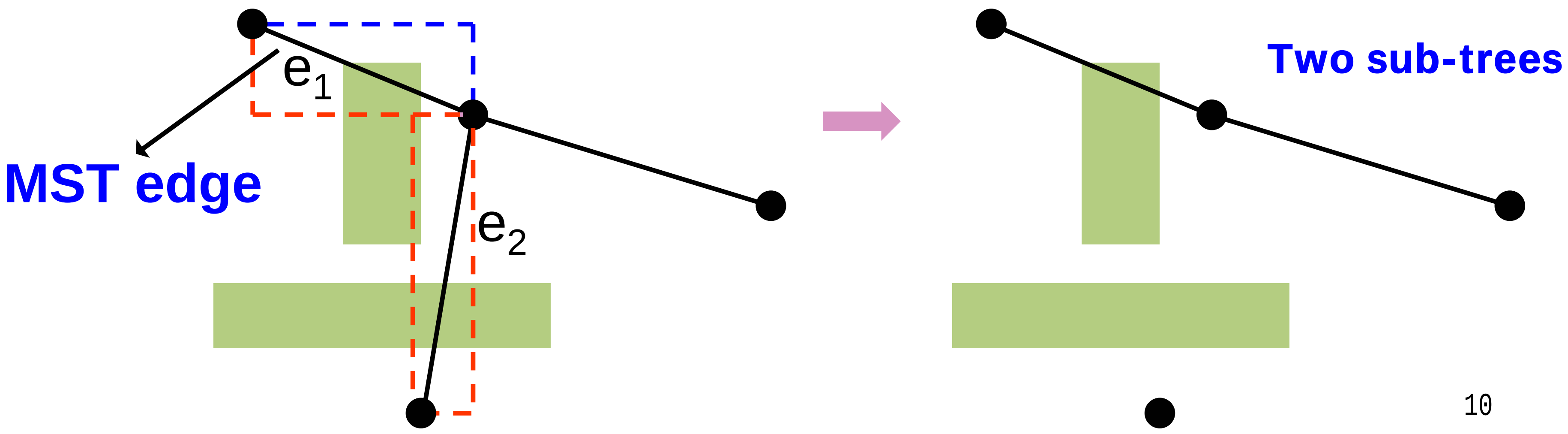


obstacle_penalty of e1 = L_1

obstacle_penalty of e2 = L_2

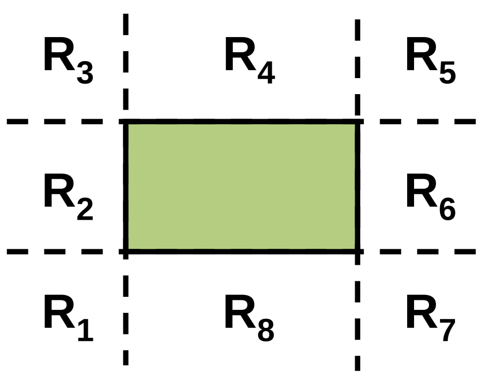
Step 1: Partitioning (2)

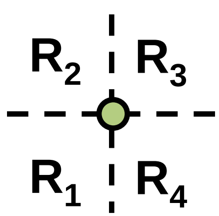
- Remove edges whose L-shaped segments (upper/lower) both intersect obstacles.

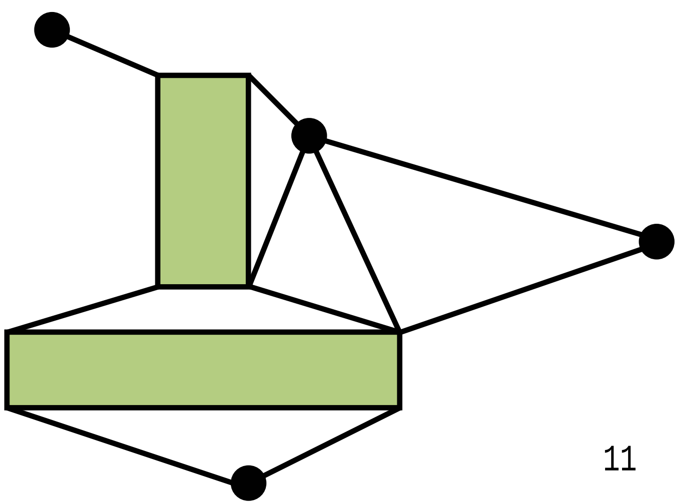


Step 2: Spanning graph

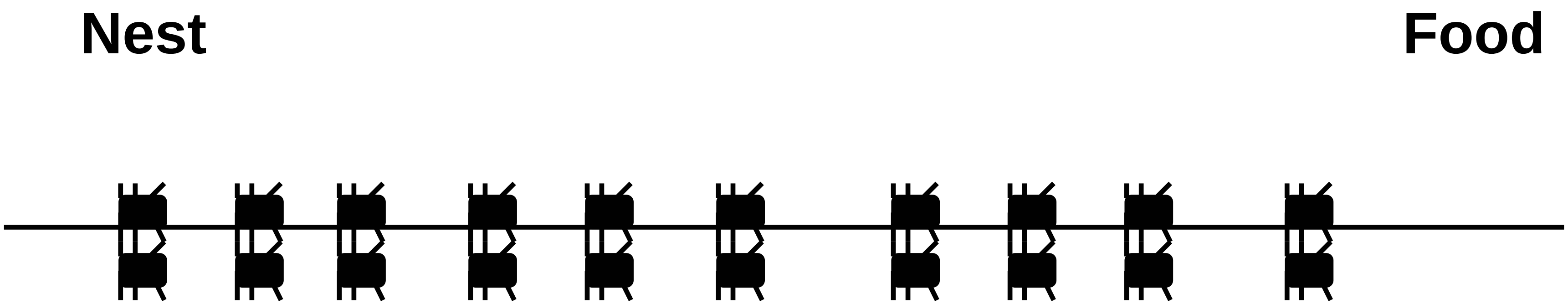
- $O(n \log n)$ algorithm [Shen et al, ICCD 2005]
- The size is proportional to the number of terminals plus obstacles
- Connect every vertex (terminals/corners of each obstacle) to the nearest vertices in its four directions, upper-right, upper-left, lower-right and lower-left.



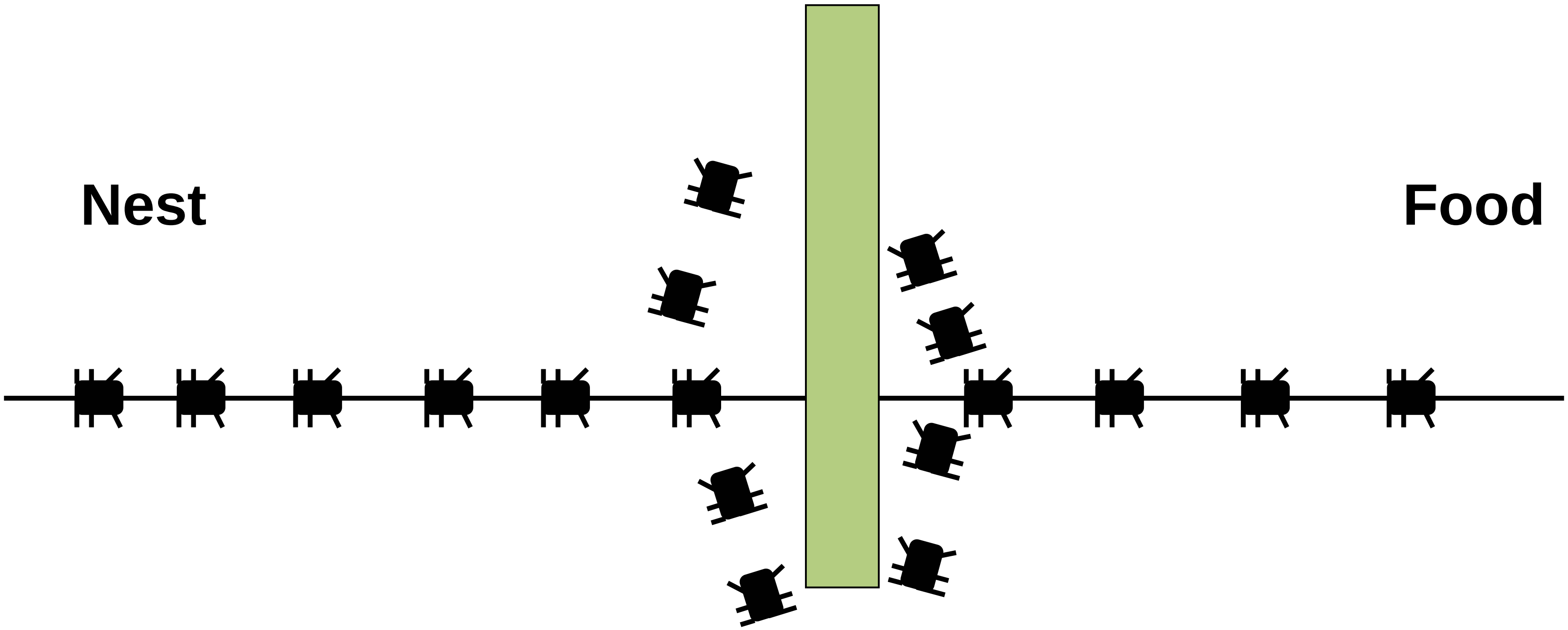




Idea of Ant Colony Optimization (ACO)

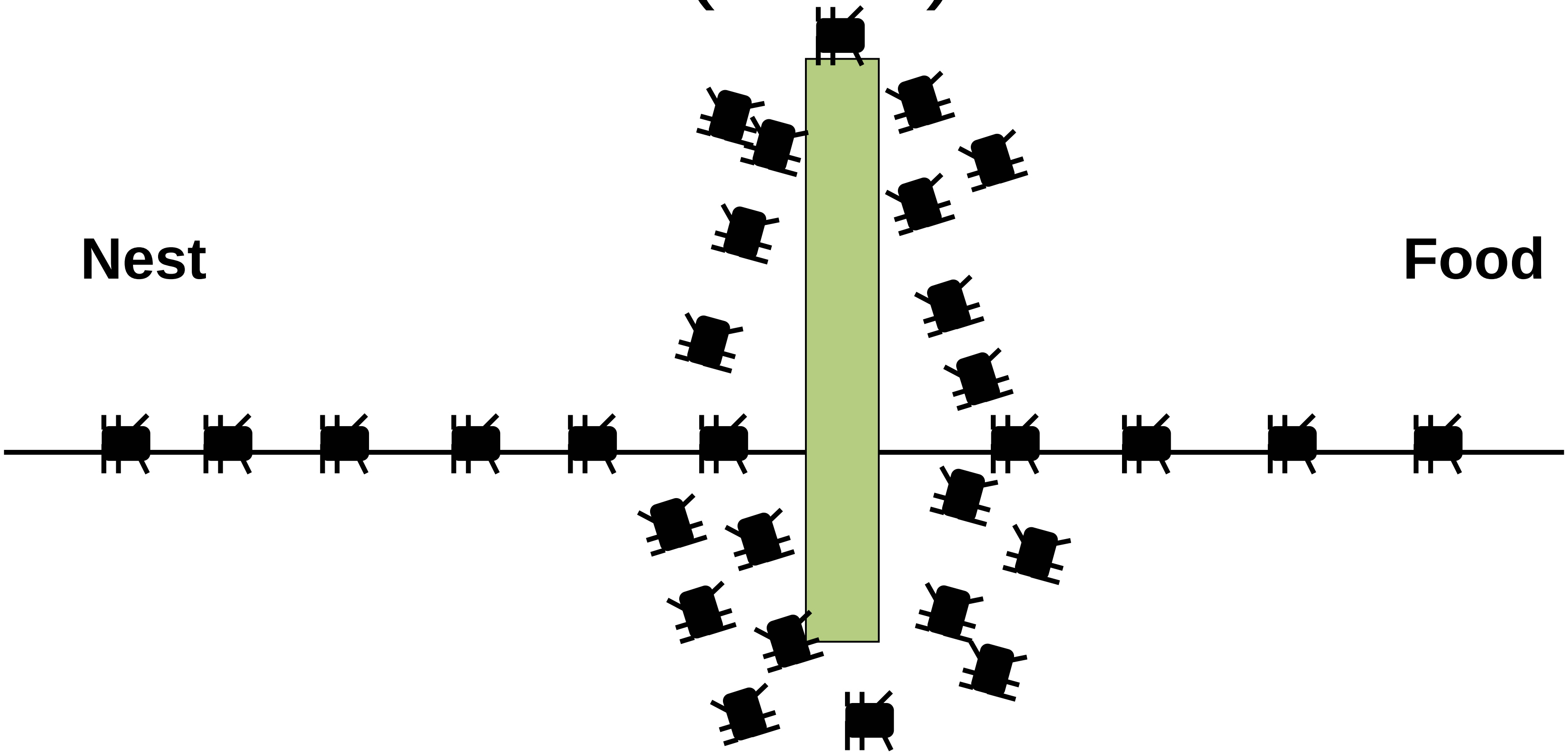


Idea of Ant Colony Optimization (ACO)



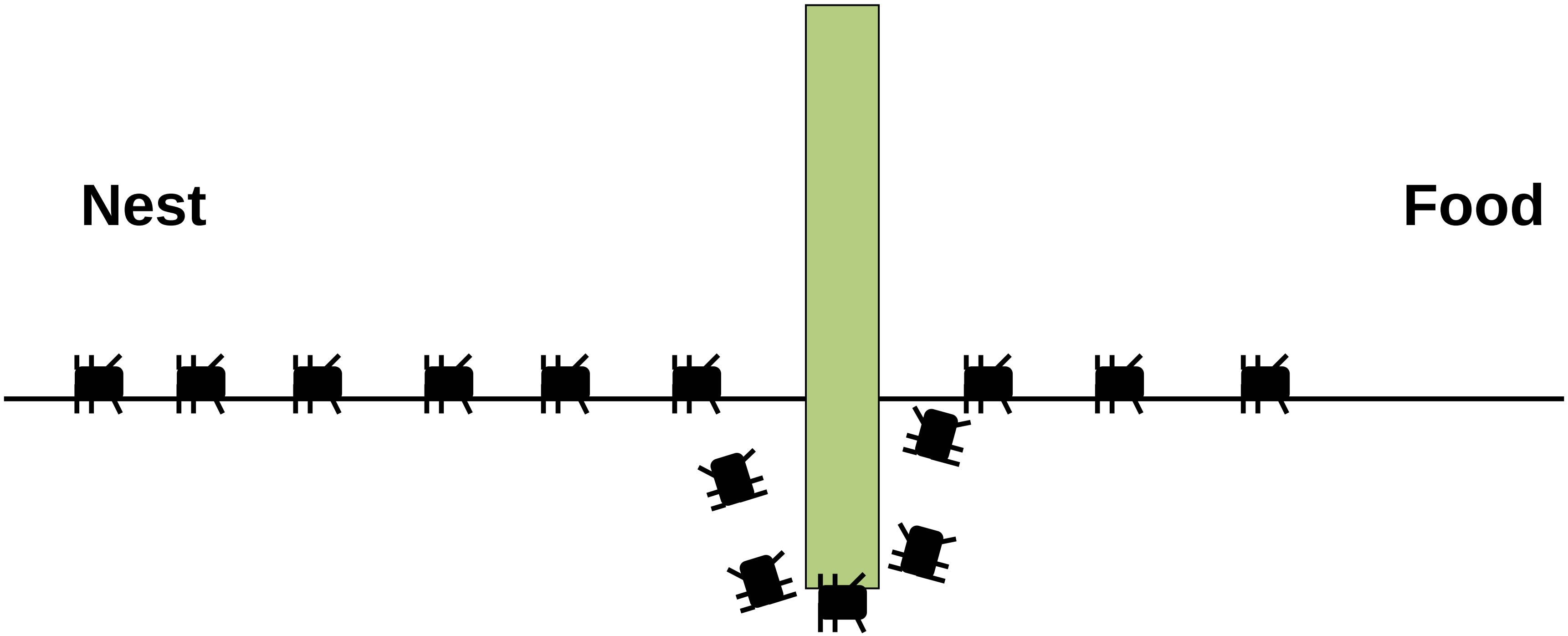
An obstacle has block the path!
Ants would choose whether to turn left or right with equal probability.

Idea of Ant Colony Optimization (ACO)



Ants leave pheromone in the edges just passed. Pheromone is deposited more quickly on the shorter path.

Idea of Ant Colony Optimization (ACO)



All ants have chosen the shorter path.

Step 3: Merge the sub-trees

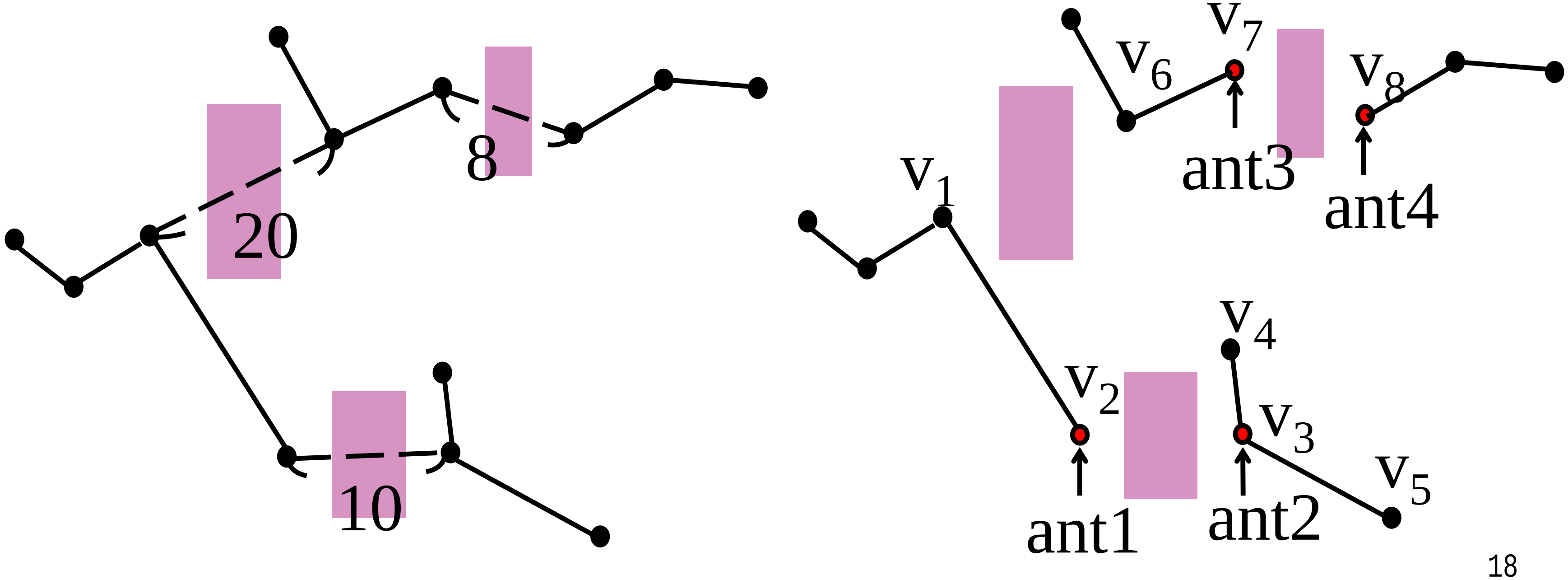
- Goal: merge the sub-trees to obtain an OASMT
  - The wirelength of edges used to merge sub-trees is as small as possible
- Modified ant-colony optimization based algorithm
  - Applied on the spanning graph
- Assume sub-trees are already passed by ants

Modified ACO based algorithm

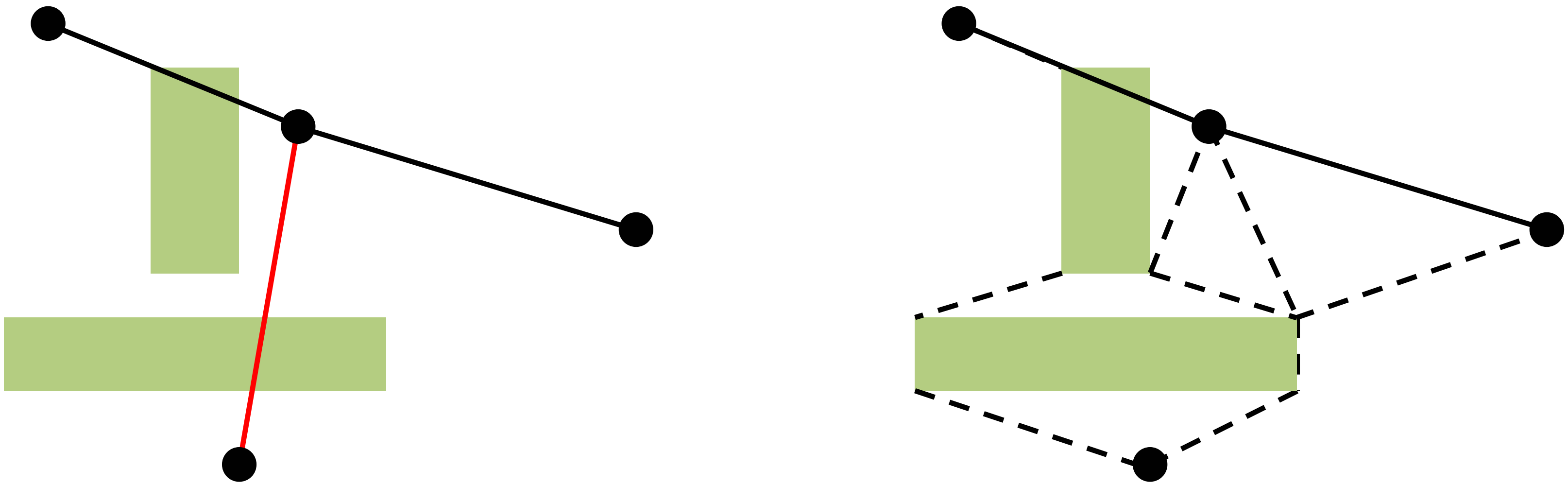
- Place an ant for each sub-tree in the beginning
- Ant selects the next wanted vertex by some user defined rules
  - trail intensity
    - Like pheromone
    - evaporate in a constant rate
  - desirability
    - Choose a path which can connect other ants as soon as possible
- When ant *A* meets ant *B*, ant *A* dies
  - Connecting path traversed by *A* and path traversed by *B*
- Only one ant left $\rightarrow$ obtain an OASMT
- Multiple iterations $\rightarrow$ get the best OASMT among all iterations

Place ants for sub-trees

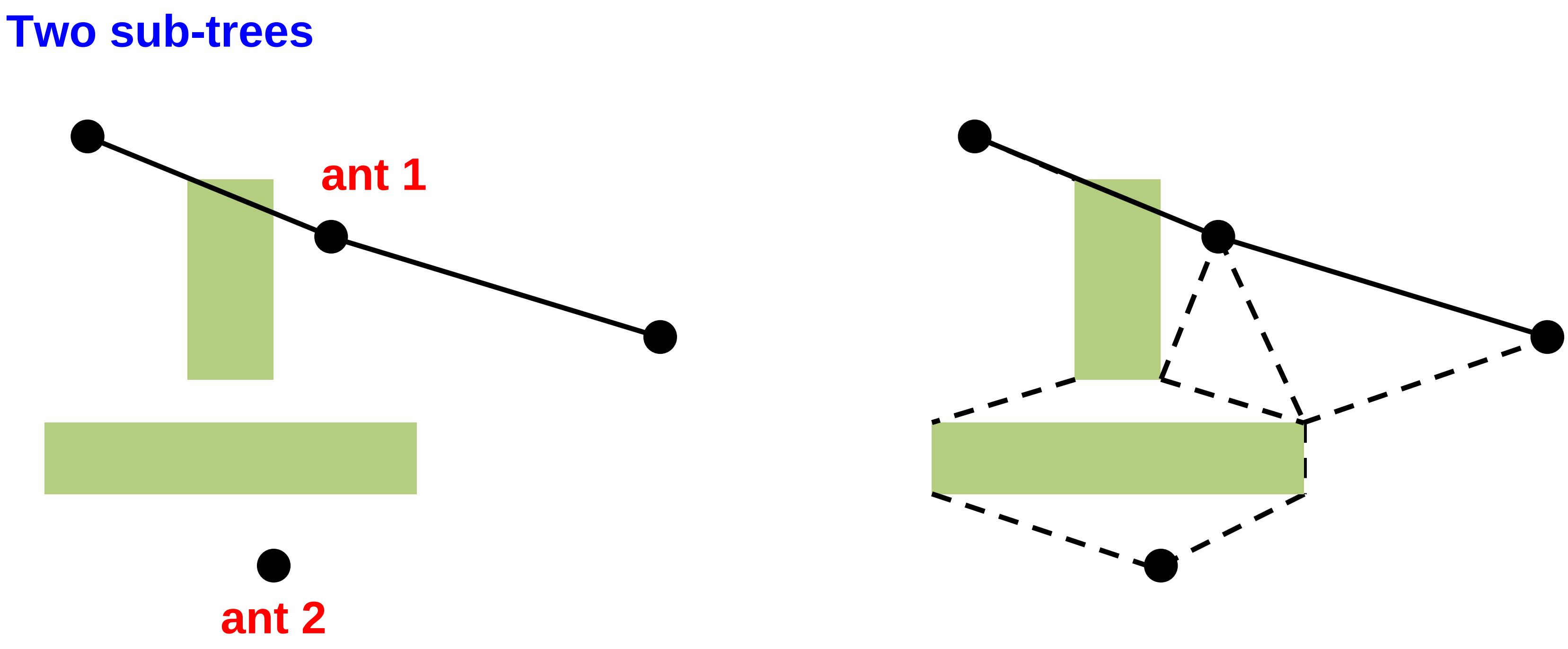
- many locations of a sub-tree
- A greedy method: determined by removed edges of MST
  - End point of removed edge with smaller wirelength



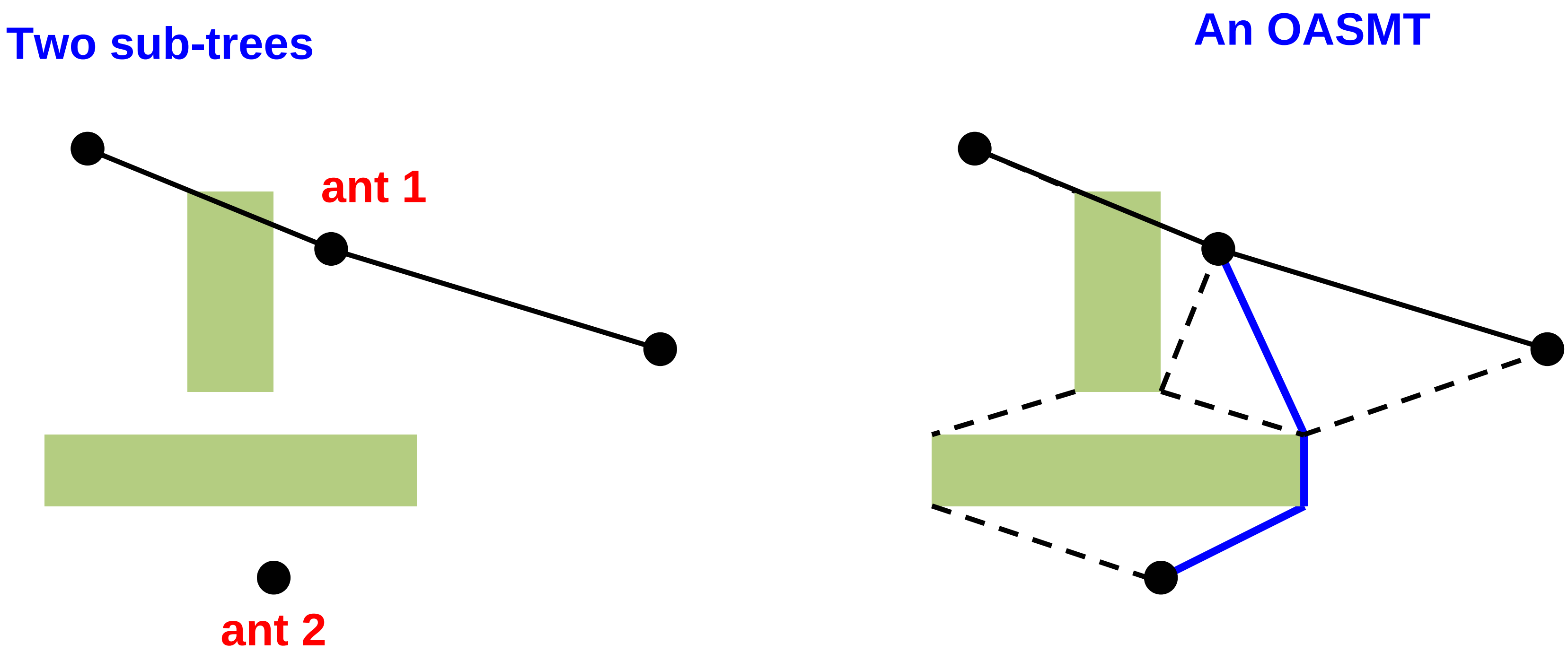
Merge sub-trees: using the ACO based algorithm



Merge sub-trees: using the ACO based algorithm

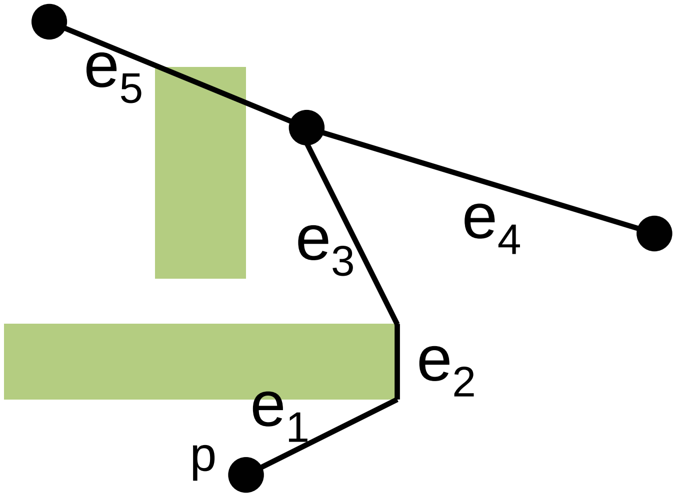


Merge sub-trees: using the ACO based algorithm



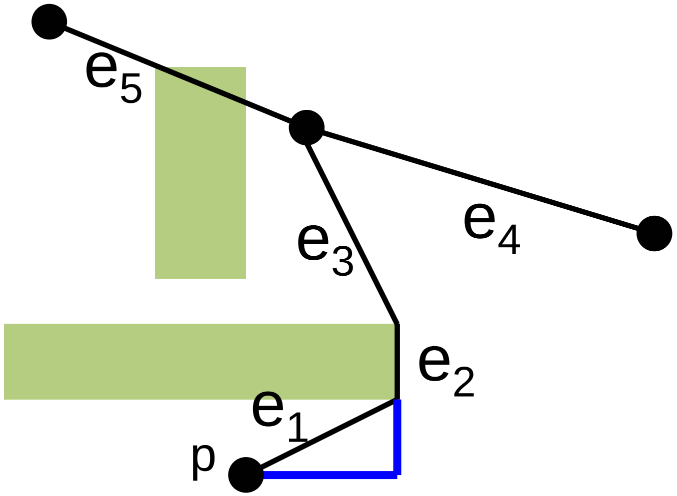
Step 4.1: Rectilinearization

- To modify all tree edges into either horizontal or vertical segments
- Goal: share as many segments as possible
  - Use BFS to traverse the tree from a 1-degree terminal
  - Follow the preferred L-shaped segment to generate rectilinearized segments greedily



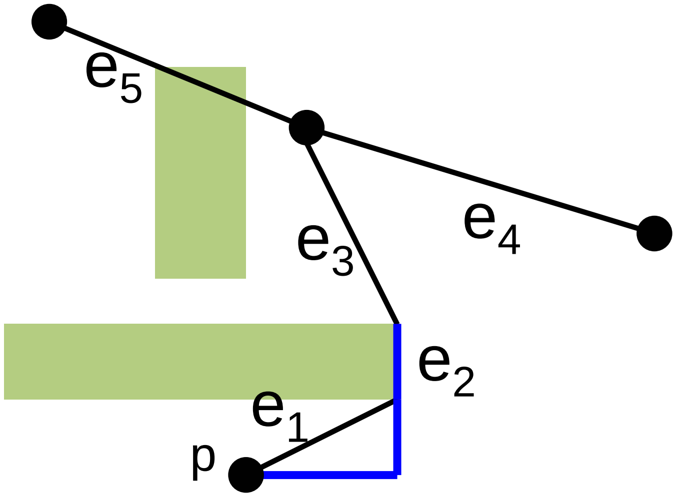
Step 4.1: Rectilinearization

- To modify all tree edges into either horizontal or vertical segments
- Goal: share as many segments as possible
  - Use BFS to traverse the tree from a 1-degree terminal
  - Follow the preferred L-shaped segment to generate rectilinearized segments greedily



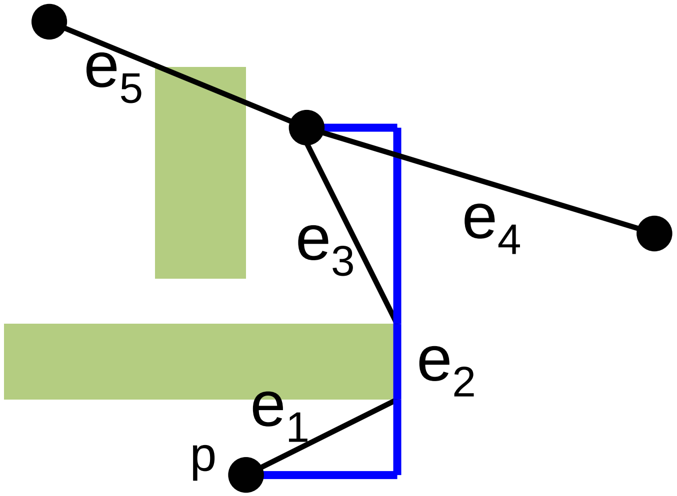
Step 4.1: Rectilinearization

- To modify all tree edges into either horizontal or vertical segments
- Goal: share as many segments as possible
  - Use BFS to traverse the tree from a 1-degree terminal
  - Follow the preferred L-shaped segment to generate rectilinearized segments greedily



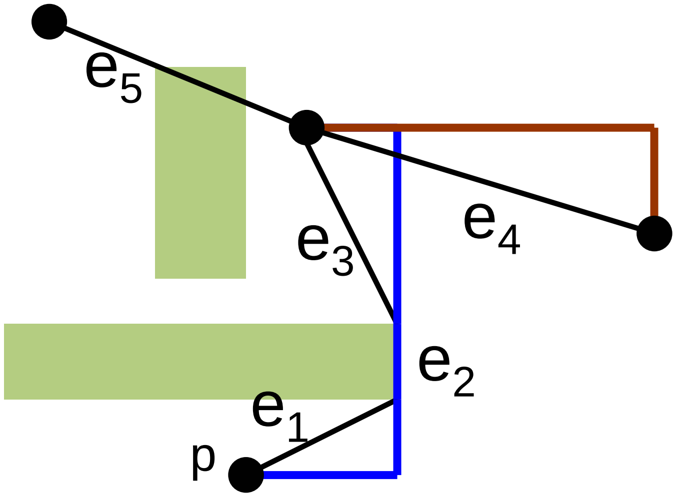
Step 4.1: Rectilinearization

- To modify all tree edges into either horizontal or vertical segments
- Goal: share as many segments as possible
  - Use BFS to traverse the tree from a 1-degree terminal
  - Follow the preferred L-shaped segment to generate rectilinearized segments greedily



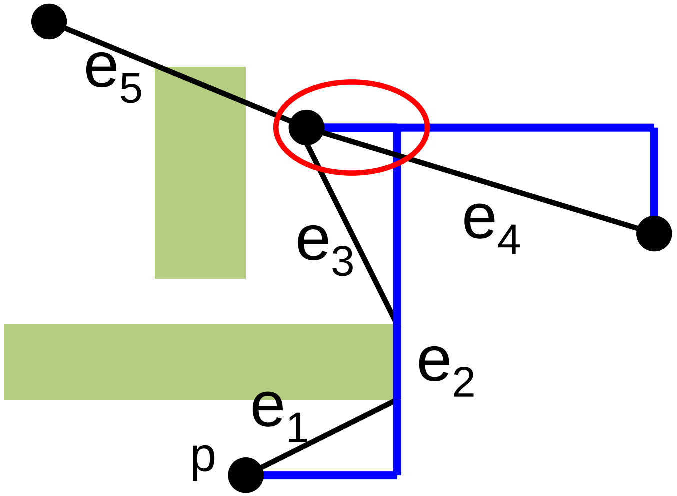
Step 4.1: Rectilinearization

- To modify all tree edges into either horizontal or vertical segments
- Goal: share as many segments as possible
  - Use BFS to traverse the tree from a 1-degree terminal
  - Follow the preferred L-shaped segment to generate rectilinearized segments greedily



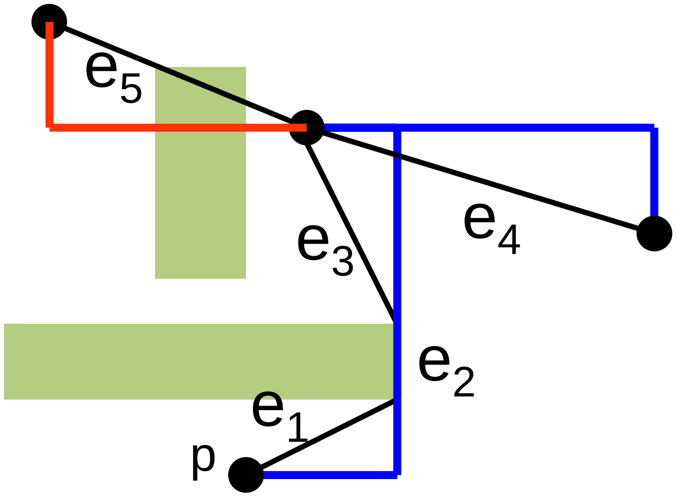
Step 4.1: Rectilinearization

- To modify all tree edges into either horizontal or vertical segments
- Goal: share as many segments as possible
  - Use BFS to traverse the tree from a 1-degree terminal
  - Follow the preferred L-shaped segment to generate rectilinearized segments greedily



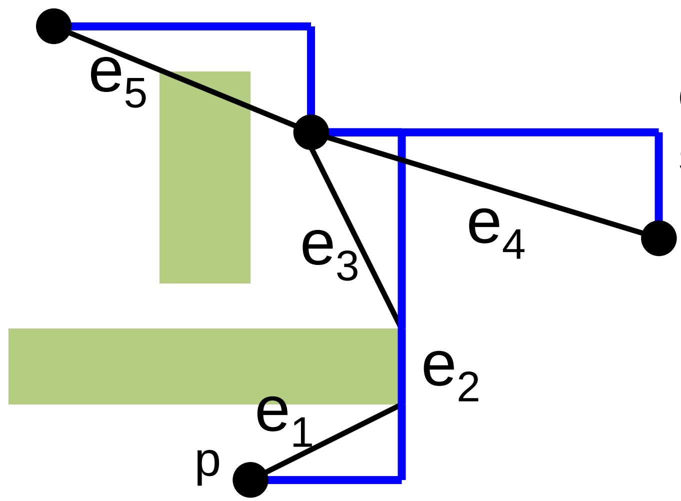
Step 4.1: Rectilinearization

- To modify all tree edges into either horizontal or vertical segments
- Goal: share as many segments as possible
  - Use BFS to traverse the tree from a 1-degree terminal
  - Follow the preferred L-shaped segment to generate rectilinearized segments greedily



Step 4.1: Rectilinearization

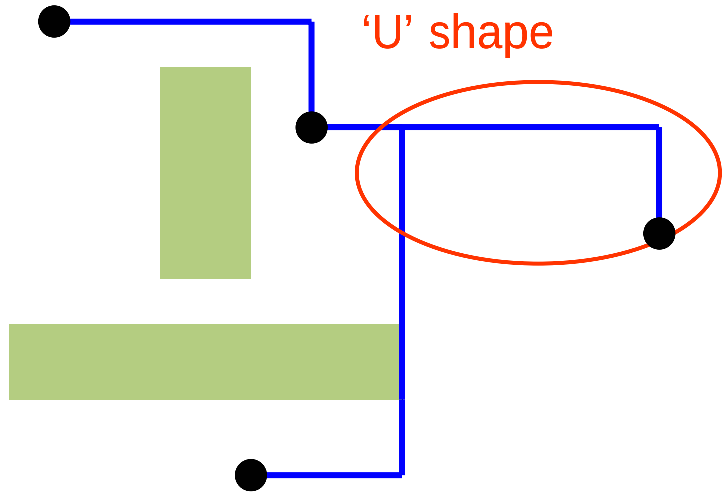
- To modify all tree edges into either horizontal or vertical segments
- Goal: share as many segments as possible
  - Use BFS to traverse the tree from a 1-degree terminal
  - Follow the preferred L-shaped segment to generate rectilinearized segments greedily



Obtain an OARSMT solution

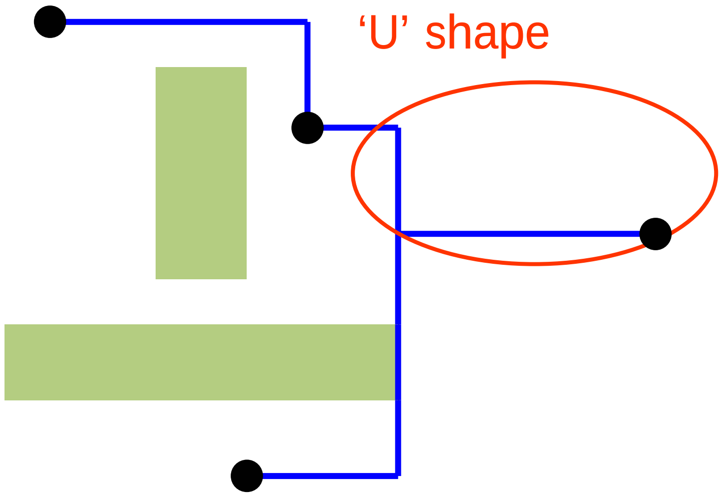
Step 4.2: Refinement

- Goal: To further improve the wirelength of the solution obtained in step 4.1
- Eliminate ‘U’ shape connections in the OARSMT



Step 4.2: Refinement

- Goal: To further improve the wirelength of the solution obtained in step 4.1
- Eliminate ‘U’ shape connections in the OARSMT



Outline

- Introduction
- Problem formulation
- Algorithm
- Experimental results
- Conclustion

Experimental Setup

- Comparison targets and platforms
  - **An-OARSMan**: 755MHz CPU and 4GB memory
  - **2-OASMT:** 755MHz CPU and 4GB memory
  - **CDCTree**: 2.66G CPU and 1G memory
  - **Spanning graph**: 1200MHz CPU and 8GB memory
- Platform of ours: 1200MHz CPU and 8GB
- Benchmarks
  - Industrial cases
  - Randomly generated cases in 10000 x 10000 plane

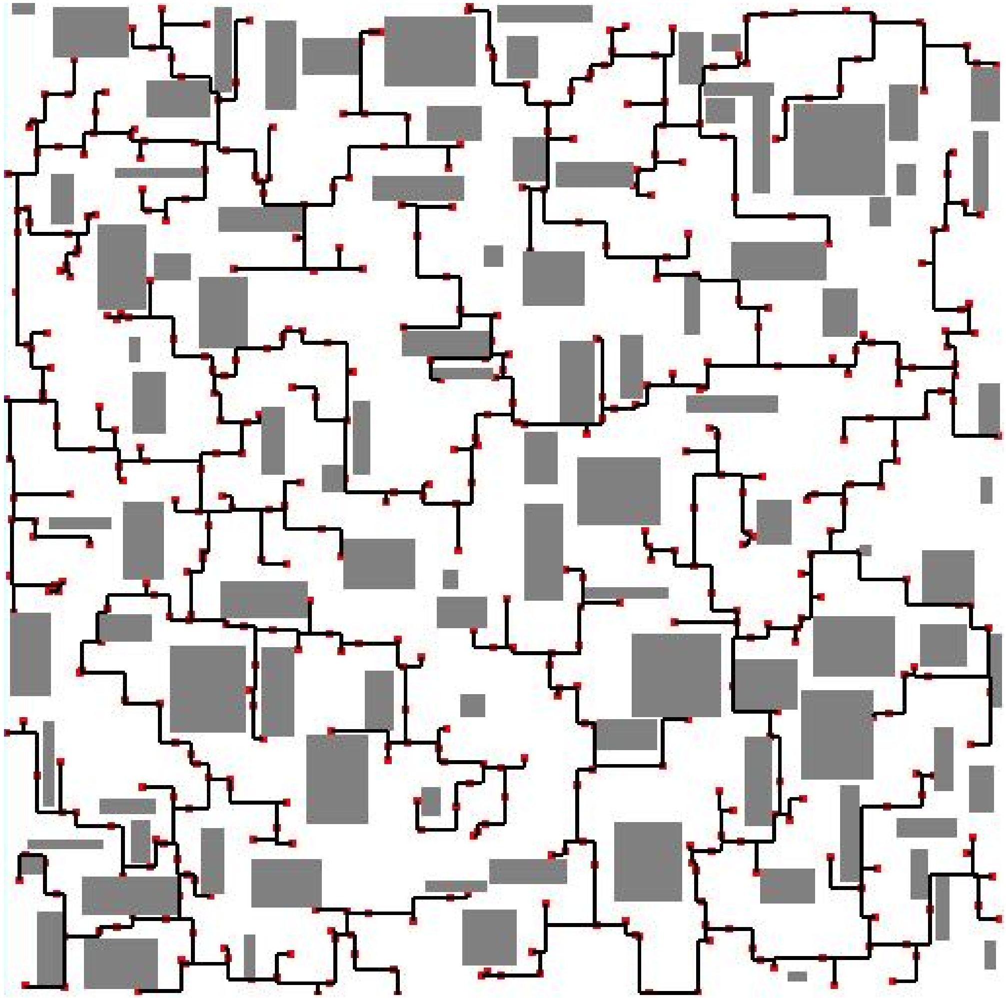
Percentage of wirelength improvement

Term#	obs#	Wirelength improvement (%)			
		An-OARSMan	CDCTree	2-OASMT	Spannging graph
10	32	-	-	-	2.80
74	625	-	-	-	5.26
115	1024	-	-	-	4.62
10	10	2.12	-1.04	10.39	7.06
30	10	-0.30	-3.65	5.30	0.41
50	10	10.67	9.43	3.53	0.91
70	10	7.87	8.22	3.55	1.32
100	10	6.62	3.92	7.55	1.75
100	500	-	-	43.69	2.82
200	500	-	-	36.37	2.29
200	800	-	-	39.54	0.64
200	1000	-	-	44.14	0.46
500	100	-	-	12.89	1.11
1000	100	-	-	4.87	1.84
1000	10000	-	-	54.37	-
average		5.40	3.38	22.18	1.87

Runtime comparison

Term#	obs#	Runtime (s)				
		An-OARSMan	CDCTree	2-OASMT	Spannging graph	Ours
10	32	-	-	-	<0.01	<0.01
74	625	-	-	-	14.17	0.1
115	1024	-	-	-	60.69	0.21
10	10	0.164	0.485	0.002	<0.01	<0.01
30	10	1.075	1.034	0.003	<0.01	<0.01
50	10	3.504	8.79	0.004	0.01	<0.01
70	10	10.552	67.62	0.004	0.01	<0.01
100	10	26.974	595.1	0.004	0.02	<0.01
100	500	-	-	0.057	12.49	0.31
200	500	-	-	0.062	28.15	0.36
200	800	-	-	0.095	72.66	1.53
200	1000	-	-	0.129	112.29	1.8
500	100	-	-	0.026	4.14	0.27
1000	100	-	-	0.037	35.34	0.81
1000	10000	-	-	2.823	-	4.2

Routing result with 500 terminals and 100 obstacles



Outline

- Introduction
- Problem formulation
- Algorithm
- Experimental results
- Conclusion

Conclusion

- A fast and stable approach for obstacle-avoiding rectilinear Steiner minimal tree construction is presented.
- Compared with state-of-the-art works, our approach has the best wirelength performance in most of the cases and the runtime is very small even for large cases.
- The high efficiency and good solution quality of our approach makes it extremely practical in the routing process.

Thank You