

Global Routing

Guillermo Vidal Sulé

Universitat Politècnica de Catalunya
Algorithms for VLSI

Professor:
Jordi Cortadella Fortuny

Algorithms for VLSI
January 11, 2026



Table of Contents

Introduction

The problem

Routing

Experiments

The code

Table of Contents

Introduction

The problem

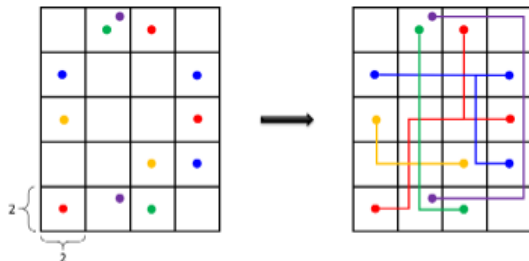
Routing

Experiments

The code

The problem

- 1 Route all nets
- 2 Avoid congestions
- 3 Optimize wirelength



Steiner Tree Algorithms: FLUTE vs. GeoSteiner

Key Differences:

- **GeoSteiner:**

- Computes the *Exact* optimal RSMT.
- Uses Branch-and-Cut & Dynamic Programming.
- High complexity; used for benchmarking.

- **FLUTE (Fast Lookup Table Based RSMT):**

- Pre-computes optimal patterns for low-degree nets (≤ 9 pins).
- Uses "Net Breaking" for high-degree nets.
- Extremely fast ($O(n \log n)$); industry standard.

FLUTE

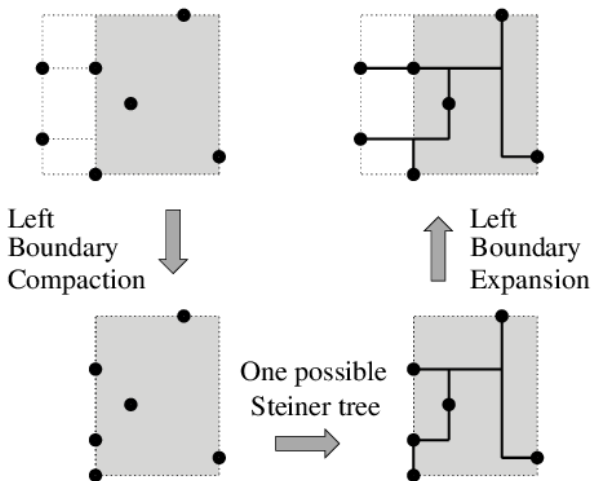


Table of Contents

Introduction

The problem

Routing

Experiments

The code

Formalization

The problem involves the following, including some personal extras:

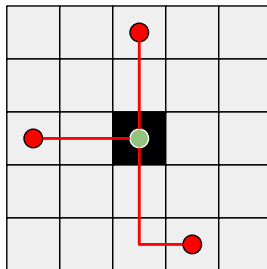
- ① All nets must be routed (if possible)
- ② **Each** edge has its own capacity.
- ③ There are obstacles.
- ④ The wirelength must be minimal

Shortcomings of FLUTE

Problem

The FLUTE algorithm is blind, meaning that it does not account for collisions or congestions, potentially producing illegal Steiner points.

5x5 Matrix



Shortcomings of FLUTE ii

5x5 Matrix

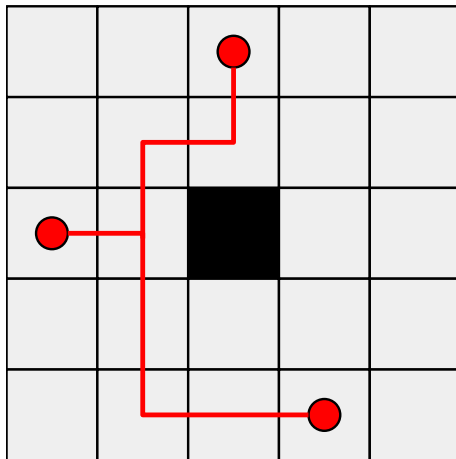


Table of Contents

Introduction

The problem

Routing

Experiments

The code

Full algorithm

Algorithm 1 Global Route Strategy

```

1: Input: Net  $N$ , Grid  $G$ 
2:  $T \leftarrow \text{FLUTE}(N.pins)$  ▷ Imported C function
3: for each  $branch(u, v) \in T$  do
4:   if  $\text{IS\_OBSTACLE}(u)$  or  $\text{IS\_OBSTACLE}(v)$  then
5:      $u, v \leftarrow \text{Find\_Nearest}(G, u, v)$  ▷ BFS to fix illegal points
6:   end if
7:    $\text{ROUTE\_POINTS}(G, u, v, \text{Net\_Id})$  ▷ A* for final path
8: end for

```

Find_Nearest

Algorithm 2 BFS to Escape Obstacles

```

1: function FIND_NEAREST( $G, P$ )
2:    $Q.enqueue(P, score = 0)$ 
3:   while  $Q$  is not empty do
4:      $Curr \leftarrow Q.dequeue()$ 
5:     if not IS_OBSTACLE( $G, Curr$ ) then return  $Curr$ 
6:     end if
7:     for each neighbor  $N$  of  $Curr$  do
8:       if FEASIBLE( $G, N$ ) and  $N \notin Visited$  then
9:          $Q.enqueue(N, score + 1)$ 
10:      end if
11:    end for
12:  end while
13: end function

```

A*

Algorithm 3 A* Point-to-Point Routing

```

1: function ROUTE_POINTS( $G, Start, Goal, Net\_Id$ )
2:    $H(P) \leftarrow |P.x - Goal.x| + |P.y - Goal.y|$ 
3:    $OpenSet.push(Start, F = H(Start))$ 
4:   while  $OpenSet$  is not empty do
5:      $Curr \leftarrow OpenSet.pop\_lowest\_F()$ 
6:     if  $Curr = Goal$  then return RECONSTRUCT_PATH
7:     end if
8:     for each neighbor  $N$  of  $Curr$  do
9:       if HAS_CAPACITY( $G, Curr, N$ ) then
10:         $newG \leftarrow G\_Scores(Curr) + 1$ 
11:        if  $newG < G\_Scores(N)$  then
12:           $Came\_From(N) \leftarrow$  opposite direction
13:           $OpenSet.push(N, F = newG + H(N))$ 
14:        end if
15:      end if
16:    end for
17:  end while
18: end function

```

Table of Contents

Introduction

The problem

Routing

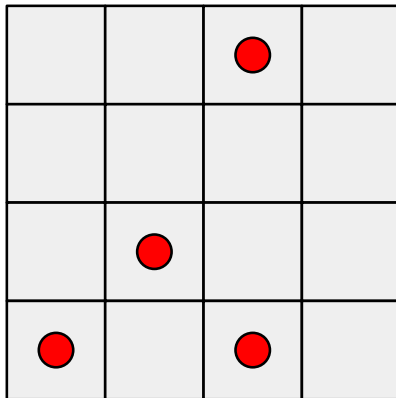
Experiments

The code

Simple test

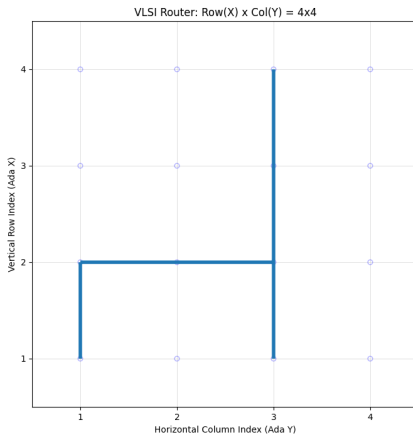
The first and most simple test. No restrictions and no obstacles.

4x4 Matrix



Simple test - result

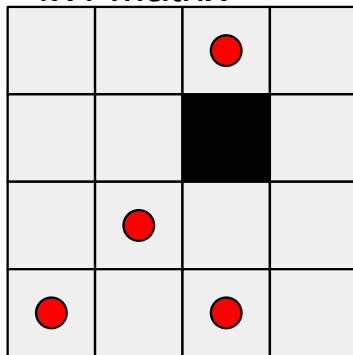
Success. Total length = 6. Minimal.



Collision test

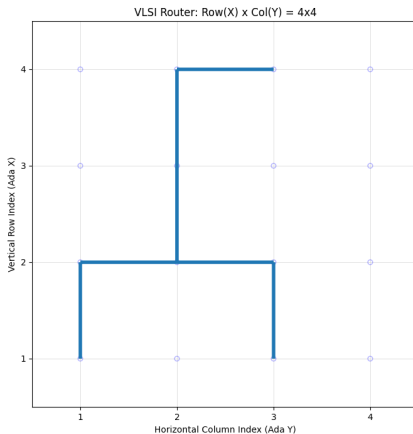
No restrictions. One obstacle.

4x4 Matrix



Collision test - result

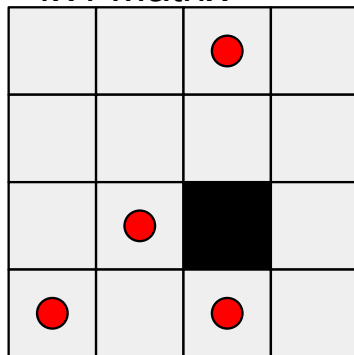
Success. Total length = 7. Minimal.



Illegality test

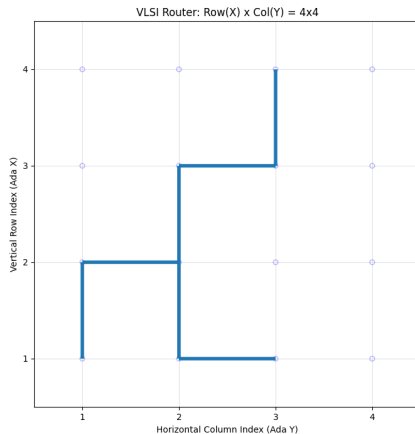
No restrictions. One obstacle. One illegal Steiner point.

4x4 Matrix



Illegality test - result

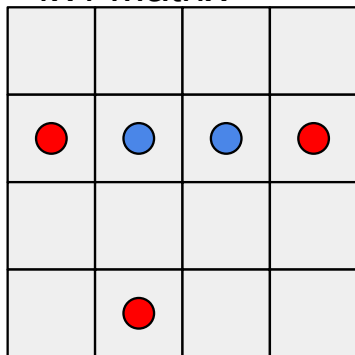
Success. Total length = 7. Minimal.



Capacity test

All edges have a maximum capacity of 1. Two nets.

4x4 Matrix



Capacity test - result

Success. Total length = 8. Minimal.

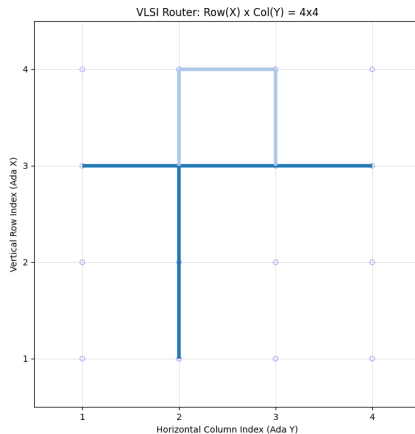


Table of Contents

Introduction

The problem

Routing

Experiments

The code

Minor comments

```
1 type Direction is (Up, Left, Down, Right);
2 function "+" (L : Point; R : Direction)
3 return Point is
4     Result : Point := L;
5 begin
6     case R is
7         when Up      => Result.X := - 1;
8         when Left    => Result.Y := - 1;
9         when Down    => Result.X := + 1;
10        when Right   => Result.Y := + 1;
11    end case;
12    return Result;
13 end "+";
```

Thank You