Paired-Spacing-Constrained Package Routing with Net Ordering Optimization

Presenter: Ying-Jie Jiang

Advisor: Shao-Yun Fang

The Electronic Design Automation Laboratory
Department of Electrical Engineering
National Taiwan University of Science and Technology
Taipei 106, Taiwan

THE EDA LAB
TAIWAN TECH

Outline

- Introduction
- Algorithm Flow
  - Net Ordering Determination
  - Tile and Network Graph Construction
  - MCMF-based Global Routing Derivation
- Experimental Results
- Conclusion

Introduction

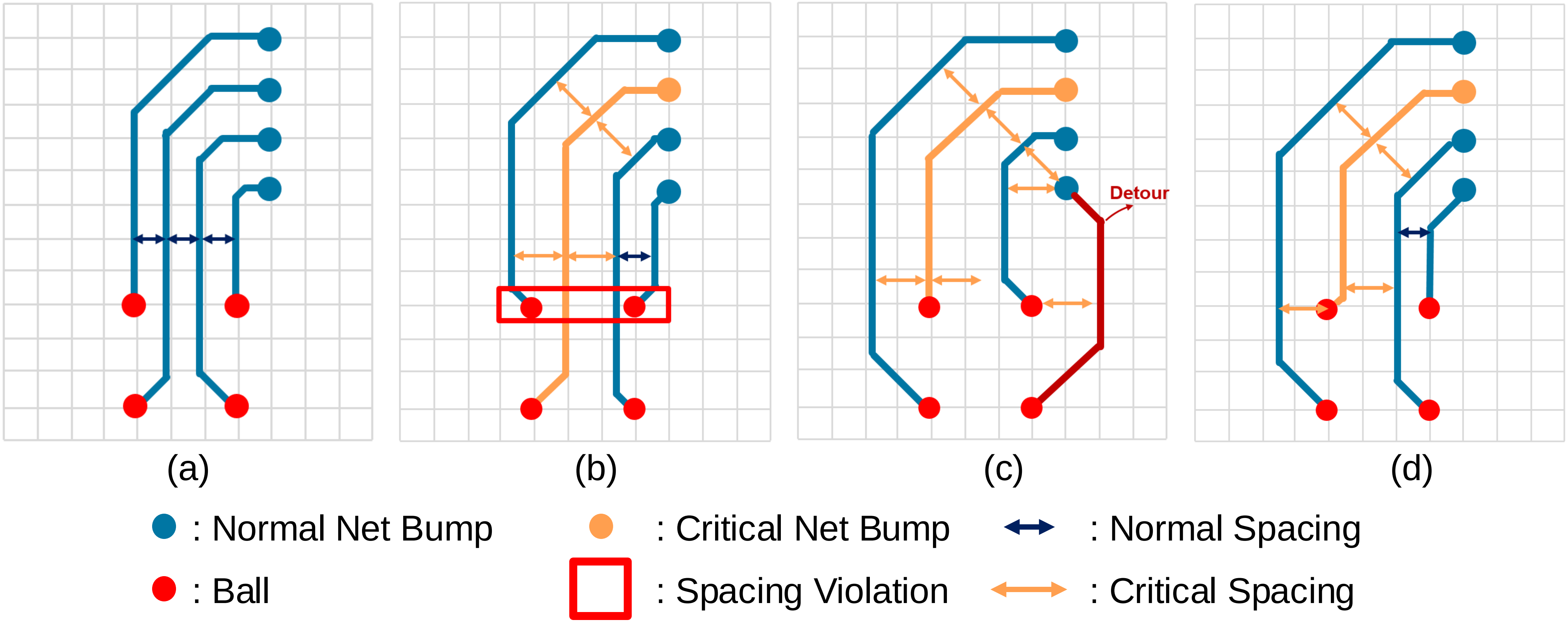
Introduction

- **Paired-spacing constraint** is one of the most important constraints in advanced package technology
  - A net may require different spacing depending on the nets it is adjacent to
  - There can be at most C_2^n spacing rules for n nets

- It is important to establish different spacing rules for different bump nets
  - Ensure signal integrity
  - Minimize crosstalk

Example of Pair Spacing

- If we apply post-refinement or a pessimistic spacing rule
  - Result in suboptimal solution



Design Rules

- Non-crossing Constraint
  - Two nets cannot be routed across each other
- Routing Direction Constraint
  - Routed in four kinds of orientation
    - Vertical, horizontal, 45°, and 135°
- Routing Angle Constraint
  - Reversed 45° turn is not allowed
- Paired Spacing Constraint
  - For normal (non-critical) nets, keep at least a distance S_n from other nets
  - For critical nets, keep at least a distance S_c from other nets

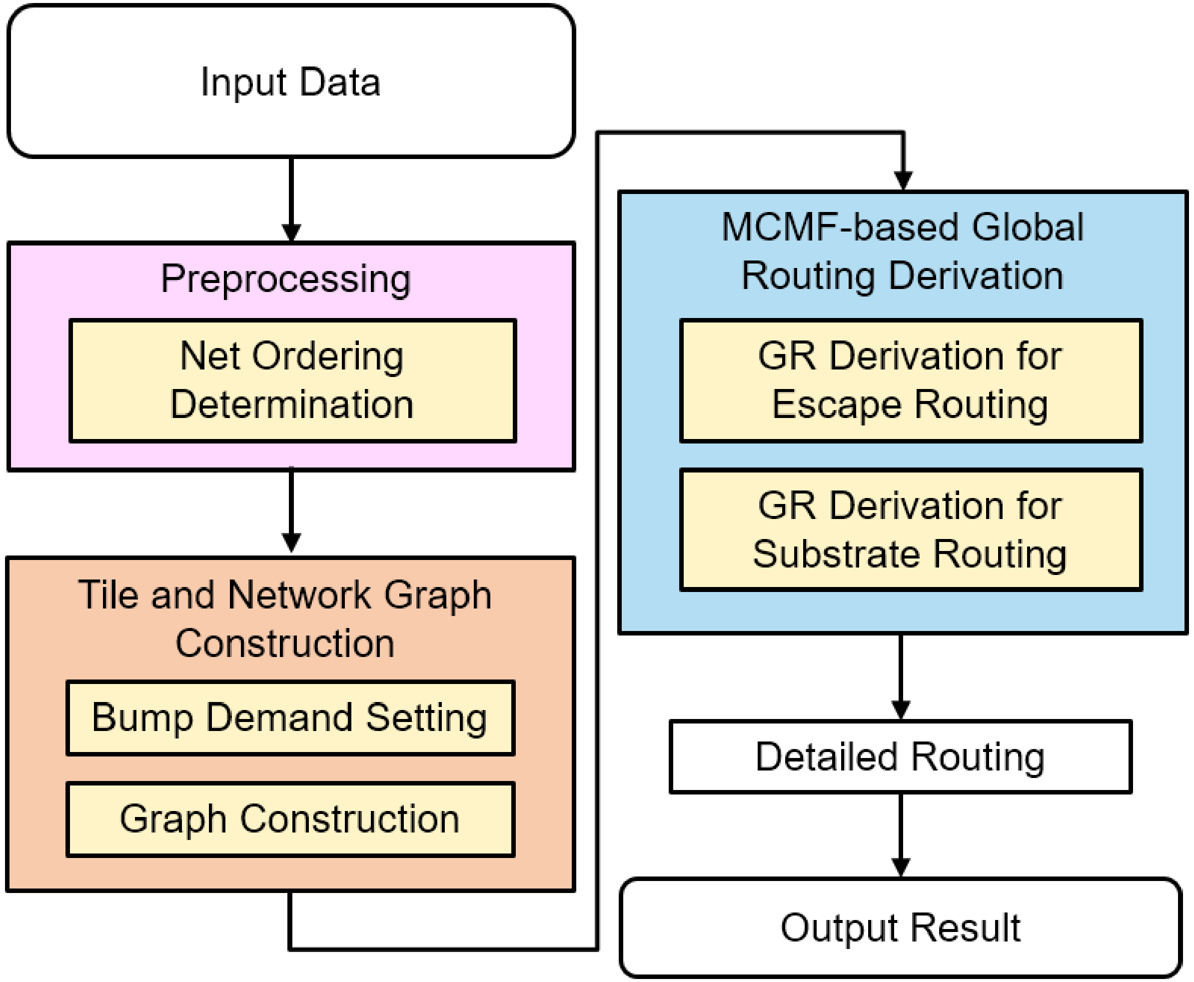
Problem Formulation

- Given
  - A chip set, a normal bump set, a critical bump set, and a ball grid array (BGA)
  - Wire width
  - Spacing rules for normal bumps and critical bumps
- Output
  - Connect all bumps to balls
    - Minimized wirelength
    - Without any design rule violation
- For better readability and more concise explanations
  - Only consider two spacing constraints in our work
    - Critical spacing rule, normal spacing rule.
  - The proposed algorithm can be trivially extended to general problem instances
    - Arbitrary spacing value can be set between each pair of adjacent nets.

Algorithm Flow

Algorithm Overview

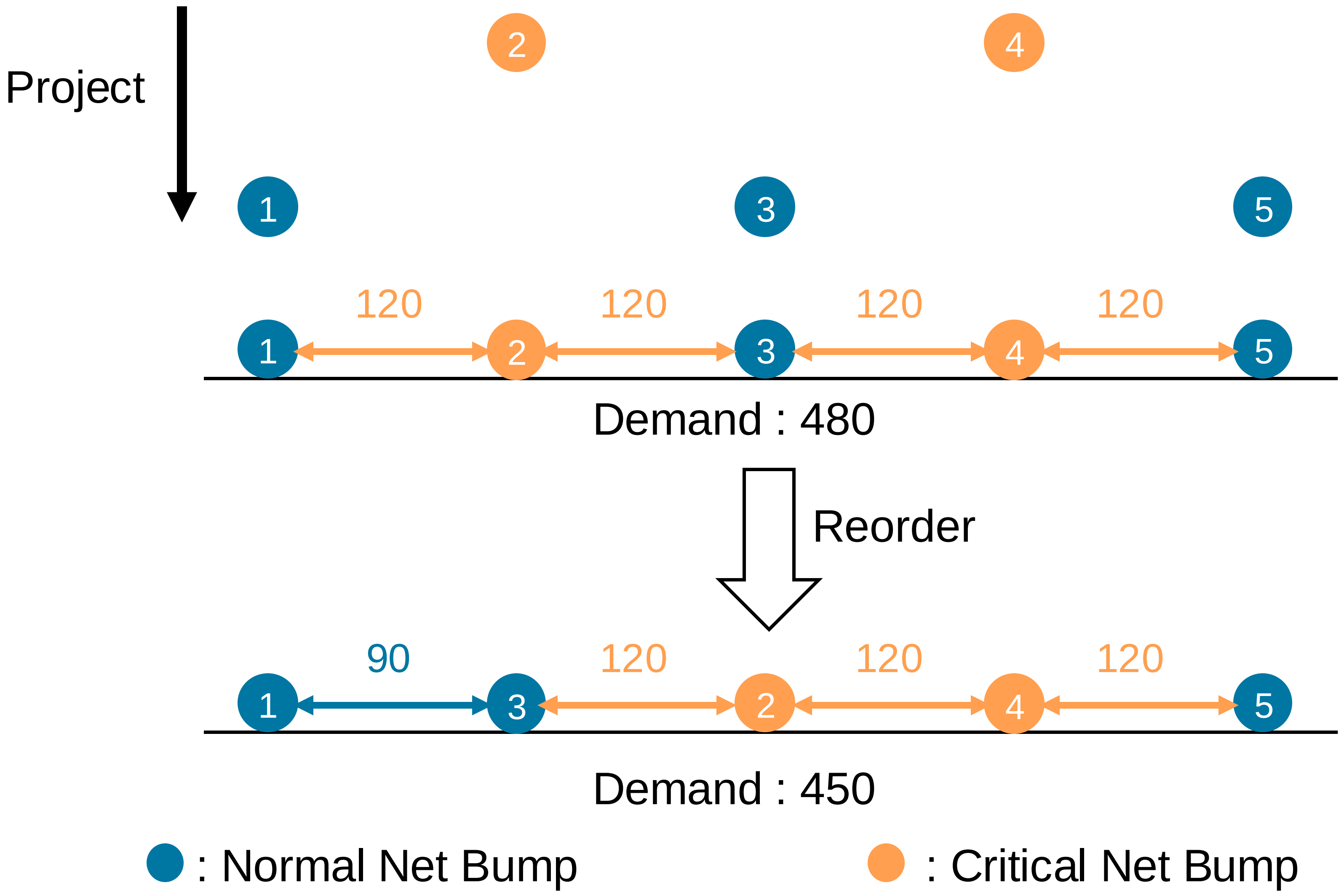
- Preprocessing
  - Determine the ordering of the nets routed
    - Minimize the routing resource demand
    - Better overall routability and wirelength
- Tile and network graph construction
  - Build a network graph
    - Both escape routing and substrate routing
- MCMF-based global routing derivation
  - Determine the global routing (GR) path
    - Ball assignment for each bump
- Detailed Routing
  - A* search is applied to build the path



Net Ordering Determination

Impact of Bump Order

- The order of bumps will result in different requirements of spacing resources



Three Adjacency Types

- Each bump can be reordered with one of its adjacent bumps
  - To achieve a good balance between reducing spacing and minimizing detours
- Three adjacency types are defined under this reordering assumption

Type \ Index	Movement		Arrangement		
	i - 1	i	i - 2	i - 1	i
0	No Change	No Change	i-2	i-1	i
1	Change	No Change	i-1	i-2	i
2	No Change	Change	i-2	i	i-1
3	Change	Change	i-1	i	i-2

i-2

Displacement > 1

Dynamic Programming Algorithm

- Based on dynamic programming
  - For different type
    - The prior optimal solution differs

Algorithm 1: Net ordering algorithm

```
input  : A sequence of bumps {b_0, b_2..., b_{n-1}}
Initialize d;
for i = 3 to n − 1 do
    d[i][0] = min(d[i−1][0], d[i−1][1]) + Cost(∅, b_{i−1}, b_i)          type1
    d[i][1] = d[i−1][2] + Cost(∅, b_{i−2}, b_i)                          type2
    d[i][2] = min(d[i−2][0] + Cost(b_{i−2}, b_i, b_{i−1}),               type3
                  d[i−2][1] + Cost(b_{i−2}, b_i, b_{i−1})),
                  d[i−2][2] + Cost(b_{i−3}, b_i, b_{i−1}))
end
totalSpacing ← ∞ ;
for j = 0 to 2 do
    totalSpacing ← min(d[n−1][j], totalSpacing)
end
return totalSpacing
```

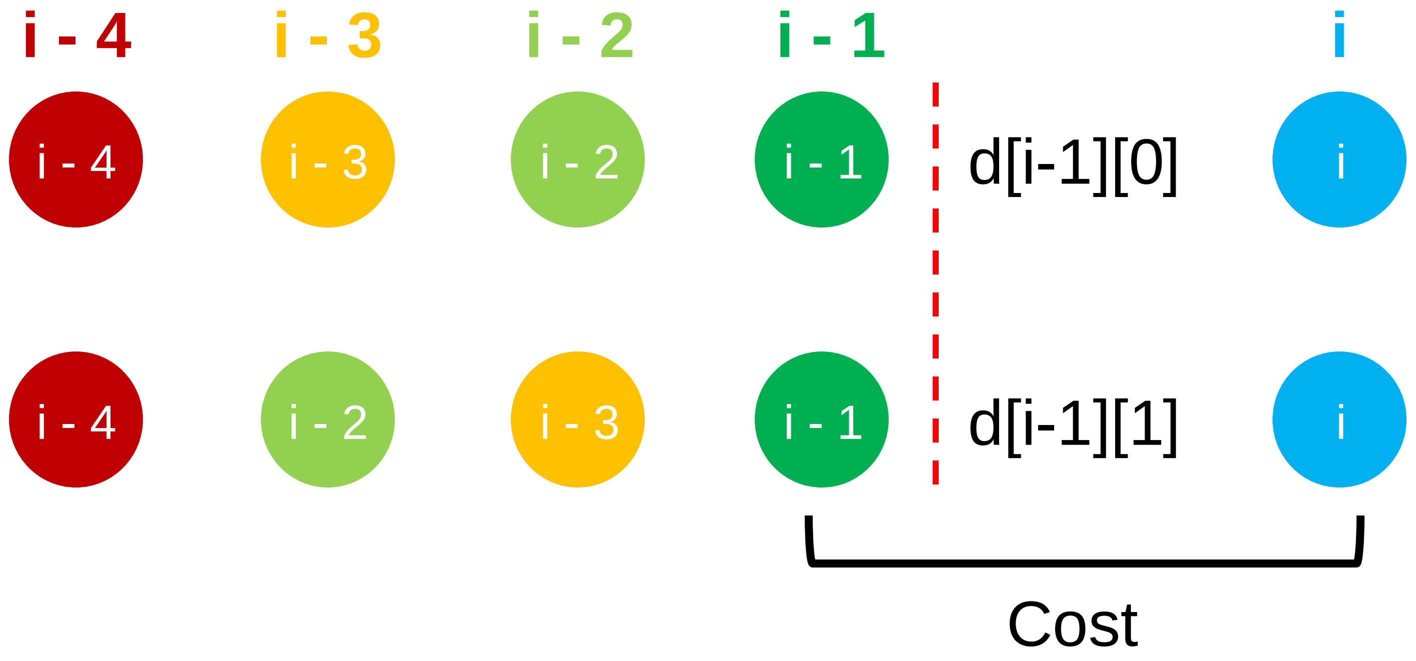
- Spacing cost for three bumps
  - Maximum of adjacent bumps

$$Cost(b_i, b_j, b_k) = \max(Spacing(b_j), Spacing(b_k)) + \begin{cases} \max(Spacing(b_i), Spacing(b_j)) & \text{if } b_i \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

Dynamic Programming Algorithm (cont’d)

- The example of Type 0:
  - $d[i][0] = \min(d[i-1][0],\ d[i-1][1]) + cost(\emptyset, b_{i-1}, bi)$
  - The optimal solution only comes from type 0 and type 1 at $d[i-1]$
    - Type 2 at $d[i-1]$ is Change, it is not feasible

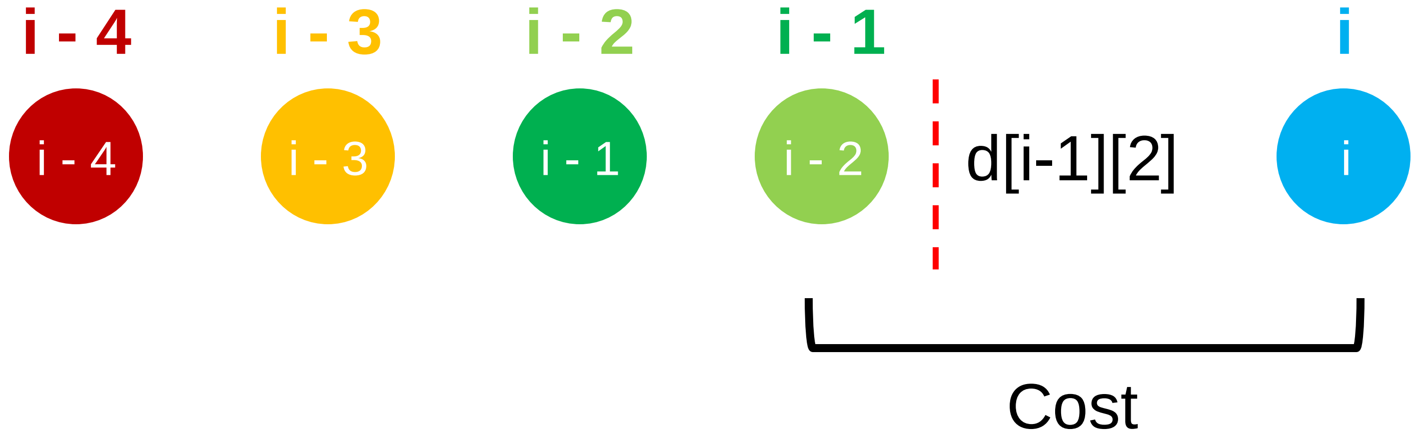
Index / Type	Movement	
	i - 1	i
0	No Change	No Change
1	Change	No Change
2	No Change	Change



Dynamic Programming Algorithm (cont’d)

- The example of Type 1:
  - $d[i][1] = d[i-1][2] + cost(\emptyset, b_{i-2}, bi)$
  - The optimal solution only comes from type 2 at $d[i-1]$
    - Both type 0 and type 1 at $d[i-1]$ is No Change, it is not feasible

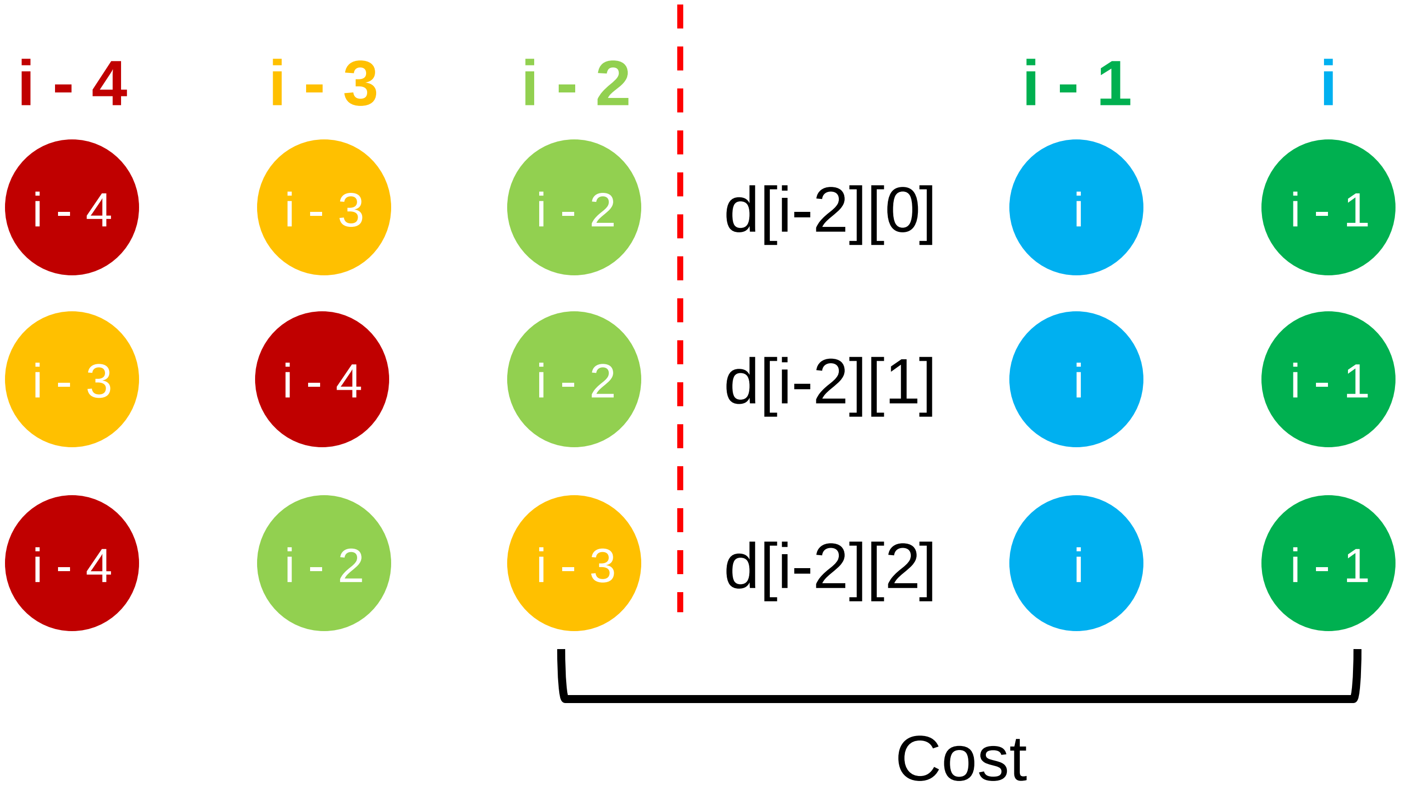
Type \ Index	Movement: i - 1	Movement: i
0	No Change	No Change
1	Change	No Change
2	No Change	Change



Dynamic Programming Algorithm (cont'd)

- The example of Type 2:
  - $d[i][2] = \min\begin{pmatrix} d[i-2][0] + cost(b_{i-2}, b_i, b_{i-1}), \\ d[i-2][1] + cost(b_{i-2}, b_i, b_{i-1}), \\ d[i-2][2] + cost(b_{i-3}, b_i, b_{i-1}), \end{pmatrix}$
  - The optimal solution comes from type 0, type 1 and type 2 at $d[i-2]$

	Movement	
Index / Type	i - 1	i
0	No Change	No Change
1	Change	No Change
2	No Change	Change



Dynamic Programming Algorithm (cont'd)

- Compare the three types at $d[i-1]$
  - Get optimal solution for bumps ordering
- Spacing cost for three bumps
  - Maximum of adjacent bumps

Algorithm 1: Net ordering algorithm

input : A sequence of bumps $\{b_0, b_2 ..., b_{n-1}\}$

Initialize d;

for $i = 3$ **to** $n-1$ **do**

  $d[i][0] = \min(d[i-1][0], d[i-1][1]) + Cost(\emptyset,\ b_{i-1}, b_i)$

  $d[i][1] = d[i-1][2] + Cost(\emptyset,\ b_{i-2}, b_i)$

  $$d[i][2] = \min(d[i-2][0] + Cost(b_{i-2},\ b_i,\ b_{i-1}), \\ d[i-2][1] + Cost(b_{i-2},\ b_i,\ b_{i-1})), \\ d[i-2][2] + Cost(b_{i-3},\ b_i,\ b_{i-1}))$$

end

$totalSpacing \leftarrow \infty$;

for $j = 0$ **to** 2 **do**

  $totalSpacing \leftarrow \min(d[n-1][j], totalSpacing)$

end

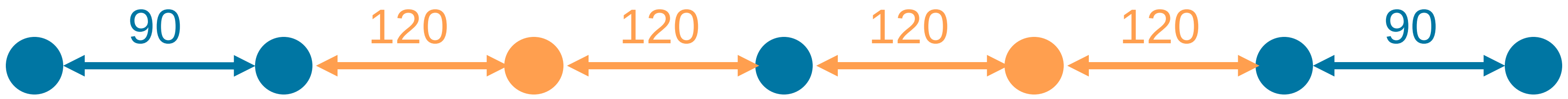
return $totalSpacing$

$$Cost(b_i, b_j, b_k) = \max(Spacing(b_j), Spacing(b_k)) + \begin{cases} \max(Spacing(b_i), Spacing(b_j)) & \text{if } b_i \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

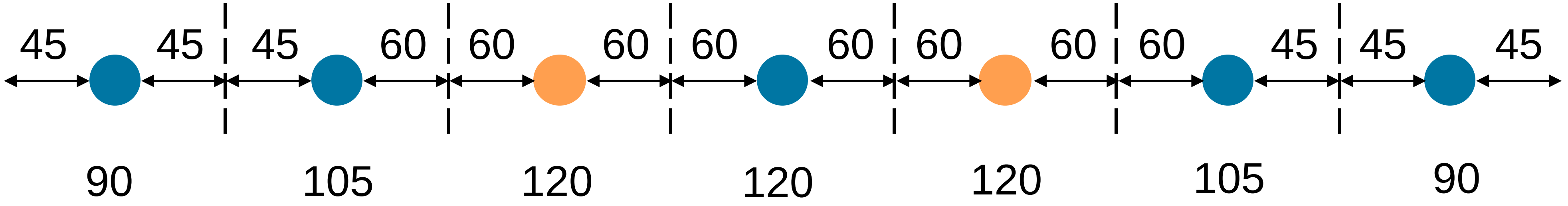
Tile and Network Graph Construction

Calculate Bump Demands

- Before graph construction, we need to calculate bump demands for all bumps
  - Each bump demand is decided by the types of adjacent bumps
    - 90, 105 and 120



(a) The spacing demand between bumps

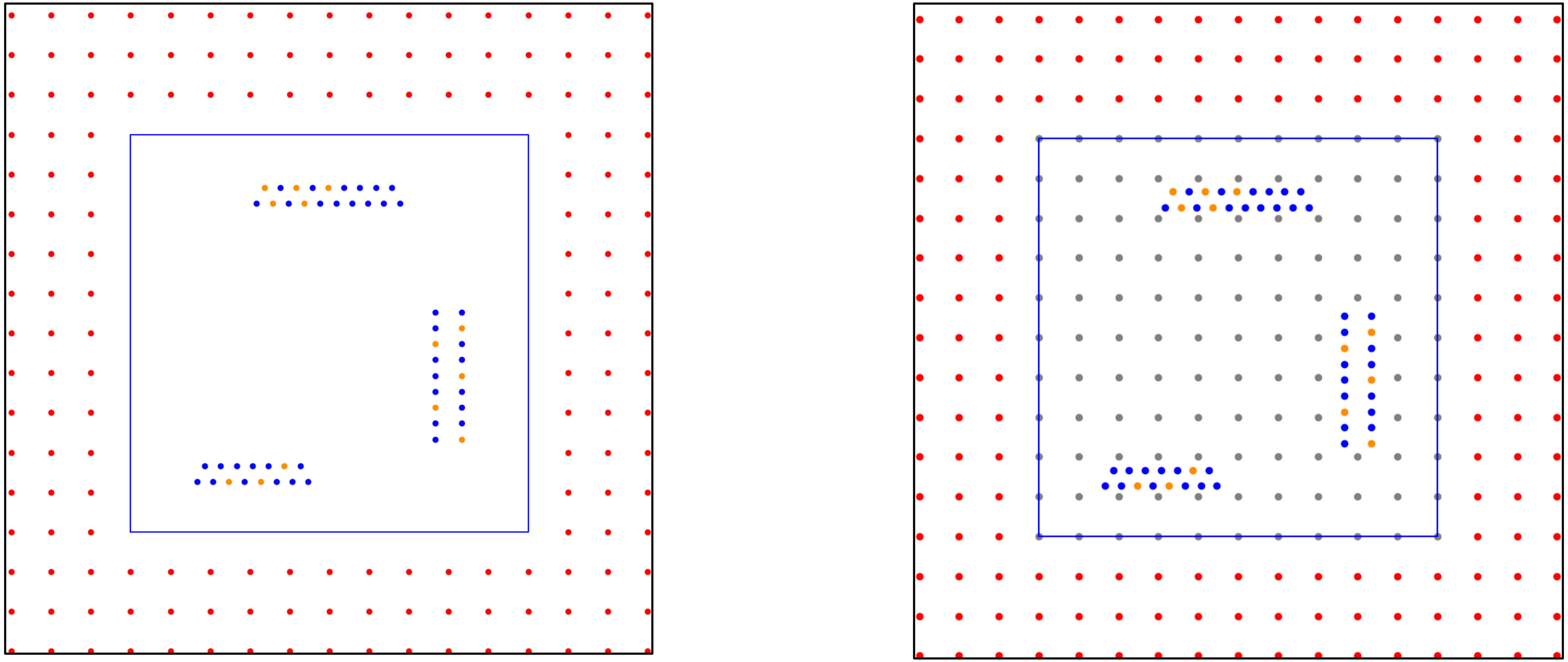


(b) Bump demand

: Normal Net Bump : Critical Net Bump $s_n = 50$ $s_c = 80$ $w = 40$

Add Virtual Points to Layout

- ❑ Add virtual points into the initial layout to facilitate the graph construction
  - Distance between virtual points is the same as the distance between balls



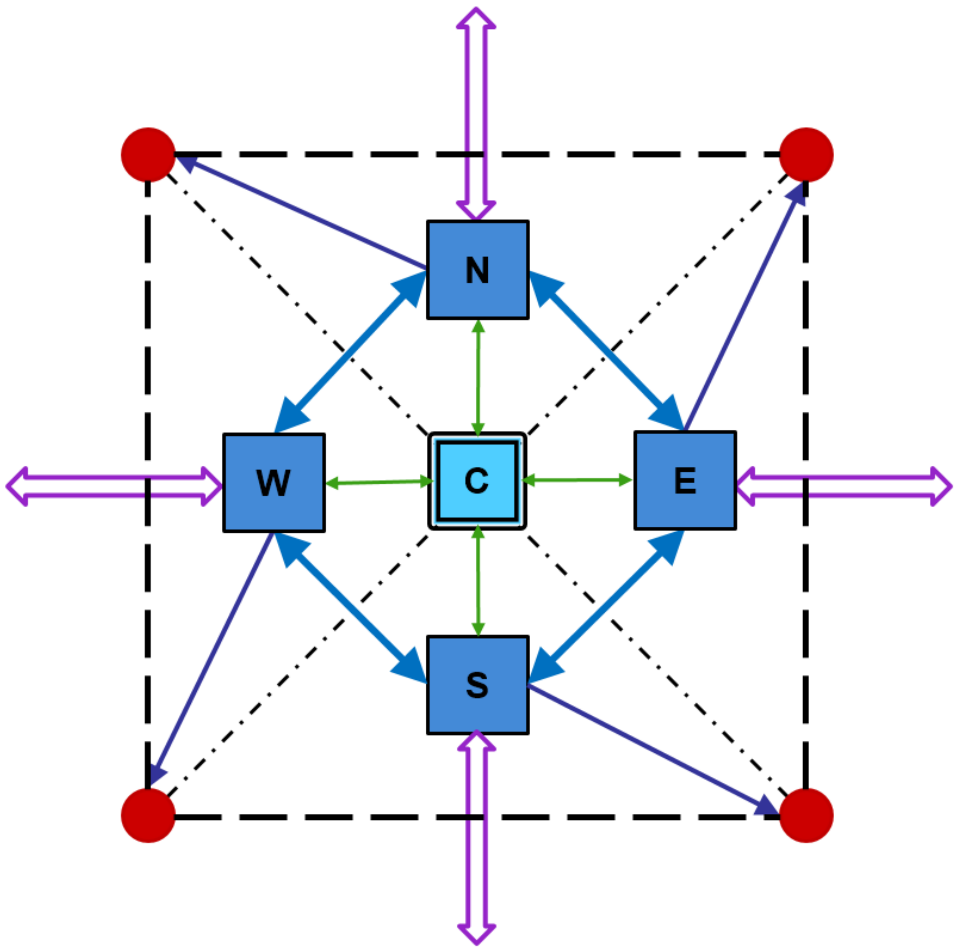
(a) Initial layout

(b) Add virtual point

: Normal Net Bump : Critical Net Bump : Ball : Virtual Point : Chip

An Existing Tile Model [Yan and Wong, TCAD'12]

- Correctly model the capacity of a square tile formed by four balls
  – It can only account for the uniform spacing constraint



O_{cap} = #nets that can pass an orthogonal cut

D_{cap} = #nets that can pass a diagonal cut

C : node capacity = D_{Cap} - 2 * [O_{Cap} / 2]

: capacity = ∞

: capacity = (O_{Cap}) / 2

: capacity = O_{Cap}

: capacity = 1

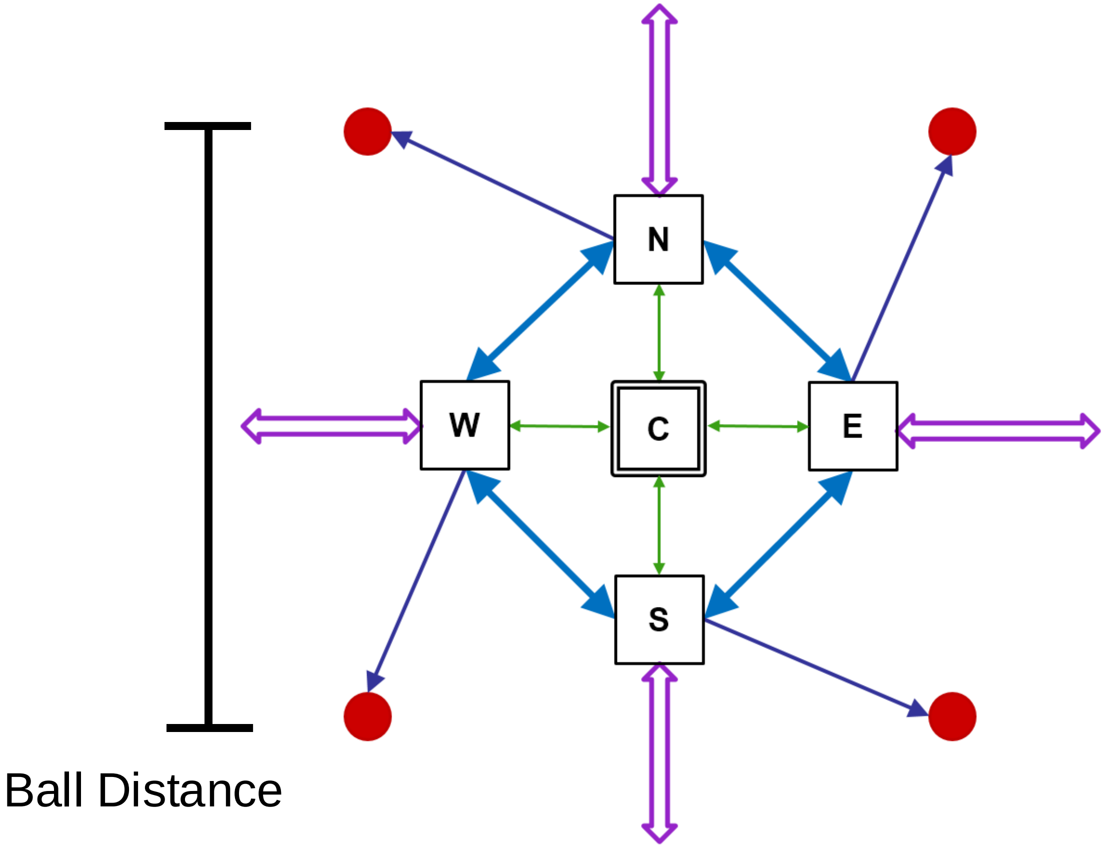
: diagonal cut

: orthogonal cut

: ball

Our Tile Model

- Convert the ball distance into the edge capacity
  - Based on the model from [Yan and Wong, TCAD'12]



C : node capacity = Floor($(\sqrt{2} - 1)$ * Ball Distance)

: capacity = ∞

: capacity = (Ball Distance) / 2

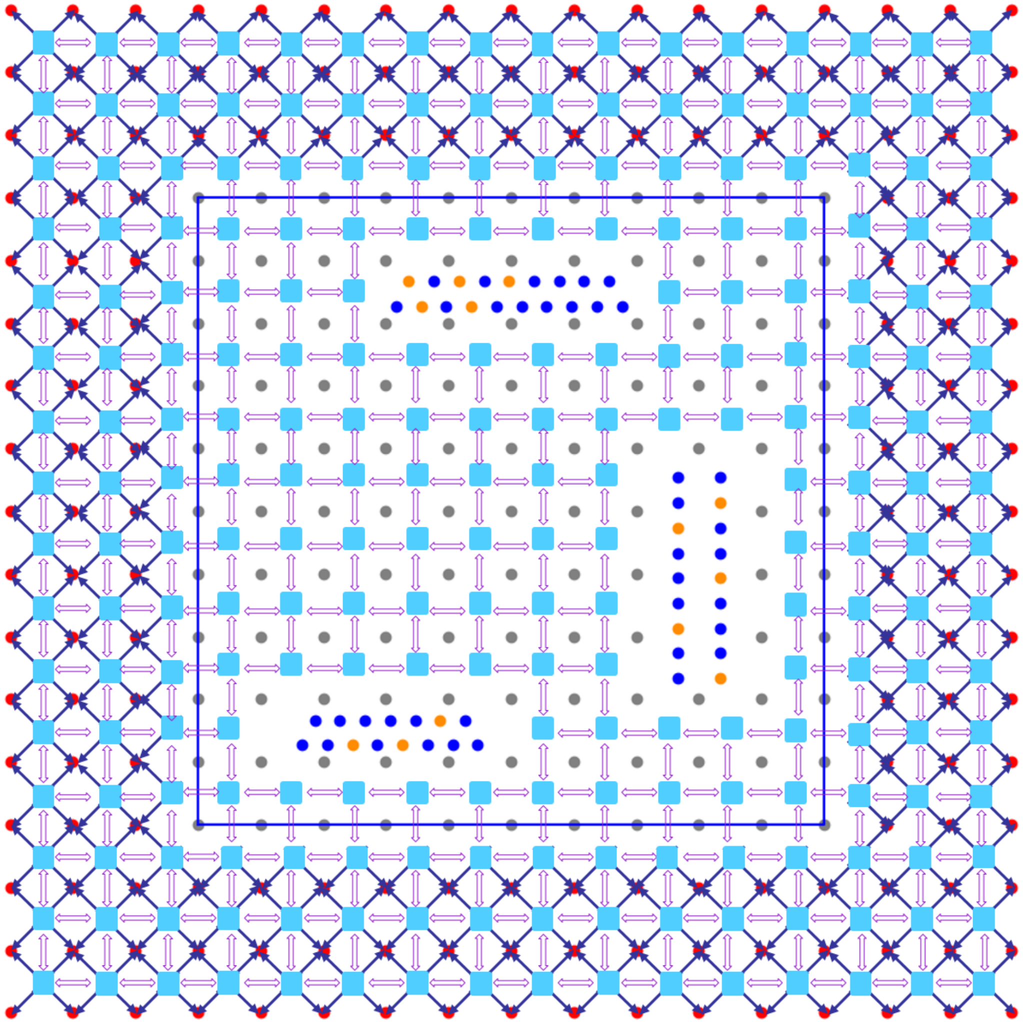
: capacity = Ball Distance

: capacity = $w + s_n$

: ball

Network Graph Construction

- The four adjacent balls or virtual points form a tile model
  - Connect two adjacent tile models with capacity = ball distance



: Normal Net Bump

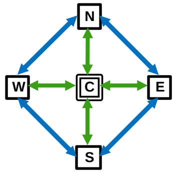
: Critical Net Bump

: Ball

: Virtual Point

: Chip

: Flow Tile

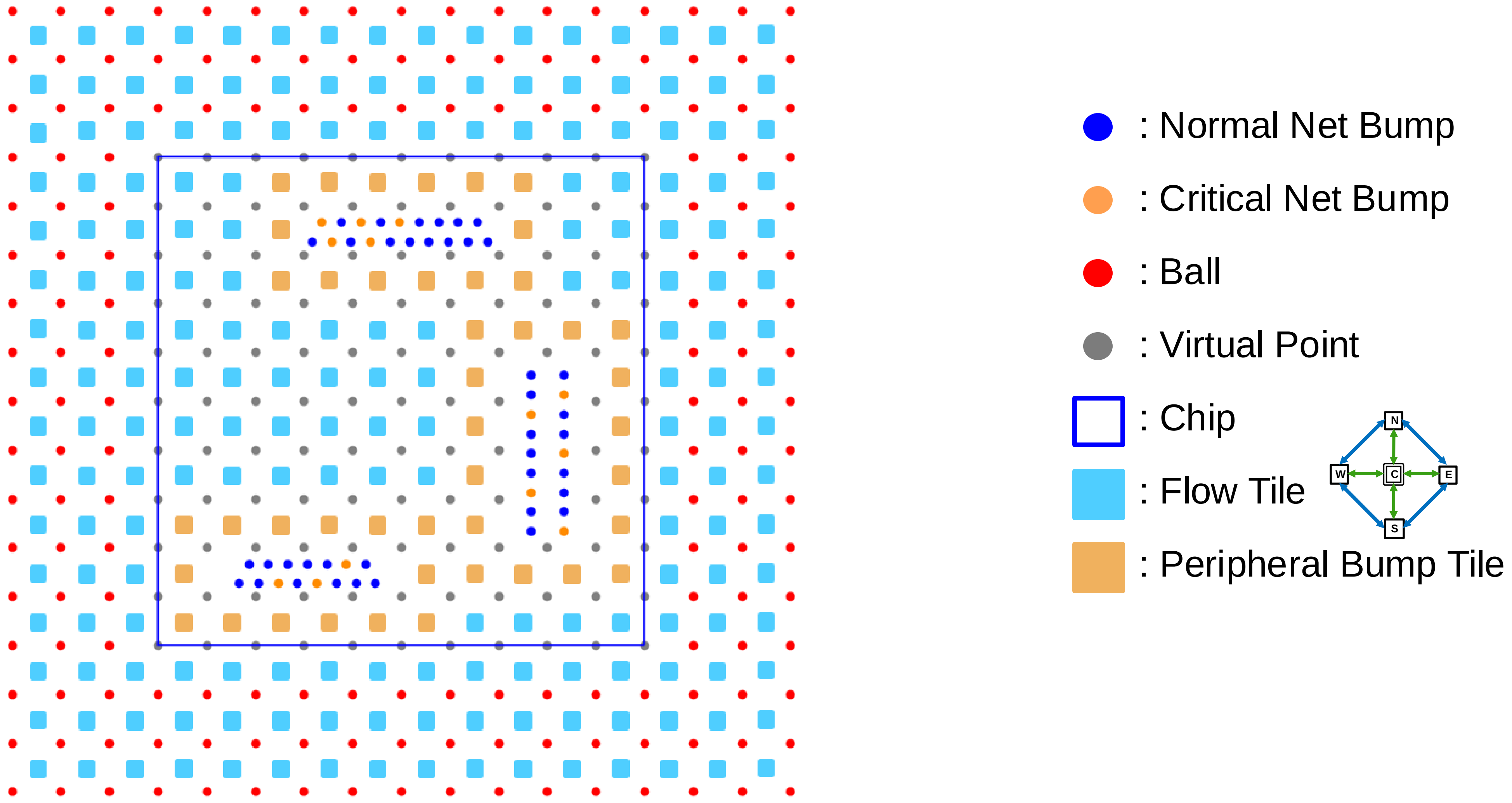


: Tile to tile edge,
capacity = Ball distance

: Tile to ball edge,
capacity = 1

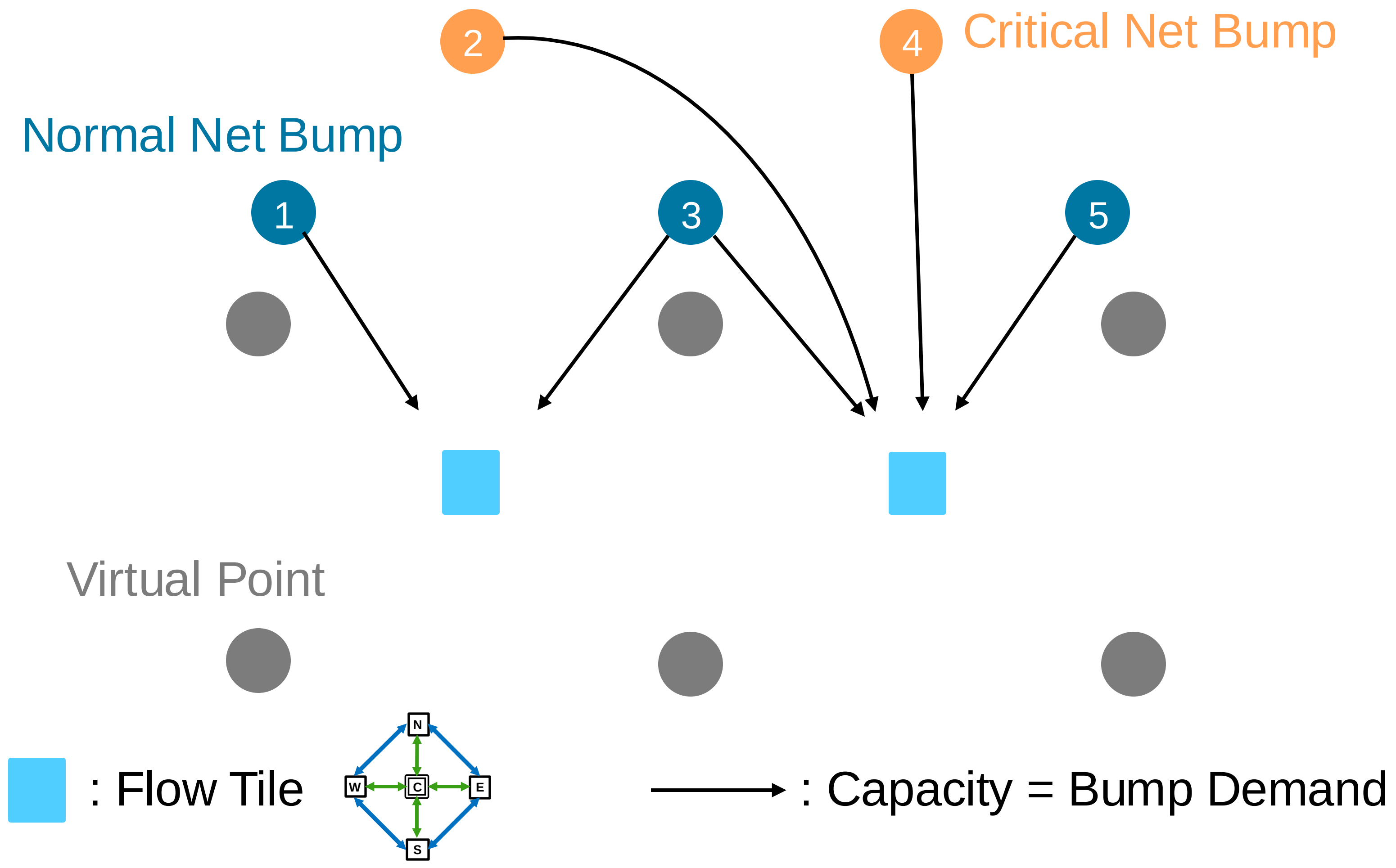
Identify Peripheral Bump Tile

- Identify the tiles whose distances to the nearest bumps are less than 1.5 times the ball distance.



Connect Bumps to the Network Graph

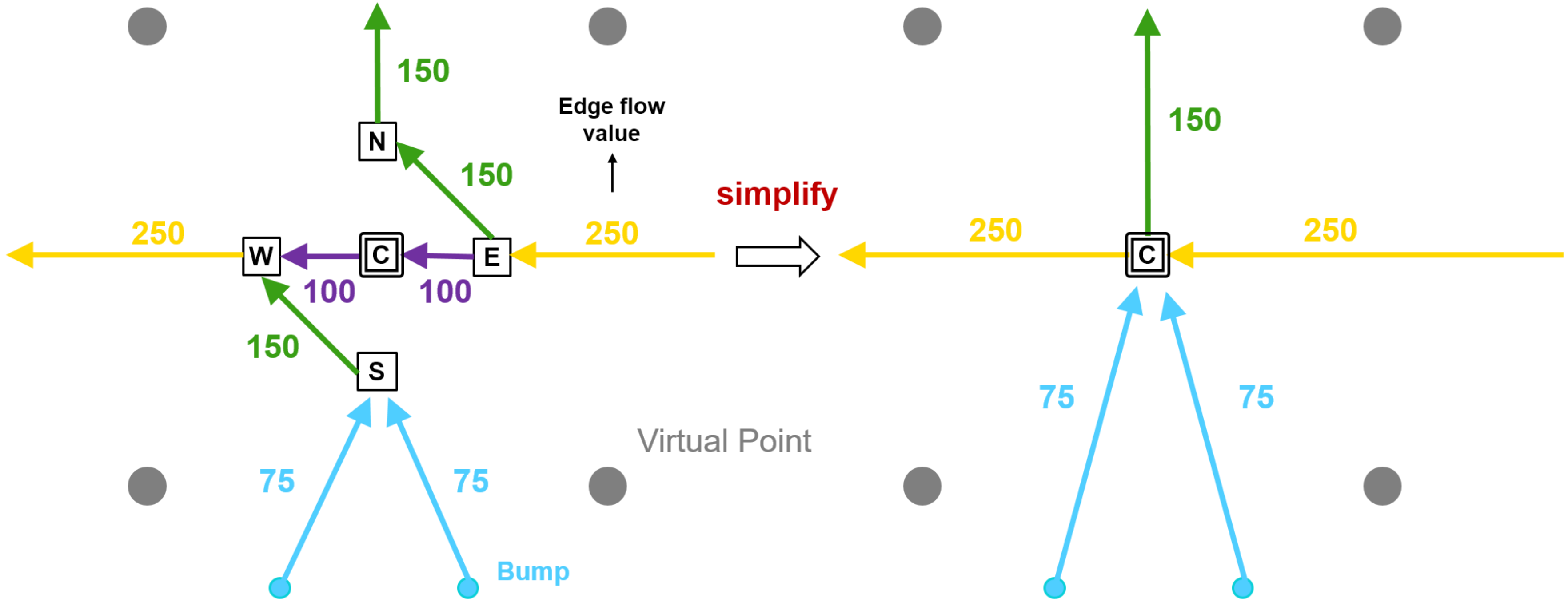
- Add the bumps to the network with the ordering determined in the pre-processing stage



MCMF-based Global Routing Derivation

Apply MCMF

- Add a super source and a super sink to the network graph
  - The super source is connected to all bumps
  - The super sink is connected to all balls
- Apply MCMF to derive the flow of each edge
- Simplify the MCMF solution to reduce the complexity
  - Only keep the cross-tile flow edges

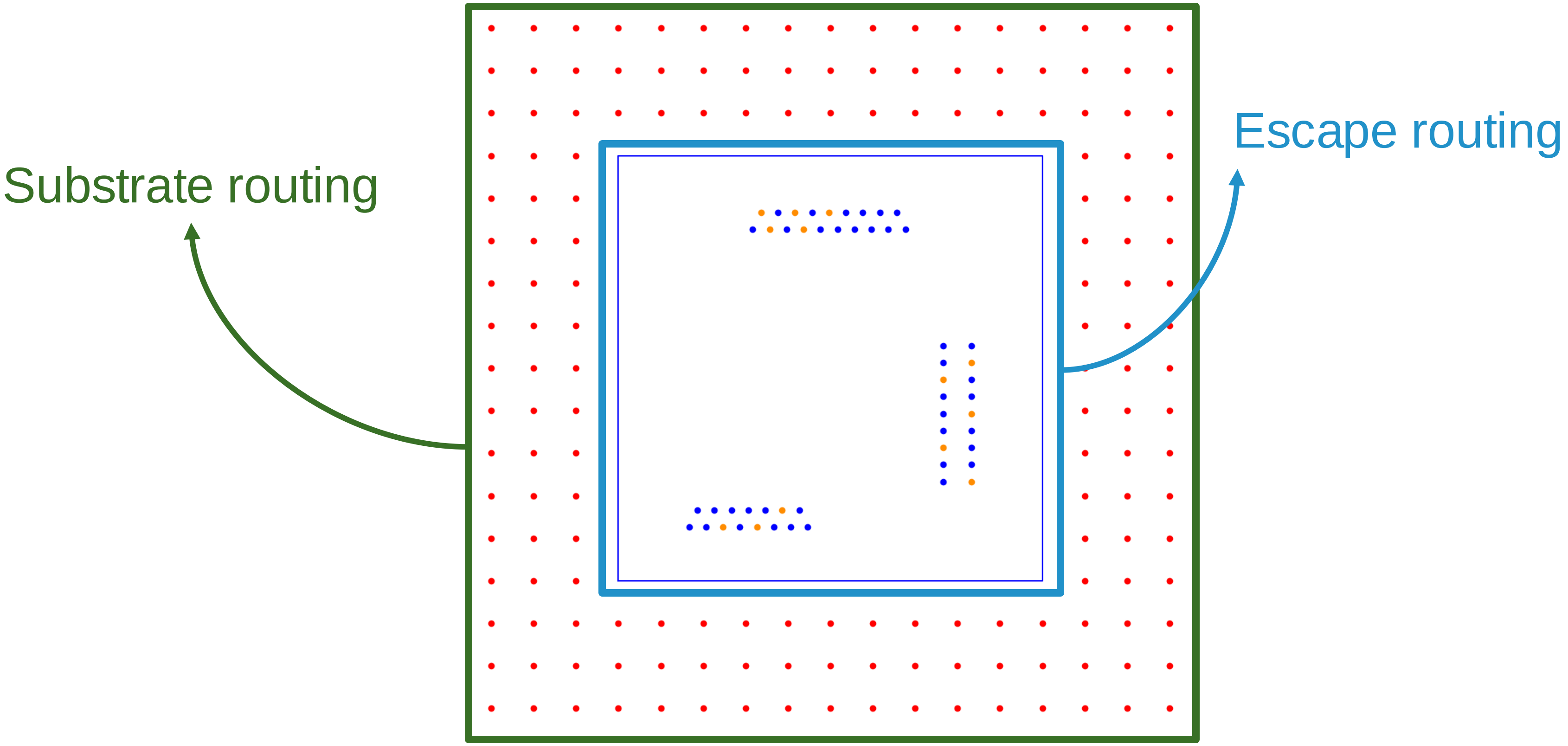


(a) MCMF result

(b) After simplify

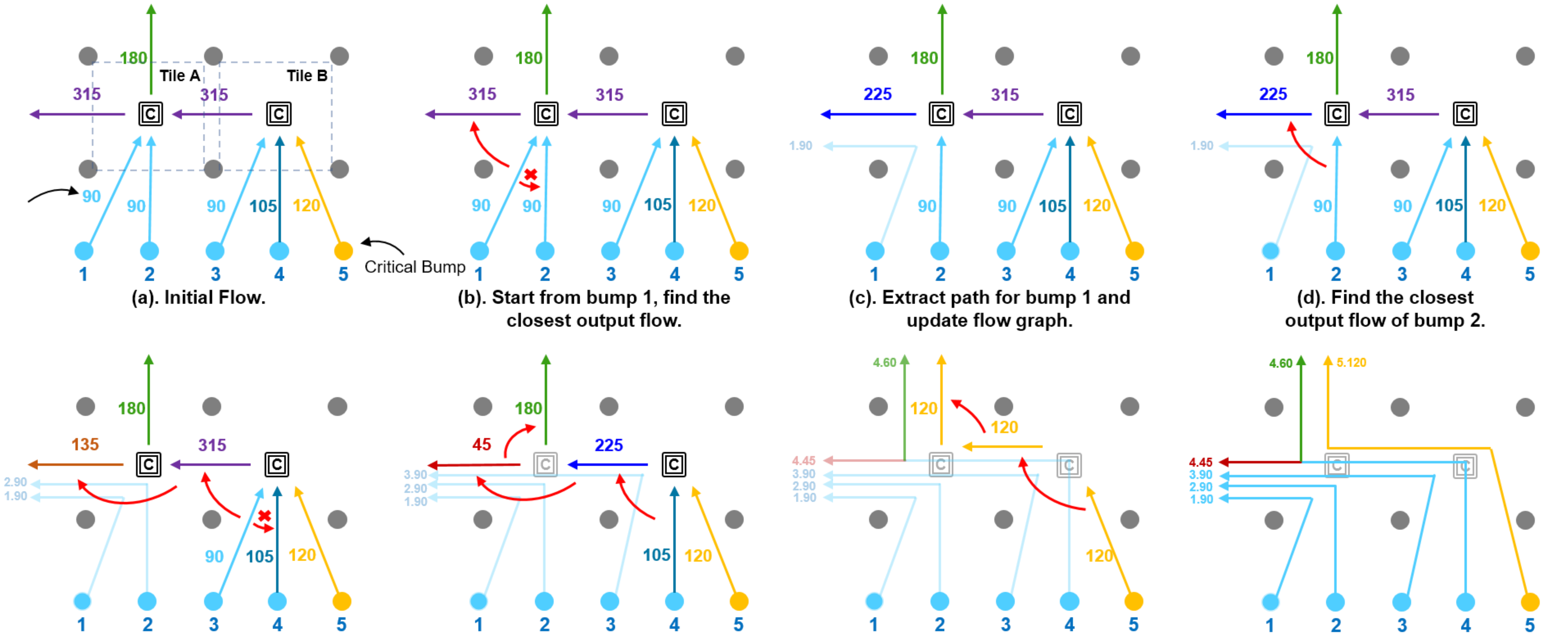
MCMF-based Global Routing Derivation

- Global routing consists of two steps
  - GR Derivation for Escape Routing
  - GR Derivation for Substrate Routing



GR Derivation for Escape Routing

- Escape routing considers the GR paths from bumps to chip boundaries



(a). Initial Flow.

(b). Start from bump 1, find the closest output flow.

(c). Extract path for bump 1 and update flow graph.

(d). Find the closest output flow of bump 2.

(e). Find the closest output flow for Tile B

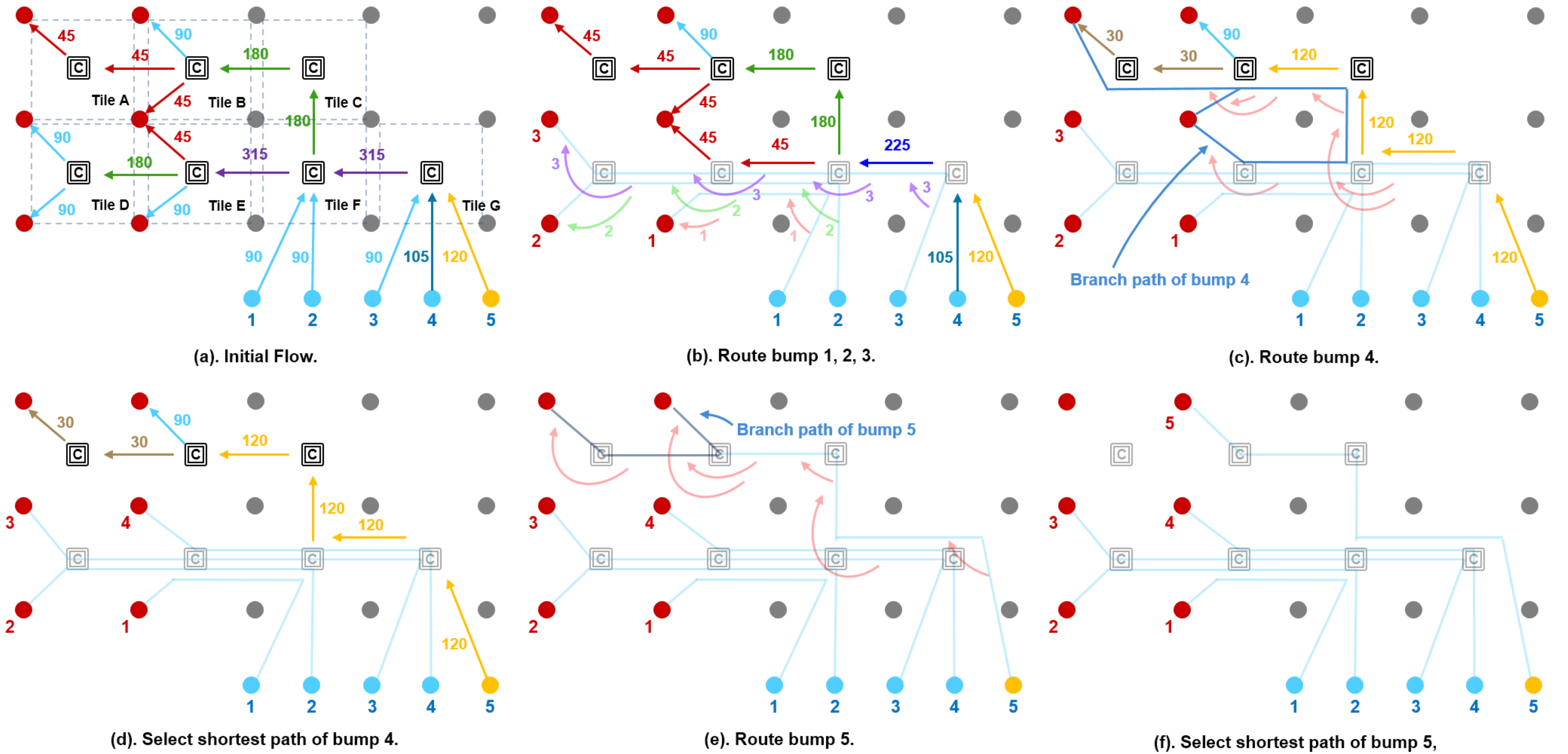
(f). Extract path for bump 3 and find the closest output flow of bump 4.

(g). Extract path for bump 4 and find the closest output flow of bump 5.

(h). Result.

GR Derivation for Substrate Routing

- Substrate routing extends the paths of escape routing to the balls



(a). Initial Flow.

(b). Route bump 1, 2, 3.

(c). Route bump 4.

(d). Select shortest path of bump 4.

(e). Route bump 5.

(f). Select shortest path of bump 5, and the final assignment result.

Detailed Routing

- Adopt a similar approach to that in [Lin et al., ASP-DAC'21]
  - The entire area is divided into grids
  - The components are mapped onto these grids
  - A* search routes the bumps
    - updating the grids after routing each bump
    - process continues until all bumps are routed

- Reserve the necessary space for all bumps in global routing
  - No design rule violations occur at this stage.

Experimental Results

Benchmark

- Two sets of benchmarks
  - Set 1: bumps are arranged in a single row.
  - Set 2: bumps are arranged in two rows.
    - #Bump (Number of Bump), #CB (Number of Critical Bump), #Chip (Number of Chips)

Case Name	Set 1		
	#Bump	#CB	#Chip
Case1	268	48	1
Case2	311	35	1
Case3	307	41	2
Case4	360	52	2
Case5	390	51	2
Case6	396	61	2
Case7	352	42	3
Case8	476	82	3
Case9	1065	167	7
Case10	3388	573	7

Case Name	Set 2		
	#Bump	#CB	#Chip
Case11	57	20	1
Case12	222	75	1
Case13	354	108	3
Case14	380	129	2
Case15	472	150	4
Case16	856	270	4
Case17	1528	470	4
Case18	2392	736	4

Experimental Results

- Compare the results with the extended [Yan and Wong, TCAD'12]
  - Routed (Routability), UN (number of Unrouted Net), WL (WireLength), SWL (Scaled WL)

	[Yan and Wong, TCAD'12]					Our Model			
	Routed	UN	WL	SWL	Time	Routed	UN	WL	Time
Case1	87%	35	>367787	422743	47	100%	0	421530	81
Case2	92%	26	>526177	571931	68	100%	0	560018	137
Case3	91%	29	>331604	364400	24	100%	0	366759	23
Case4	88%	45	>430855	489607	39	100%	0	500110	36
Case5	90%	38	>441303	490336	43	100%	0	488791	127
Case6	90%	40	>480585	533983	113	100%	0	514302	81
Case7	82%	64	>496408	605375	111	100%	0	614695	374
Case8	83%	83	>674837	813056	68	100%	0	840909	84
Case9	90%	110	>1144542	1271713	137	100%	0	1265298	177
Case10	86%	470	>2896512	3368037	546	100%	0	3401829	318
Comp	0.87	-	-	1.00	0.85	1.00	-	1.00	1.00

Analysis of DP Ordering

- Analysis of the effectiveness of the proposed DP ordering.
  - Routed (Routability), UN (Number of Unrouted Net), SWL (scaled WL)

Case Name	With Net Ordering				Without Net Ordering			
	Routed	UN	WL	Time	Routed	UN	SWL	Time
Case11	100%	0	70301	3	98%	1	70598	4
Case12	100%	0	351002	18	91%	21	407784	49
Case13	100%	0	412555	26	92%	27	466900	95
Case14	100%	0	607515	31	94%	22	593317	78
Case15	100%	0	573769	35	92%	40	624266	123
Case16	100%	0	1047083	65	90%	83	1235965	386
Case17	100%	0	1877131	135	90%	152	2357943	732
Case18	100%	0	2944009	254	91%	221	3349184	1028
Comp	1.00	-	1.00	1.00	0.92	-	1.15	4.40

Conclusion

Conclusion

- The proposed algorithm effectively handles the paired-spacing constraint
- Experimental results show that our approach achieves 100% routability in all cases and outperforms [Yan and Wong, TCAD'12] with a 13% improvement in routability
- Experimental results demonstrate the effectiveness of the proposed dynamic programming-based bump reordering algorithm

Q&A