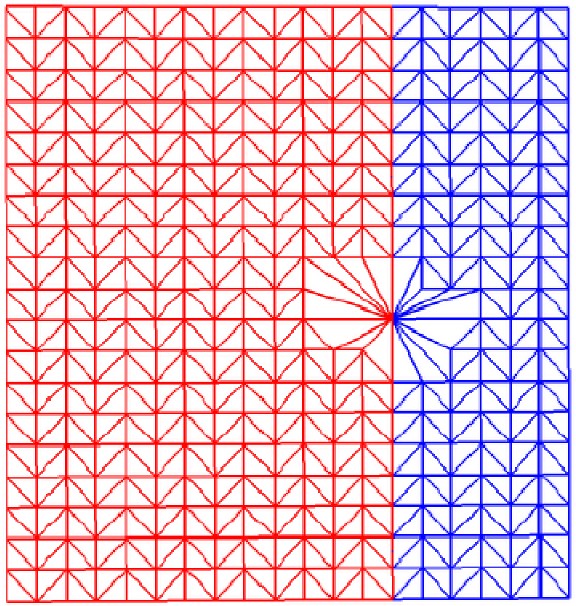
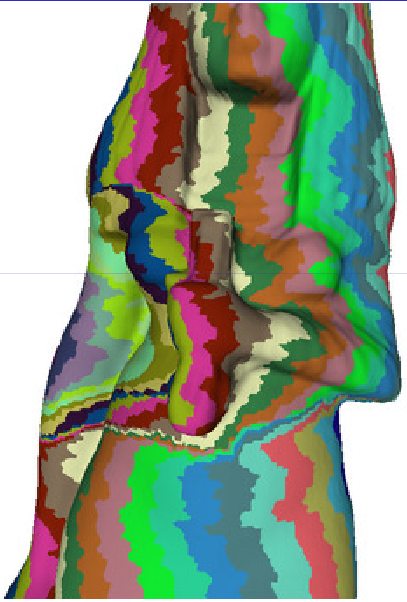
Geometry Engine Optimization: Cache Friendly Compressed Representation of Geometry



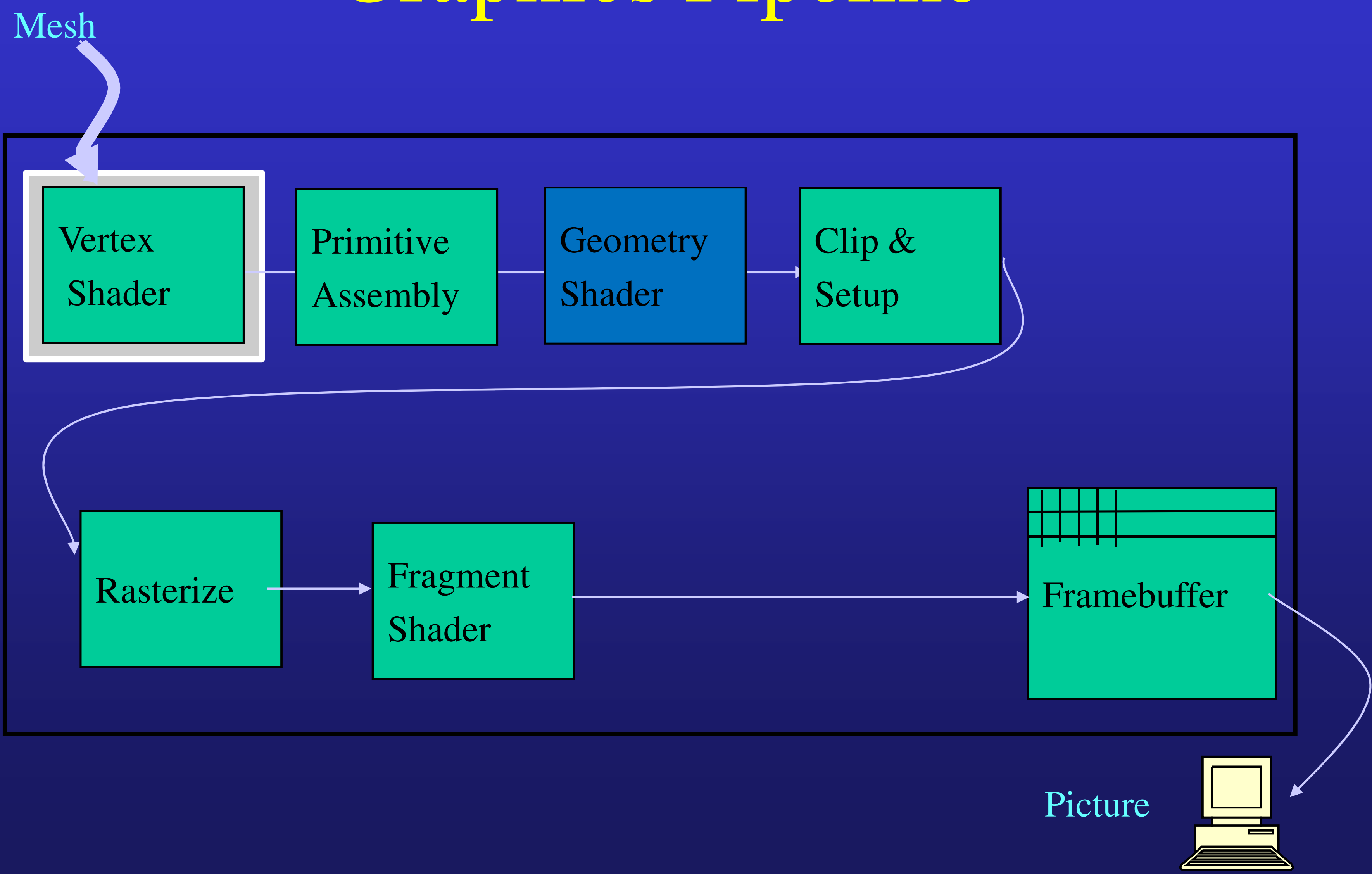


Jatin Chhugani and Subodh Kumar

Intel Corporation, CA

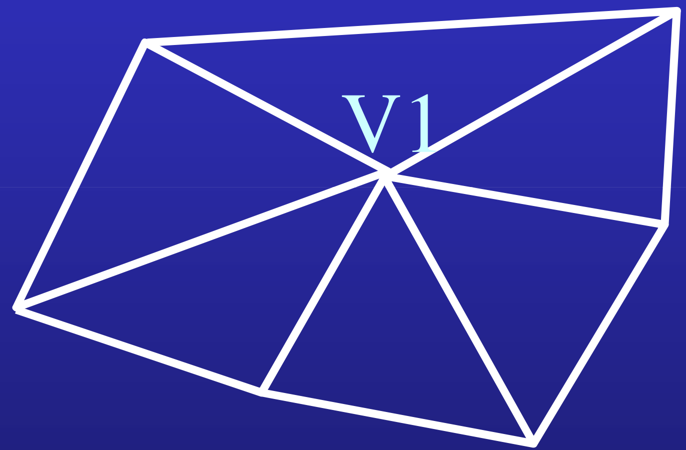
I.I.T., Delhi, India

Graphics Pipeline



Triangle List Representation

$V1_x$ $V1_y$ $V1_z$
$V2_x$ $V2_y$ $V2_z$
$V3_x$ $V3_y$ $V3_z$



m triangles => 3*m* vertices

Triangle List Representation

$V1_x$ $V1_y$ $V1_z$ N1 T1 U1 ..
$V2_x$ $V2_y$ $V2_z$ N2 T2 U2 ..
$V3_x$ $V3_y$ $V3_z$ N3 T3 U3 ..

-
-

m triangles => 3*m* vertices

Adjacency List Representation

$V1_x$ $V1_y$ $V1_z$
$V2_x$ $V2_y$ $V2_z$
$V3_x$ $V3_y$ $V3_z$
$V4_x$ $V4_y$ $V4_z$
$V5_x$ $V5_y$ $V5_z$
•
•
•

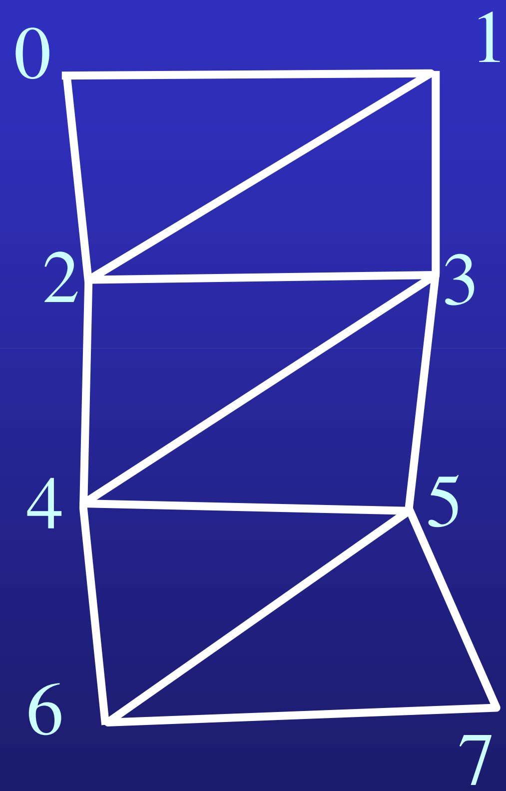
$i1_0$ $i1_1$ $i1_2$
$i2_0$ $i2_1$ $i2_2$
$i1_0$ $i1_1$ $i1_2$
•
•

m triangles => n vertices, $3m$ indices

Triangle Strip Representation

$V1_x$ $V1_y$ $V1_z$
$V2_x$ $V2_y$ $V2_z$
$V3_x$ $V3_y$ $V3_z$
$V4_x$ $V4_y$ $V4_z$
$V5_x$ $V5_y$ $V5_z$
•
•
•

0 1 2
1 2 3
2 3 4
•
•



m triangles => n vertices

Triangle Strip Representation?

1

$V1_x\ V1_y\ V1_z$

$V2_x\ V2_y\ V2_z$

$V3_x\ V3_y\ V3_z$

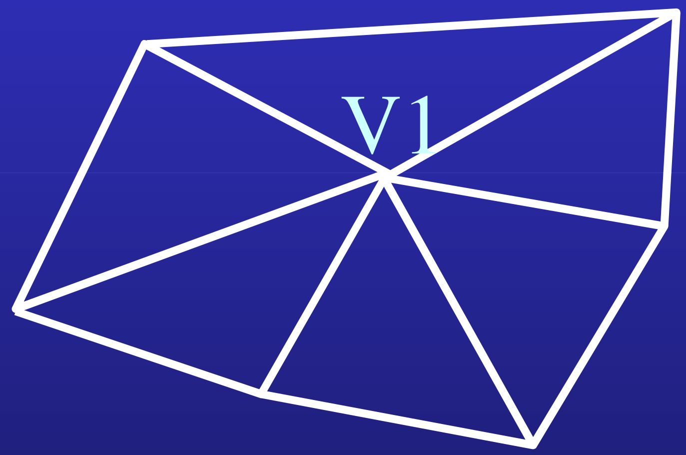
$V4_x\ V4_y\ V4_z$

$V5_x\ V5_y\ V5_z$

•

•

•



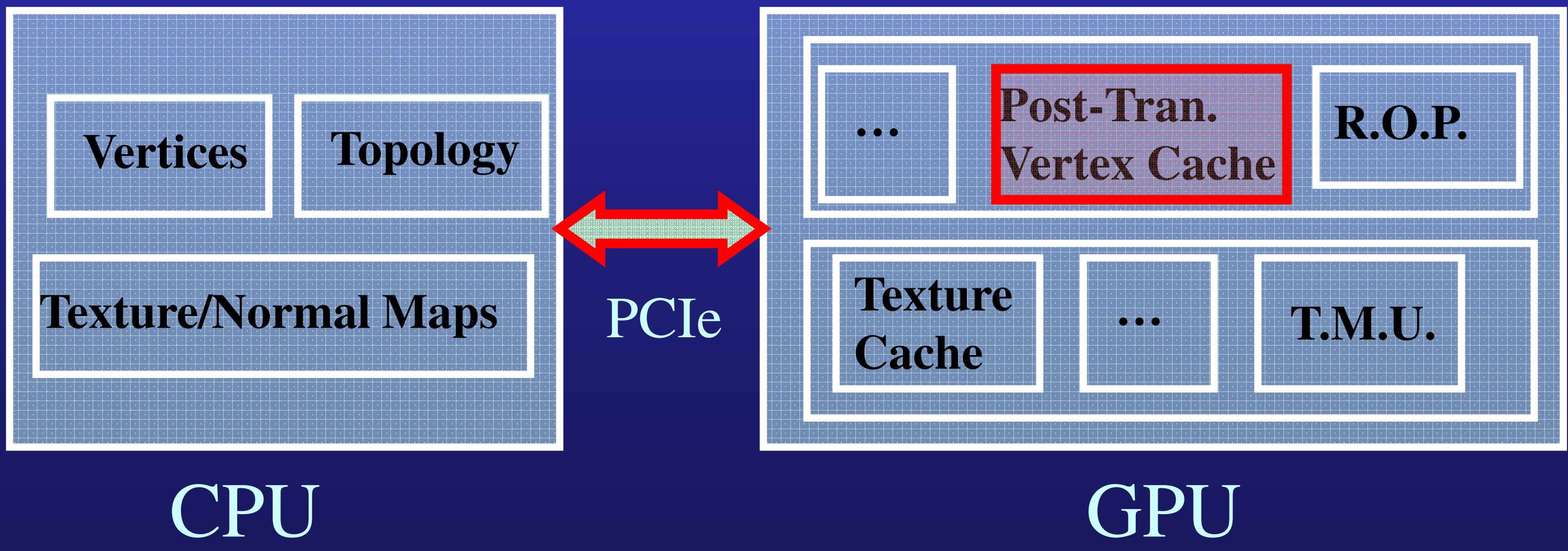
m triangles => *n* vertices

Relationship between *m* and *n*

- Euler's relation:
  - #V + #F – #E = 2
- #V + #F – (3/2)#F = 2
  - #V ≈ ½ #F
- *n* is approximately half of *m*
  - *m* triangles imply 3*m* indices
  - An index is re-used about 6 times

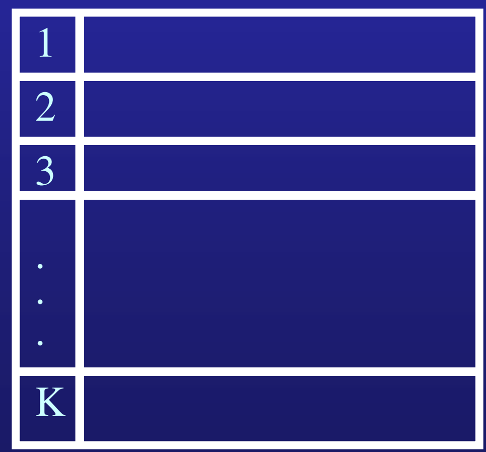
Vertex Shading Review

- Triangle mesh is
  - a set of Vertices
  - Each vertex has attributes
  - Set of Triangles (topology information)
    - Each triangle indexes *three* of the input vertices



Post-Transform Vertex Cache

- Cache results of vertex shading
- Helps reuse computation and reduce b/w
- Typical cache sizes are between 8 and 32 entries
- FIFO cache replacement policy



Post-Transform Vertex Cache

- Triangles reuse vertices
  - Average degree of a vertex is ~6
  - Would like to transform a vertex *once*
    - ~85% reduction in vertex computation
- Specify triangles in an order that exploits the cache
  - Vertex reuse is clustered in the order

Reorder triangles to maximize cache utilization

Geometry Specification

- Huge Meshes
  - Hundreds of megabytes to store and transfer
- Bus bandwidth and video memory size are bottlenecks
- Need to *compress both* the vertex data and the topology
  - Hardware supported
- Topology compression is required as well
  - Lossless compression to preserve the mesh structure
  - Little hardware support in current GPU's

Compress input geometry with efficient hardware decompression and minimal hardware changes

Problem Statement

"Cache Friendly Compressed Representation of Input Geometry with efficient decompression"

- high compression of topology
- high cache coherence
- Inexpensive and Efficient decompression hardware
- Minimal API change
- Friendly to vertex attribute compression
  - not a subject of this paper

Some Previous Work

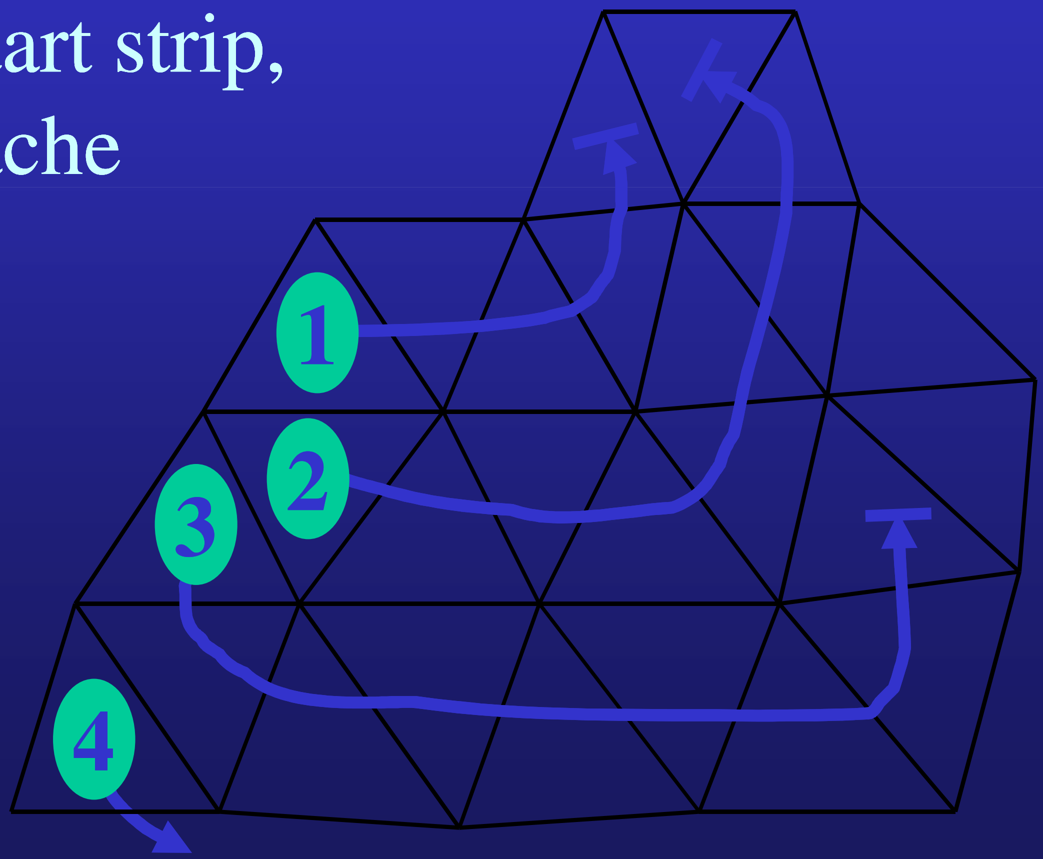
- Compression – Deering/Chow
- Cache simulation and optimization – Hoppe et al.
- Stack Buffer – BarYehuda-Gotsman
- Other work to just compress topology – Edgebreaker
- Cache oblivious work – Yoon et al.
- Single Strip triangulation work – Gopi

Compressed Stream [Deering..]

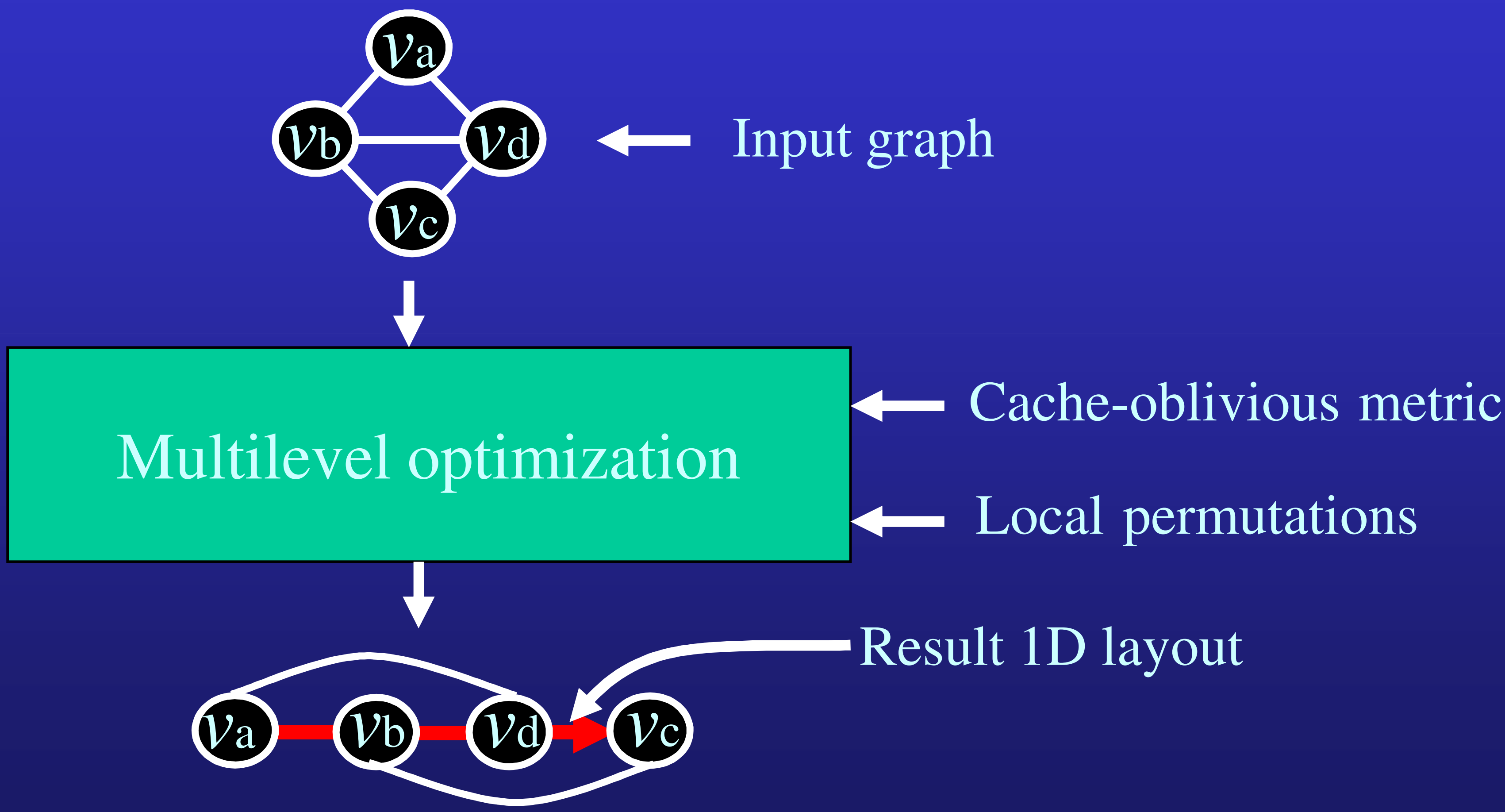
- Turn meshes into a stream of data and instructions
- ‘Generalized’ Mesh
- Include special LOAD instructions
  - Restart, Replace Oldest, Replace Middle
  - Push into mesh-buffer
  - Control which index goes into the buffer and which is evicted

Greedy strip-growing [Hoppe]

To decide when to restart strip, perform look-ahead cache simulation



Cache Oblivious Layout [Yoon..]

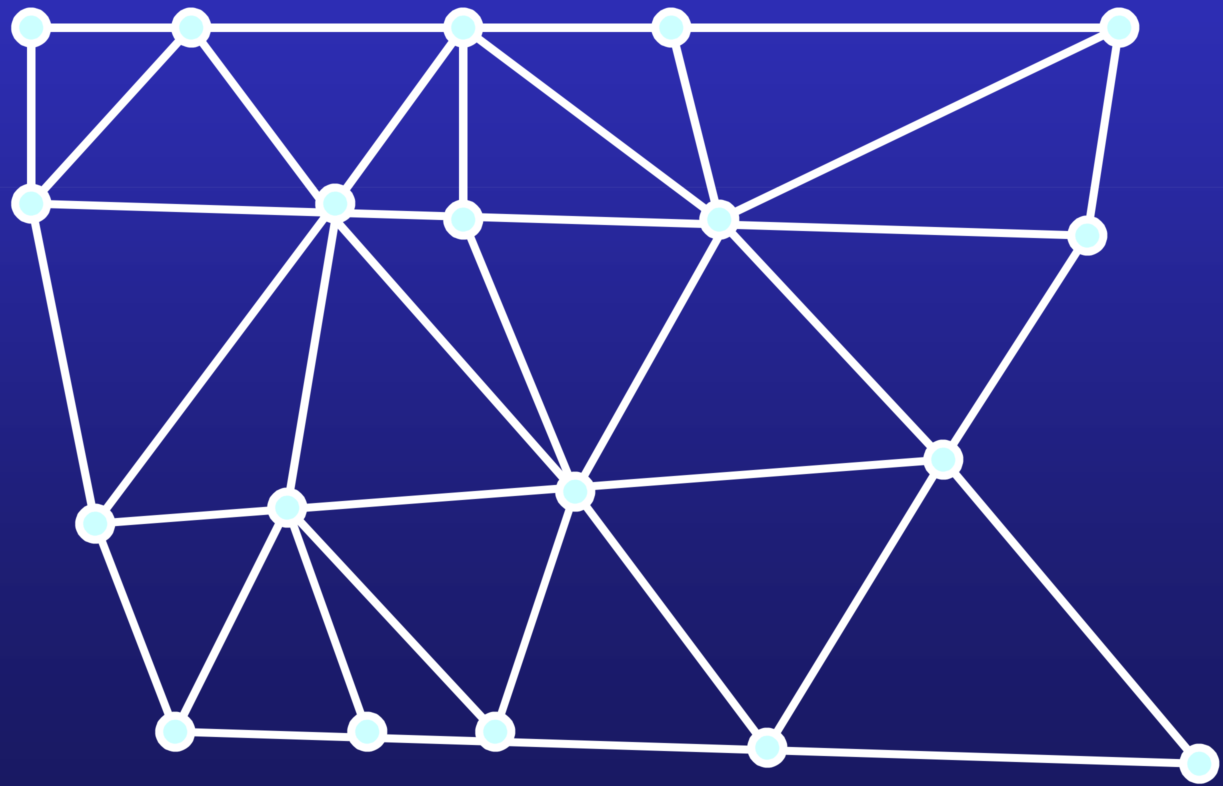


Our Approach

- *n* vertices imply at least *n* cache misses
  - Minimize the use of a vertex when not in cache
- Visit all triangles adjacent to a vertex before it is evicted
  - Cannot guarantee for every vertex
  - A vertex is ‘hit’ only for a fixed number of cache misses (FIFO)
- Directly re-order the vertices, rather than triangles
  - Connectivity of the vertices dictates the triangle order to follow

Illustration

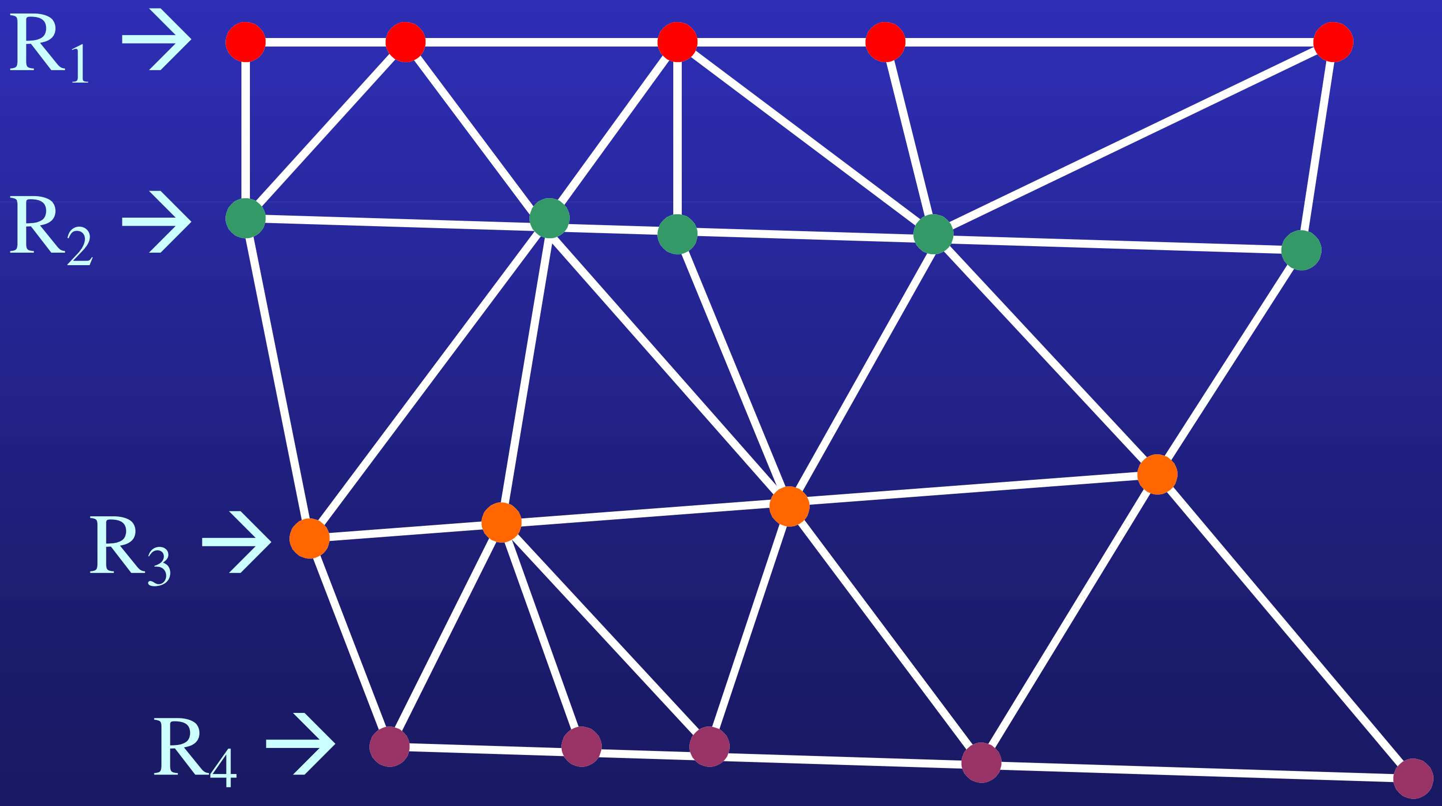
Input Mesh with 19 vertices and 22 triangles



Illustration

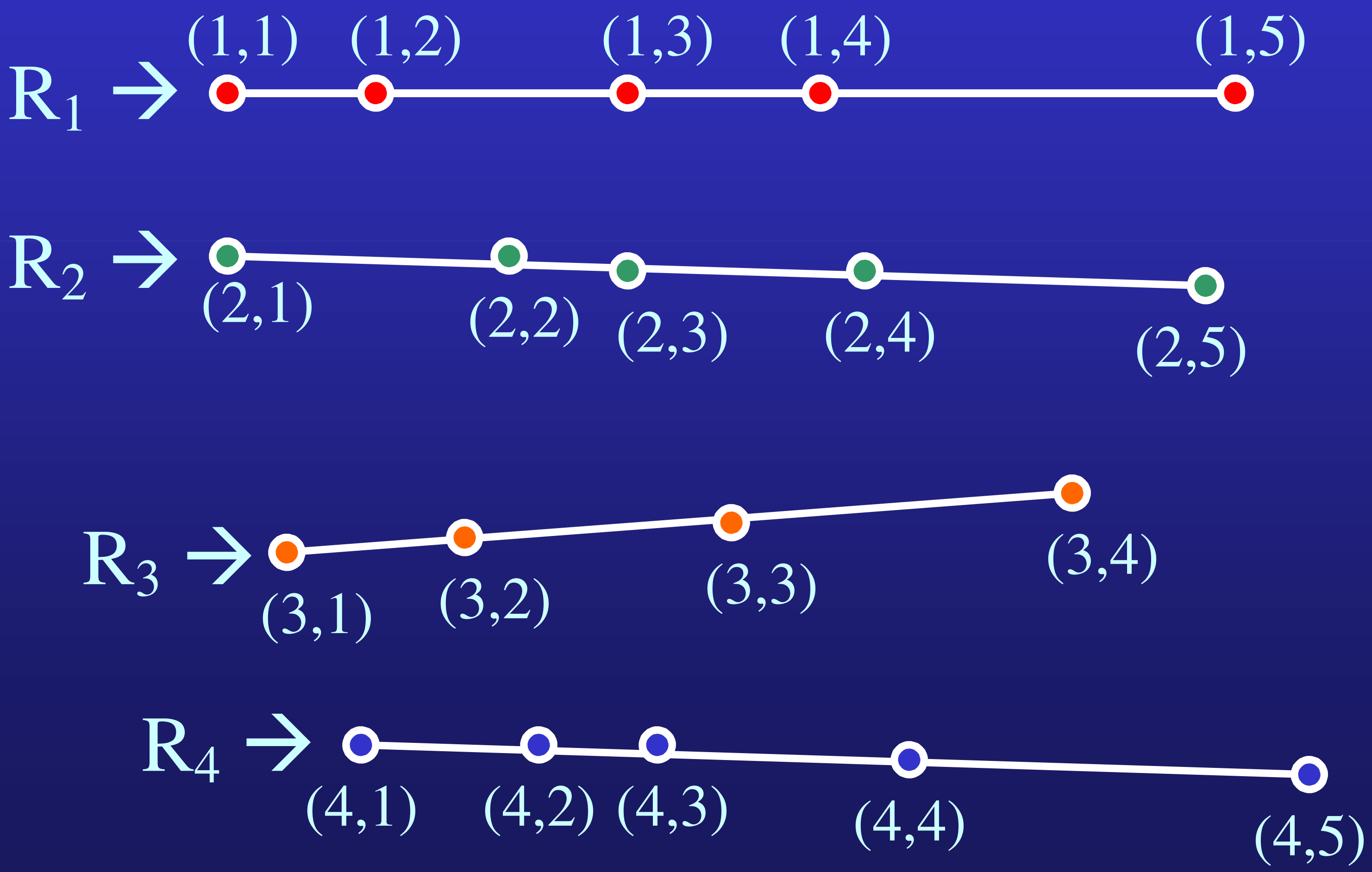
Step 1:Divide the mesh into rows (chains) of vertices

→ Triangles exist only between consecutive chains



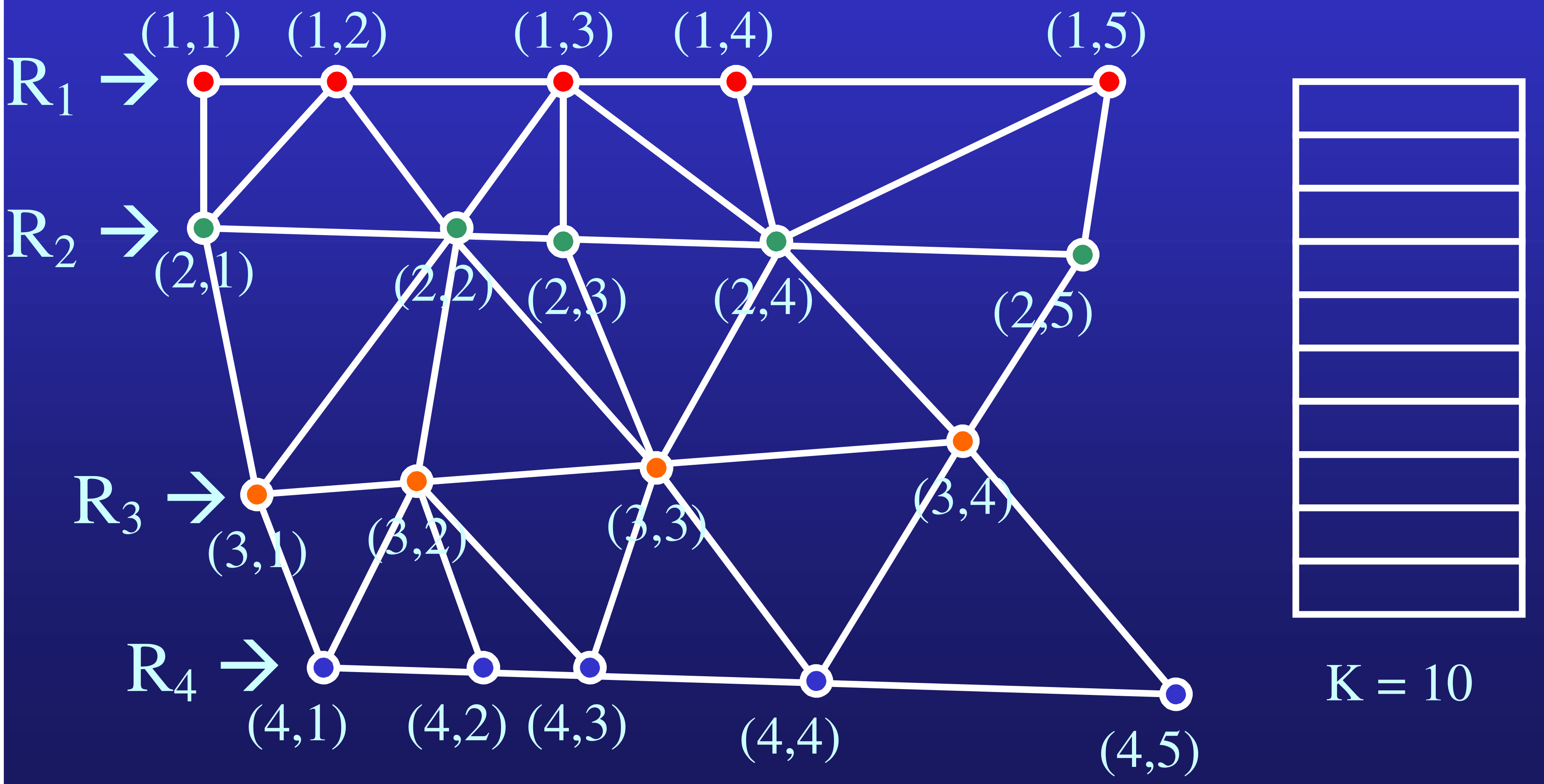
Illustration

Step 2: Order Vertices within every chain



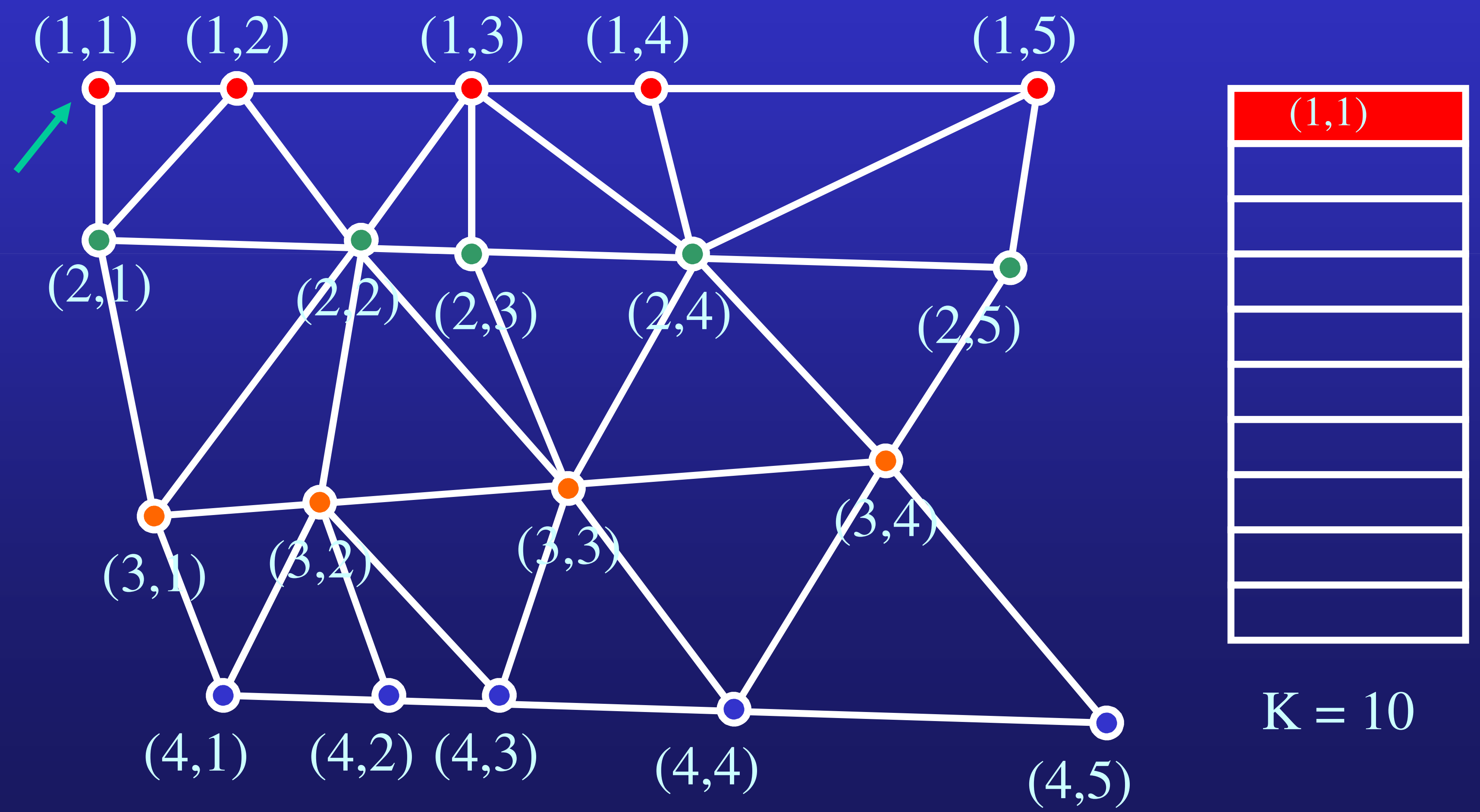
Illustration

Step 3: Render triangles in an order that introduces vertices in R1 before R2 and so on, in the order specified within each row



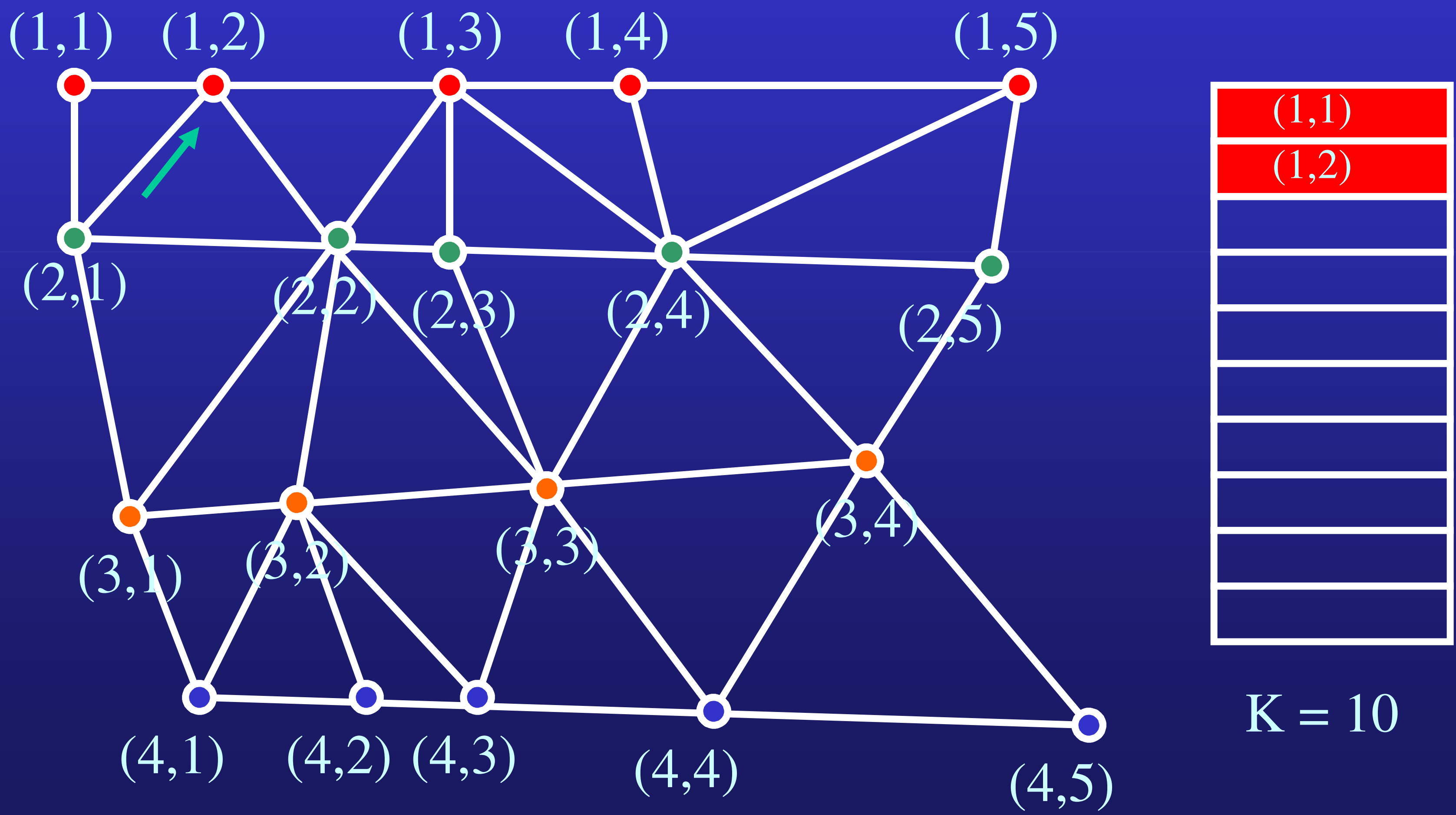
Illustration

Render Degenerate Triangle → [$V_{(1,1)}$ $V_{(1,1)}$ $V_{(1,1)}$]



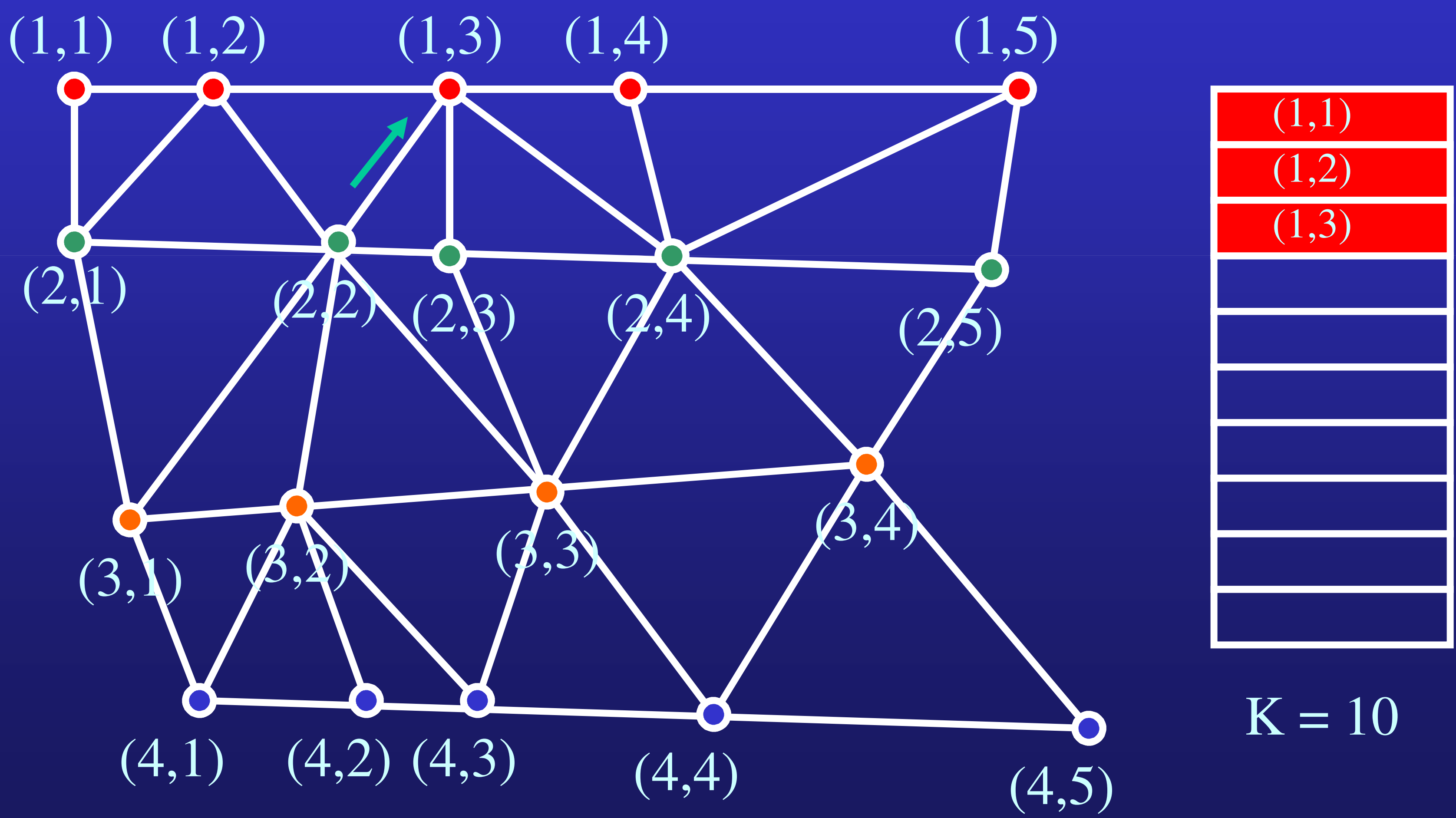
Illustration

Render Degenerate Triangle → [$V_{(1,2)}$ $V_{(1,2)}$ $V_{(1,2)}$]



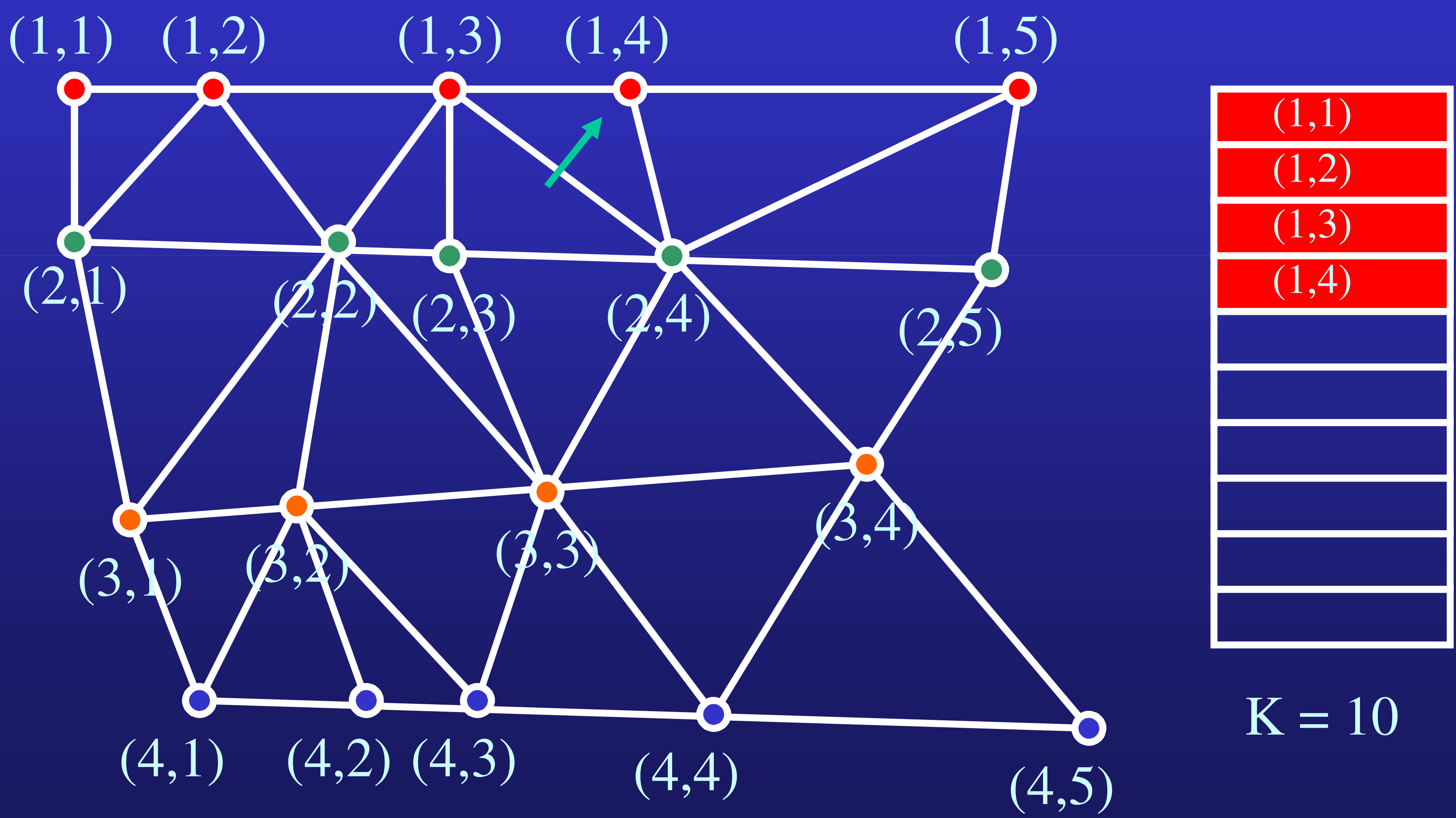
Illustration

Render Degenerate Triangle → [$V_{(1,3)}$ $V_{(1,3)}$ $V_{(1,3)}$]



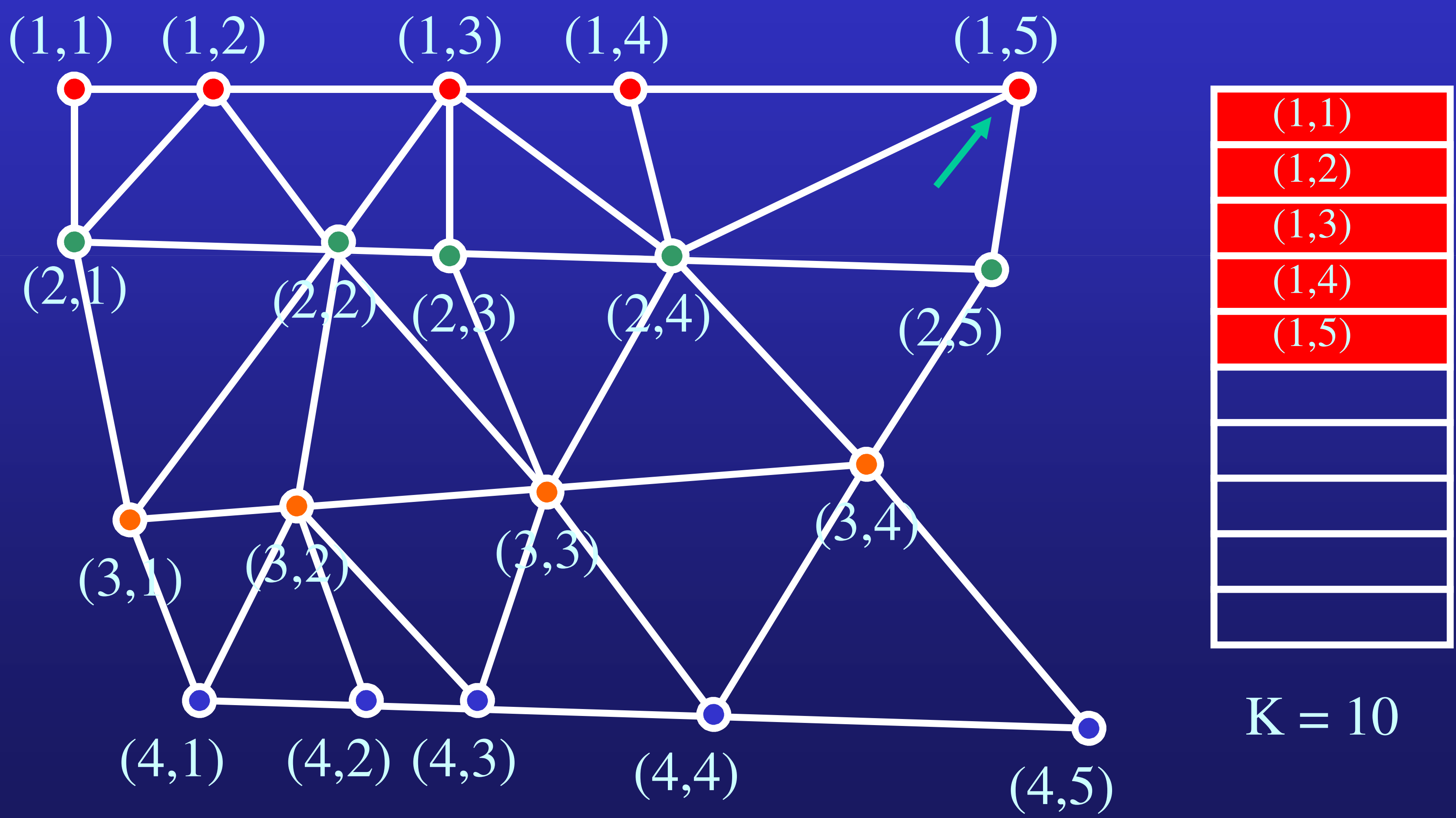
Illustration

Render Degenerate Triangle → [$V_{(1,4)}$ $V_{(1,4)}$ $V_{(1,4)}$]



Illustration

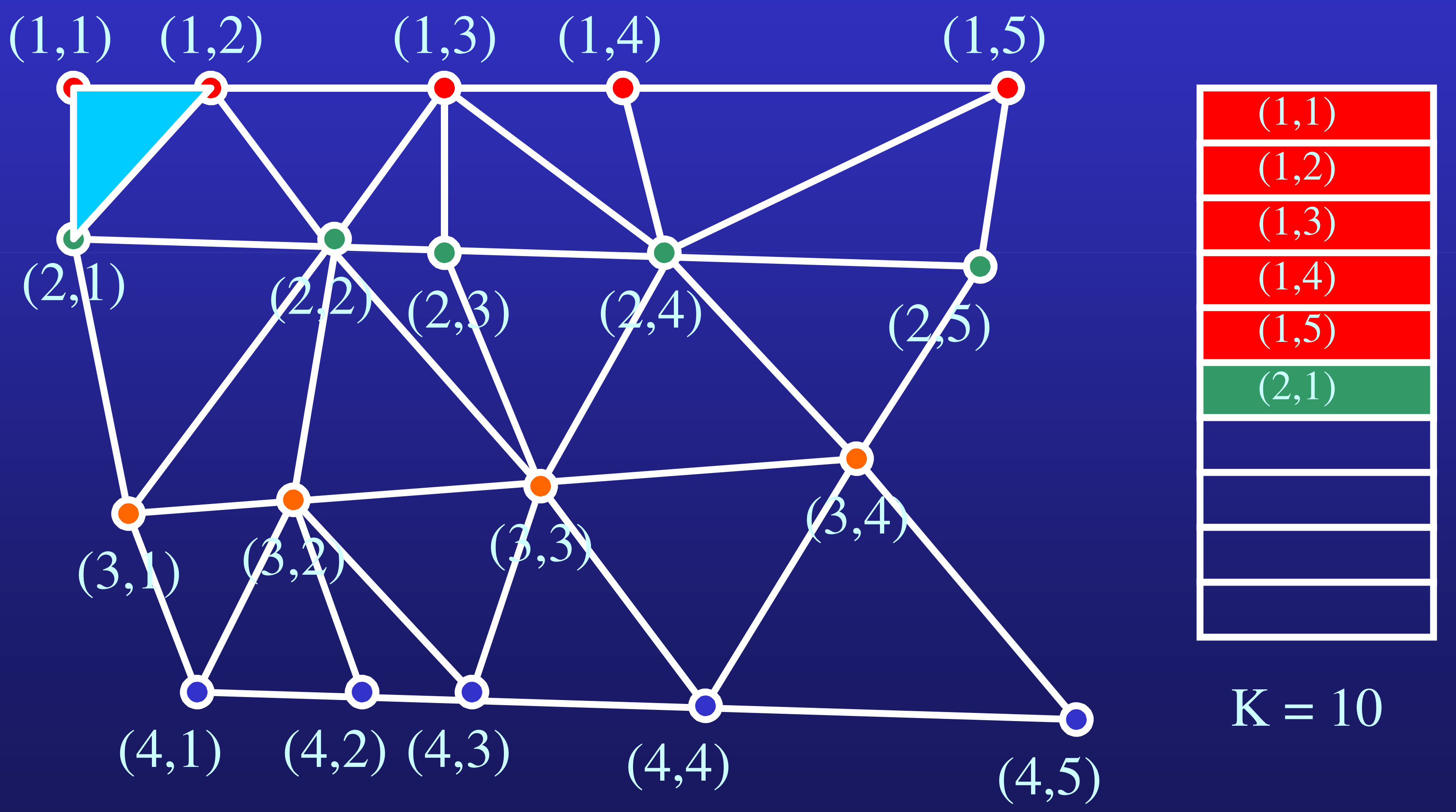
Render Degenerate Triangle → [$V_{(1,5)}$ $V_{(1,5)}$ $V_{(1,5)}$]



K = 10

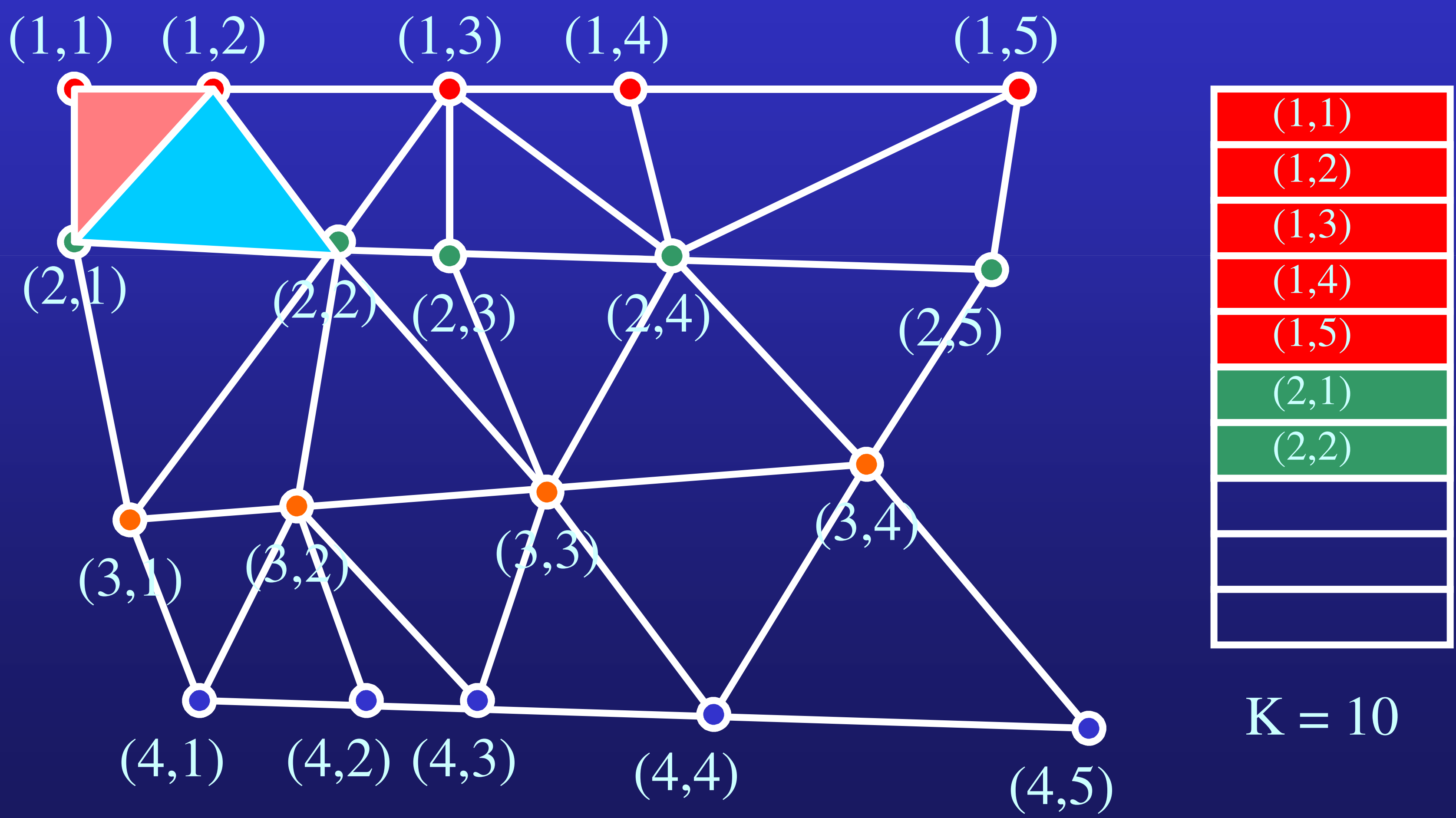
Illustration

Render Triangle → [$V_{(1,1)}$ $V_{(1,2)}$ $V_{(2,1)}$]



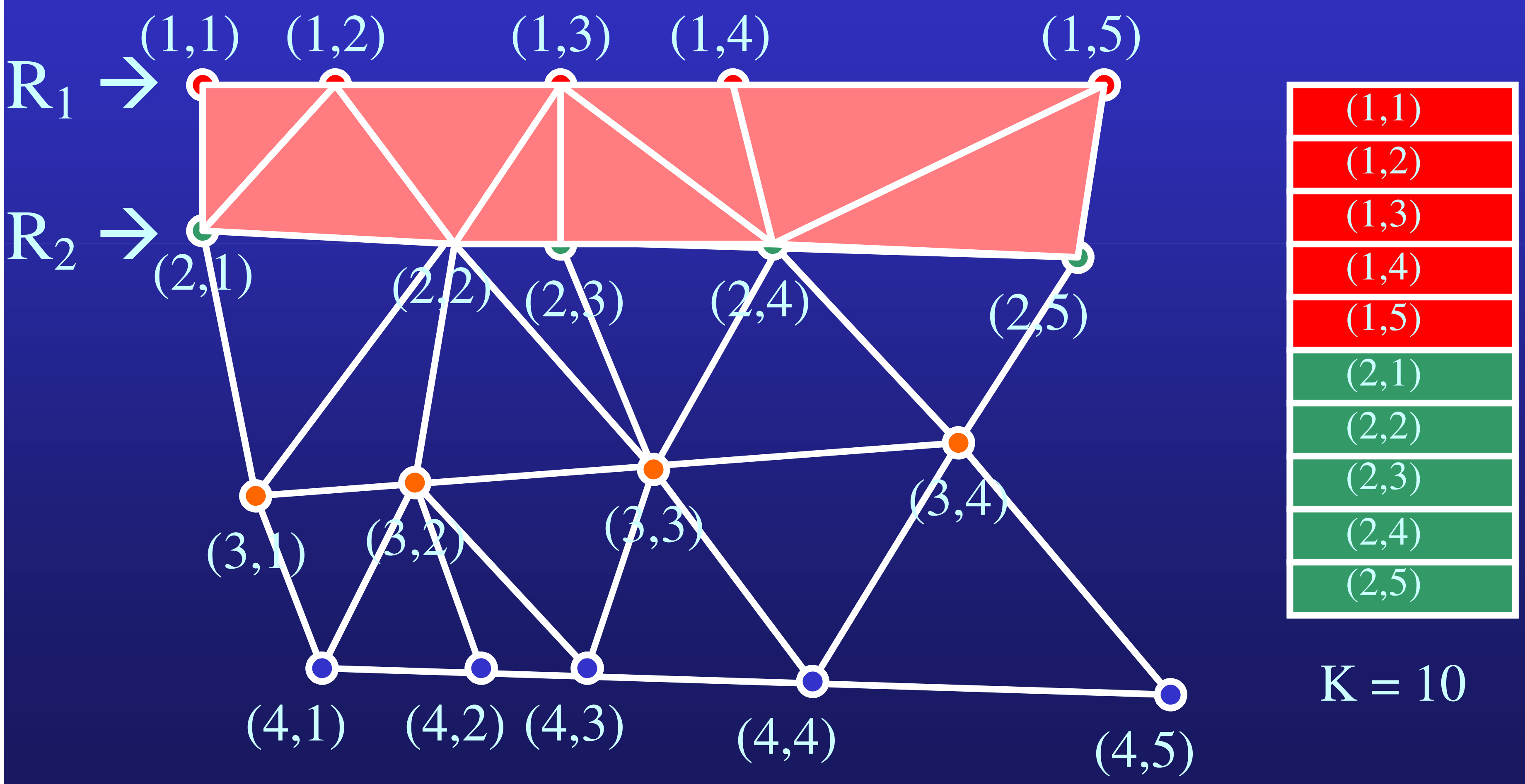
Illustration

Render Triangle → [$V_{(1,2)}$ $V_{(2,1)}$ $V_{(2,2)}$]



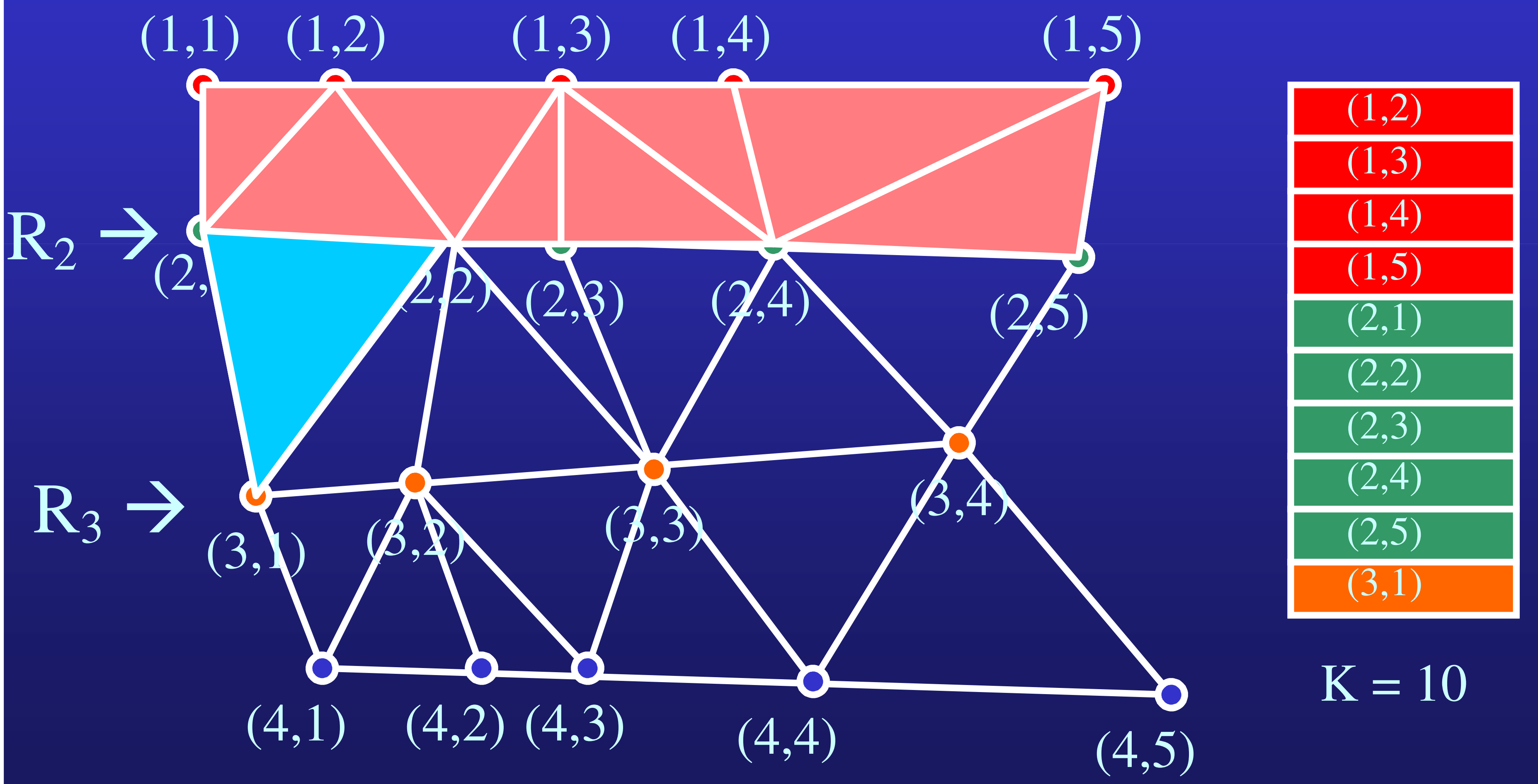
Illustration

Render triangles between R_1 and R_2



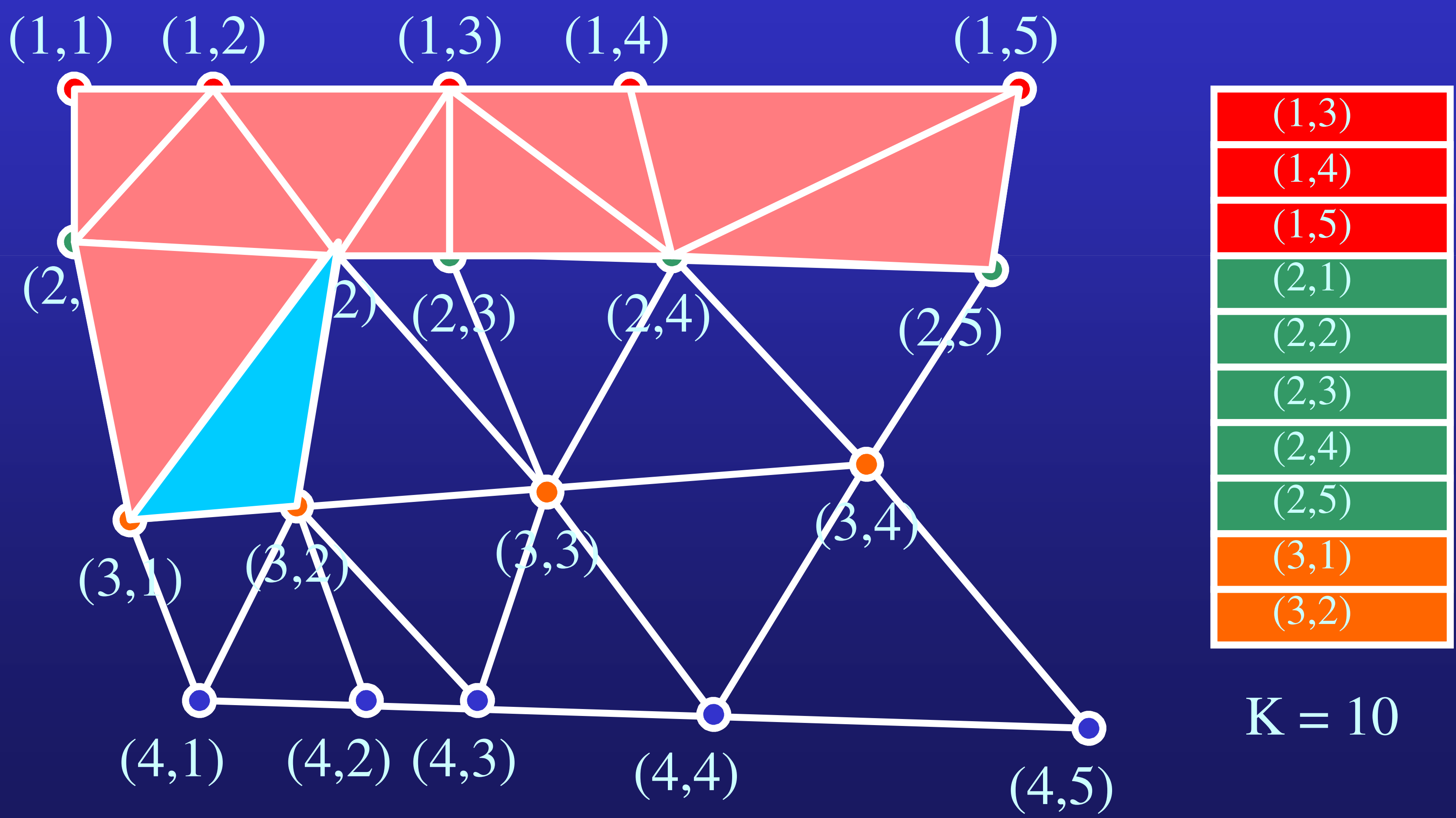
Illustration

Render triangle → [$V_{(2,1)}$ $V_{(2,2)}$ $V_{(3,1)}$]



Illustration

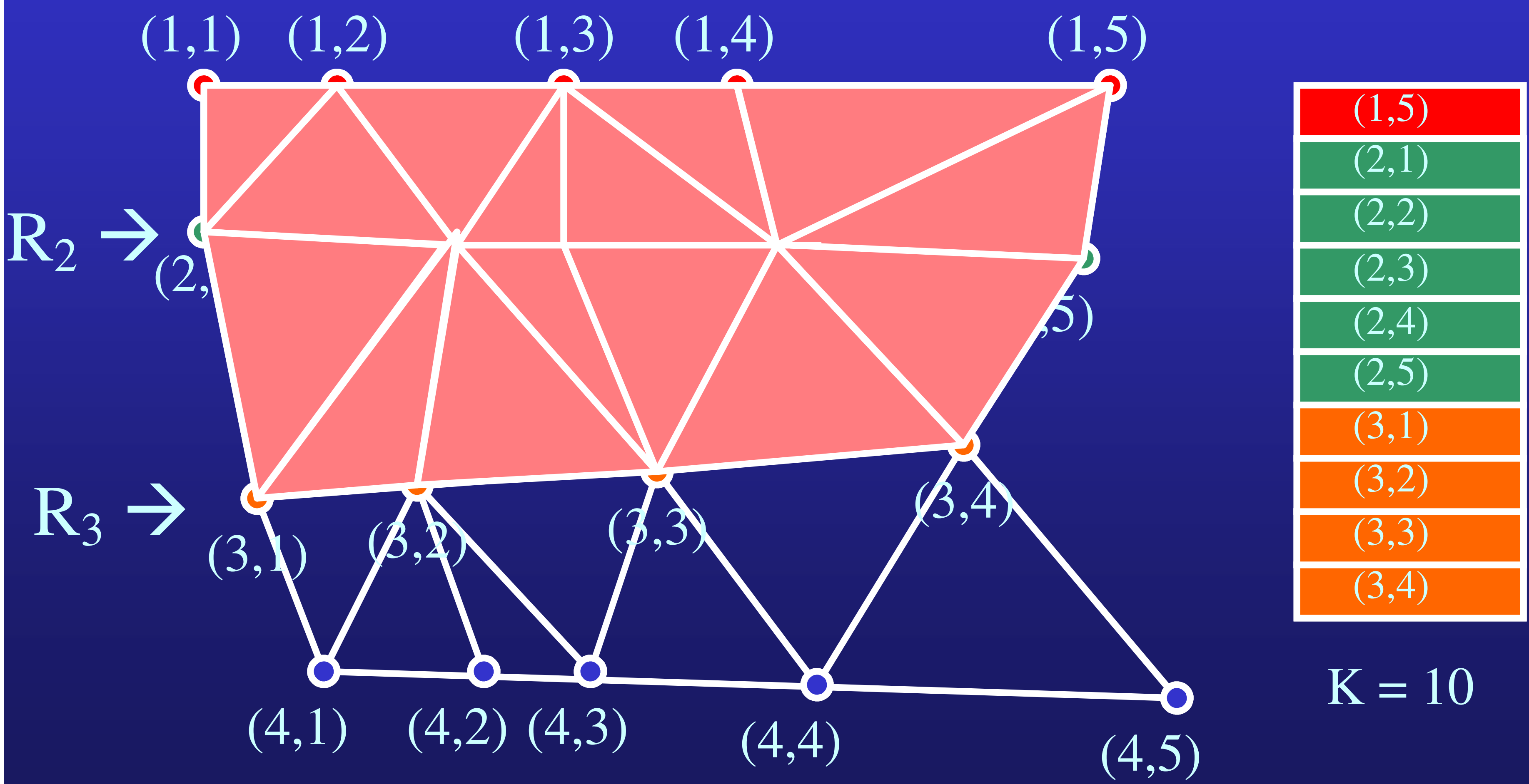
Render triangle → [$V_{(2,2)}$ $V_{(3,1)}$ $V_{(3,2)}$]



K = 10

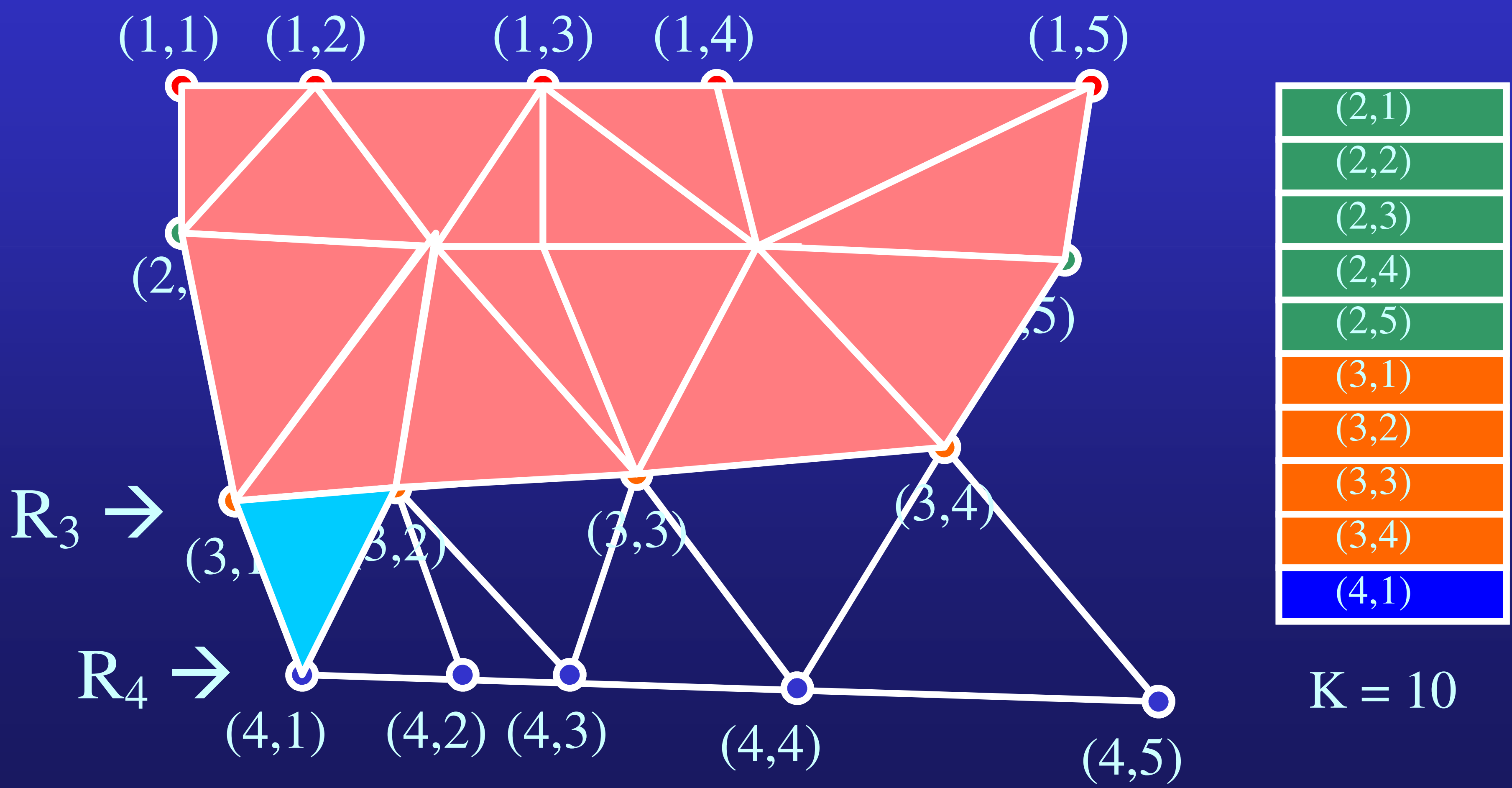
Illustration

Render triangles between R_2 and R_3



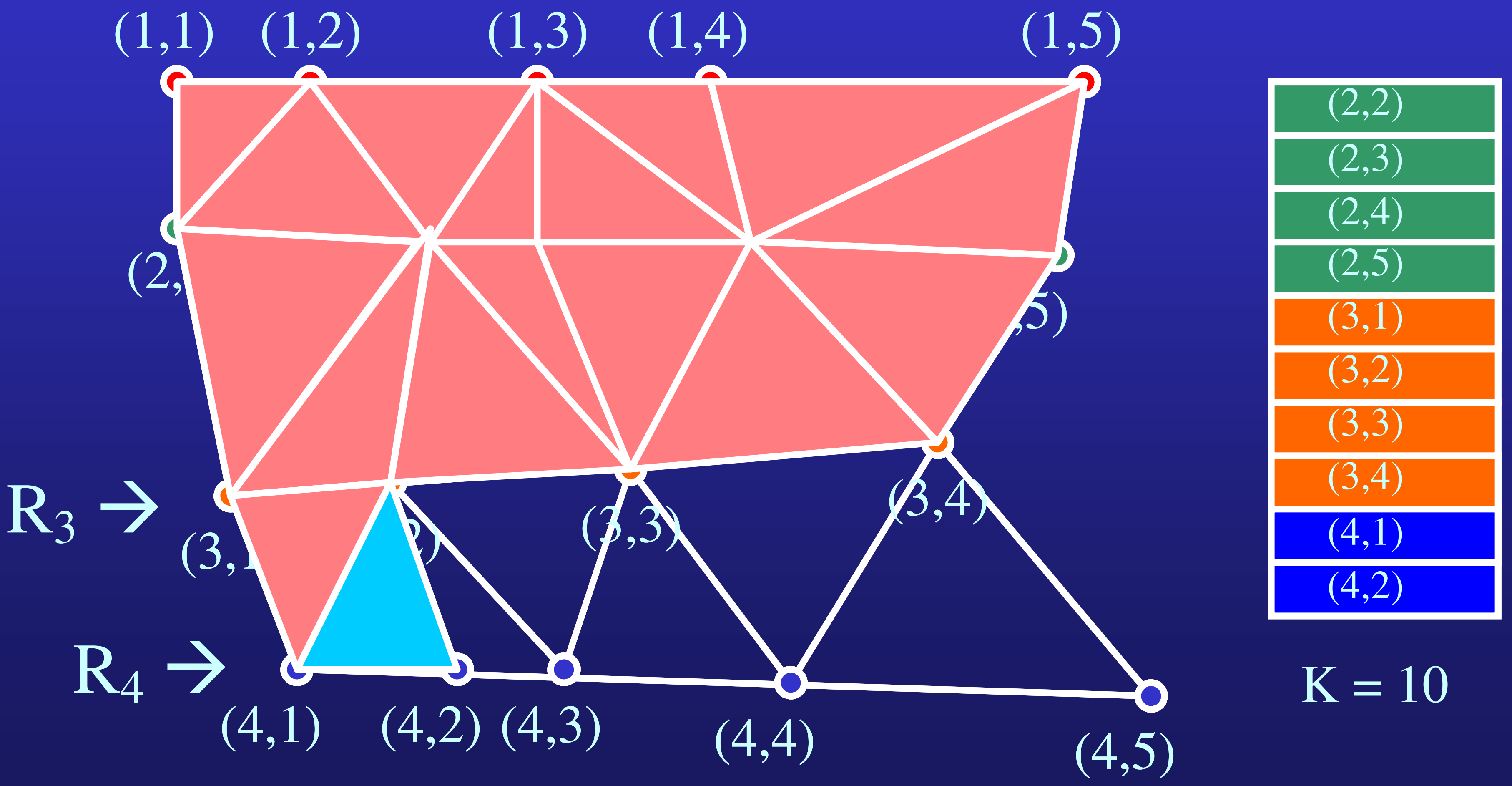
Illustration

Render Triangle → [$V_{(3,1)}$ $V_{(3,2)}$ $V_{(4,1)}$]



Illustration

Render Triangle → [$V_{(3,2)}$ $V_{(4,1)}$ $V_{(4,2)}$]



Illustration

Render triangles between R_3 and R_4

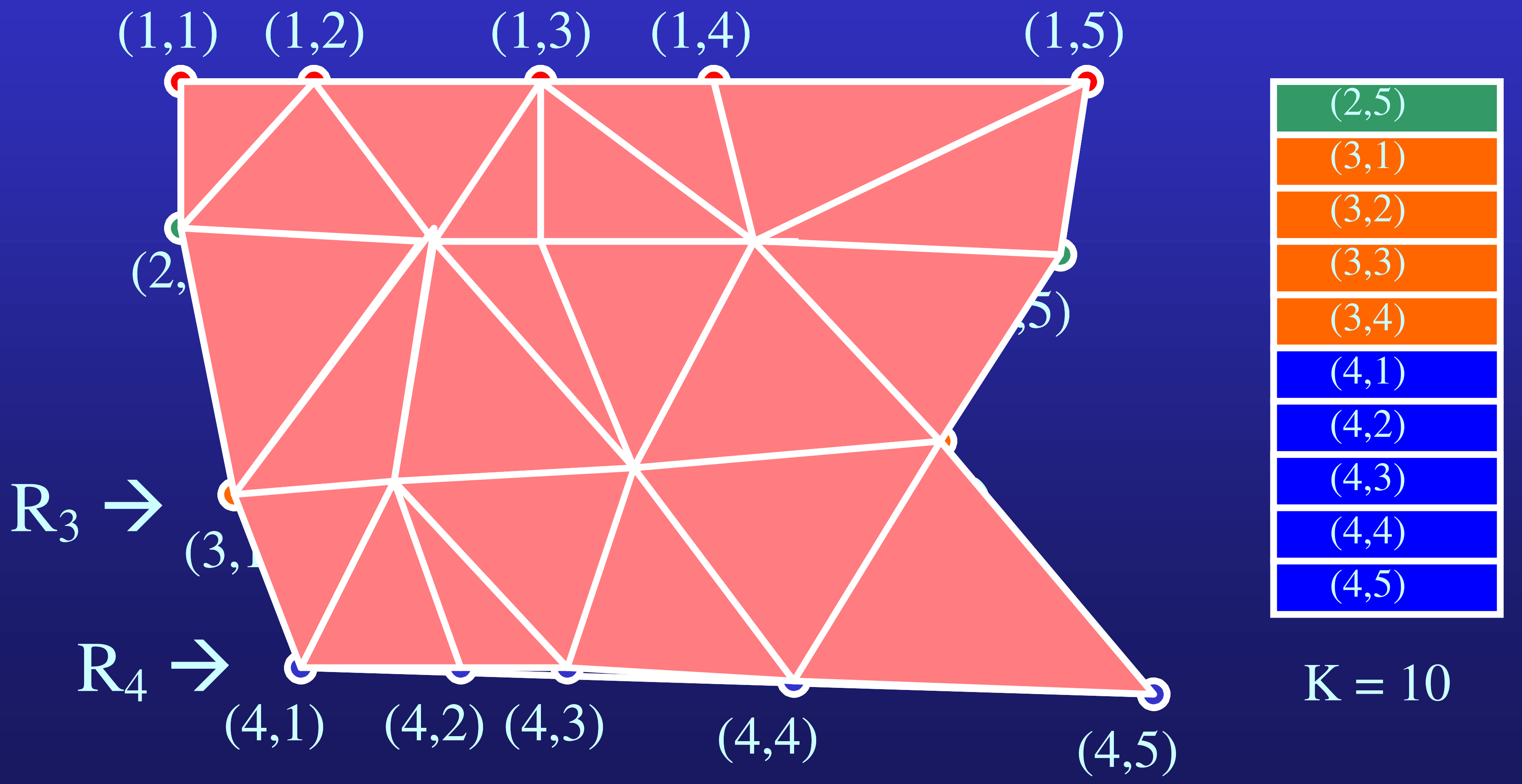


Illustration (contd.)

- Each vertex was loaded *only once* into the cache
  - Optimal solution
- Generated some degenerate triangles to warm-up the cache
- However, we allowed a potentially large cache!
- What happens when the cache size (K) is not that large?
  - Choose a subset of vertices along each chain
  - Each subset is referred to as a *cut*.
  - Vertices shared by two cuts need to be reloaded

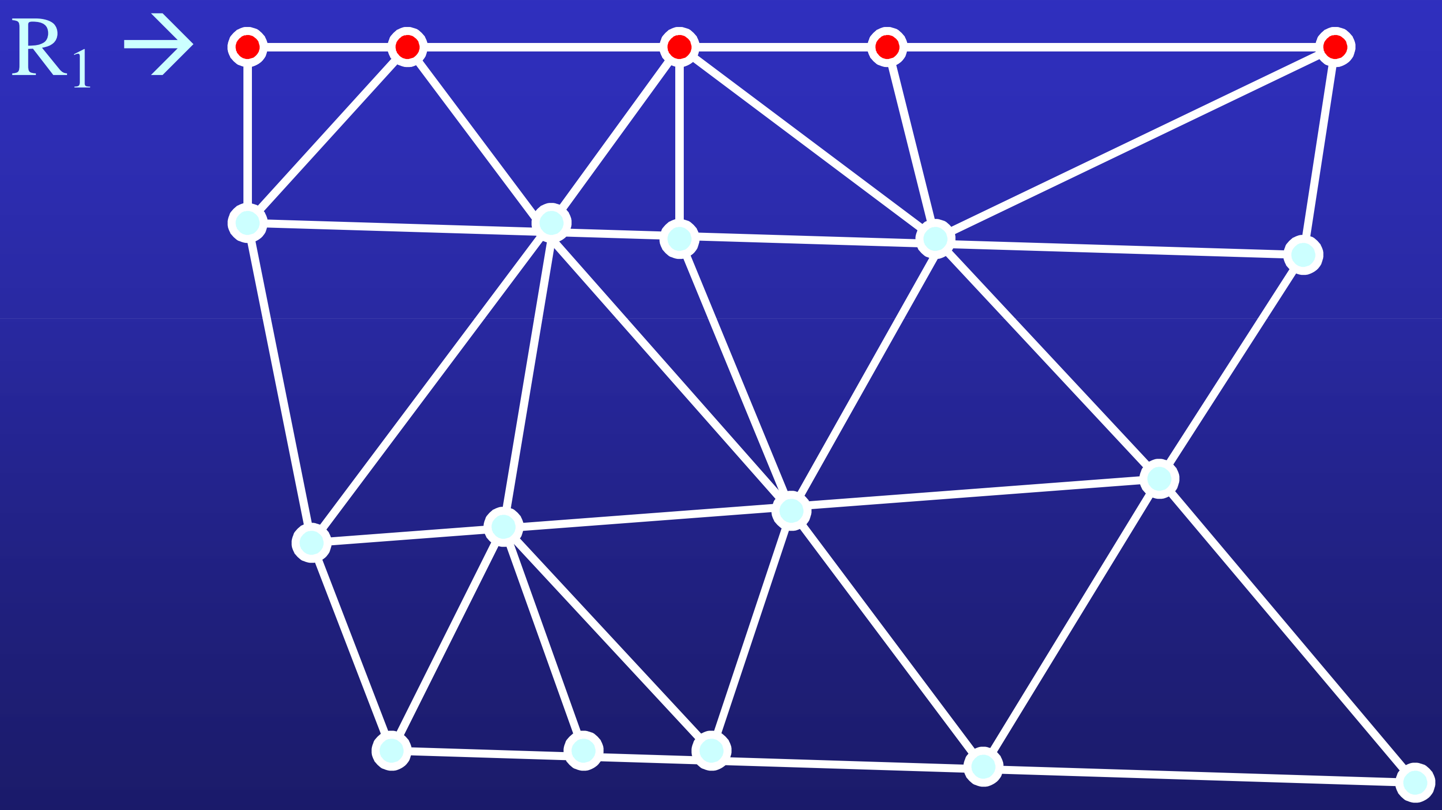
Algorithm Overview

1. Form chains (rows of vertices) for the mesh
2. Order the vertices within each chain
3. Form cuts of vertices for each row
   - Function of connectivity and cache size K
4. Form list of triangles that preserves the vertex order

1. Forming Chains On The Mesh

- Given a mesh, choose a subset of vertices that form a connected path
  - can choose any single vertex as well
  - denote this set as R_1

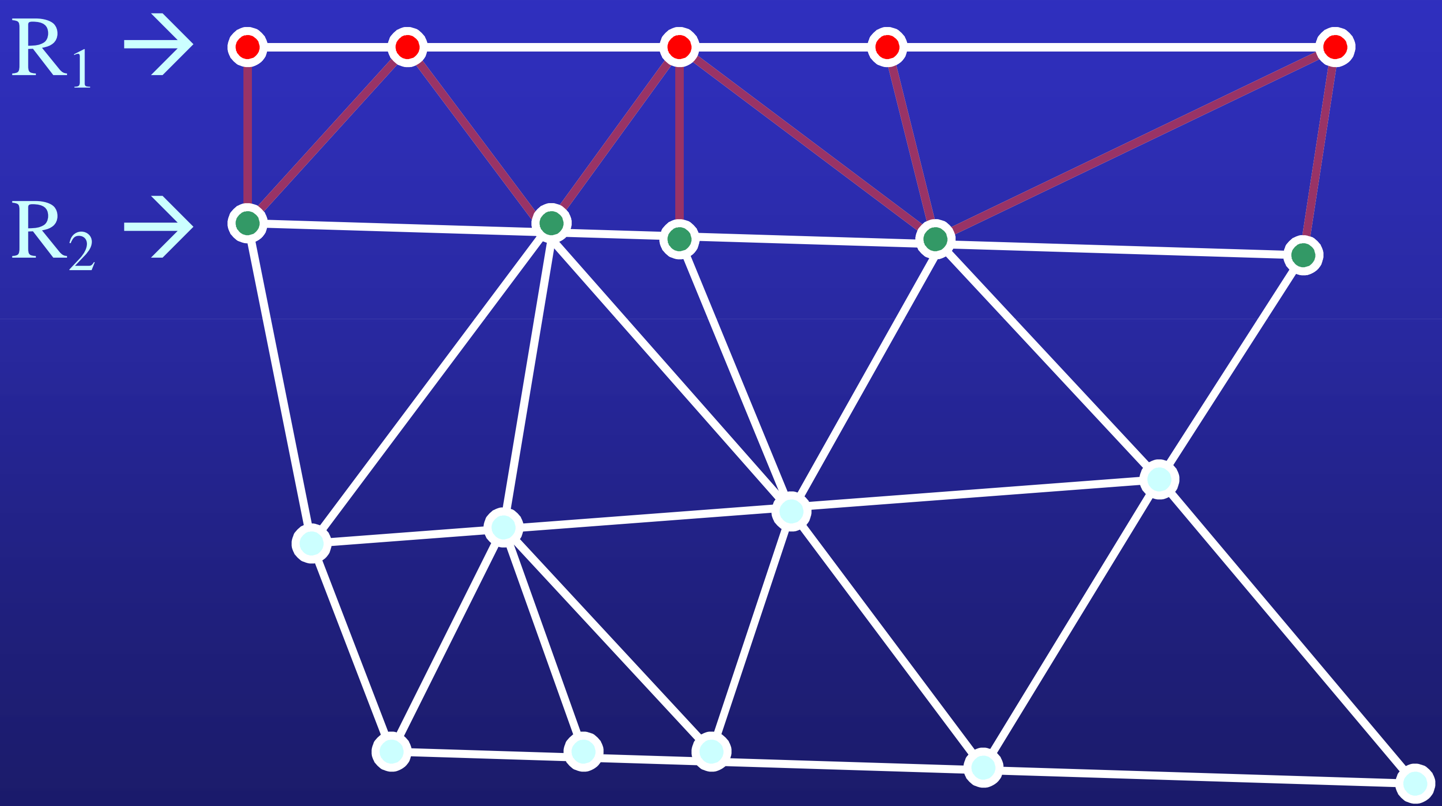
1. Forming Chains On The Mesh



1. Forming Chains On The Mesh

- Given a mesh, choose a subset of vertices that form a connected path
  - can choose any single vertex as well
  - denote this set as R_1
- Perform a Breadth First Search and find all vertices connected to at least one vertex in R_1
  - Forms the chain R_2

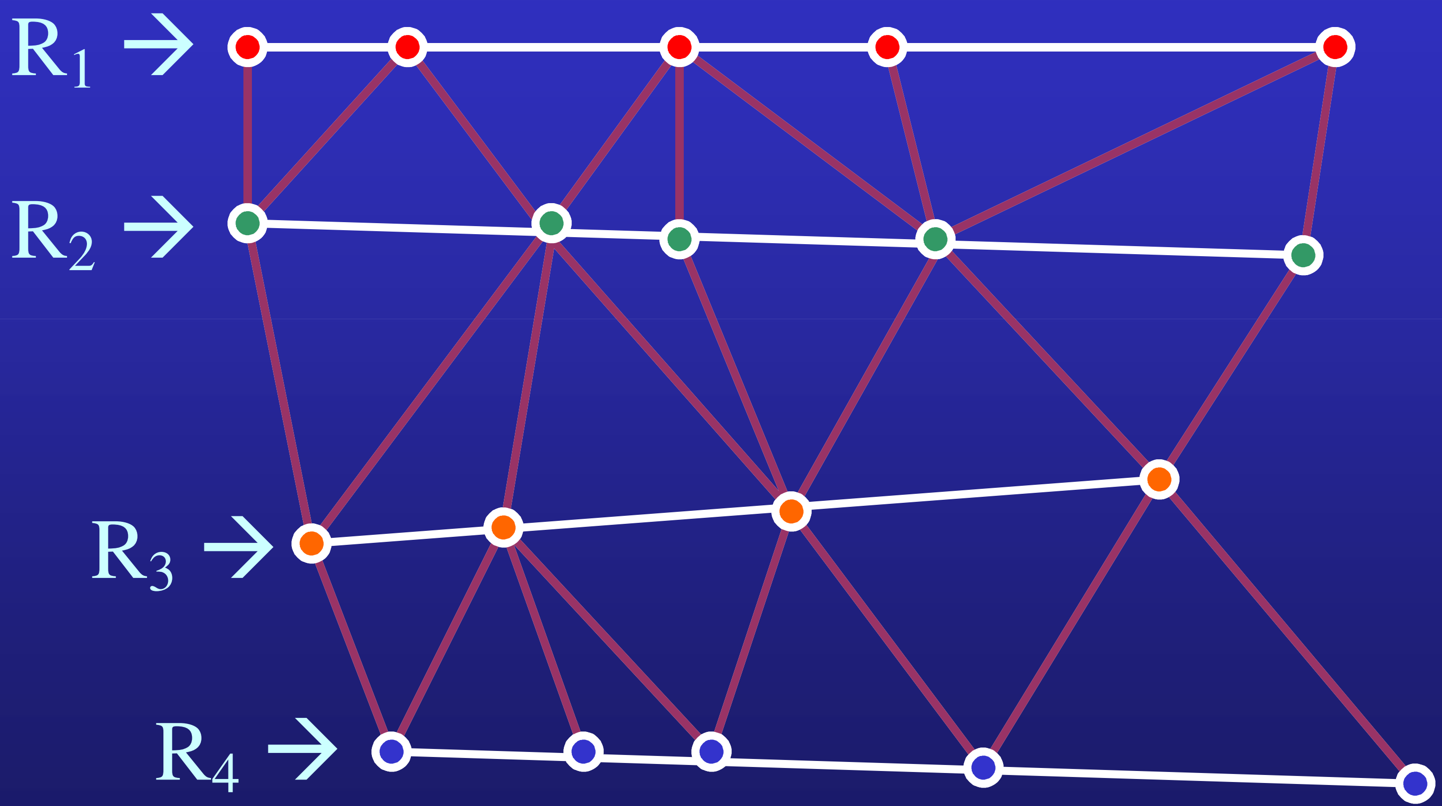
1. Forming Chains On The Mesh



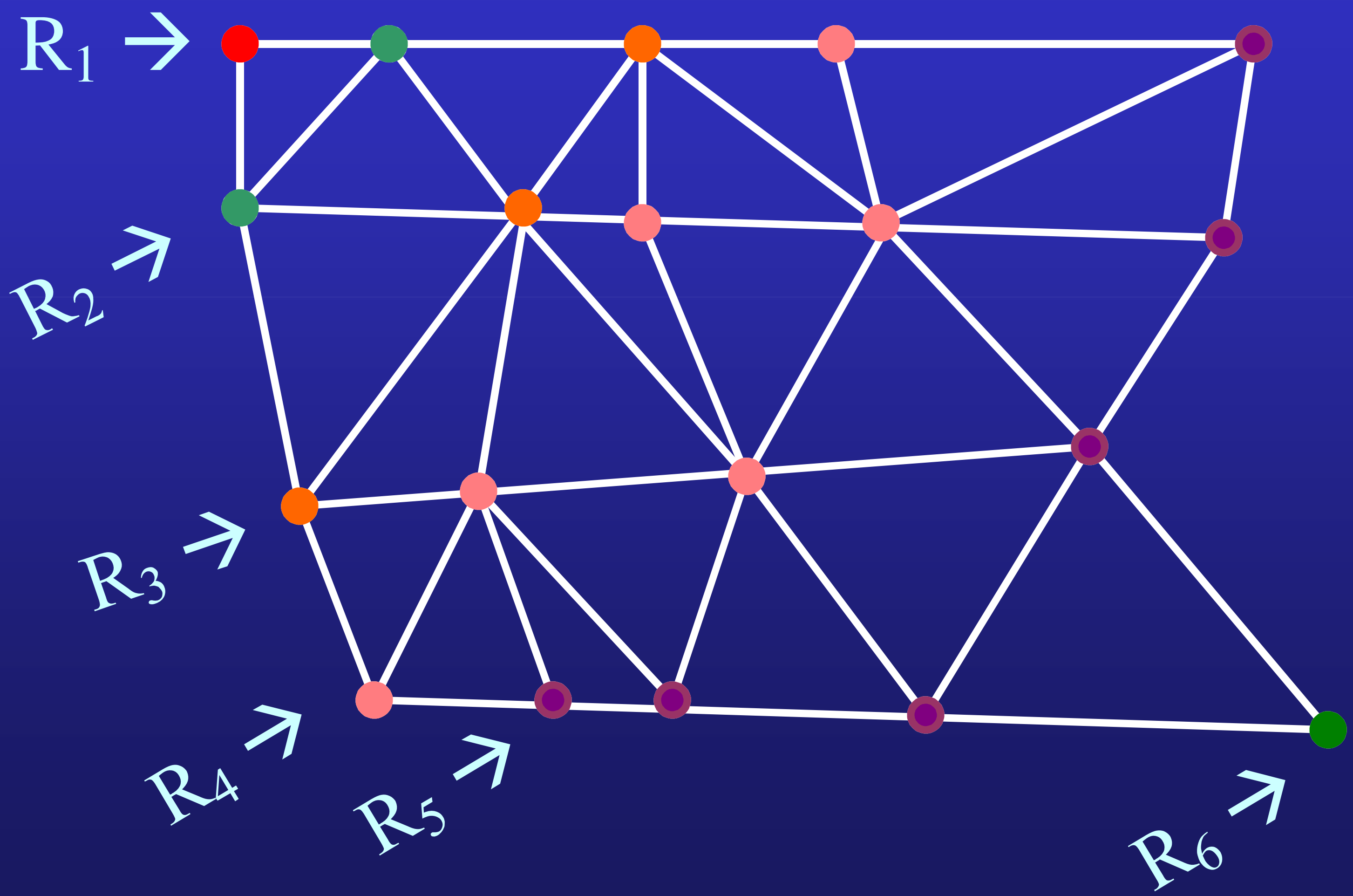
1. Forming Chains On The Mesh

- Given a mesh, choose a subset of vertices that form a connected path
  - can choose any single vertex as well
  - denote this set as R_1
- Perform a Breadth First Search and find all vertices connected to at least one vertex in R_1
  - Forms the chain R_2
- Continue forming chains until each vertex belongs to some chain
  - Some chains may not form a connected path
- Running time of O $(n + m)$

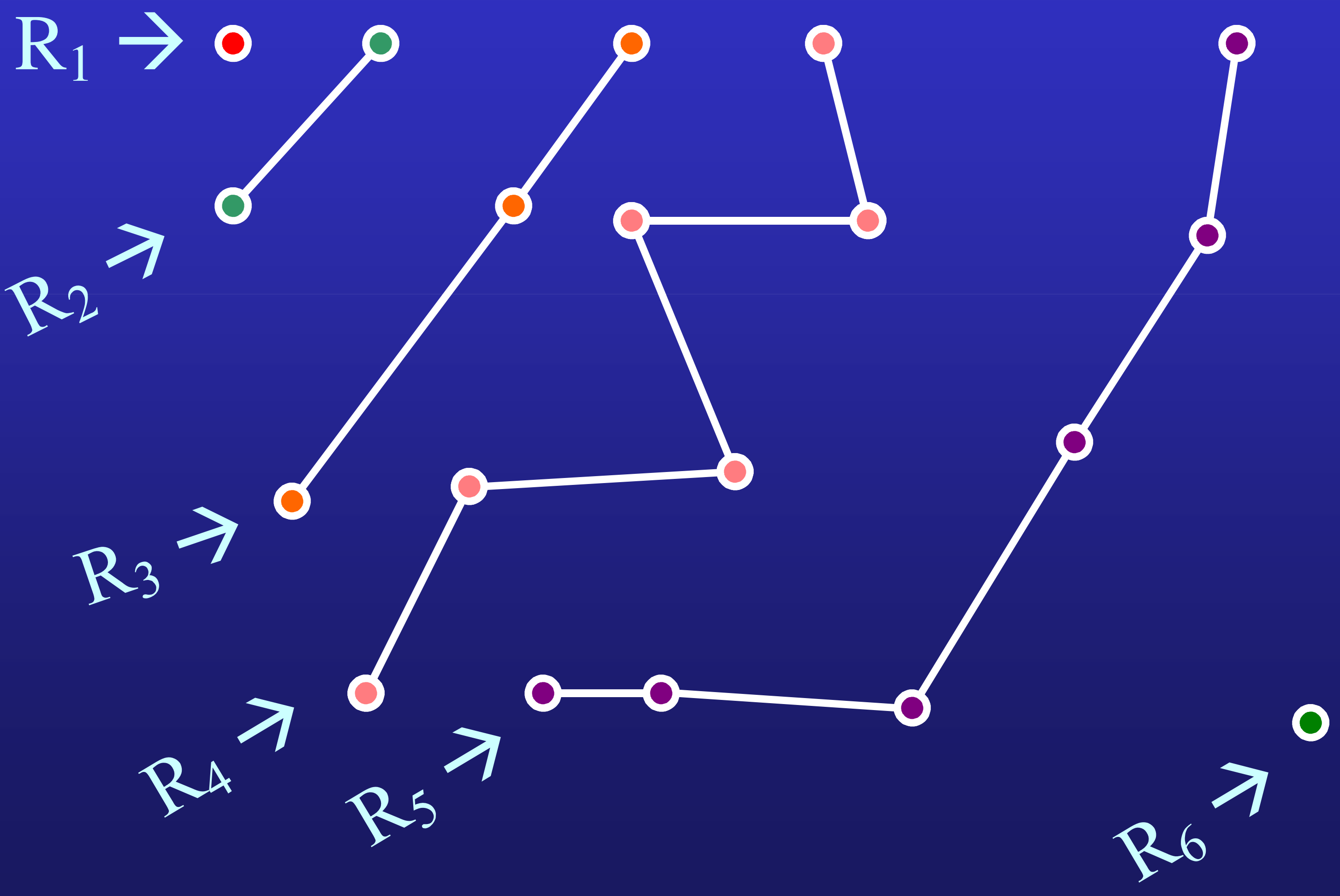
1. Forming Chains On The Mesh



1. Forming Chains On The Mesh
Example 2



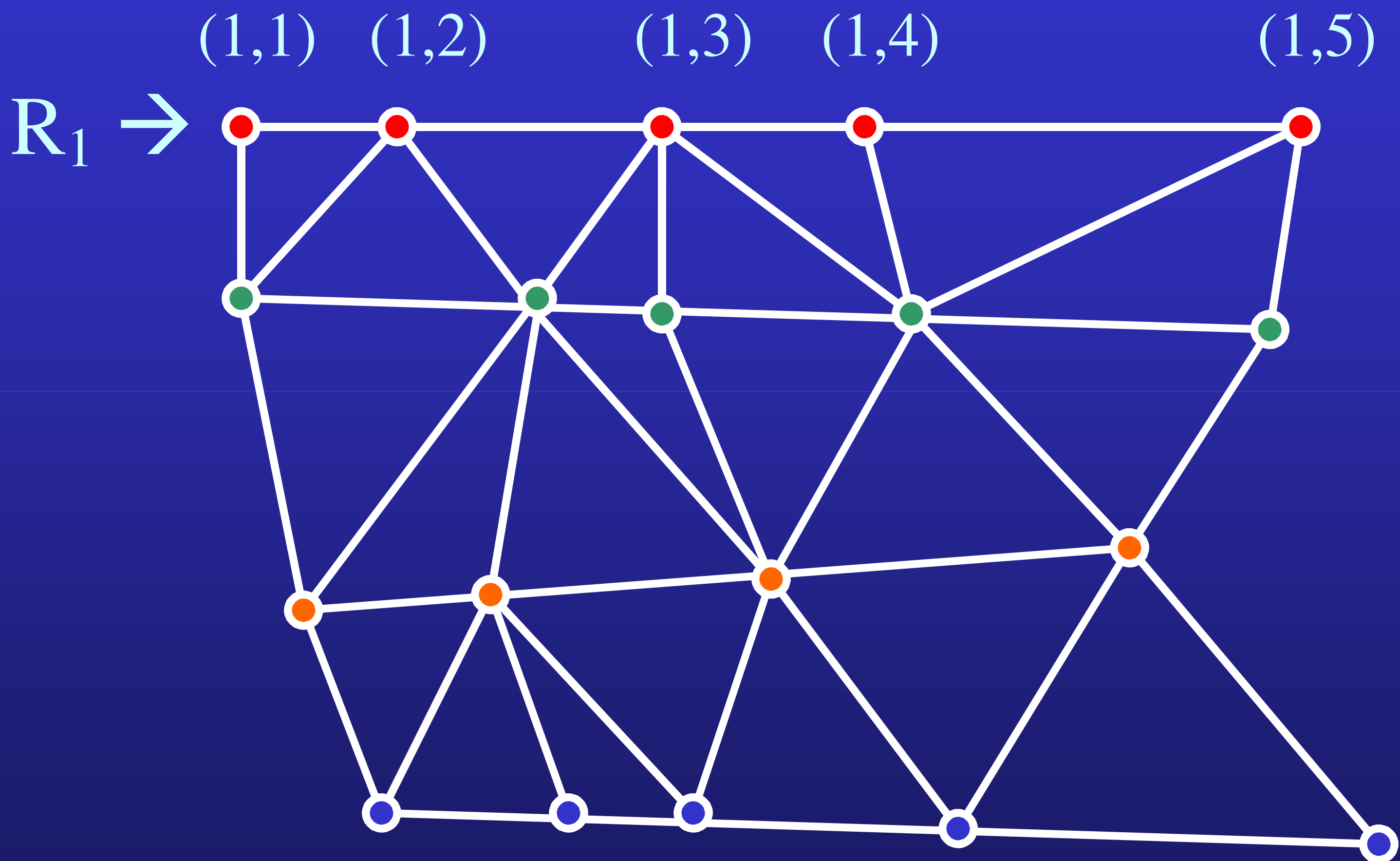
1. Forming Chains On The Mesh
Example 2



2. Ordering Vertices Within A Chain

- R_1 is already ordered to start with

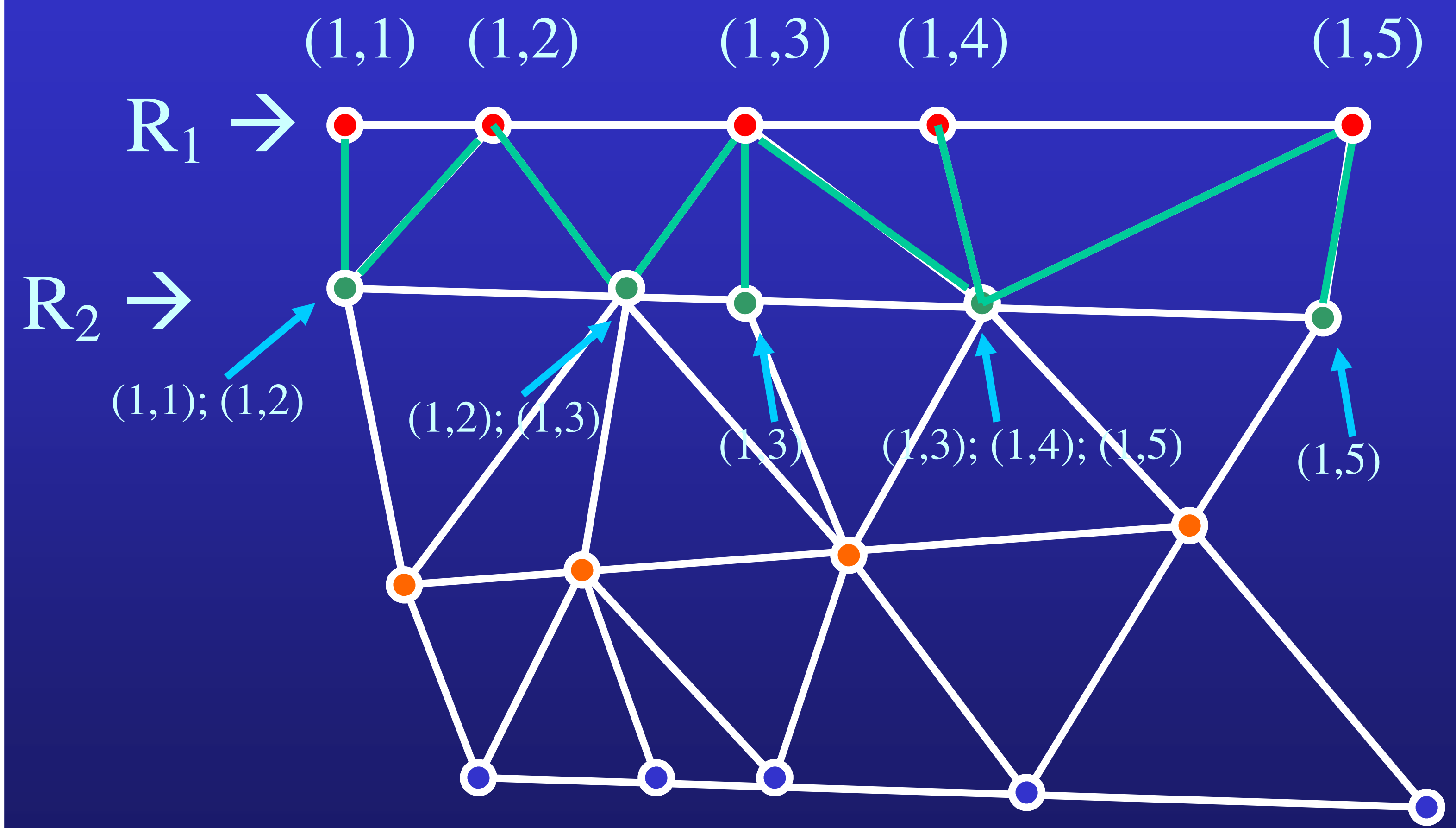
2. Ordering Vertices Within A Chain



2. Ordering Vertices Within A Chain

- R_1 is already ordered to start with
- For each vertex in R_2
  - Store the ID of vertices in R_1 (L_v) sharing an edge with it

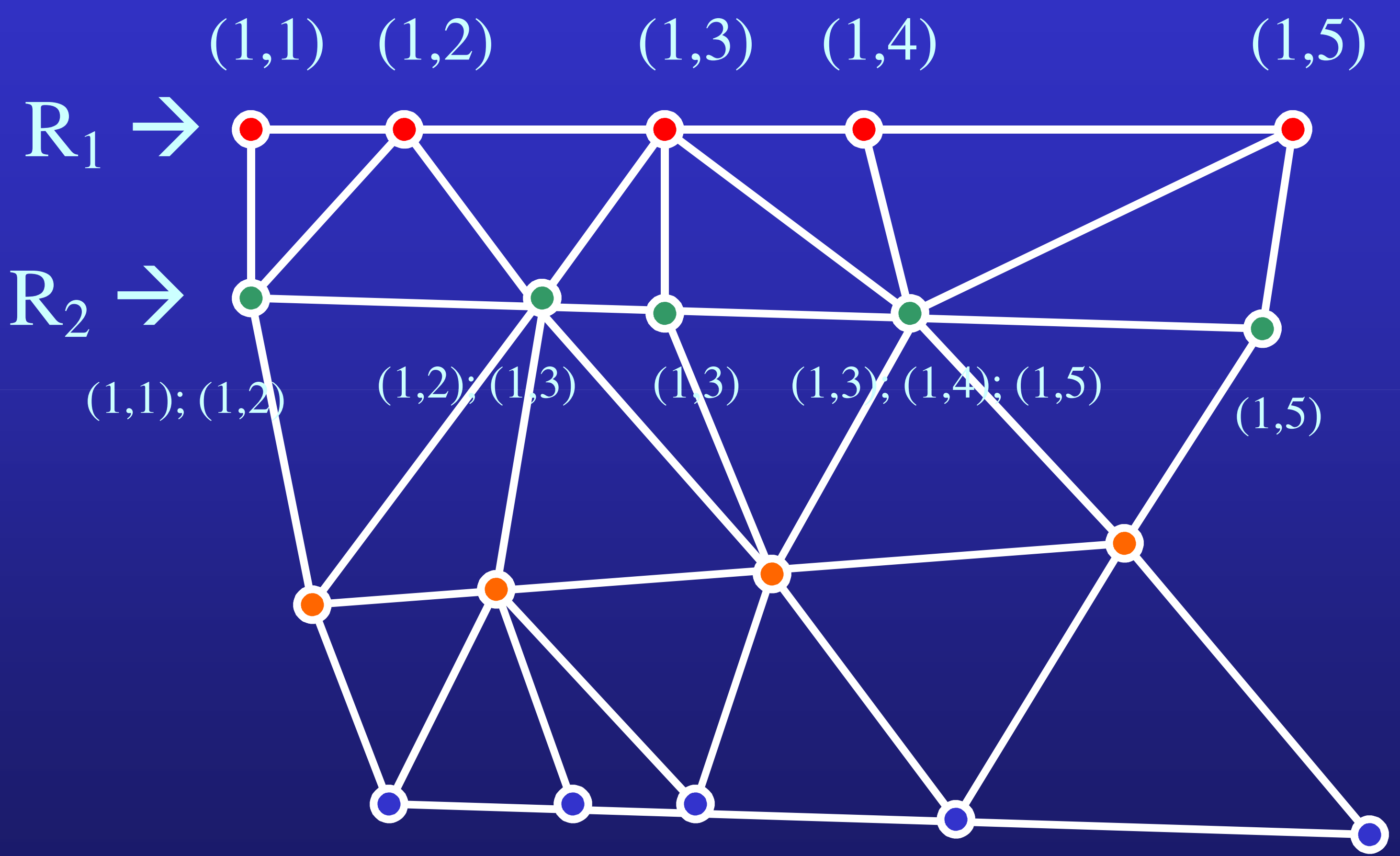
2. Ordering Vertices Within A Chain



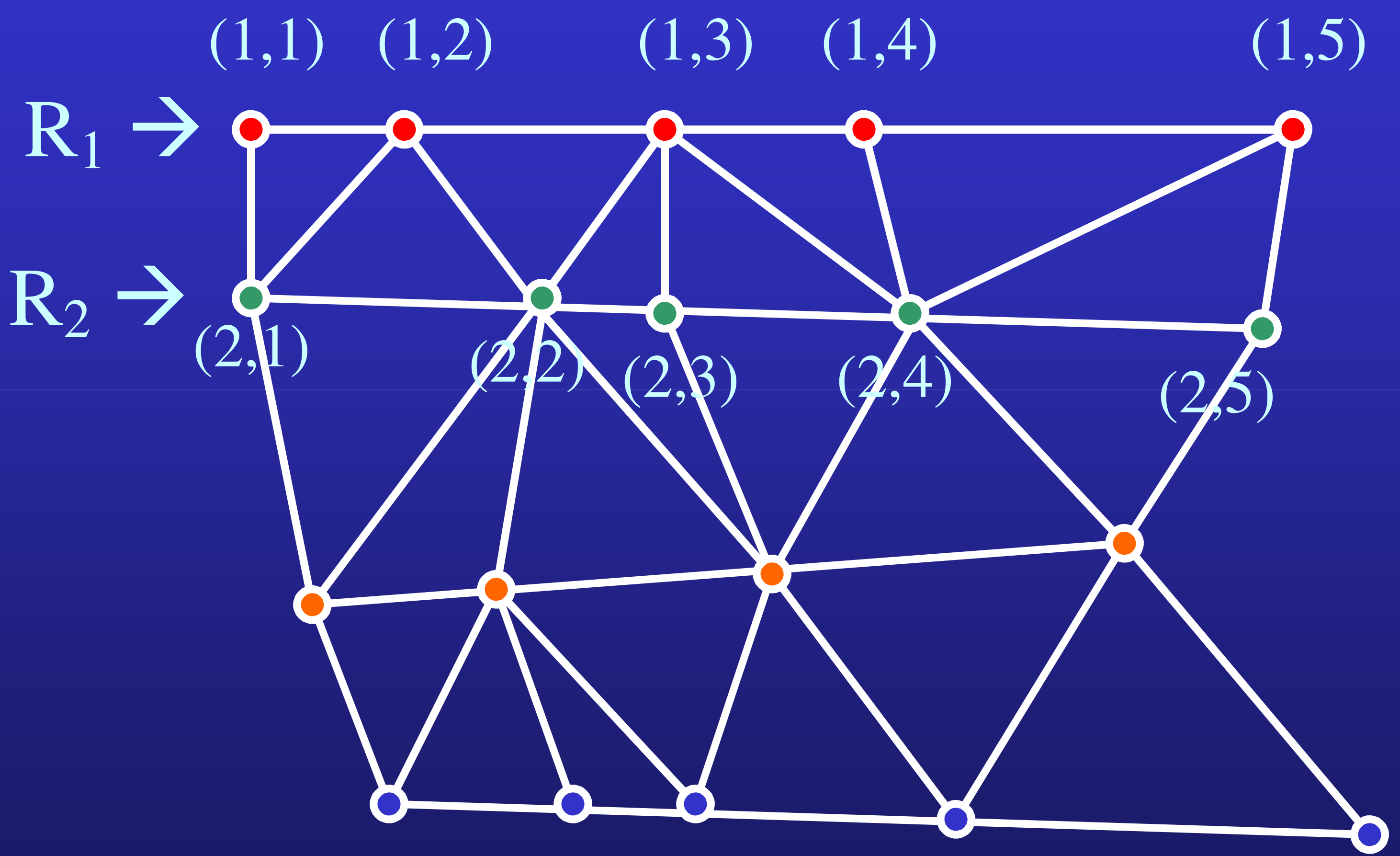
2. Ordering Vertices Within A Chain

- R_1 is already ordered to start with
- For each vertex in R_2
  - Store the ID of vertices in R_1 (L_v) sharing an edge with it
- Sort the vertices in R_2 based on L_v
  - Specific rules to break ties
- This defines the order within R_2

2. Ordering Vertices Within A Chain



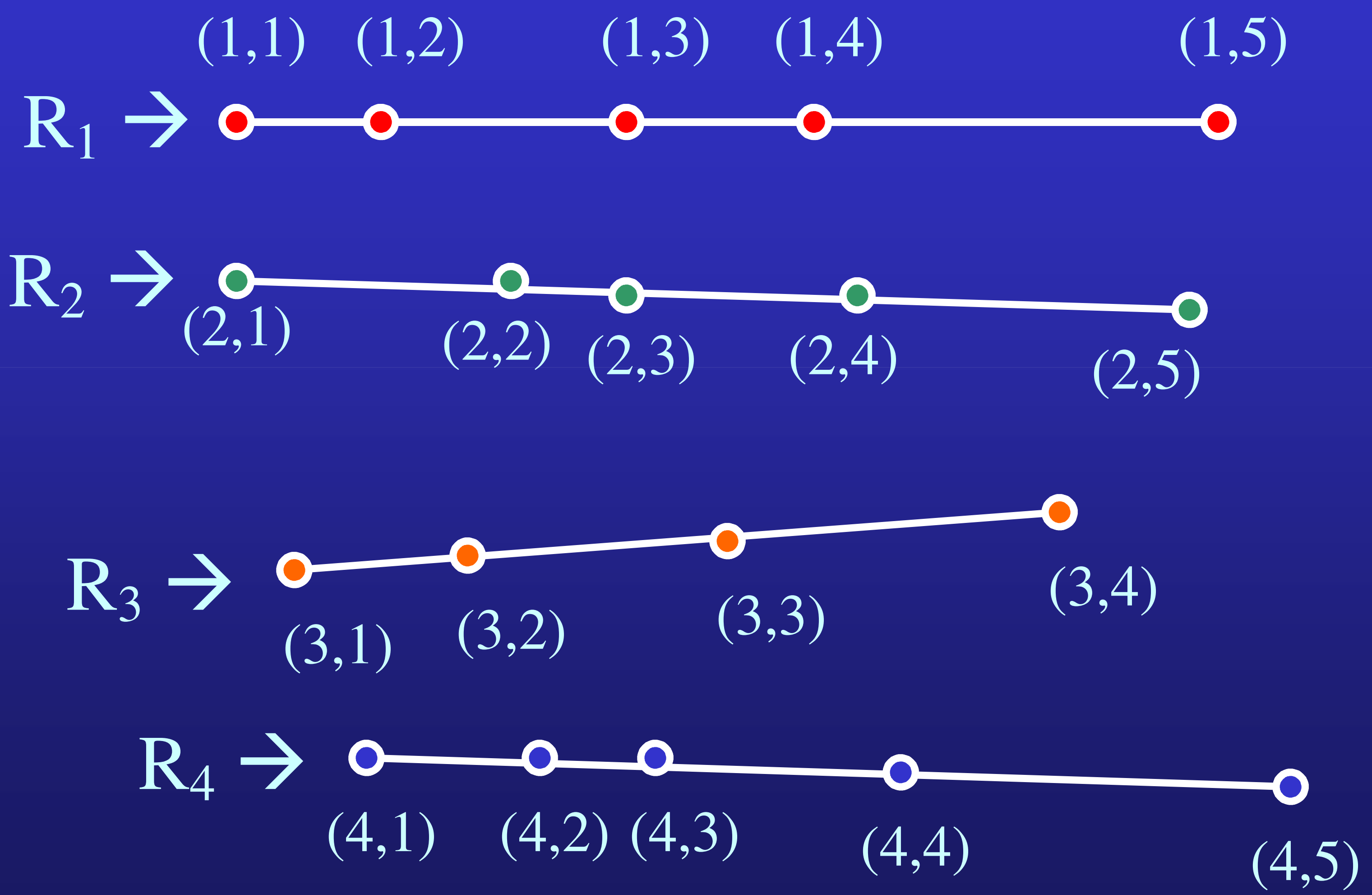
2. Ordering Vertices Within A Chain



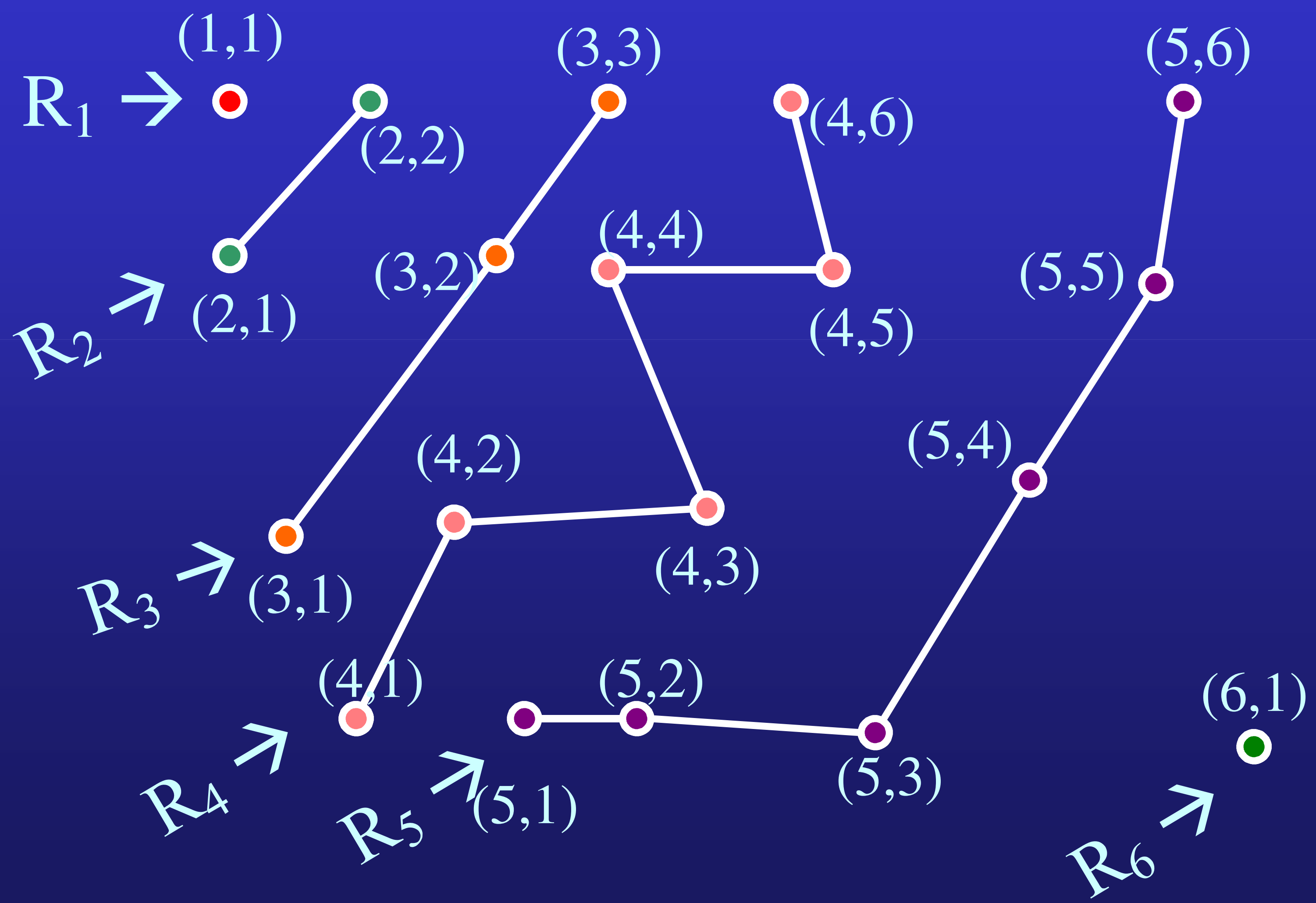
2. Ordering Vertices Within A Chain

- P_1 is already ordered to start with
- For each vertex in P_2
  - Store the ID of vertices in P_1 (L_v) sharing an edge with it
- Sort the vertices in P_2 based on L_v
  - Specific rules to break ties
- This defines the order within P_2
- Perform similar computation for each subsequent chain.
- Running time of O (n log(n))

2. Ordering Vertices Within A Chain



2. Ordering Vertices Within A Chain

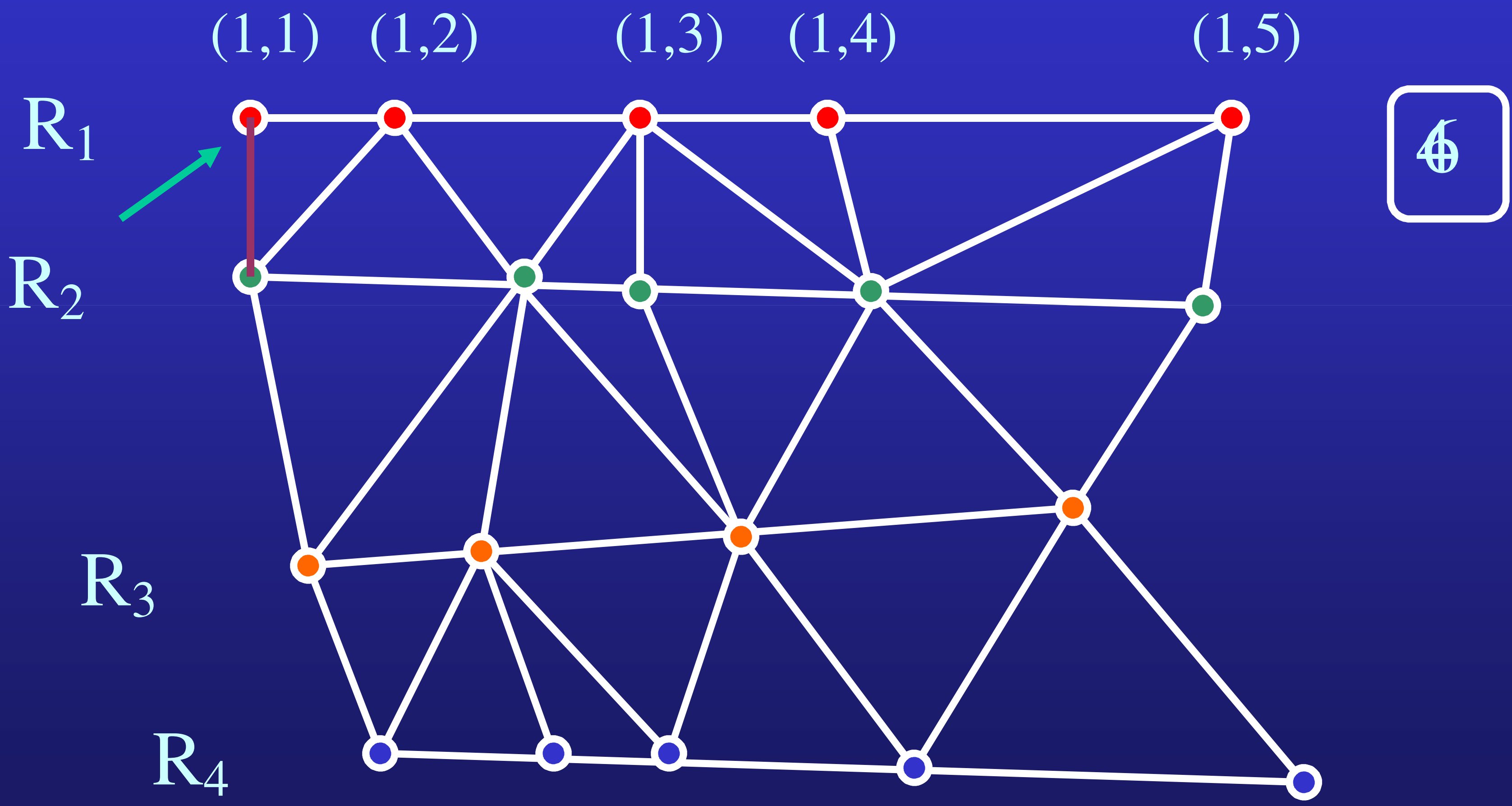


3. Forming Cuts Of Vertices

- Each cut is defined as a subset of vertices in each row
- Consider a row R_i
  - say a vertex *v* is introduced into the cache by some triangle joining vertices in R_{i-1} and R_i
  - The subset in R_i is chosen in a fashion that ensures that every vertex *v* remains in the cache when triangles joining *v* to vertices in R_{i+1} are traversed
  - keep a counter which keeps track of the number of vertices that can be loaded from R_{i+1} and continue while counter > 0

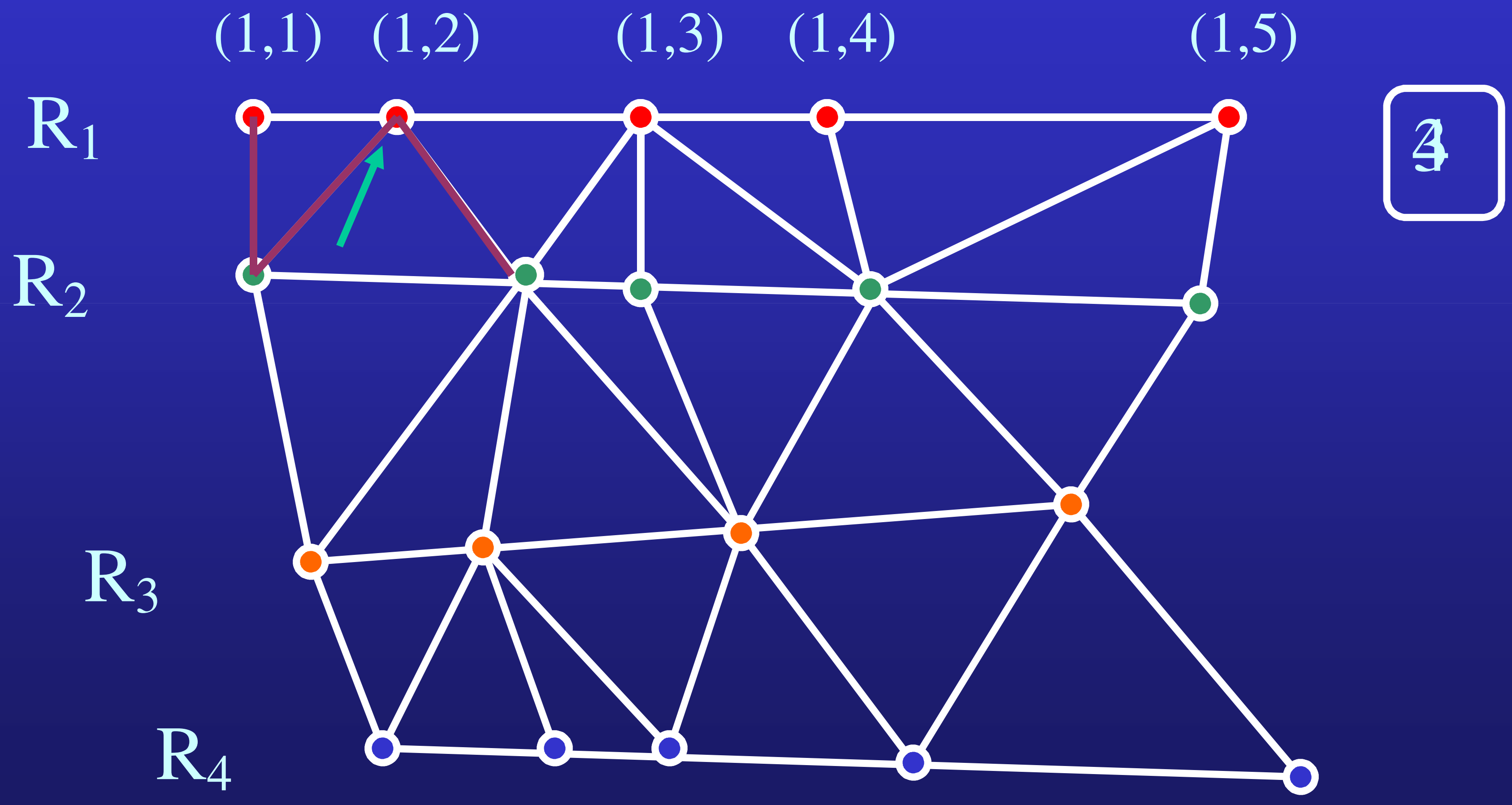
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



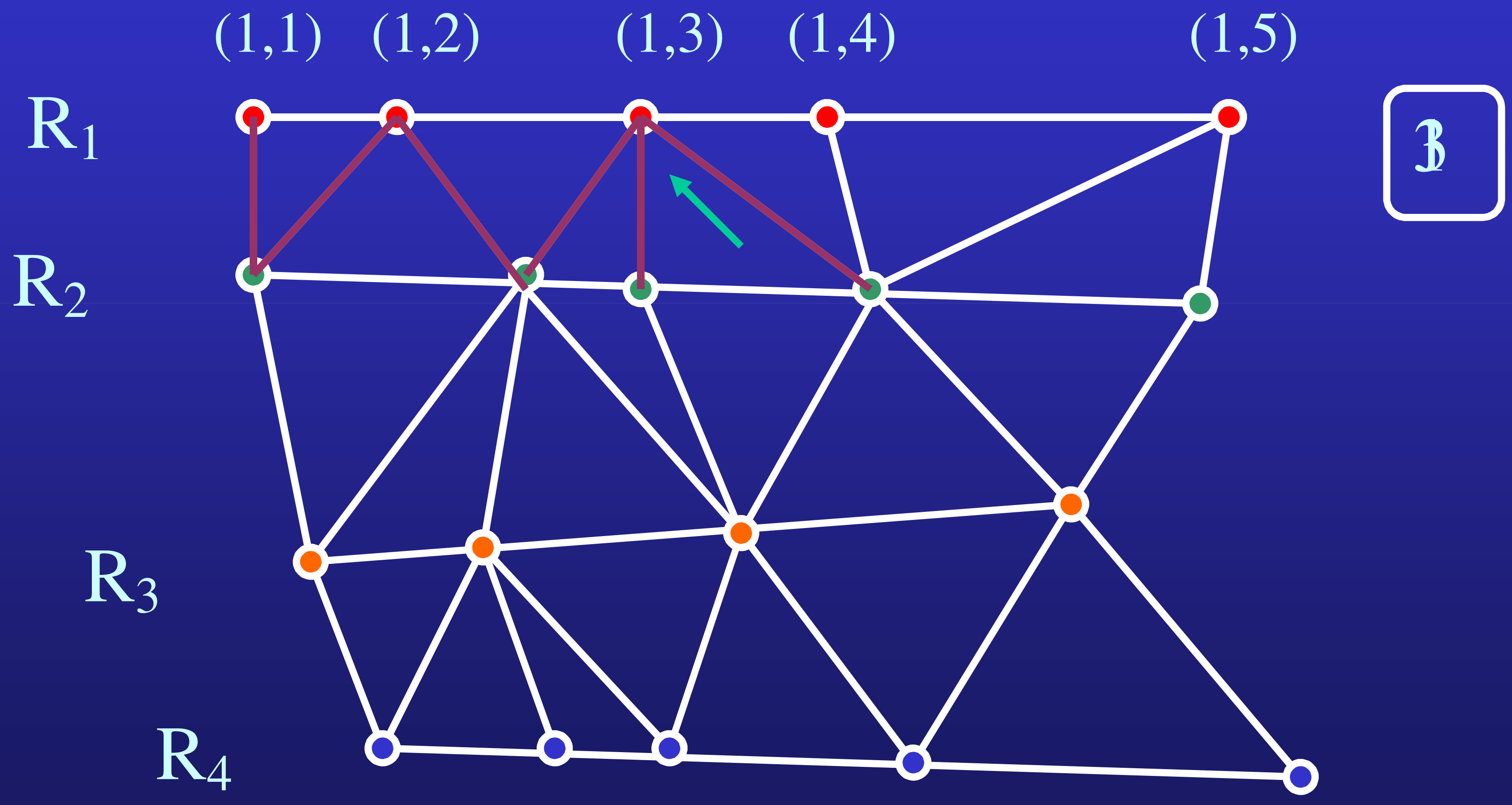
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



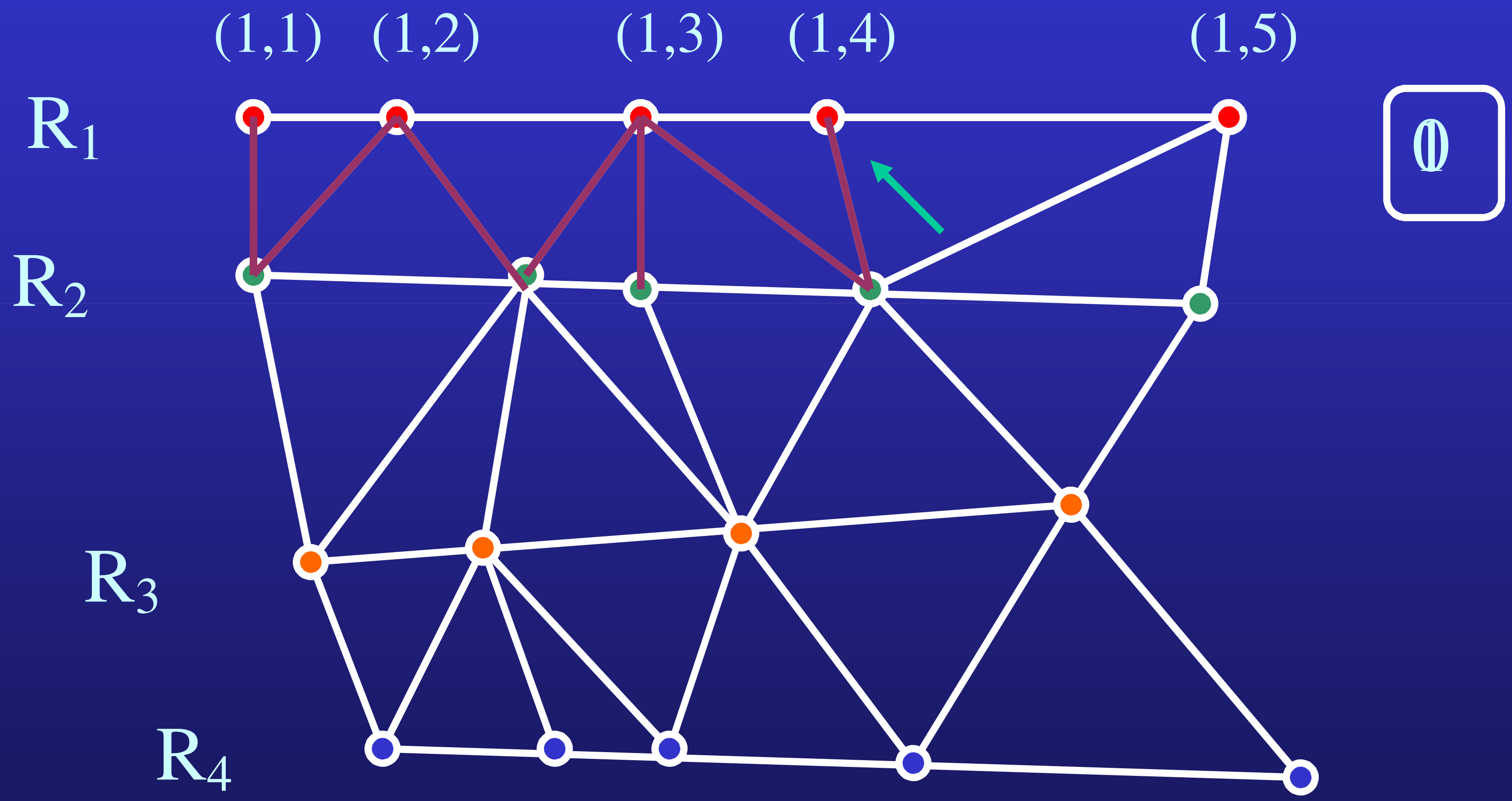
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



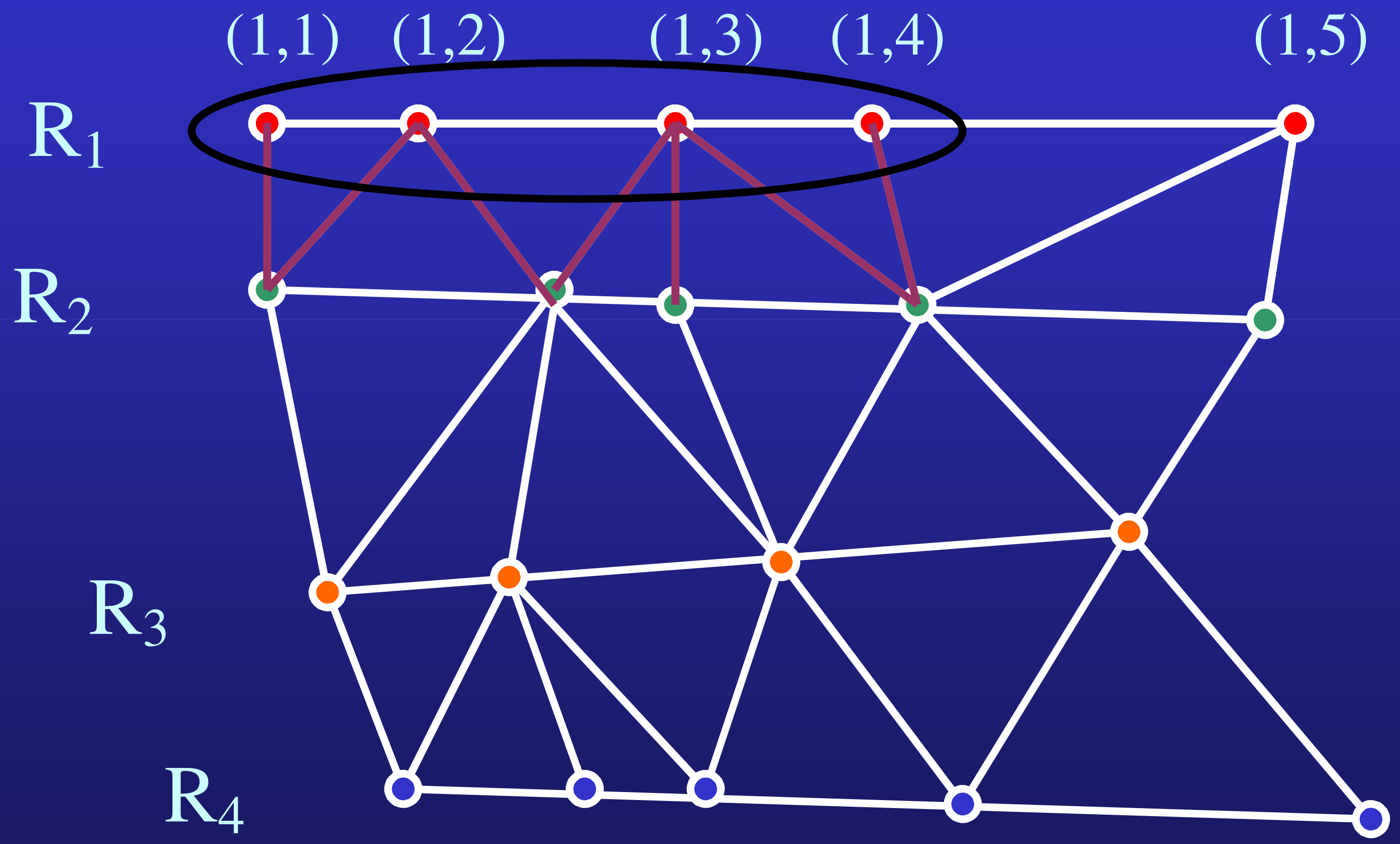
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



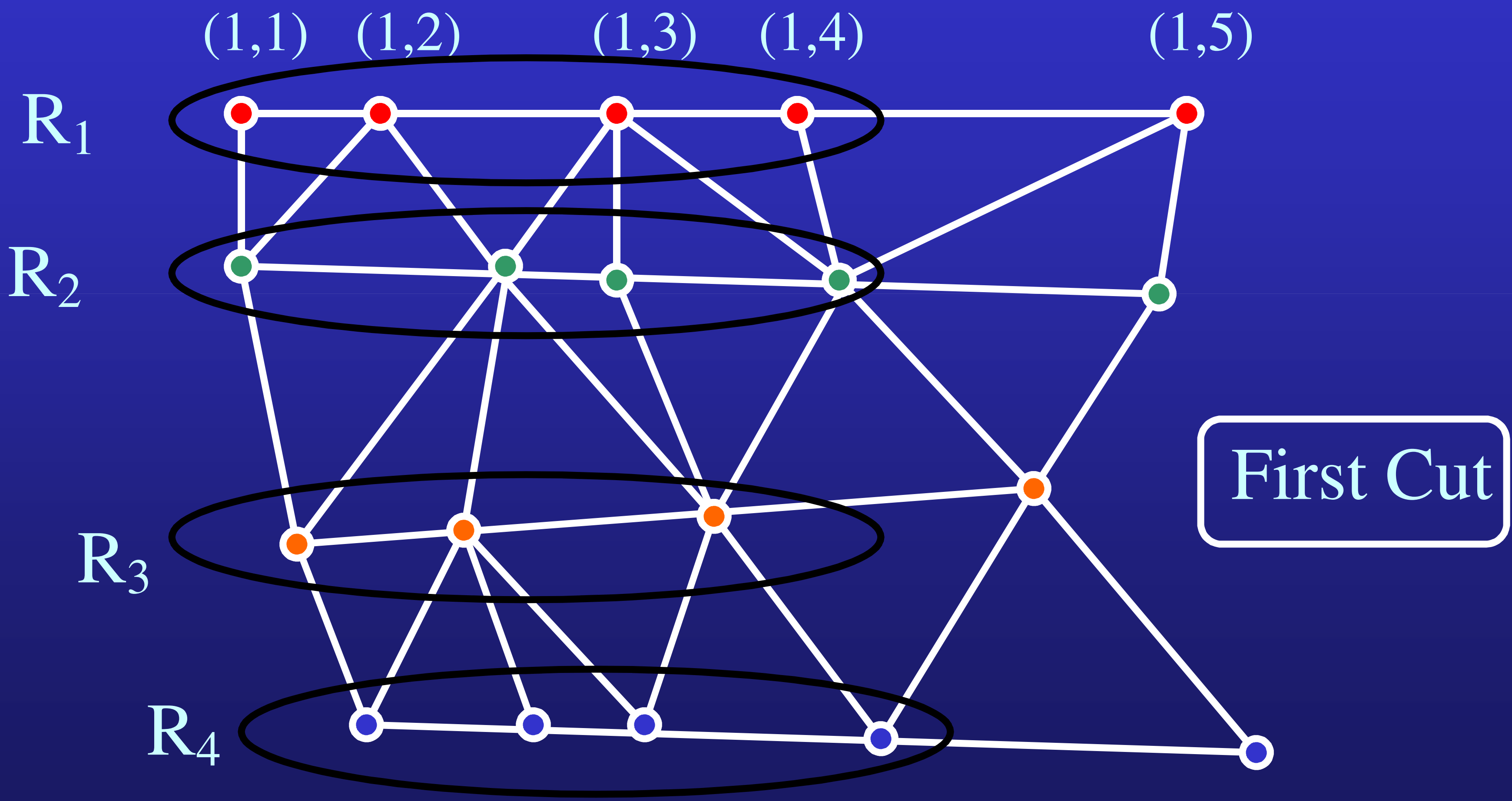
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



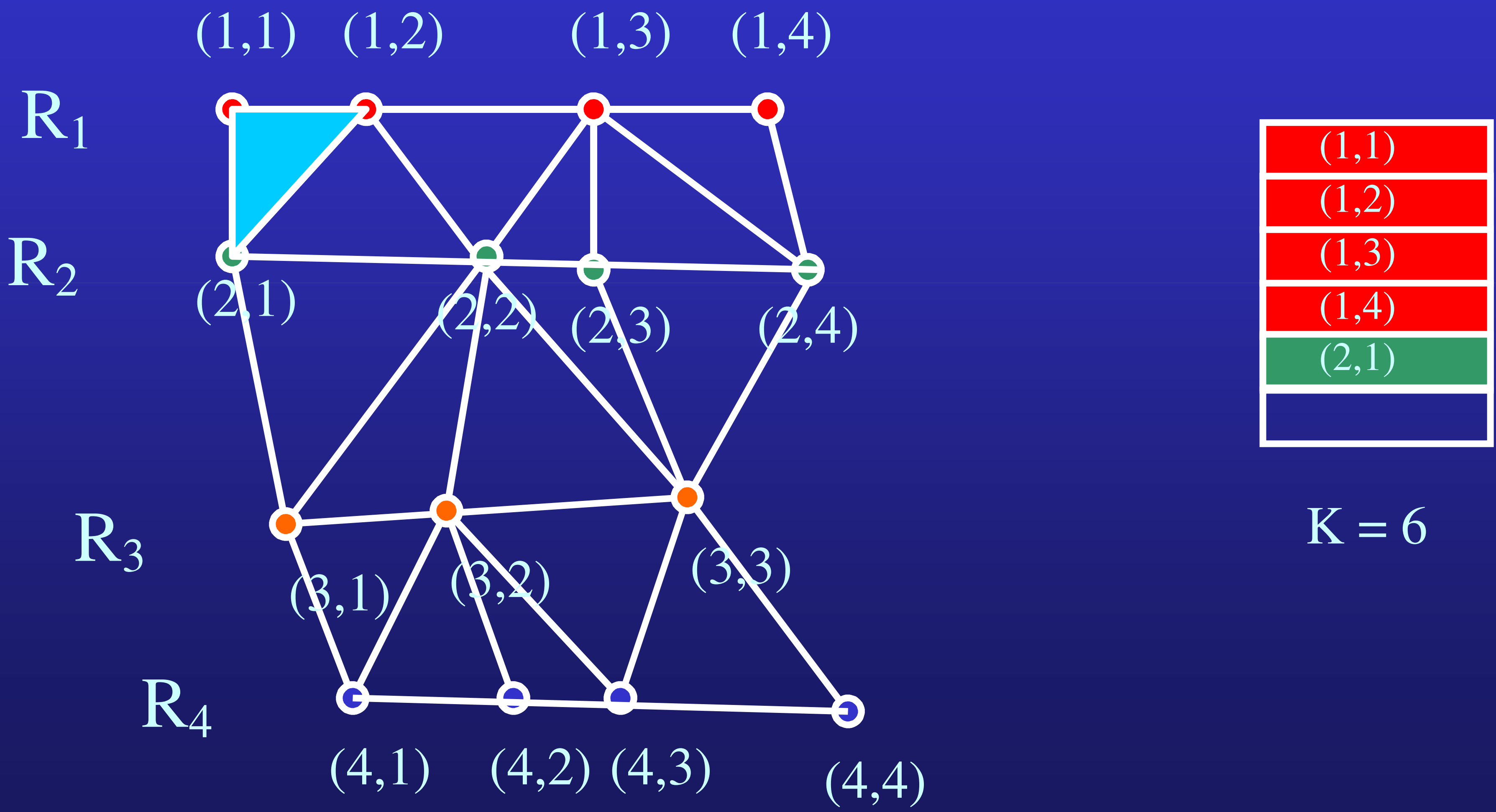
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



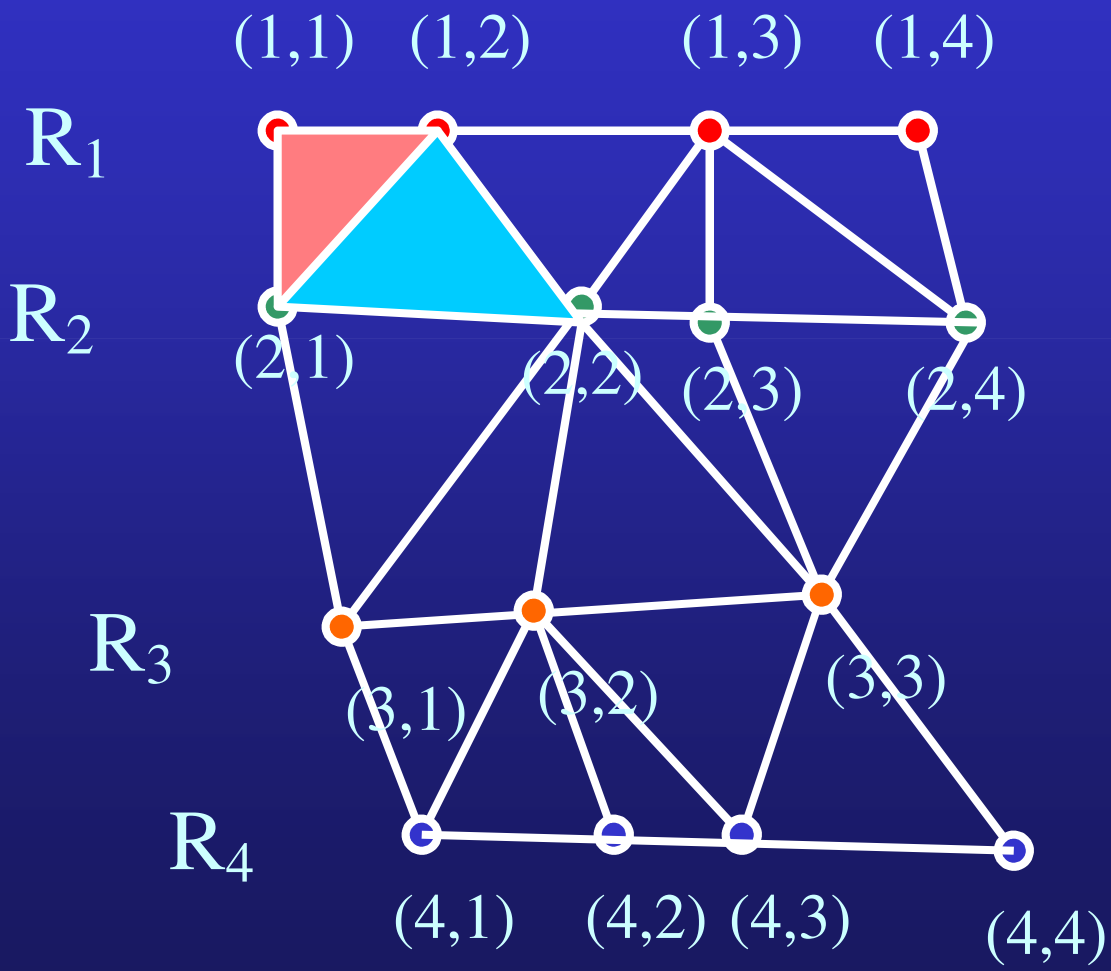
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



3. Forming Cuts Of Vertices

- Assume cache size (K) = 6

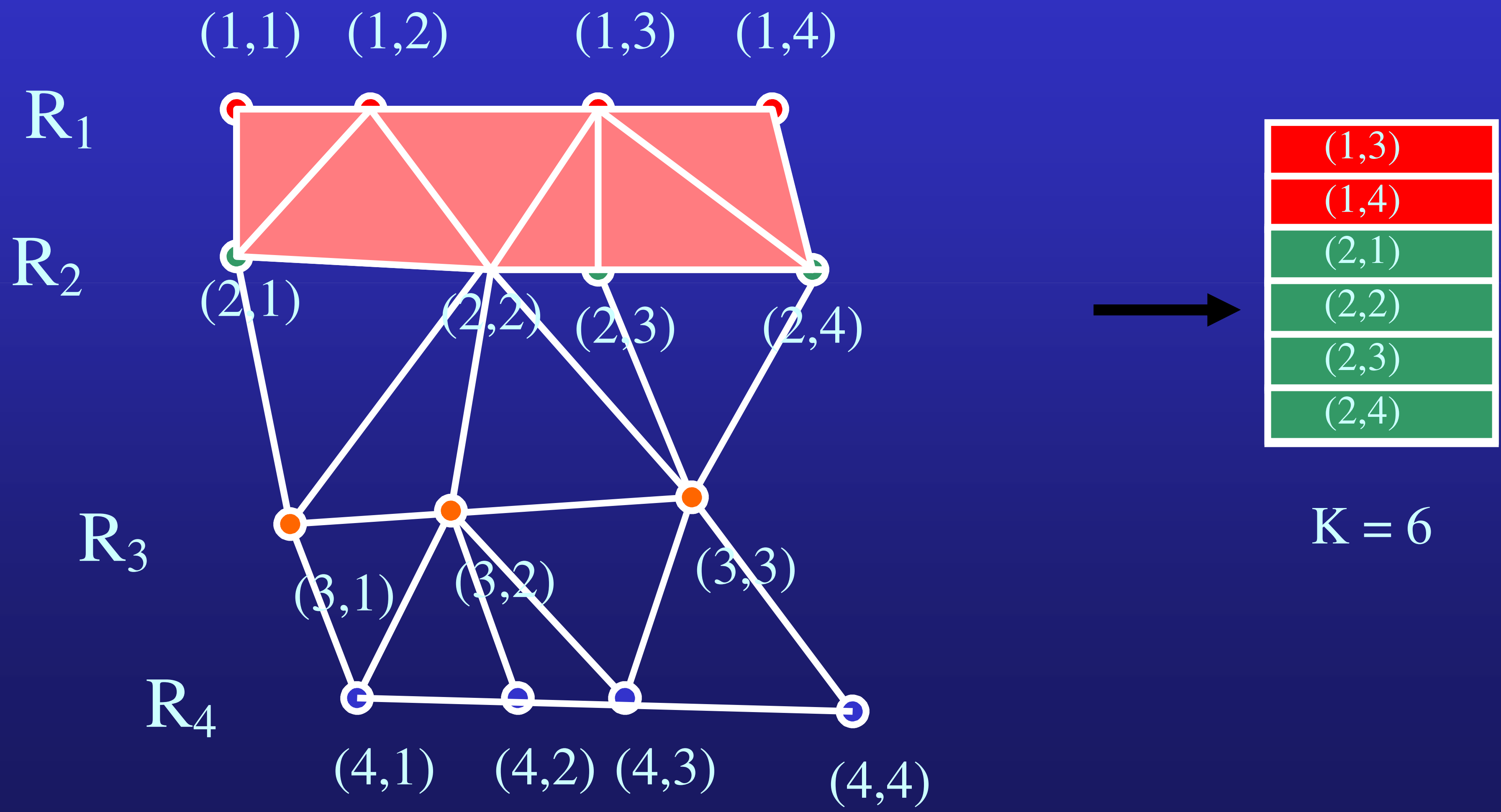


(1,1)
(1,2)
(1,3)
(1,4)
(2,1)
(2,2)

K = 6

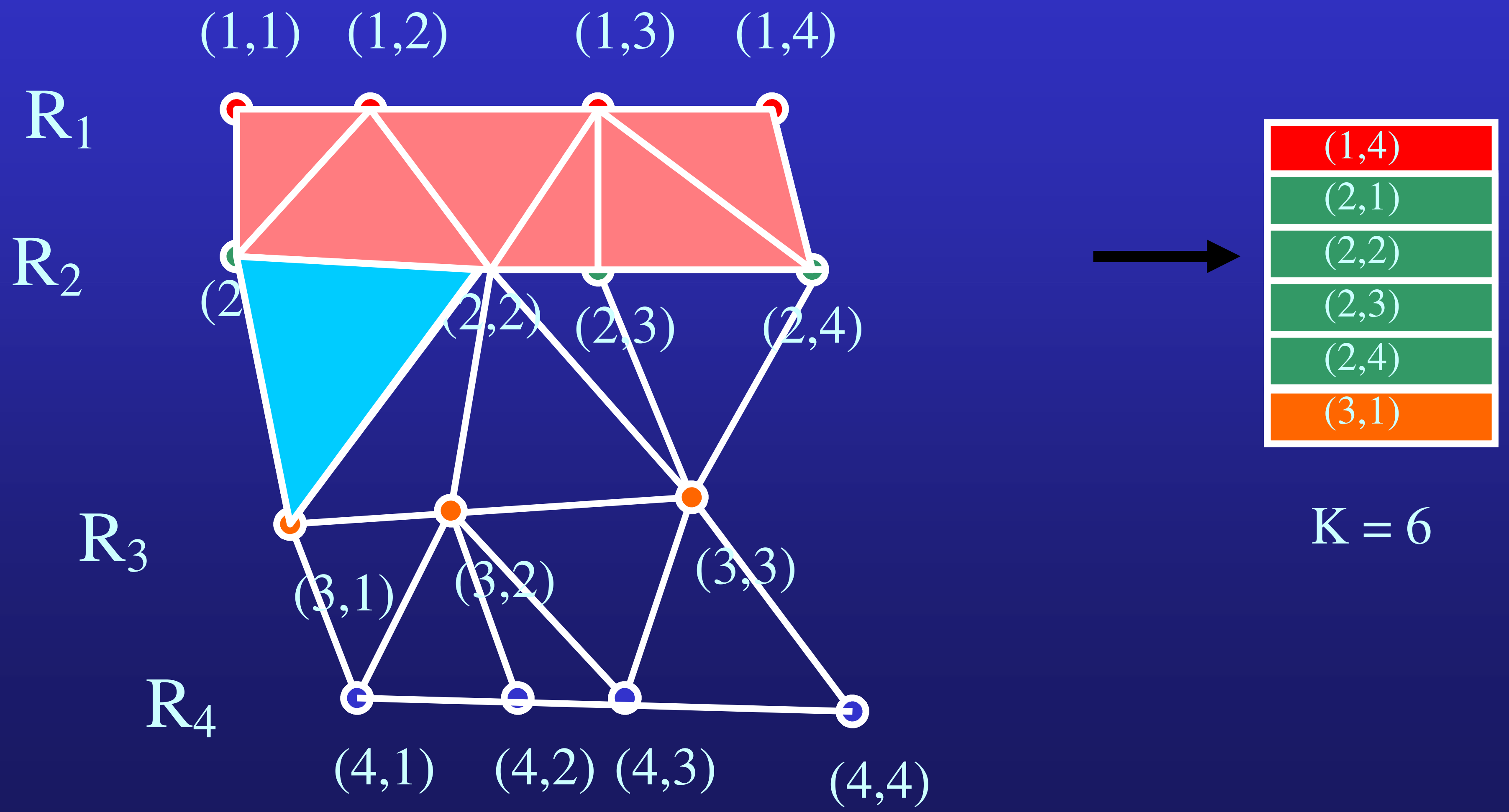
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



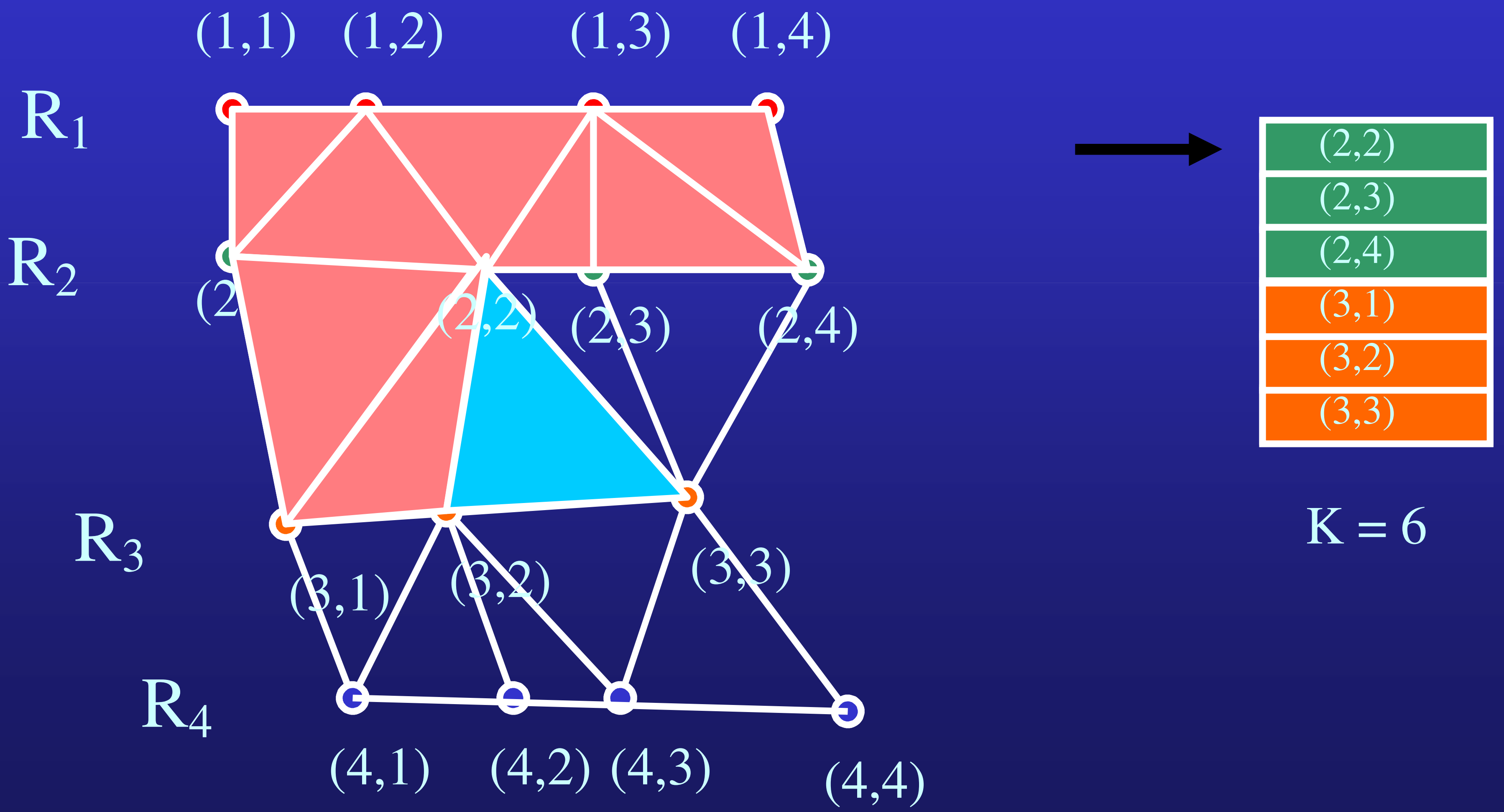
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



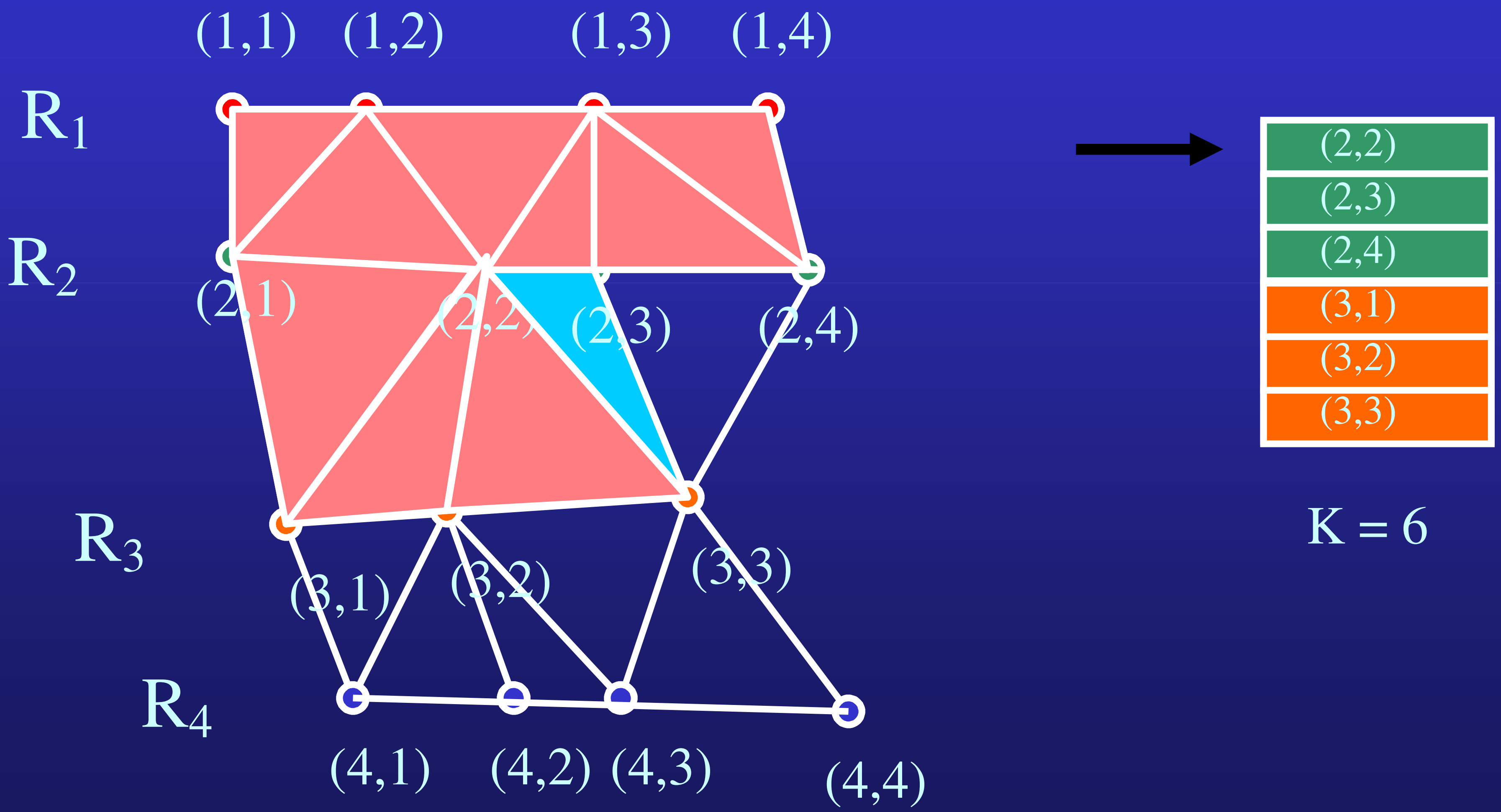
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



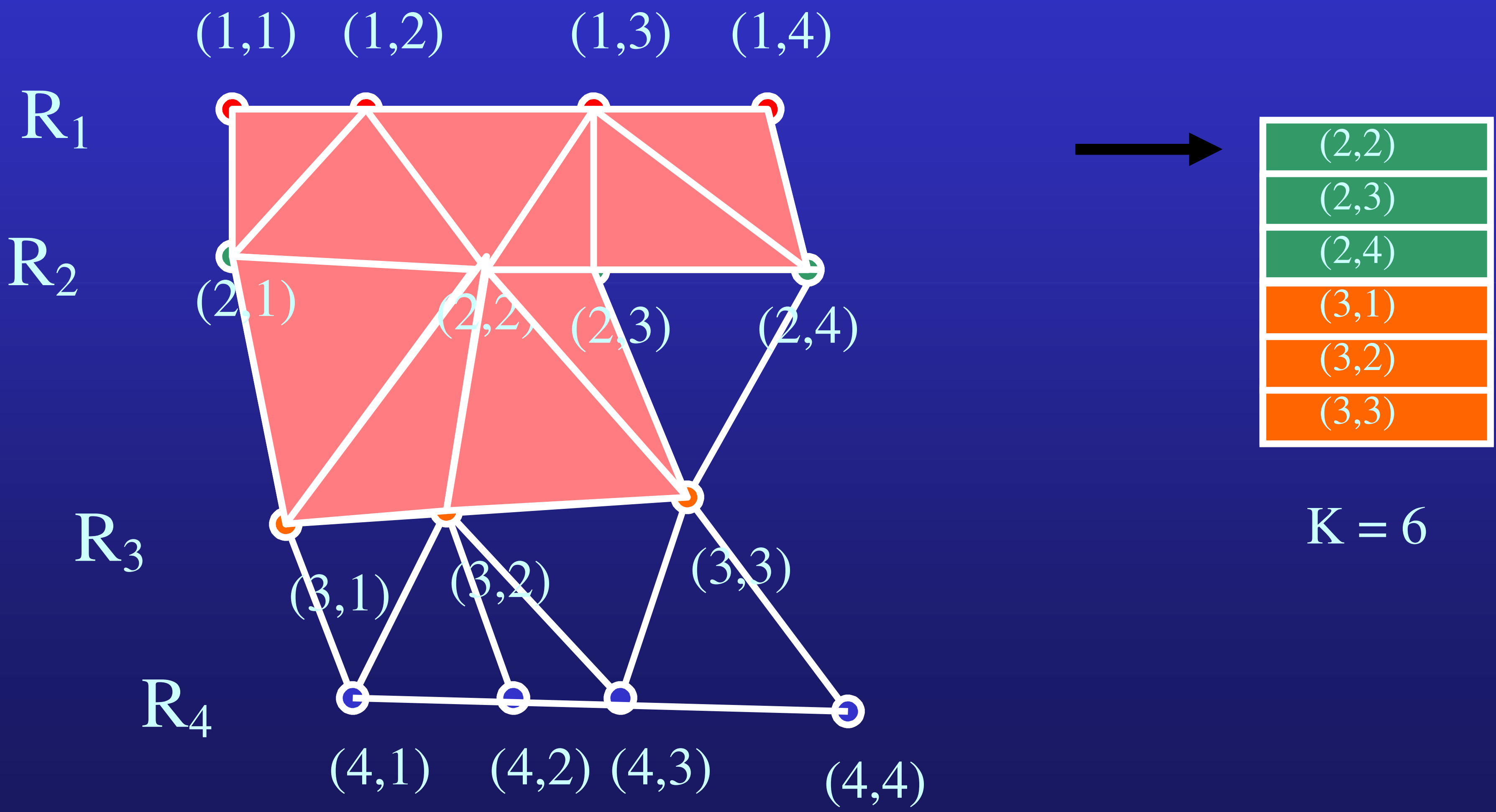
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



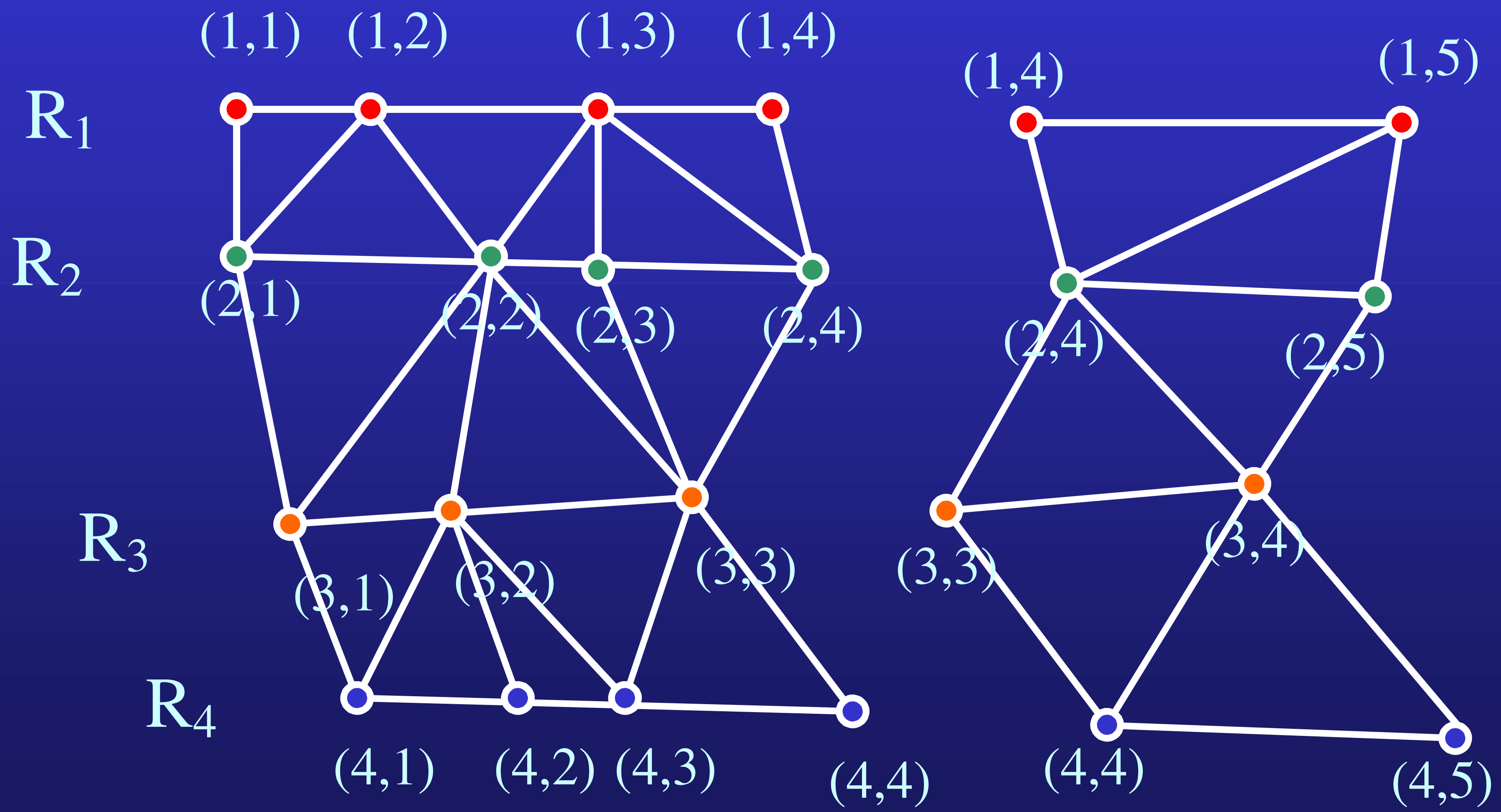
3. Forming Cuts Of Vertices

- Assume cache size (K) = 6



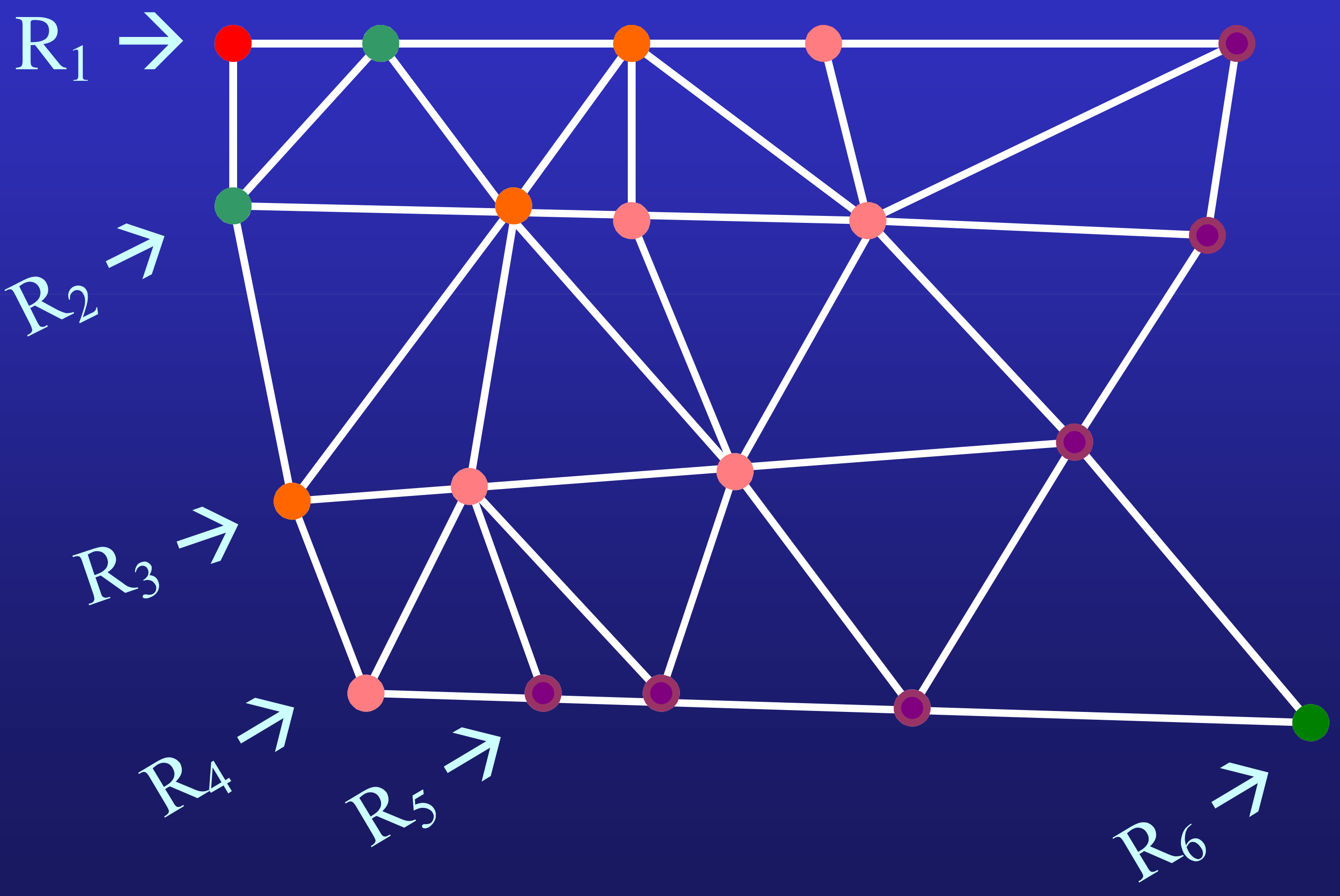
3. Forming Cuts Of Vertices

2 cuts are formed and 4 vertices need to be reloaded



3. Forming Cuts Of Vertices

3 vertices need to be reloaded

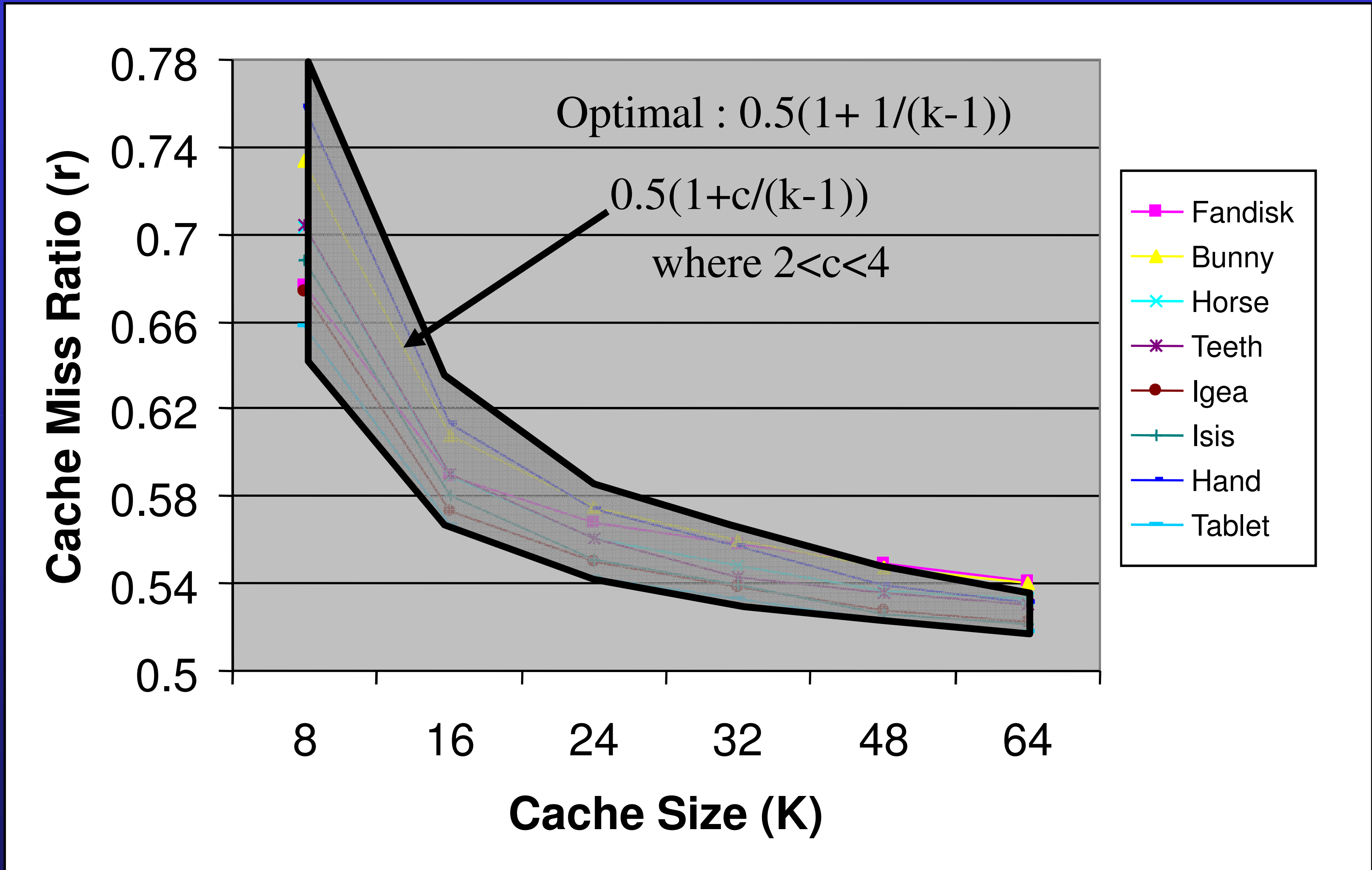


Results (Cache Size K = 16)

Model	Vertices (n)	Triangles (m)	Cache Miss Rate (r)	Lin et al. (r)	Degenerate Tris (%)
Grid20	391	704	0.580	0.605	2.9
Fandisk	6,475	12,946	0.588	0.595	2.4
Bunny	35,947	71,884	0.608	0.597	3.6
Horse	48,485	96,966	0.589	0.599	2.6
Teeth	116,604	233,204	0.590	0.604	2.9
Igea	134,556	269,108	0.573	0.601	2.3
Isis	187,644	375,284	0.580	0.603	3.4
Hand	327,323	654,666	0.612	0.606	4.7
Tablet	539,446	1,078,890	0.567	0.580	2.3

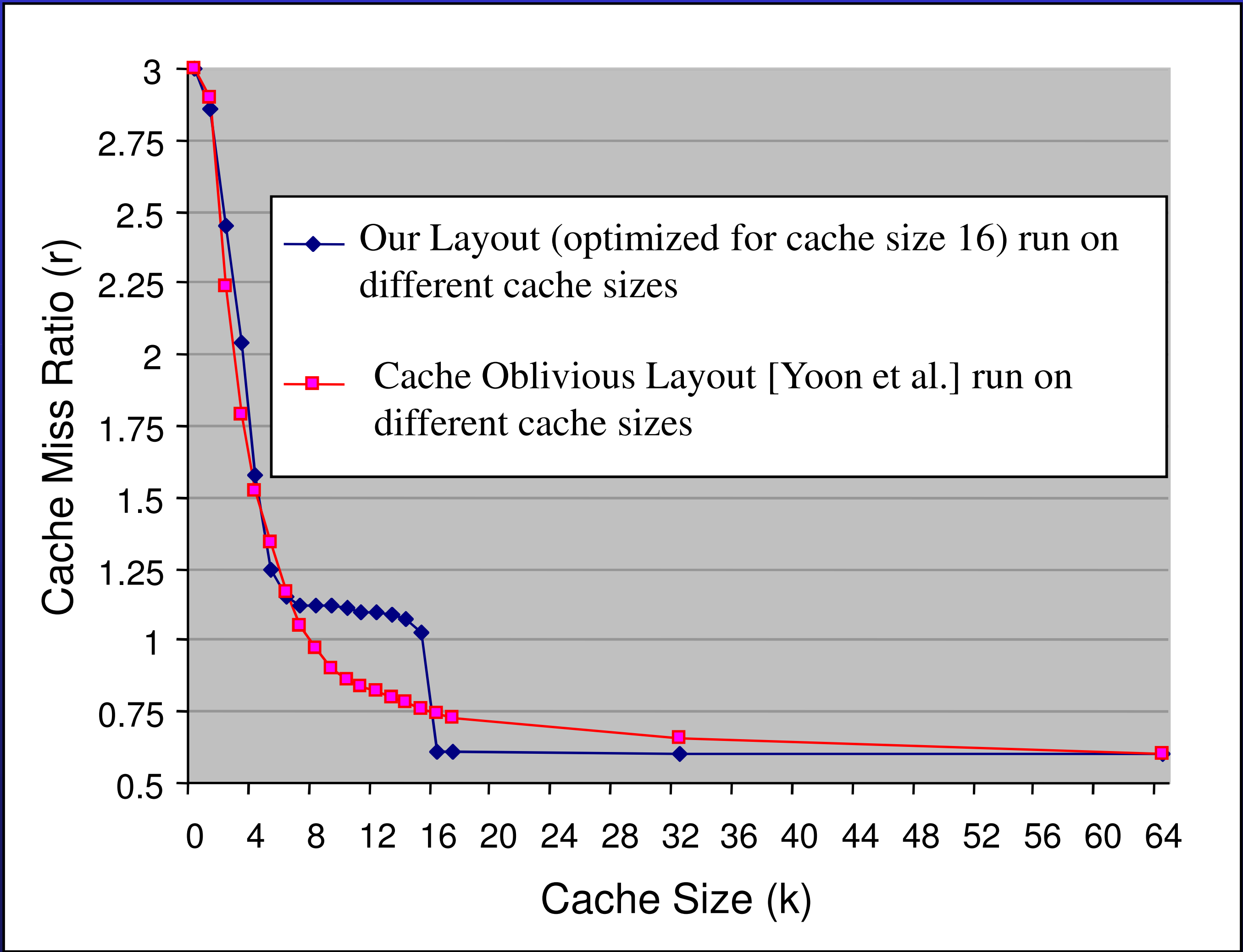
r = Cache Misses Per Triangle

Results



r with varying cache sizes for different models

Comparison with Cache Oblivious Layout



Topology Encoding

- Out of 3*m* input indices (for *m* triangles)
  - *m/2* indices refer to vertices encountered for the first time
    - Implicit; no bits necessary
  - *2.4m* indices are cache hits
    - Encode by their cache address
  - *(r-1/2)m ~ 0.1m* indices are *re*loaded post cache-eviction
    - Require explicit vertex-index
  - Huffman encode these
    - <5 bits per index, on average
    - Variable length encoding

Fixed-length Compression

Let *F* represent the case where an index is referred for the first time

Let *C* represent the case where an index is in the cache

Let *R* represent the case where an index is reloaded into the cache

Encode an entire triangle:

- *FFF* : fetch the next 3 vertices from vertex array
- *FFC* : fetch the next 2 vertices and encode the cache-position of the third
  - requires log(*K*) bits (4 bits for a 16-entry cache)
- *FFR*: fetch next 2 from vertex array; encode a previously used index
- *FCR:* fetch next vertex, one in-cache, one previously used
- *FCC* : fetch the next vertex; encode two cache positions
  - Requires ~8 bits for a 16-entry cache
- *CCC* : ~8 bits
- *CCR* : ~8 bits plus the reload index
  - reloaded vertices exist at the boundary of the cuts
  - bound the number of rows in a cut to keep *R* small
  - If R does not fit, change case

Unfavorable Cases

- RRR
  - Guarantee to not exist, by construction
- RRX (*ijx*)
  - Rare, $O(n^{(1/4)})$
  - Fewer than 0.1% of triangles in experiments
  - Still disallowed
    - Convert to two triangles
    - DDR (*iii*) , CRX (*cjx*)

Unfavorable Cases

- CCC
  - Enforce one of the cache entries 0,1,2
  - In fact, if we eliminate FFF
    - One of the CCC must be in 0,1
    - Split into cases CC0 and CC1
- CCR
  - R may not fit
  - Make a DDR
    - Followed by CCC

Decompression Scheme and Hardware Extensions

- Decompression is simple
  - decode the case number for each triangle
  - decode the various addresses required
  - *directly* get the cache address of vertices already in cache
  - fetch the vertices loaded for the first time or reloaded
- Hardware changes
  - decoding logic to compute the case number
  - Mux to fetch addresses
    - Bypass cache-tag lookup
  - Counter for ‘F’ vertices

Results and Comparisons

- Compressed stream requires 8-9 bits per triangle
- Can potentially exploit coherence between triangles
  - 70% triangles have 1 index common with previous triangle
  - 40% triangles have 2 indices common with previous triangle
- Mesh representations by Chow et al. and Deering et al. take 20-40 bits per triangle
  - our scheme achieves ~2-4X better compression

Conclusions

Unified approach for cache efficiency and bandwidth reduction

Near optimal

Pre-processing required

Only applicable for parts of the model where order is not pre-determined

Acknowledgements

- Jonathan Cohen
- Budirijanto Purnomo
- Stanford University for various models
- Johns Hopkins University for tablet model
- NSF