degree

clustering

cliques

paths

Connected Components

Strong/Weak Connectivity

breadth-first search

Dijkstra

SYSM 6302

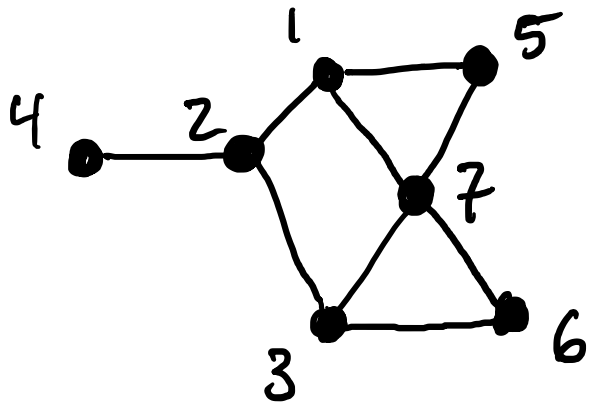
CLASS 5

Clustering Coefficient: a statistic that captures how closely connected "friends-of-friends" are.

Local Clustering:

$$C_i = \frac{\text{△}}{\text{∠}} = \frac{\#\text{ of connected pairs of neighbors}}{\#\text{ pairs of neighbors}}$$

Typically orientation is dropped in the case of directed networks.



Node 4: 1 neighbor $\Rightarrow$ $\emptyset$ pairs of neighbors

Node 2: 3 neighbors $\Rightarrow$ $\binom{3}{2} = 3$ pairs of nbrs.

$\{1,3\}, \{1,4\}, \{3,4\}$

$\Rightarrow$ None closed! $\Rightarrow C_2 = 0$

Node 1: 3 nbrs

$\Rightarrow$ 3 pairs of nbrs

$\{2,5\}, \{2,7\}, \{5,7\}$

$C_1 = \frac{1}{3}$

Node 5: 2 nbrs

$\Rightarrow$ 1 pair of nbrs

$\{1,7\}$

$C_5 = \frac{1}{1} = 1$

Node 7: 4 nbrs

$\Rightarrow$ 6 pairs of nbrs

$\{1,3\}, \{1,5\}, \{1,6\}, \{3,5\},$

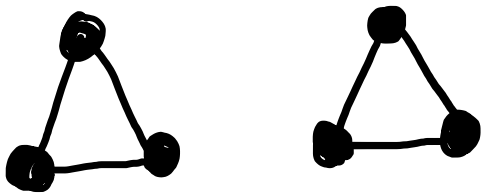
$\{3,6\}, \{5,6\}$

$C_7 = \frac{2}{6} = \frac{1}{3}$

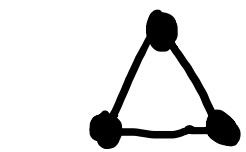
Global Clustering

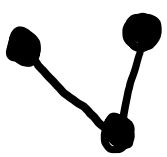
(the connection between local & global clustering is dependent on the network structure so no fixed relationship exists)

$$C = \frac{\triangle}{\angle} = \frac{\#\text{ of closed paths of length 2}}{\#\text{ of paths of length 2}}$$



$$= \frac{\#\text{ triangles} \times 6}{\#\text{ of connected triples} \times 2}$$





transitivity: (u,v) and $(u,w) \Longrightarrow (v,w)$

(think about "=": $u=v$ and $u=w \Rightarrow v=w$)

Reciprocity the shorter version of clustering

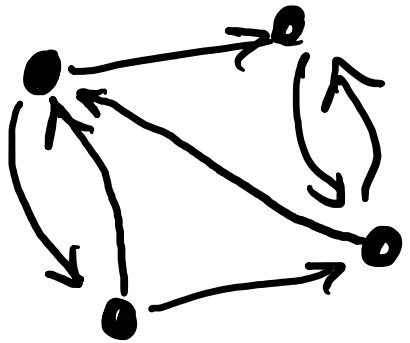
$i \to j$ then $j \to i$ the edge is reciprocated

As a statistic of a network:

$$R = \frac{\#\text{ of reciprocated edges}}{\#\text{ of edges}}$$

should be even!

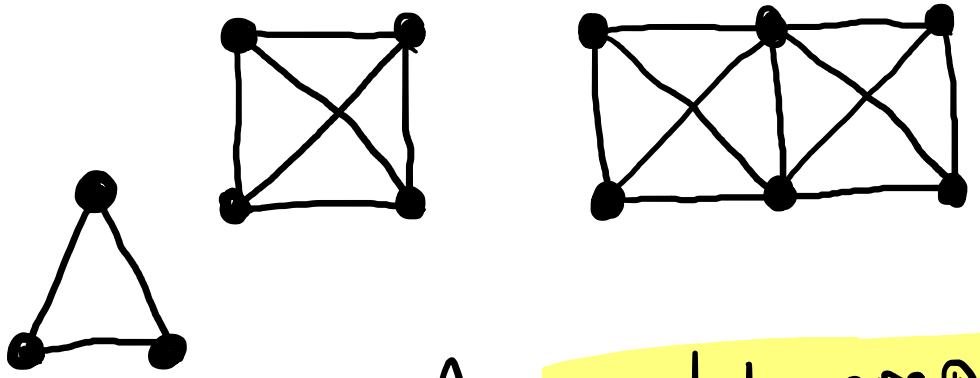
$R = \frac{4}{7}$



Cliques

A maximal subset of vertices such that each vertex is connected by an edge to every other vertex

No other node could be added to the subset while preserving the property



How many edges? $\binom{n}{2} = \frac{n(n-1)}{2}$

there are n nodes

each has n-1 neighbors (possible edges)

A **complete graph** is a giant clique!

k-plex of size n is a maximal subset such that each vertex is connected by an edge to at least n-k other vertices

k-core is a maximal subset such that each vertex is connected to at least k other vertices.

(many others)

Connectivity (finally!)

A graph is connected if it is possible to travel between any two nodes along the edges of the graph

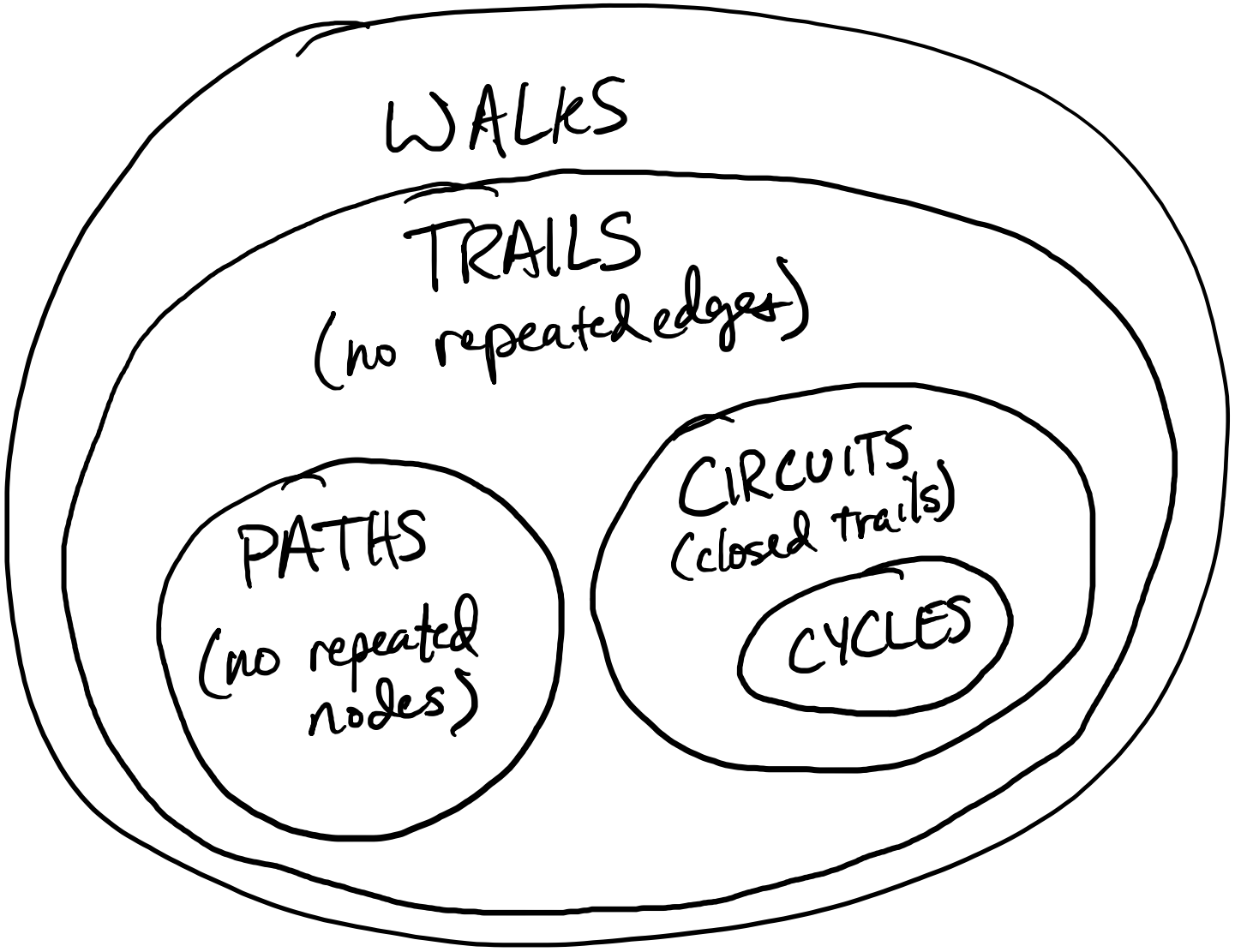
moving between nodes along incident edges

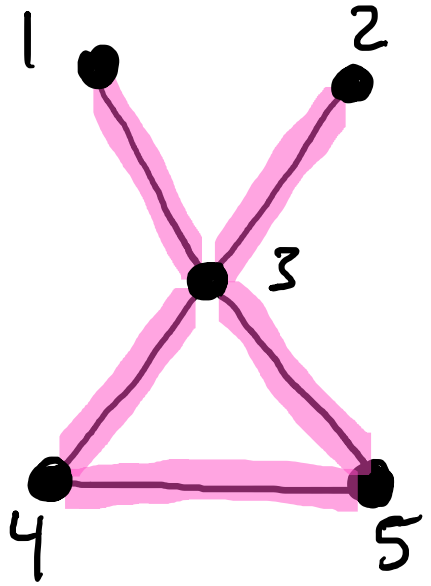
walk uv : sequence of nodes and edges starting at u and ending at v

walk is **closed** if $u=v$.

trail if edges are distinct
circuit if closed

path if nodes are distinct
cycle if closed





(1, 3, 5, 4, 2) is a trail, but not a path

Components are maximal induced subgraphs such that the subgraph is connected.

→ a solitary vertex is a component!

→ Because the definition is maximal, each vertex belongs to <u>only one</u> component

Weak connectivity for directed graphs means the underlying graph is connected.

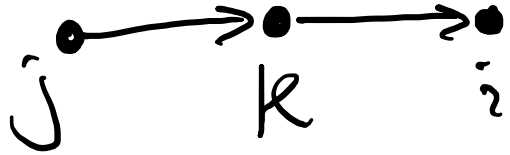
→ Use this to define **weak components** of a directed graph

Strong connectivity & **strong components** respect edge orientation

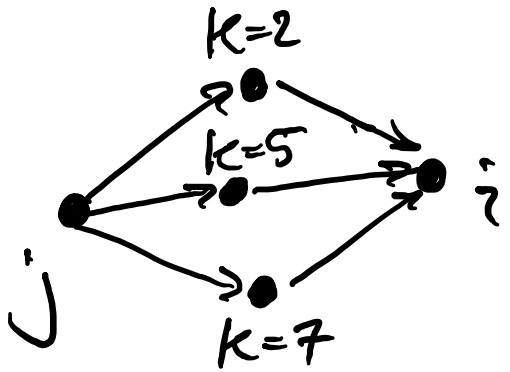
Paths with algebra!

$[A^r]_{ij}$ nonzero if $\exists$ an r-length path from j to i

$$A_{ik} A_{kj} =$$



$$\sum_{k=1}^{n} A_{ik} A_{kj} =$$



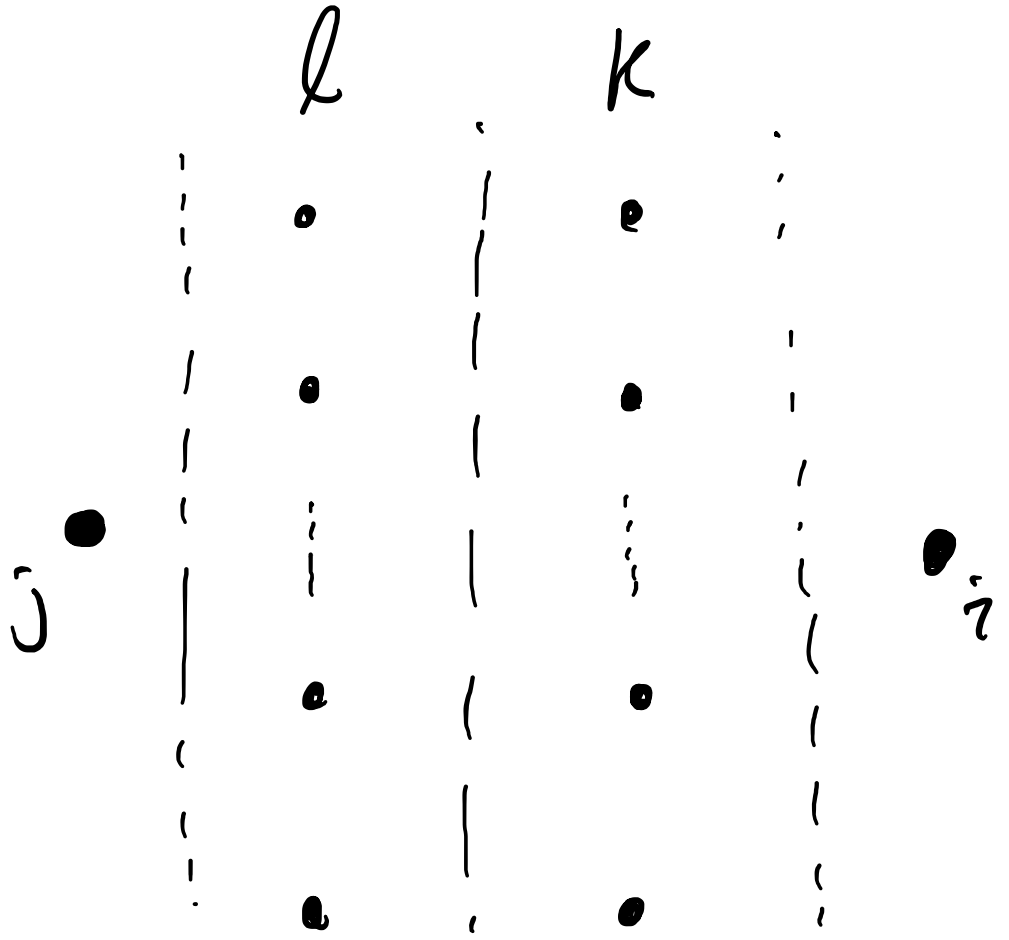
$= N_{ij} =$ # of paths available from j to i $= [A^2]_{ij}$

Assuming $A \in \{0,1\}$

$$\sum_{k=1}^{n} \sum_{l=1}^{n} A_{ik} A_{kl} A_{lj}$$

$$= \sum_{k=1}^{n} A_{ik} \underbrace{\sum_{l=1}^{n} A_{kl} A_{lj}}_{[A^2]_{kj}}$$

$$= [A^3]_{ij}$$



Loops with algebra!

$[A^r]_{ii} \neq 0 \Rightarrow$ r-length loop including node i

If $A_{ij} \in \{0,1\} \Rightarrow$ # of loops of length r $= \sum_{i=1}^{n} [A^r]_{ii} = \text{trace}(A^r)$

If graph is undirected $\Rightarrow A = A^T \Rightarrow A = U \Lambda U^T$, $U^T = U^{-1}$

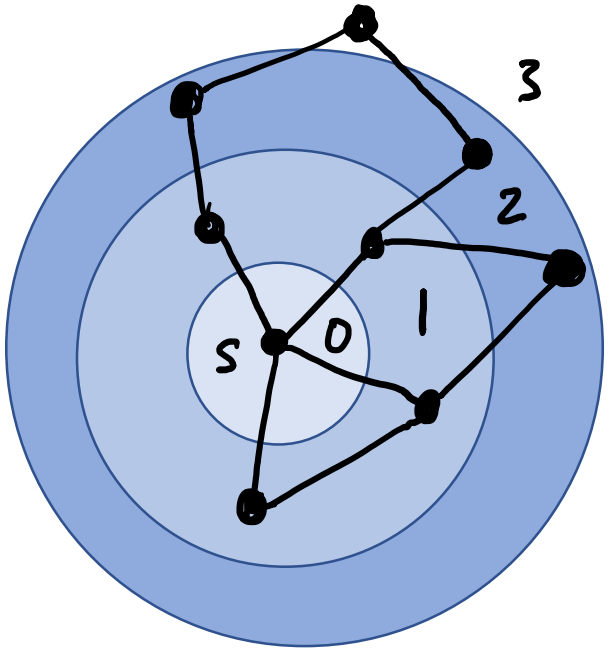
$$A^r = (U \Lambda U^T)^r = U \Lambda^r U^T$$

$$\text{trace}(A^r) = \text{trace}(U \Lambda^r U^T) = \text{trace}(U^T U \Lambda^r) = \text{trace}(\Lambda^r) = \sum_{i=1}^{n} \lambda_i^r$$

eigenvalues!

Breadth-First Search

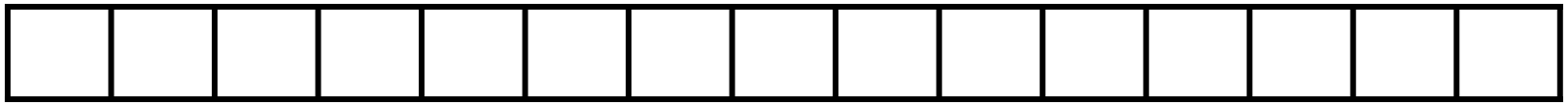
unweighted networks to find the shortest distance from a node s to every other node in the same component



Every node whose shortest path from s is length d has a neighbor node whose shortest path has length d-1.

the predecessor node on the shortest path from s

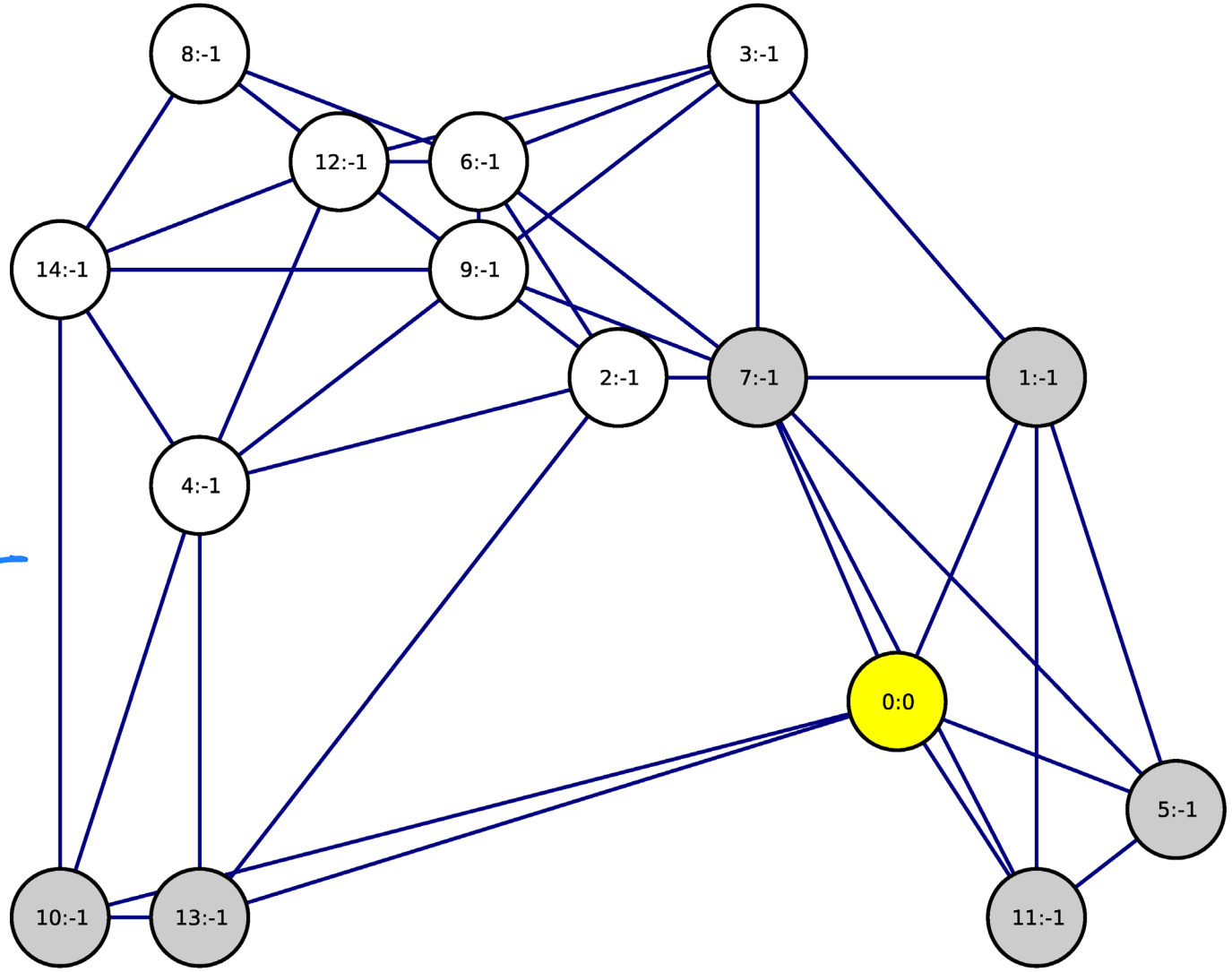
To keep track of the distances we know



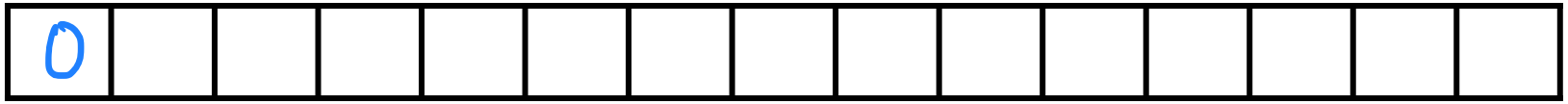
← indexed by node index

Example borrowed from Carl Kingsford @ CMU

Initialization Step

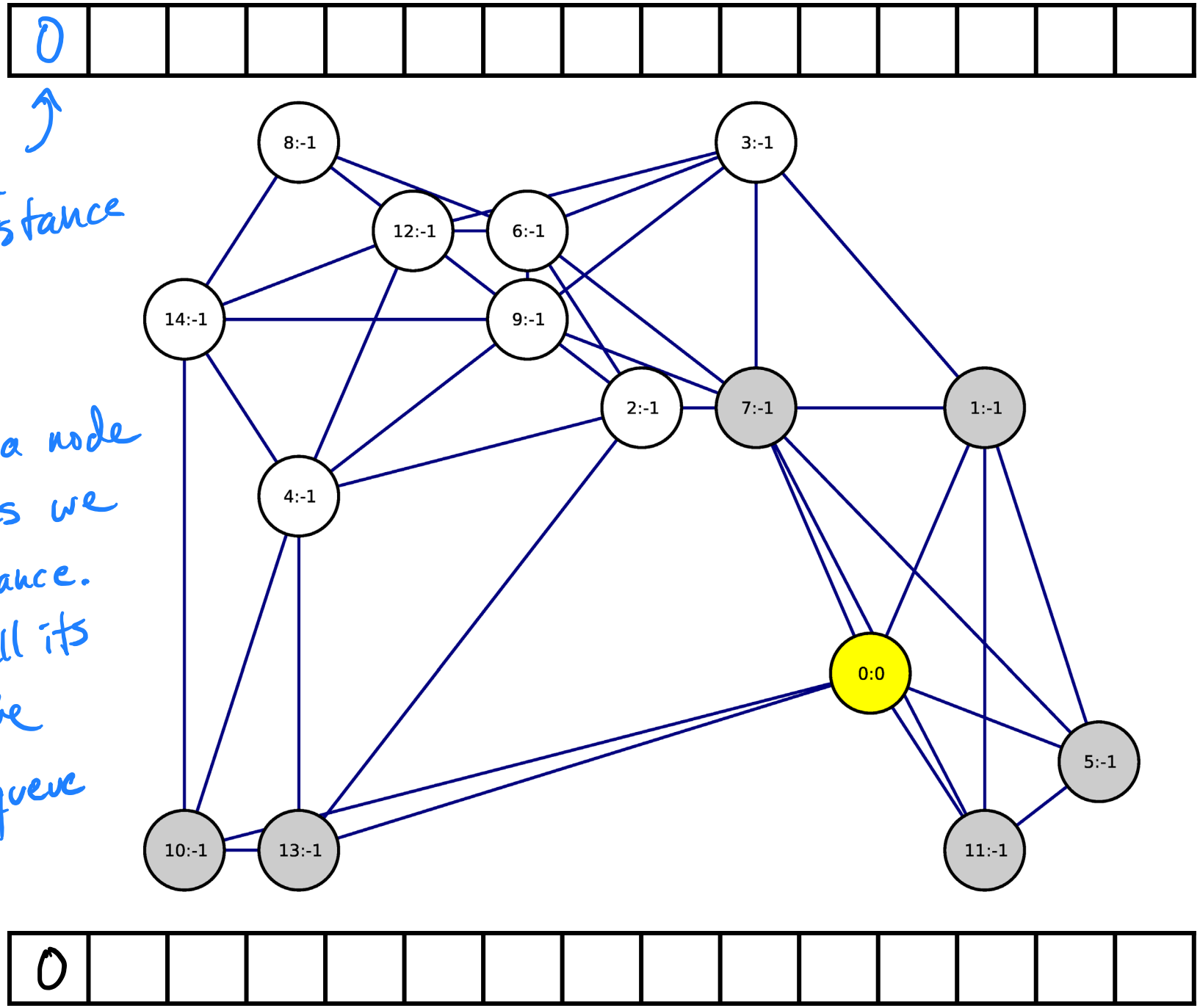


To keep track of what nodes we need to check

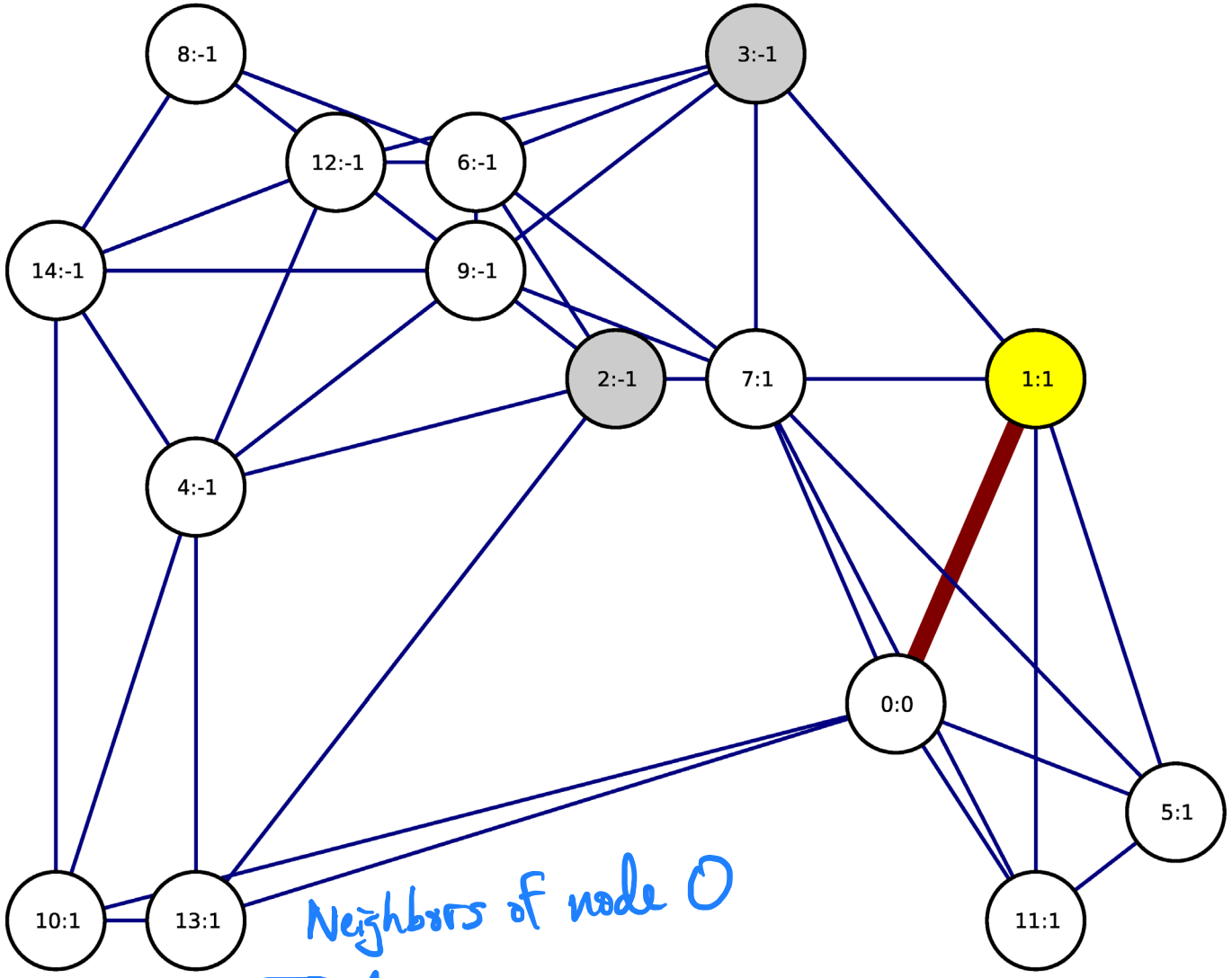


Fill it up as we go

← a queue

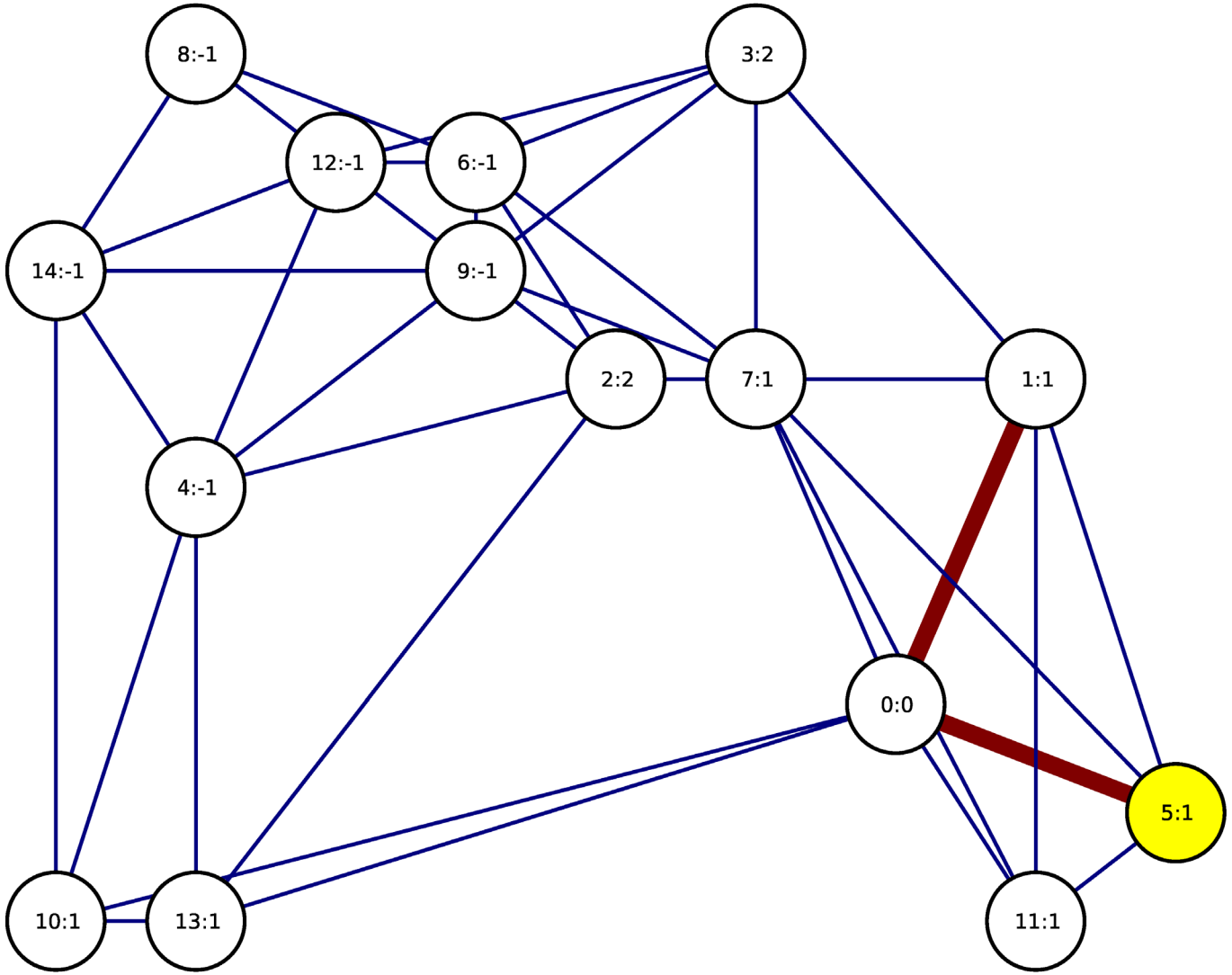

0
Known distance
When we add a node here it means we know its distance. Then we add all its neighbors to the "to-process" queue
8:-1
3:-1
12:-1
6:-1
14:-1
9:-1
2:-1
7:-1
1:-1
4:-1
0:0
5:-1
10:-1
13:-1
11:-1
0

0	1				1		1			1	1		1	



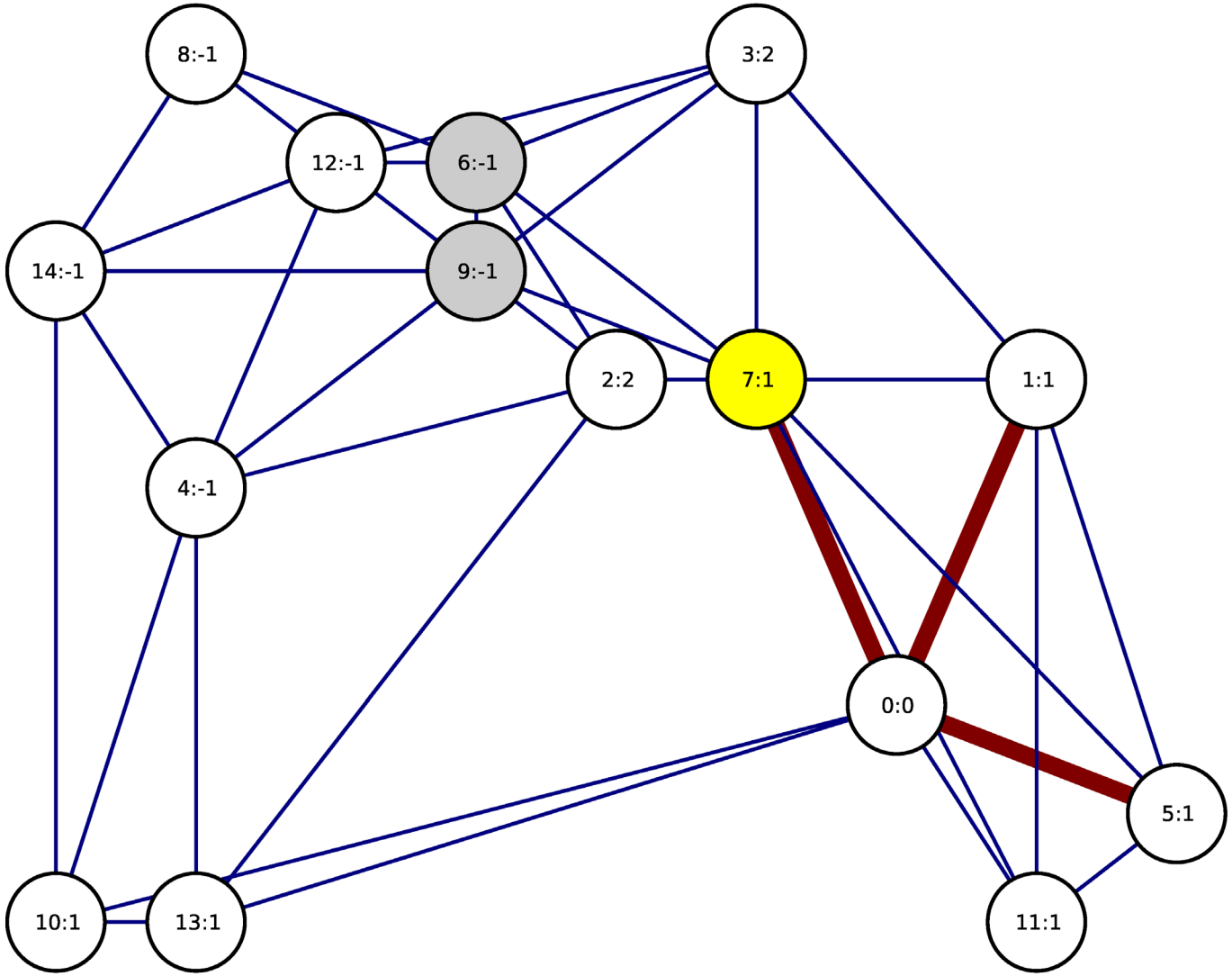
0	1	5	7	10	11	13								

0	1	2	2		1		1			1	1		1	



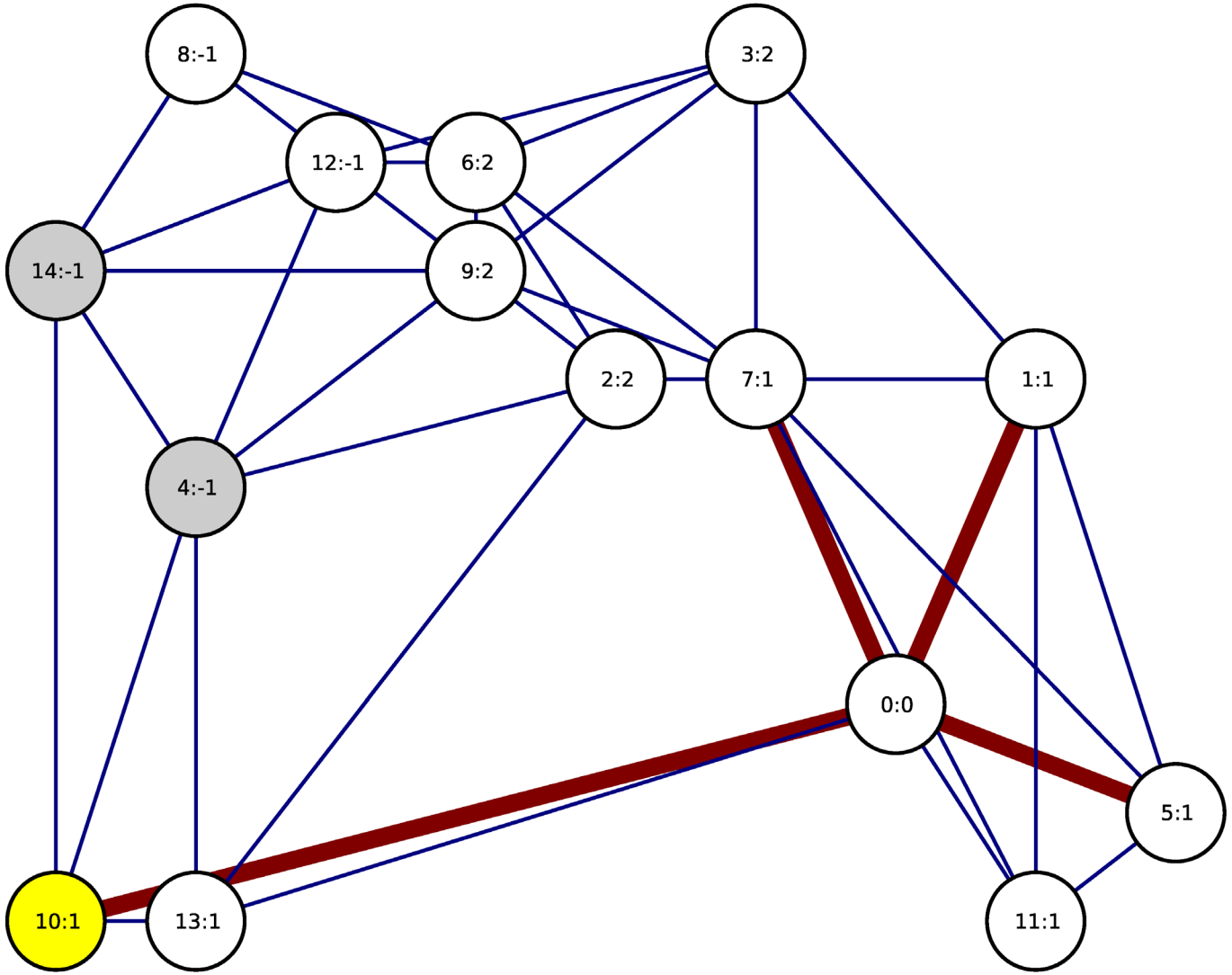
0	1	5	7	10	11	13	2	3						

0	1	2	2		1		1			1	1		1	



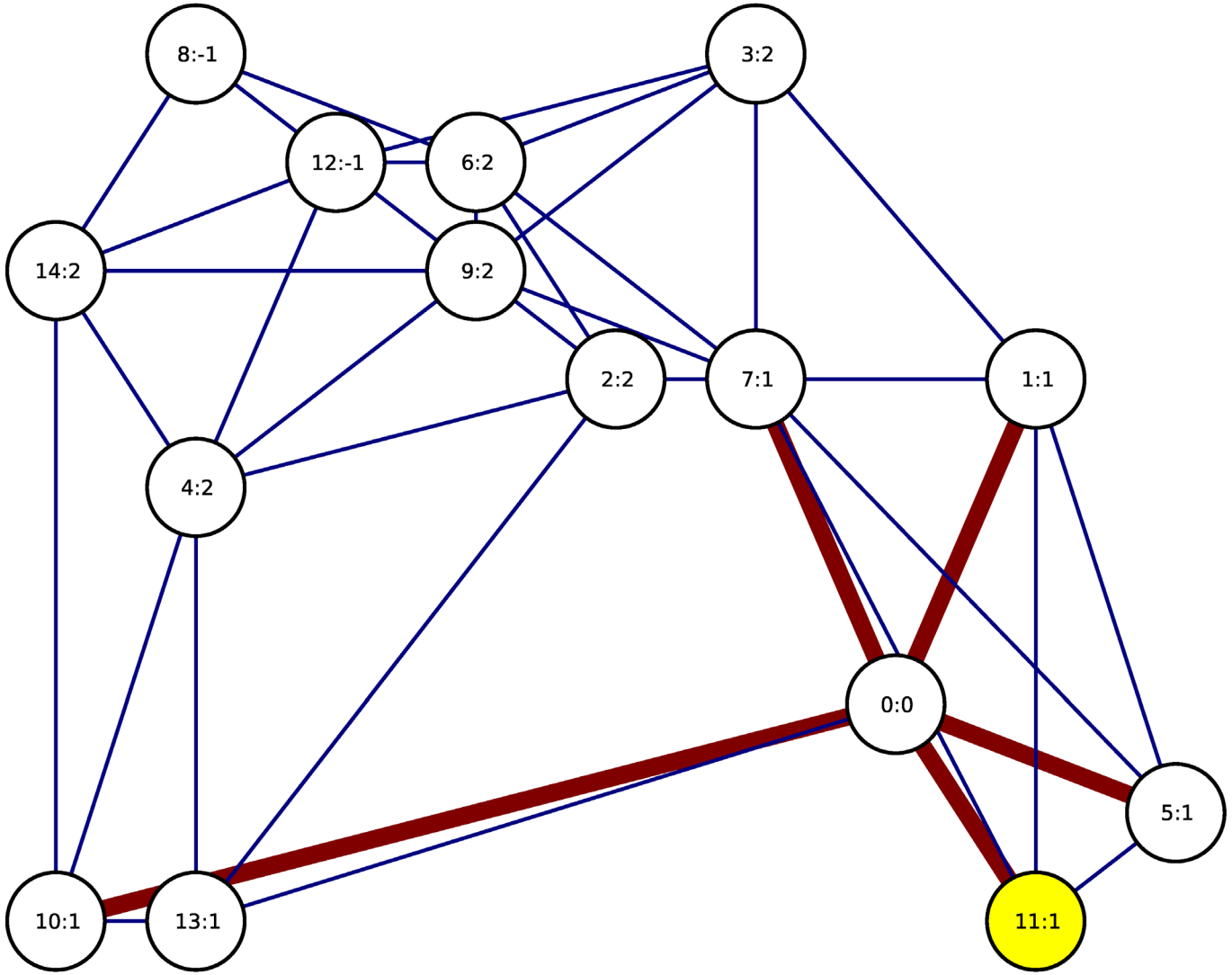
0	1	5	7	10	11	13	2	3						

0	1	2	2		1	2	1		2	1	1		1	



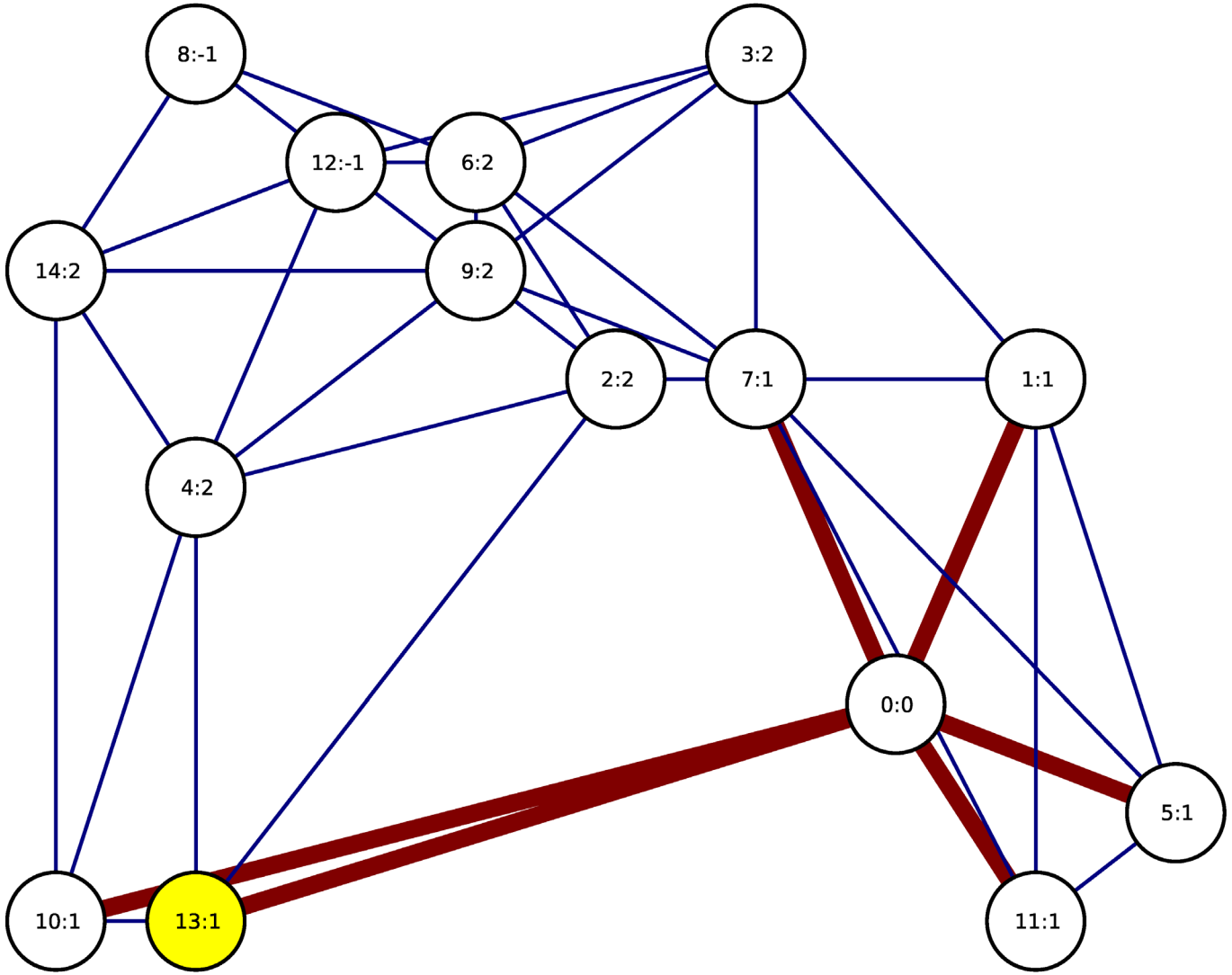
0	1	5	7	10	11	13	2	3	6	9				

0	1	2	2	2	1	2	1		2	1	1		1	2



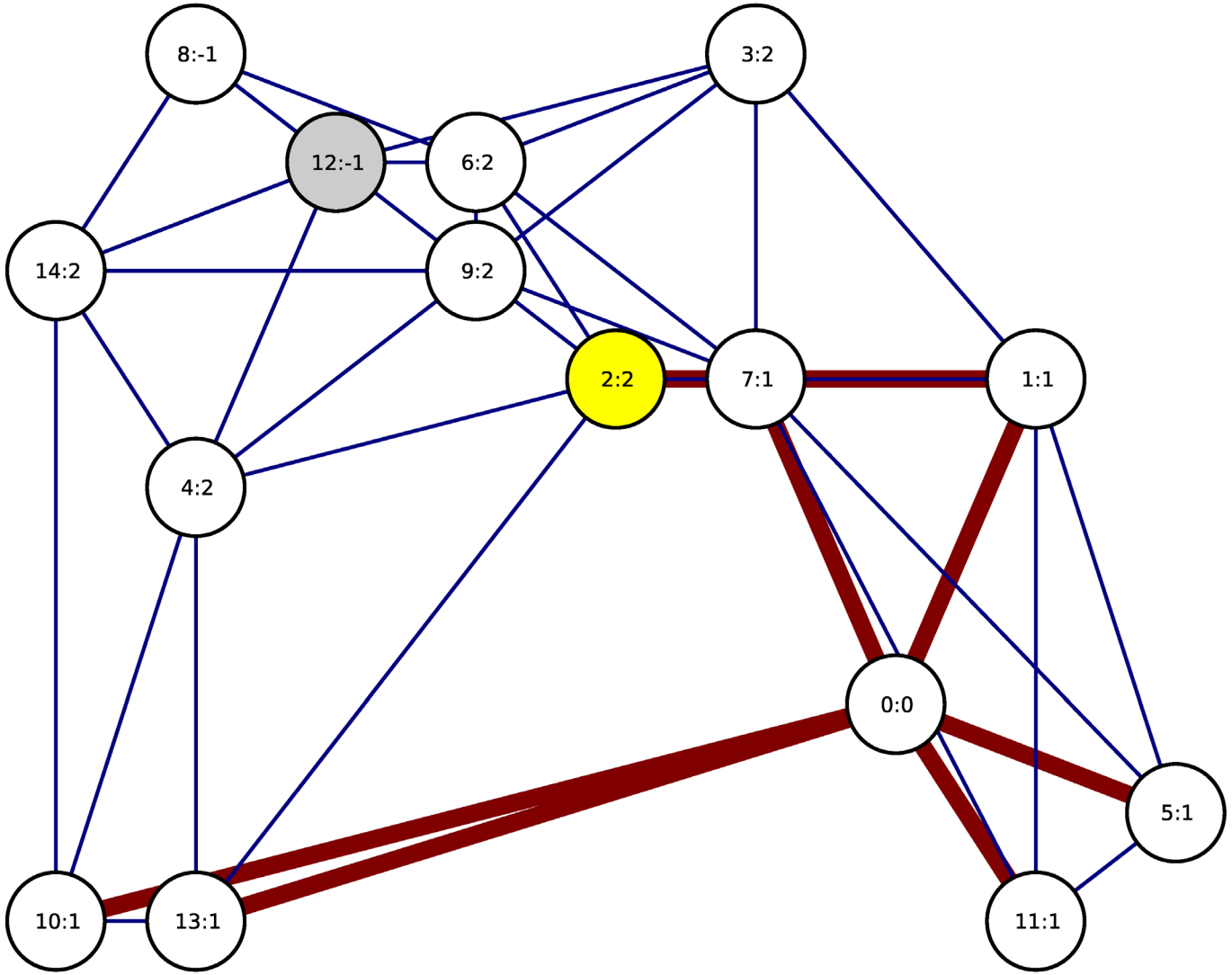
0	1	5	7	10	11	13	2	3	6	9	4	14		

0	1	2	2	2	1	2	1		2	1	1		1	2



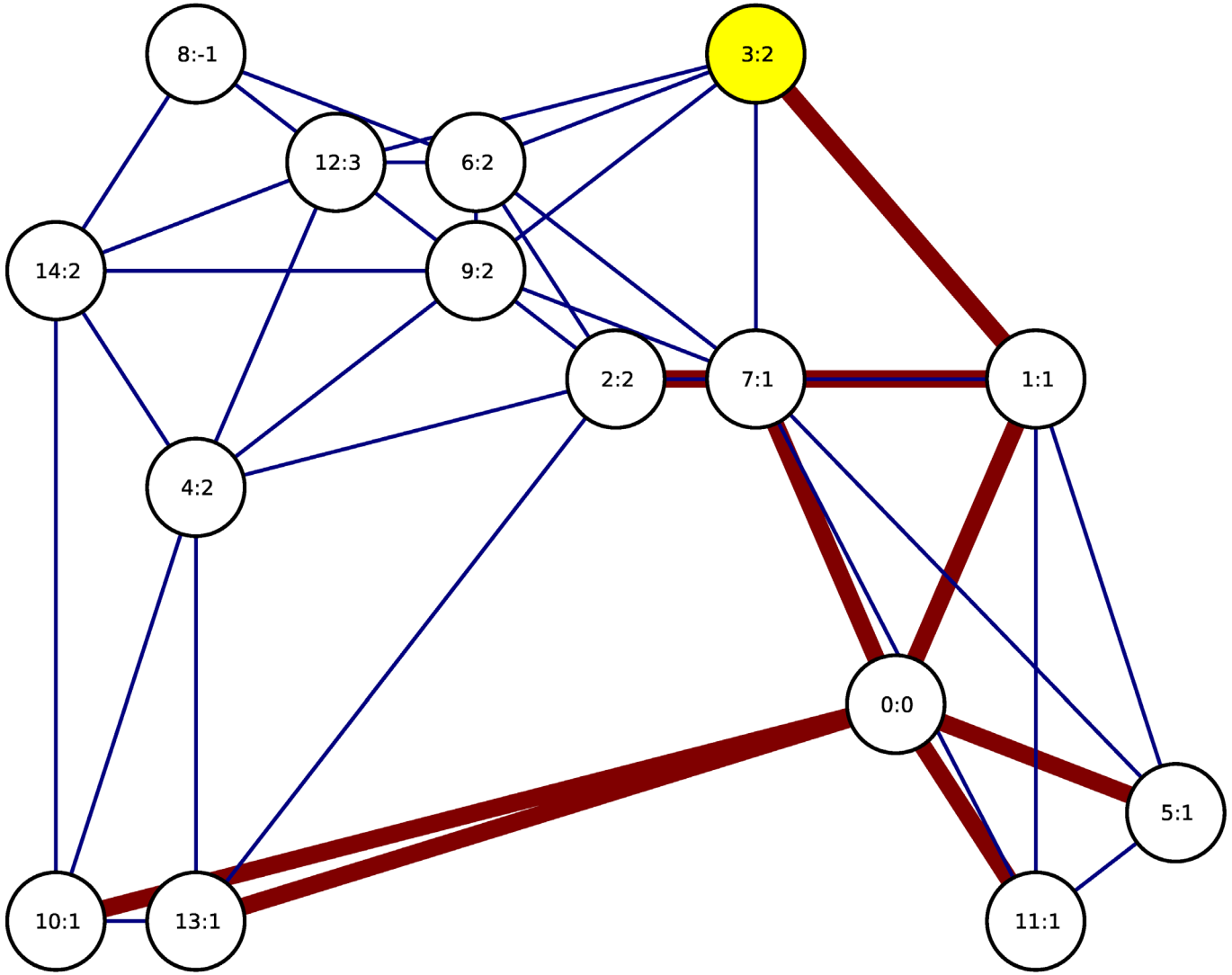
0	1	5	7	10	11	13	2	3	6	9	4	14		

0	1	2	2	2	1	2	1		2	1	1		1	2



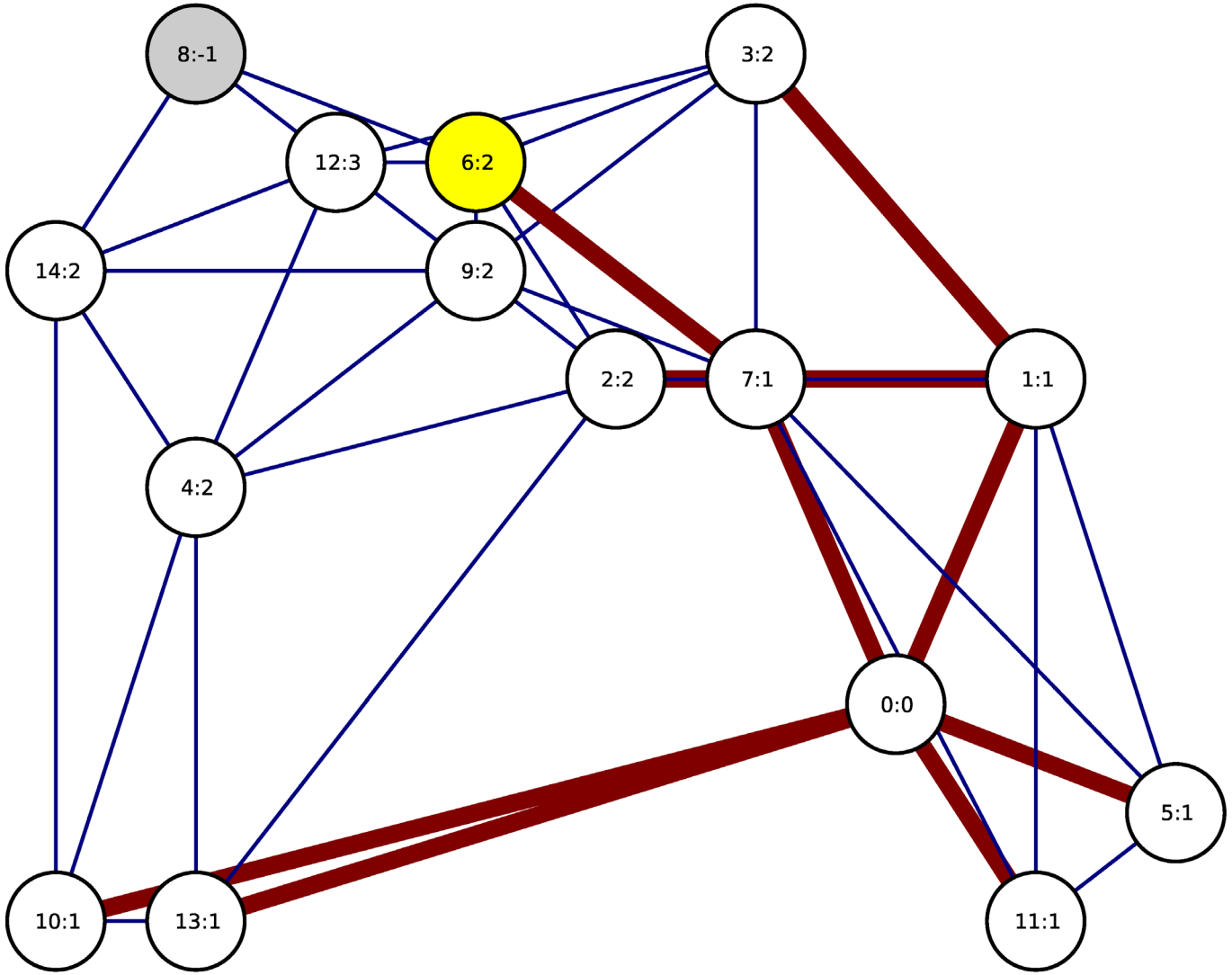
0	1	5	7	10	11	13	2	3	6	9	4	14		

0	1	2	2	2	1	2	1		2	1	1	3	1	2



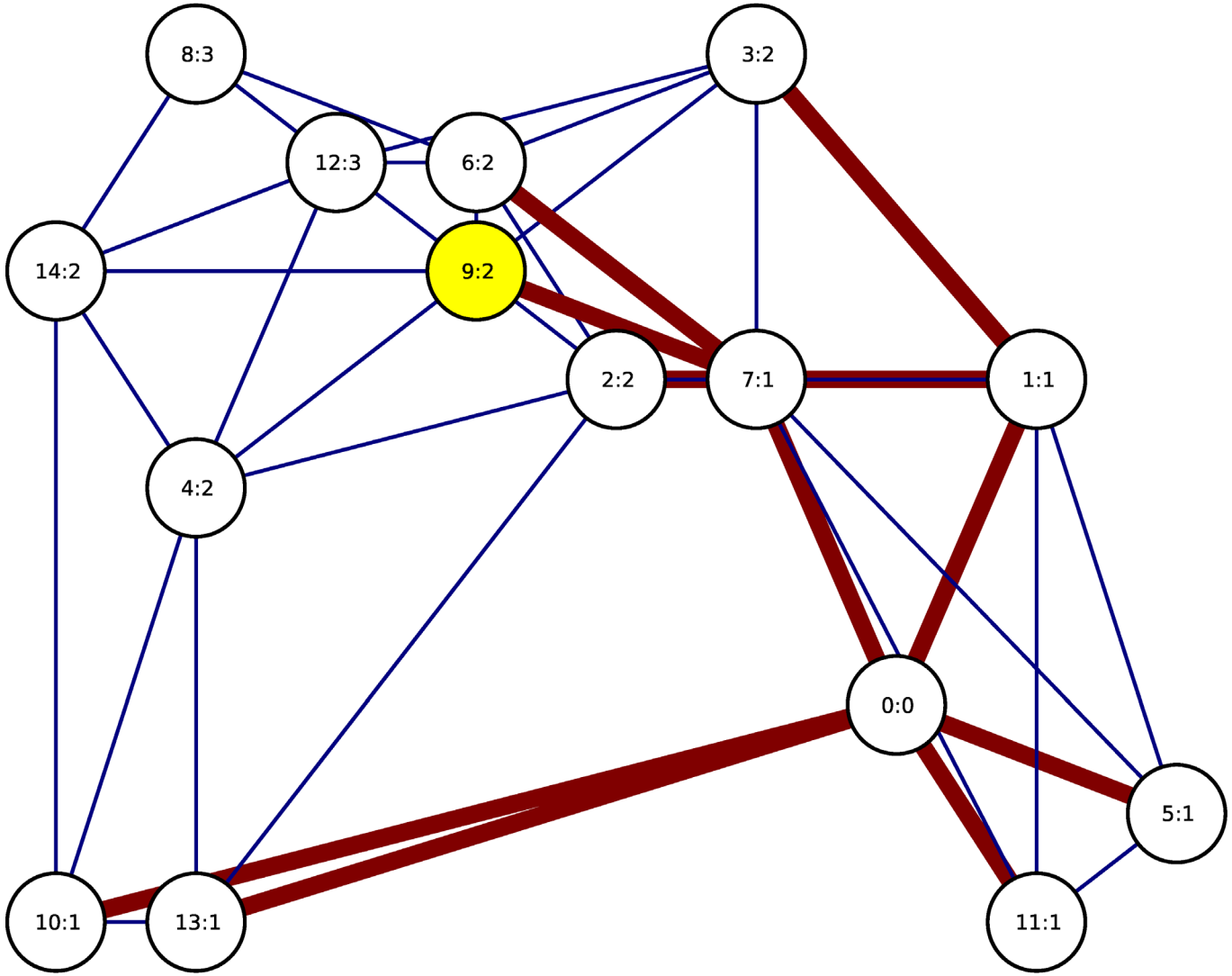
0	1	5	7	10	11	13	2	3	6	9	4	14	12	

0	1	2	2	2	1	2	1		2	1	1	3	1	2



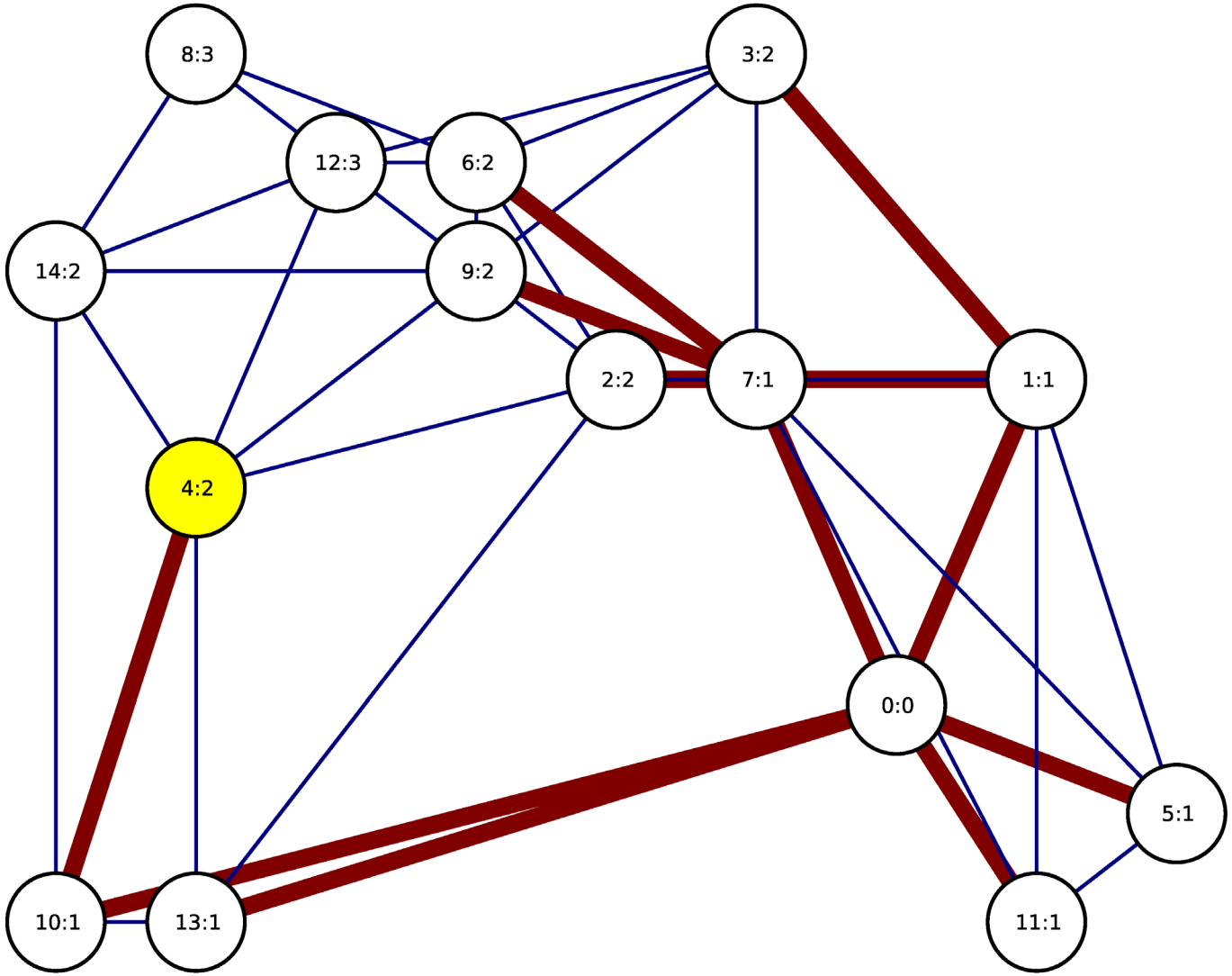
0	1	5	7	10	11	13	2	3	6	9	4	14	12	

0	1	2	2	2	1	2	1	3	2	1	1	3	1	2



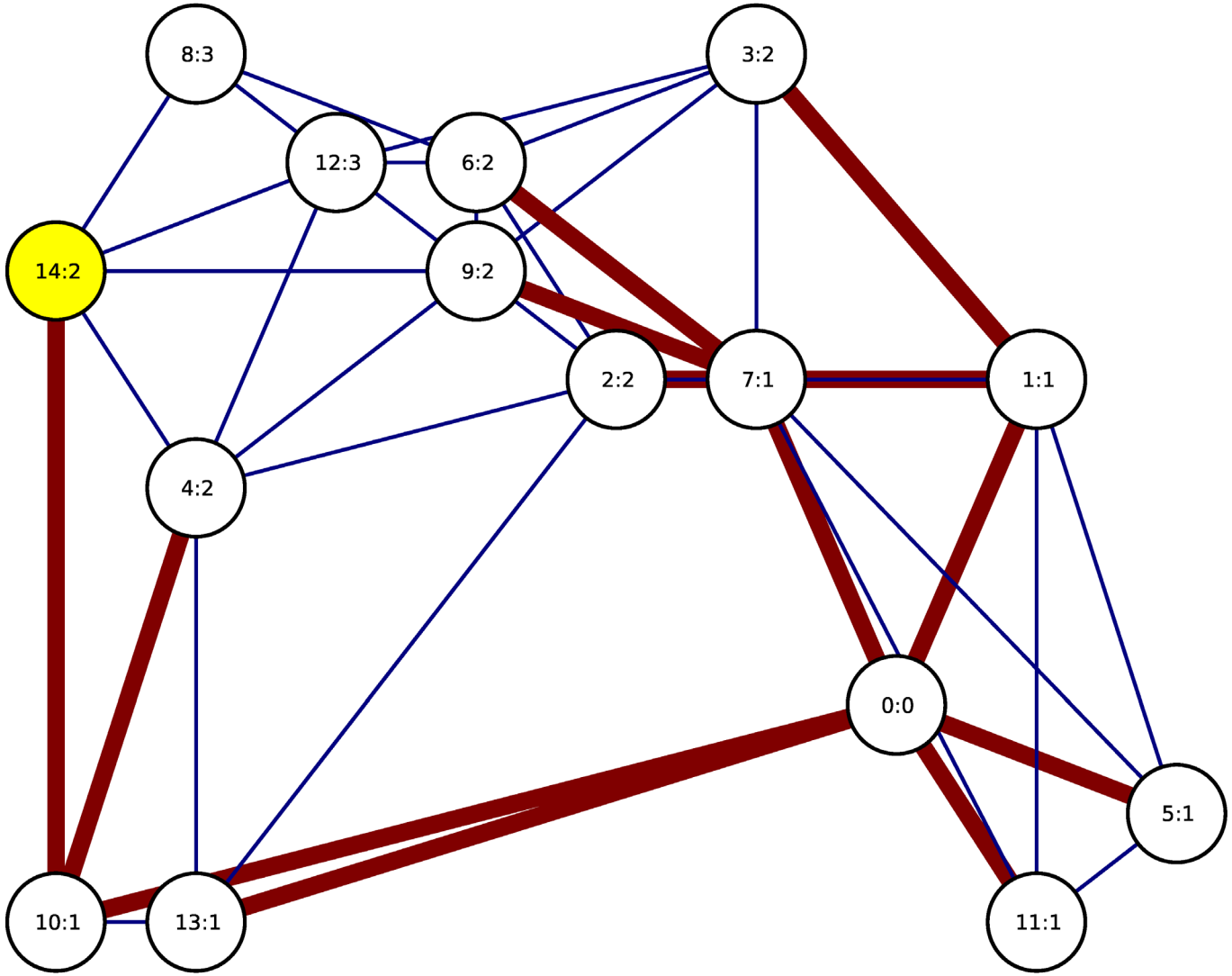
0	1	5	7	10	11	13	2	3	6	9	4	14	12	8

0	1	2	2	2	1	2	1	3	2	1	1	3	1	2



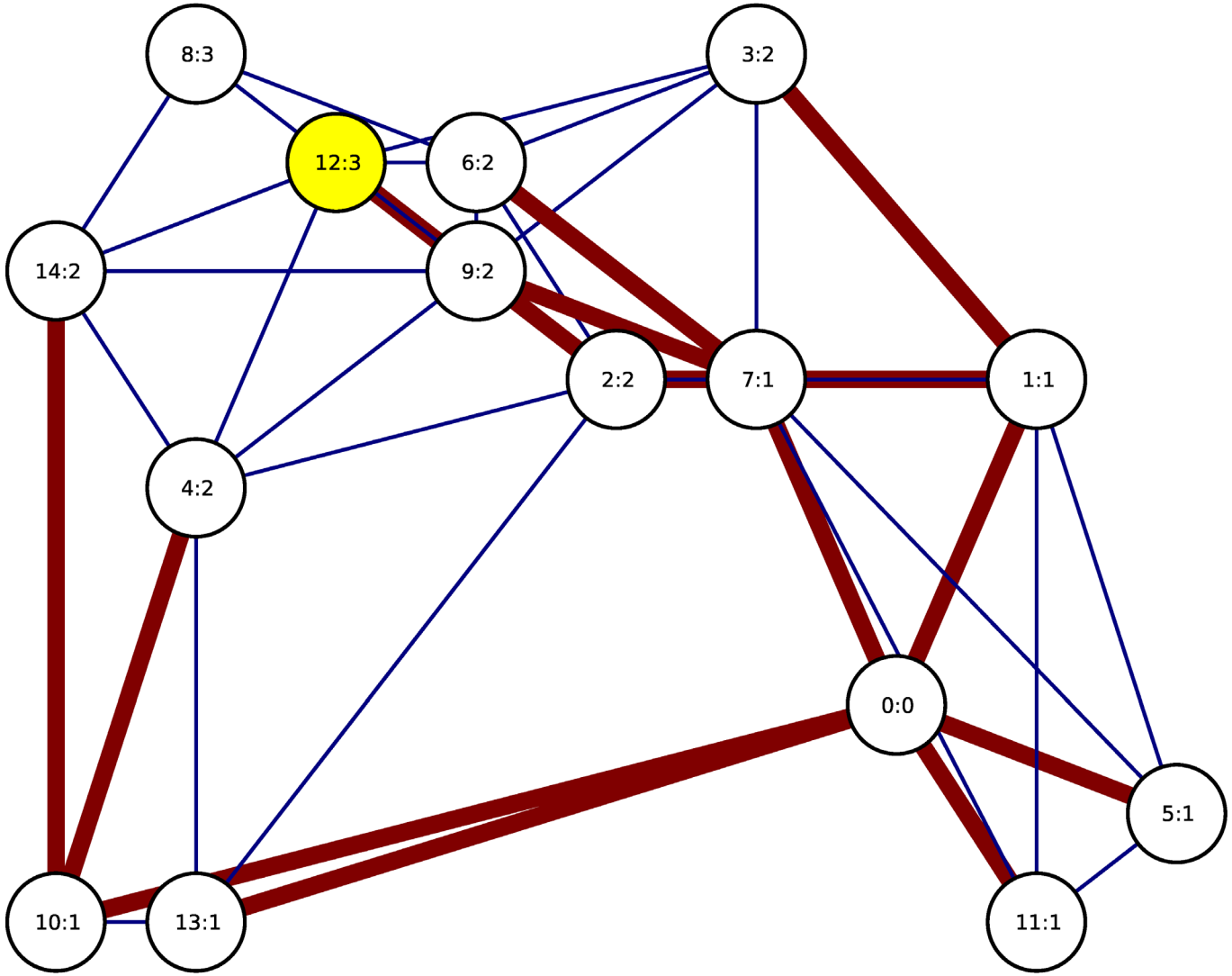
0	1	5	7	10	11	13	2	3	6	9	4	14	12	8

0	1	2	2	2	1	2	1	3	2	1	1	3	1	2



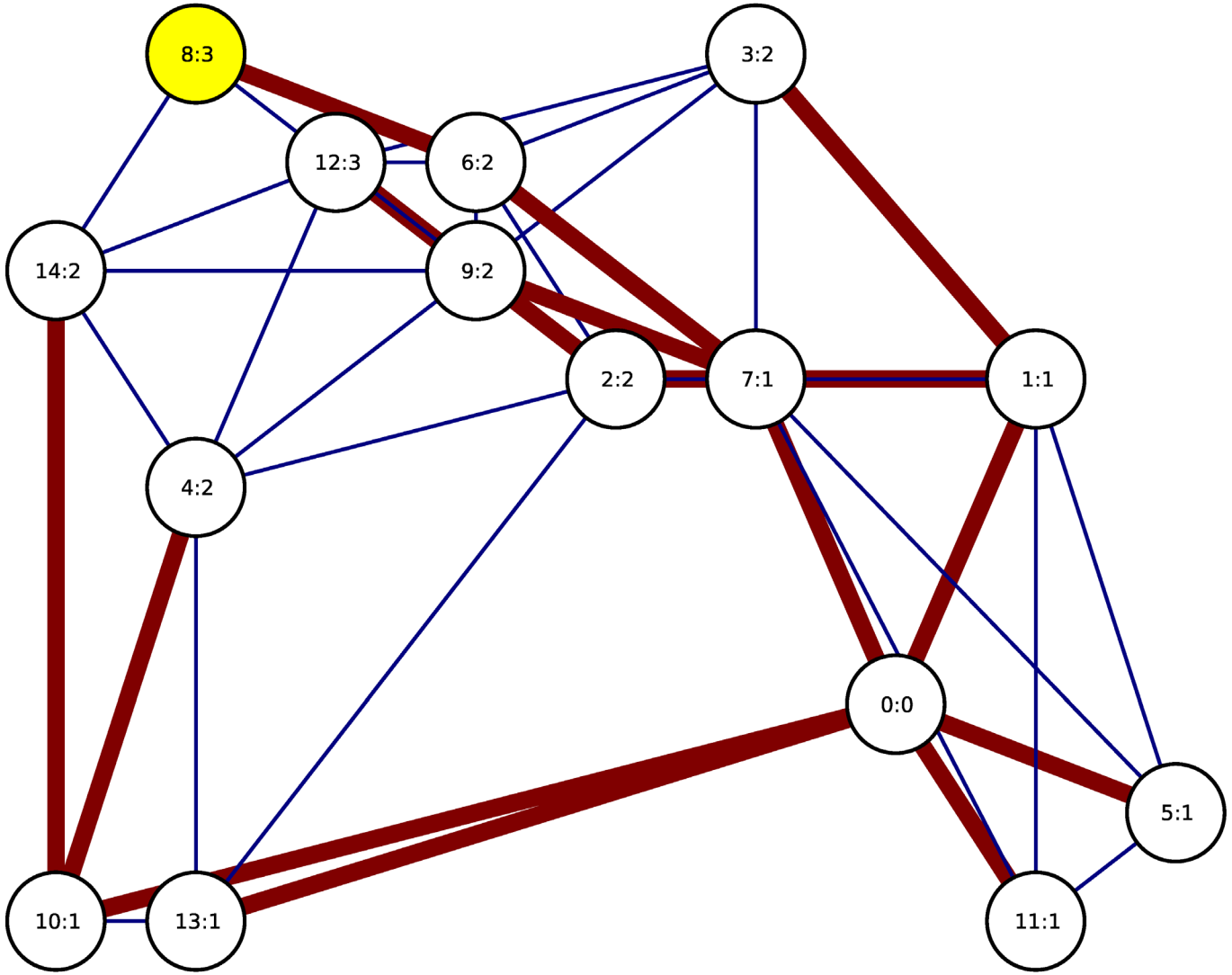
0	1	5	7	10	11	13	2	3	6	9	4	14	12	8

0	1	2	2	2	1	2	1	3	2	1	1	3	1	2

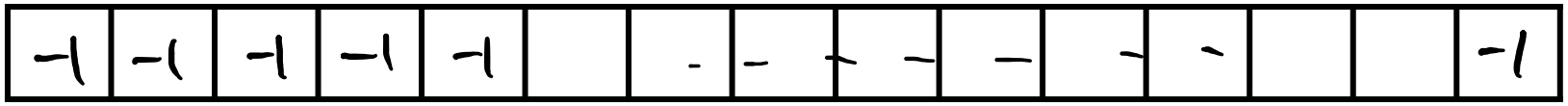


0	1	5	7	10	11	13	2	3	6	9	4	14	12	8

0	1	2	2	2	1	2	1	3	2	1	1	3	1	2



0	1	5	7	10	11	13	2	3	6	9	4	14	12	8



Needed to initialize this array first → n operations

Computation Time?

(How many operations are needed?)

$\Rightarrow O(n + n \cdot \frac{m}{n}) = O(n+m)$

we do the following n times:

check all the neighbors of the current node

on average $\sim \frac{2m}{n}$ neighbors

don't need to initialize the queue because we always write before we read

As implemented BFS finds the shortest paths from s to all other nodes

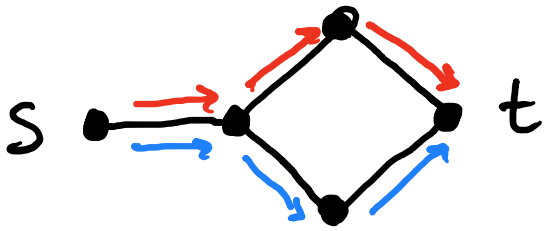
↳ Just s to t ? $O(m+n)$ (in the worst case)

↳ All-pairs ? $O(n(m+n))$

↳ Find components

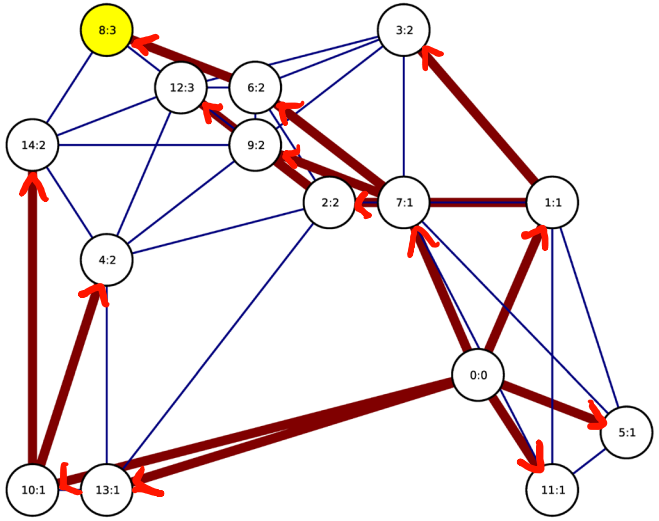
The subgraph of predecessor paths/nodes is the **shortest path tree**

↳ this is used to reconstruct the actual shortest path (not just length)

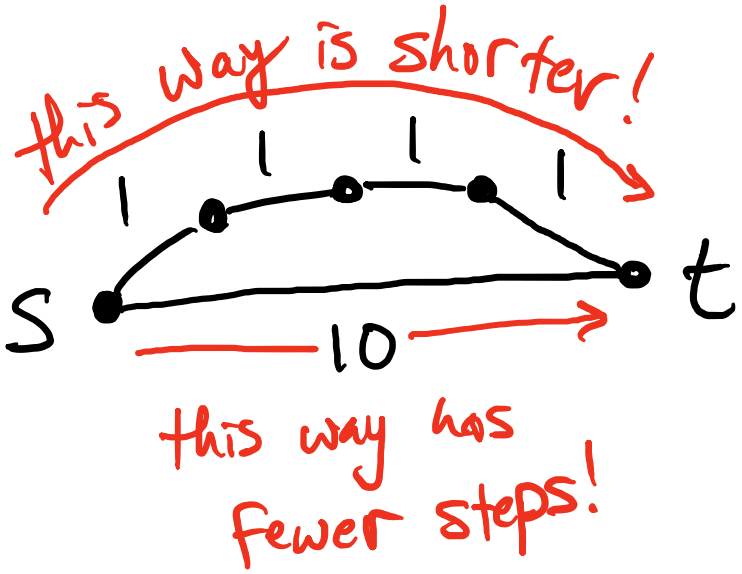


multiple shortest paths!

shortest path "tree" is most generally an acyclic graph



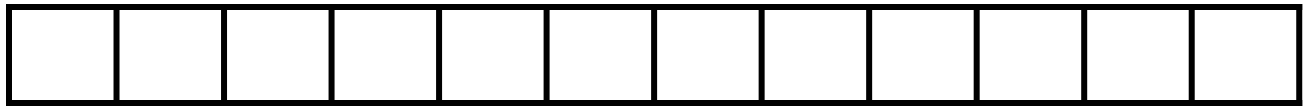
Shortest Path with weights!



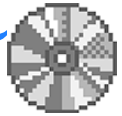
Implementation is like BFS, with more bookkeeping

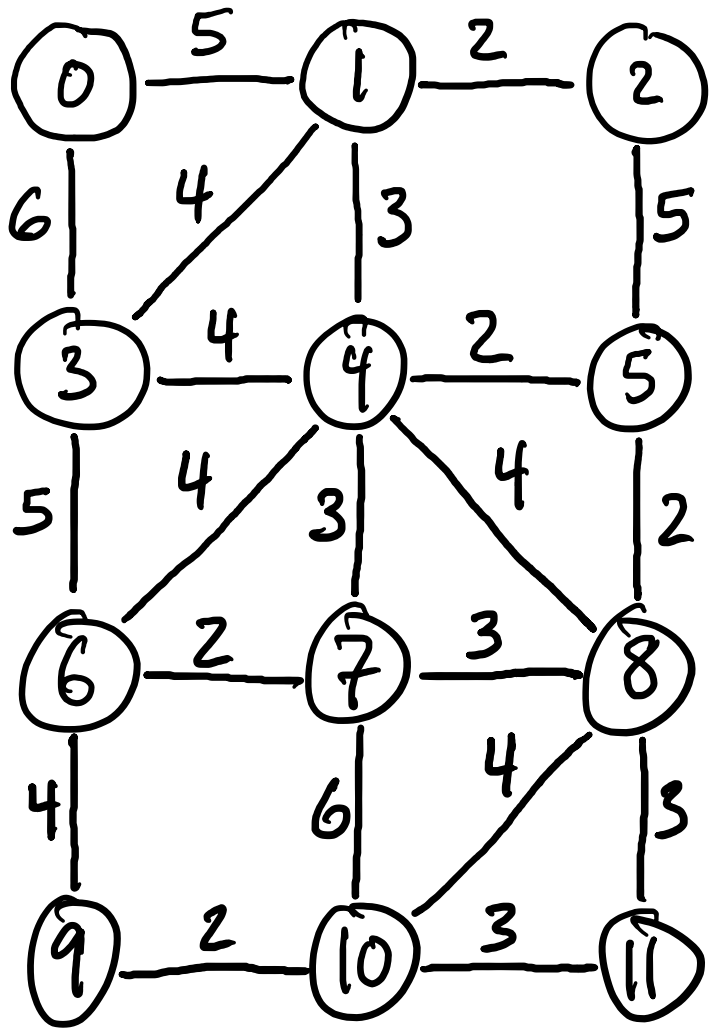
Dijkstra

upper bound on the shortest distance from s →



← indexed by node



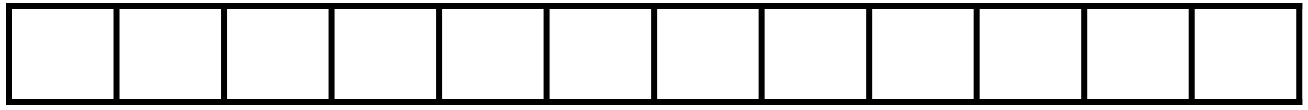


1) Find the vertex with the smallest estimated distance.

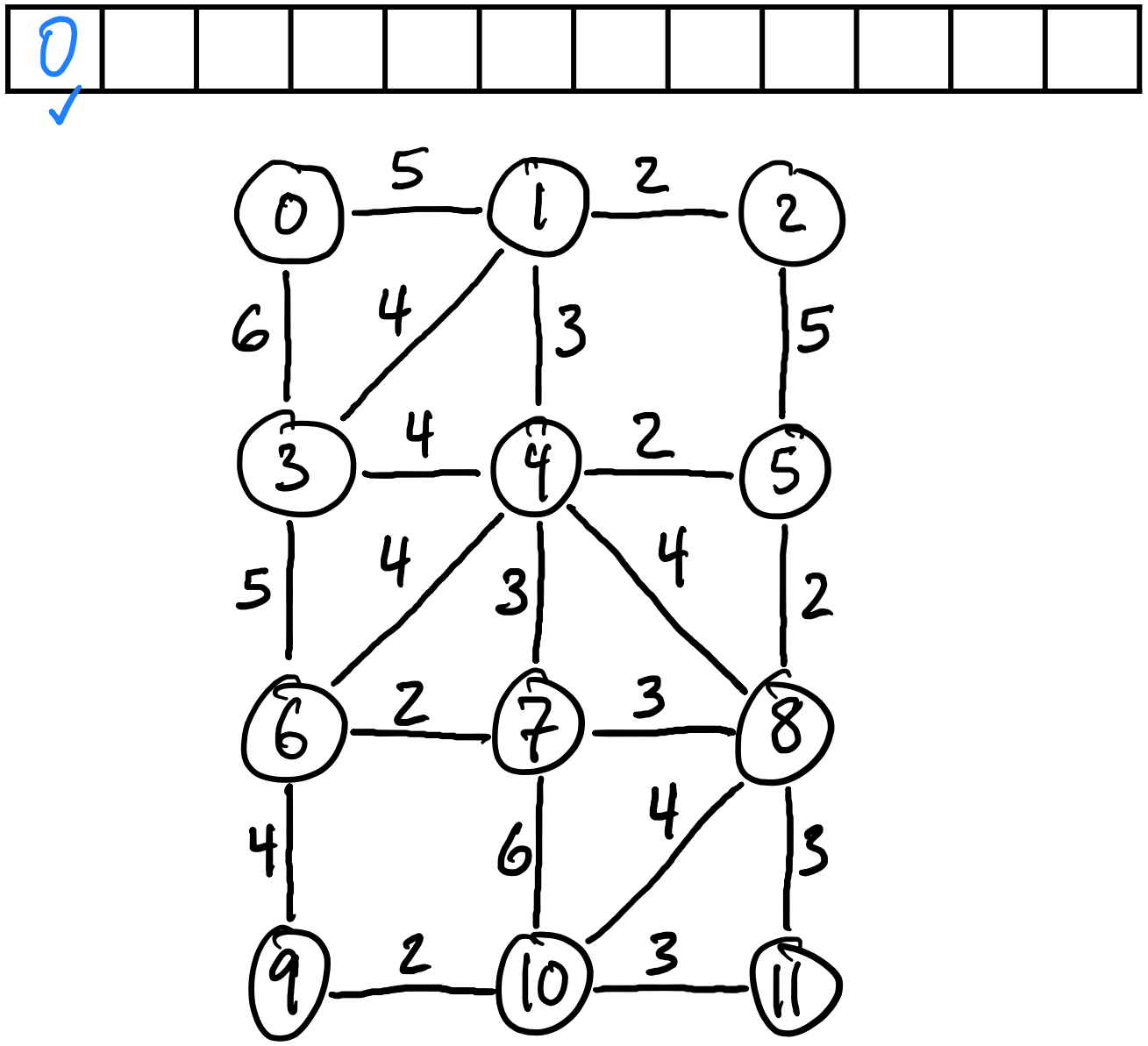
2) Mark this distance as certain.

3) update the estimates of all the neighbors of this node (if the new distance is smaller)

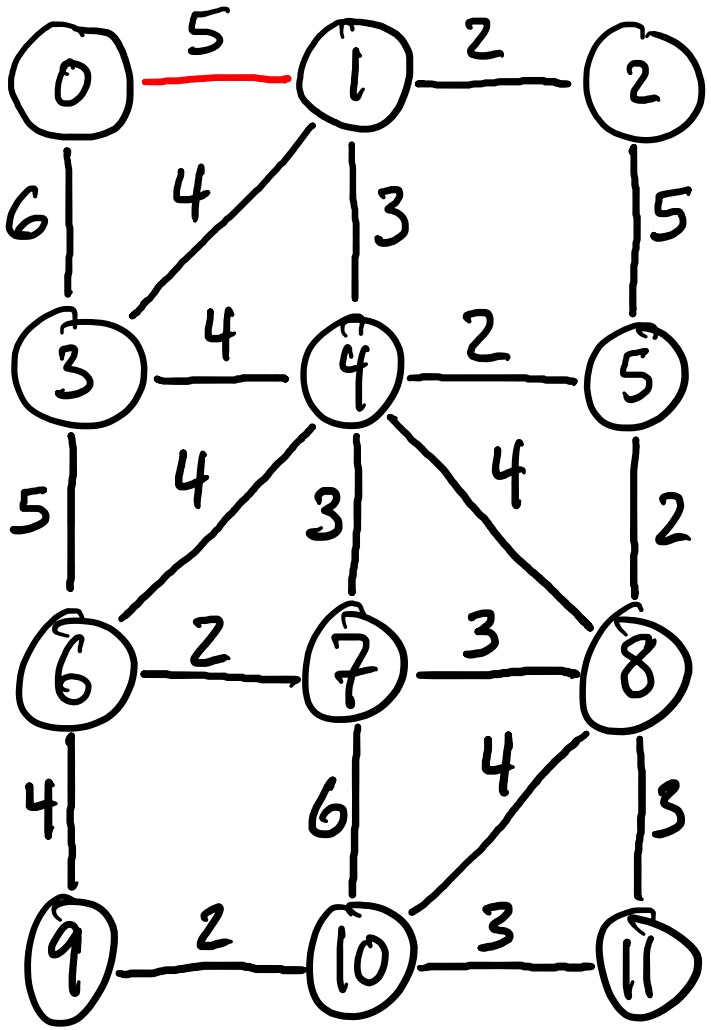
queue of nodes we need to "process" →



← Fill it up as we go


0
0
5
2
6
4
3
5
4
2
5
4
3
4
2
5
4
3
2
2
3
4
6
4
3
9
2
10
3
11
0

0	5		6								



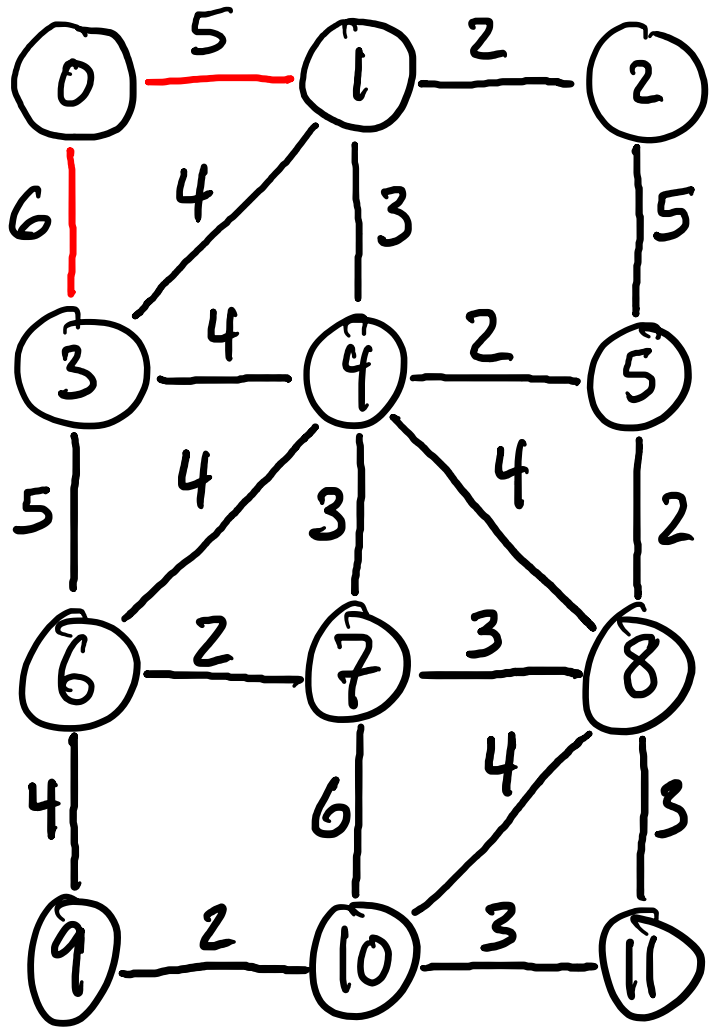
Neighbors of 0:

0 → 1 : 5 ⇐

0 → 3 : 6

0	1										

0	5	7	6	8							



Neighbors of nodes in queue:

0→3 : 6 ⇐

1→2 : 5+2

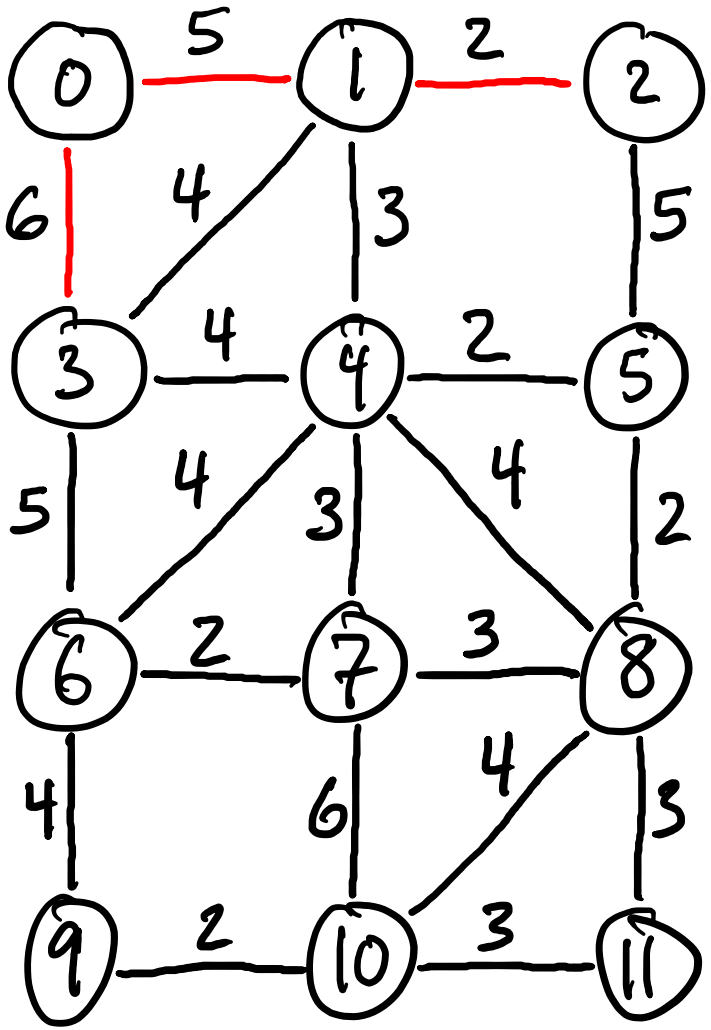
1→4 : 5+3

~~1→3 : 5+4~~

Known distance to node 1

0	1	3									

0	5	7	6	8		11					



Neighbors of nodes in queue:

1 → 2 : 5+2 ⇐

1 → 4 : 5+3

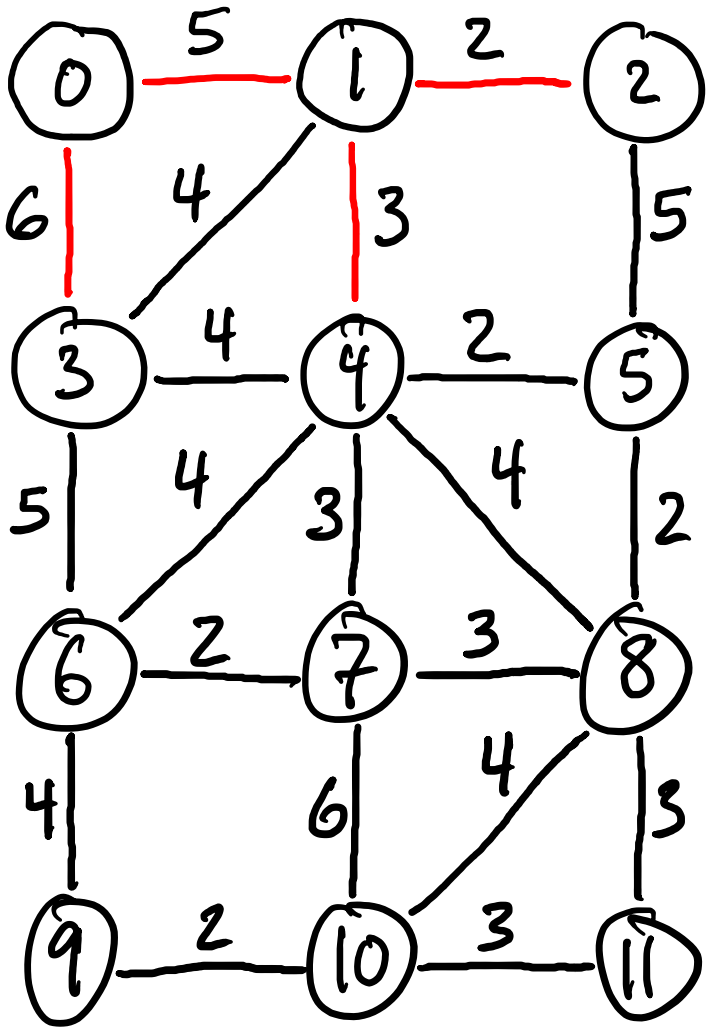
~~3 → 4 : 6+4~~

3 → 6 : 6+5

↑ known distance to node 3

0	1	3	2								

0	5	7	6	8	12	11					



Neighbors of nodes in queue:

1 → 4: 5+3 ⇐

~~3 → 4: 6+4~~

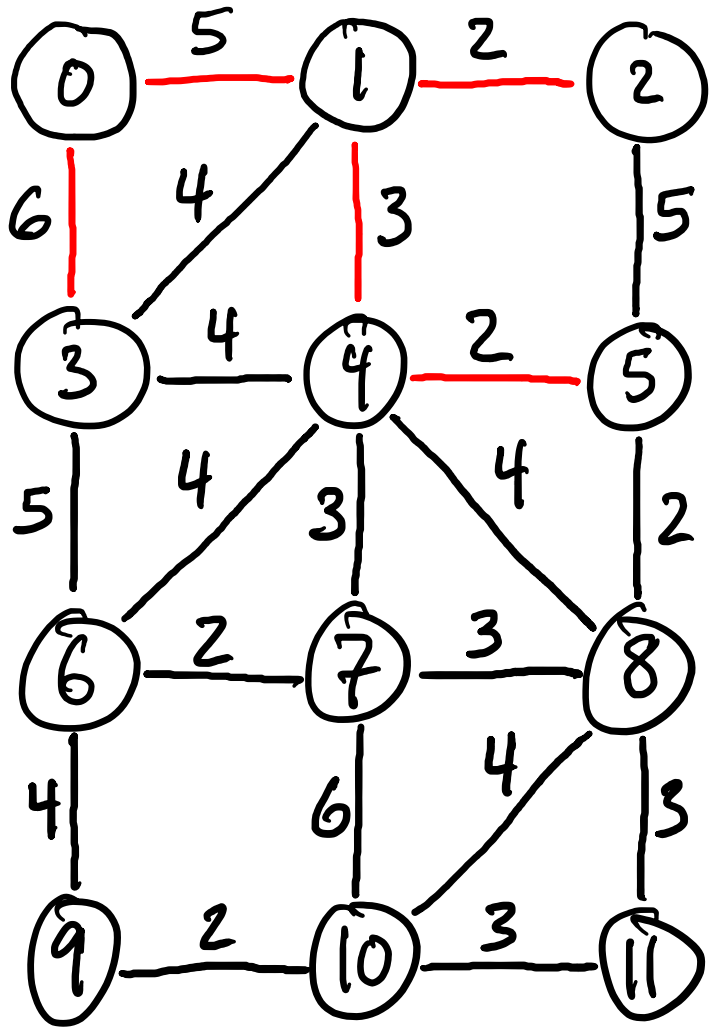
3 → 6: 6+5

2 → 5: 7+5

↑ known distance to node 2

0	1	3	2	4							

0	5	7	6	8	10	11	11	12			



Neighbors of nodes in queue:

3 → 6 : 6+5

~~2 → 5 : 7+5~~

4 → 5 : 8 + 2 ⇐

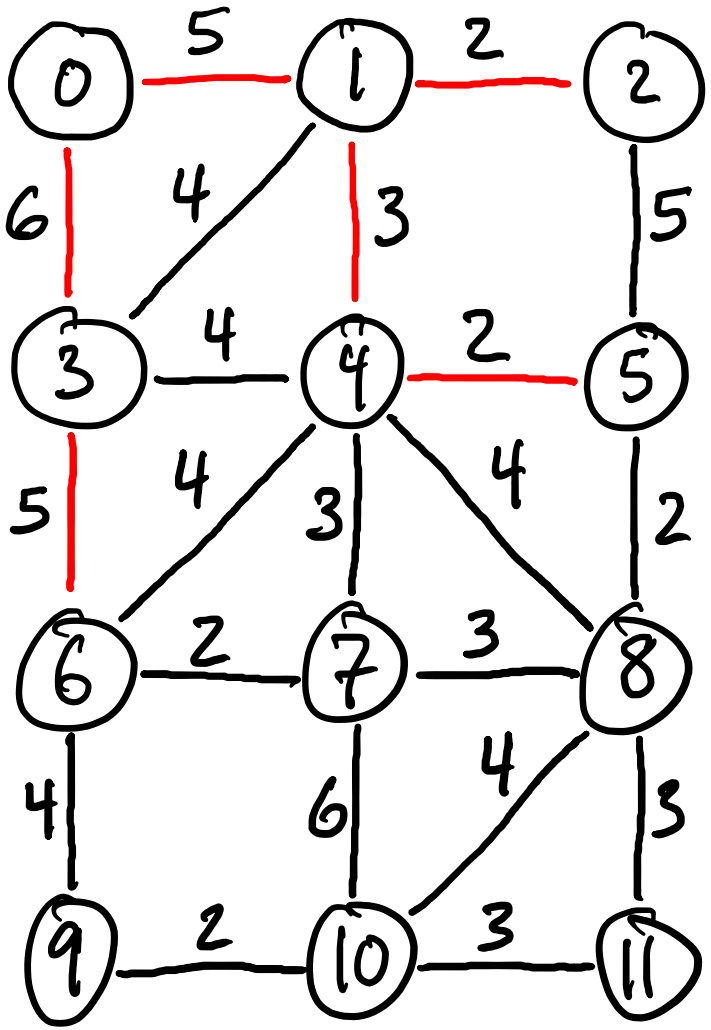
~~4 → 6 : 8 + 4~~

4 → 7 : 8 + 3

4 → 8 : 8 + 4

0	1	3	2	4	5						

0	5	7	6	8	10	11	11	12			



Neighbors of nodes in queue:

3 → 6 : 6+5 ⇐

~~4 → 6 : 8 + 4~~

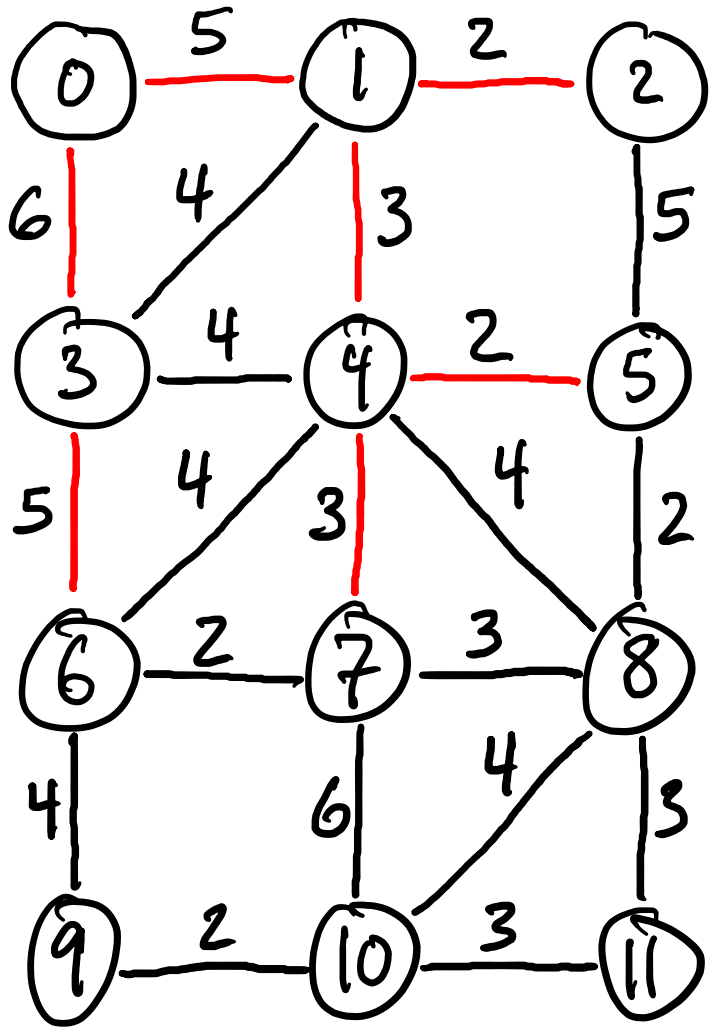
4 → 7 : 8 + 3

4 → 8 : 8 + 4

5 → 8 : 10 + 2

0	1	3	2	4	5	6					

0	5	7	6	8	10	11	11	12	15		



Neighbors of nodes in queue:

$4 \to 7: 8+3 \Leftarrow$

$4 \to 8: 8+4$

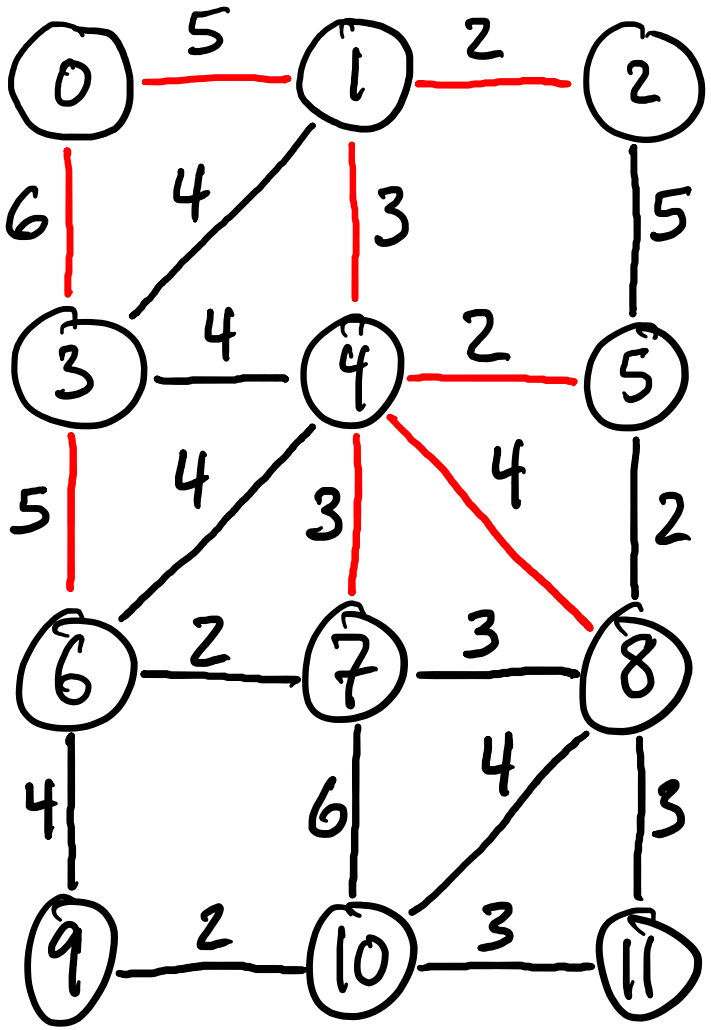
$5 \to 8: 10+2$

~~$6 \to 7: 11+2$~~

$6 \to 9: 11+4$

0	1	3	2	4	5	6	7				

0	5	7	6	8	10	11	11	12	15	17	



Neighbors of nodes in queue:

4 → 8 : 8 + 4 ⇐

5 → 8 : 10 + 2 ⇐

6 → 9 : 11 + 4

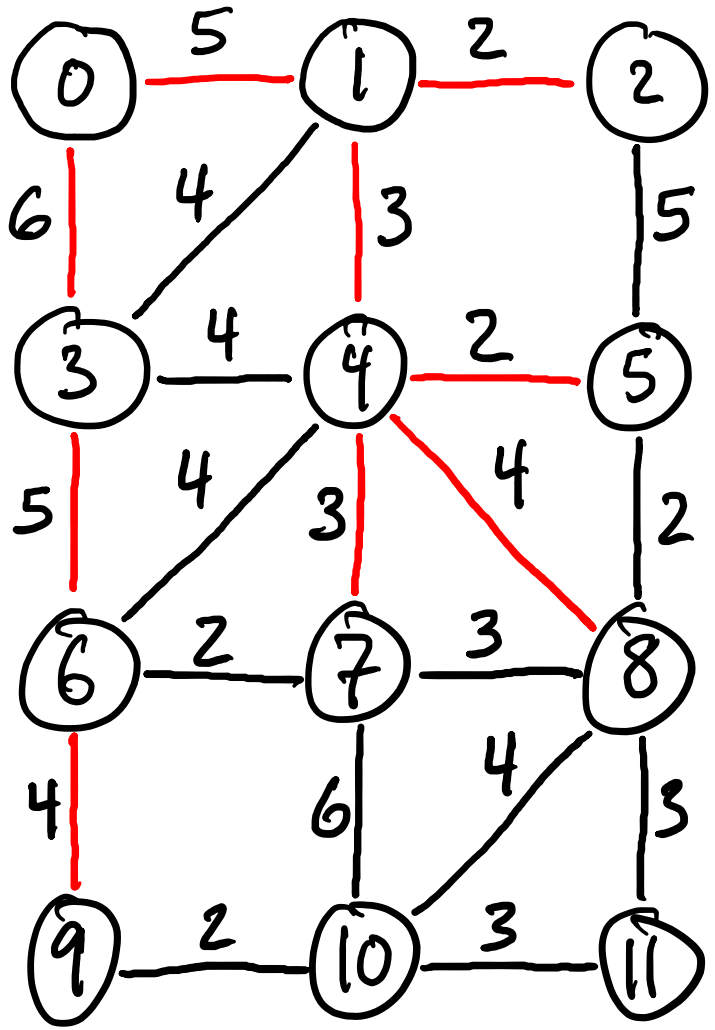
~~7 → 8 : 11 + 3~~

7 → 10 : 11 + 6

Arbitrarily pick (usually First index)

0	1	3	2	4	5	6	7	8			

0	5	7	6	8	10	11	11	12	15	16	15



Neighbors of nodes in queue:

$6 \to 9: 11+4$ ⇐

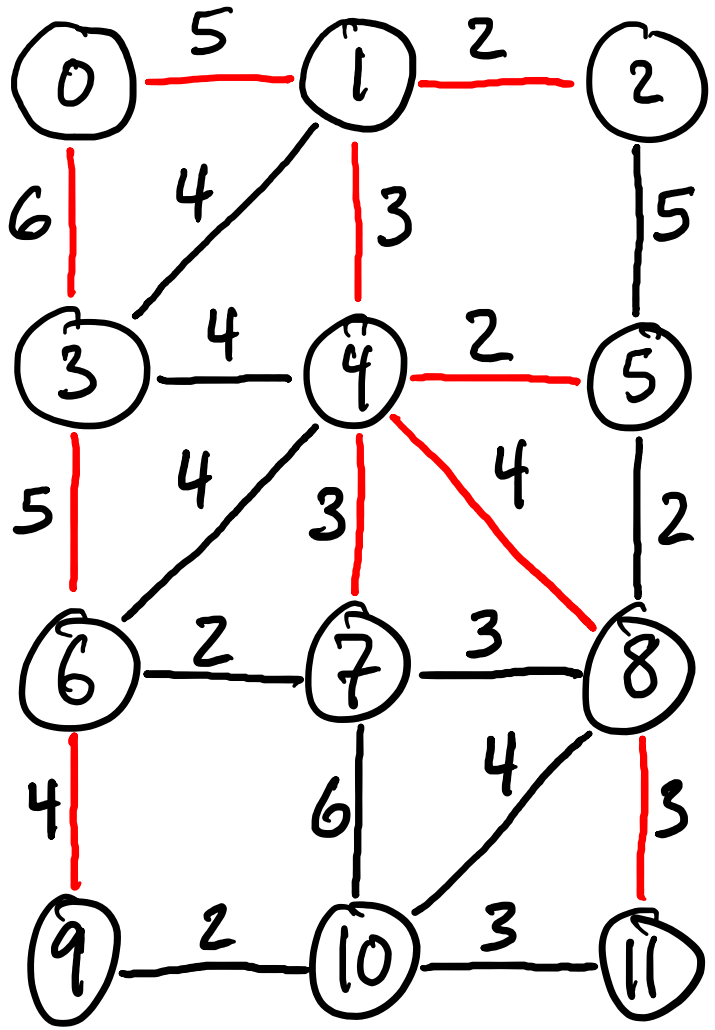
~~$7 \to 10: 11+6$~~

$8 \to 10: 12+4$

$8 \to 11: 12+3$ ⇐

0	1	3	2	4	5	6	7	8	9		

0	5	7	6	8	10	11	11	12	15	16	15



Neighbors of nodes in queue:

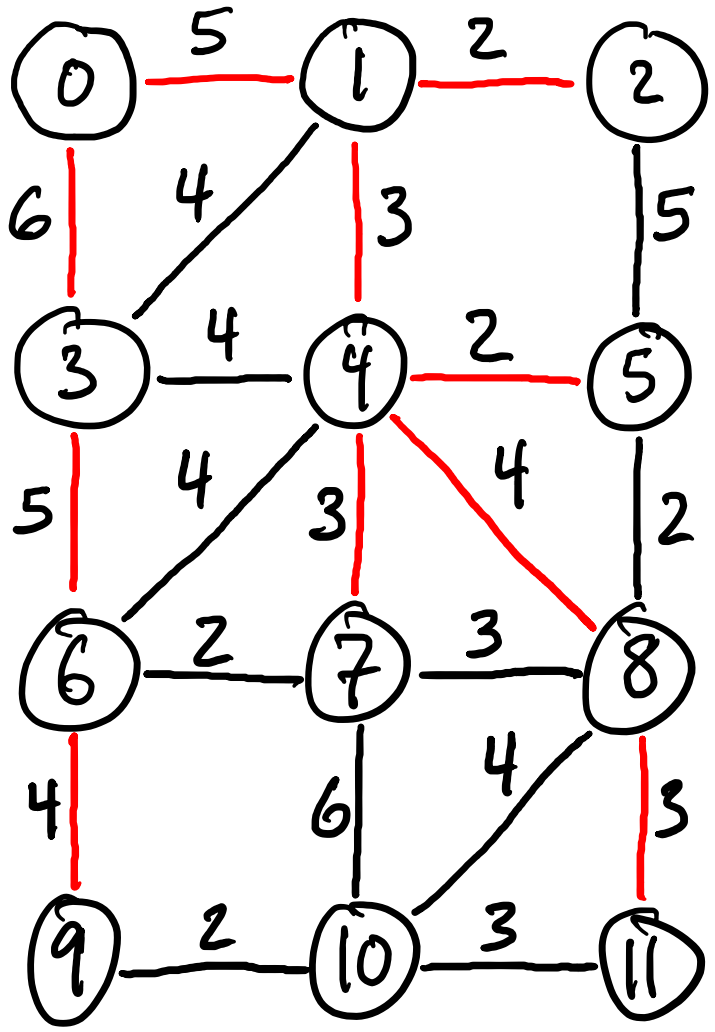
~~7 → 10: 11+6~~

8 → 10: 12+4

8 → 11: 12+3 ⇐

0	1	3	2	4	5	6	7	8	9	11	

0	5	7	6	8	10	11	11	12	15	16	15



Neighbors of nodes in queue:

~~7 → 10: 11+6~~

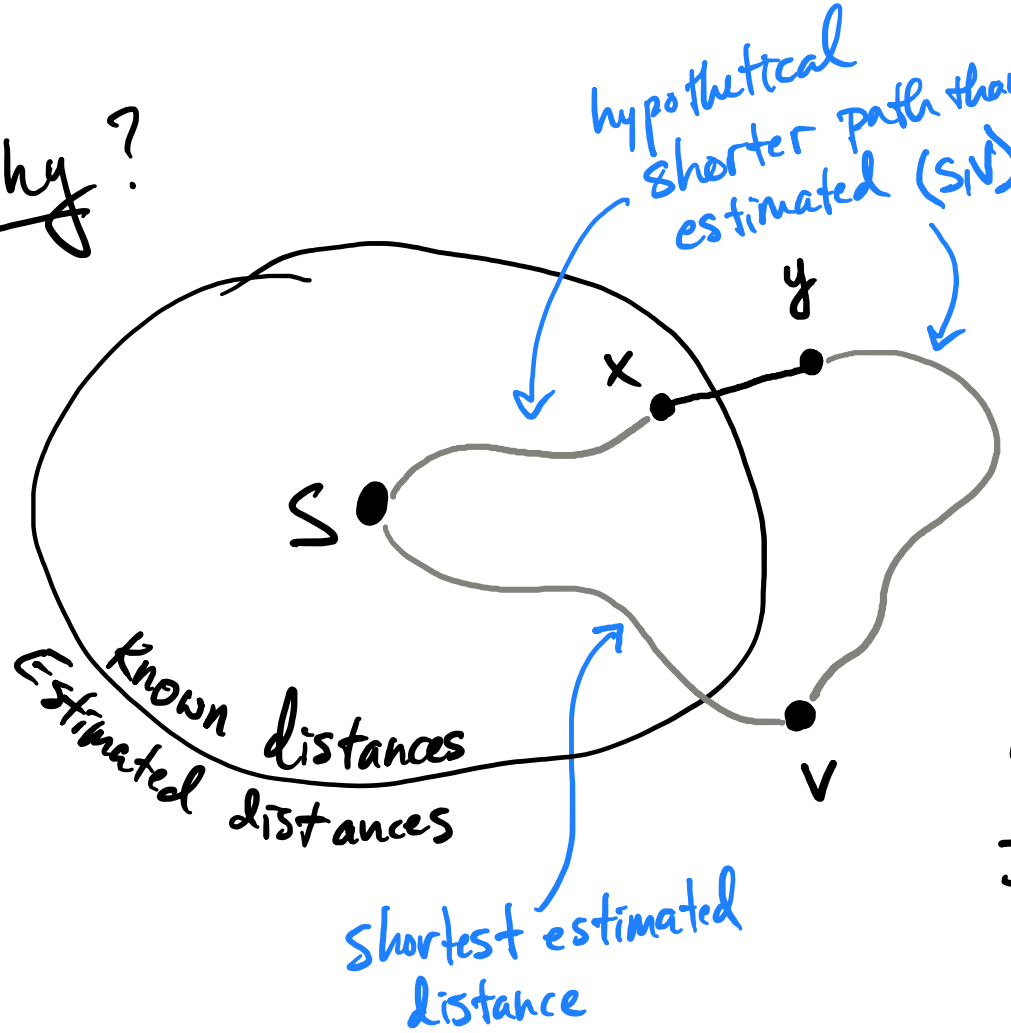
8 → 10: 12 + 4 ⇐

0	1	3	2	4	5	6	7	8	9	11	10

key to Dijkstra:

The node with the shortest distance is "final" (exact, not an upperbound)

why?



Assume node v has the shortest estimated! distance of all the neighbors of the nodes with known distance

$$v = \underset{u \in N}{\operatorname{argmin}} \; w(s,u)$$

weight of shortest estimated path

$N = \{$all nbrs of nodes with known dist$\}$

If there is another path, it must go through another node $y \in N$

If $s \to \dots \to x \to y \to \dots \to v$ is shorter than $s \to v$,

$s \to \dots \to x \to y$ should be shorter than $s \to v$

Computational Complexity of Dijkstra

We do this n times:

- loop through to find smallest estimated distance
- update estimates of neighbors' distances

$$n \times \left(n + \frac{m}{n} \right)$$

n → $\log n$

$\frac{m}{n}$ → $\frac{m}{n} \log n$ (use of optimized data structures (heap))

$$n \times \log n + \frac{m}{n} \log n$$

$$O((n+m)\log n) + O(n)$$

to set up the array, but $O(n) < O(n \log n)$, so omit