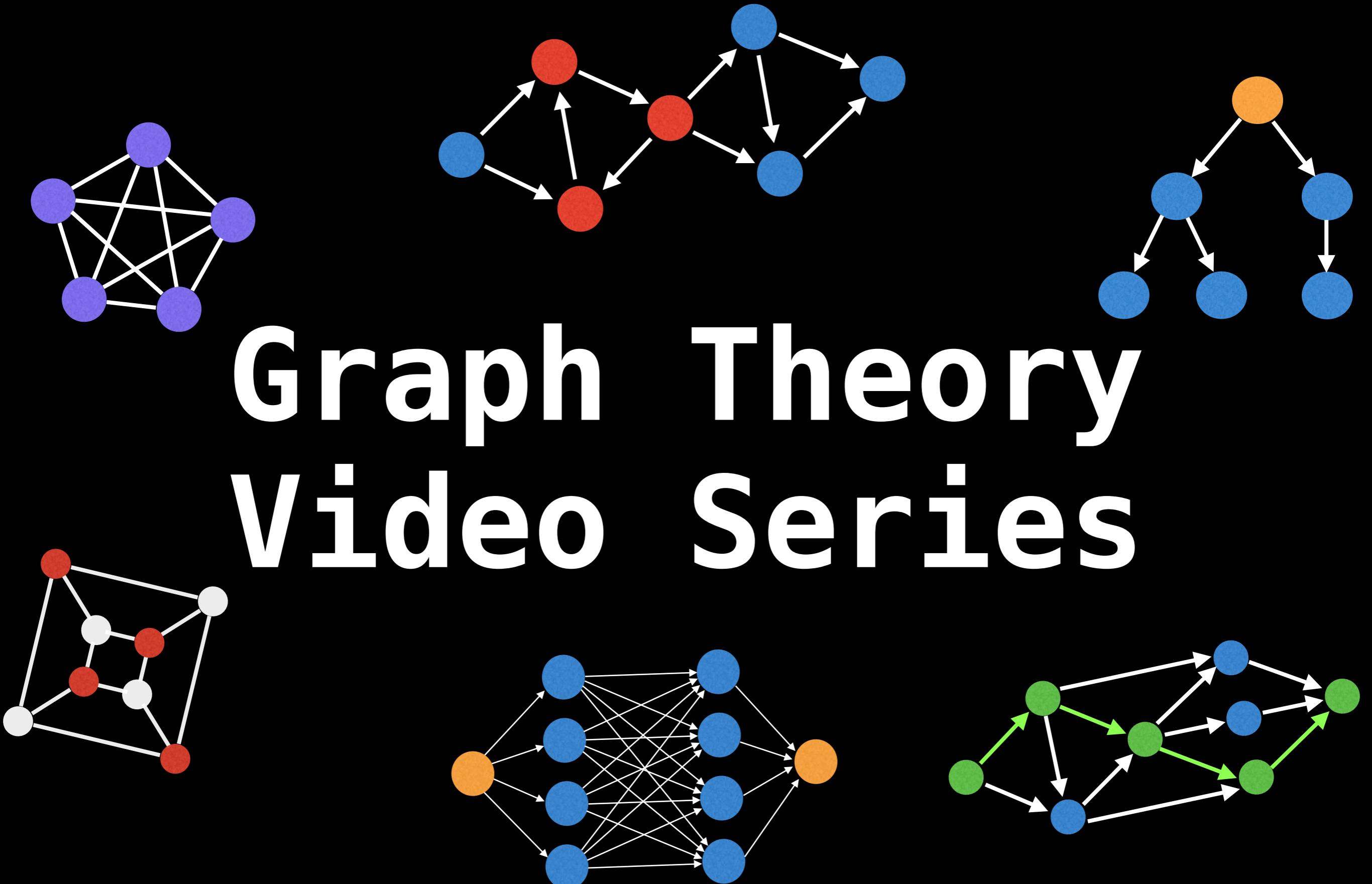


# Graph Theory Video Series

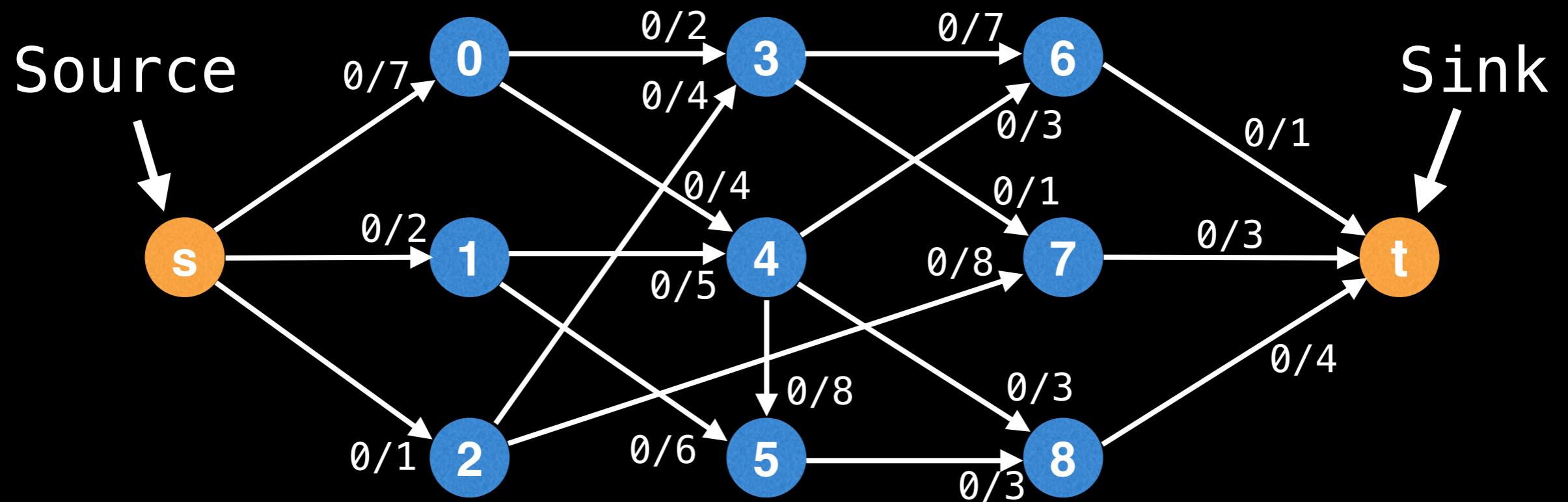


# Max Flow Ford-Fulkerson method

William Fiset

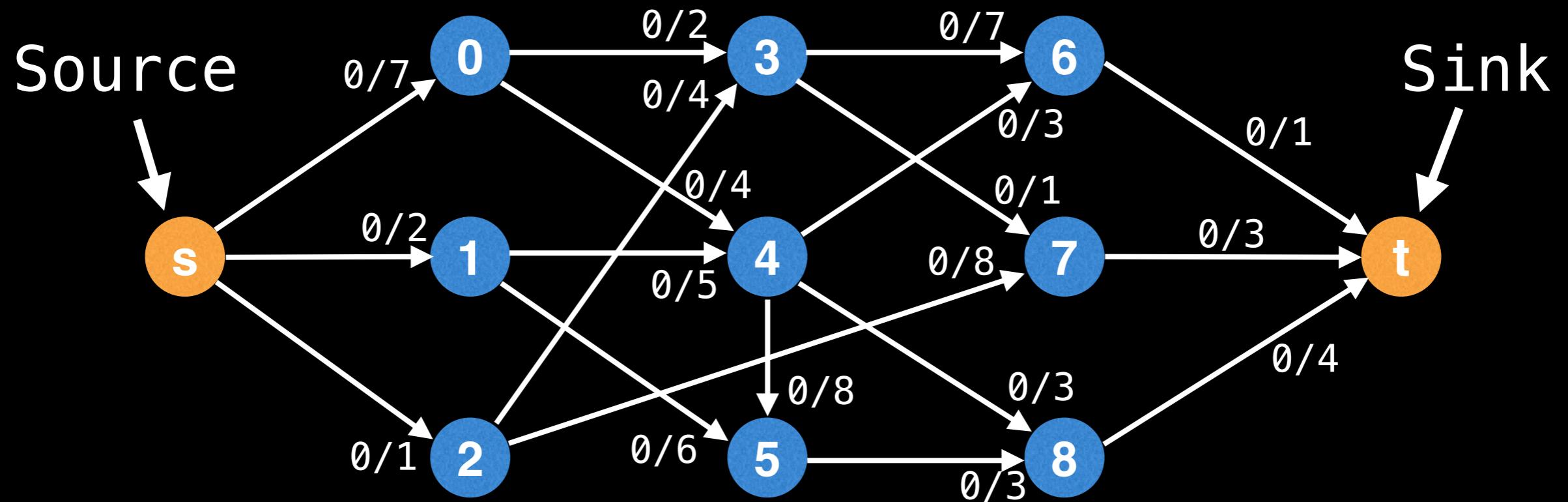
# Max flow

Q: With an infinite input source, how much “flow” can we push through the network given that each edge has a certain capacity?



# Max flow

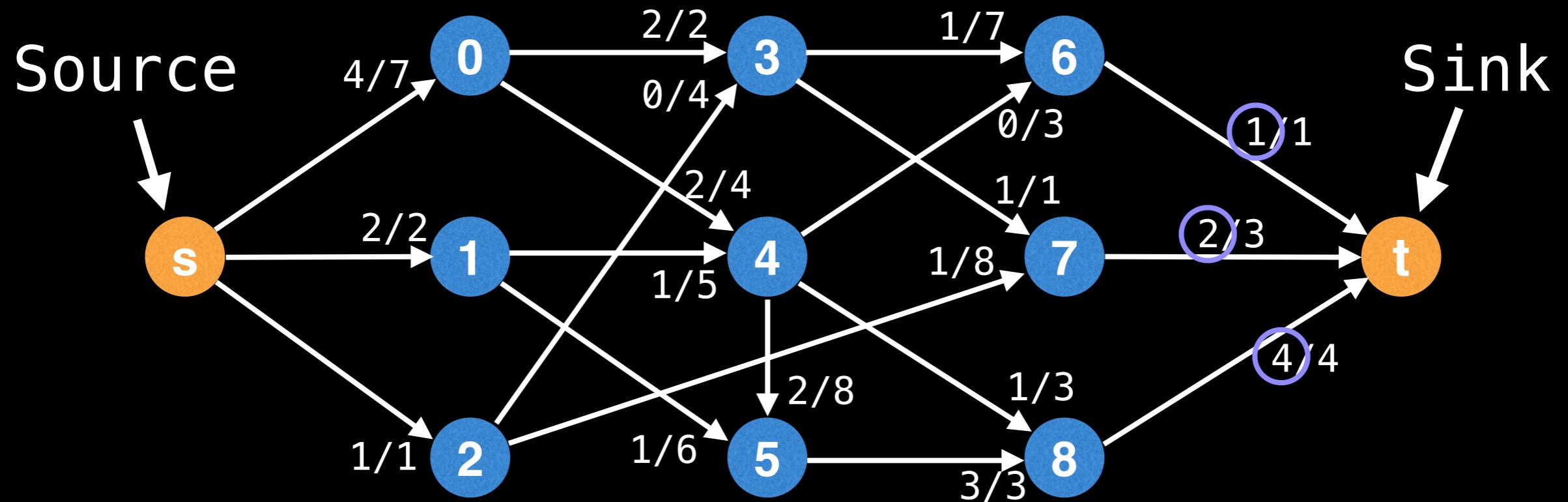
Q: With an infinite input source, how much “flow” can we push through the network given that each edge has a certain capacity?



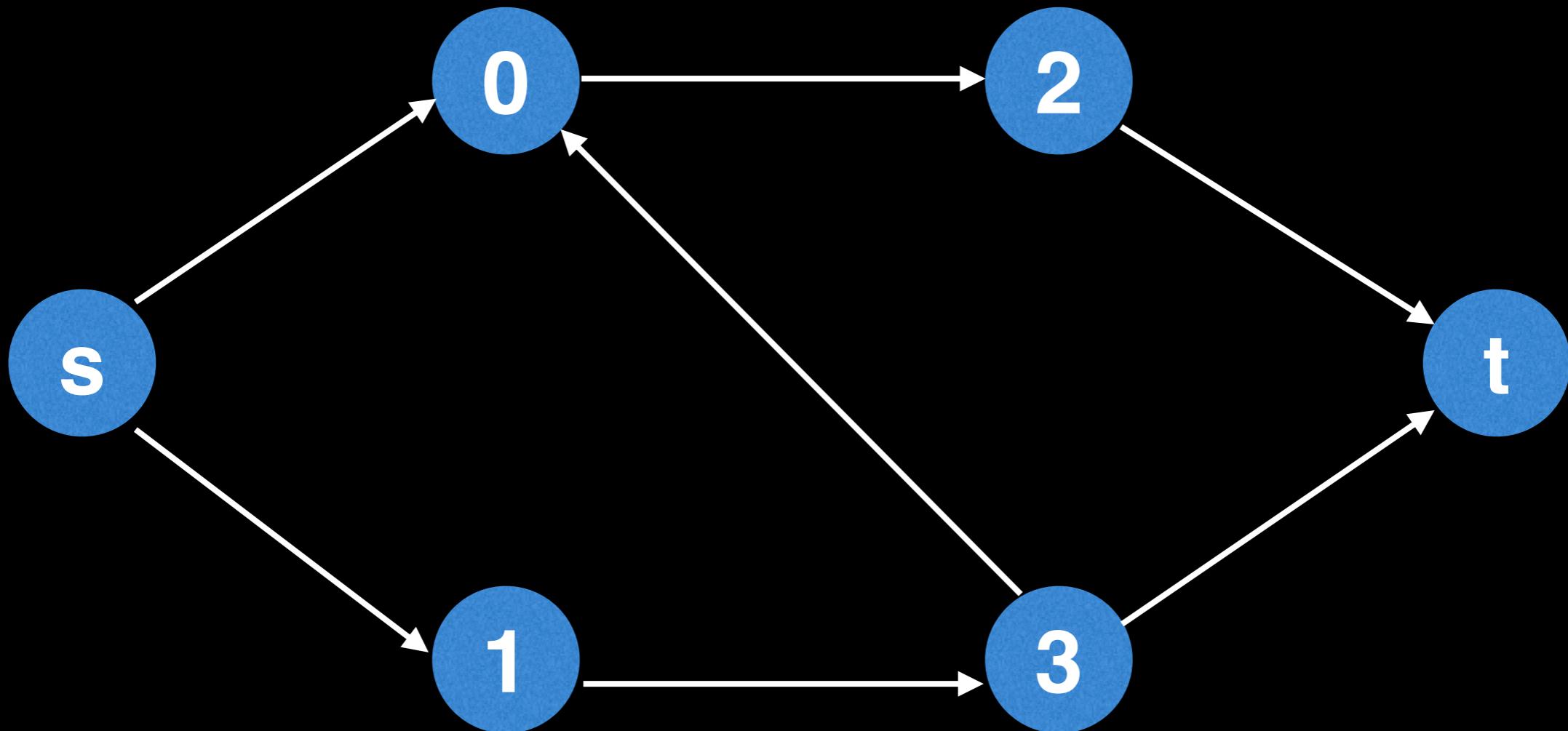
Suppose the edges are roads with cars, pipes with water or wires with electric current. Flow represents the volume of water allowed to flow through the pipes, the number of cars the roads can sustain in traffic and net electric current. Effectively, it's the “bottleneck” value for the amount of flow that can pass through the network from source to sink under all the constraints.

# Max flow

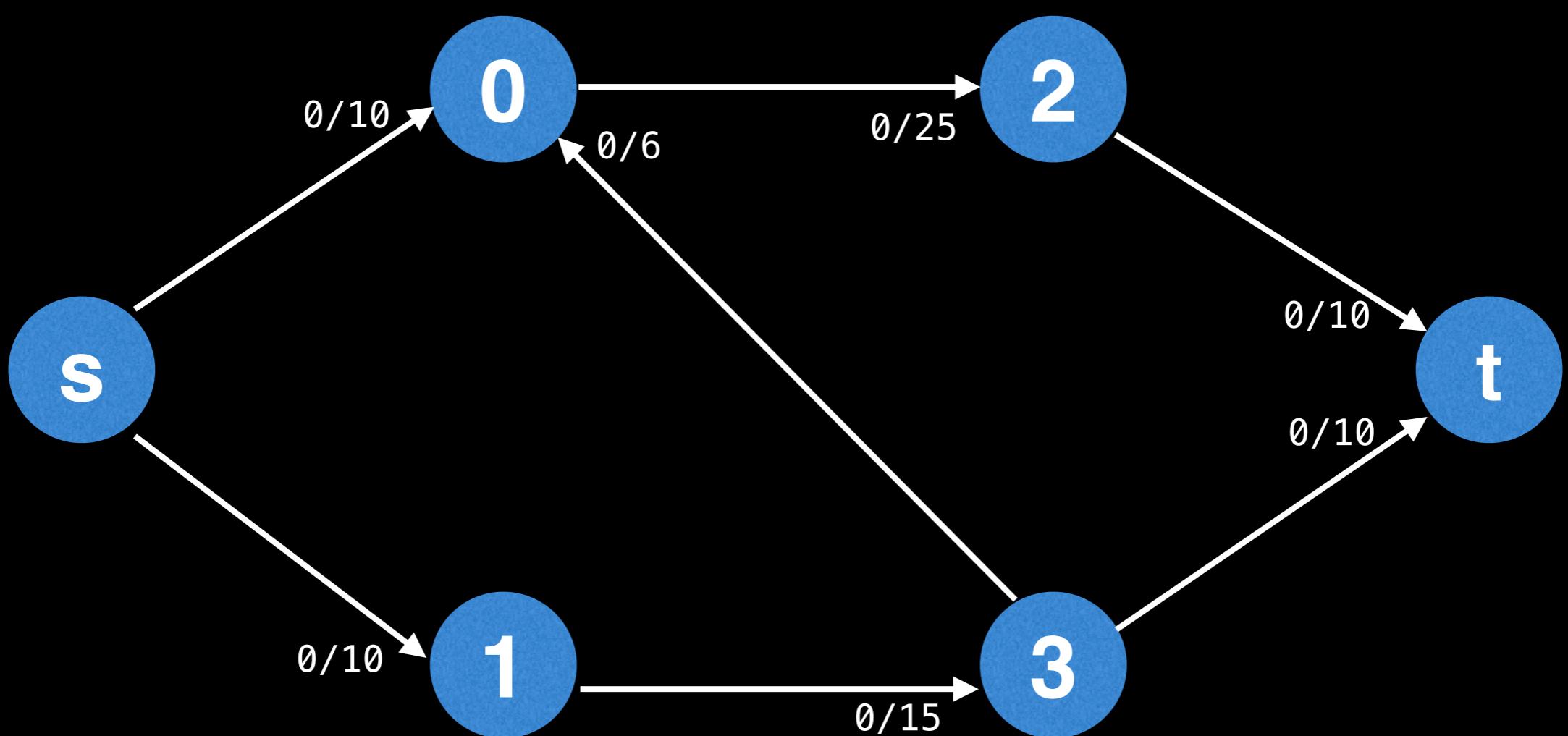
Q: With an infinite input source, how much “flow” can we push through the network given that each edge has a certain capacity?



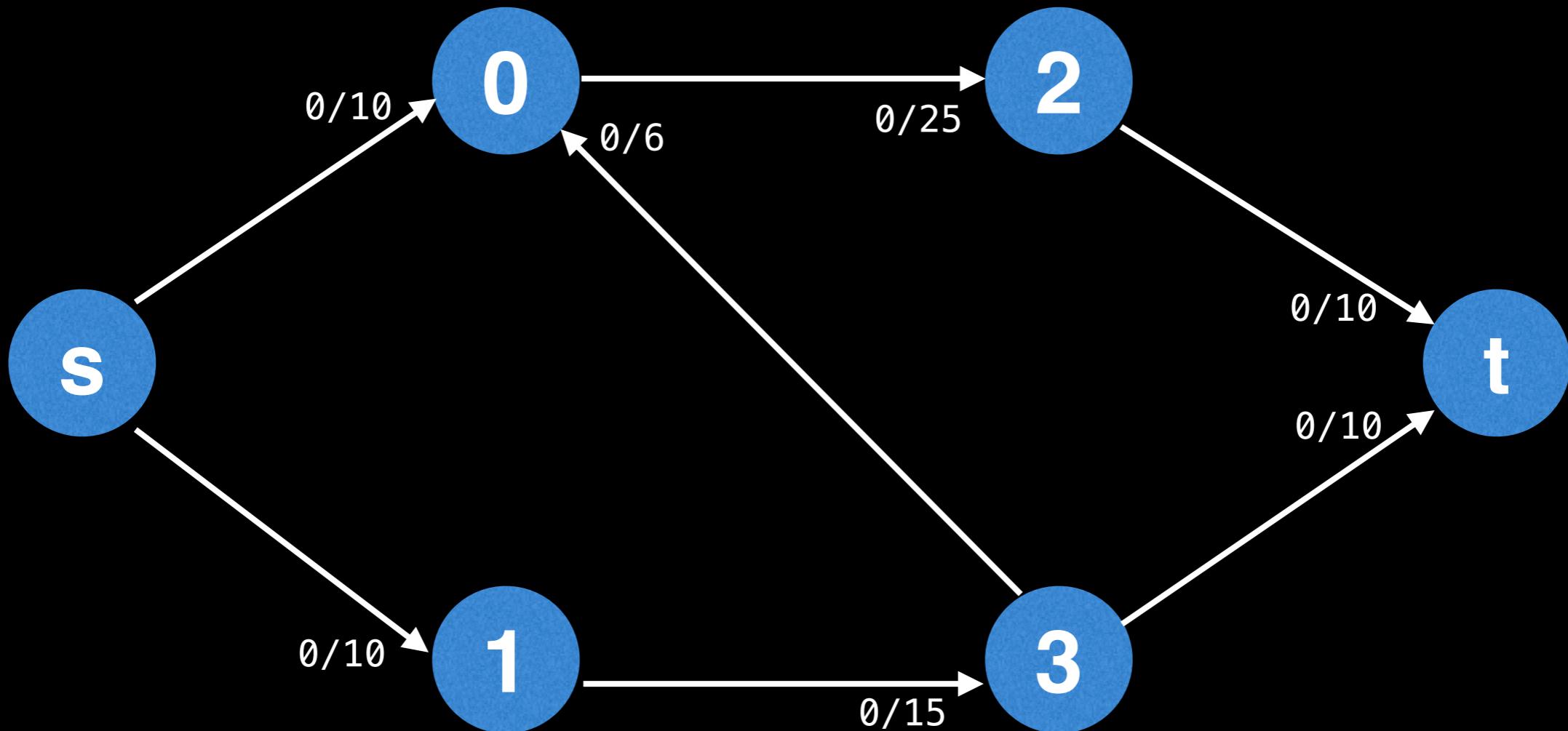
In this flow graph, the max flow is 7 since  $1+2+4 = 7$  is the sum of the flows entering the sink node which is the most we can push through the network.



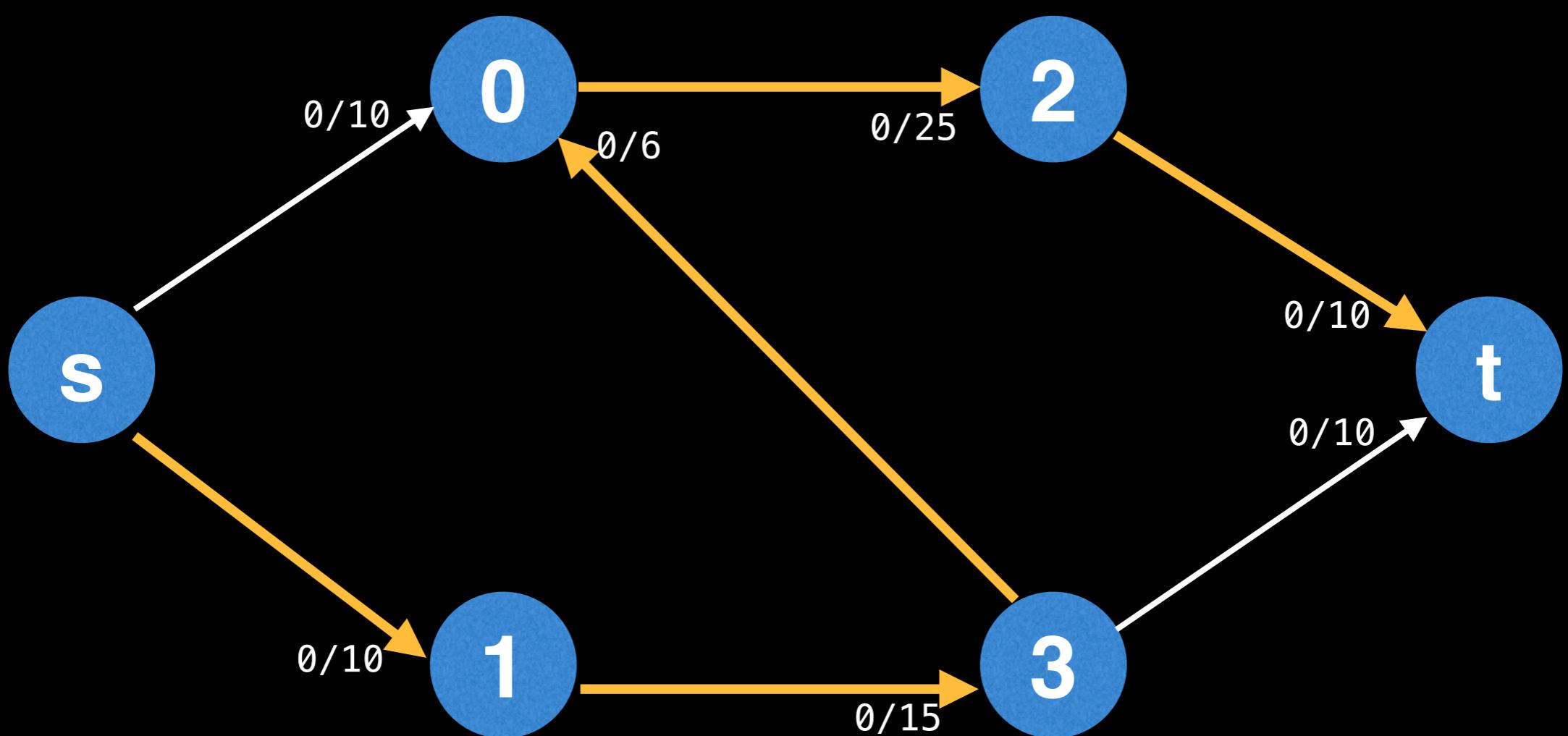
A **flow graph** (flow network) is a directed graph where each edge (also called an arc) has a certain **capacity** which can receive a certain amount of **flow**. The flow running through an edge must be less than or equal to the capacity.



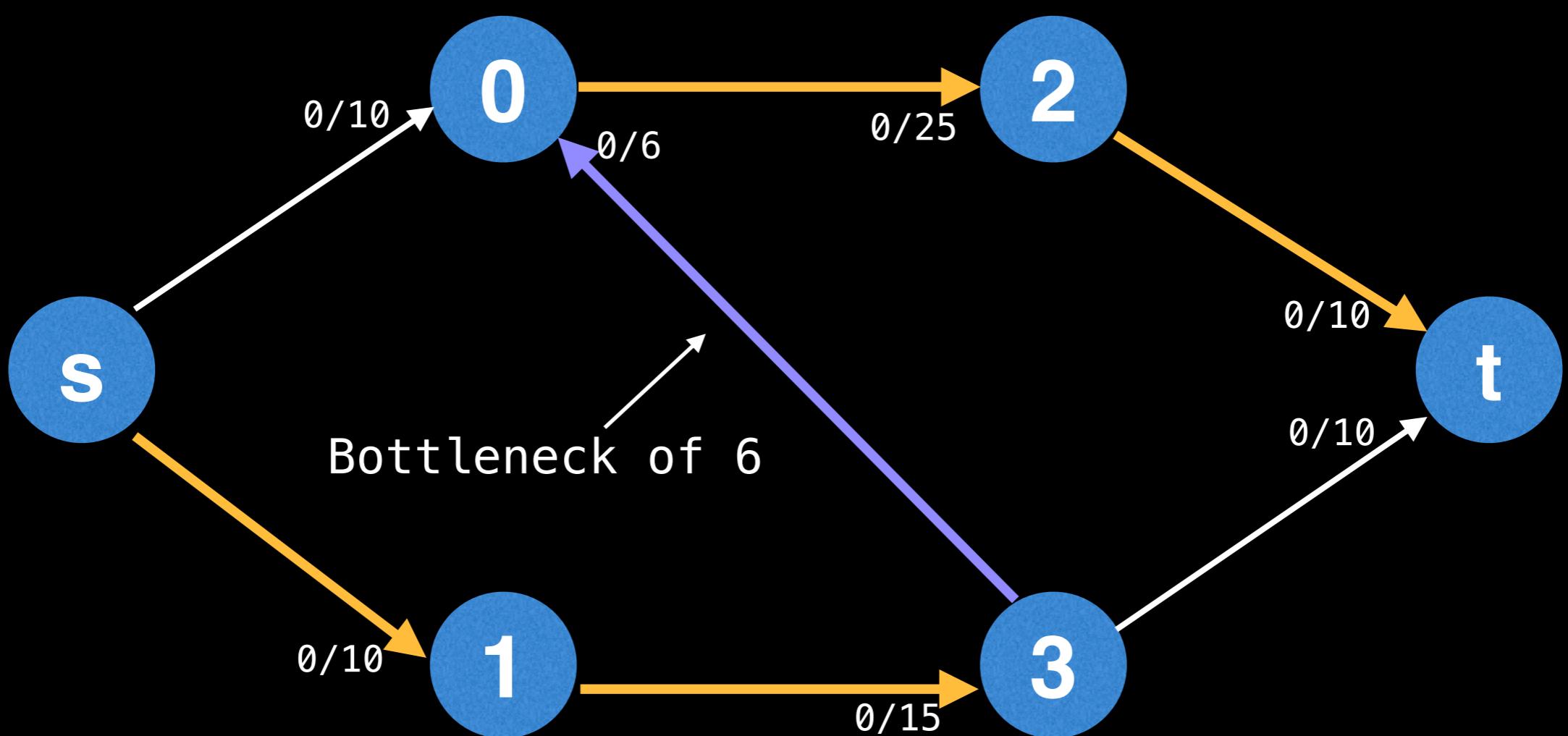
Each edge in the flow graph has a certain flow and capacity specified by the fraction adjacent to each edge. Initially, the flow through each edge is 0 and the capacity is a non negative value.



To find the maximum flow (and min-cut as a by product), the Ford–Fulkerson method repeatedly finds **augmenting paths** through the **residual graph** and **augments the flow** until no more augmenting paths can be found.



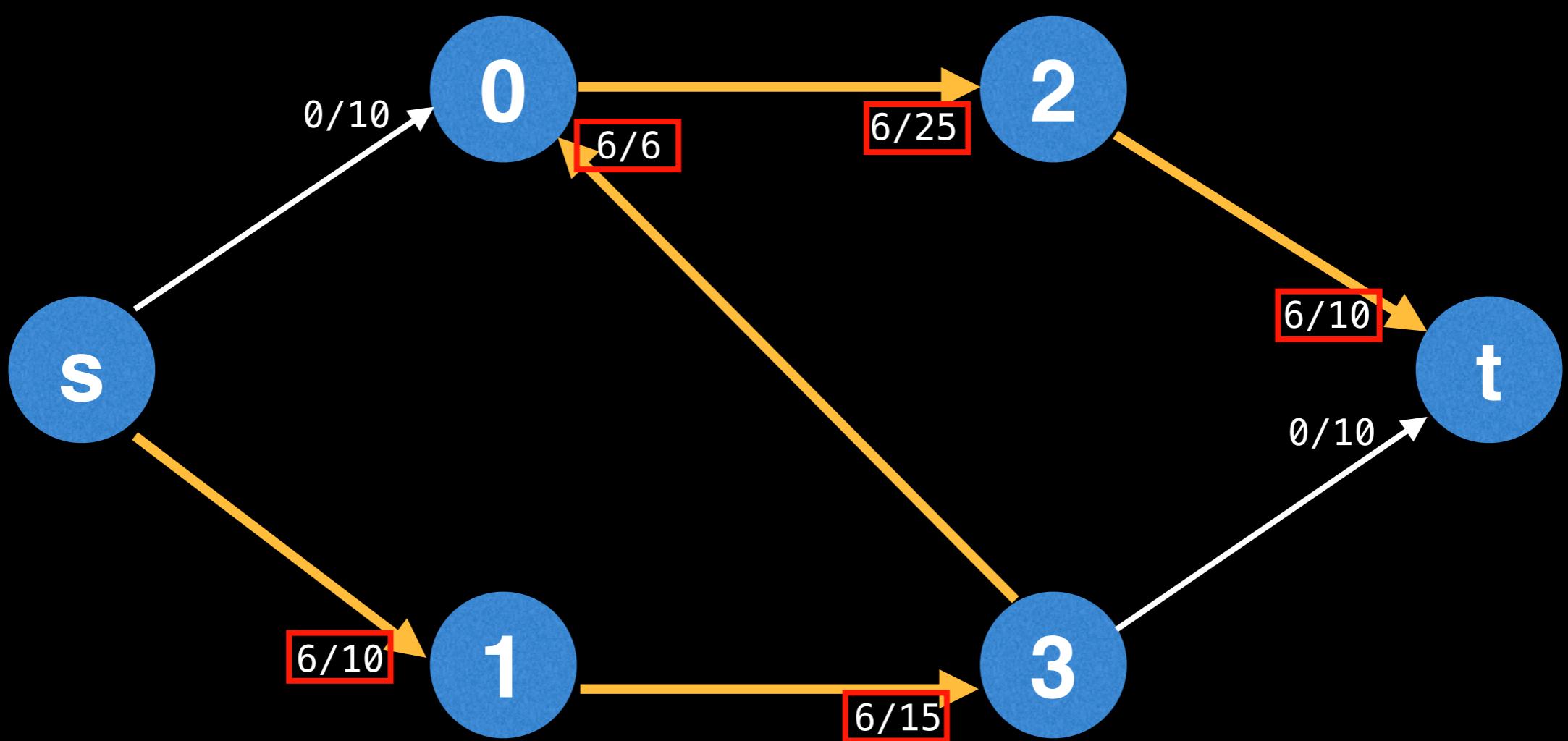
An **augmenting path** is a path of edges in the residual graph with unused capacity greater than zero from the source  $s$  to the sink  $t$ .



In the augmenting path above, the **bottleneck** is the “smallest” edge on the path. We can use the bottleneck value to **augment the flow** along the path.

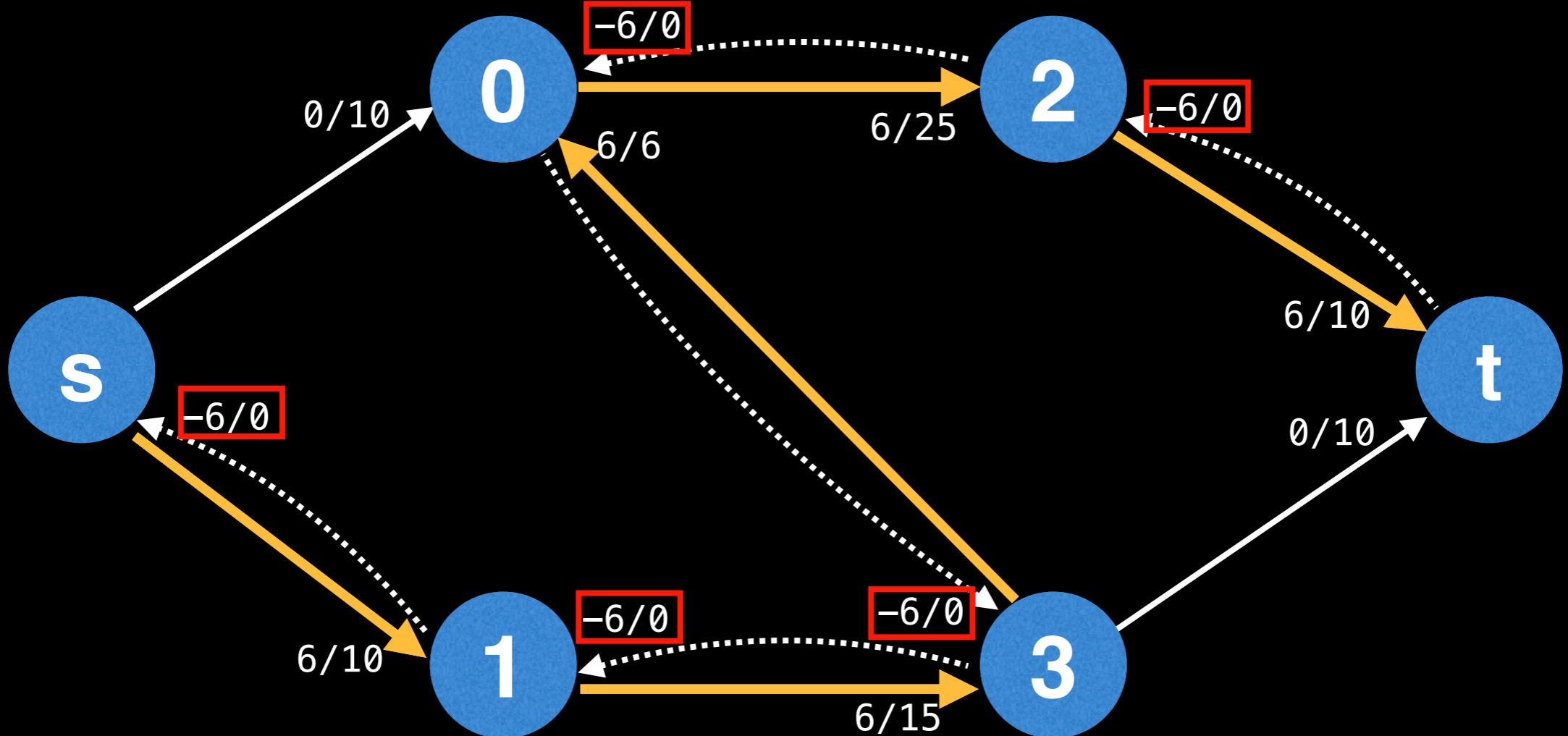
The smallest remaining capacity of any edge along the augmenting path is:

$$\min(10-0, 15-0, 6-0, 25-0, 10-0) = \min(10, 15, 6, 25, 10) = 6$$

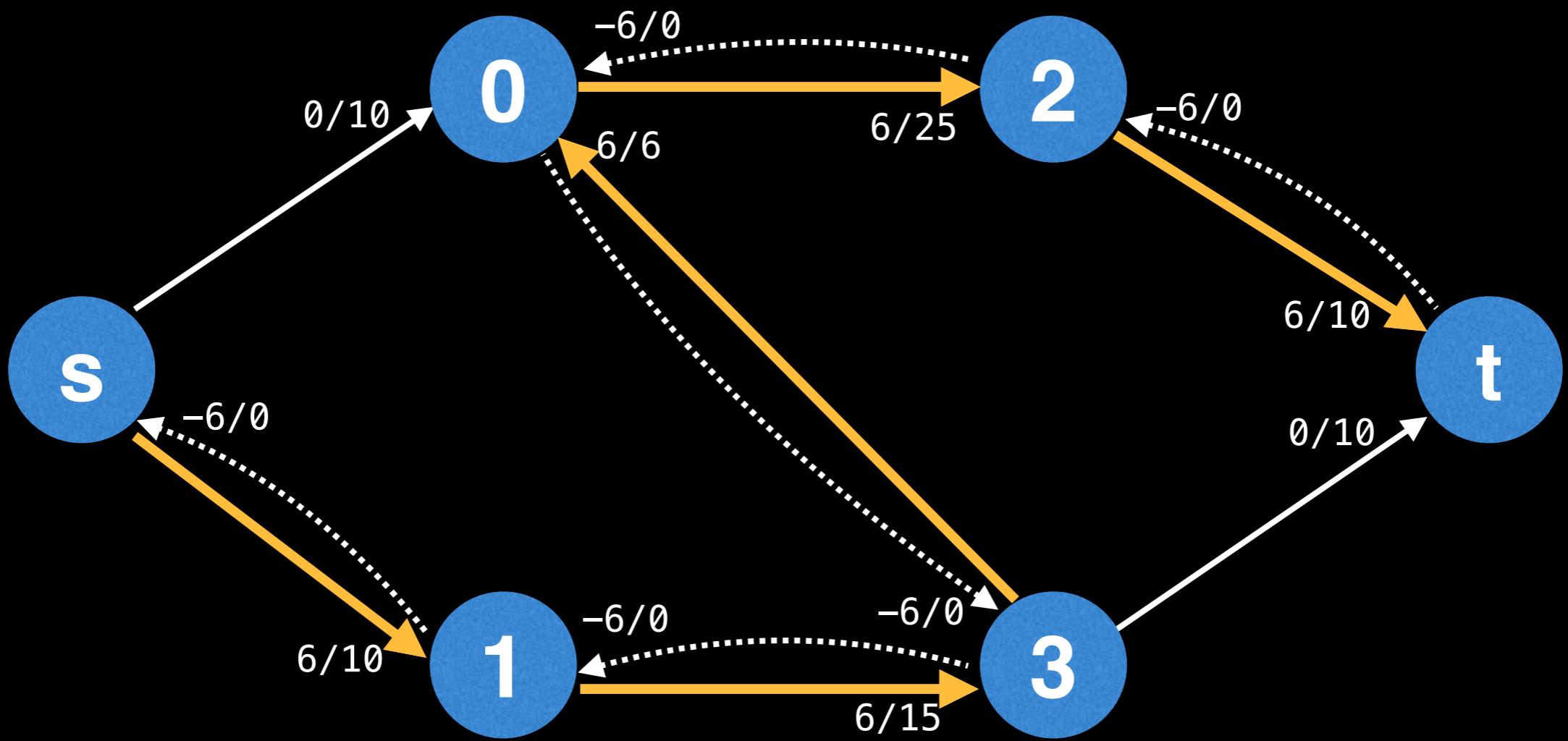


**Augmenting the flow** means *updating* the flow values of the edges along the augmenting path.

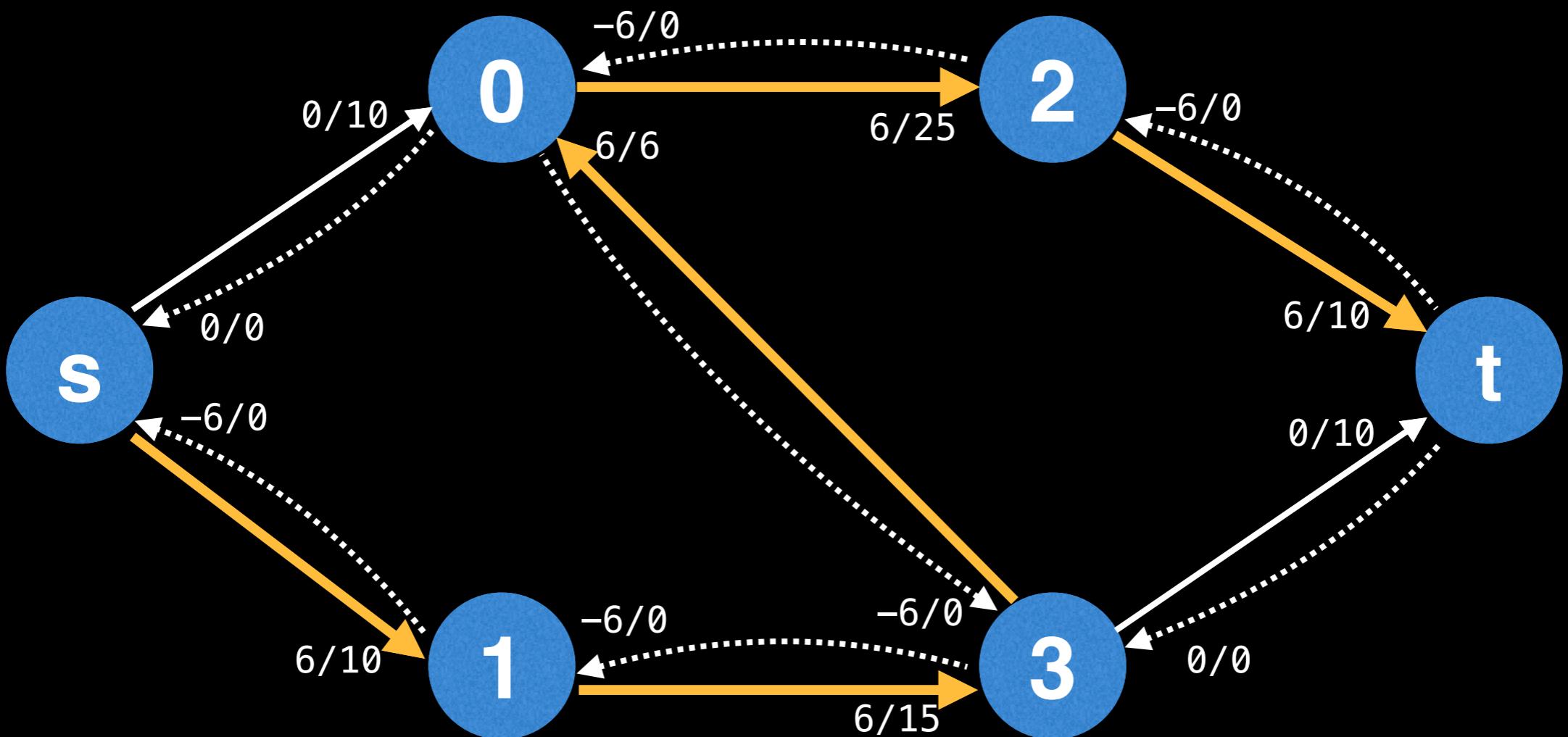
For forward edges, this means **increasing the flow** by the bottleneck value.



When augmenting the flow along the augmenting path, you also need to **decrease the flow** along each **residual edge** by the bottleneck value.

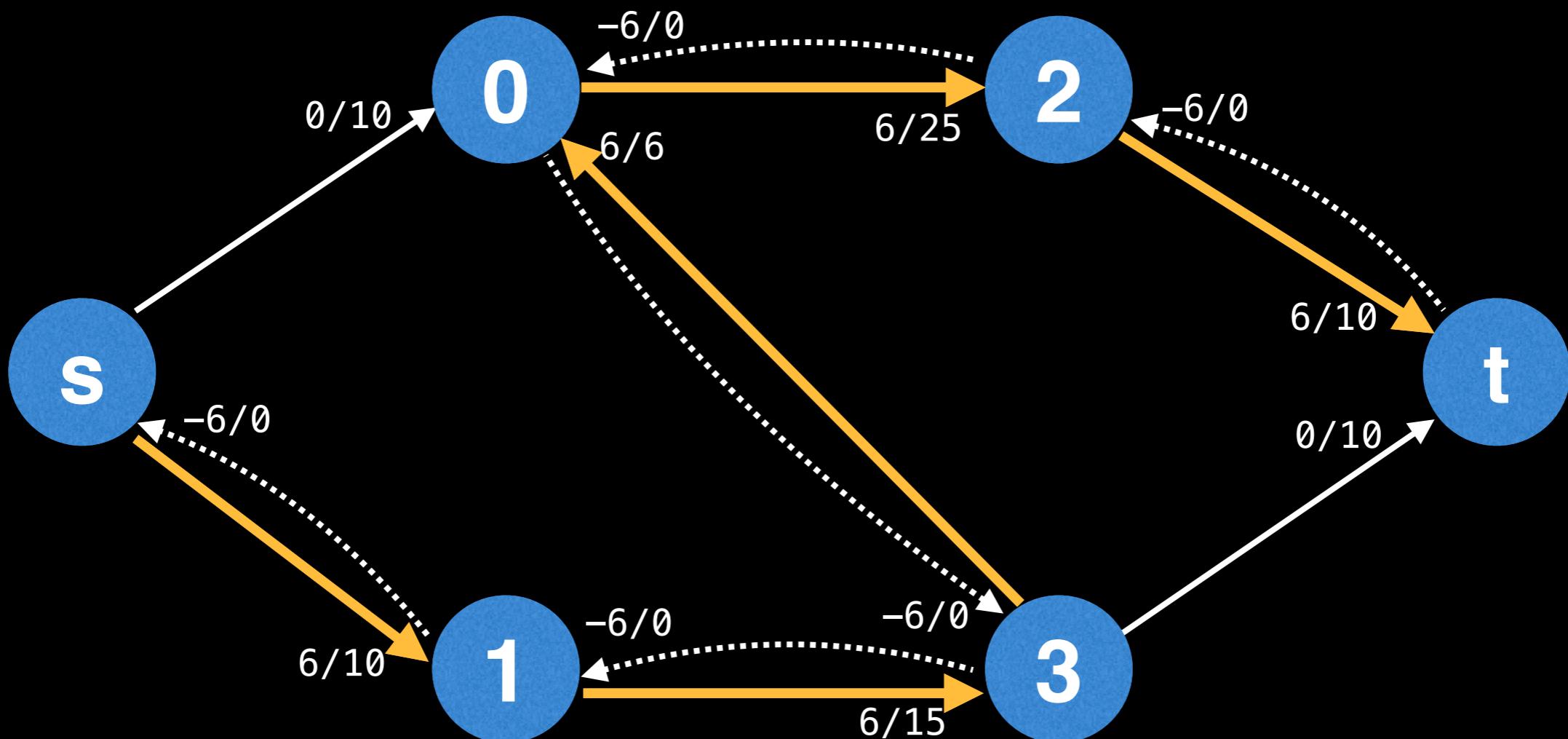


Residual edges exist to “undo” bad augmenting paths which do not lead to a maximum flow.

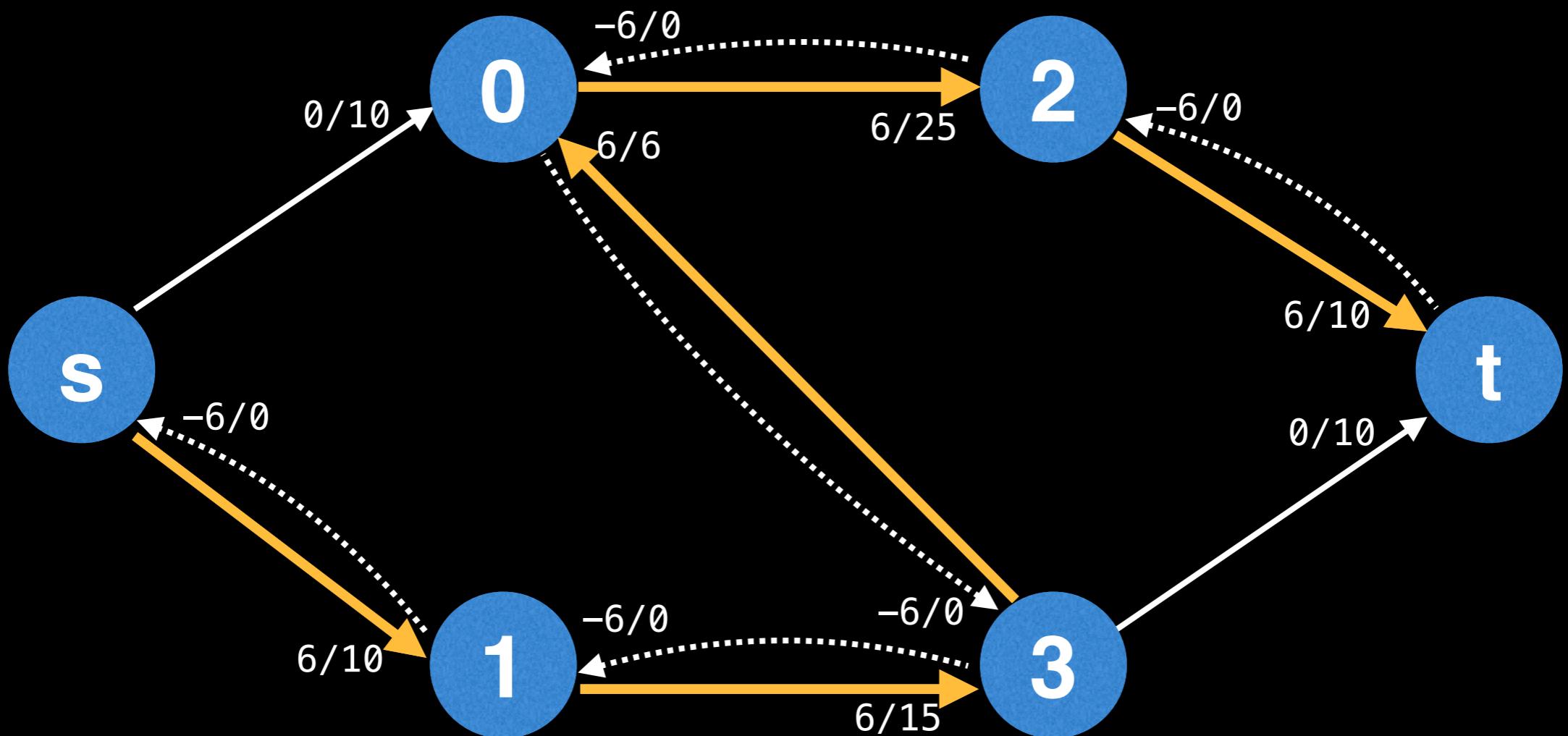


You can think of every edge in the original graph as having a residual edge with a flow/capacity of  $0/0$  which is not usually shown.

The **residual graph** is the graph which also contains residual edges. In general, when I mention the flow graph, I mean the residual graph.

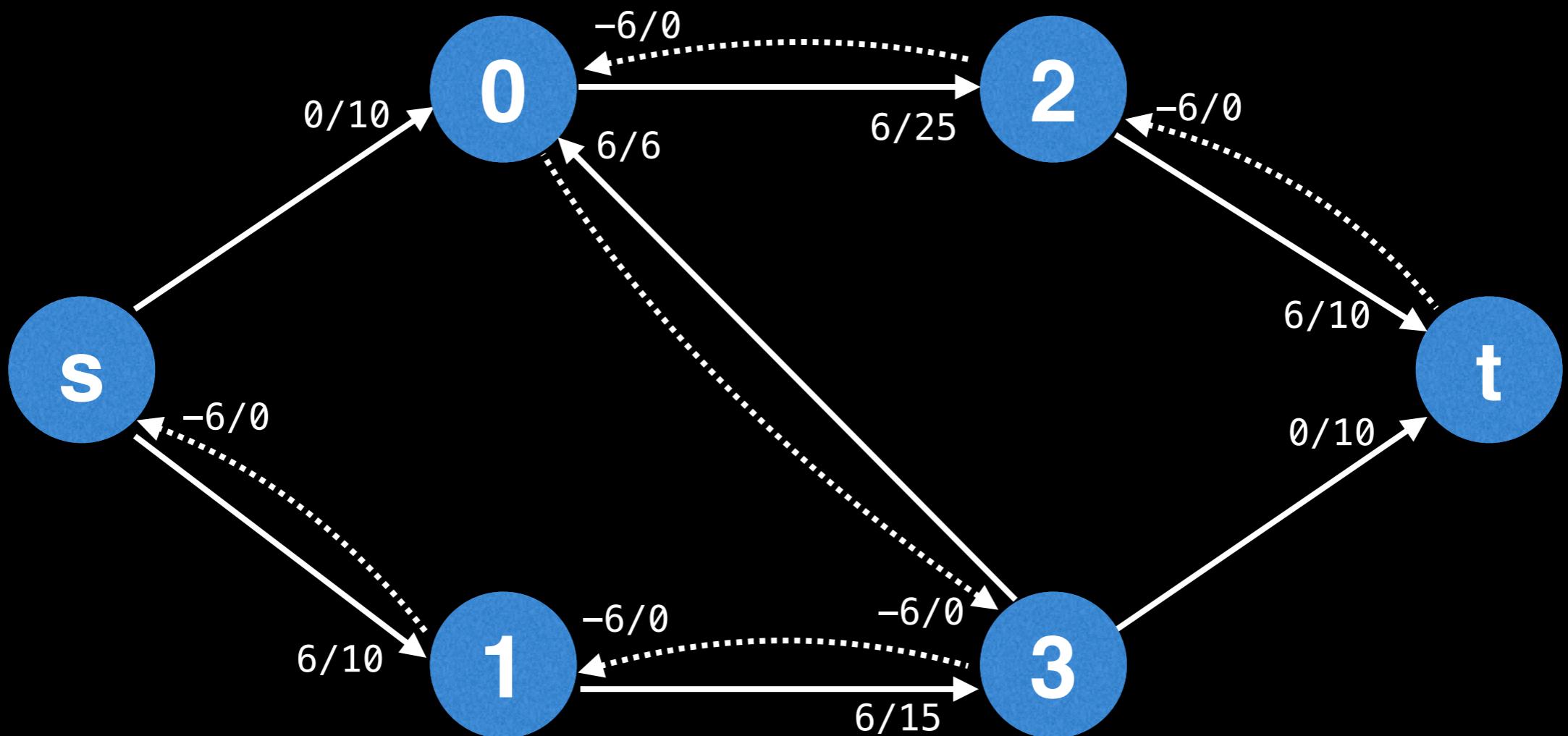


Q: Residual edges have a capacity of 0? Isn't that forbidden? How does that work?



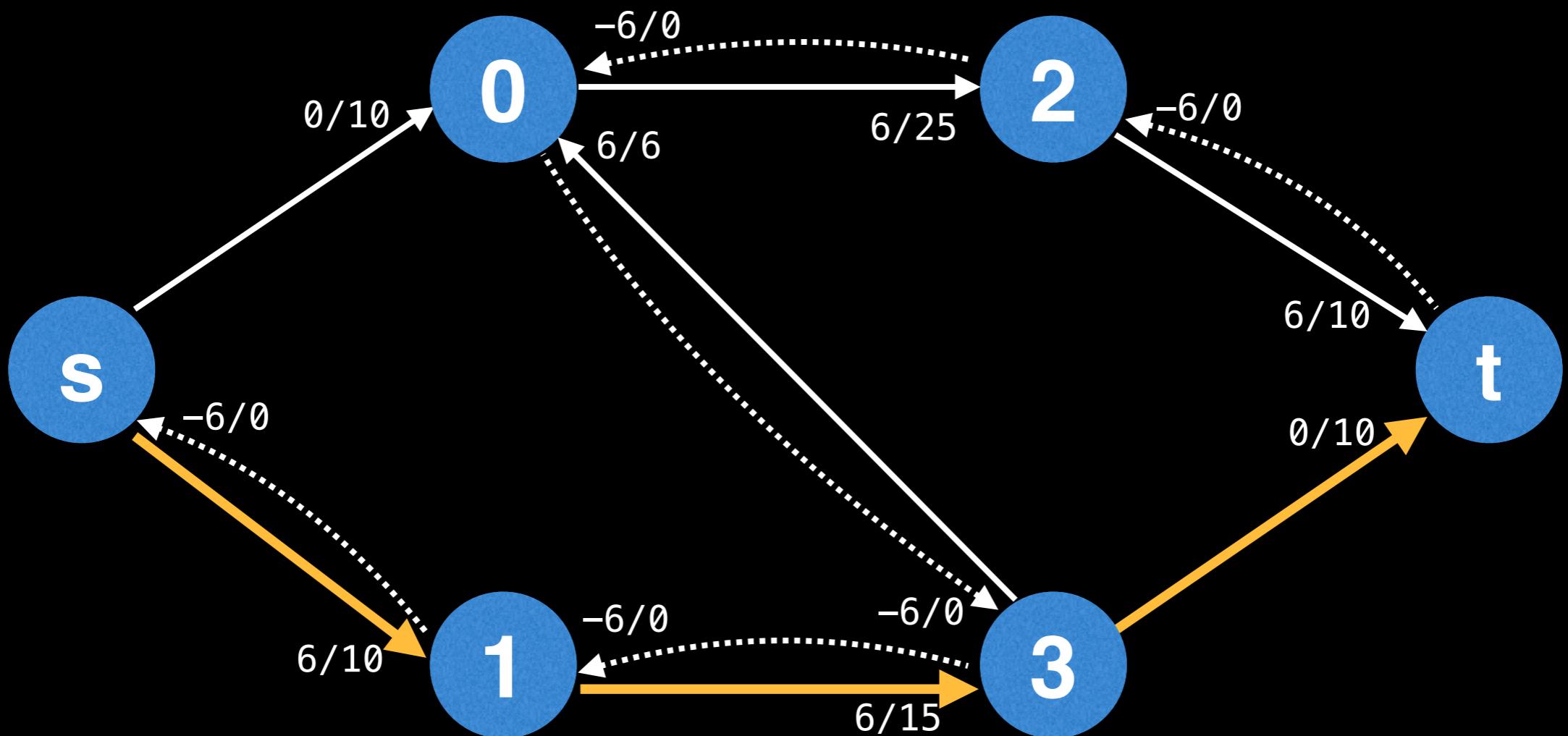
Q: Residual edges have a capacity of 0? Isn't that forbidden? How does that work?

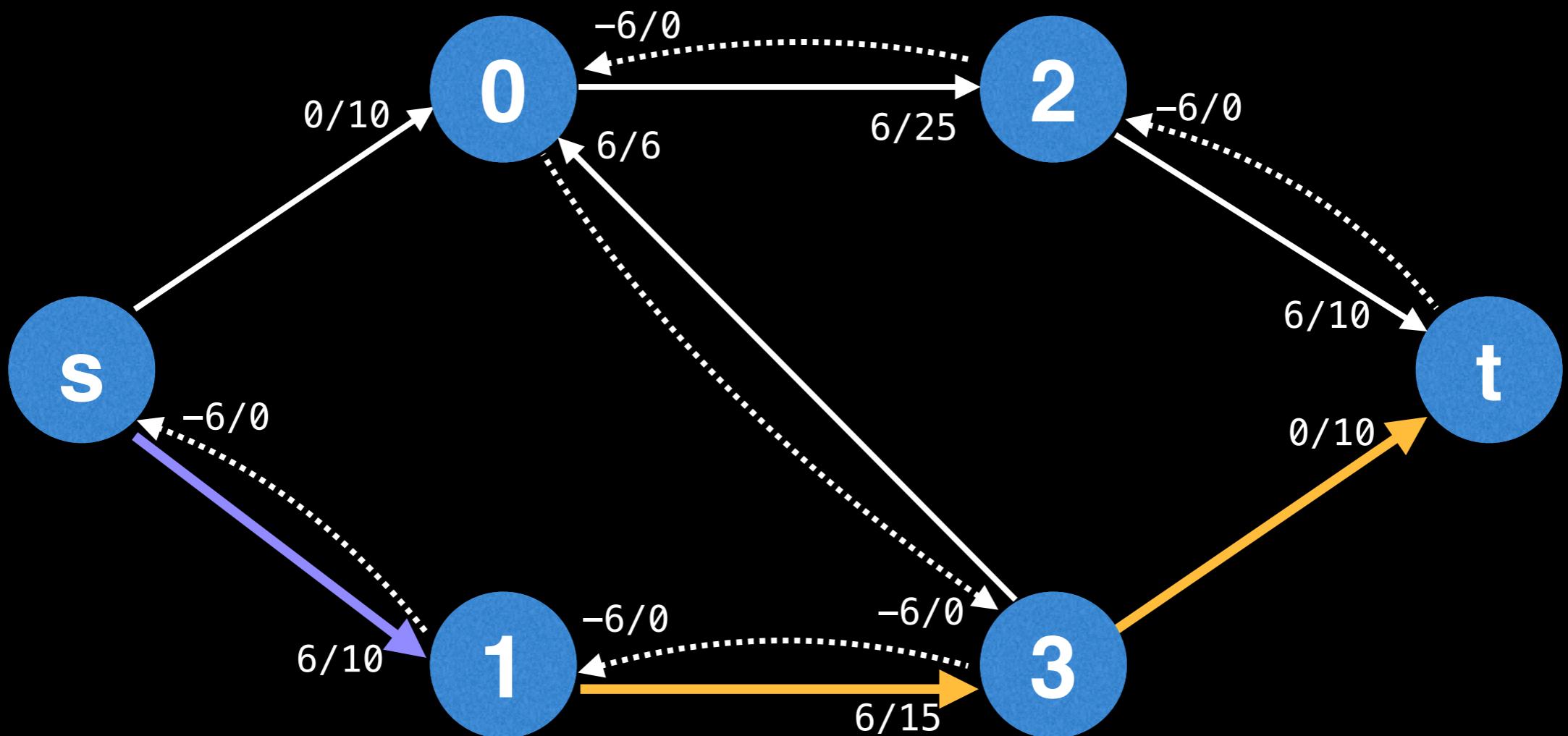
A: Think of the **remaining capacity** of an edge  $e$  (residual or not) as:  $e.\text{capacity} - e.\text{flow}$ , This ensures that the remaining capacity of an edge is always non-negative (even if the flow can be negative).



The Ford–Fulkerson method continues finding augmenting paths and augments the flow until no more augmenting paths from  $s \rightarrow t$  exist.

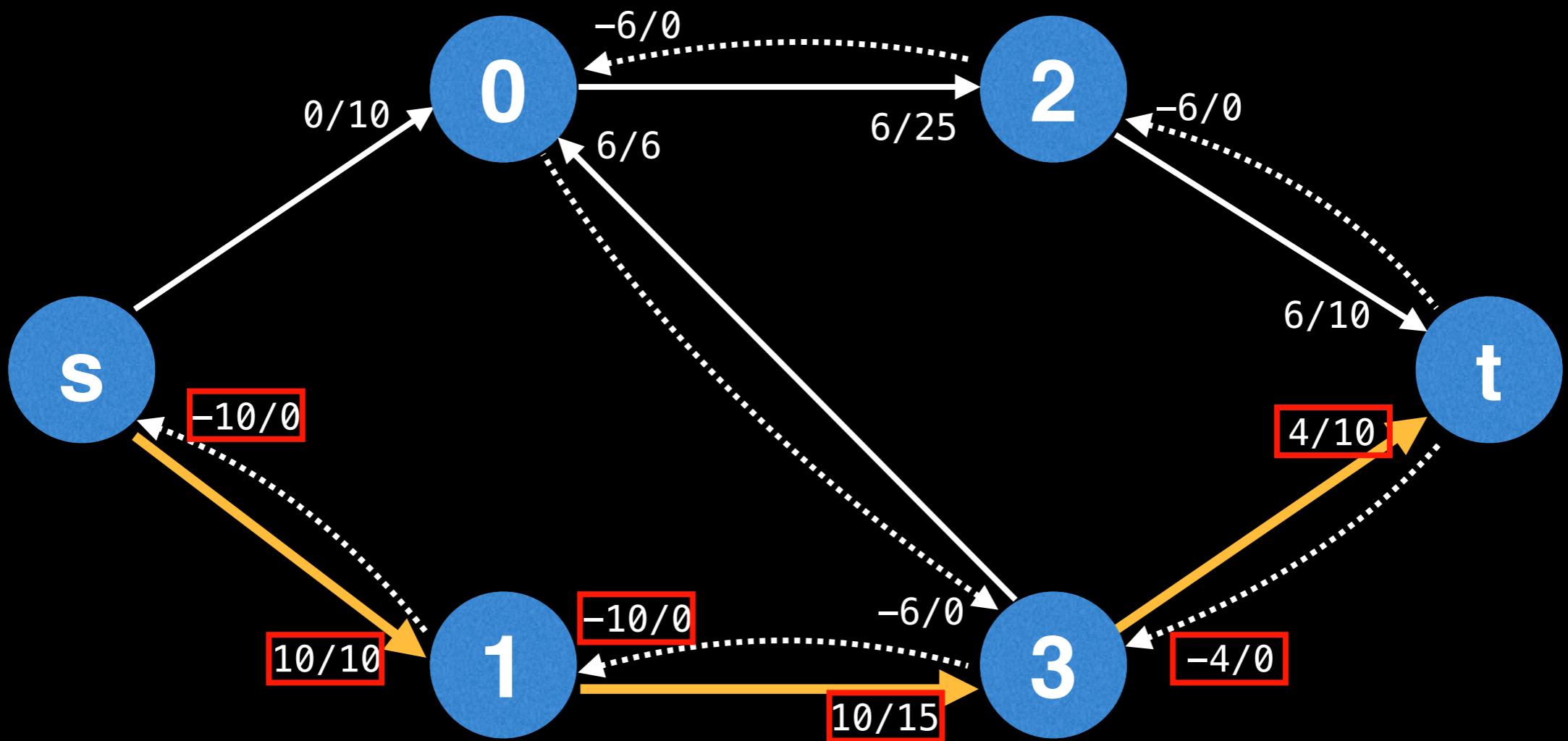
A key realization to make at this point is that the sum of the bottlenecks found in each augmenting path is equal to the max-flow!



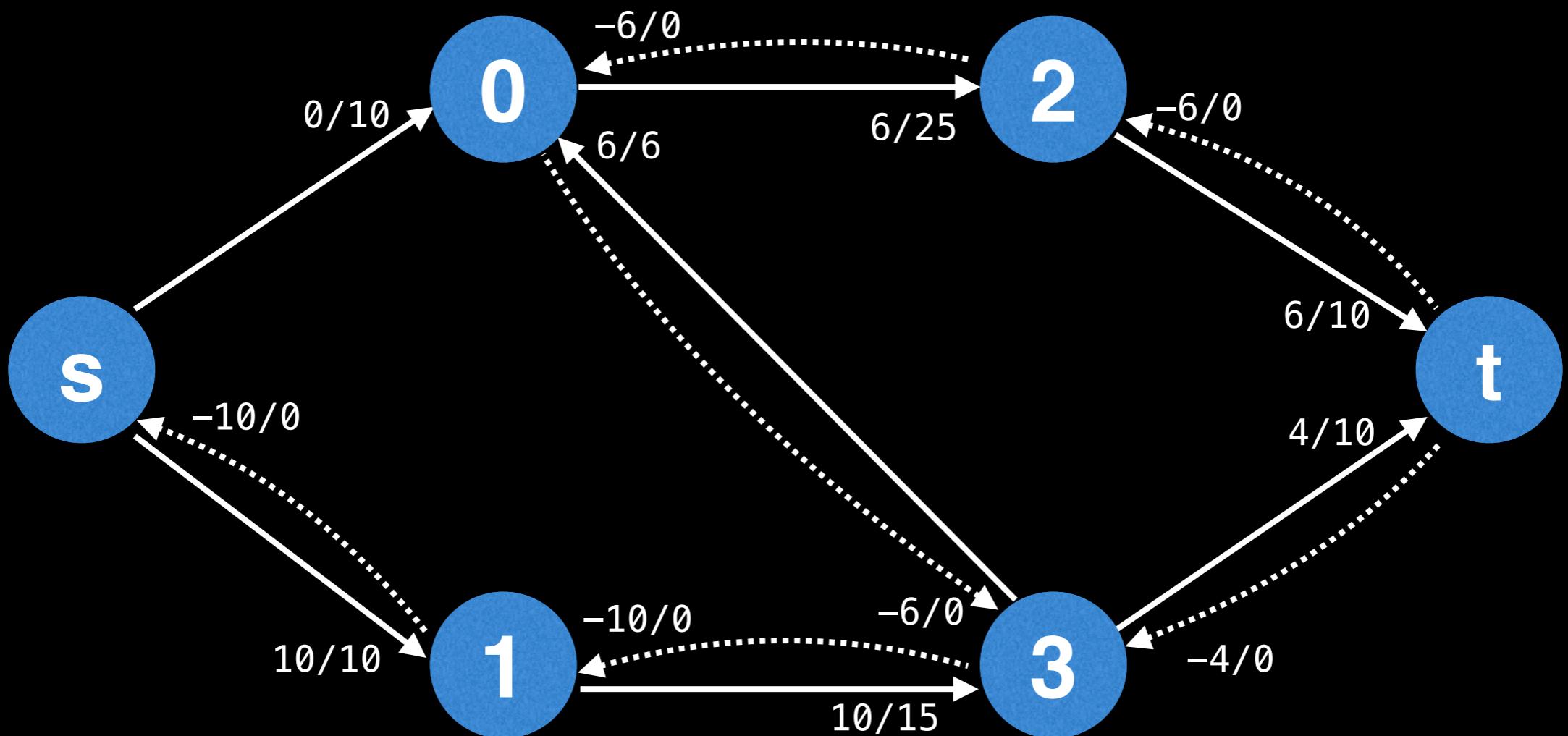


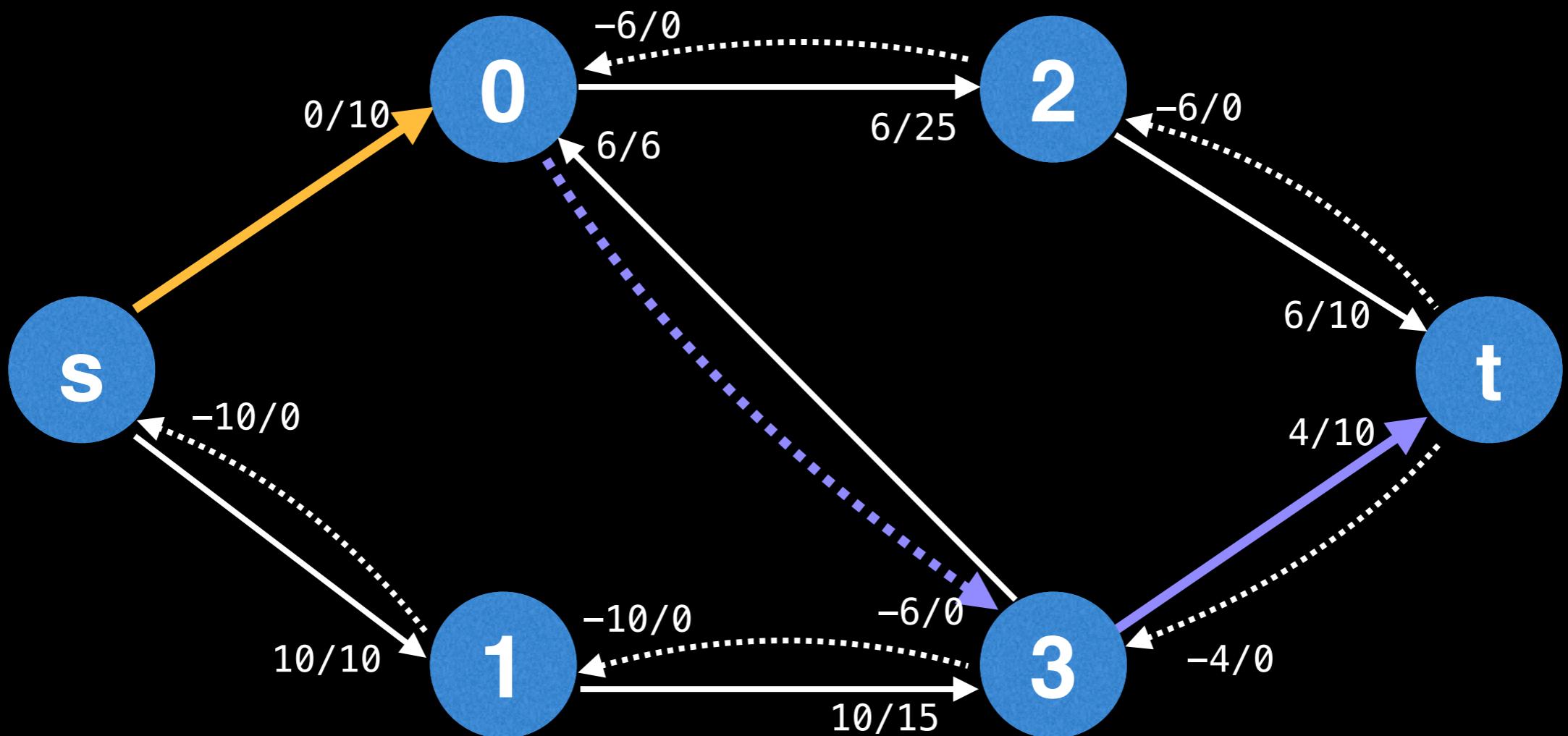
Bottleneck is 4 since the minimum of the remaining capacities on the augmenting path is 4:

$$\min(10 - 6, 15 - 6, 10 - 0) = \min(4, 9, 10) = 4$$



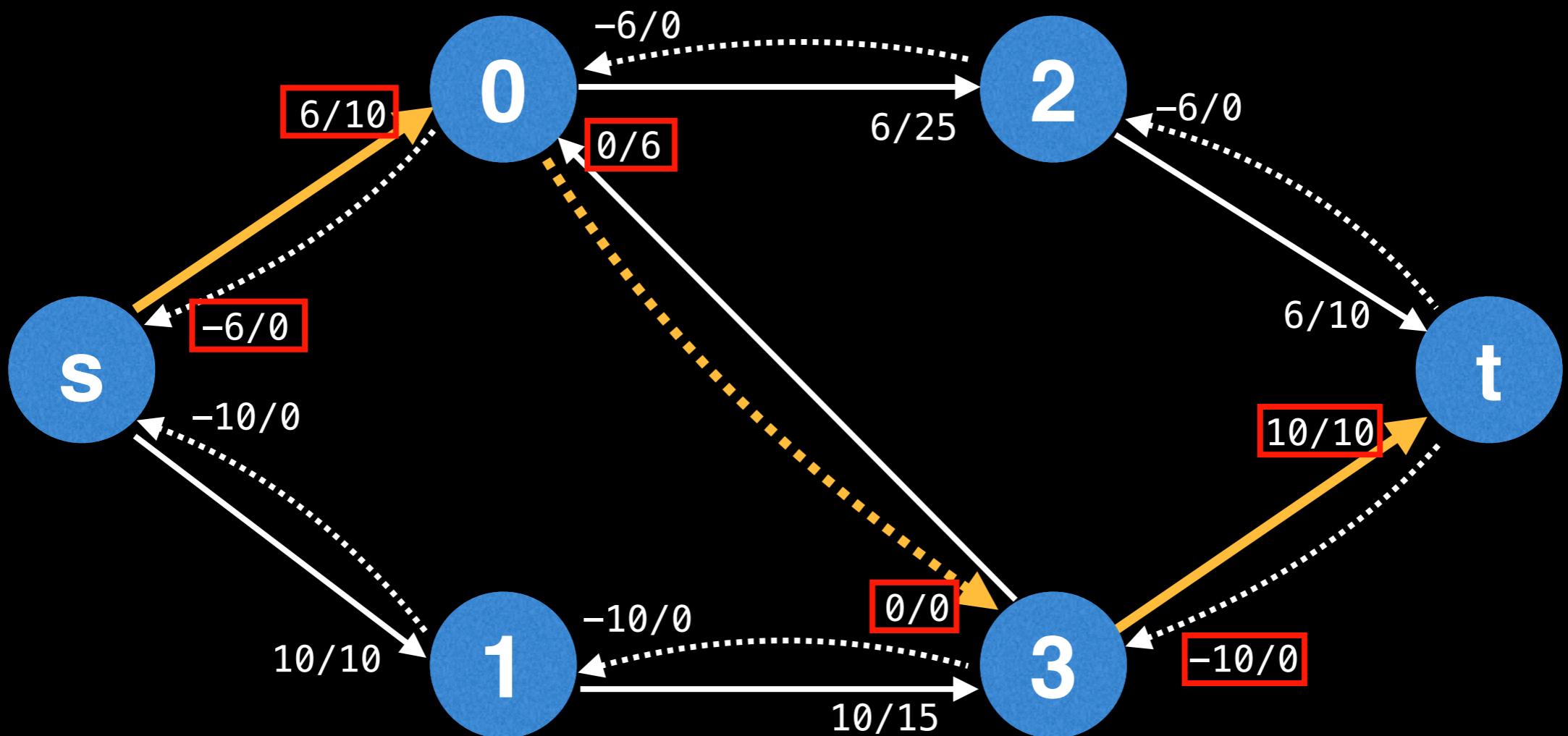
Augment the edges with bottleneck value of 4



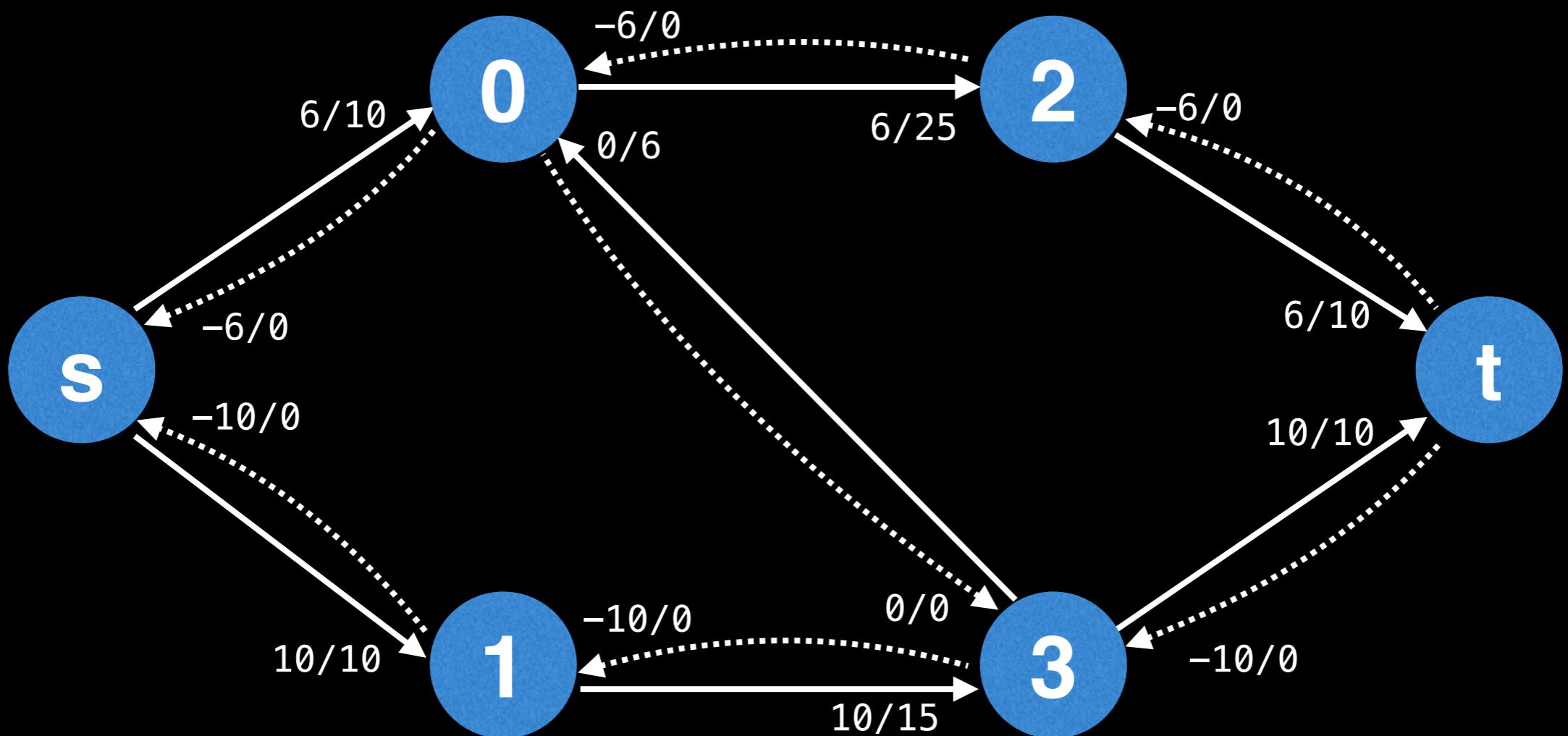


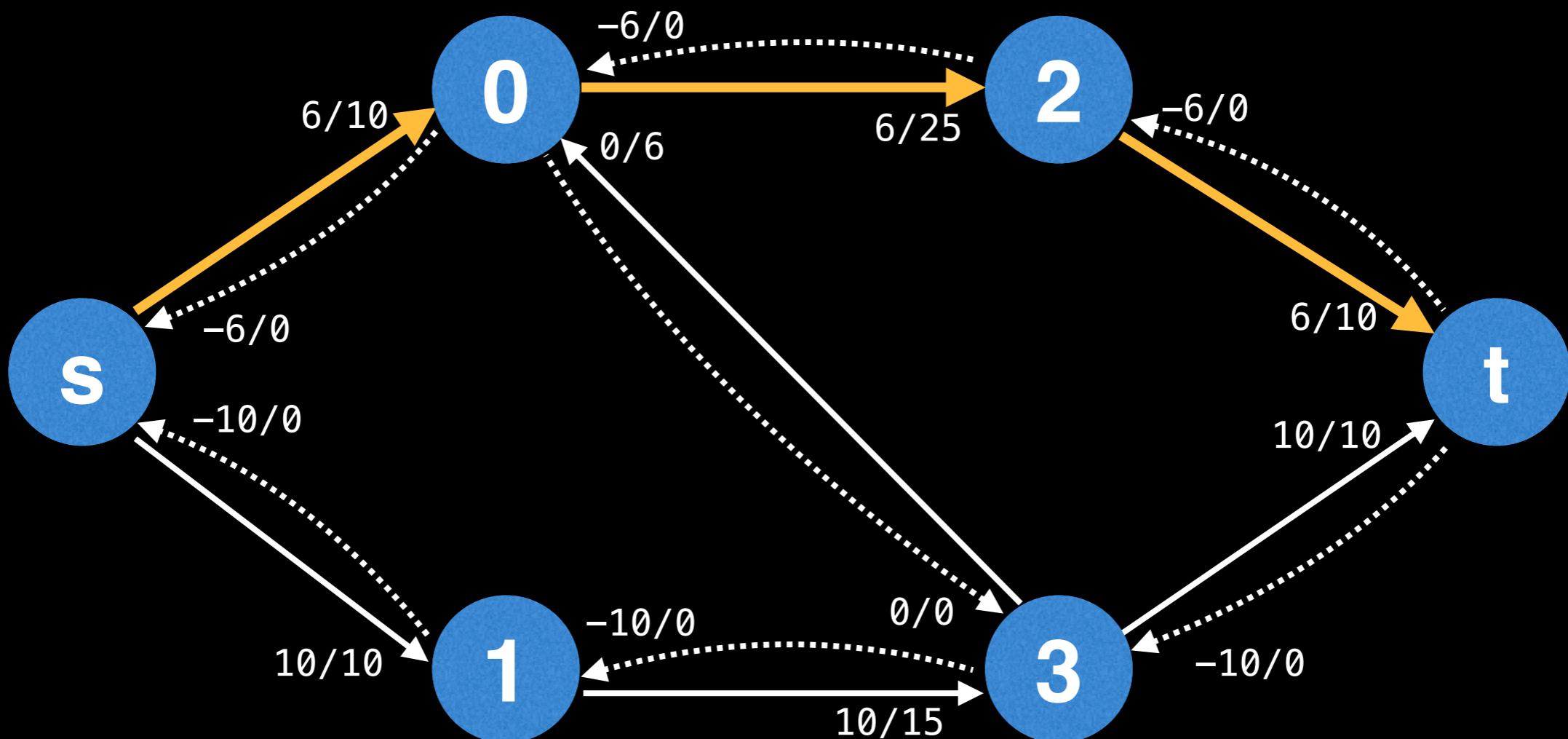
The bottleneck value is 6 since the smallest remaining capacity amongst the edges of our augmenting path is:

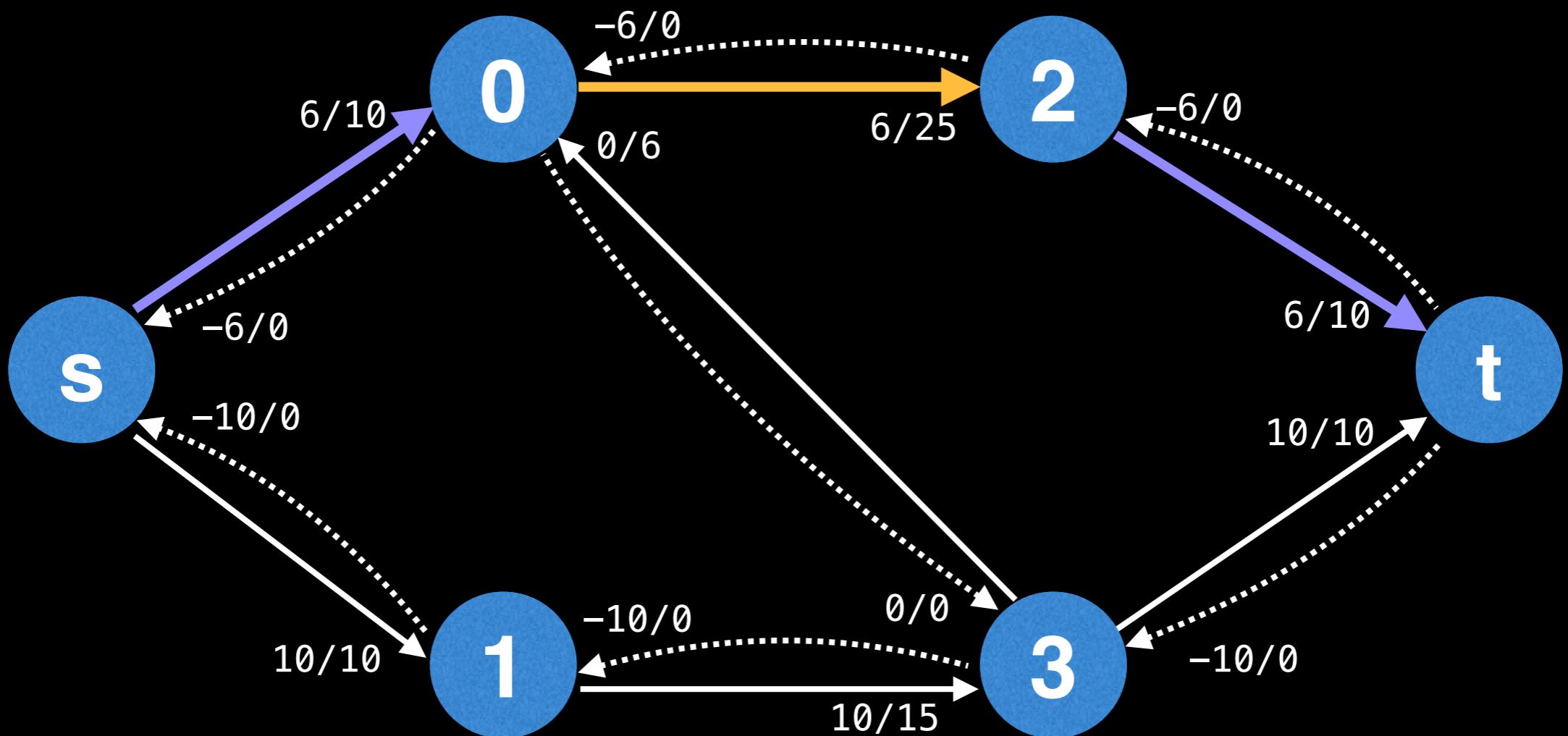
$$\min(10 - 0, 0 - -6, 10 - 4) = \min(10, 6, 6) = 6$$



Augment the edges with bottleneck value of 6

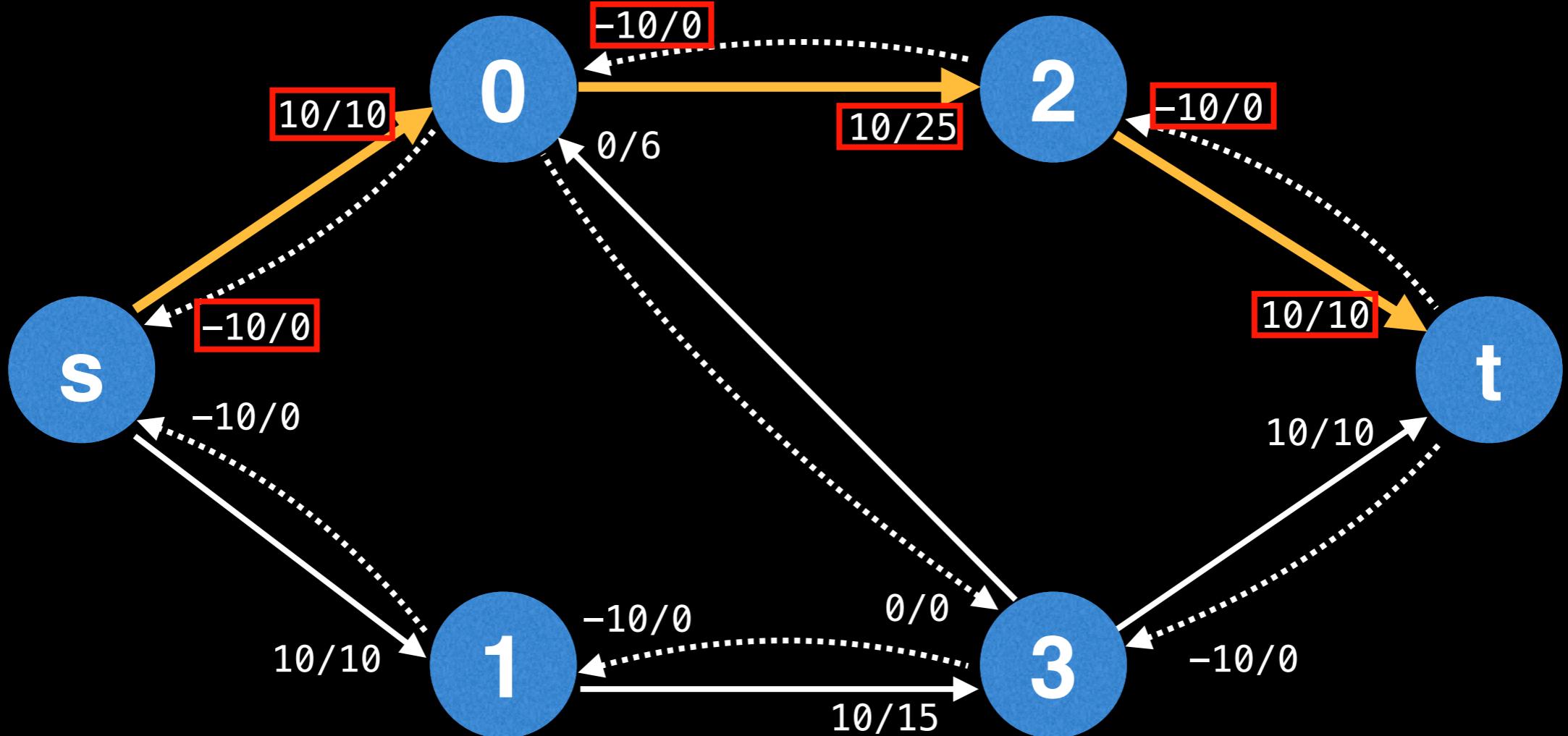




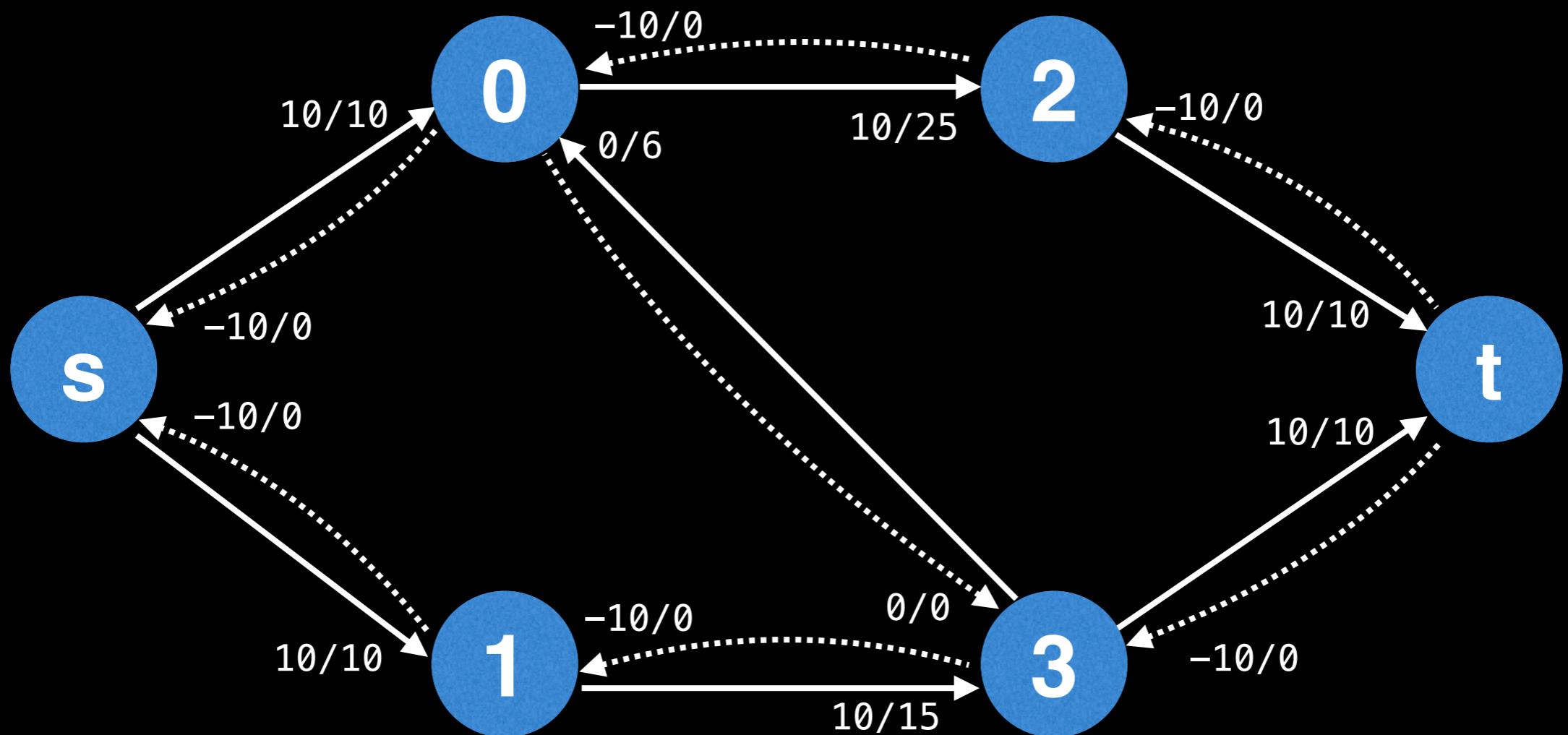


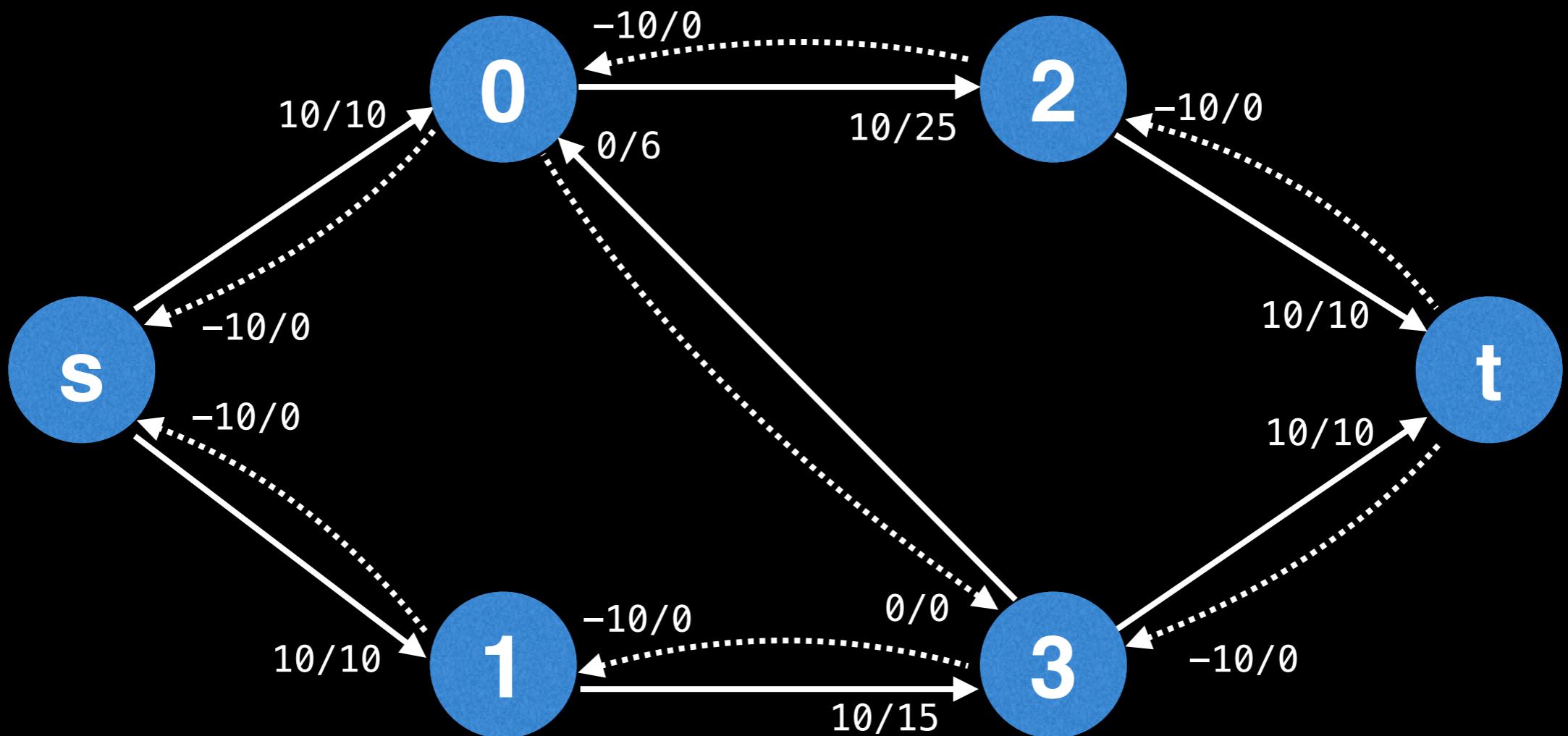
The bottleneck value is 4 since the smallest remaining capacity amongst the edges of our augmenting path is:

$$\min(10 - 6, 25 - 6, 10 - 6) = \min(4, 19, 4) = 4$$



Augment the edges with bottleneck value of 4





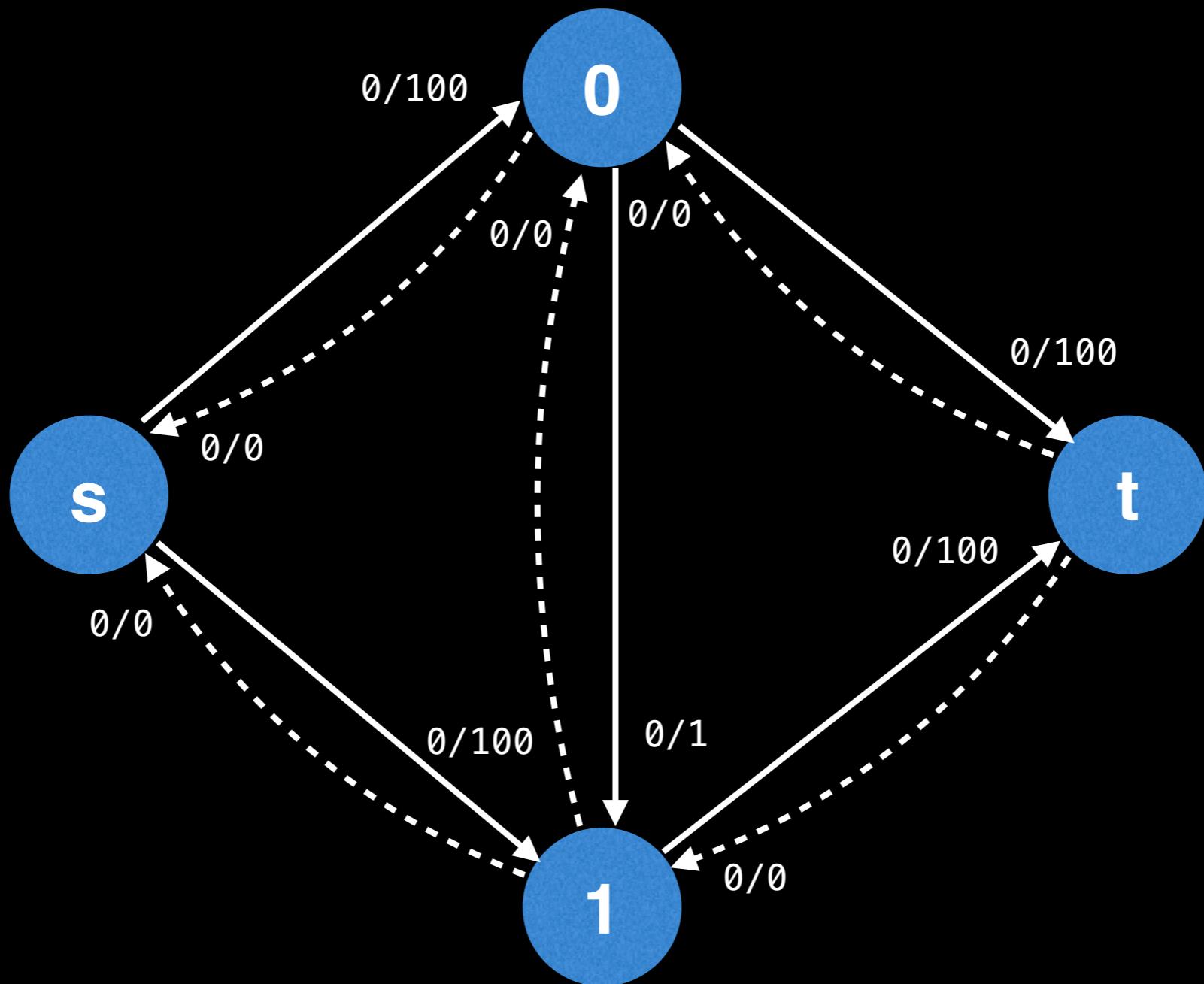
No more augmenting paths can be found,  
so the algorithm terminates!

maximum flow =  $6 + 4 + 6 + 4 = 20$   
 bottleneck values

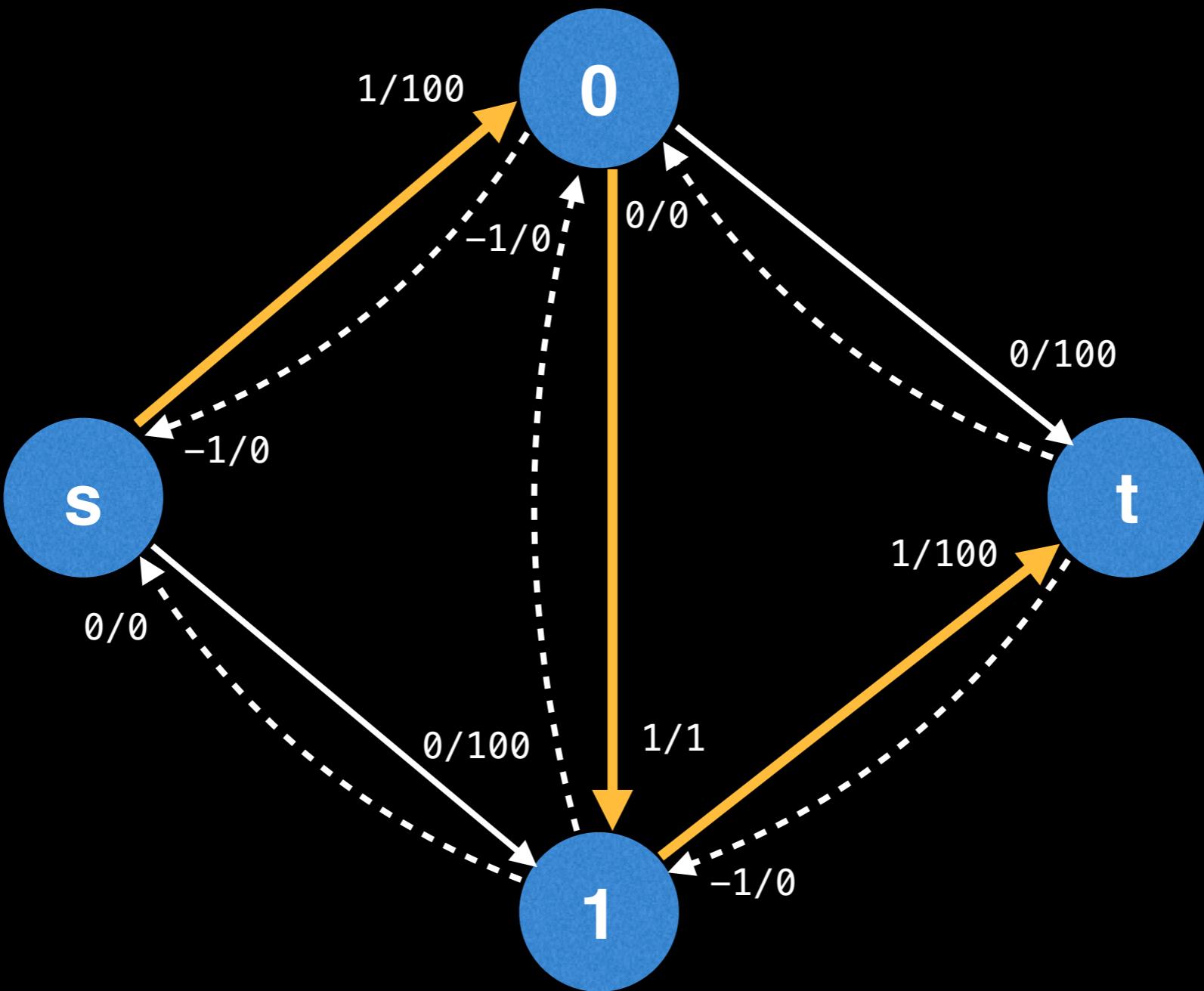
The time complexity of the Ford–Fulkerson method depends on the algorithm being used to find the augmenting paths, which is left unspecified.

Assuming the method of finding augmenting paths is by using a Depth First Search (DFS), the algorithm runs in  $O(fE)$ , where  $f$  is the maximum flow and  $E$  is the number of edges.

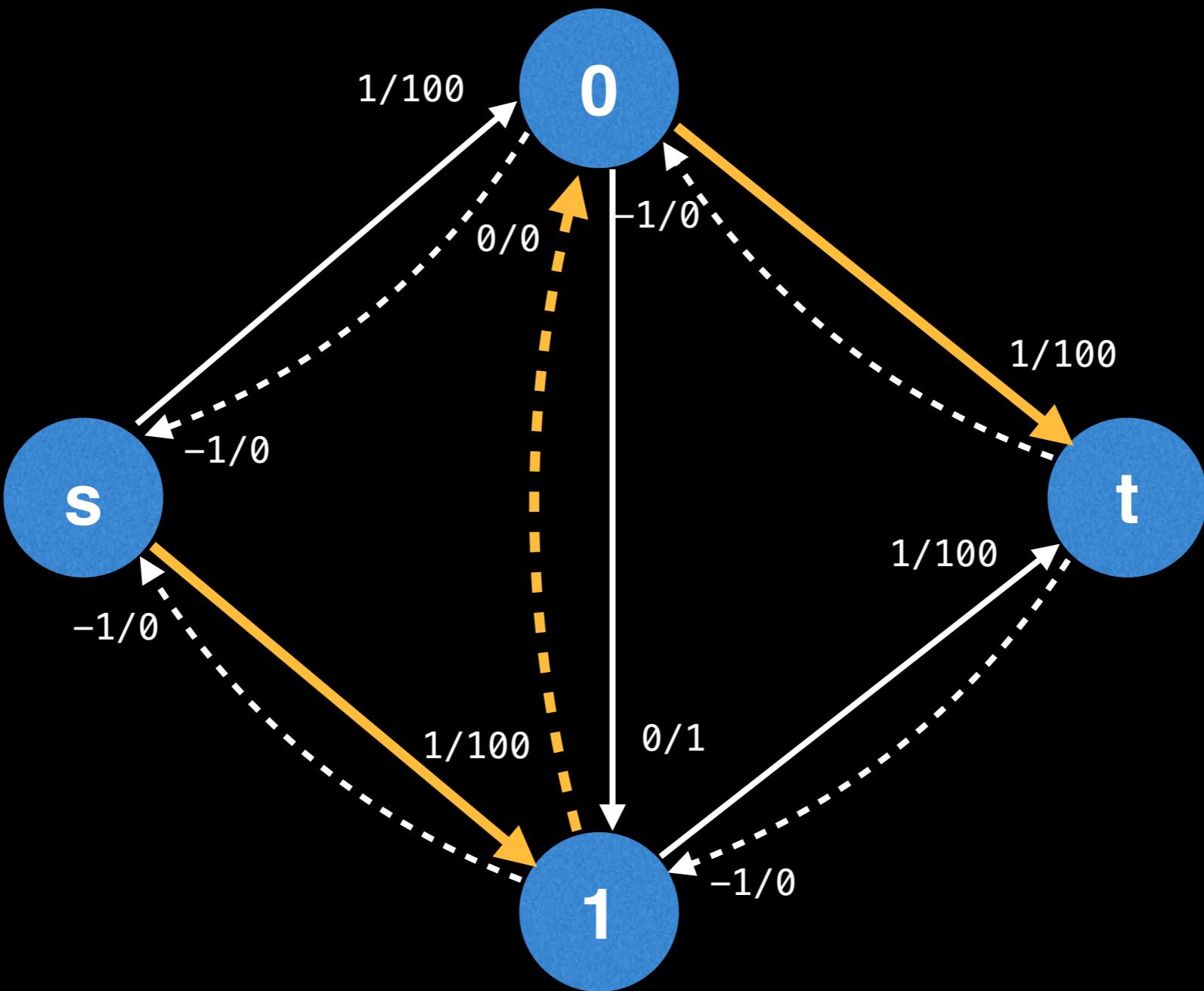
Assuming the method of finding augmenting paths is by using a Depth First Search (DFS), the algorithm runs in  $O(fE)$ , where  $f$  is the maximum flow and  $E$  is the number of edges.



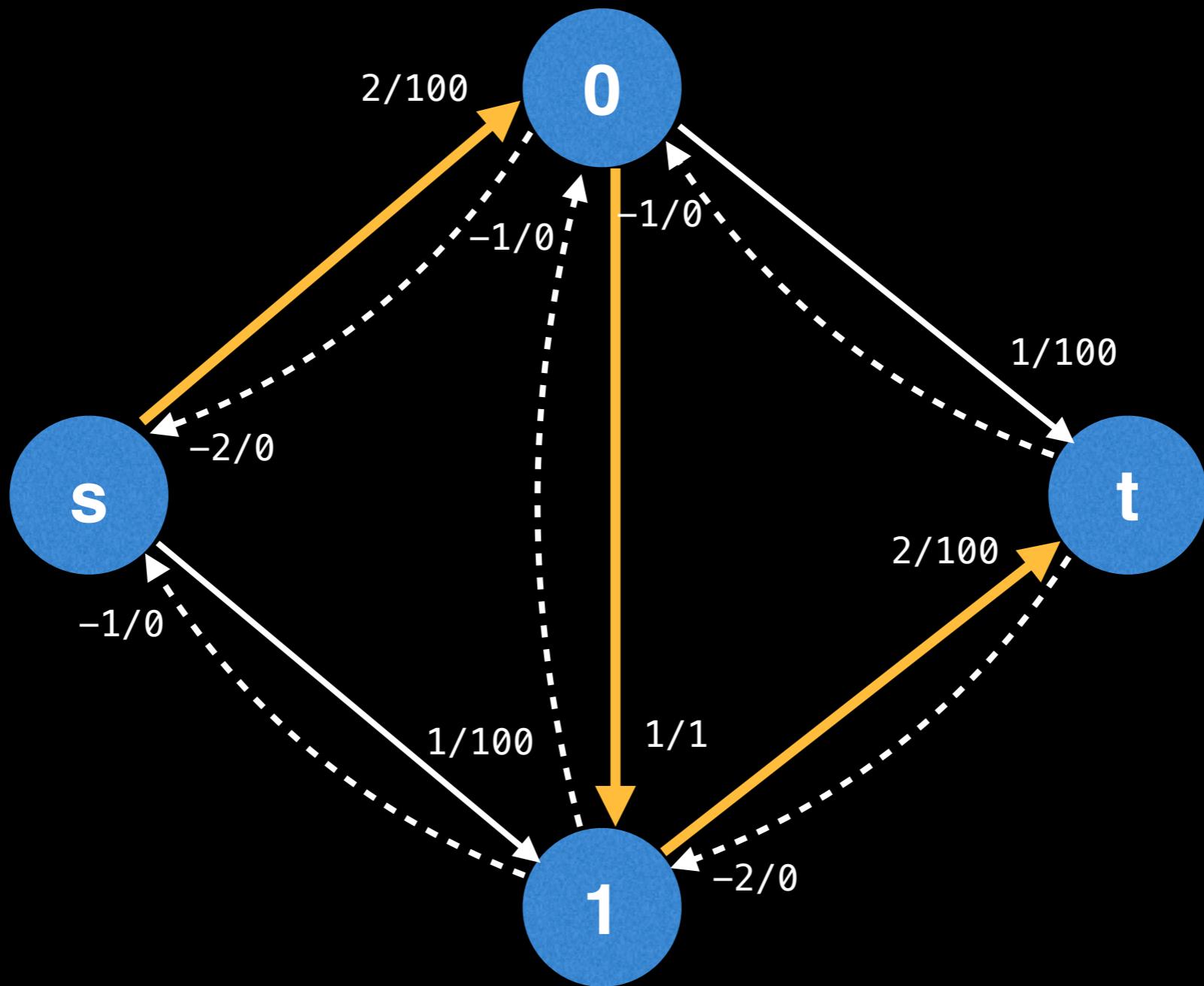
Recall that a DFS traversal chooses edges in a random order, so it is possible to pick the middle edge every time when finding an augmenting path.



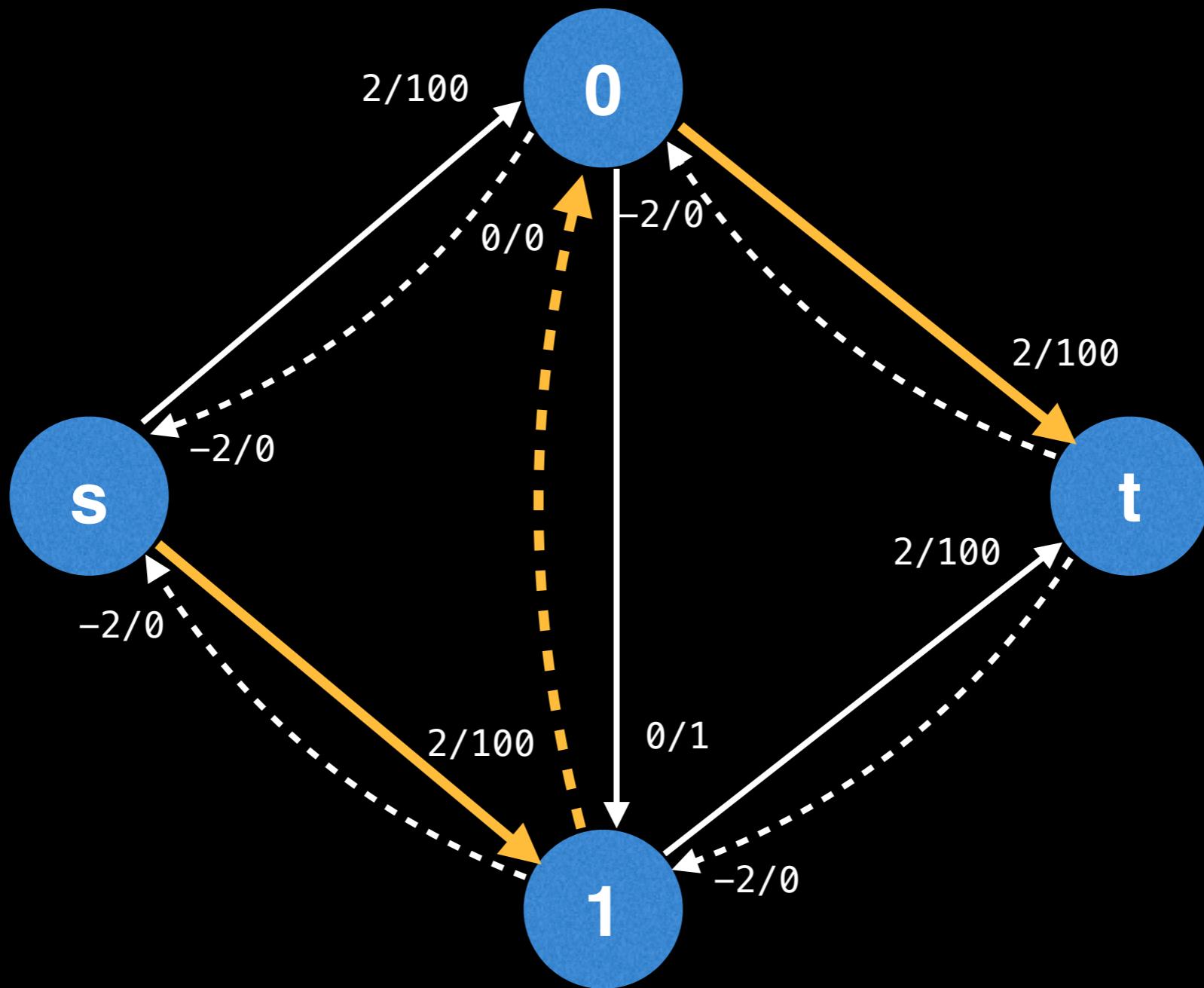
Recall that a DFS traversal chooses edges in a random order, so it is possible to pick the middle edge every time when finding an augmenting path.



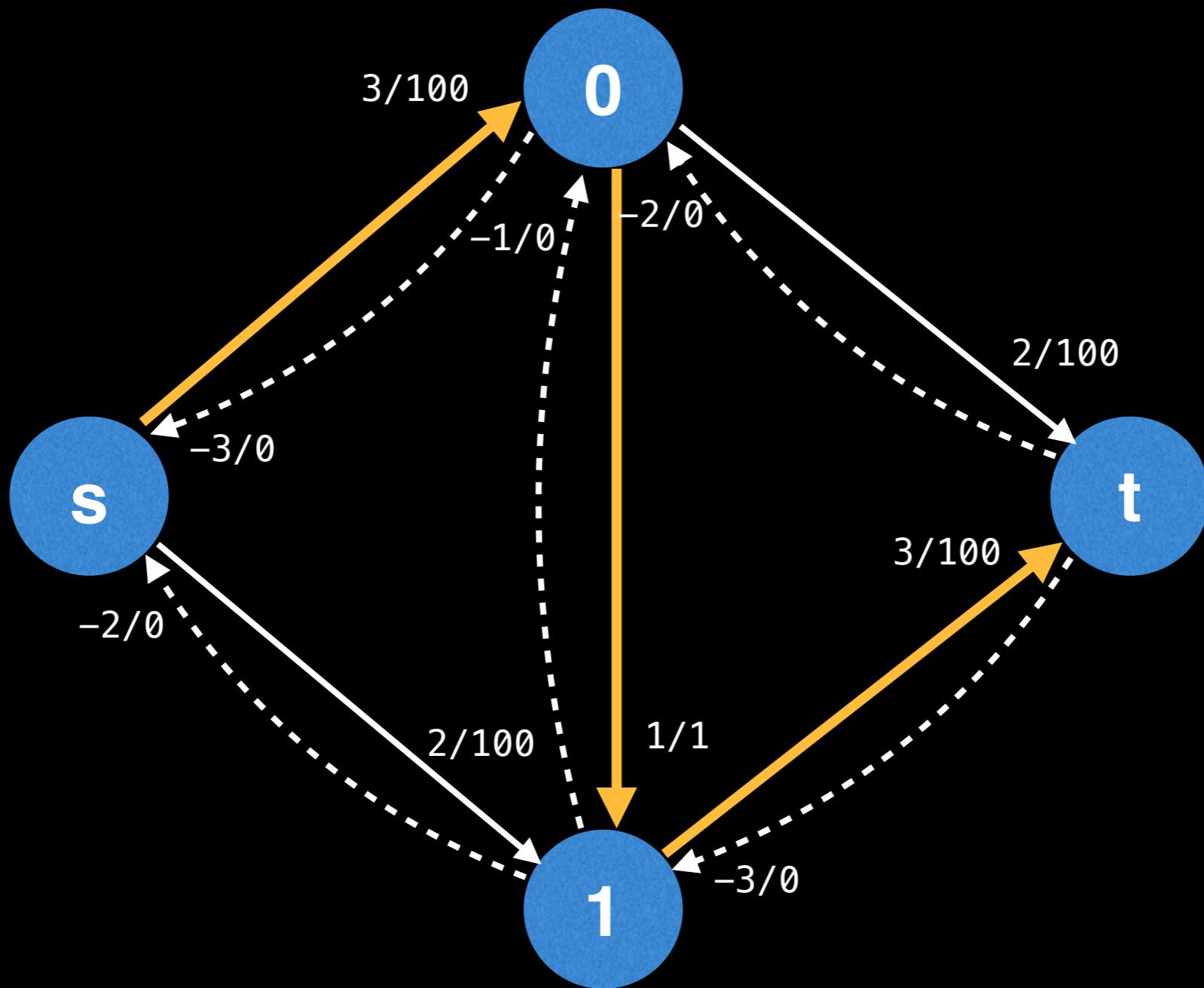
This results in flipping back and forth between the same two alternating paths for 200 iterations...



This results in flipping back and forth between the same two alternating paths for 200 iterations...



This results in flipping back and forth between the same two alternating paths for 200 iterations...



Other much faster algorithms and heuristics exist to find the maximum flow:

**Edmonds–Karp**: Uses a BFS as a method of finding augmenting paths,  $O(E^2V)$

**Capacity scaling**: Adds a heuristic on top of Ford–Fulkerson to pick larger paths first,  $O(E^2 \log(U))$  (where  $U$  is the value of the largest edge capacity in the initial flow graph).

**Dinic's algorithm**: Uses combination of BFS + DFS to find augmenting paths,  $O(V^2E)$

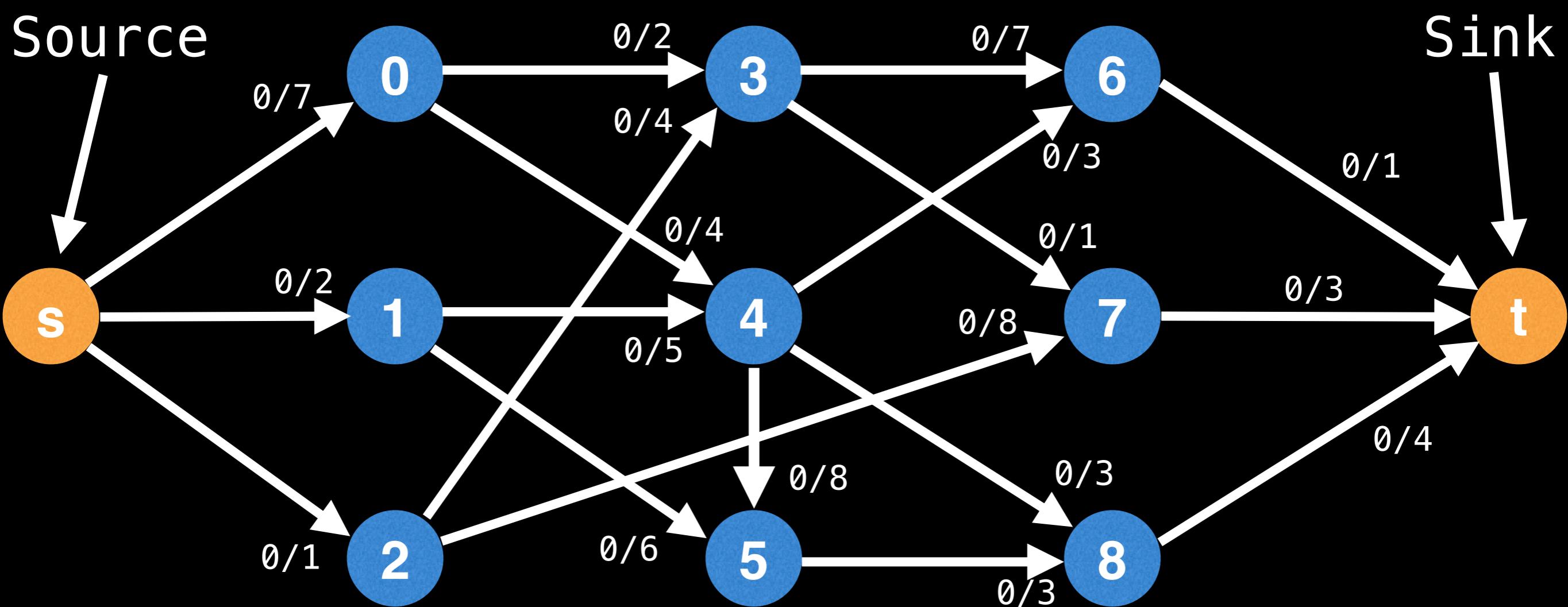
**Push Relabel**: Uses a concept of maintaining a "preflow" instead of finding augmenting paths to achieve a max-flow solution,  $O(V^2E)$  or  $O(V^2\sqrt{E})$  variant.

**NOTE:** Be mindful that the time complexities for flow algorithms are very pessimistic. In practice, they tend to operate much faster, making it hard to compare the performance of flow algorithms solely based on complexity.

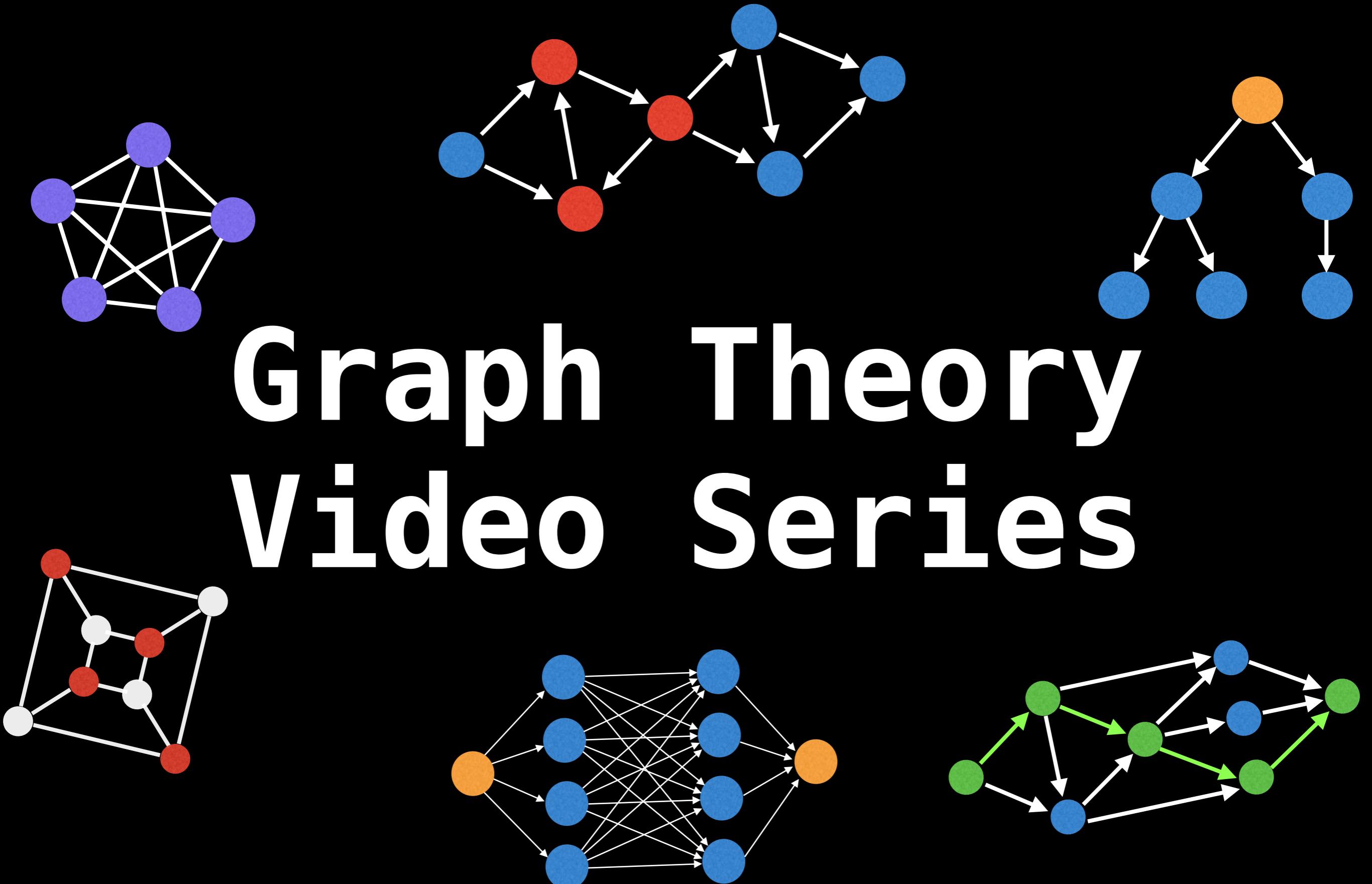
# Next Video: Source Code



# Network Flow: Ford-Fulkerson Max Flow



# Graph Theory Video Series



# Max Flow

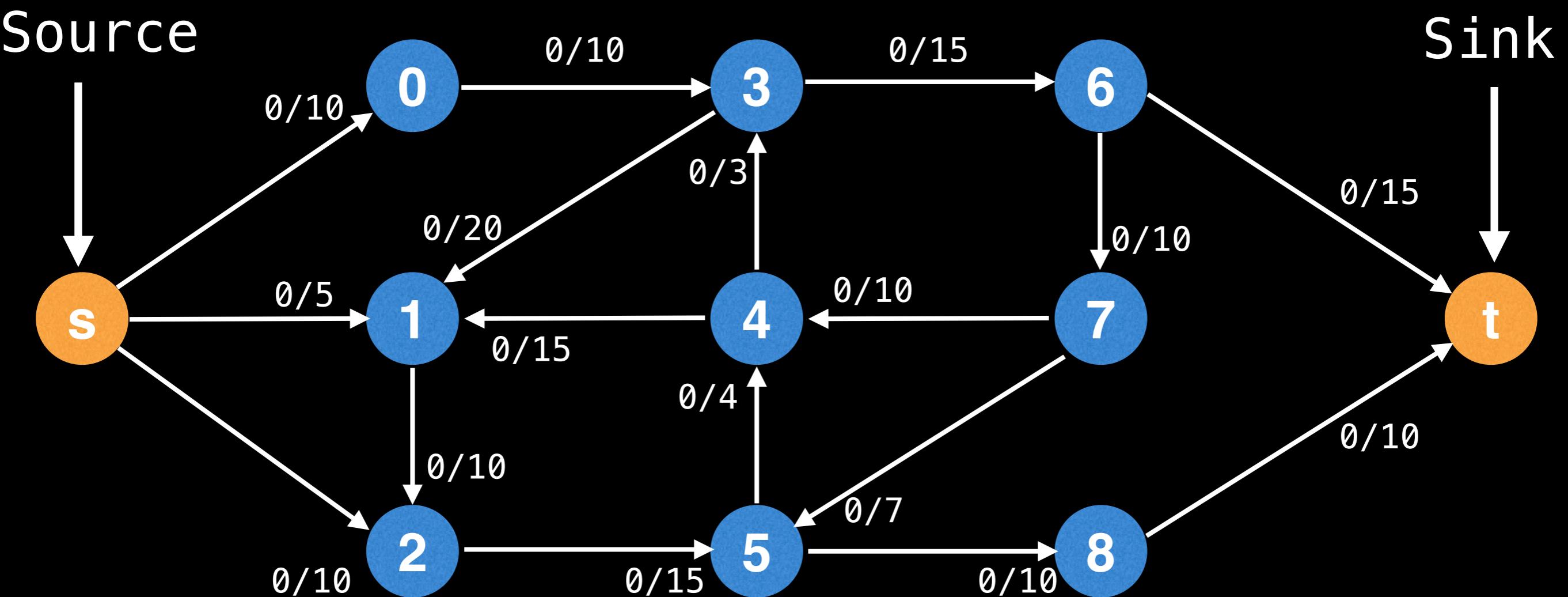
# Ford-Fulkerson

# source code

William Fiset

Previous video explaining  
Ford-Fulkerson method:

Let's setup this flow graph:

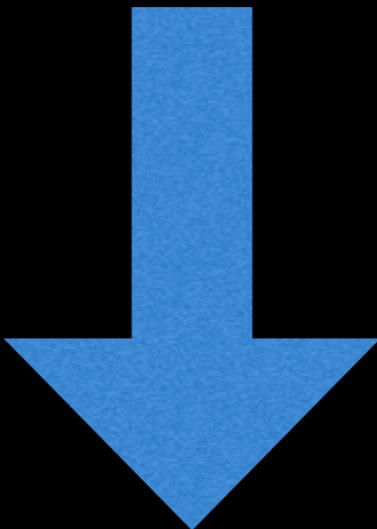


# Source Code Link

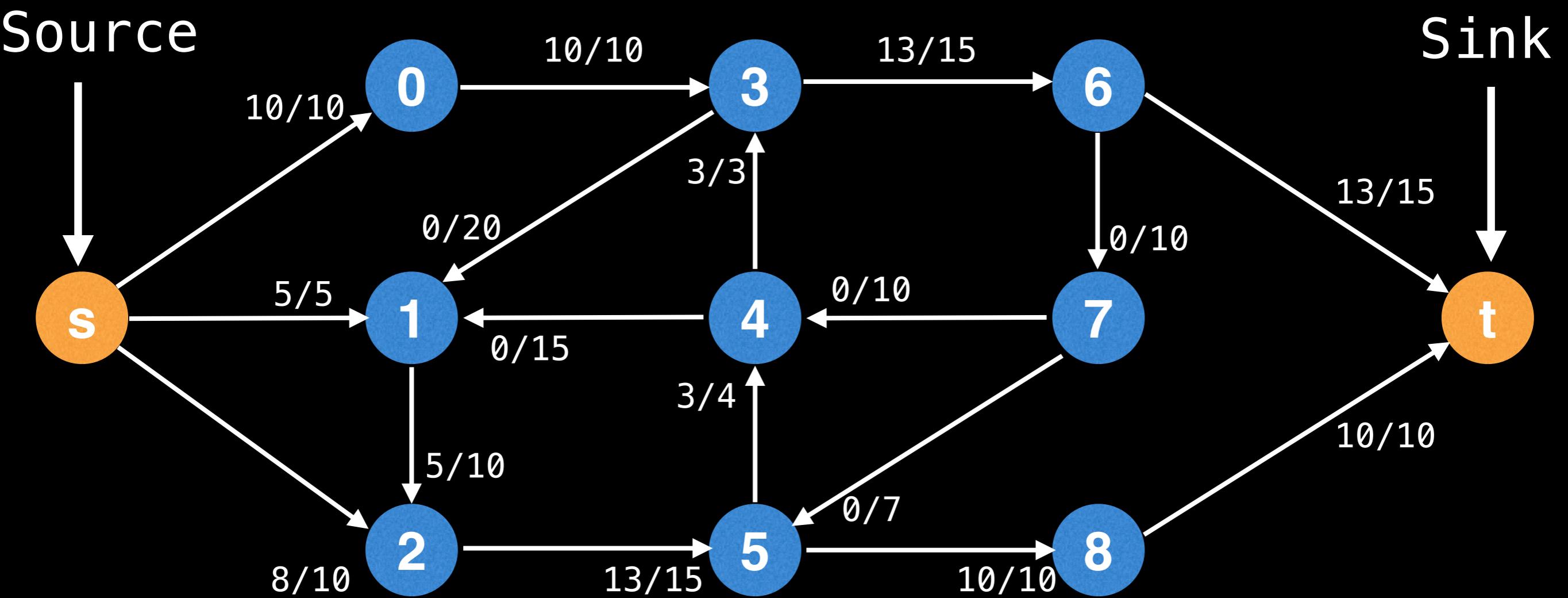
Implementation source code can  
be found at the following link:

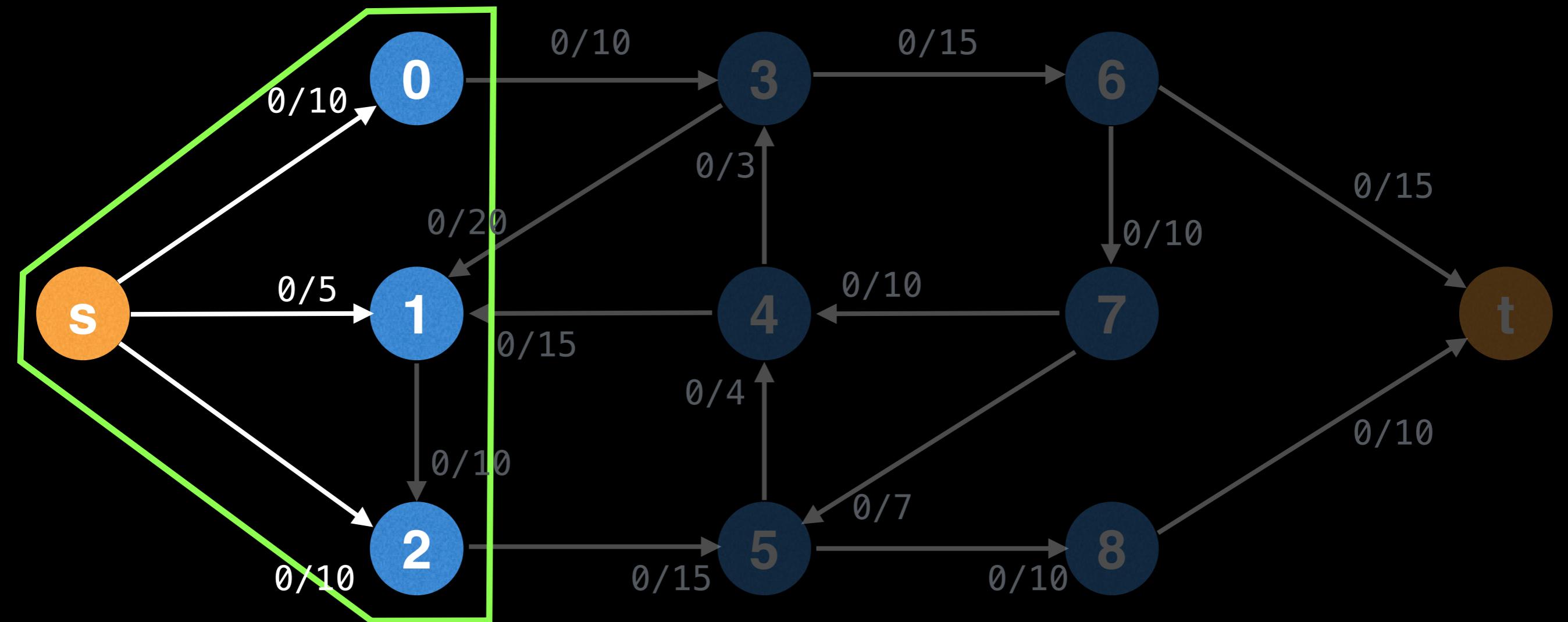
[github.com/williamfiset/algorithms](https://github.com/williamfiset/algorithms)

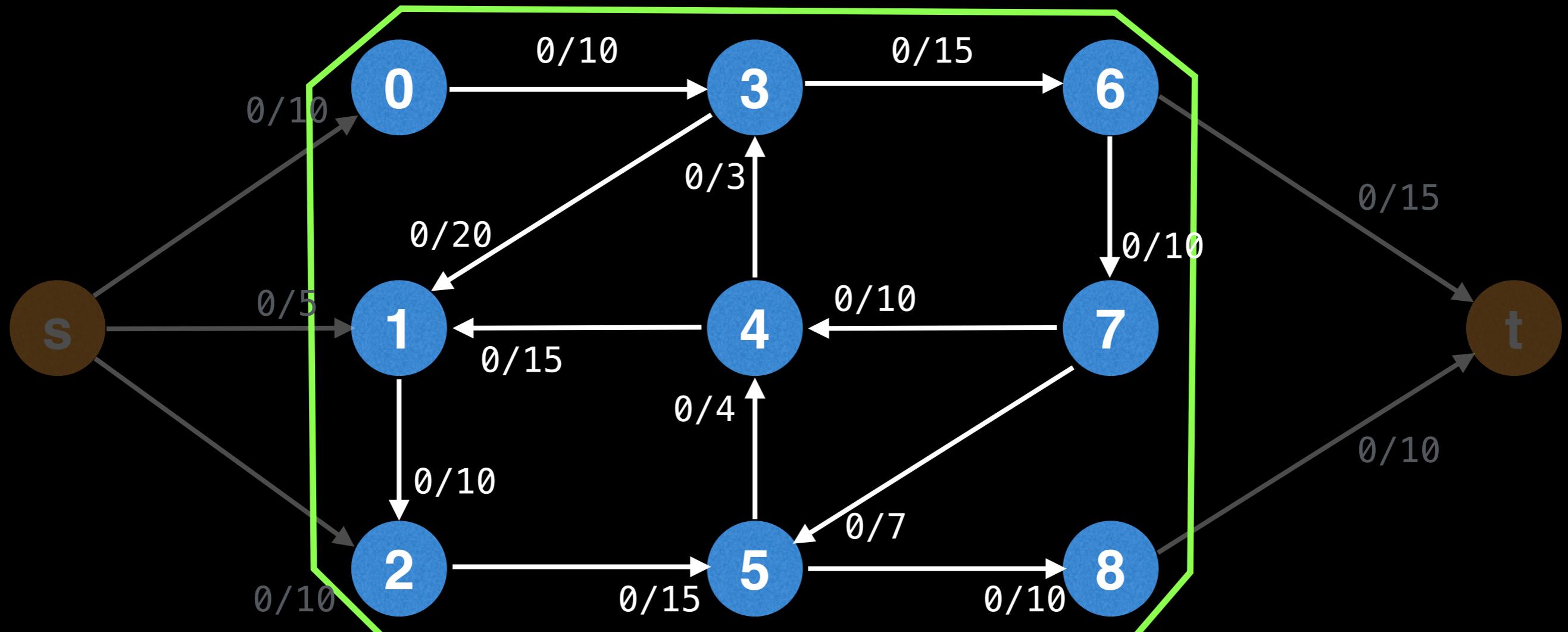
Link in the description:

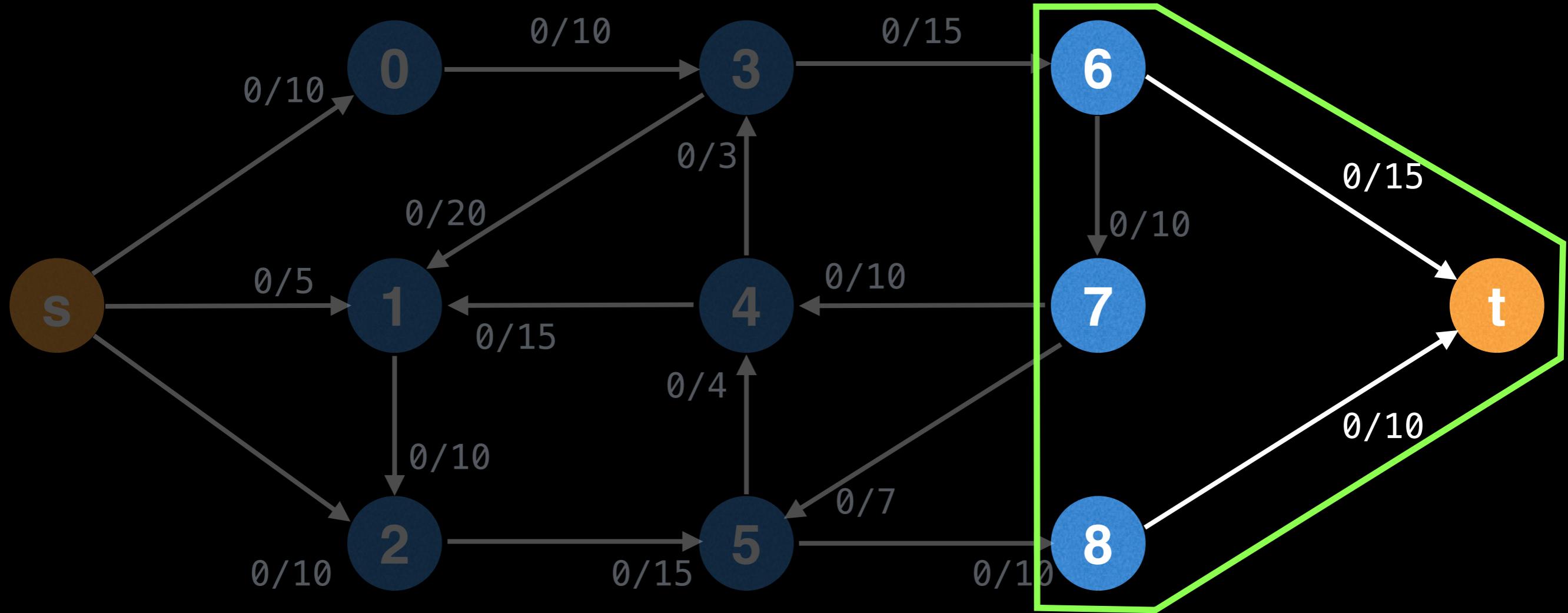


Then let's code Ford-Fulkerson and hope to achieve the following graph (or similar) with max-flow of 23:

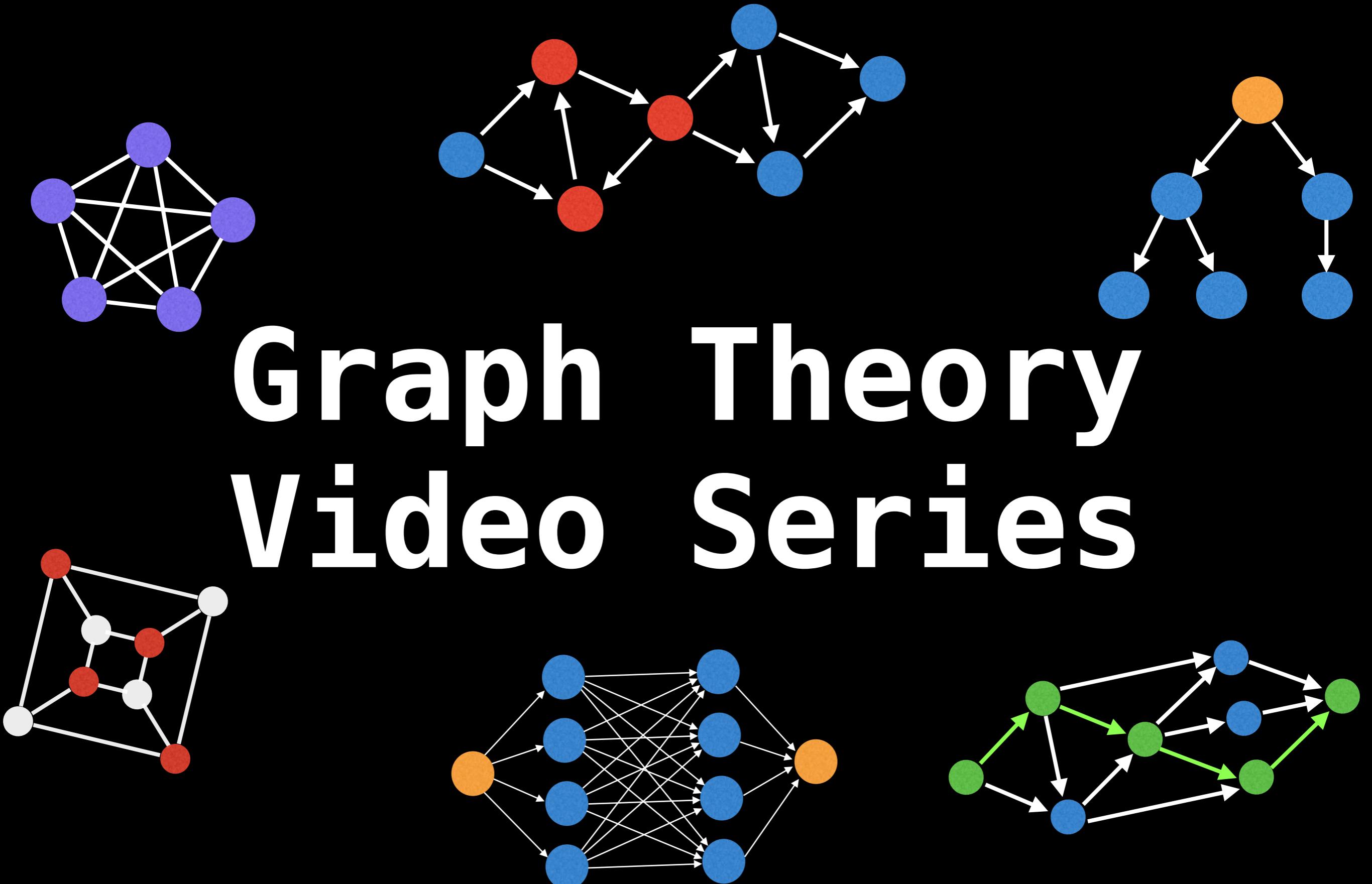








# Graph Theory Video Series



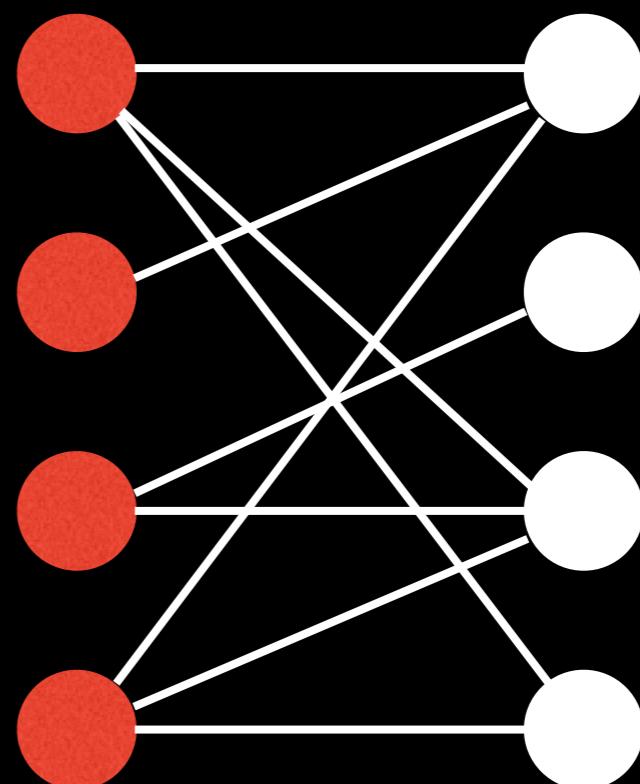
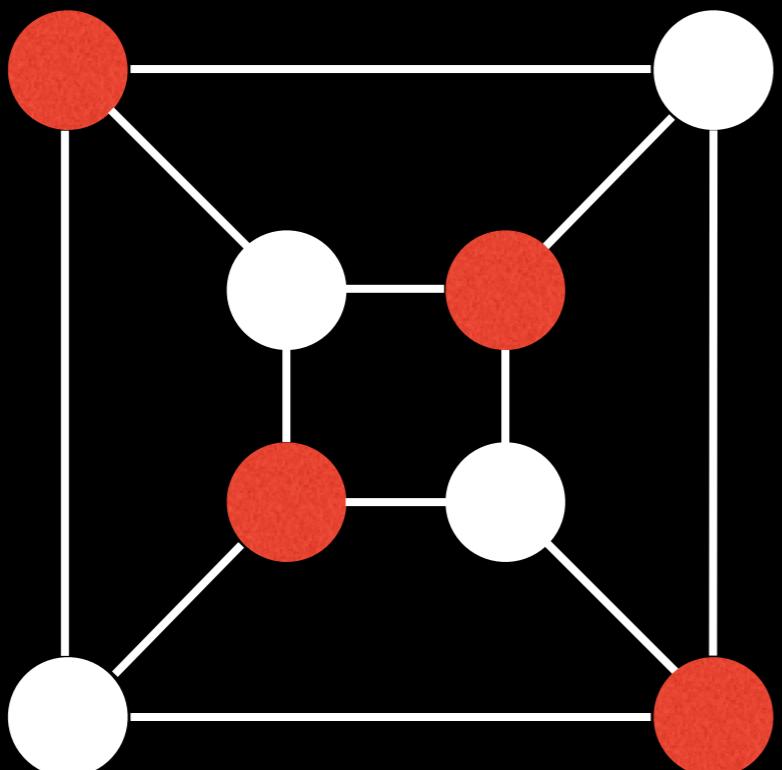
# Network Flow: unweighted bipartite graph matching

William Fiset

# Bipartite Graph

A **bipartite graph** is one whose *vertices* can be split into two independent groups  $U, V$  such that every edge connects between  $U$  and  $V$ .

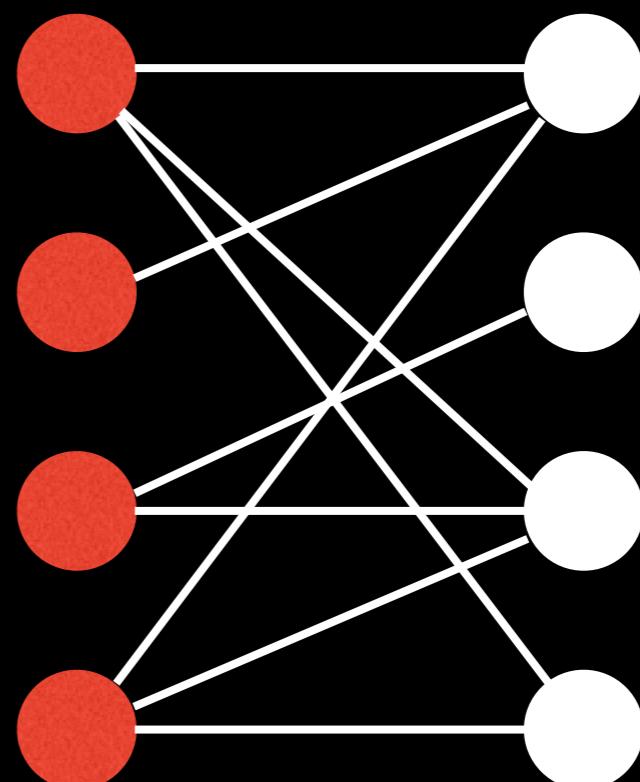
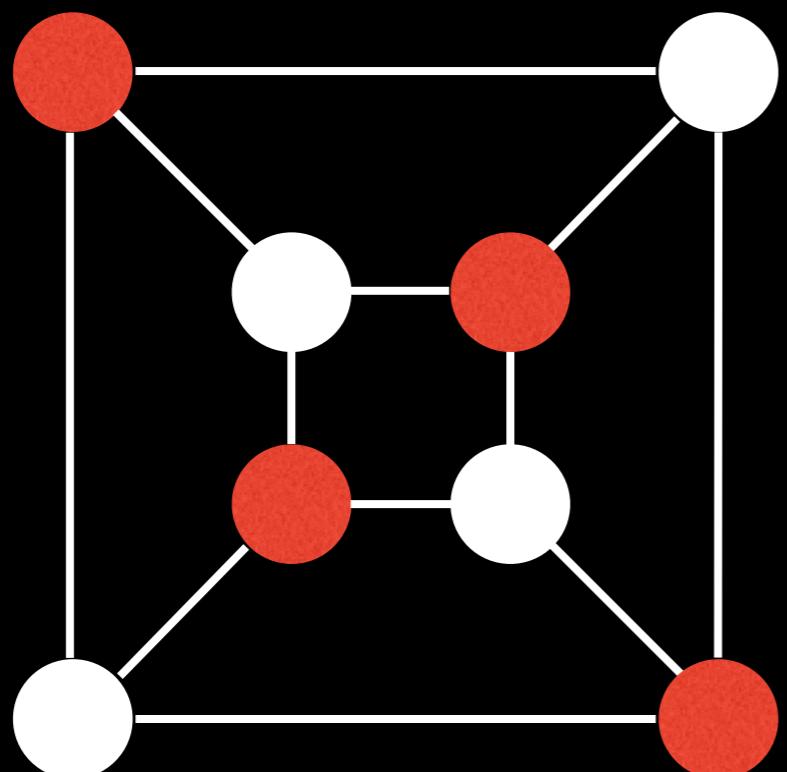
Other definitions exist such as: The graph is two colourable or there is no cycle with an odd length.



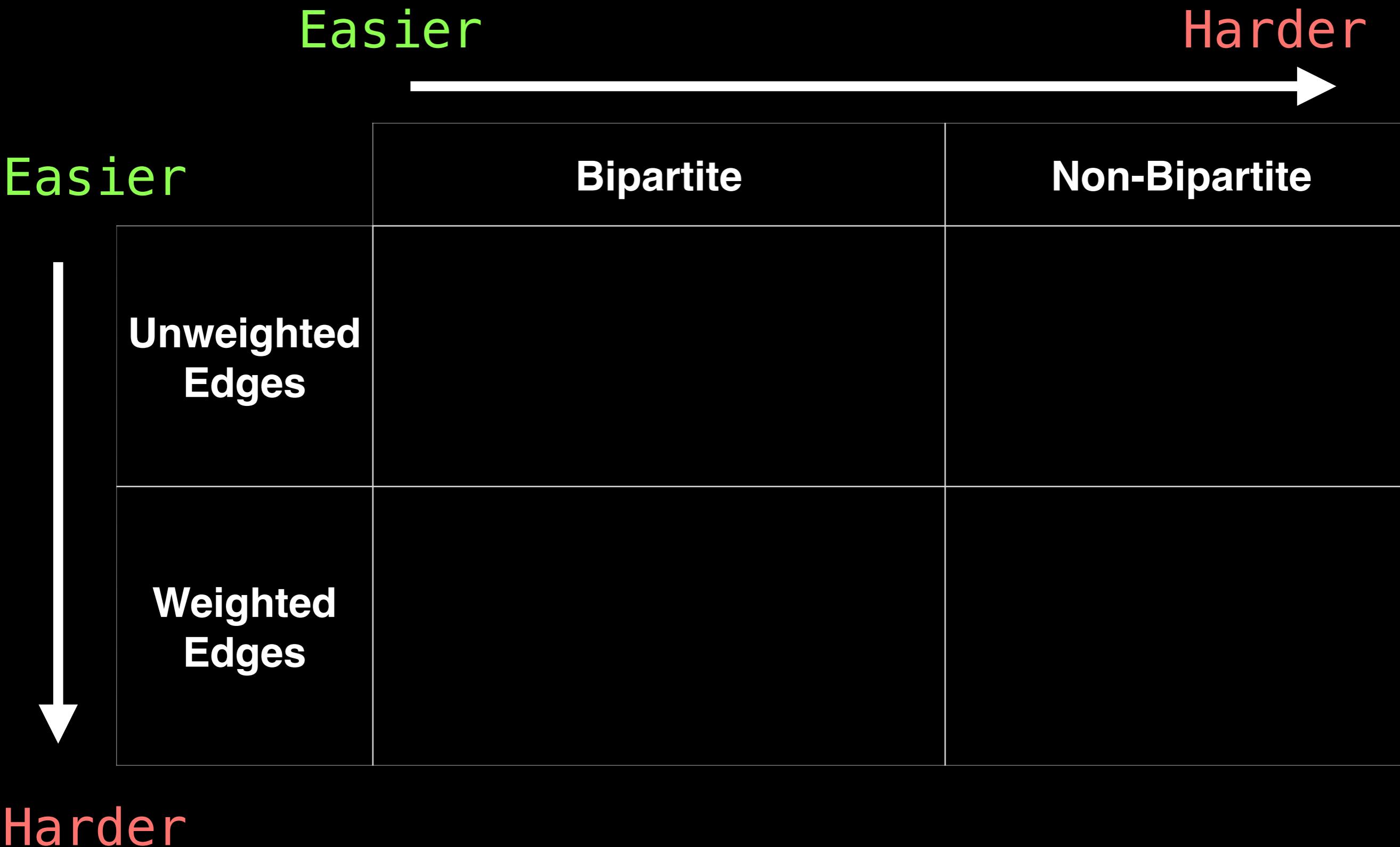
# Maximum Cardinality Matching

Generally we're interested in what's called a **Maximum Cardinality Bipartite Matching (MCBM)**. This is when we've maximized the pairs that can be matched with each other.

There exists several applications such as matching candidates to jobs, chairs to desks, surfers to surf boards, etc...

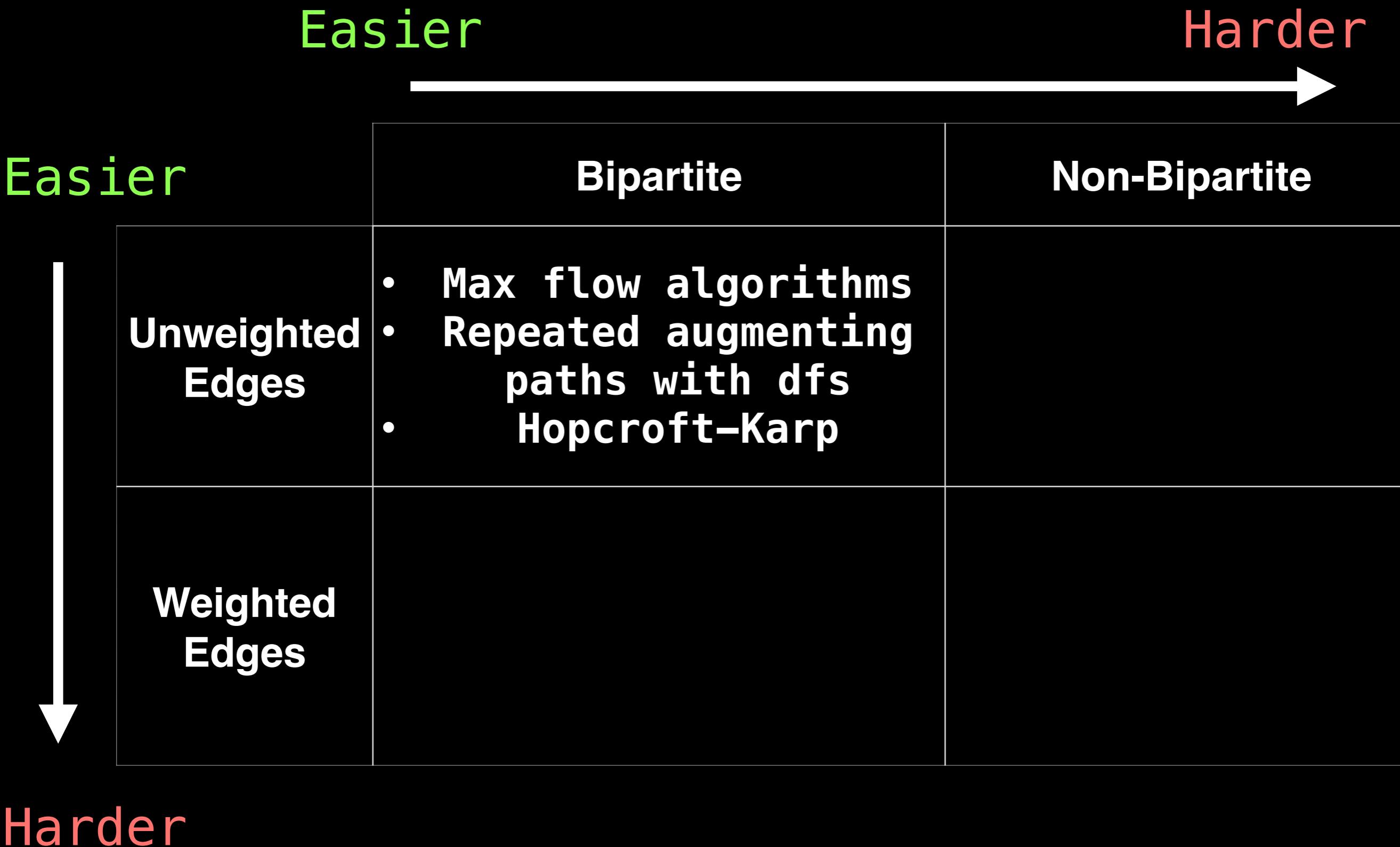


# Common matching variations



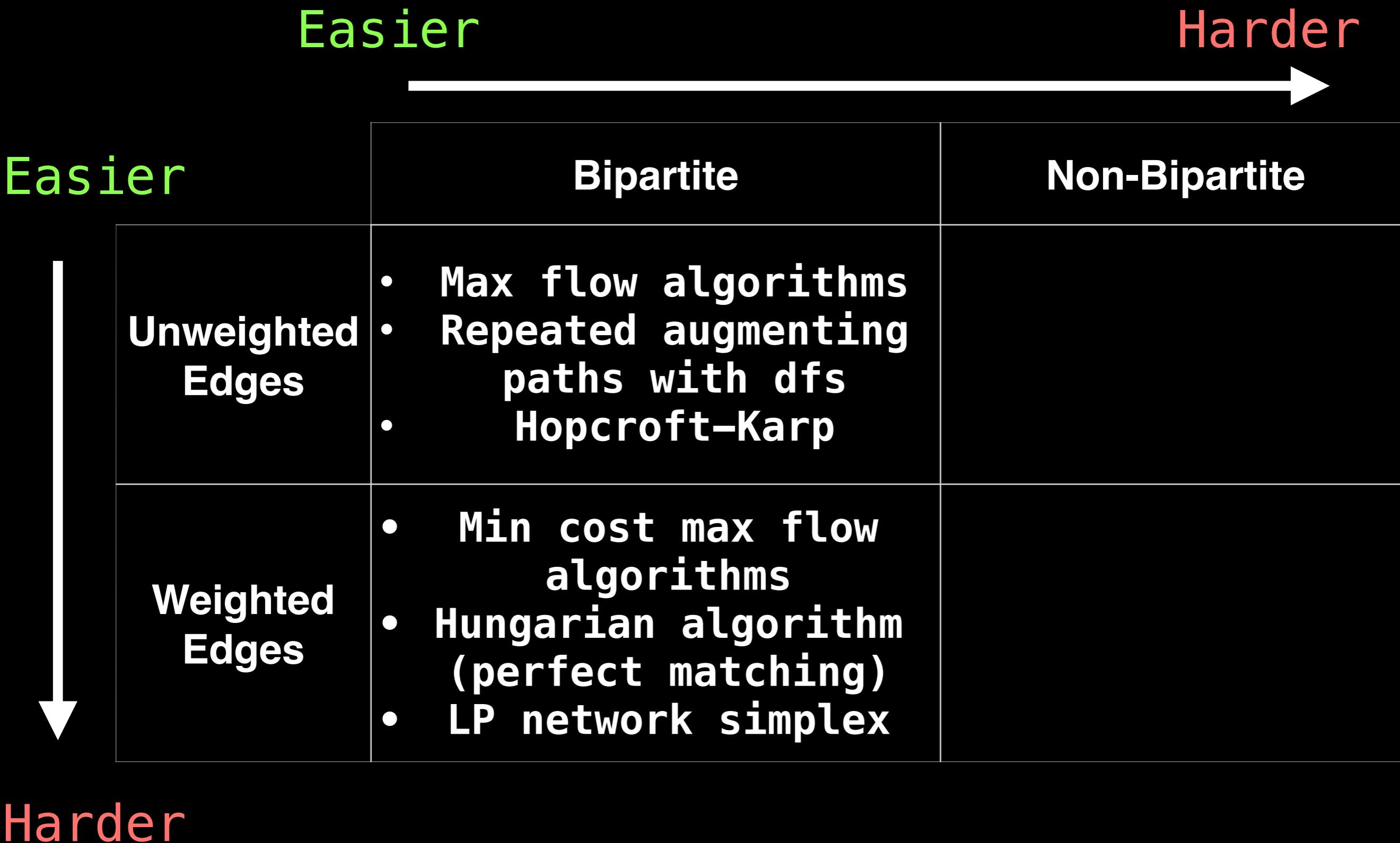
Reference: Competitive Programming 3 The New Lower Bound of Programming Contests. P 349

# Common matching variations



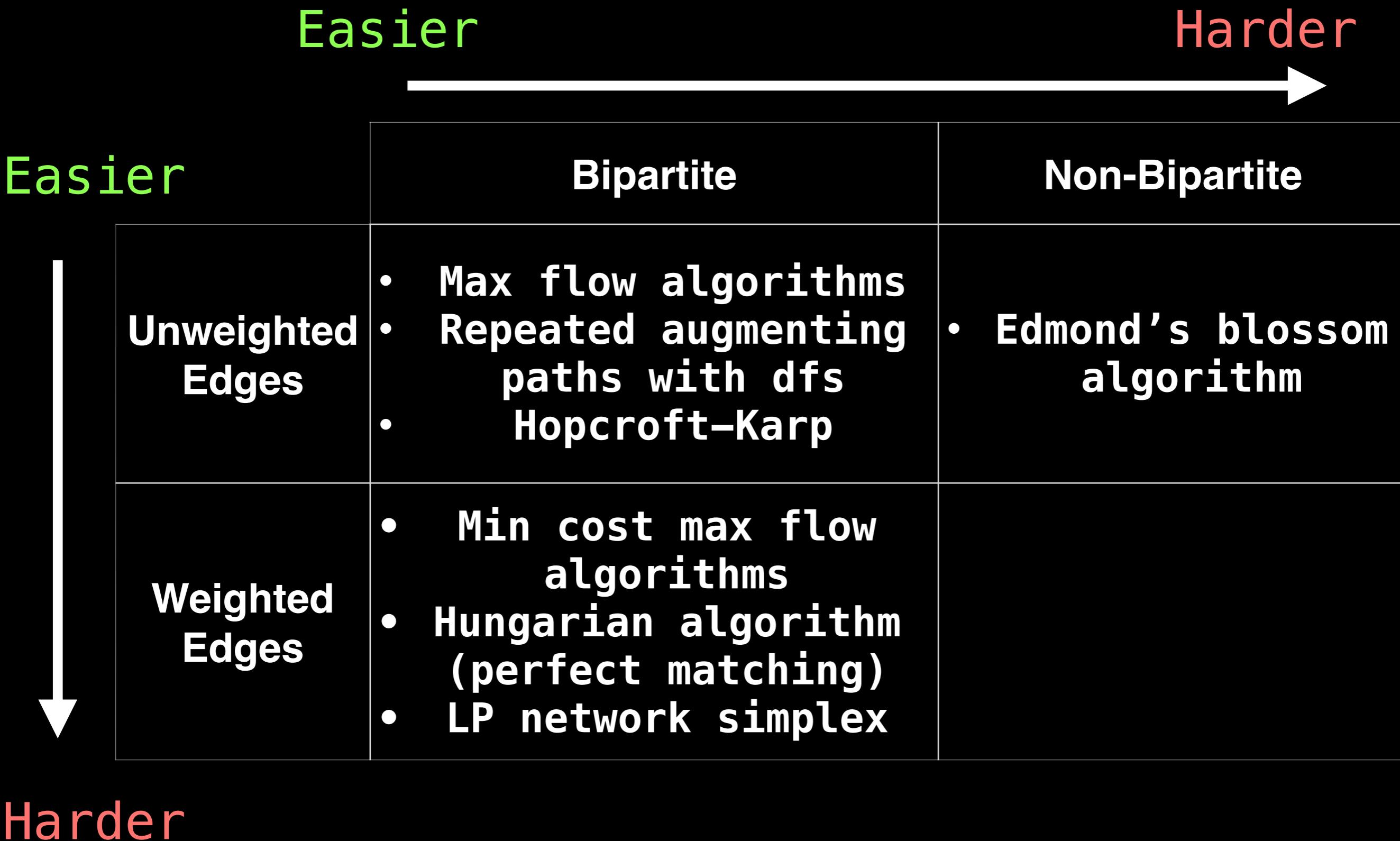
Reference: Competitive Programming 3 The New Lower Bound of Programming Contests. P 349

# Common matching variations



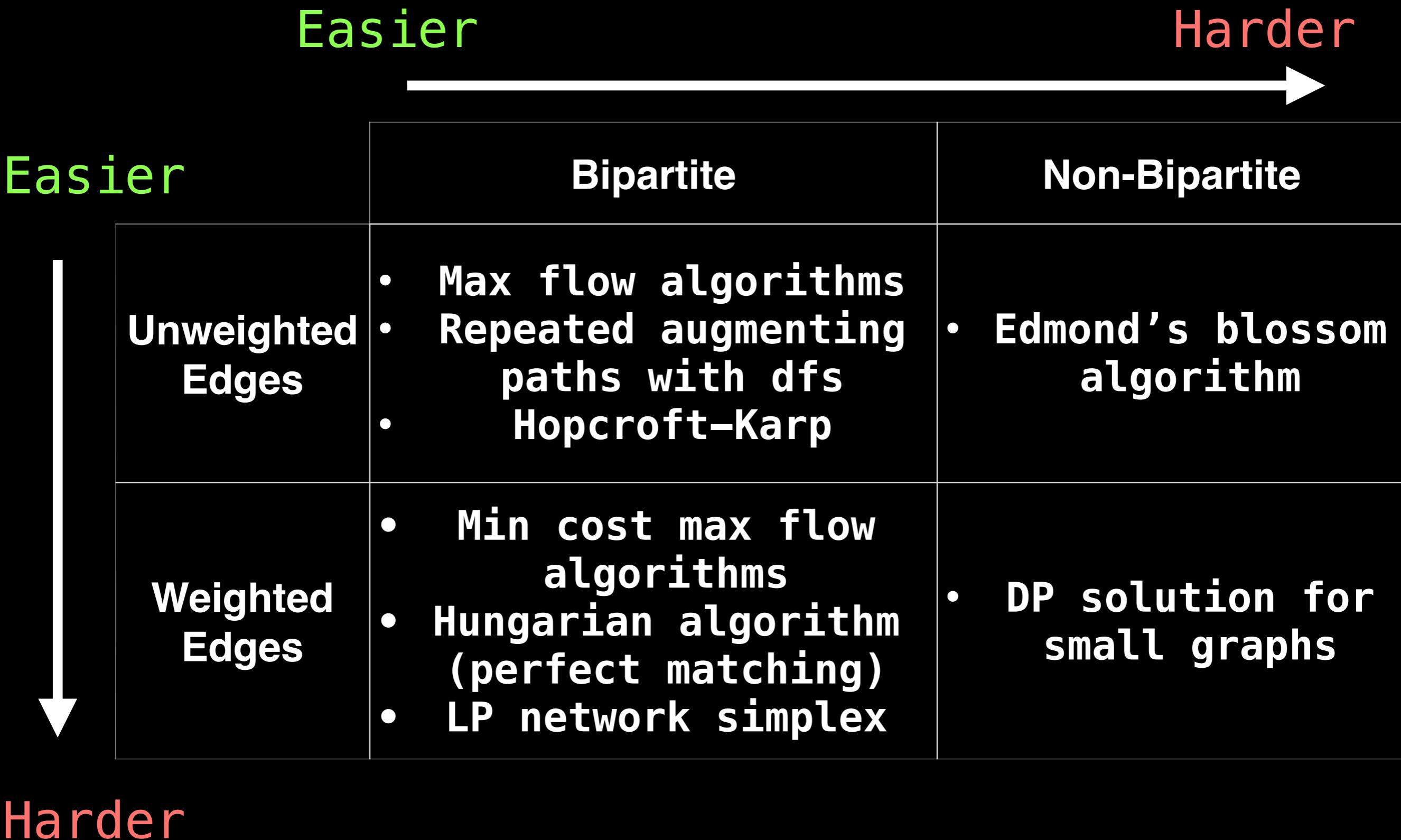
Reference: Competitive Programming 3 The New Lower Bound of Programming Contests. P 349

# Common matching variations



Reference: Competitive Programming 3 The New Lower Bound of Programming Contests. P 349

# Common matching variations

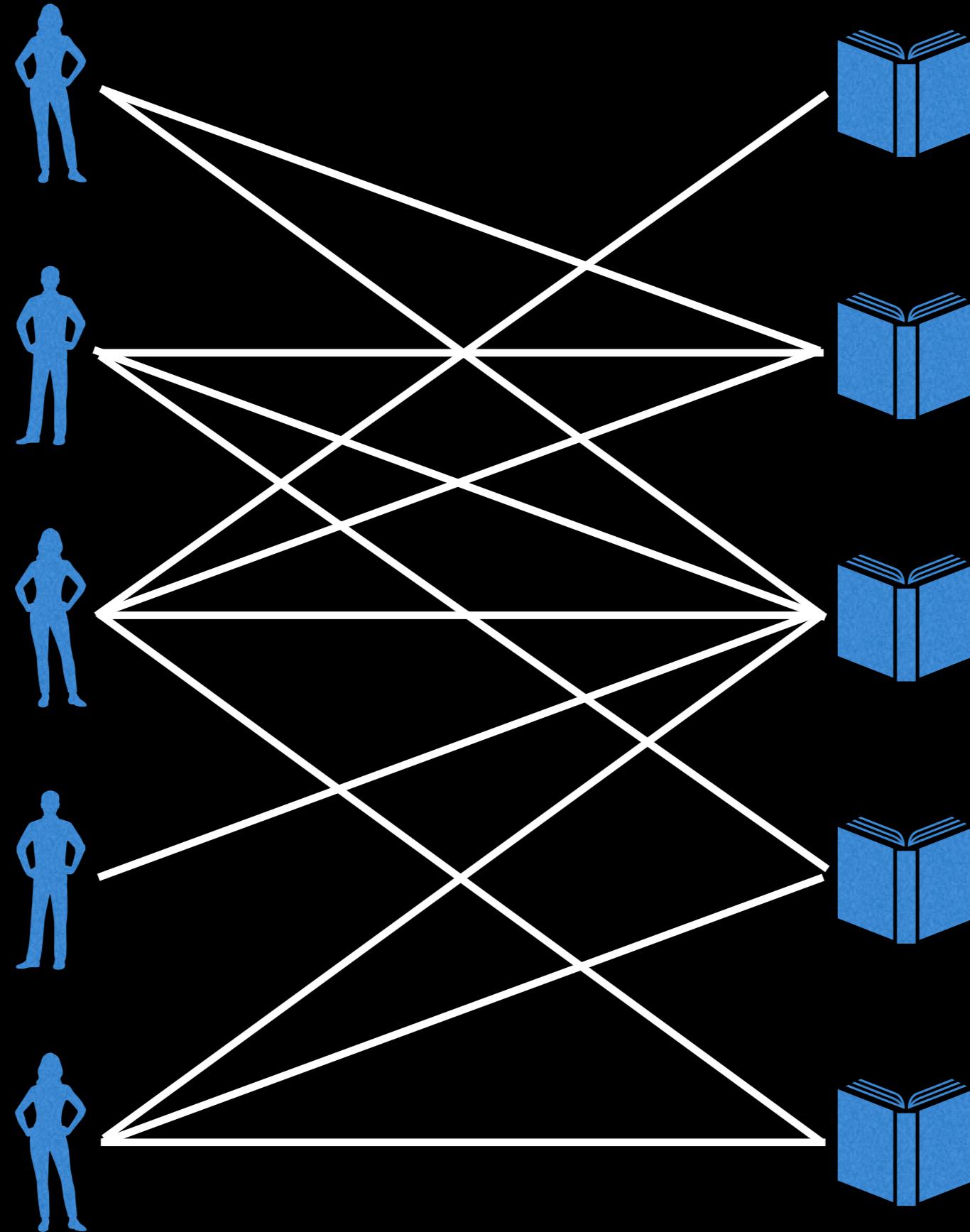


Reference: Competitive Programming 3 The New Lower Bound of Programming Contests. P 349

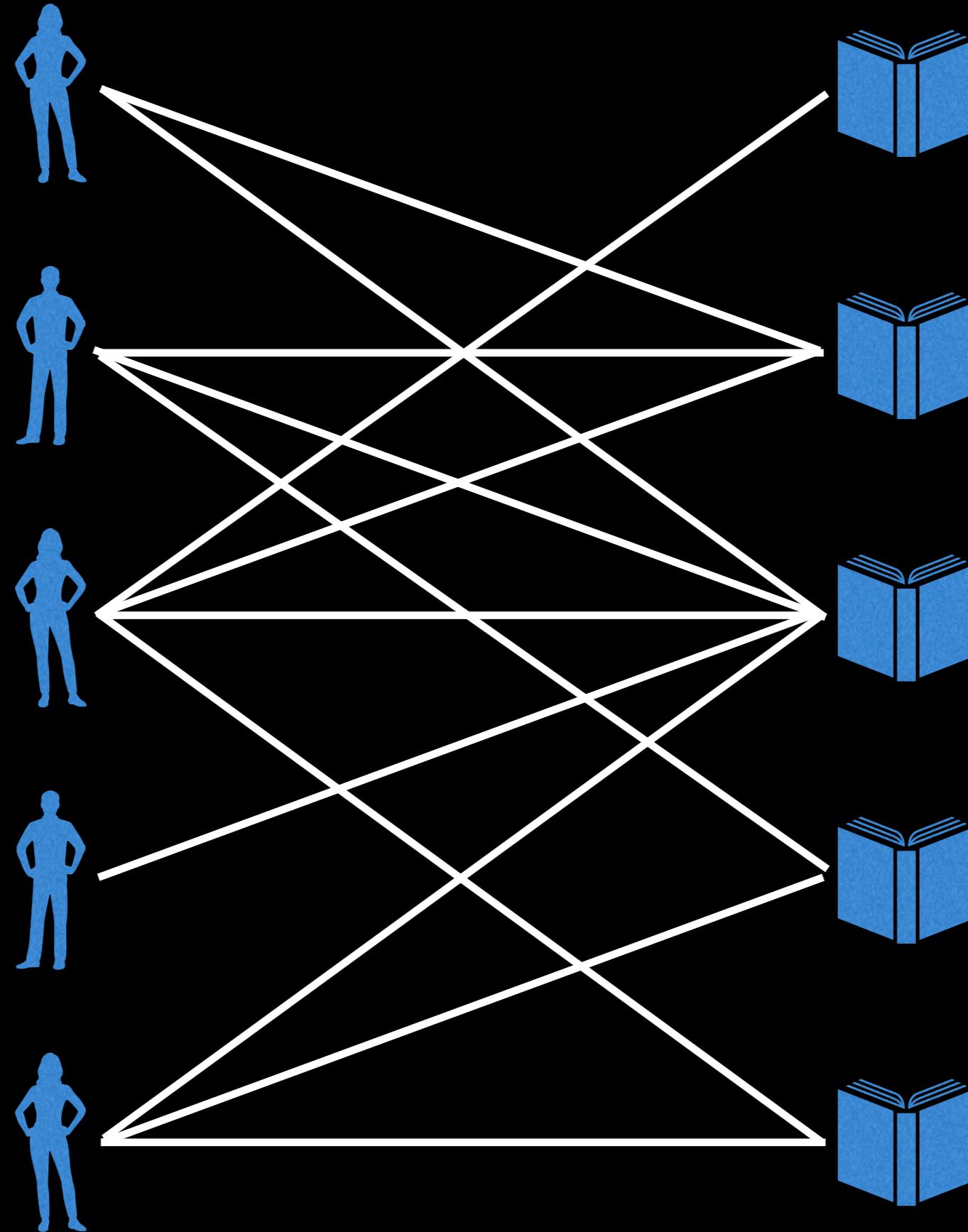
Suppose there are 5 people + 5 books and some people express interest in some books, who gets what book?



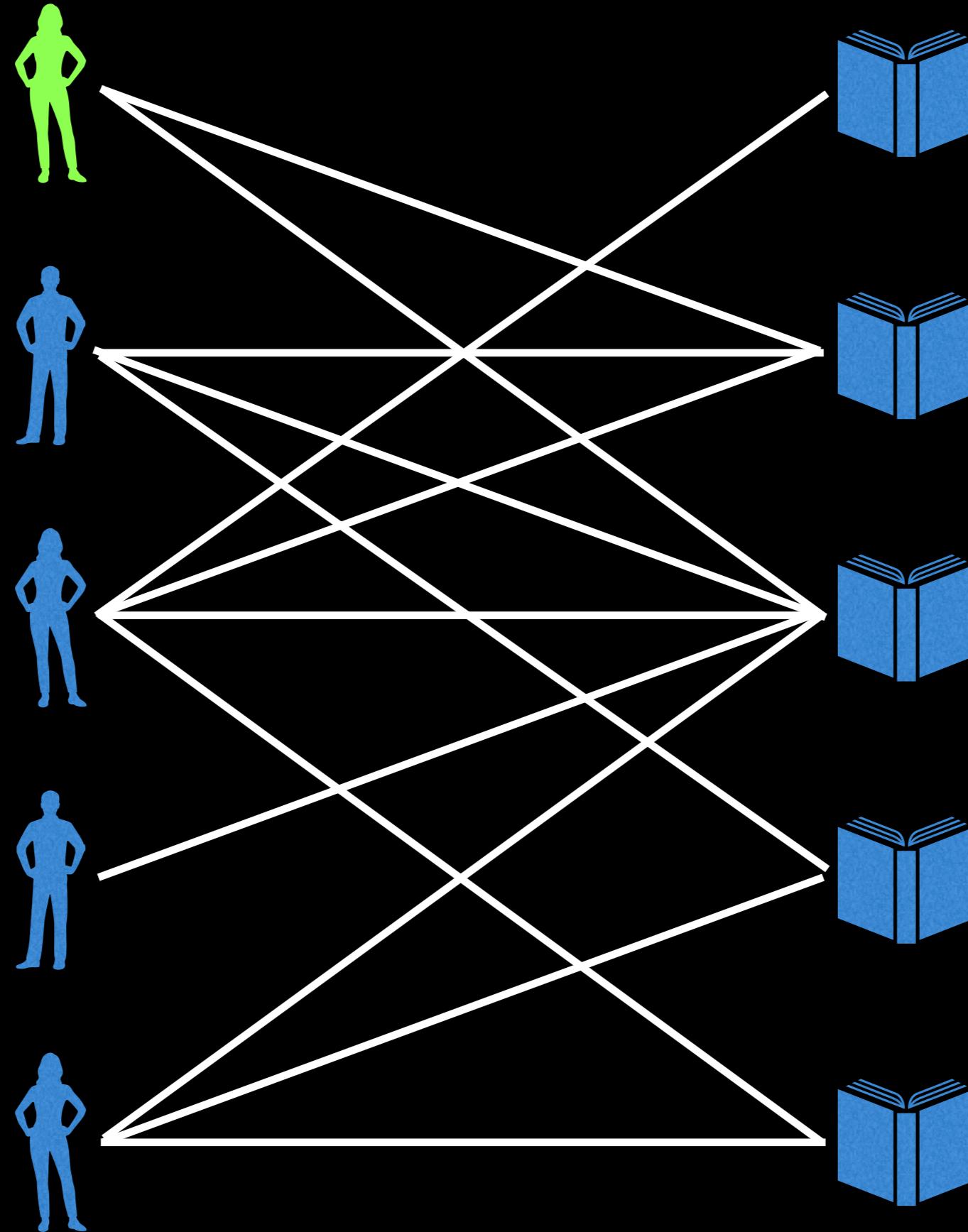
Suppose there are 5 people + 5 books and some people express interest in some books, who gets what book?



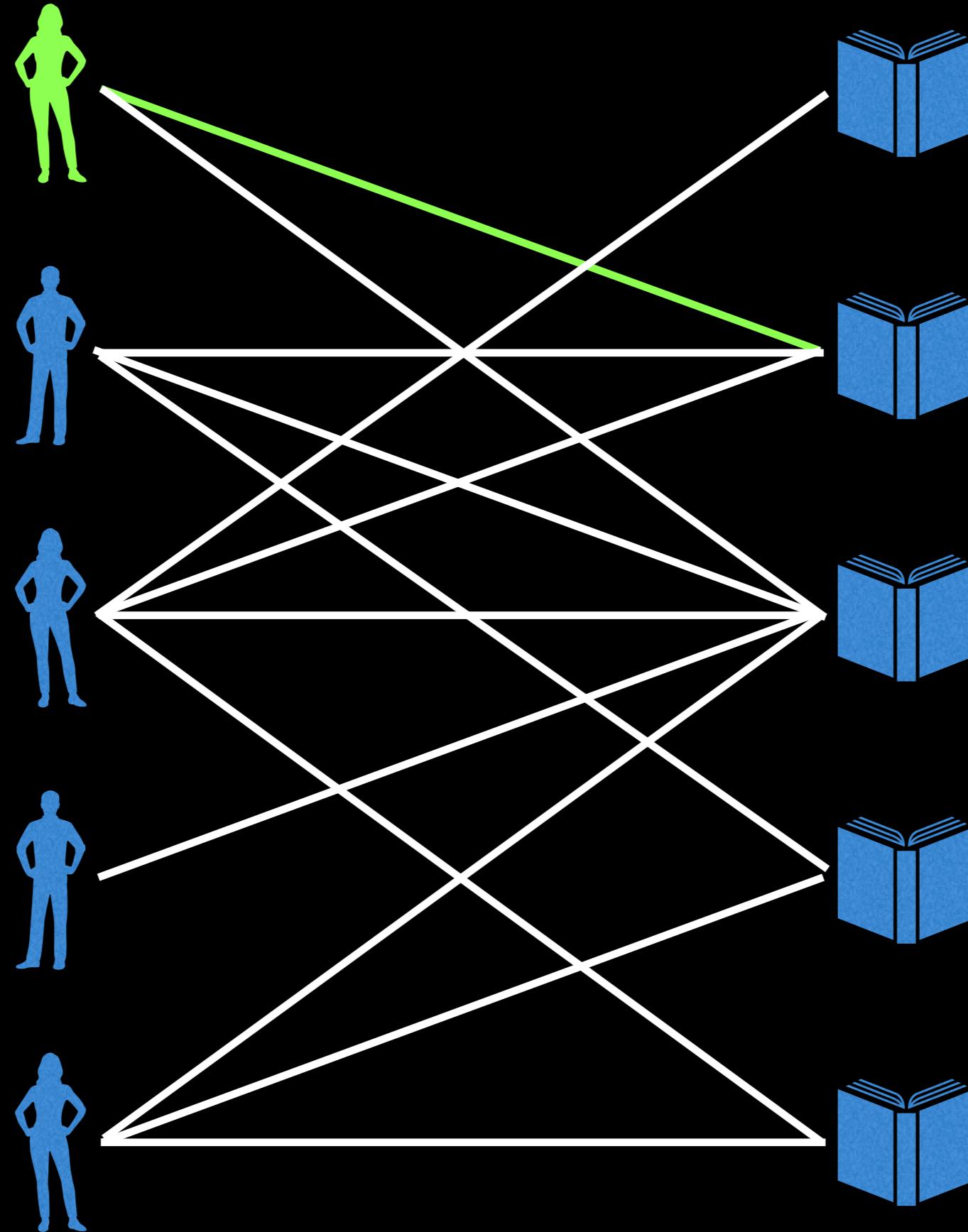
# Let's try a greedy matching solution



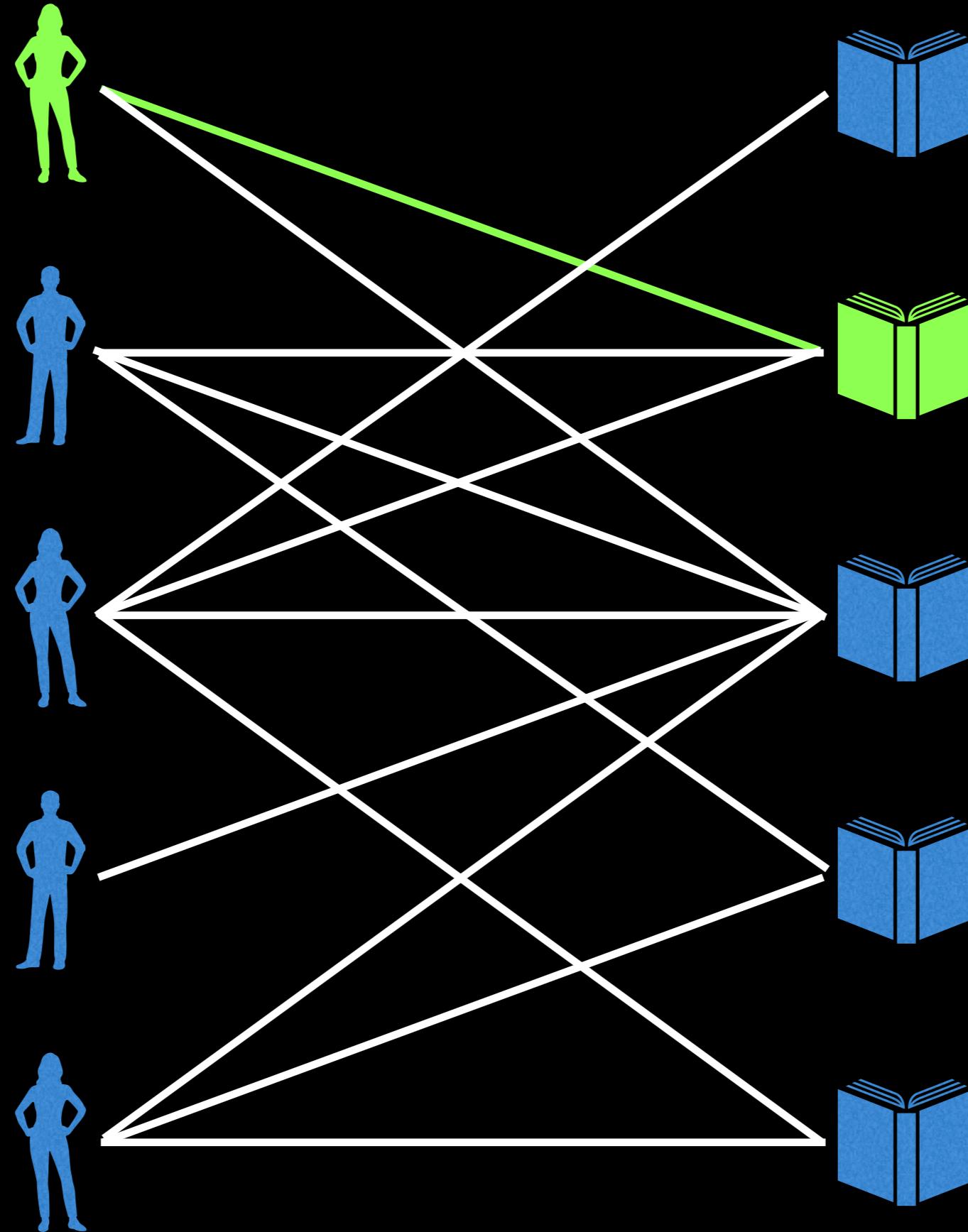
# Let's try a greedy matching solution



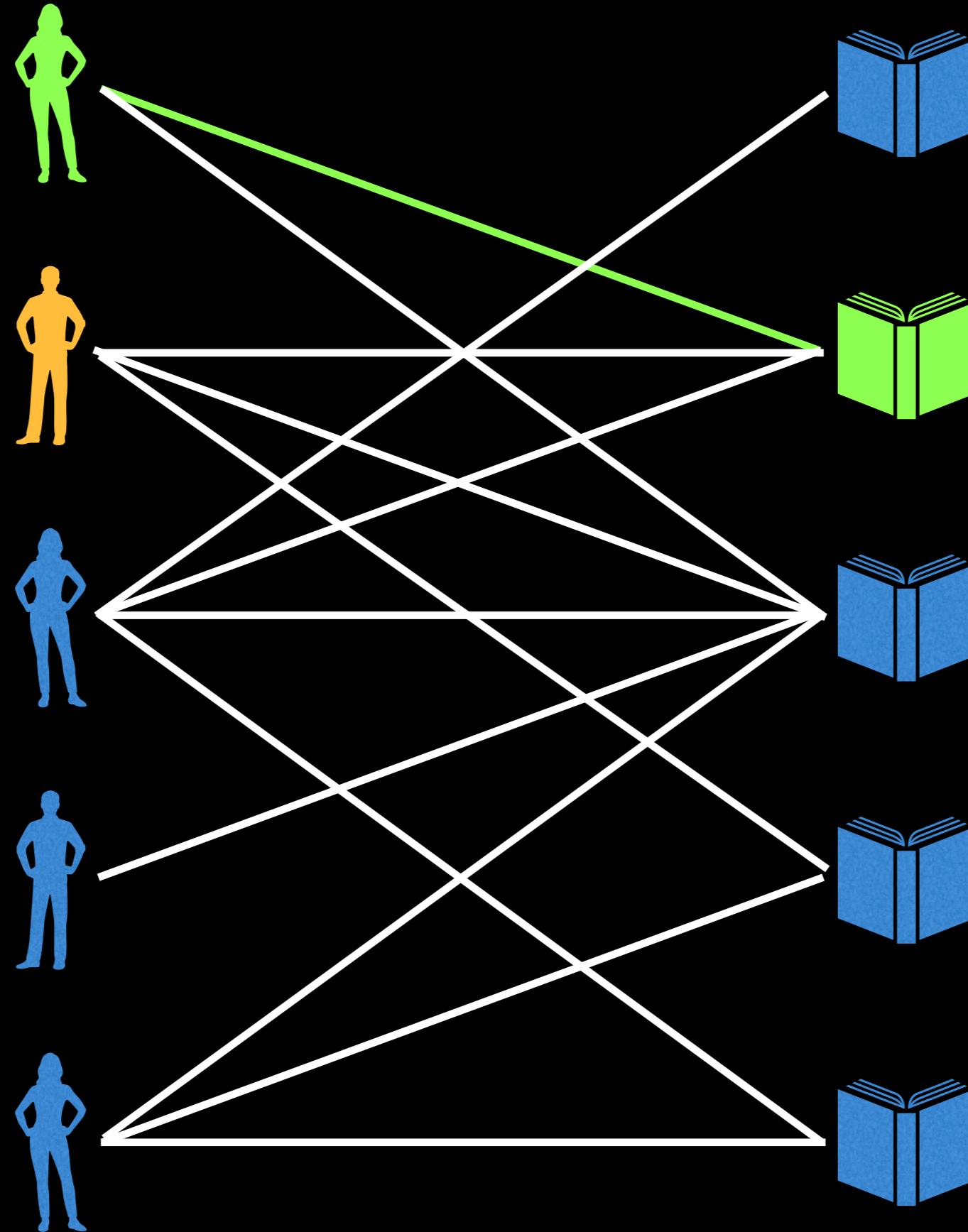
# Let's try a greedy matching solution



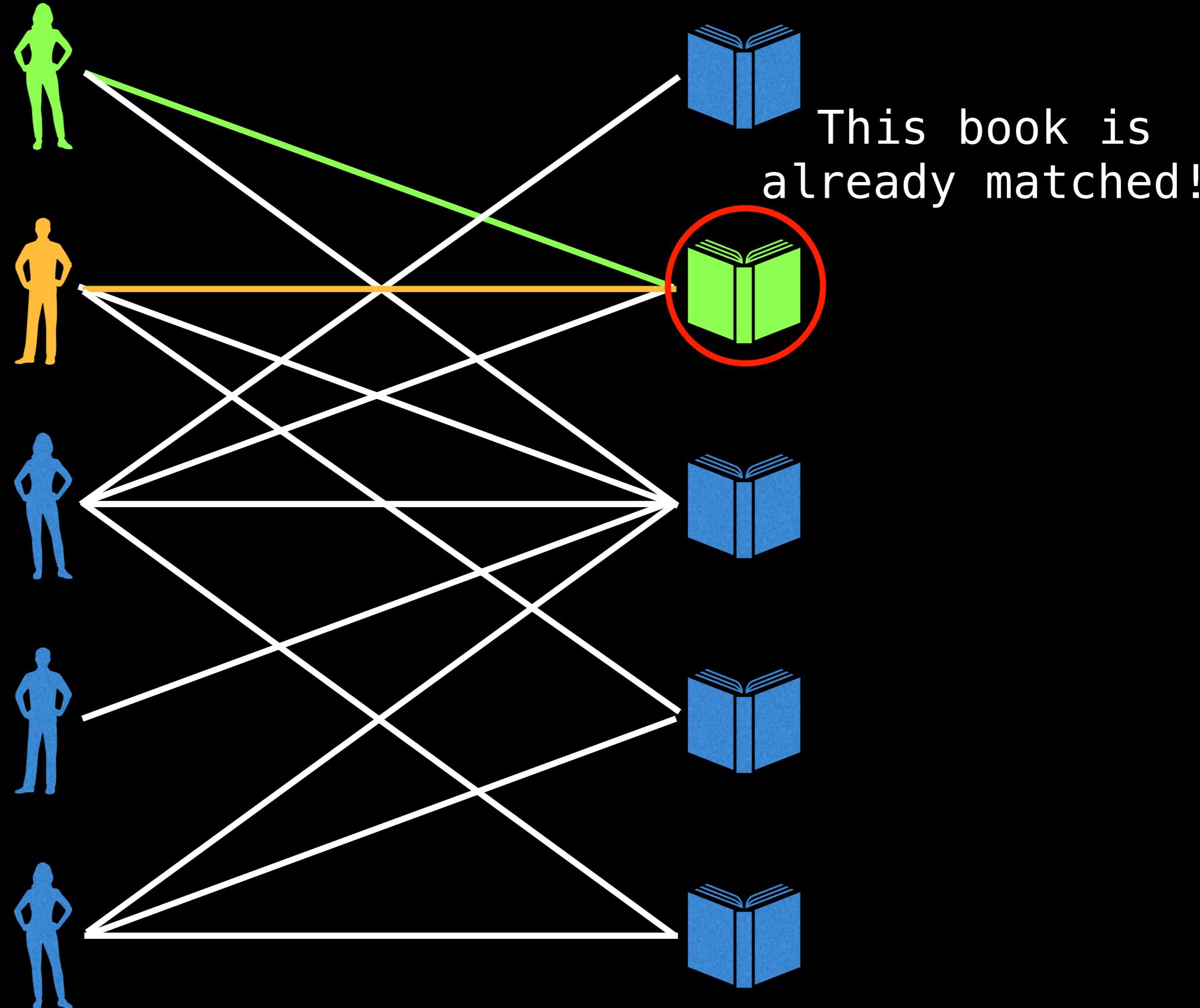
# Let's try a greedy matching solution



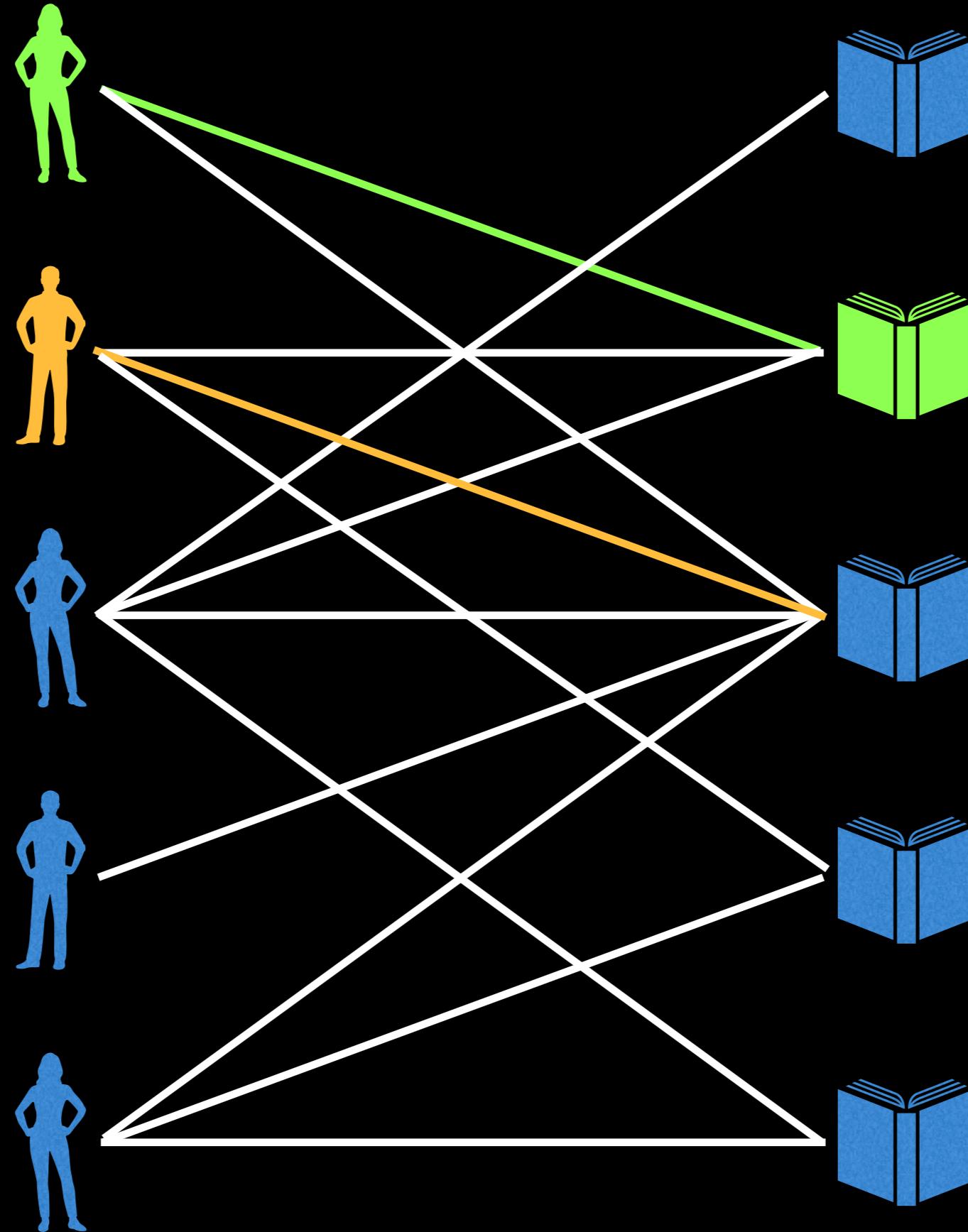
# Let's try a greedy matching solution



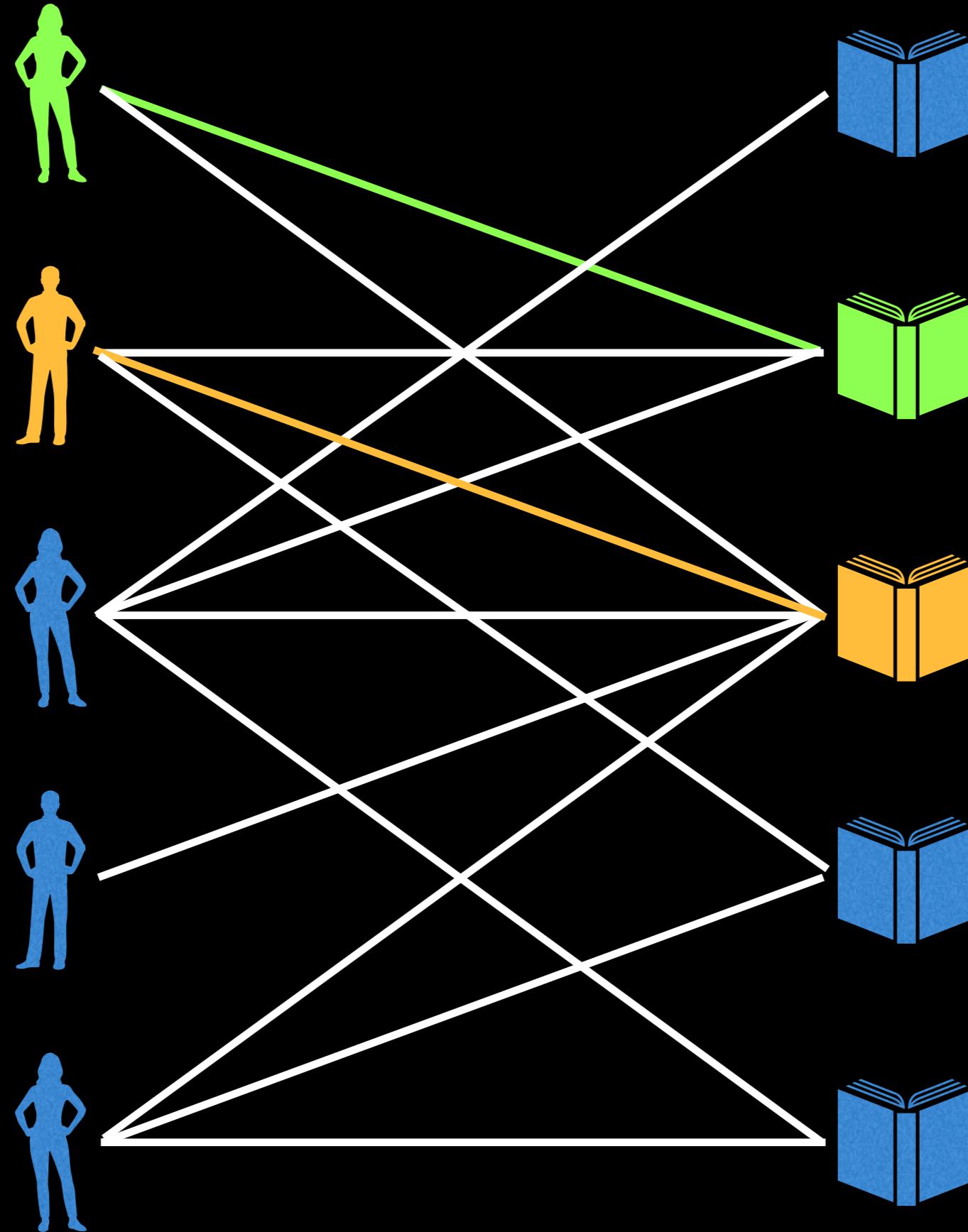
# Let's try a greedy matching solution



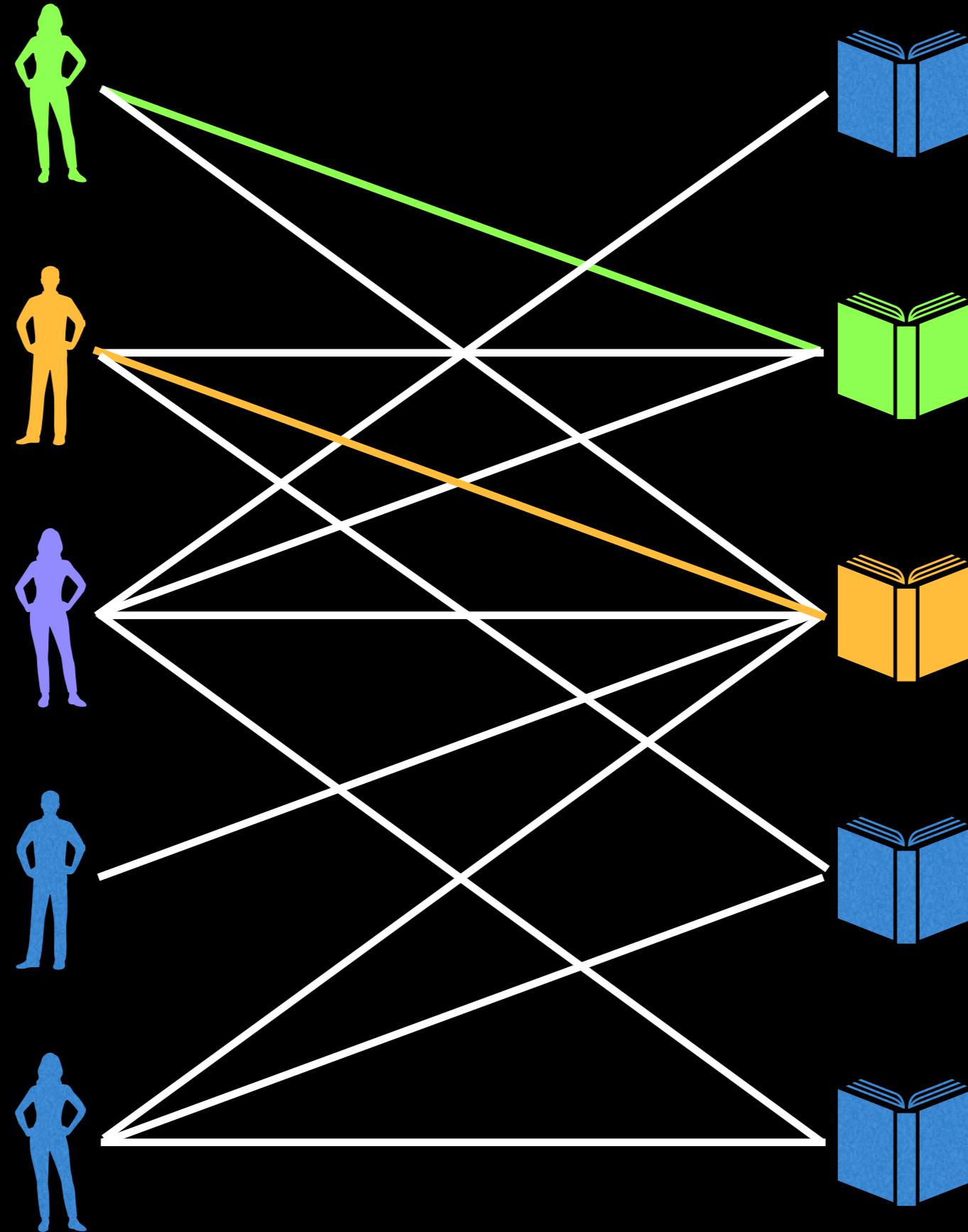
# Let's try a greedy matching solution



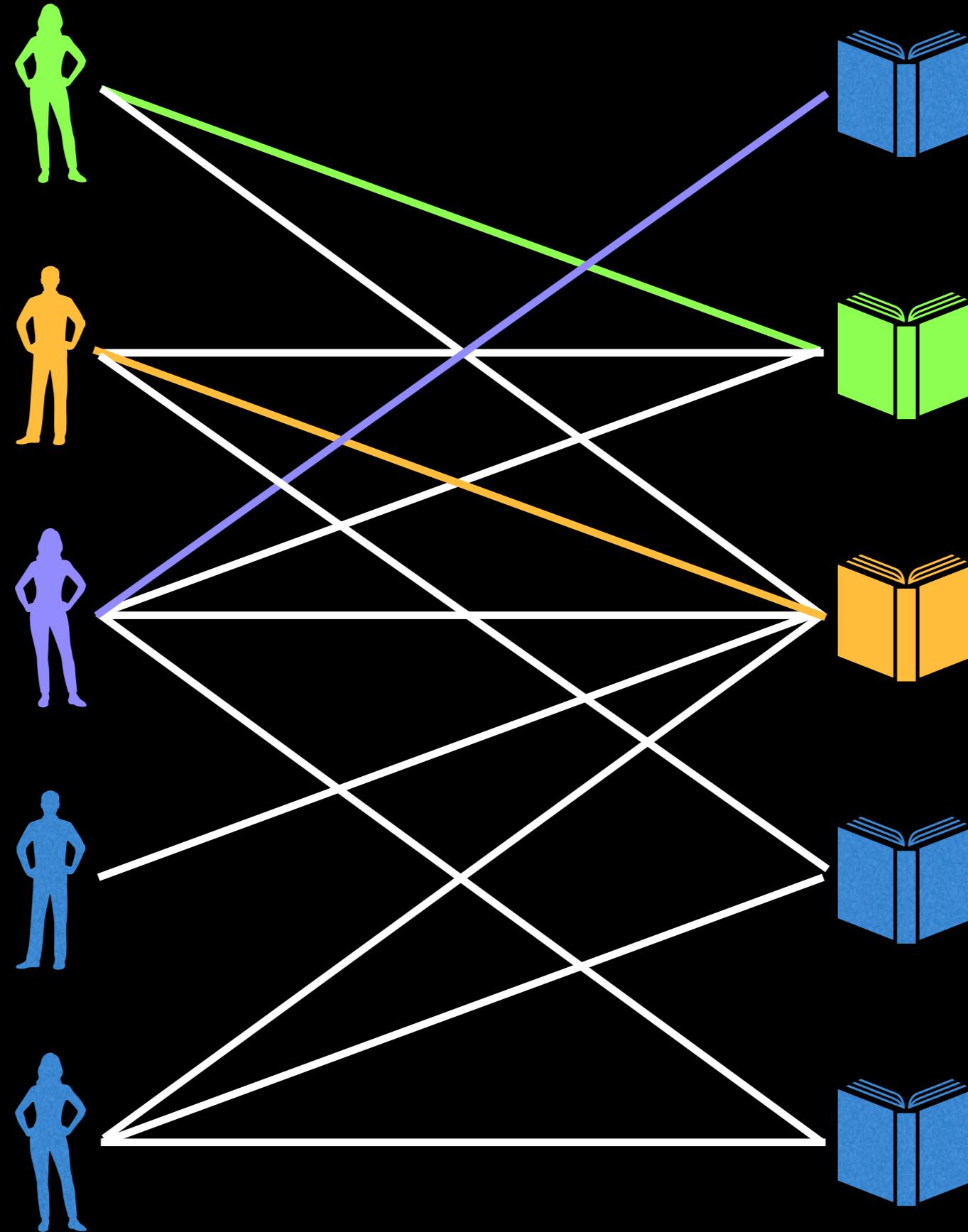
# Let's try a greedy matching solution



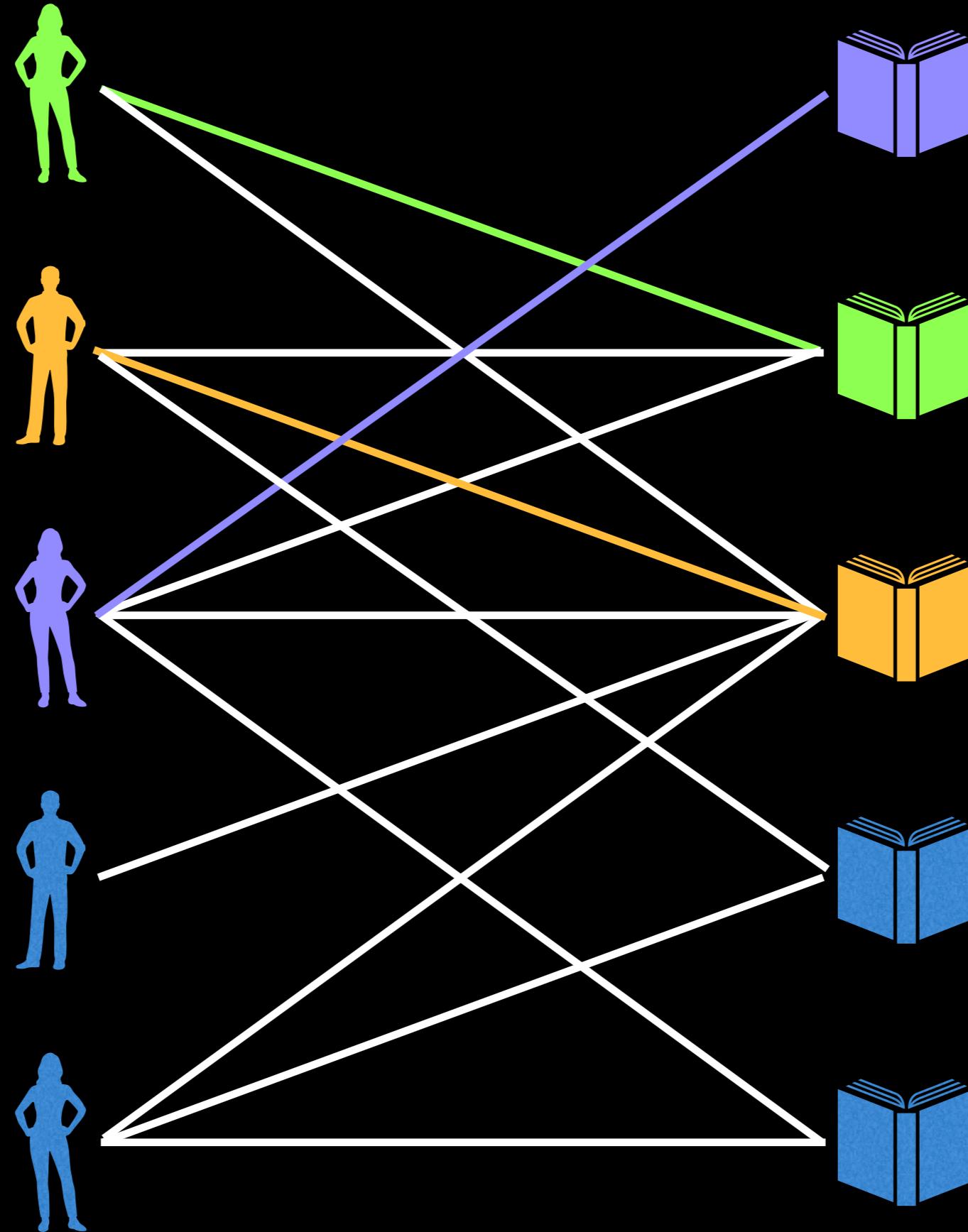
# Let's try a greedy matching solution



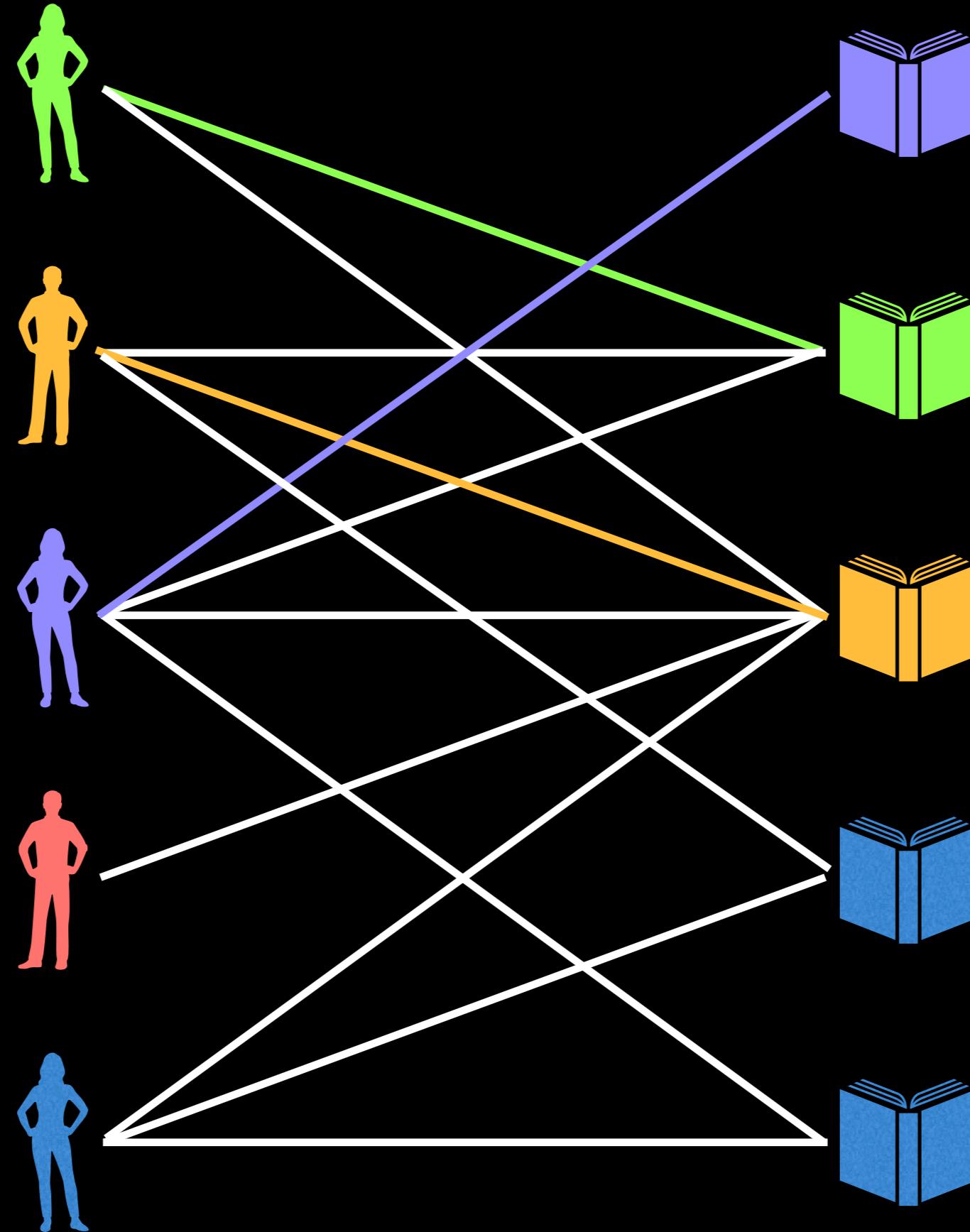
# Let's try a greedy matching solution



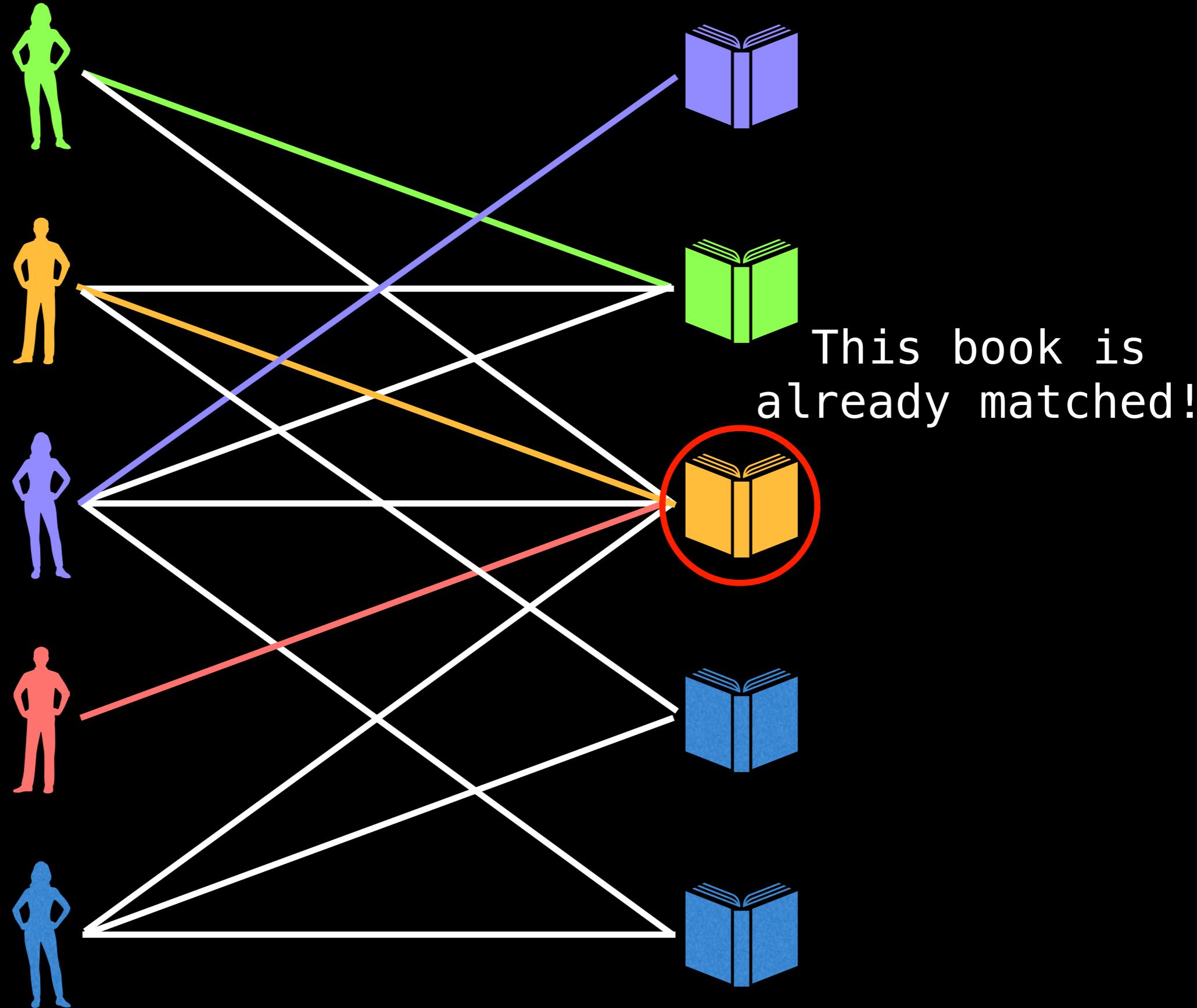
# Let's try a greedy matching solution



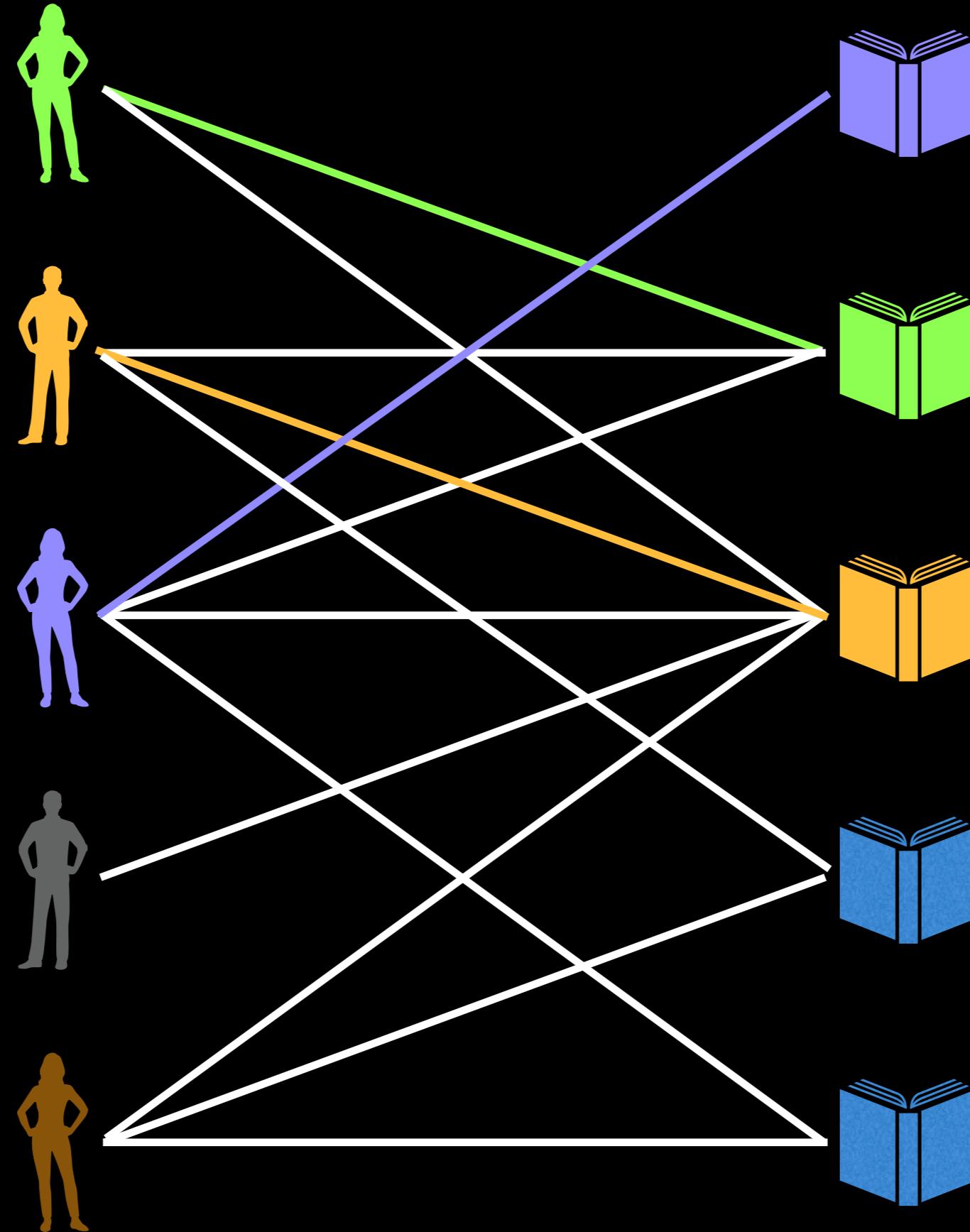
# Let's try a greedy matching solution



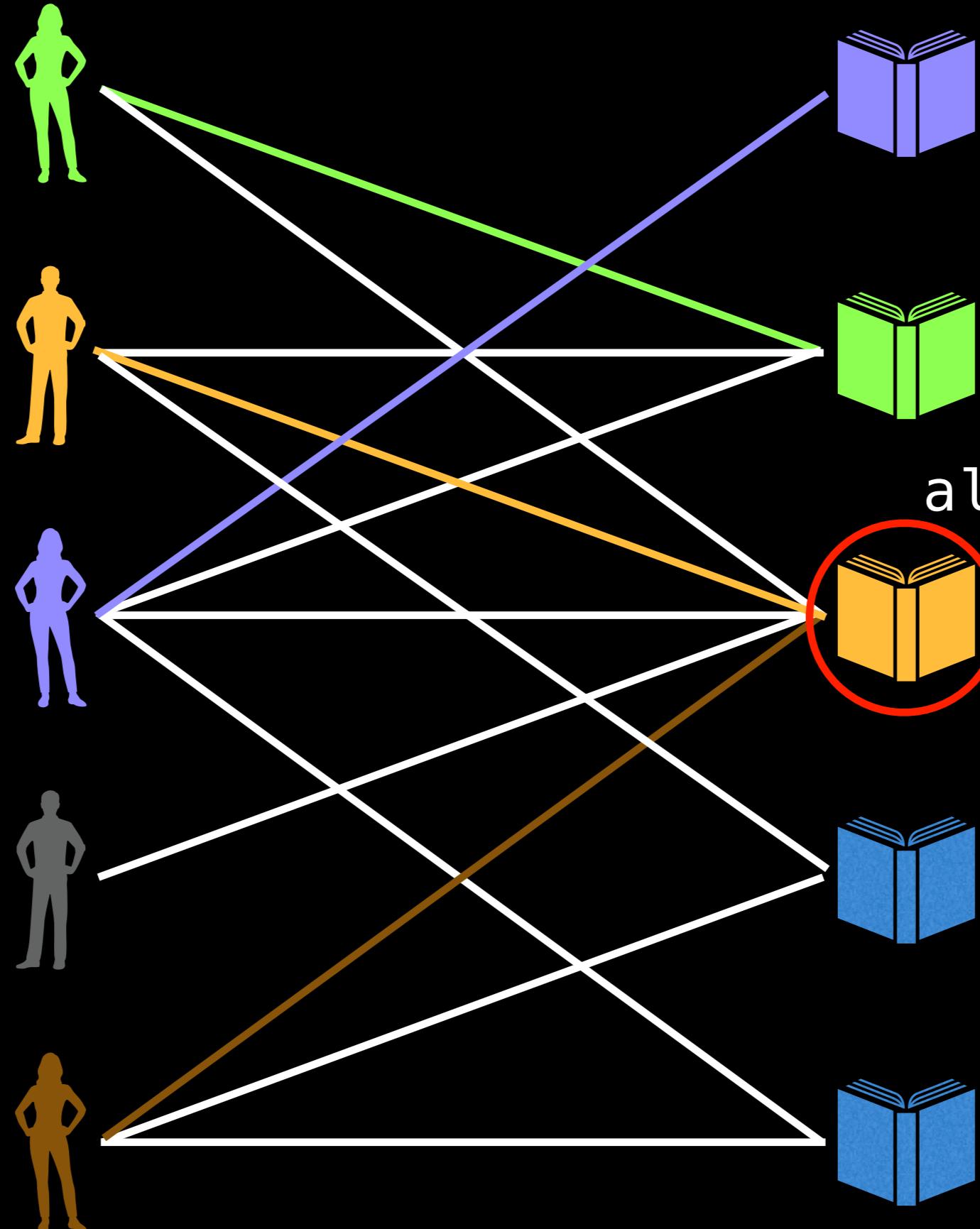
# Let's try a greedy matching solution



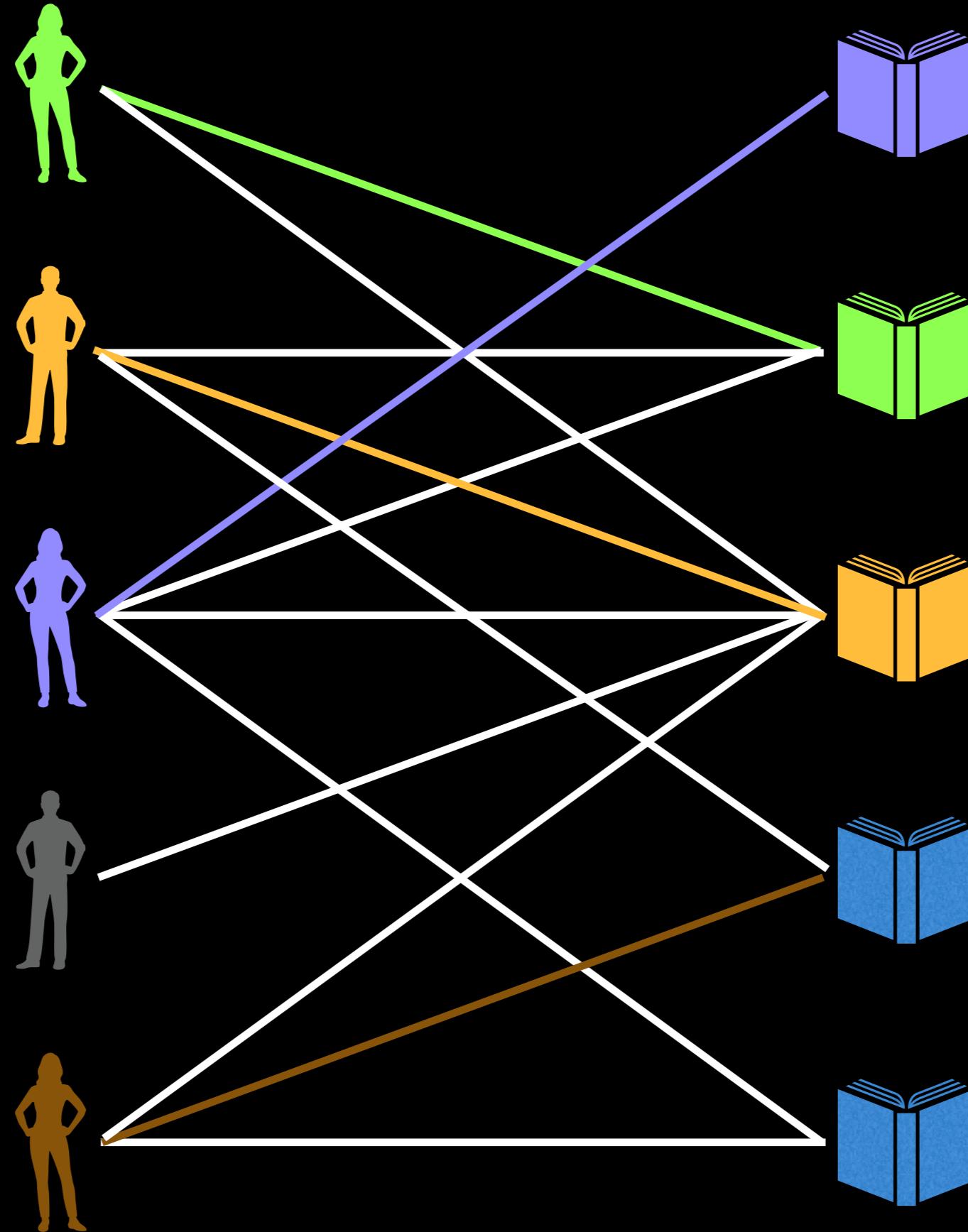
# Let's try a greedy matching solution



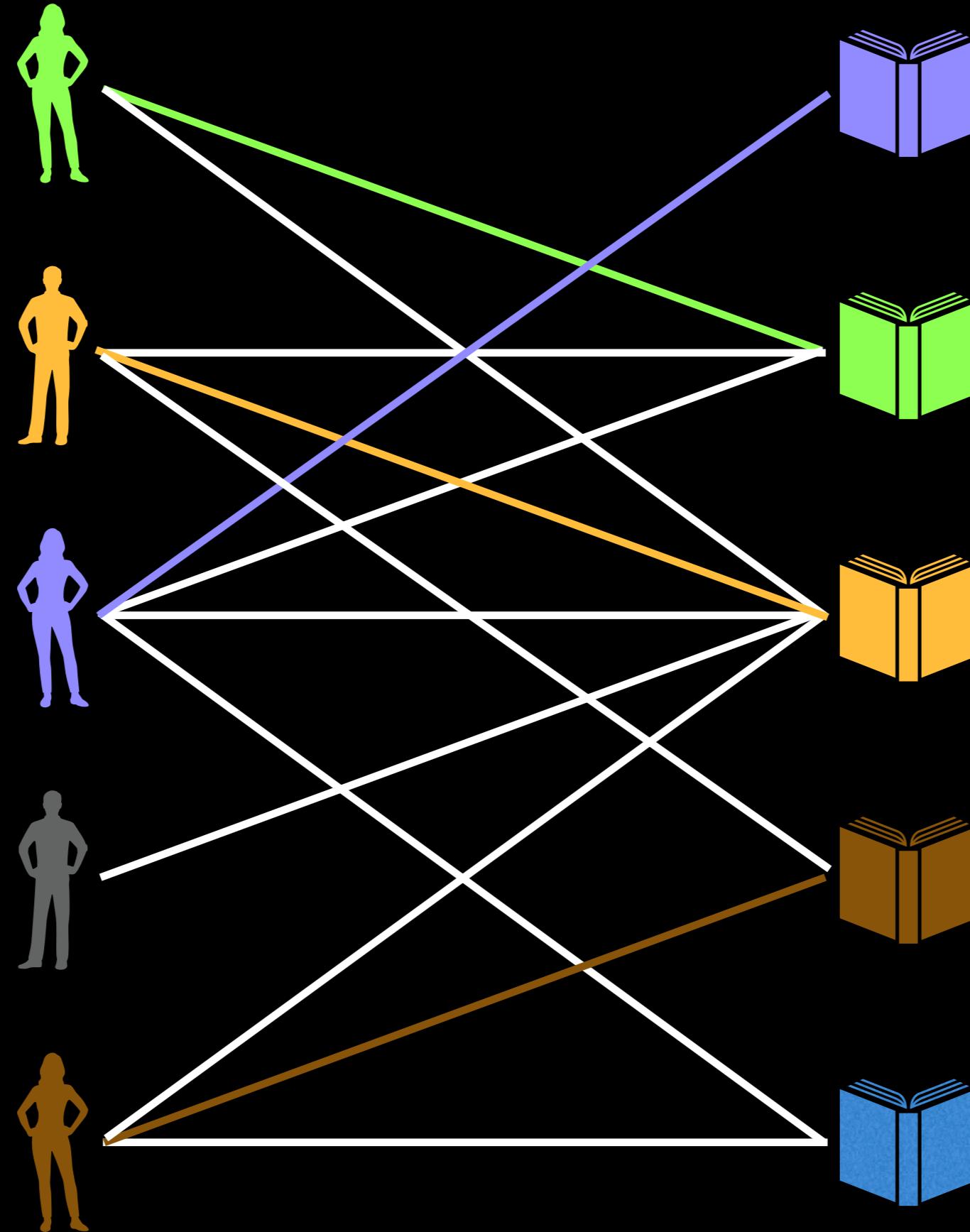
# Let's try a greedy matching solution



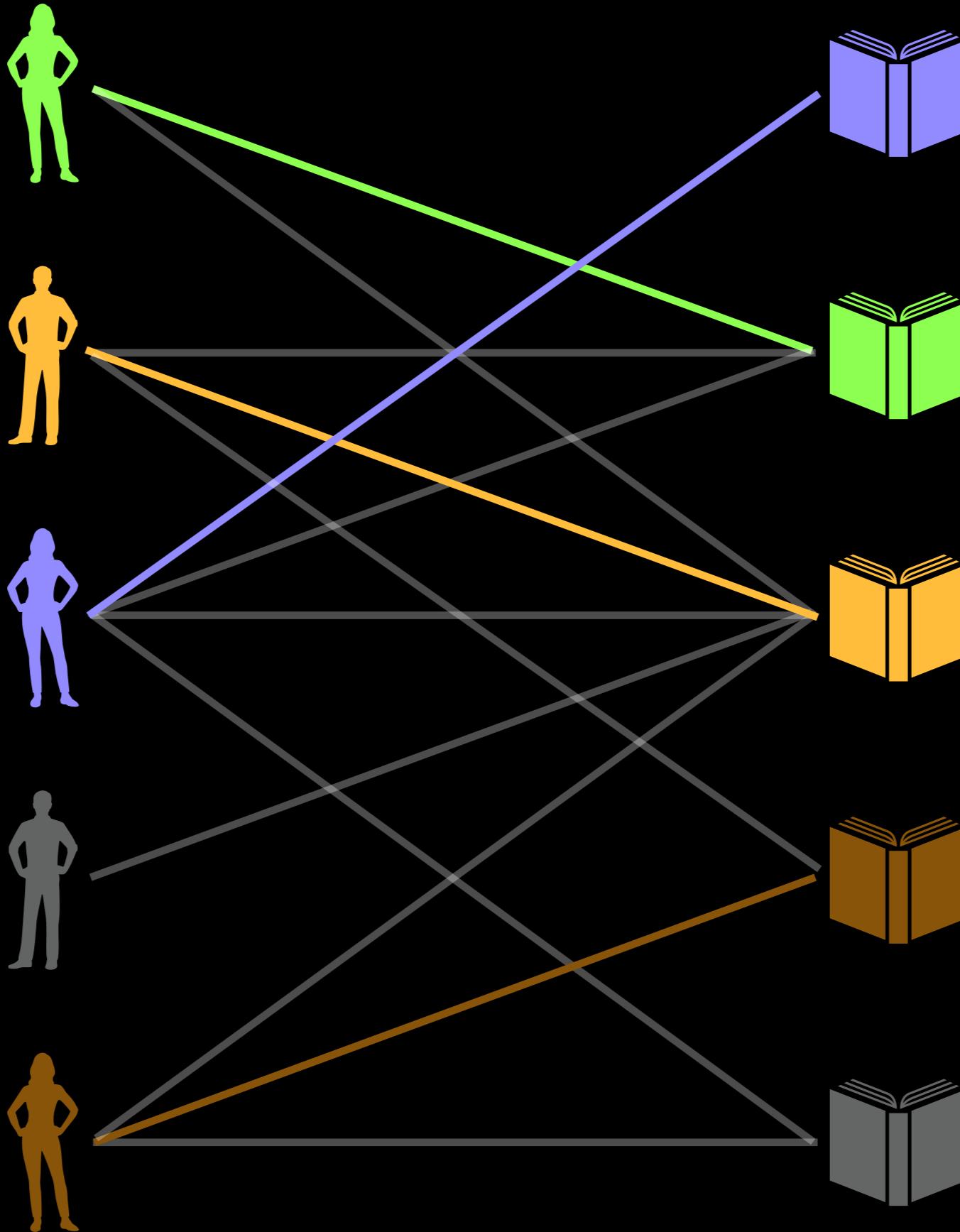
# Let's try a greedy matching solution



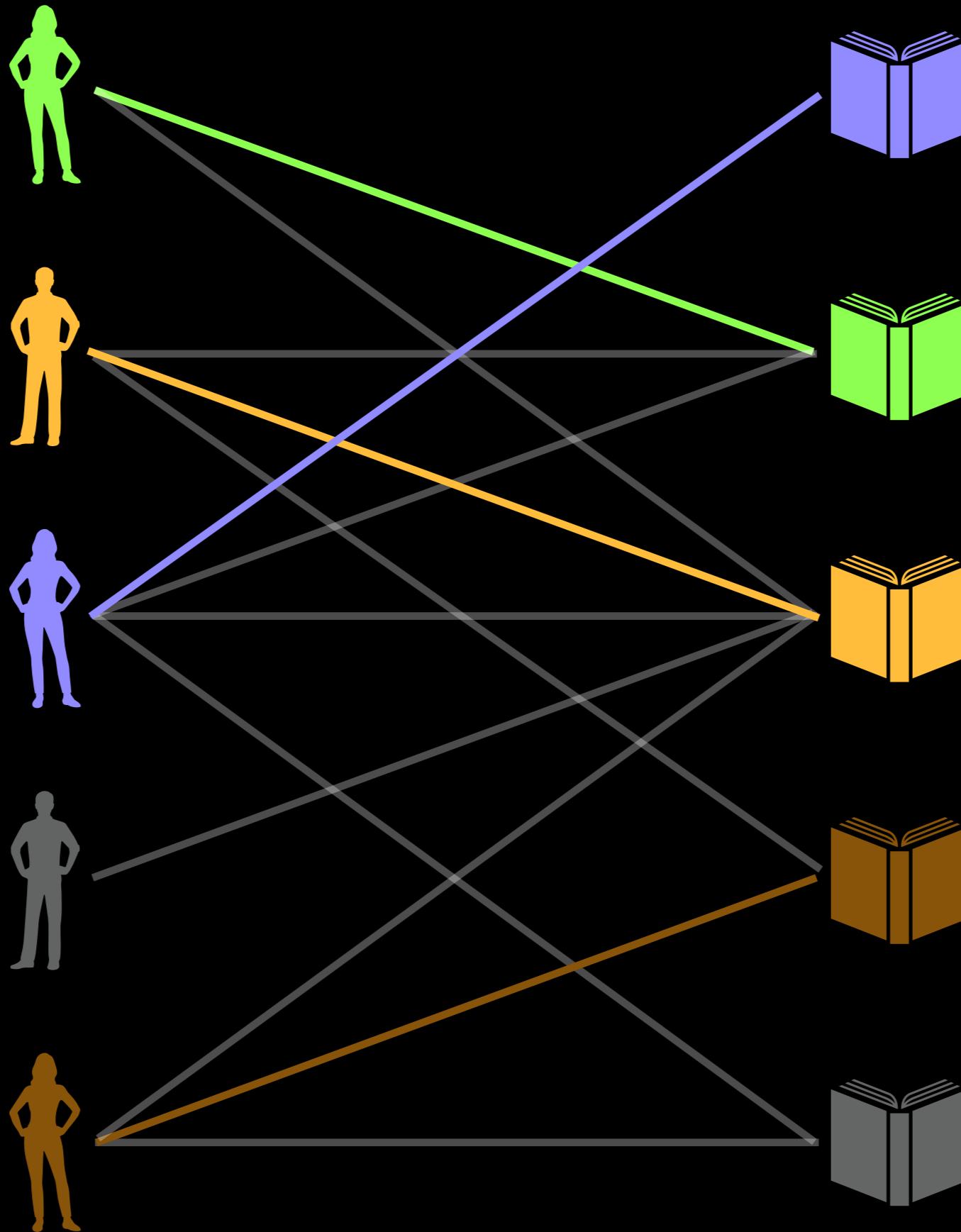
# Let's try a greedy matching solution



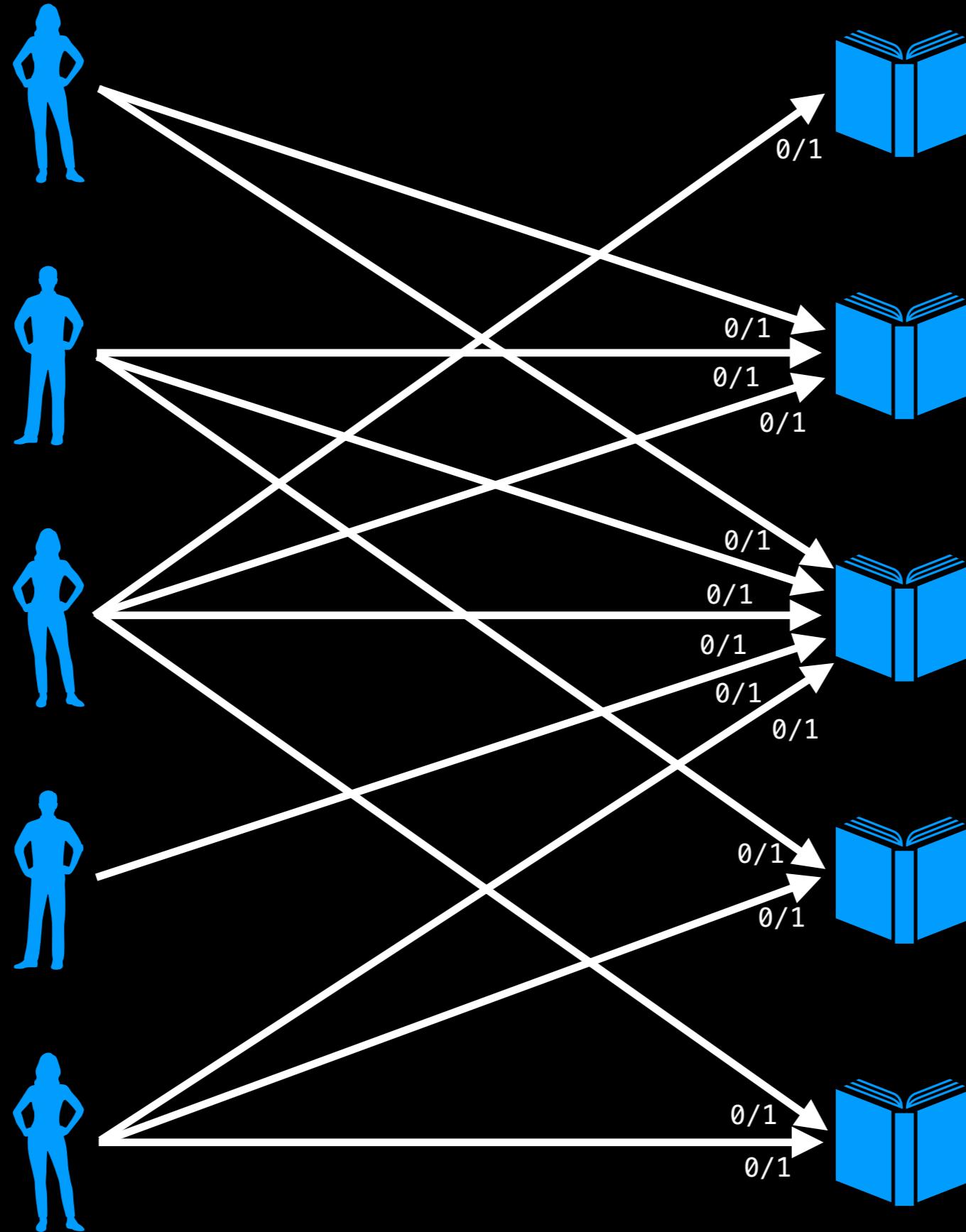
The greedy approach found 4 matchings,  
but can we do better?



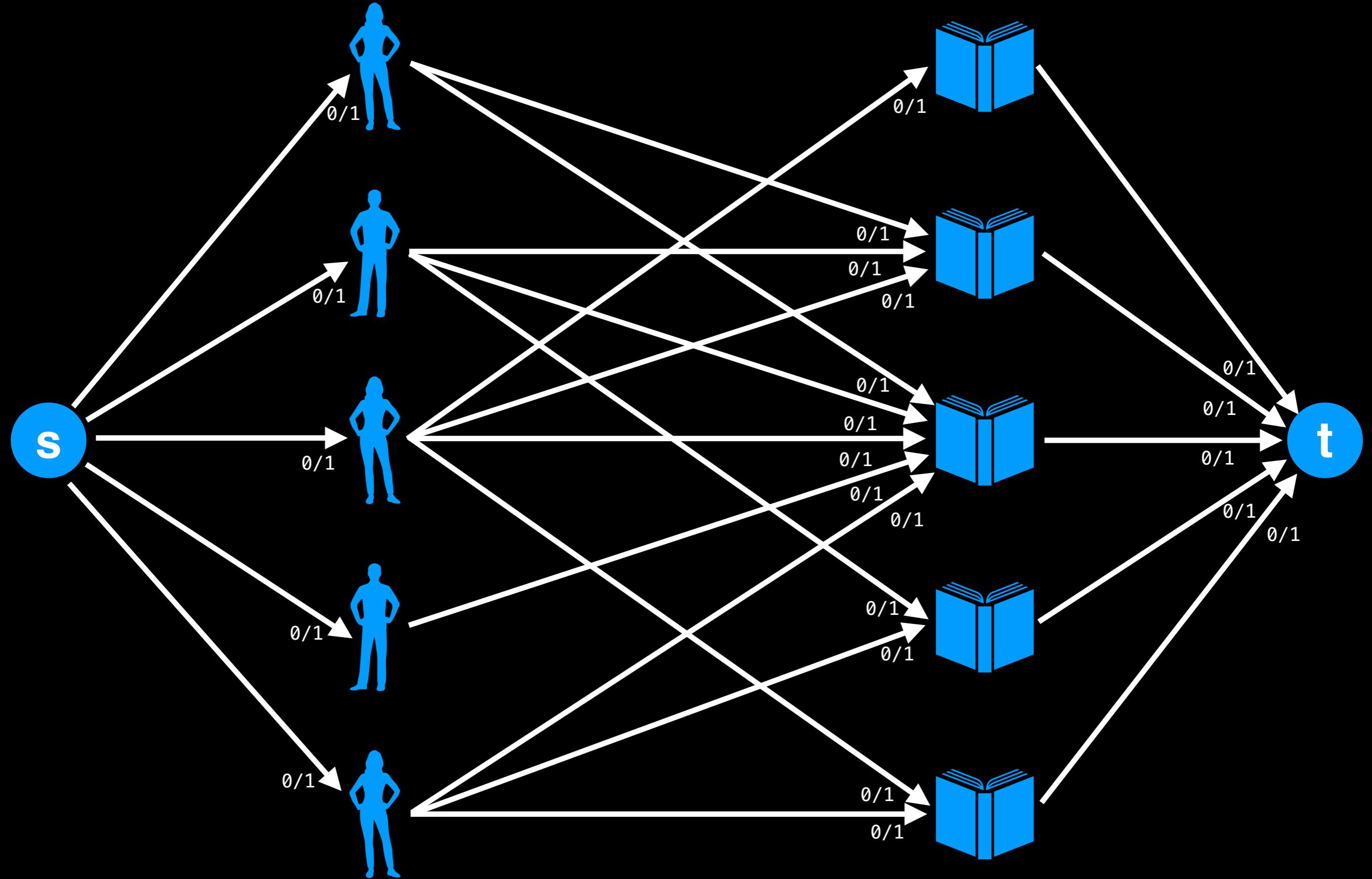
Yes! We can turn this matching problem into a network flow problem by adding a sink  $s$  and a source  $t$  and making each edge have unit capacities.



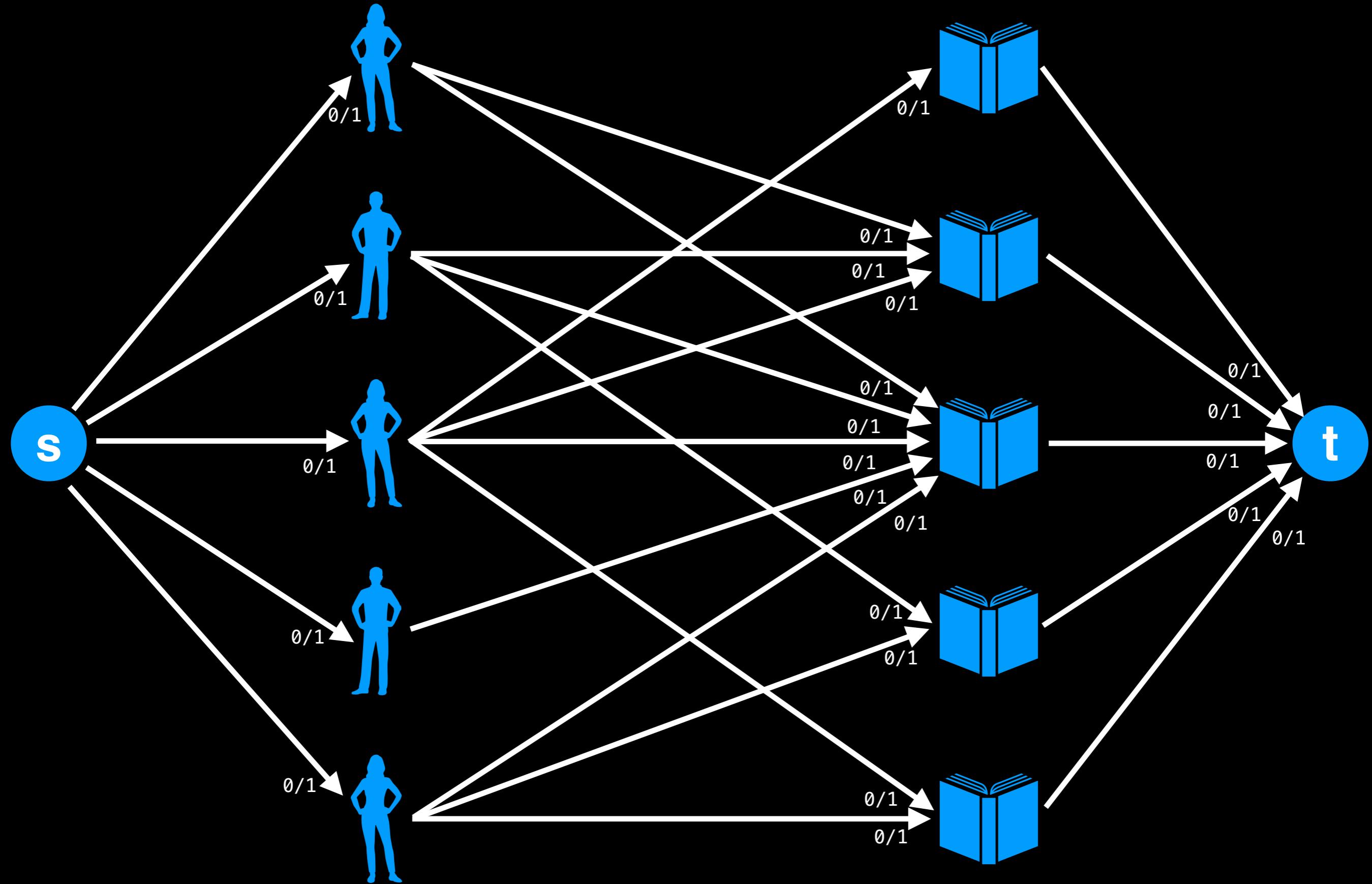
First make the edges directed and add unit capacities to each edge. The 0/1 besides each edge means 0 flow and a maximum capacity of 1.



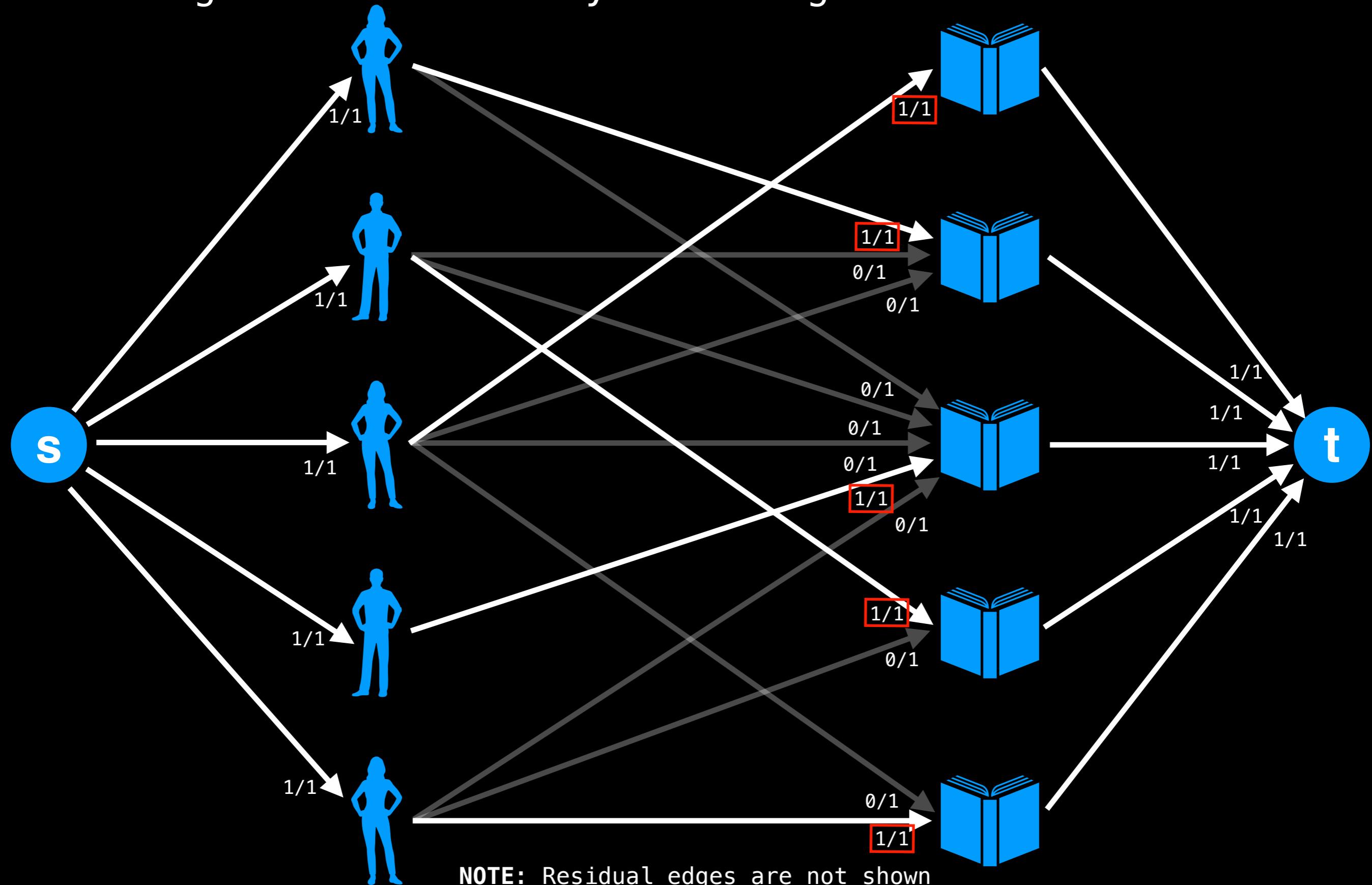
Then attach the source  $s$  and the sink  $t$  nodes allowing for one unit of capacity.



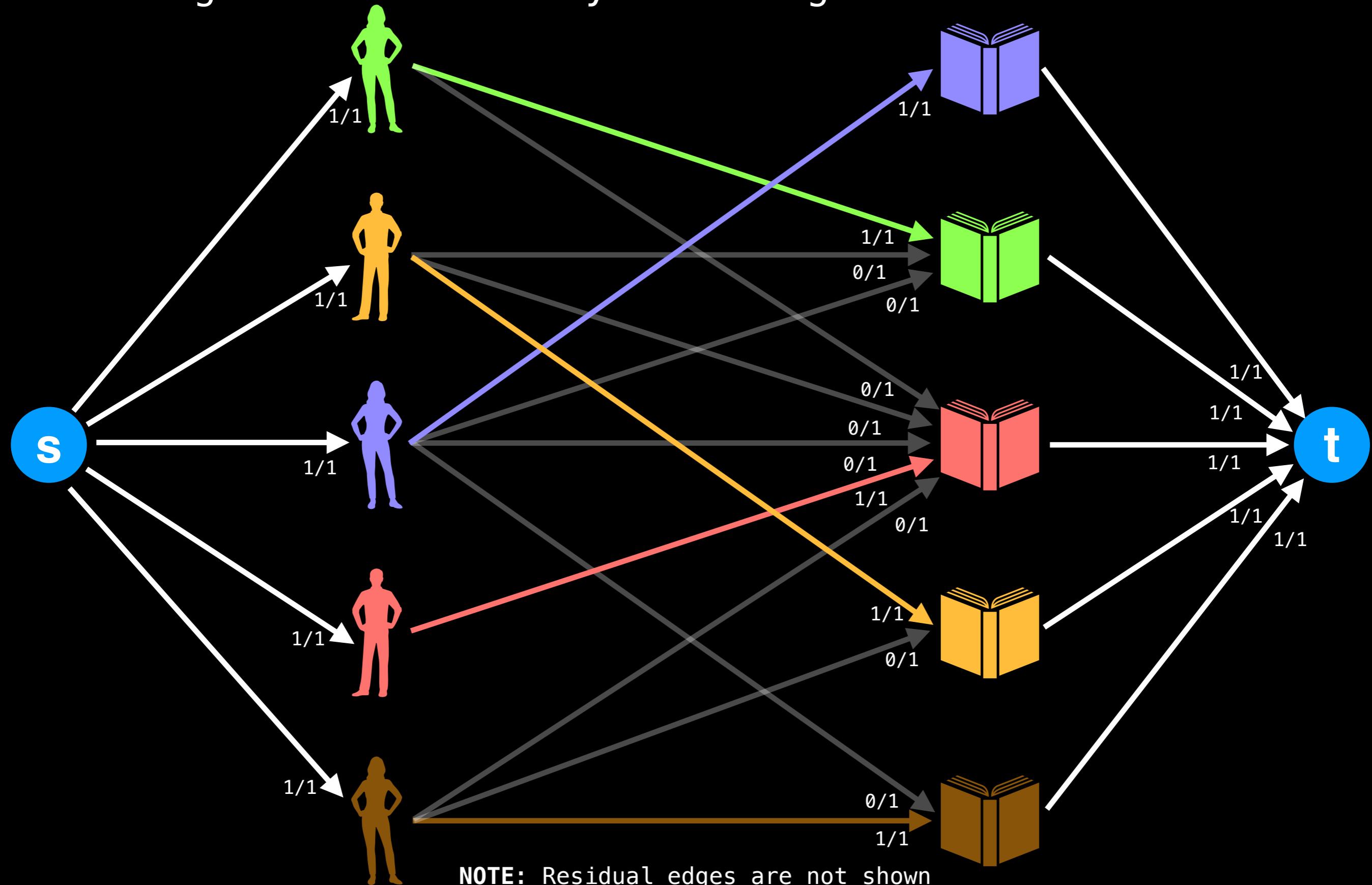
Once the flow graph is set up, use any max-flow algorithm to push flow through the network.



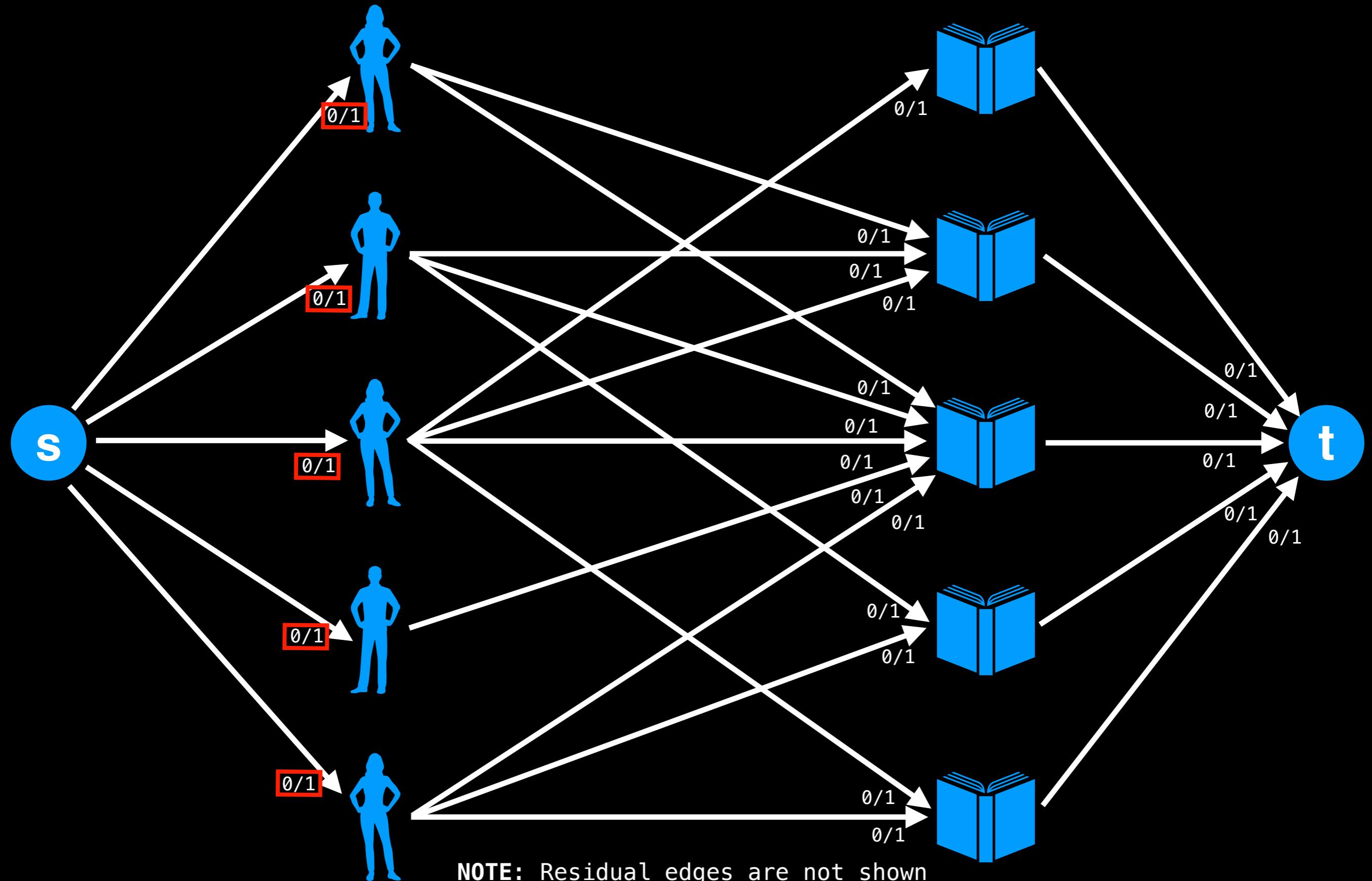
Pushing flow through the network to achieve the maximum flow changes the edge capacities. We can find which edges were taken by checking if the flow is 1



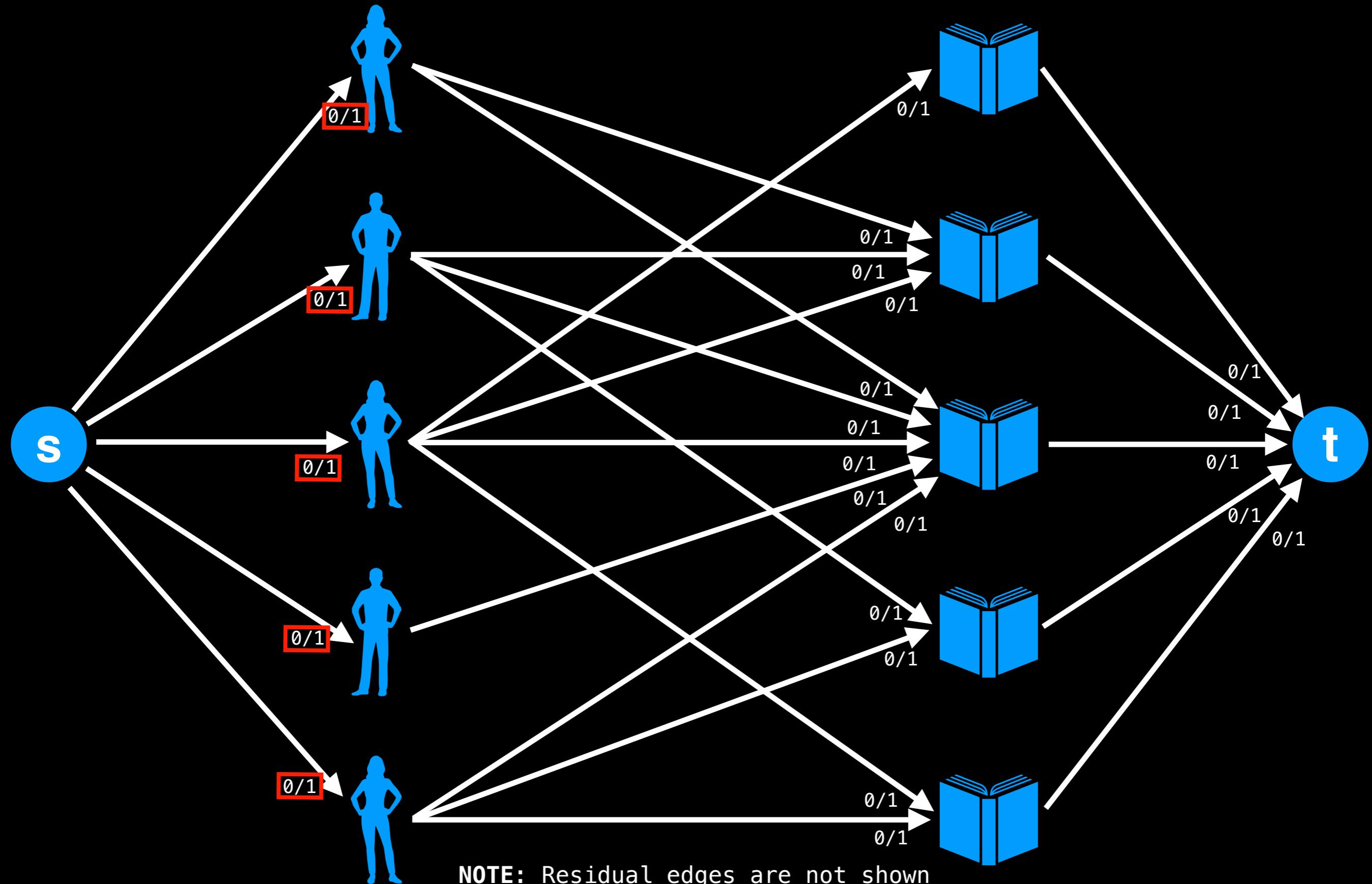
Pushing flow through the network to achieve the maximum flow changes the edge capacities. We can find which edges were taken by checking if the flow is 1



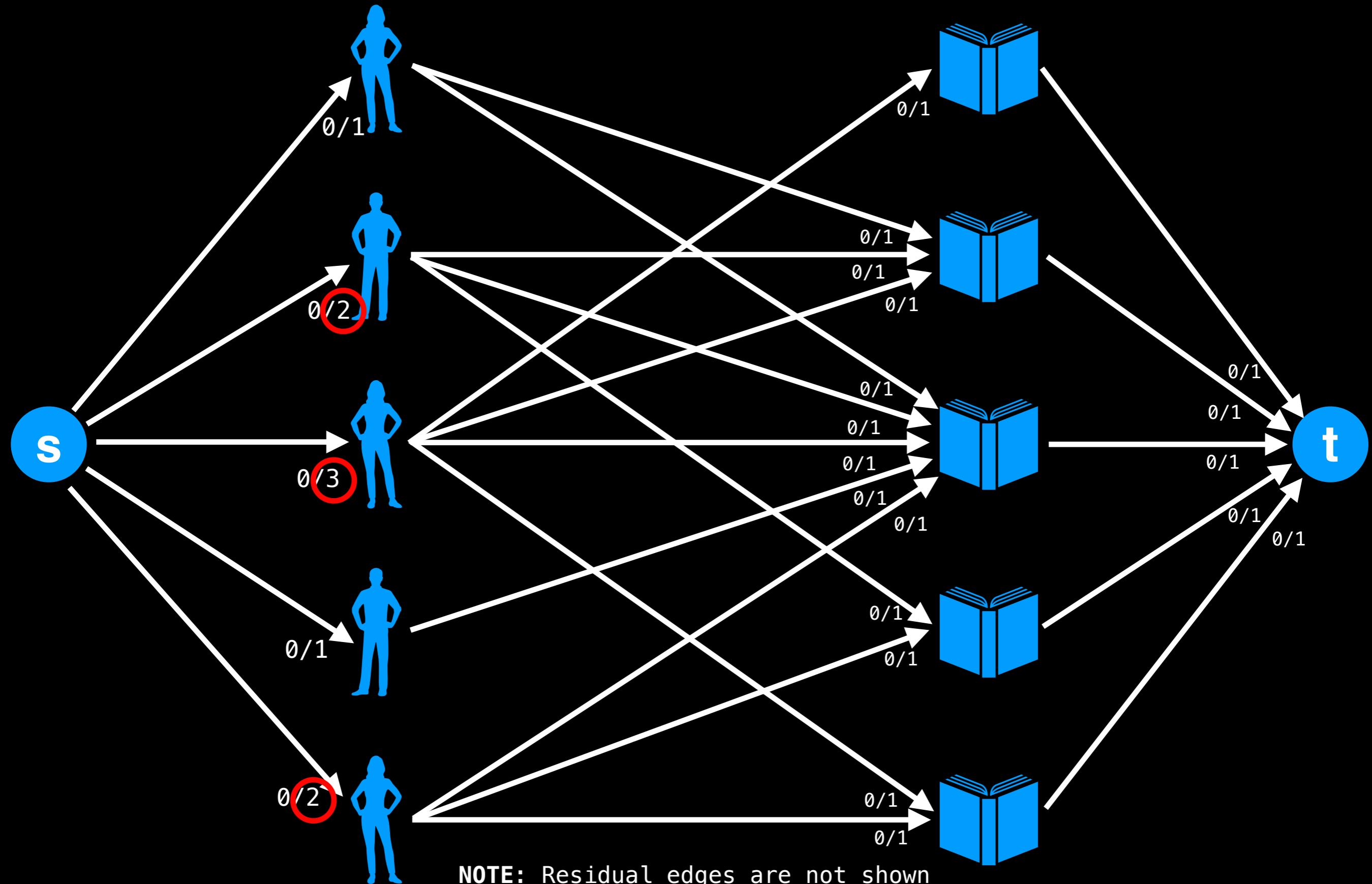
Q: We originally set the capacity of each edge from the source to each person to be 1, what constraint does this enforce?



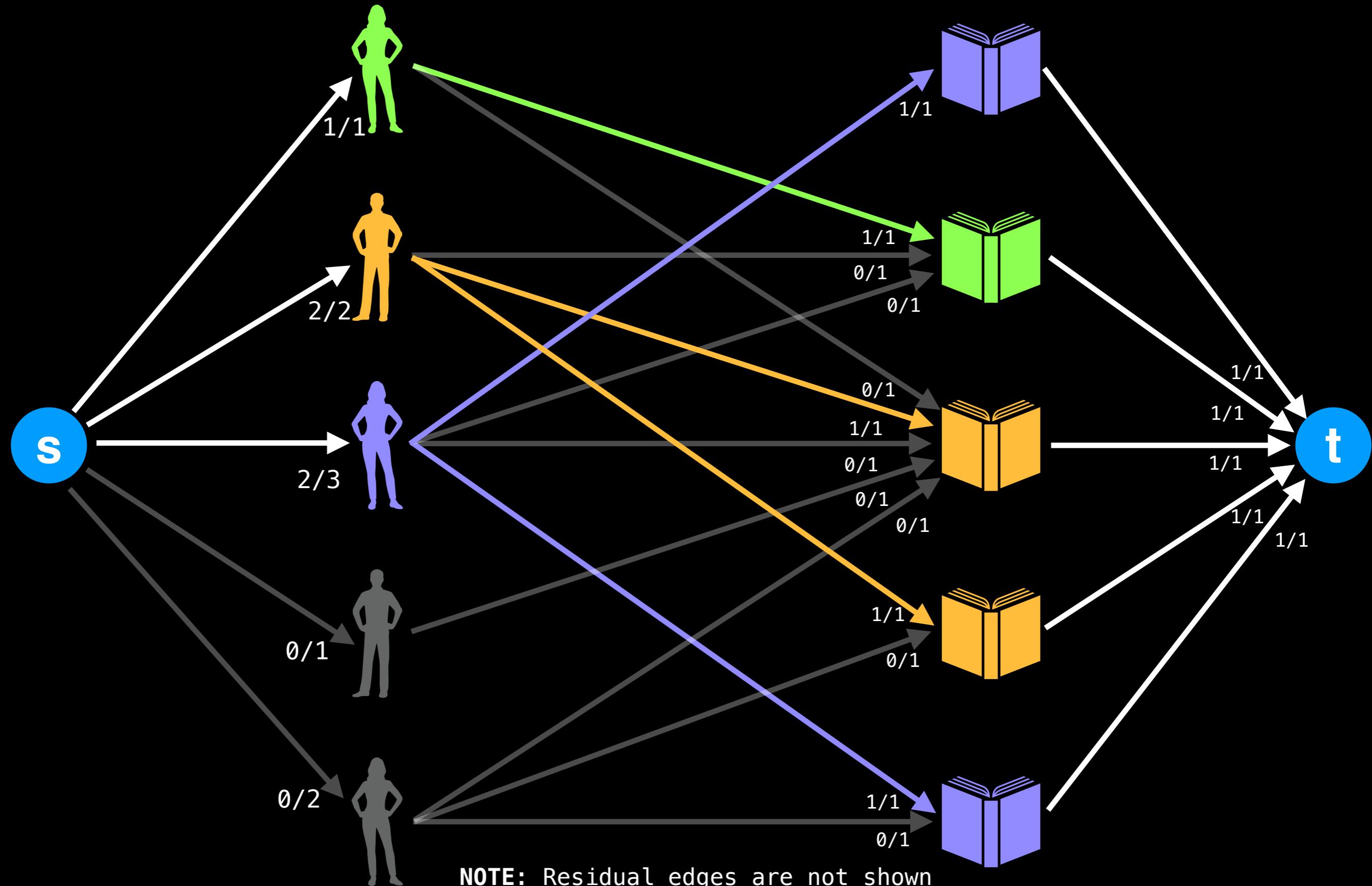
A: The capacity of 1 ensures that each person can get up to one book and no more.



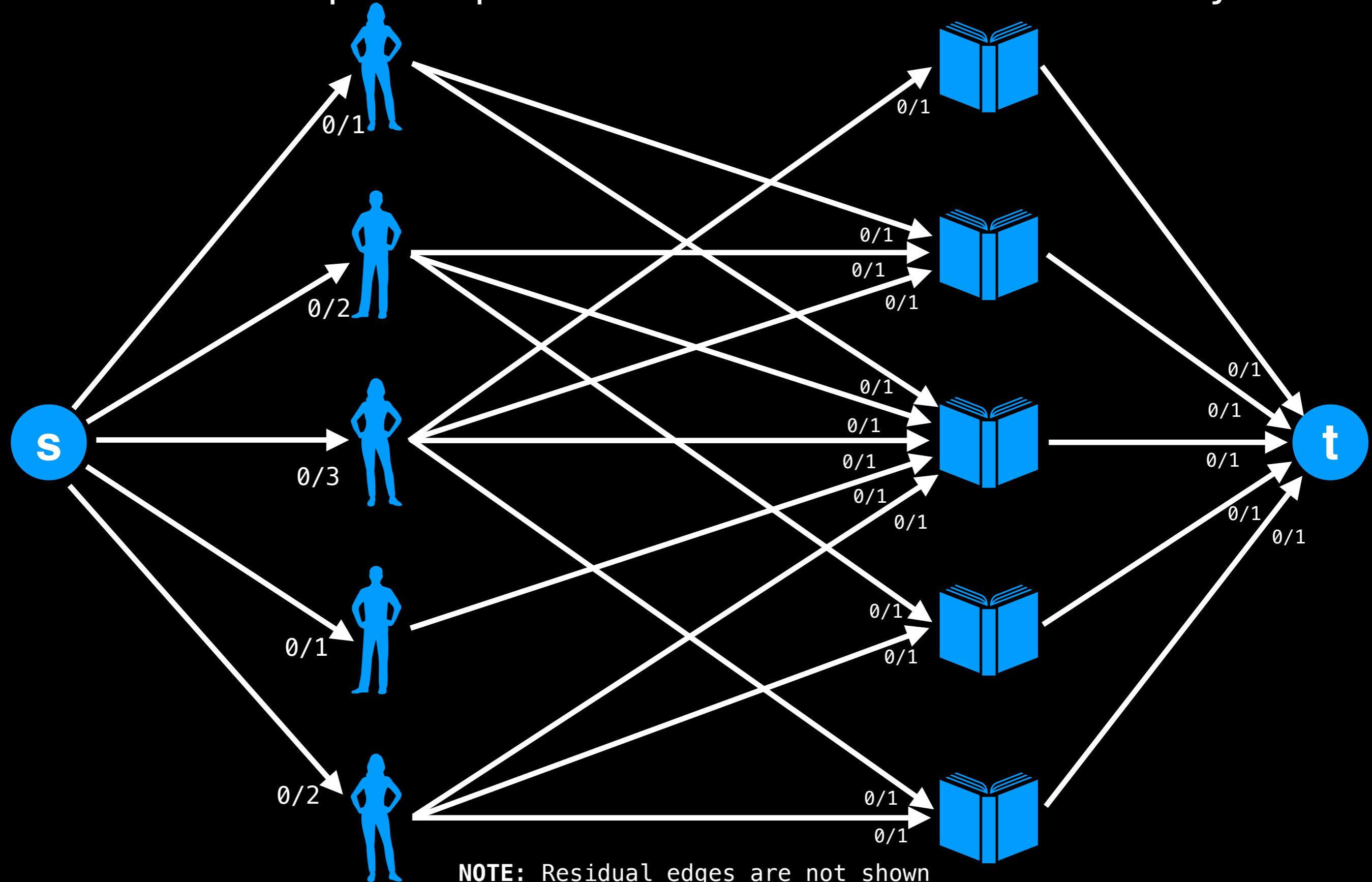
Let's increase some of these values to allow some people to possibly pick up more than one book.



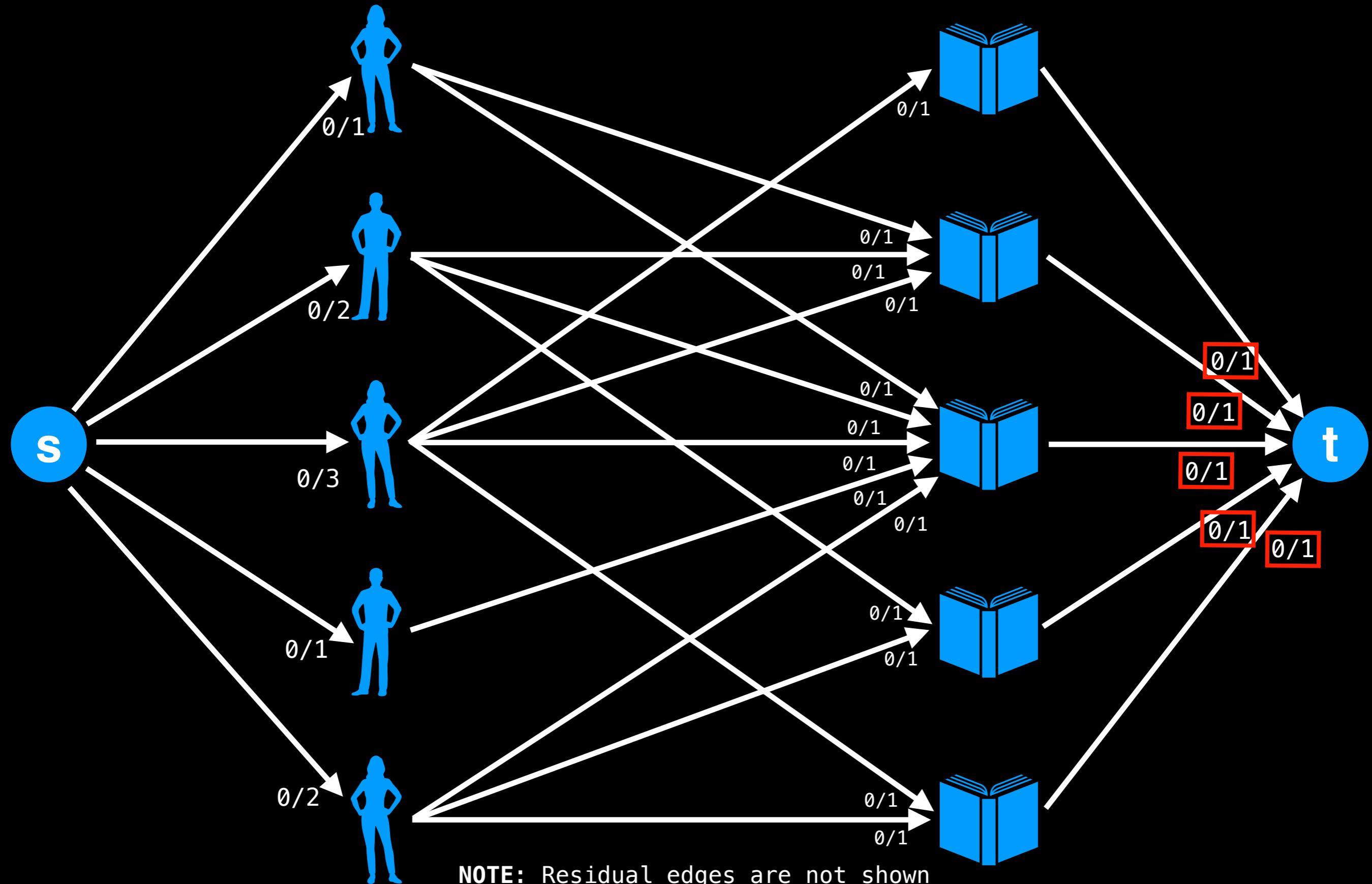
If we run max-flow again through the network we see that it's now possible for one person to have multiple books.



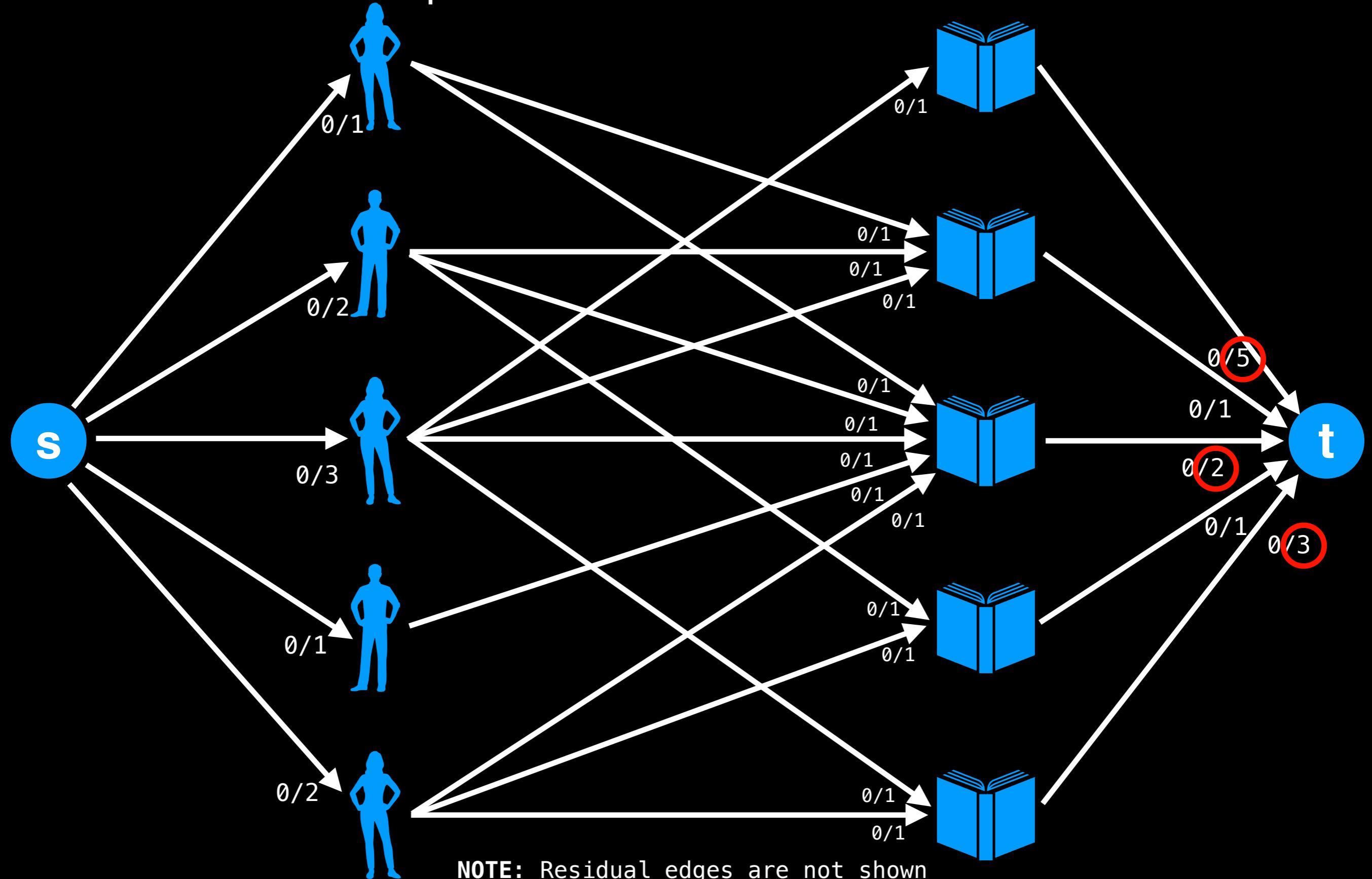
Q: What do we need to change in the flow network to allow a book to be selected multiple times (e.g there are multiple copies of the book in the library)?



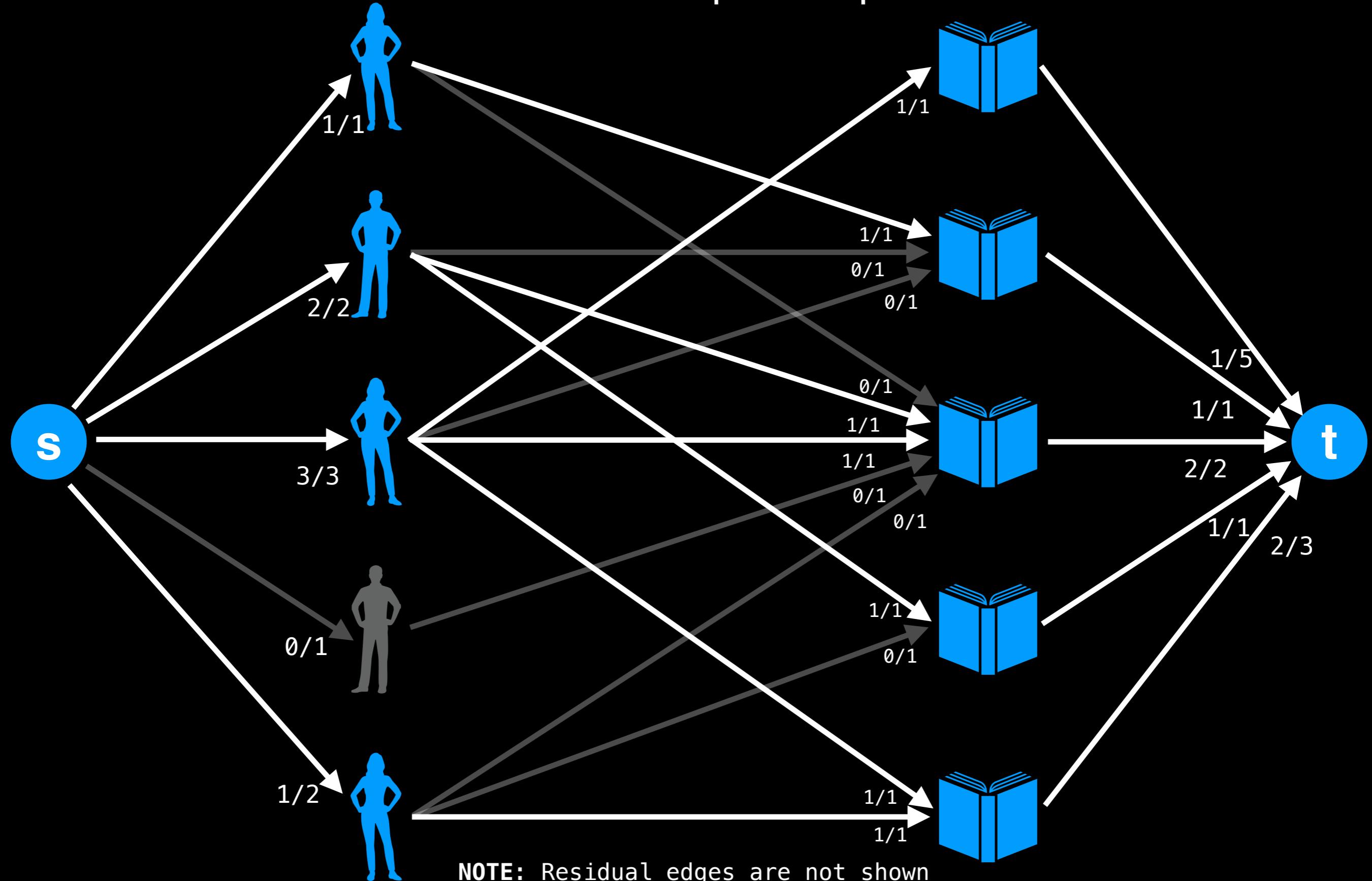
A: The number of “copies” of a book is determined by the capacity of the edges leading to the sink,  $t$ .

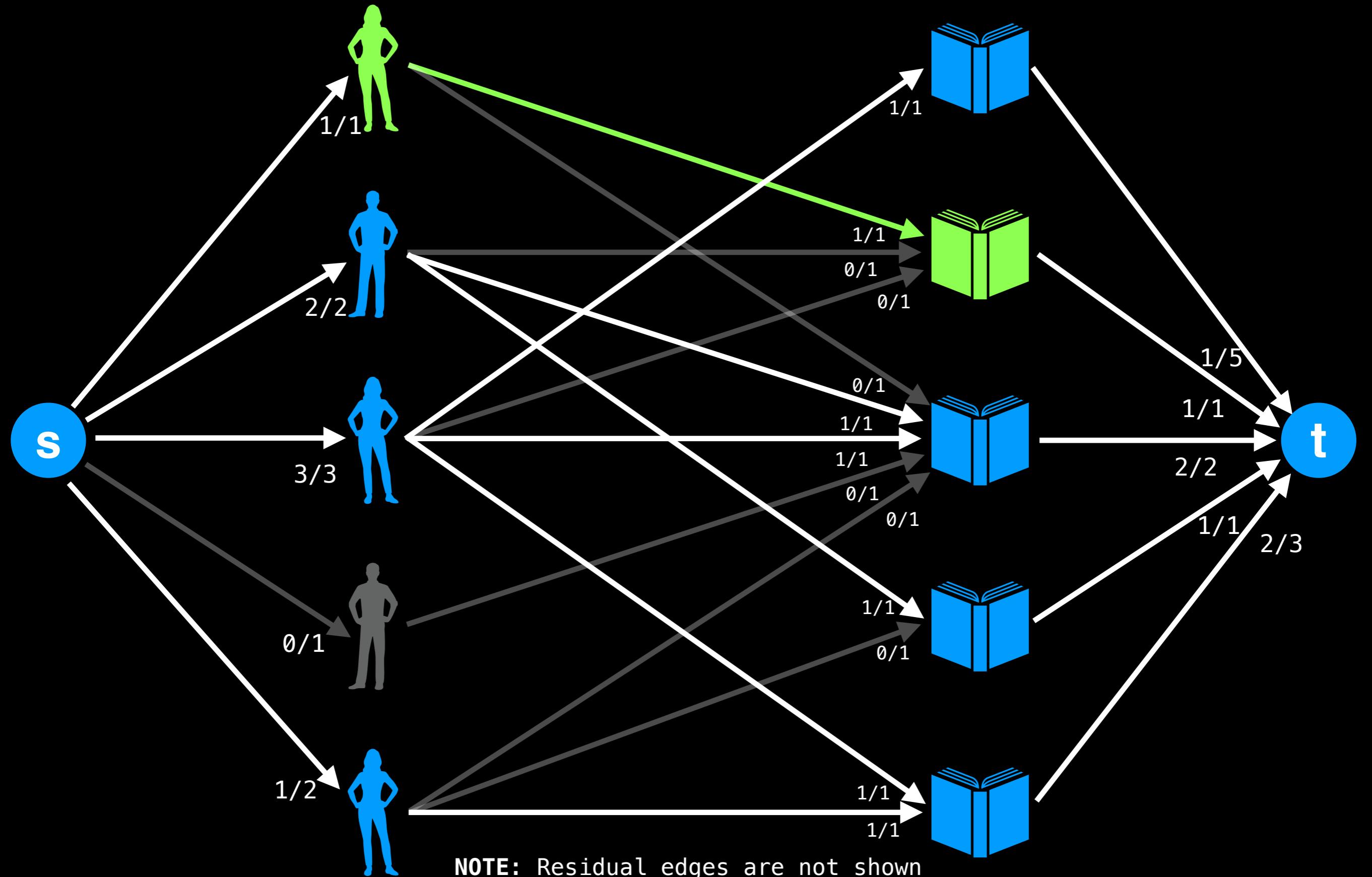


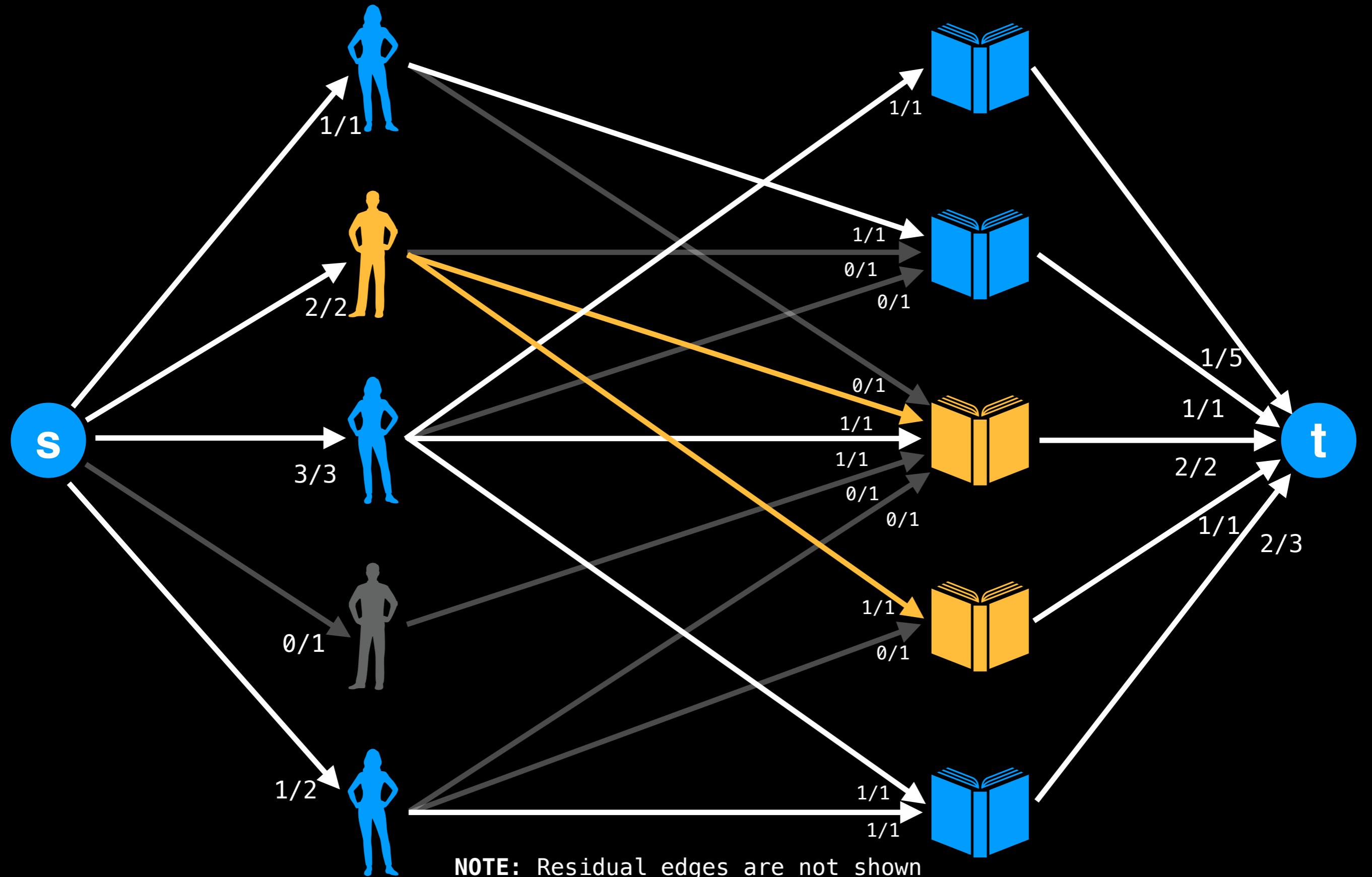
Let's change the capacity of some of the edges leading to the sink to allow having multiple copies of the same book.

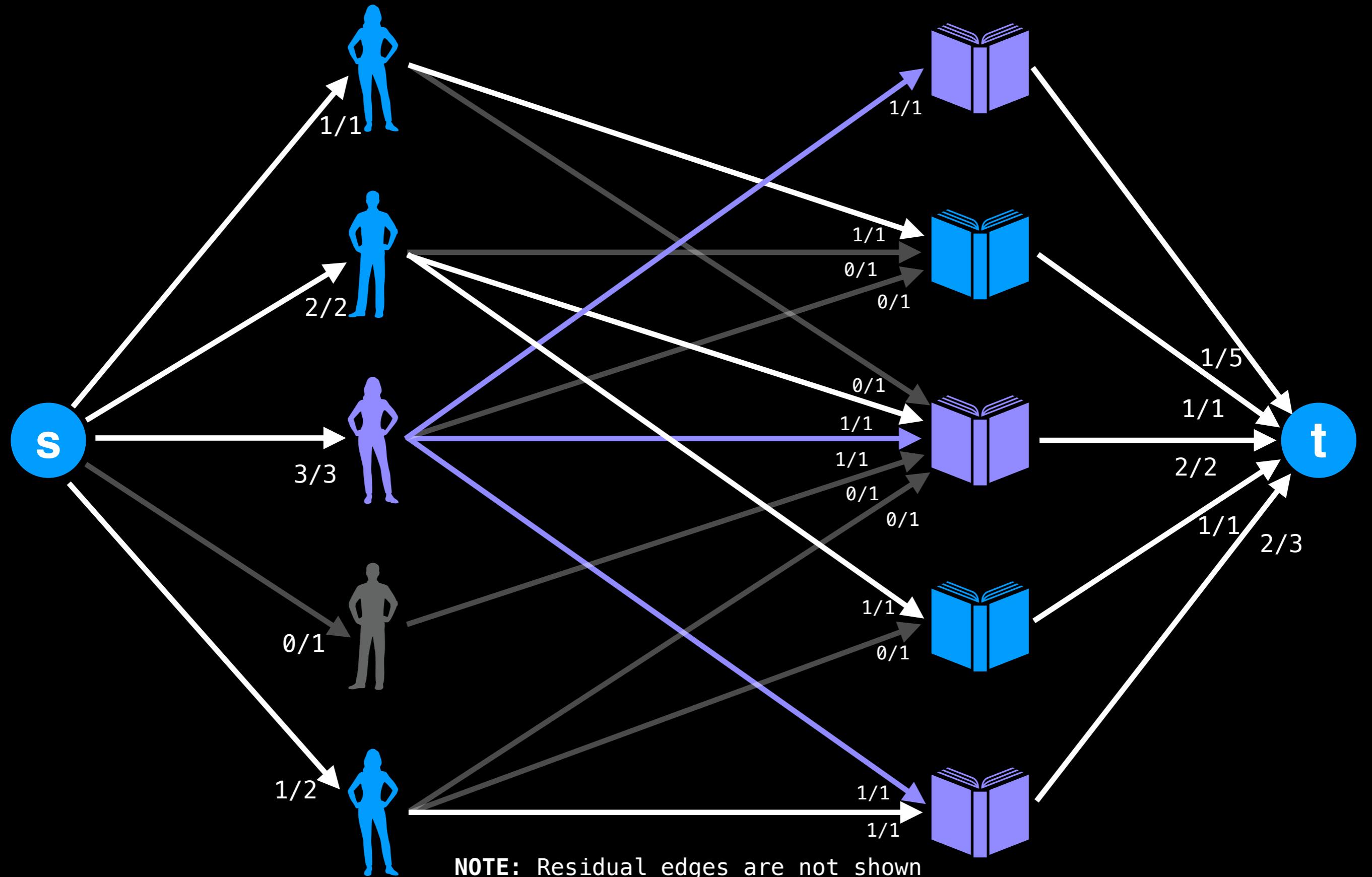


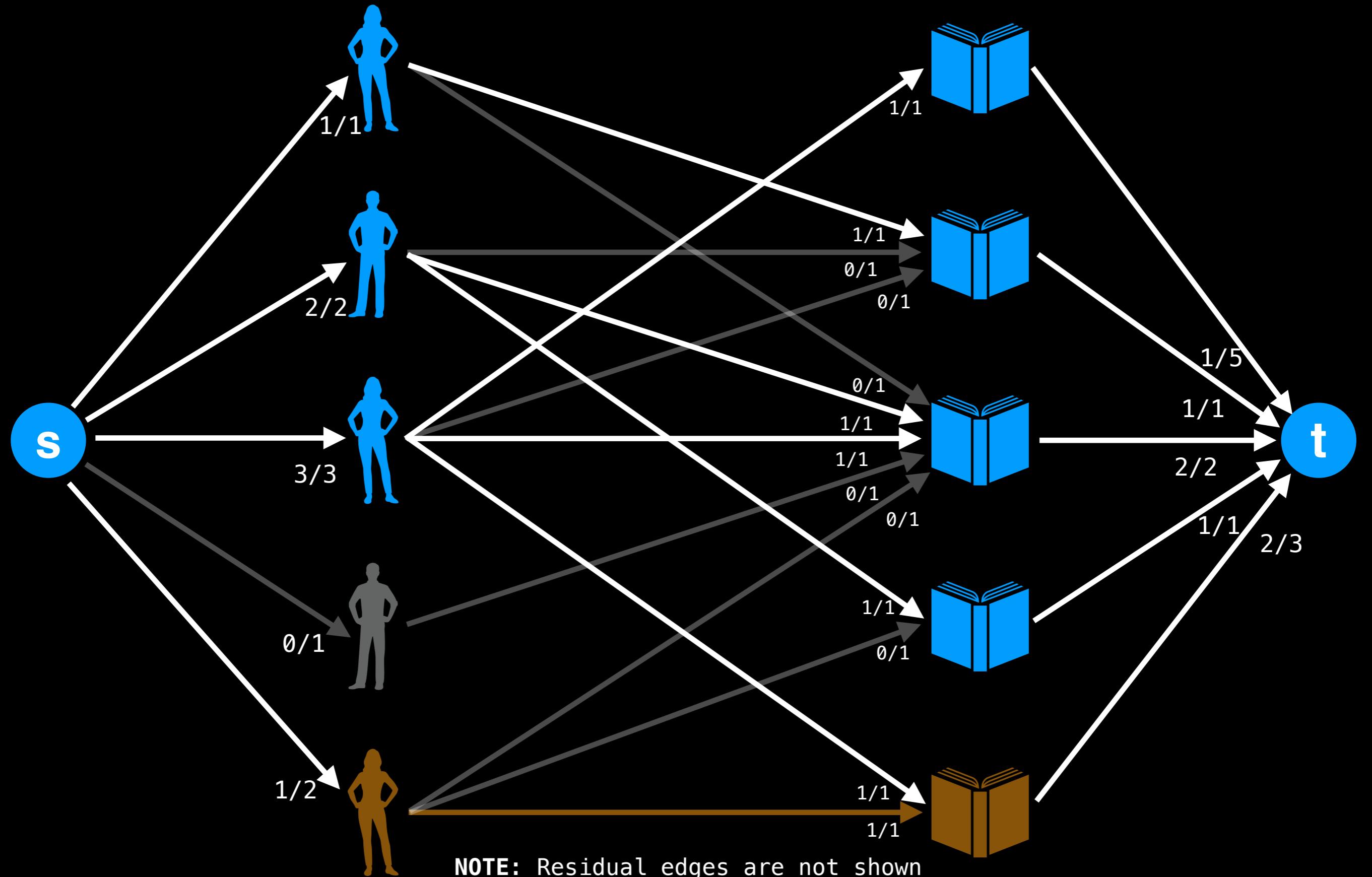
If we re-run the algorithm on the network, we see that we now have people matched with the same books because multiple copies exist.

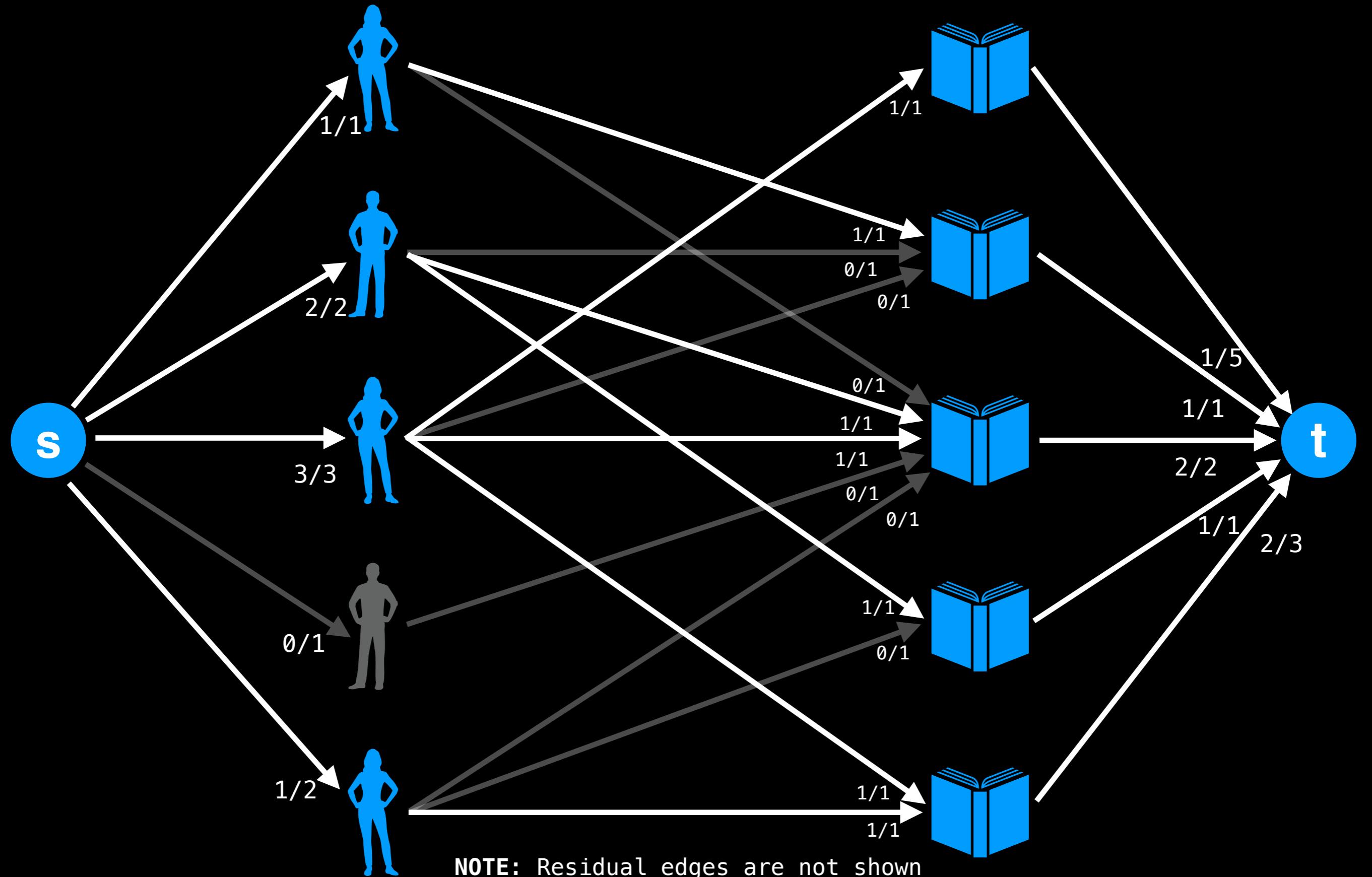




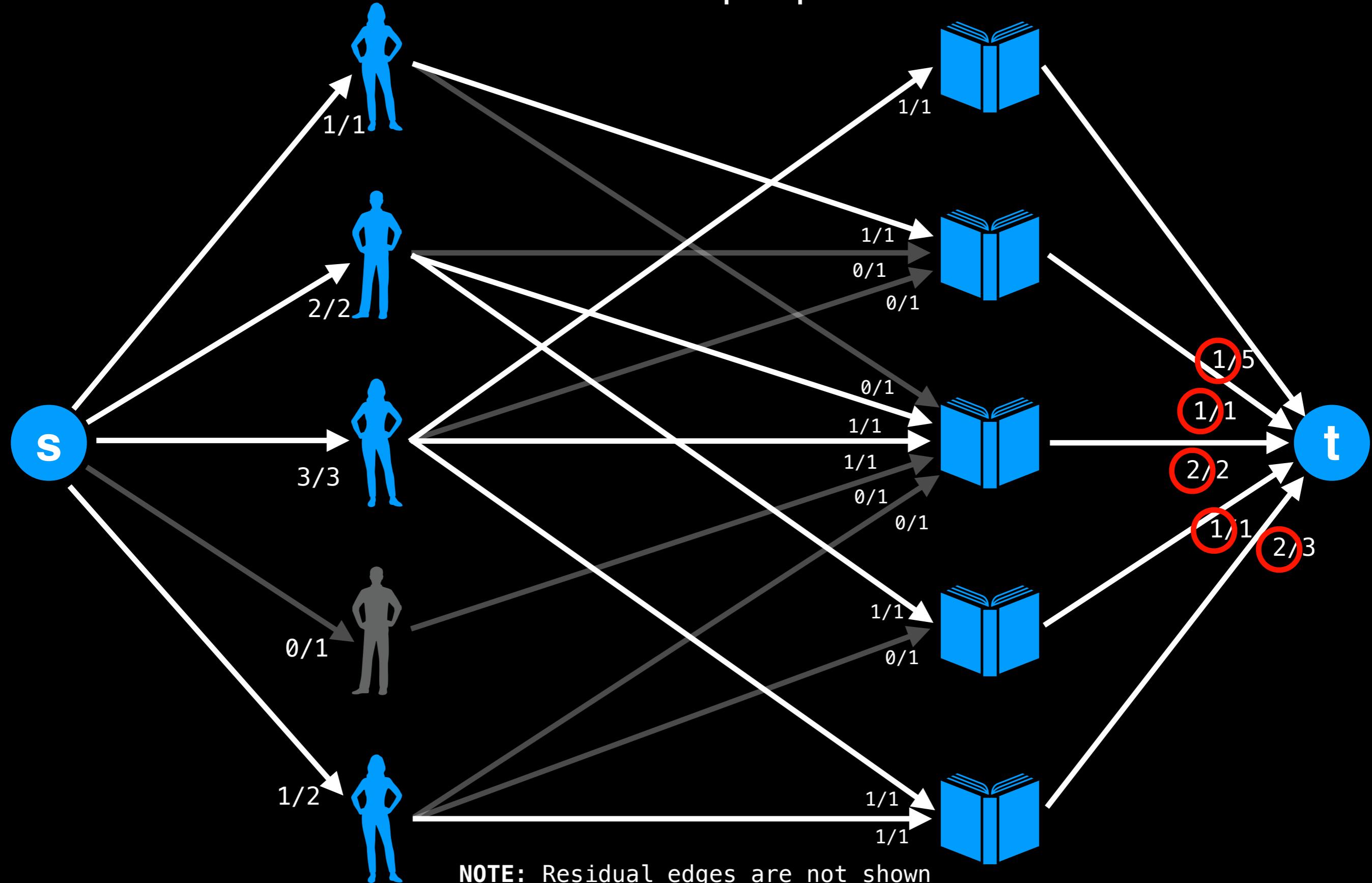




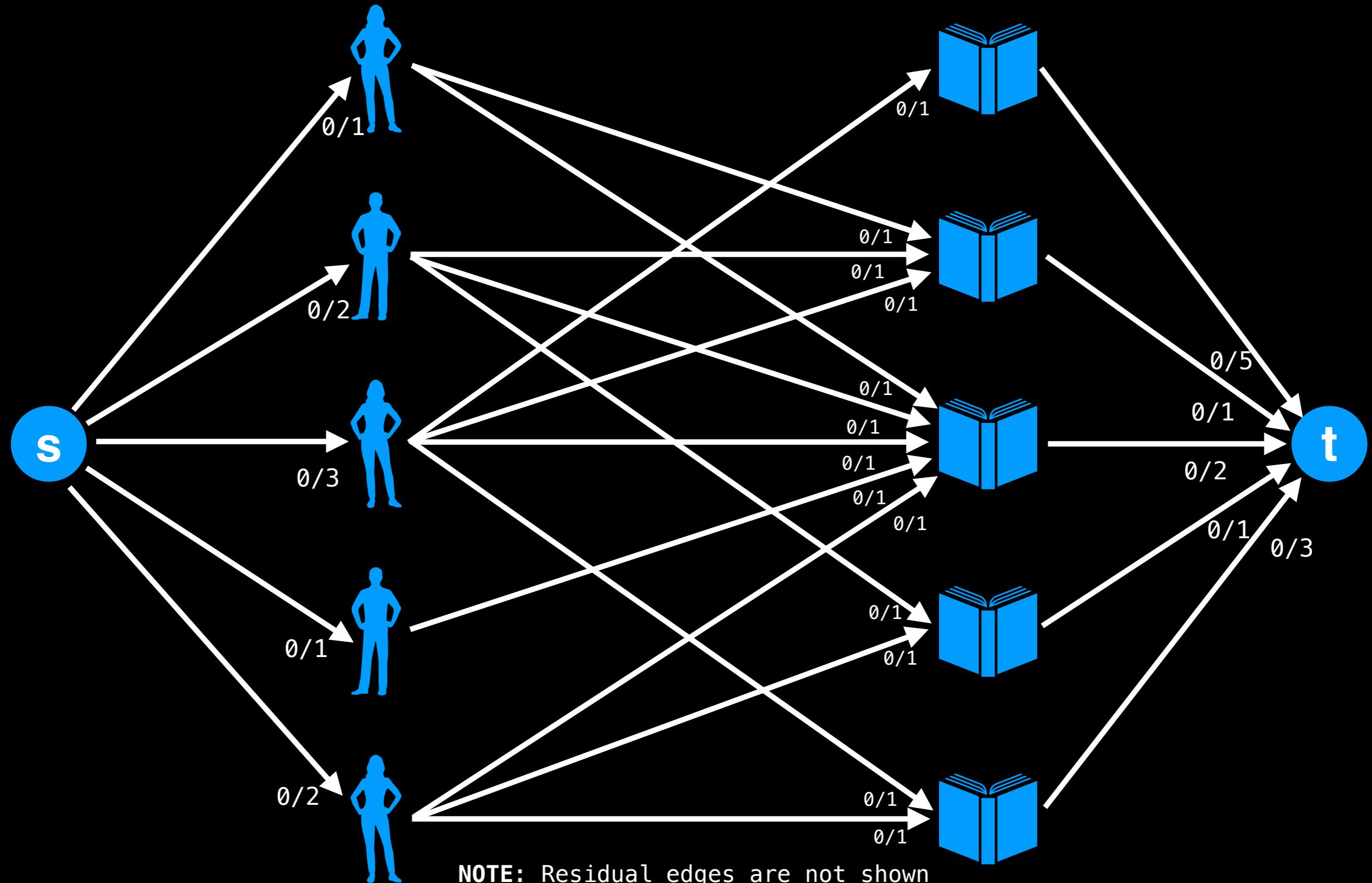




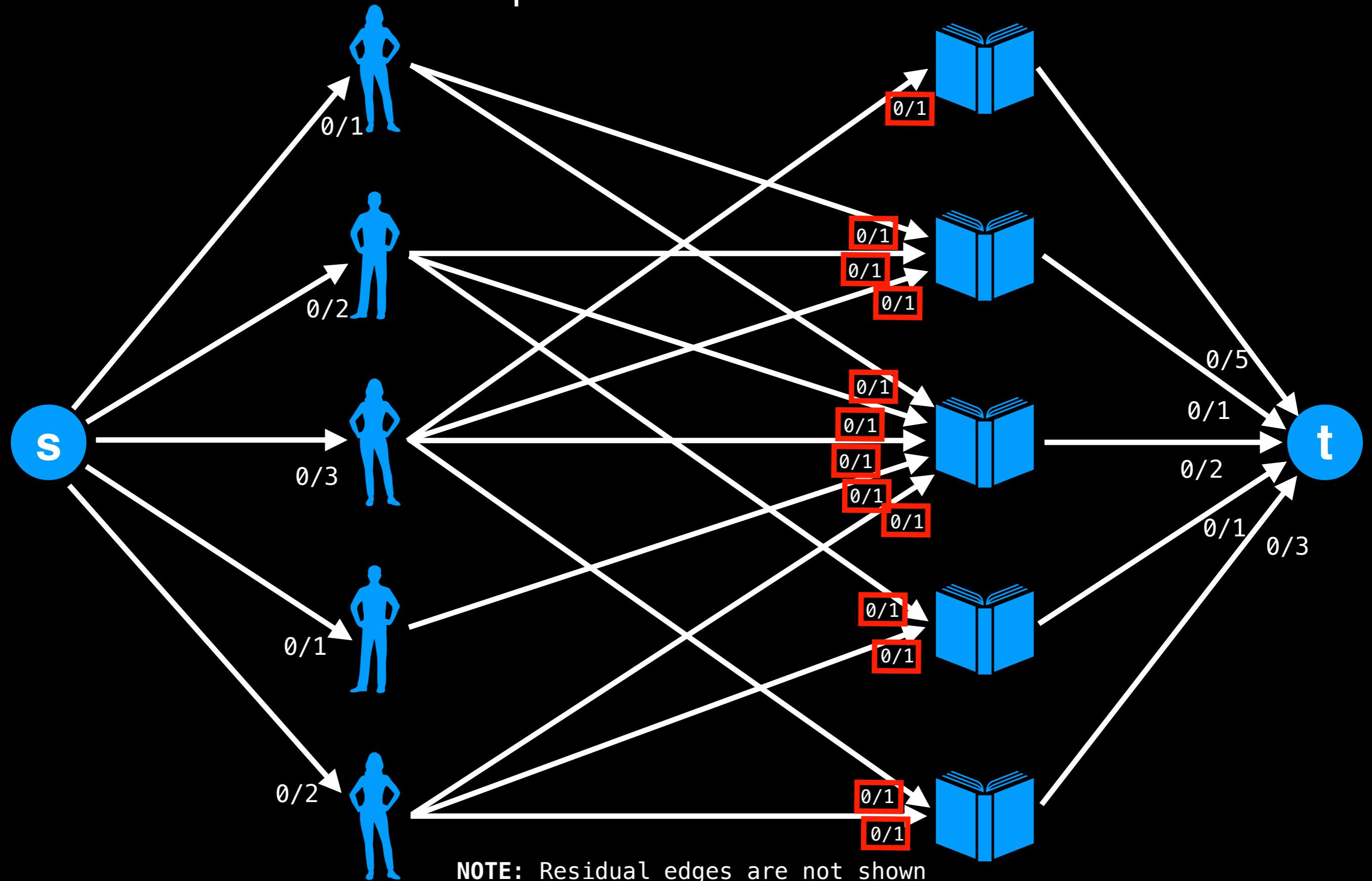
The flow value on the edge from the book to the sink indicates how many copies of that book were given out to various people.



Q: Currently, each person is only allowed to pick up one copy of each book (even though there are multiple copies of each book). How do we modify the flow network to support this?



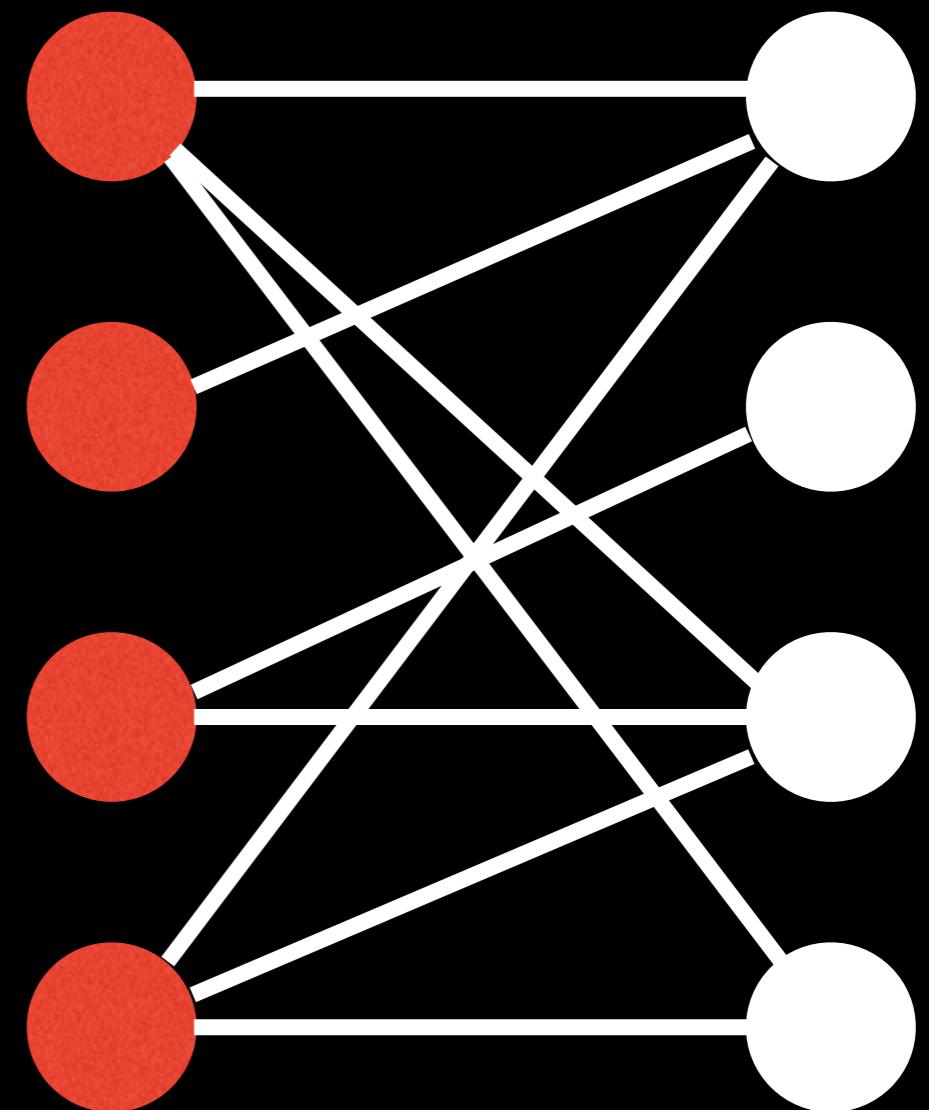
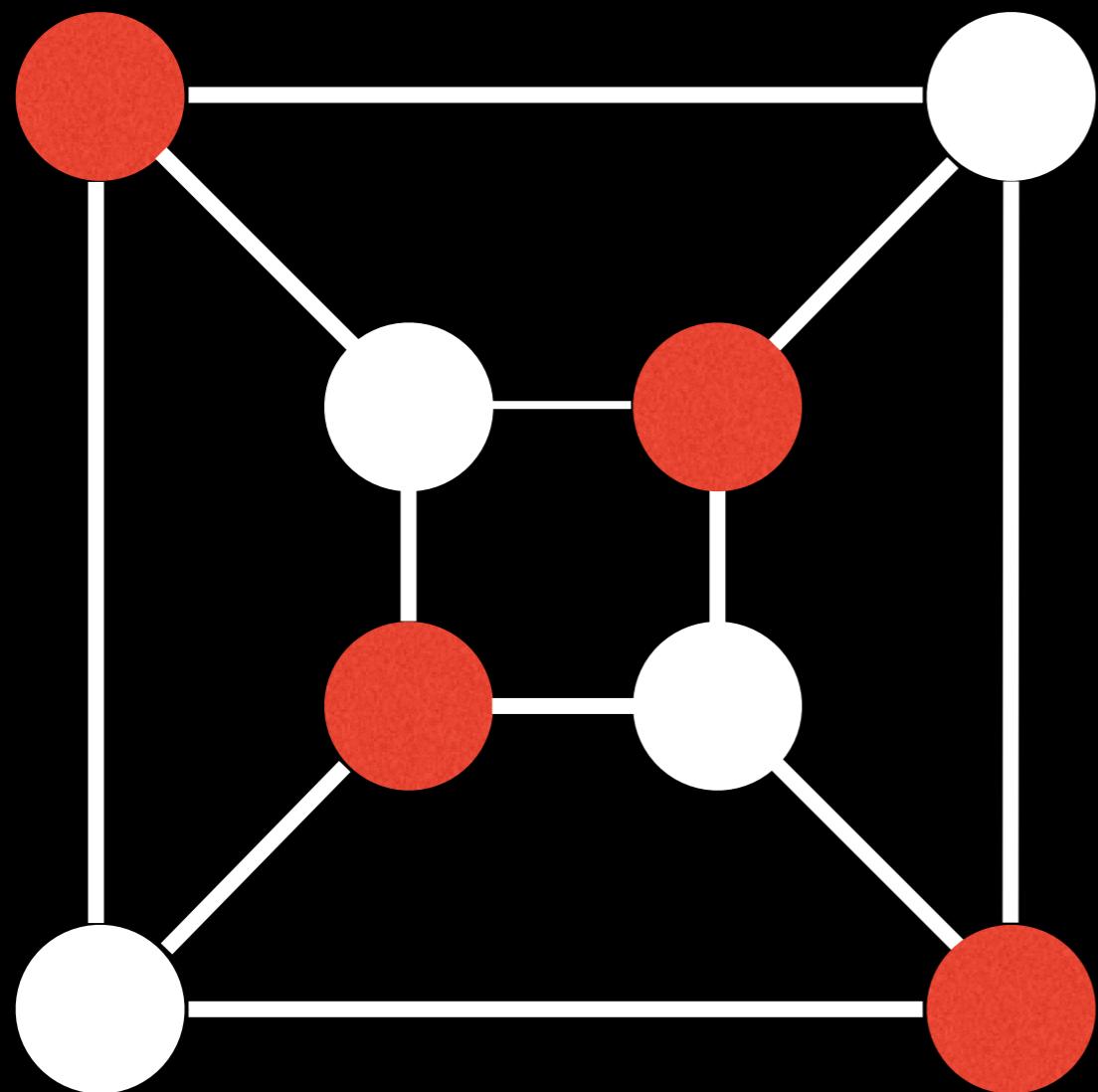
A: Modify the edge capacity between a person and a book to allow that person to pick up multiple copies of that book.



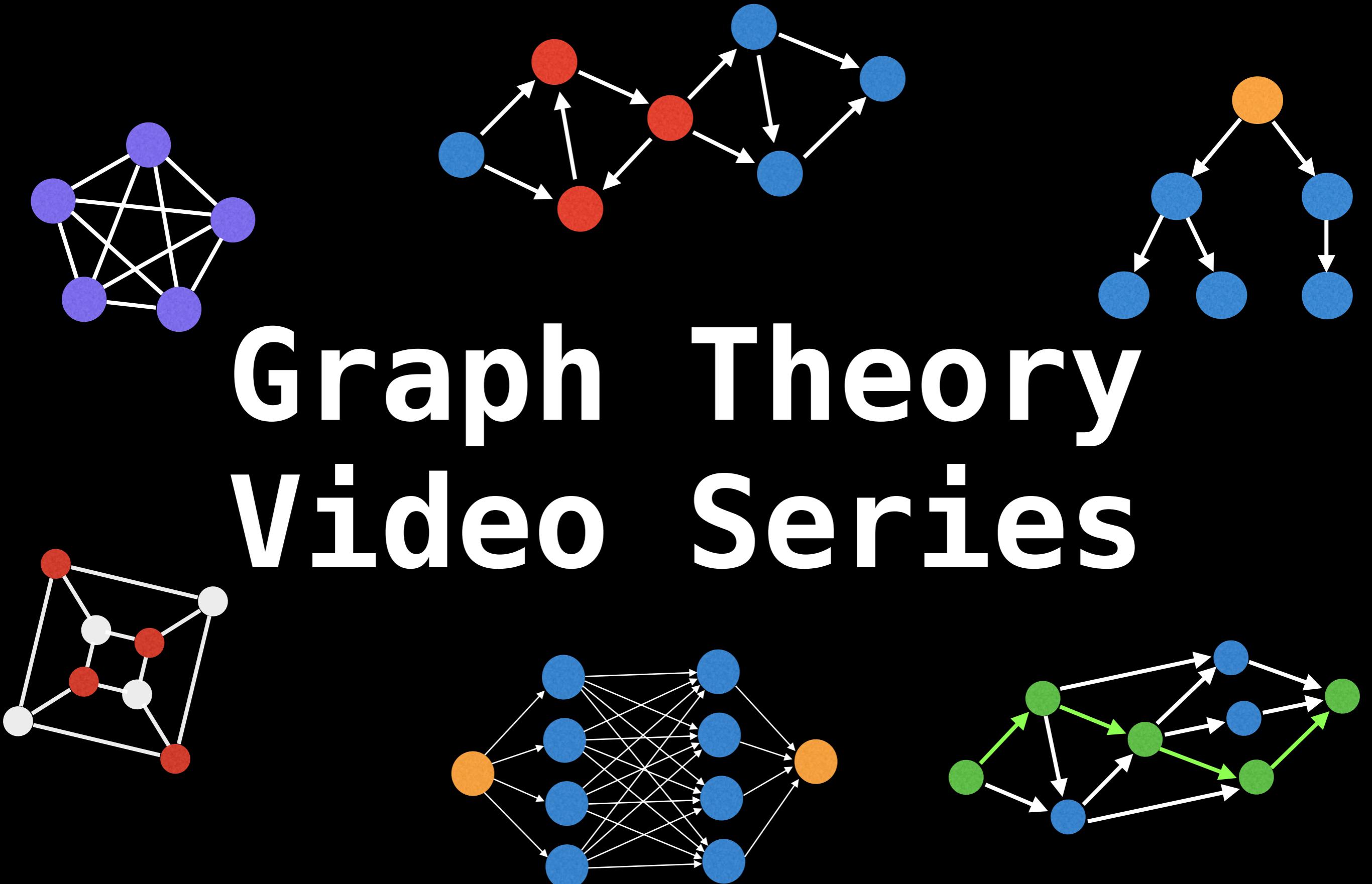


# Network Flow: Bipartite Matching

A **bipartite graph** is one whose *vertices* can be split into two independent groups  $U, V$  such that every edge connects between  $U$  and  $V$ .



# Graph Theory Video Series



# Network Flow: mice and owls



William Fiset

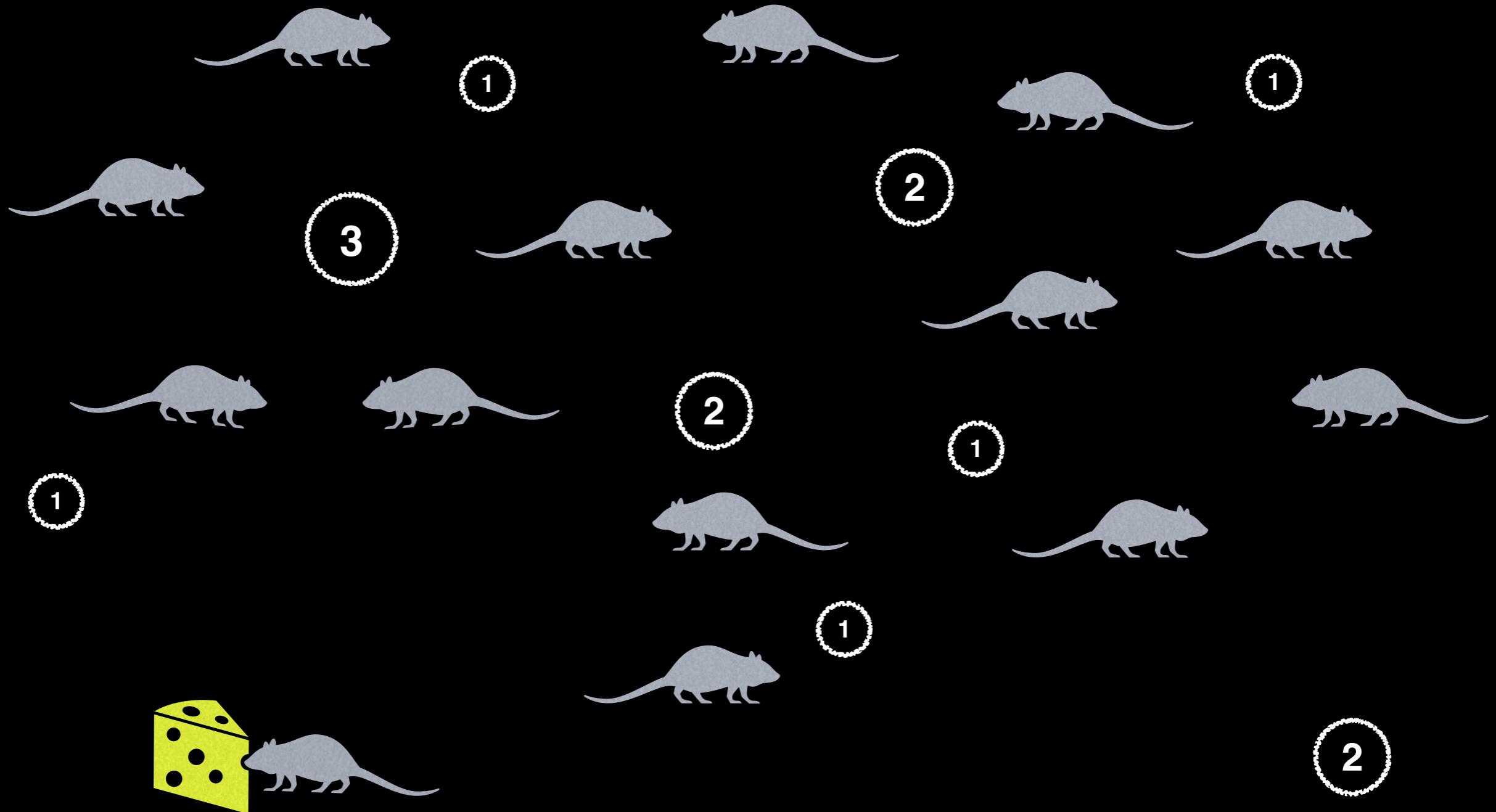


Suppose  $M$  mice are out on a field and there's a hungry owl about to make a move.



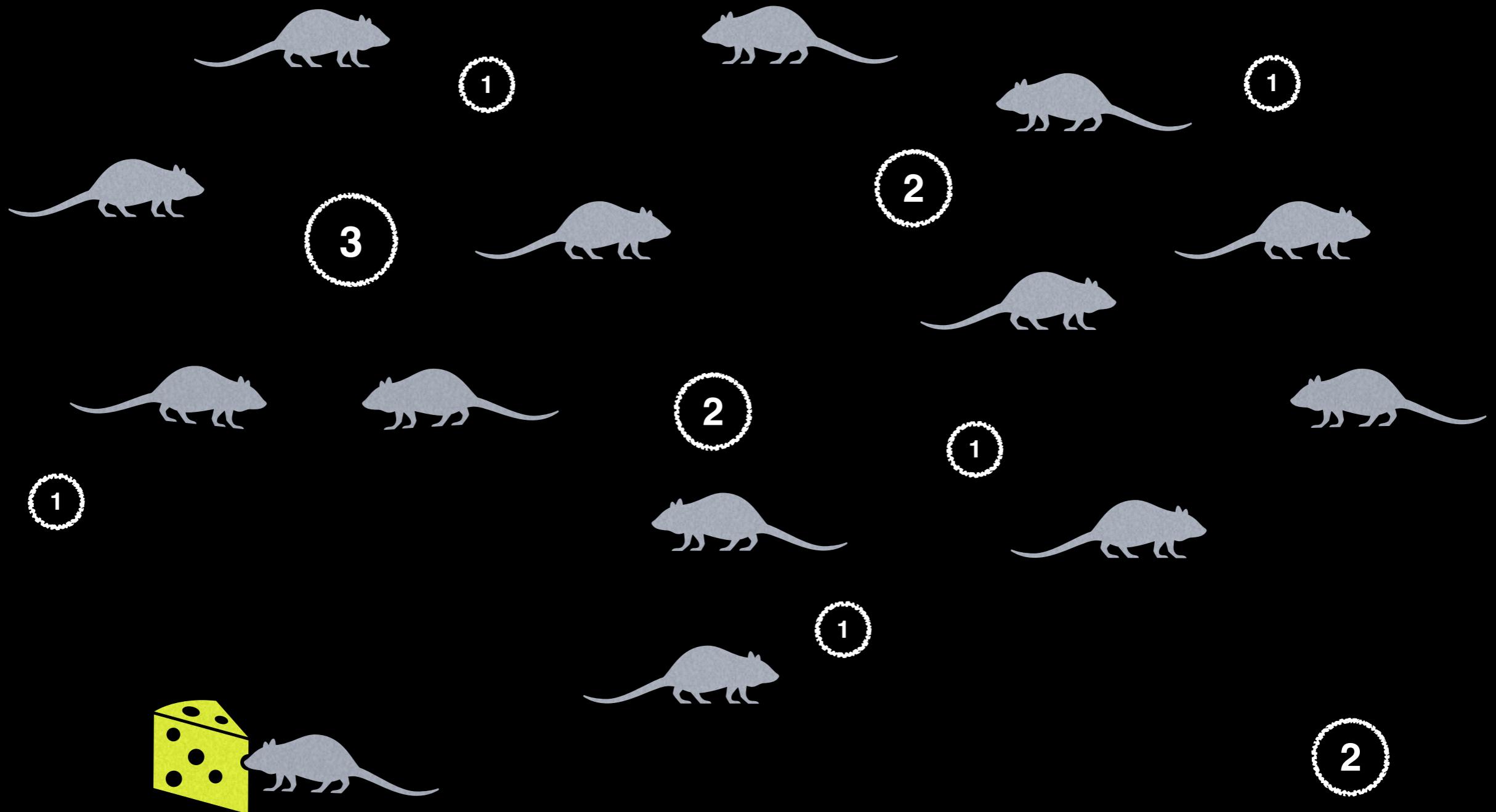


Further suppose there are  $H$  holes scattered across the ground that the mice can hide in (each having a certain capacity).



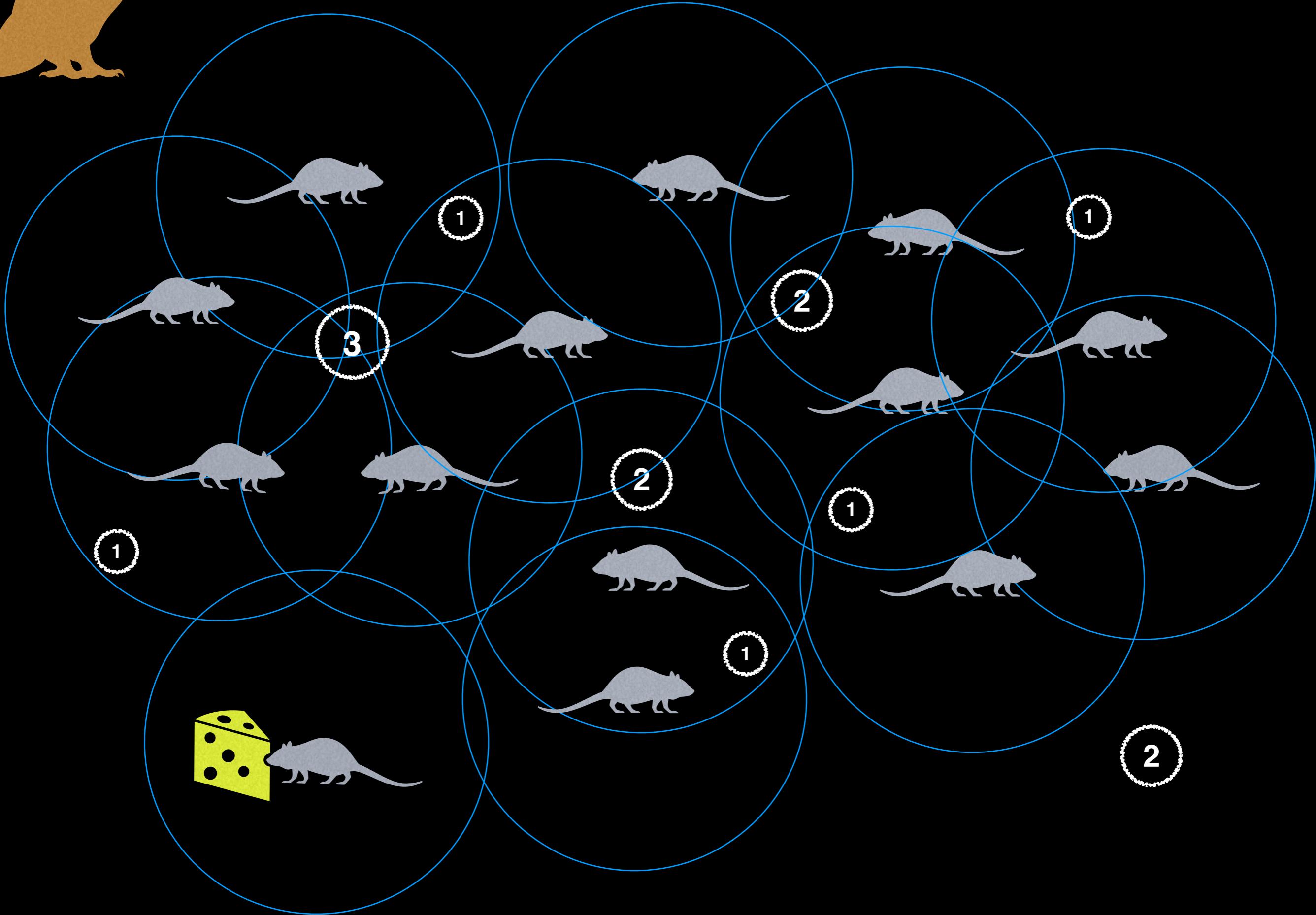


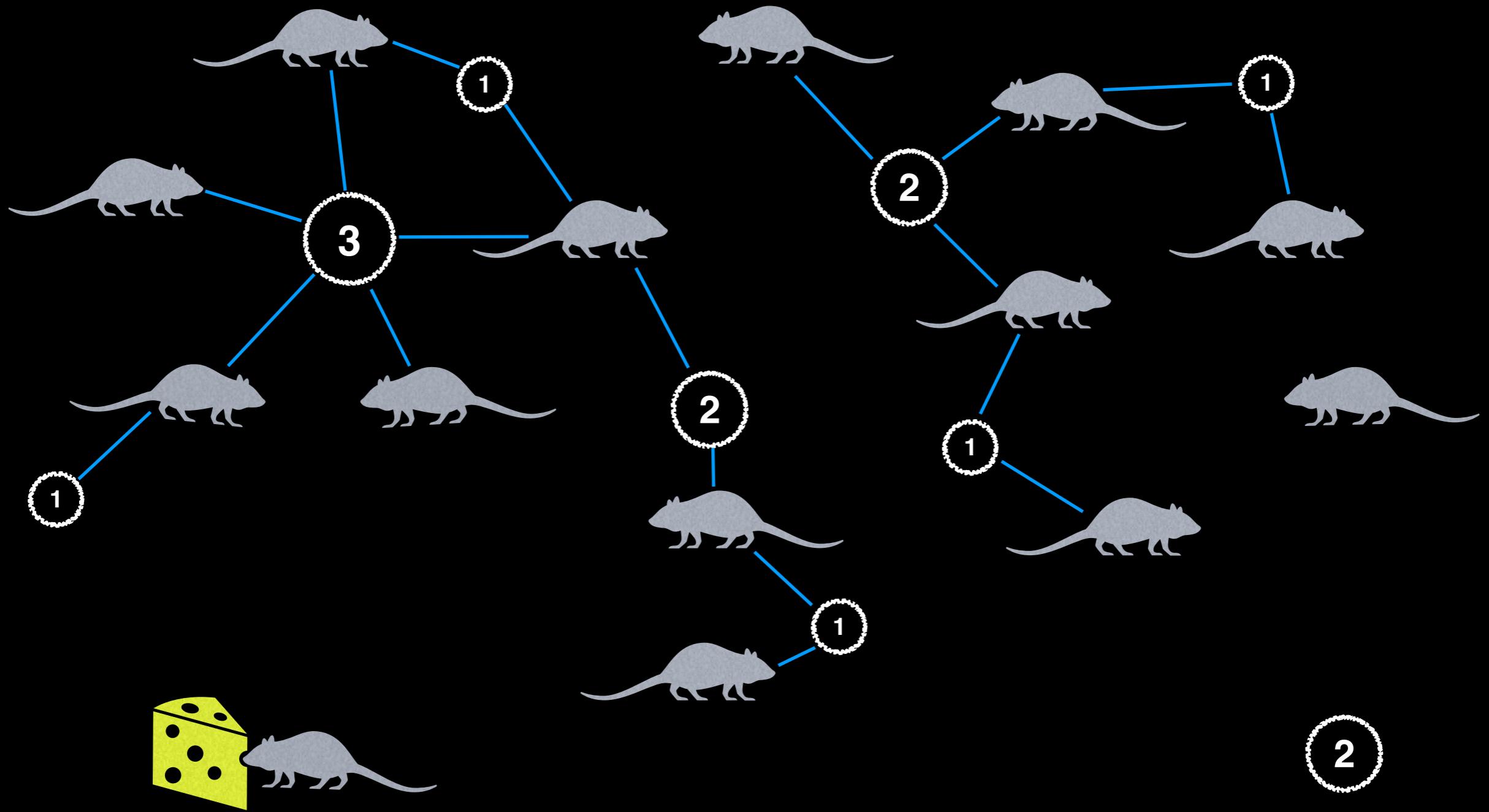
Assume every mouse is capable of running a radius of  $r$  before being caught by the owl. What is the maximum number of mice that can hide safely?





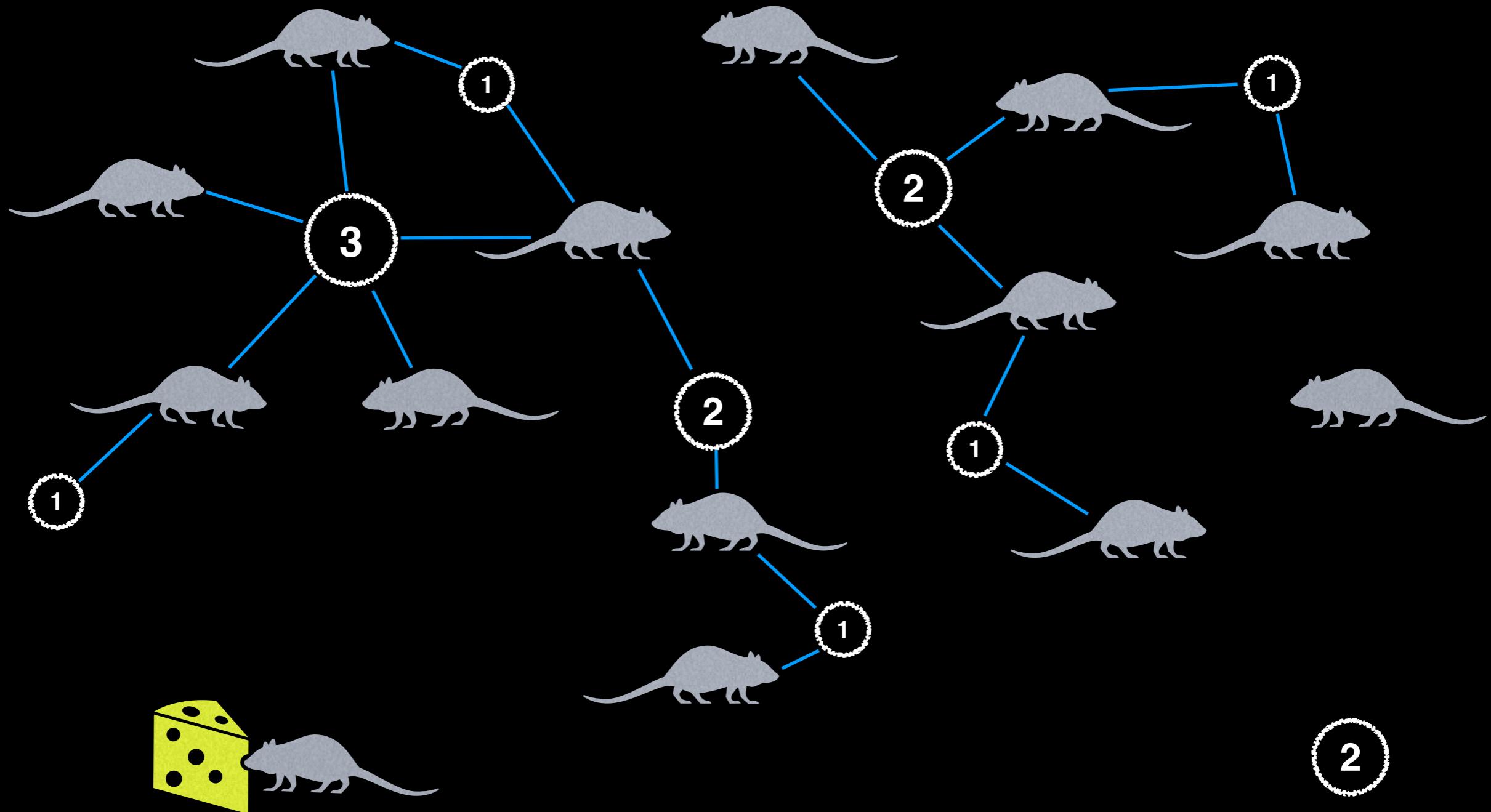
The first step is to figure out which holes each mouse can reach.





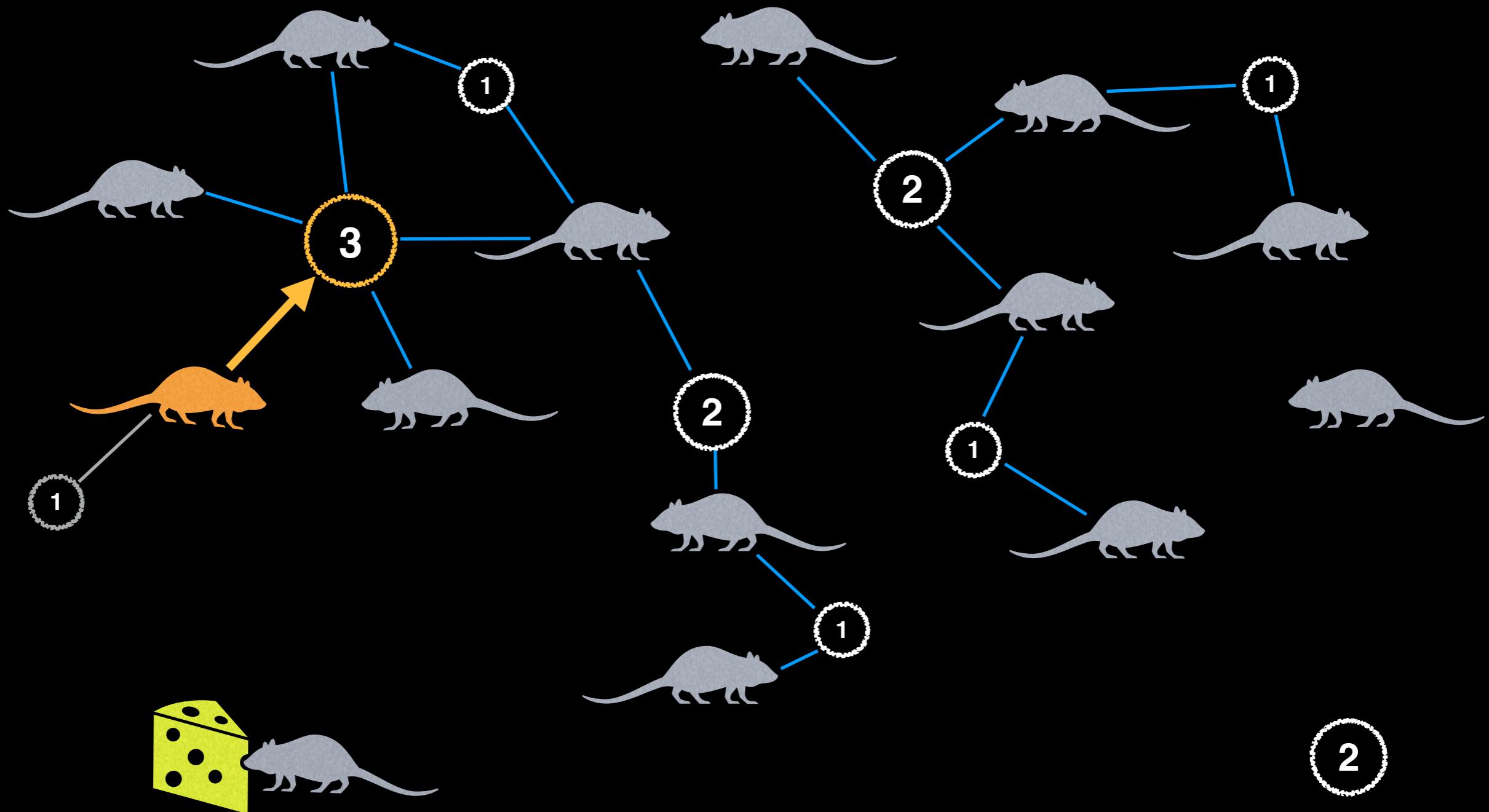


Which mice should go in which holes to maximize the overall safety of the group?



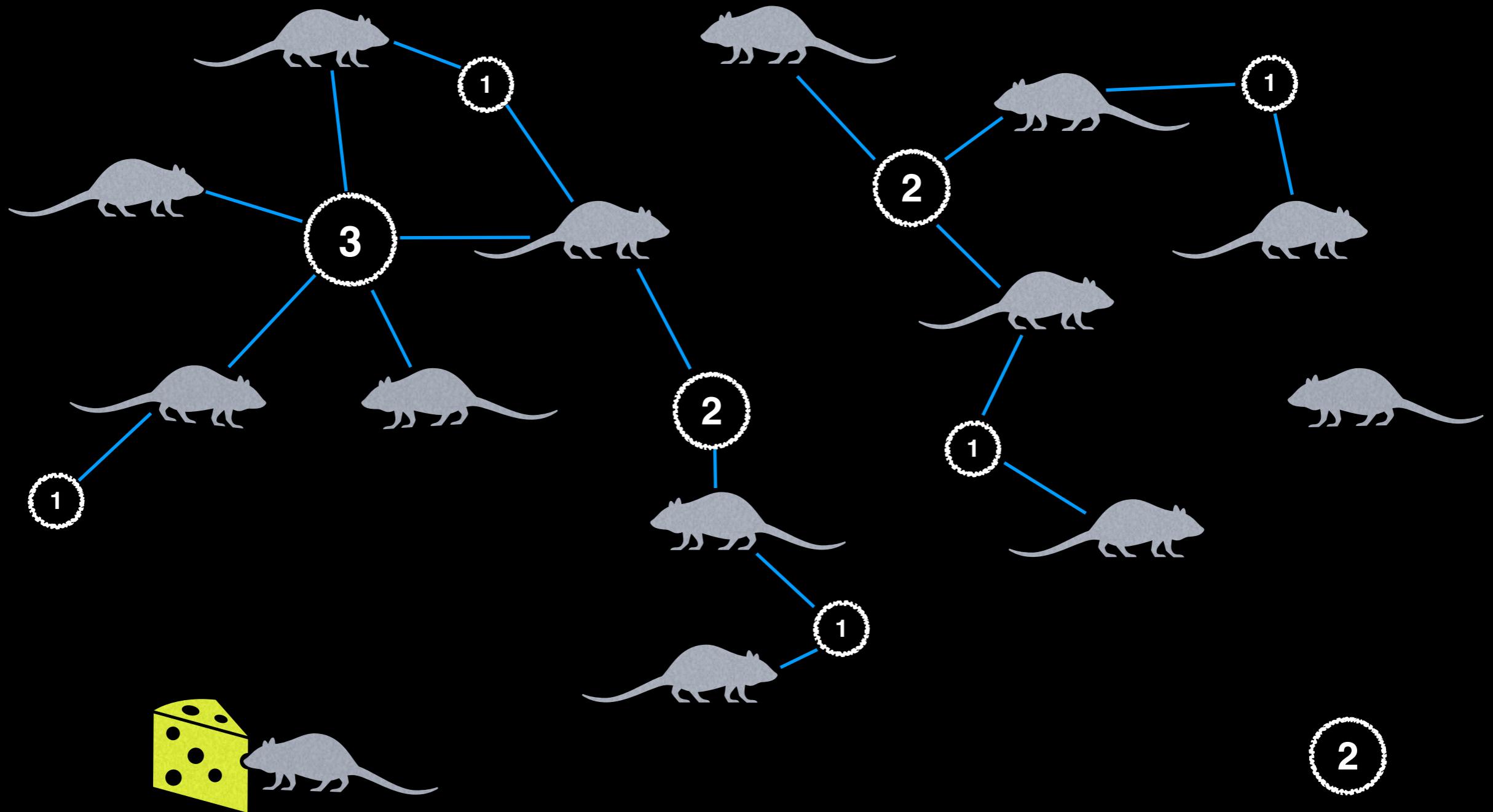


Which mice should go in which holes to maximize the overall safety of the group?

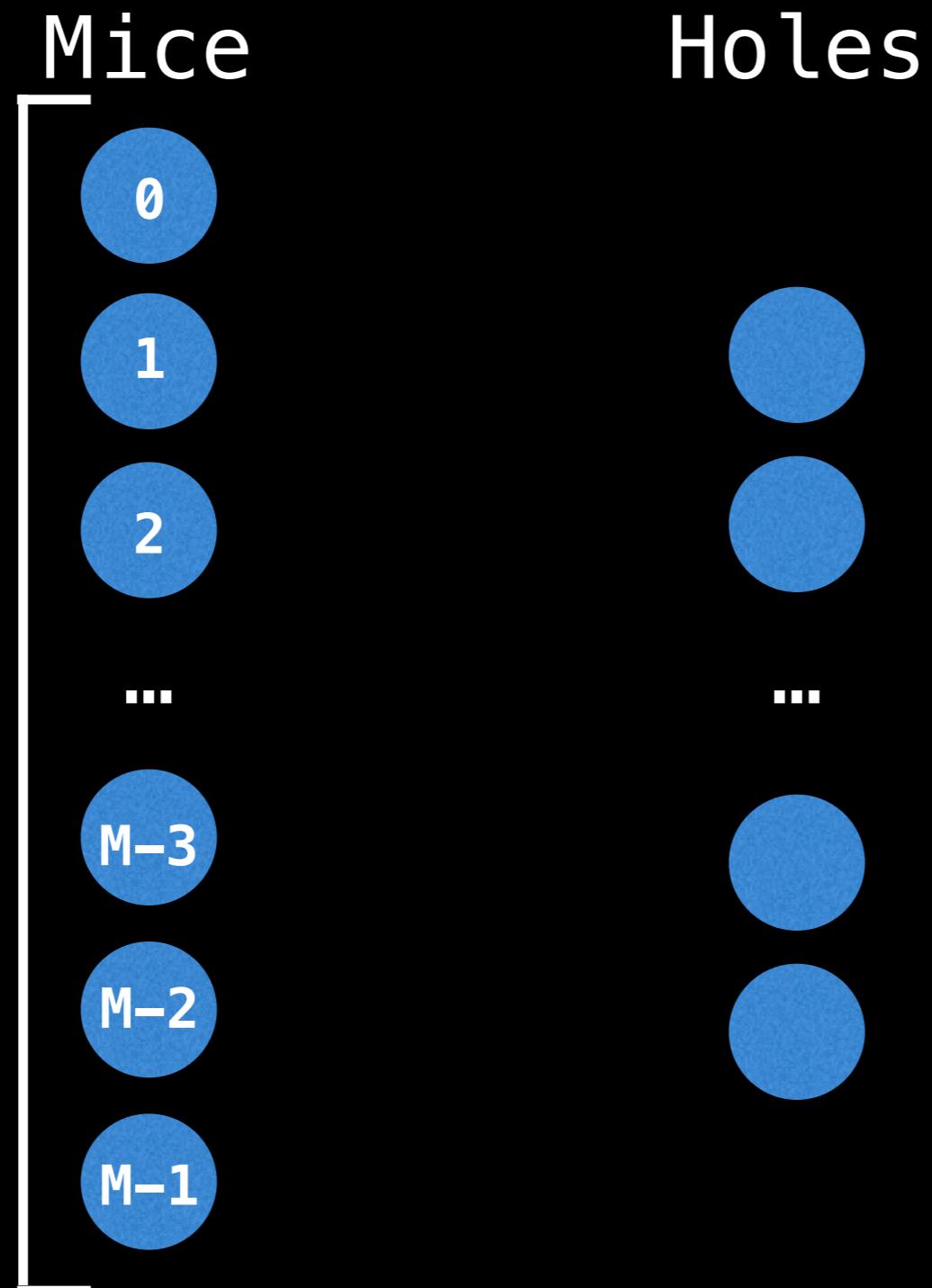




The key realization to make is that this graph is **bipartite**.



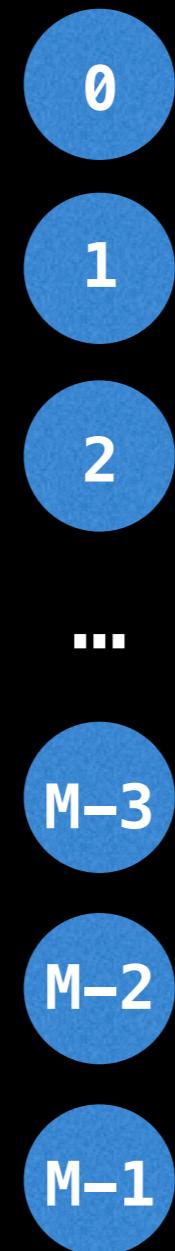
# Flow Graph Setup



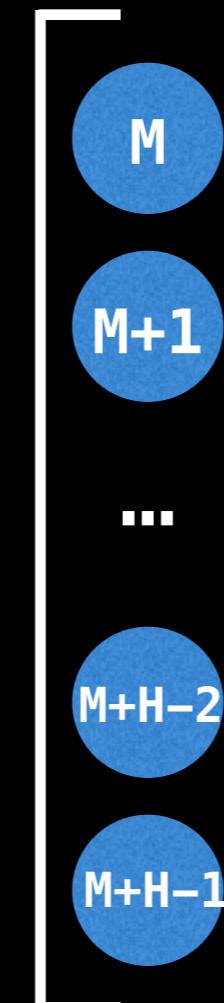
Label (index) the mice nodes  $[0, M)$

# Flow Graph Setup

Mice



Holes

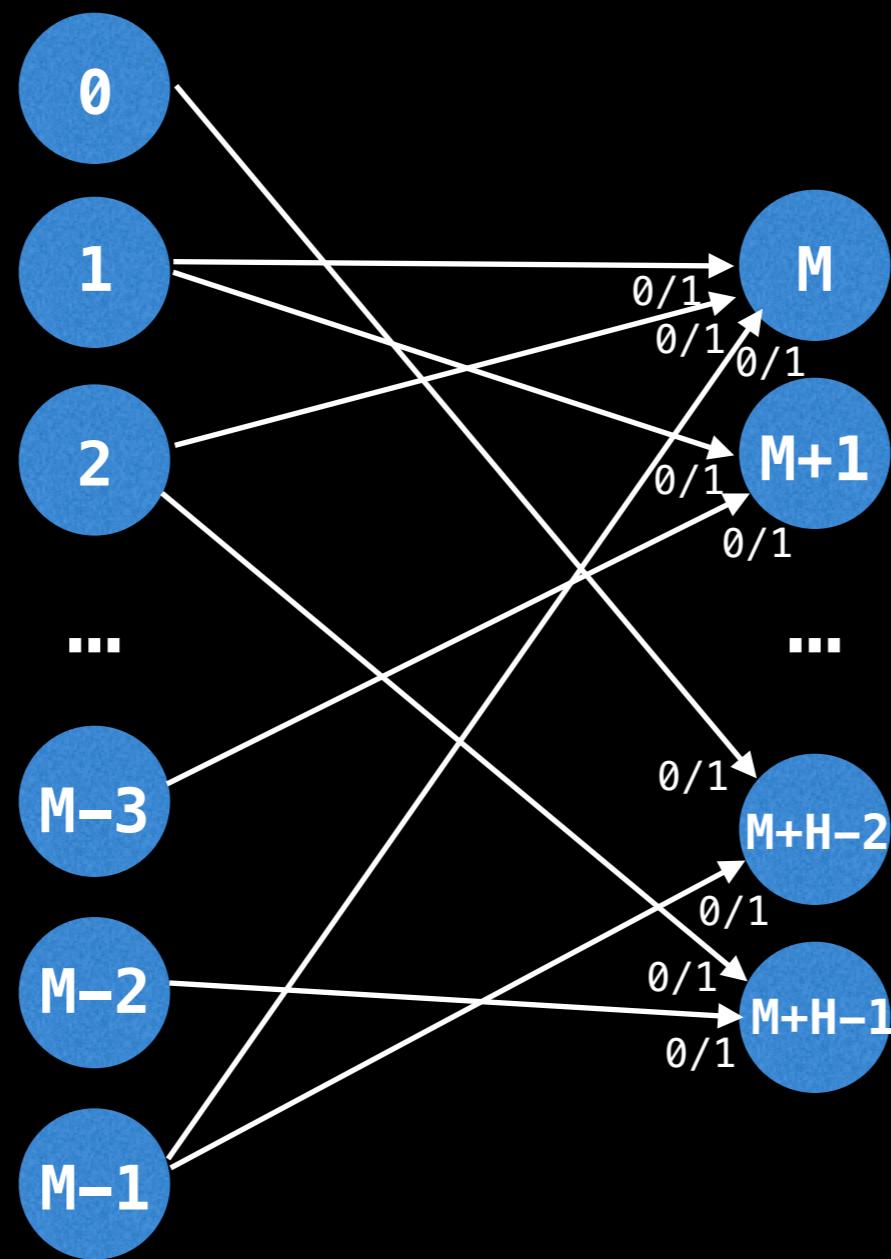


Label (index) the hole nodes  $[M, M+H)$

# Flow Graph Setup

# Mice

## Holes

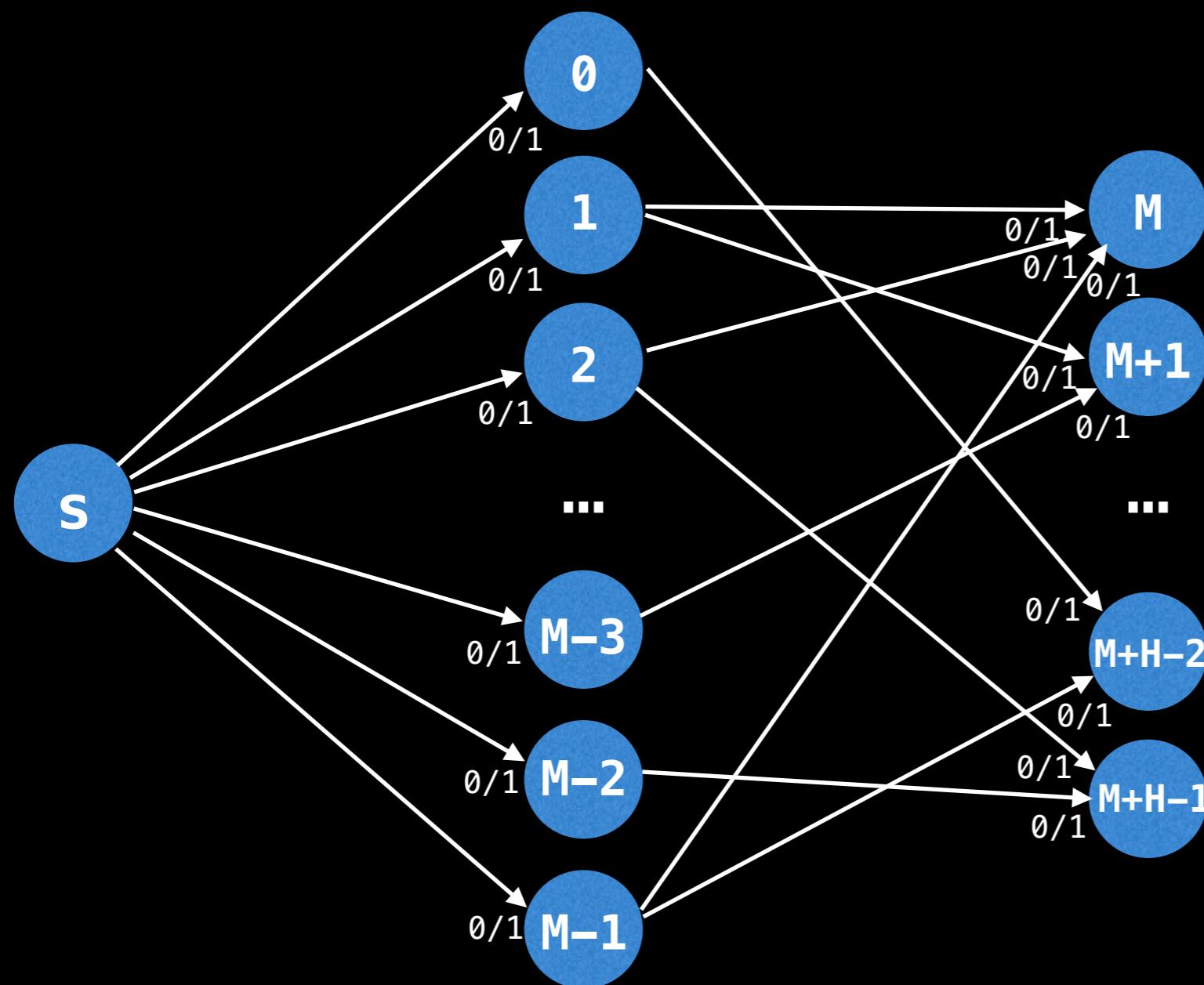


Place an edge with a capacity of 1 between the mouse and the hole if the mouse can reach the particular hole.

# Flow Graph Setup

# Mice

## Holes

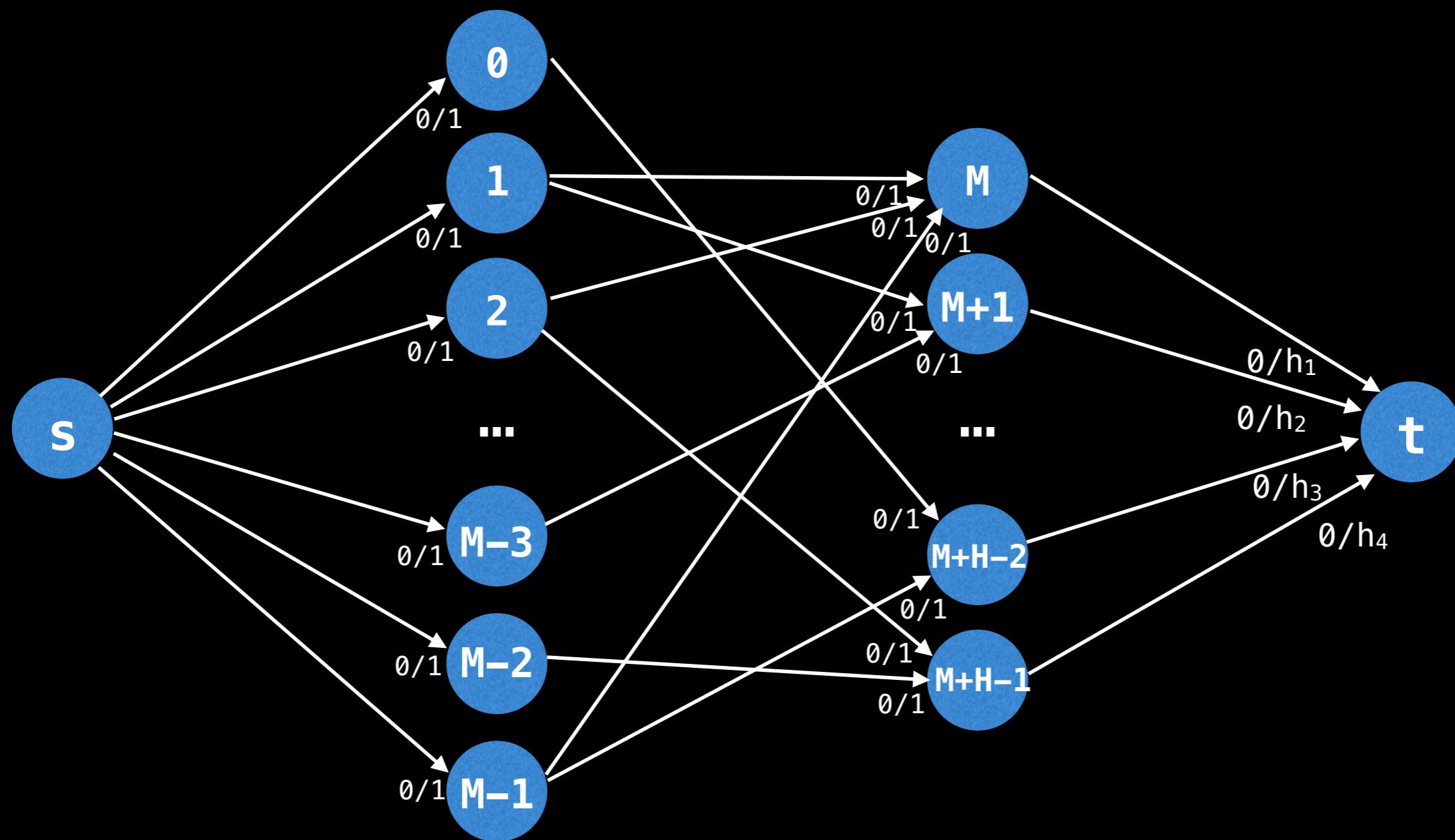


Connect an edge with capacity 1 from the source to each mouse node to indicate that each node can have at most 1 mouse.

# Flow Graph Setup

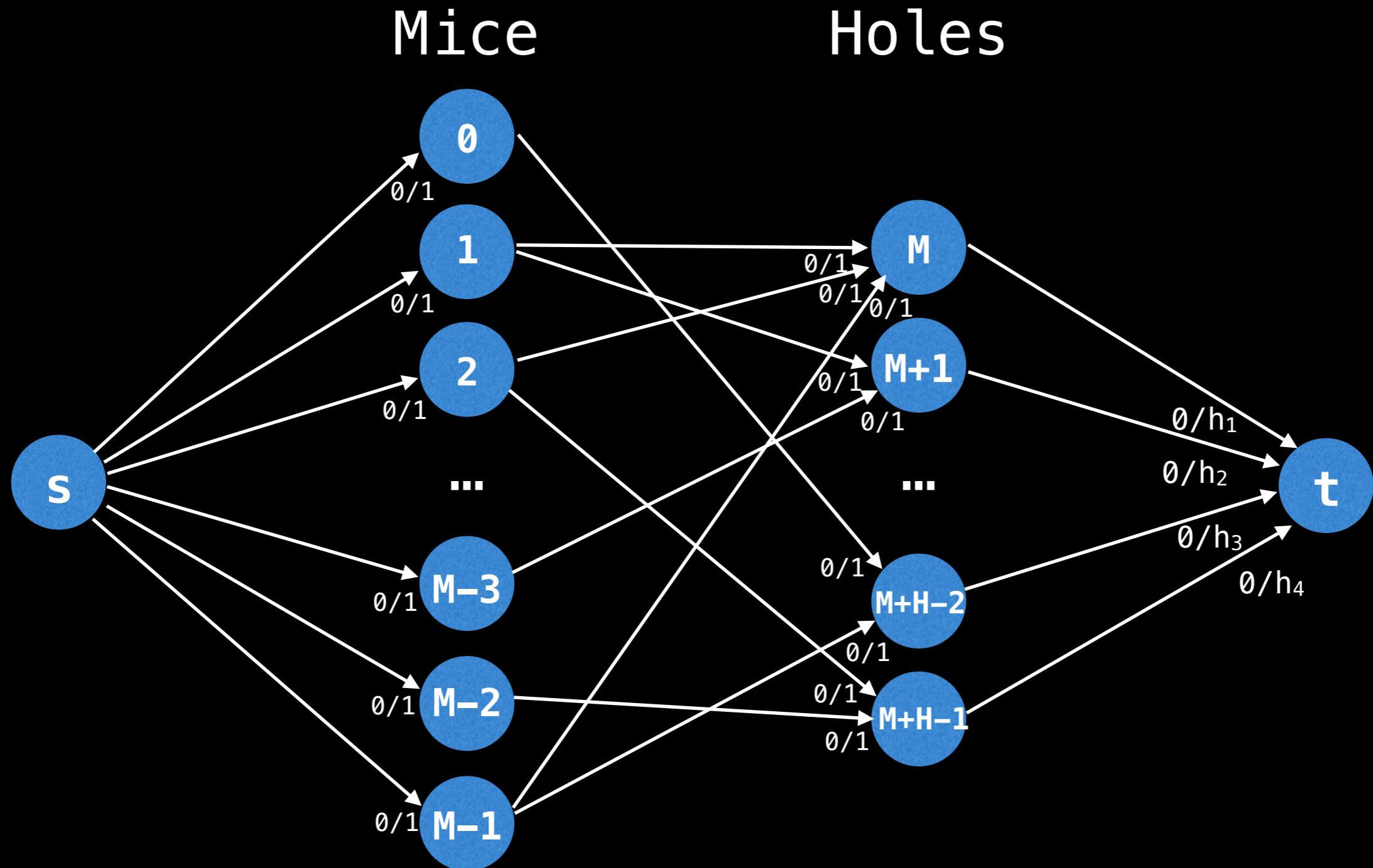
# Mice

# Holes



Connect an edge from each hole node to the sink with the capacity of the hole.

# Flow Graph Setup

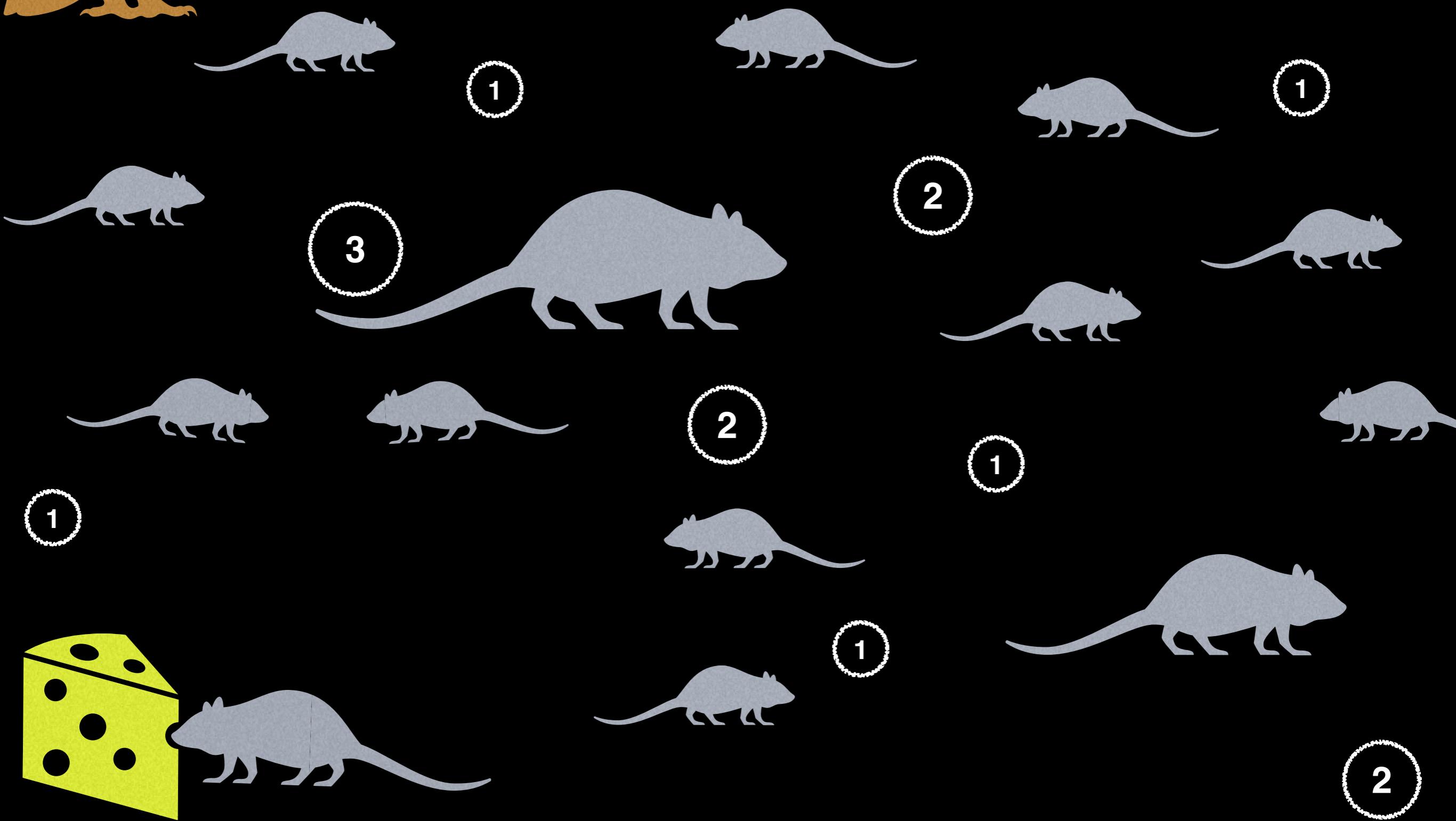


The problem has now been transformed into a maximum flow problem. Run any max-flow algorithm and the most mice that are safe is equal to the value of the max-flow.

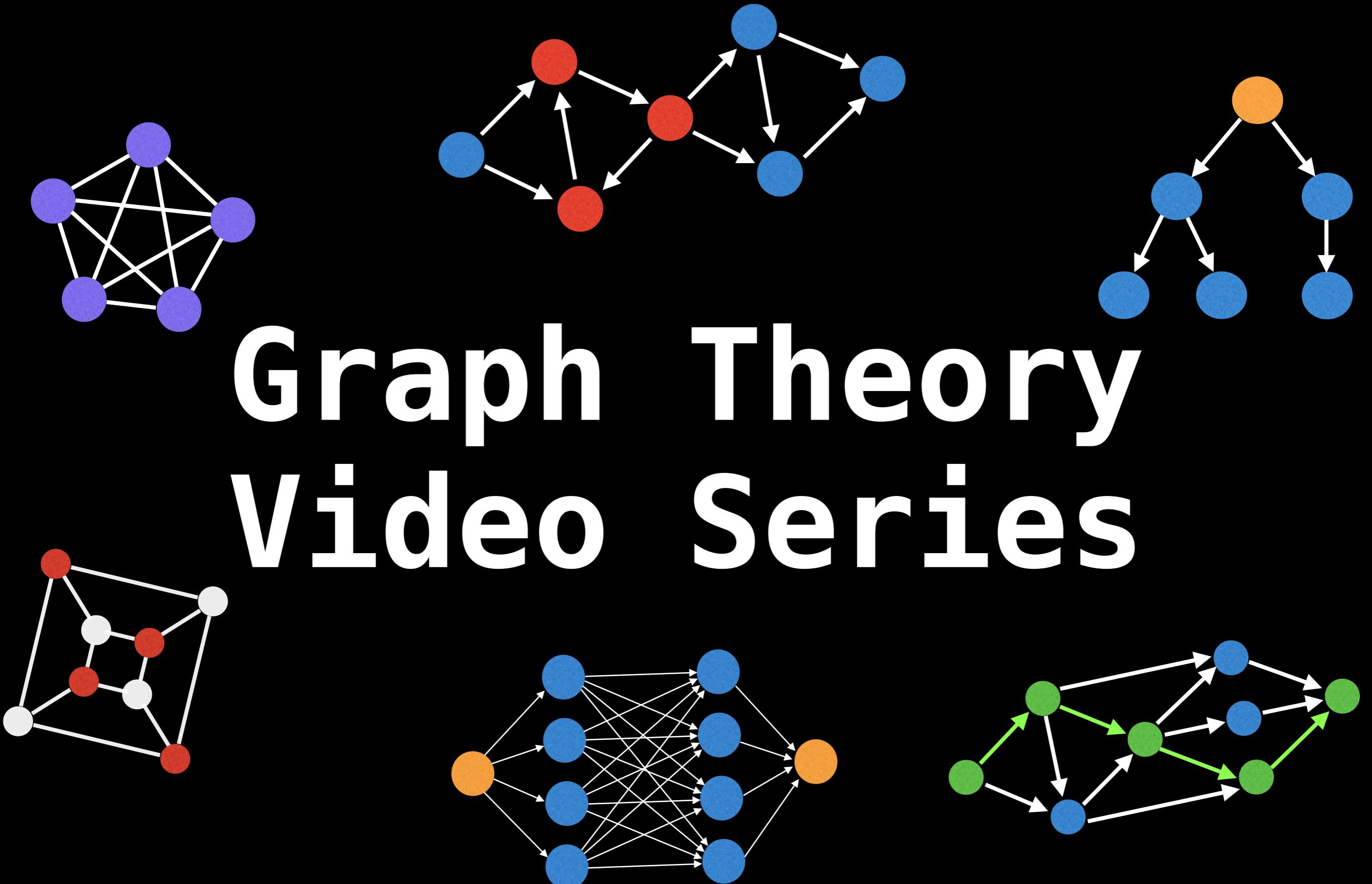


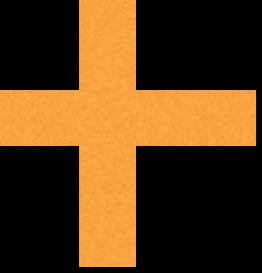


# Network Flow: Mice & Owls Problem



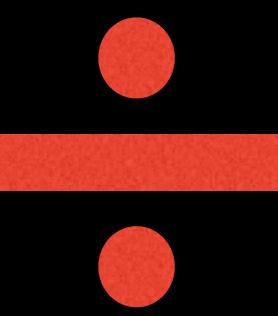
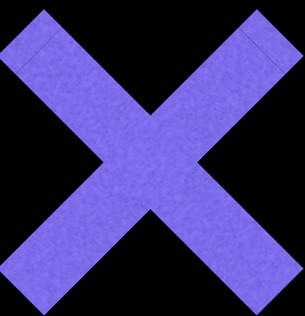
# Graph Theory Video Series

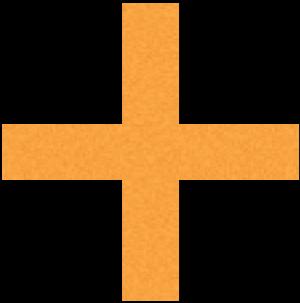




# Network Flow: elementary math

William Fiset



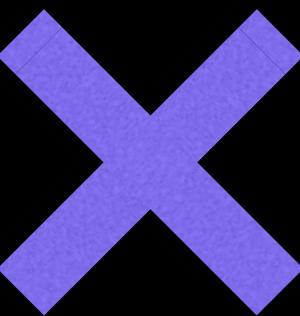


# Elementary Math



**Problem Statement:** Ellen is a math teacher who is preparing  $n$  ( $1 \leq n \leq 2500$ ) questions for her math exam. In each question, the students have to add (+), subtract (-) or multiply (\*) a pair of numbers.

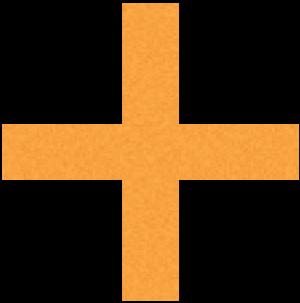
Ellen has already chosen the  $n$  pairs of numbers. All that remains is to decide for each pair which of the three possible operations the students should perform. To avoid students getting bored, Ellen wants to make sure that the  $n$  correct answers to her exam are all different.



Problem Link (in description):

<https://open.kattis.com/problems/elementarymath>



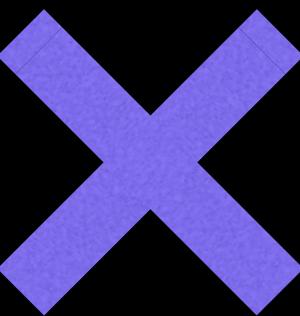


# Elementary Math



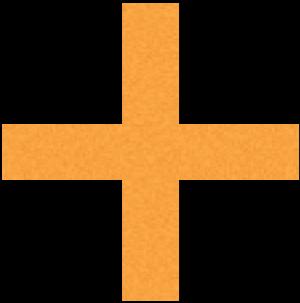
For each pair of numbers  $(a,b)$  in the same order as in the input, output a line containing a valid equation. Each equation should consist of five parts:  $a$ , one of the three operators,  $b$ , an equals sign ( $=$ ), and the result of the expression. All the  $n$  expression results must be different.

If there are multiple valid answers, output any of them. If there is no valid answer, output a single line with the string “impossible” instead.



Problem Link (in description):  
<https://open.kattis.com/problems/elementarymath>





# Elementary Math

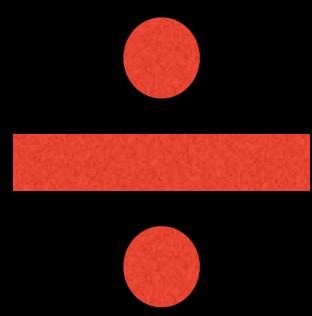
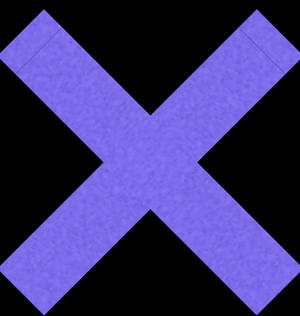


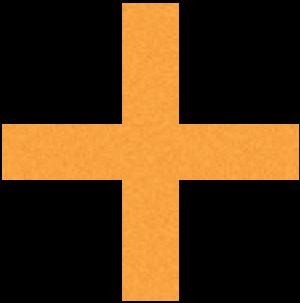
$1 \square 5 = \boxed{\phantom{00}}$

$3 \square 3 = \boxed{\phantom{00}}$

$4 \square 5 = \boxed{\phantom{00}}$

$-1 \square -6 = \boxed{\phantom{00}}$





# Elementary Math

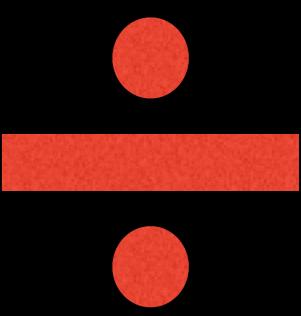
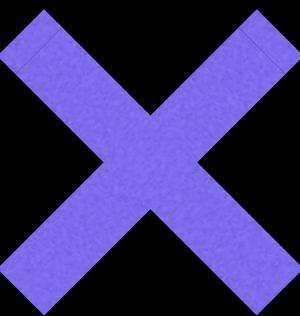


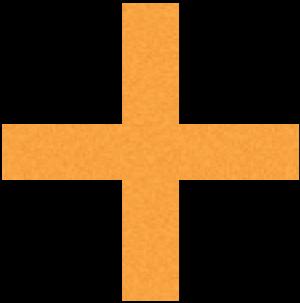
$$1 \times 5 = \boxed{5}$$

$$3 + 3 = \boxed{6}$$

$$4 - 5 = \boxed{-1}$$

$$-1 \times -6 = \boxed{6}$$





# Elementary Math



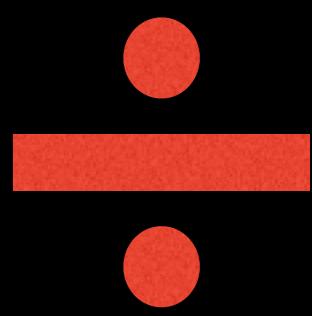
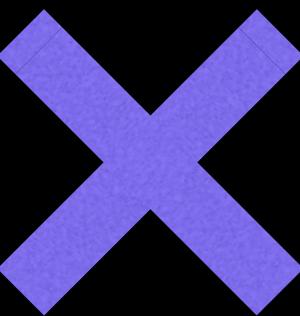
$$1 \times 5 = 5$$

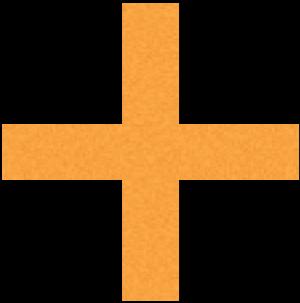
$$3 + 3 = 6$$

Answers must  
be unique!

$$4 - 5 = -1$$

$$-1 \times -6 = 6$$





# Elementary Math

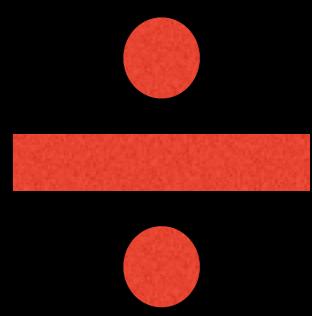
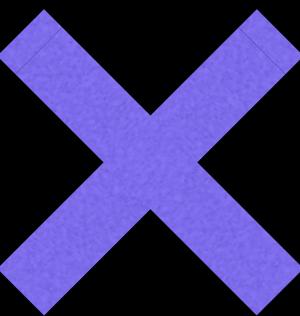


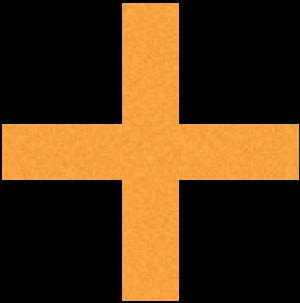
$$1 \boxed{-} 5 = \boxed{-4}$$

$$3 \boxed{-} 3 = \boxed{0}$$

$$4 \boxed{+} 5 = \boxed{9}$$

$$-1 \boxed{*} -6 = \boxed{6}$$





# Elementary Math



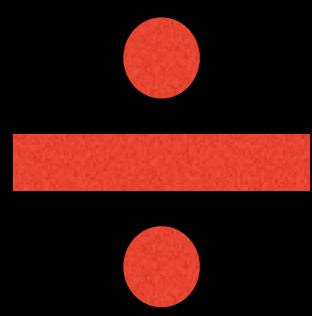
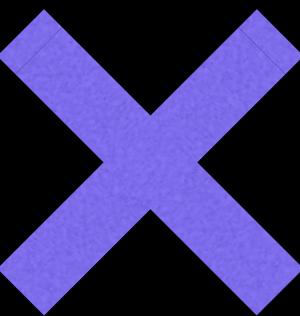
$1 \boxed{-} 5 = \boxed{-4}$

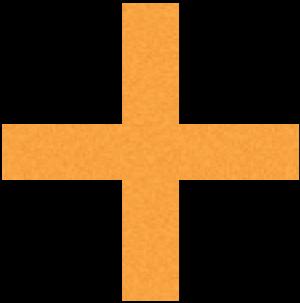
$3 \boxed{-} 3 = \boxed{0}$



$4 \boxed{+} 5 = \boxed{9}$

$-1 \boxed{*} -6 = \boxed{6}$





# Elementary Math



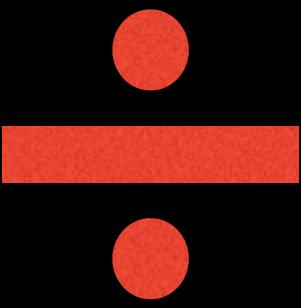
$1 \square 2 = \boxed{\phantom{00}}$

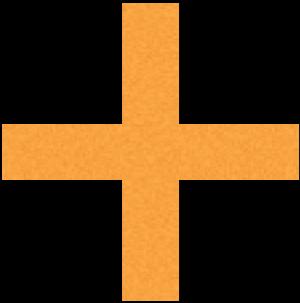
$2 \square 1 = \boxed{\phantom{00}}$

$1 \square 2 = \boxed{\phantom{00}}$

$2 \square 1 = \boxed{\phantom{00}}$

$1 \square 2 = \boxed{\phantom{00}}$





# Elementary Math



$1 \square 2 = \boxed{\phantom{00}}$

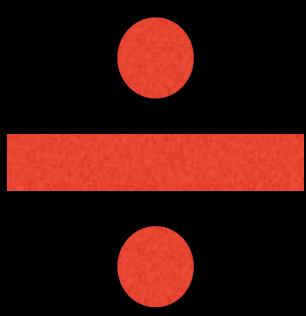
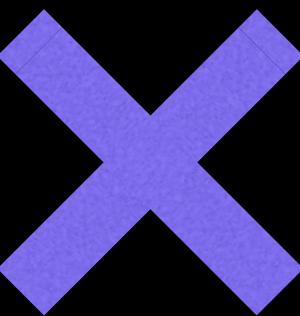
$2 \square 1 = \boxed{\phantom{00}}$

No solution exists!

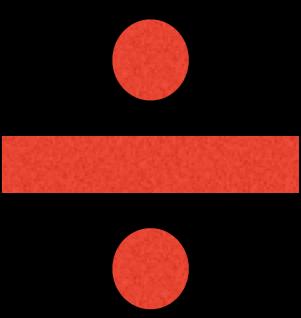
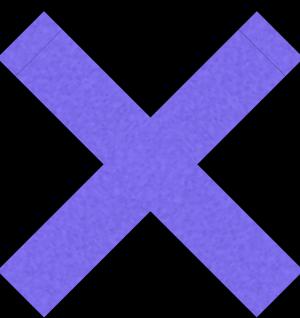
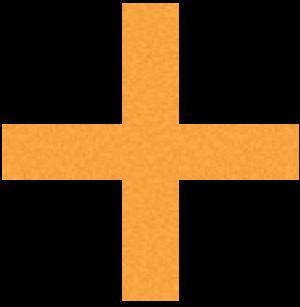
$1 \square 2 = \boxed{\phantom{00}} \quad \times$

$2 \square 1 = \boxed{\phantom{00}}$

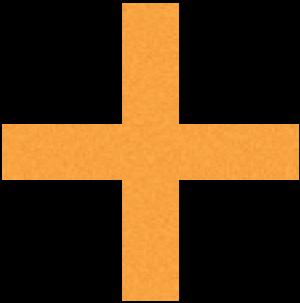
$1 \square 2 = \boxed{\phantom{00}}$



# Elementary Math



This problem presents itself as a network flow problem even though it might not be obvious at first. Take a moment and attempt to set up a flow graph that can solve this problem.



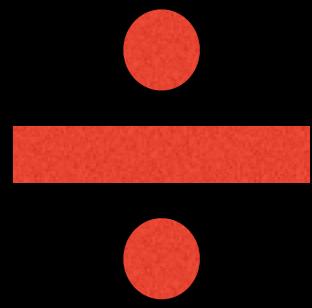
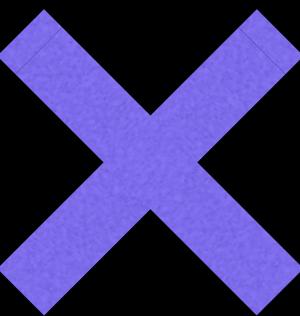
# Elementary Math

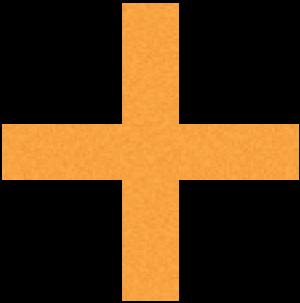


This problem presents itself as a network flow problem even though it might not be obvious at first. Take a moment and attempt to set up a flow graph that can solve this problem.

Questions to ask yourself:

Q: Is there a way that this problem can be simplified as a bipartite graph?





# Elementary Math

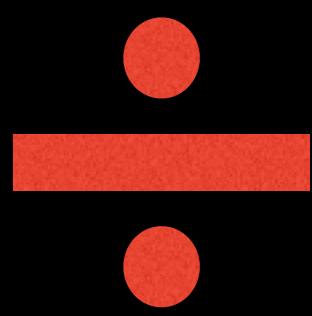
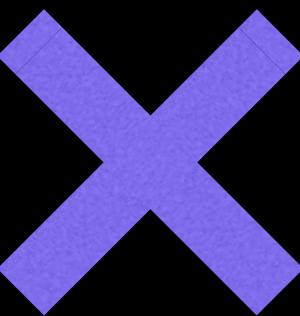


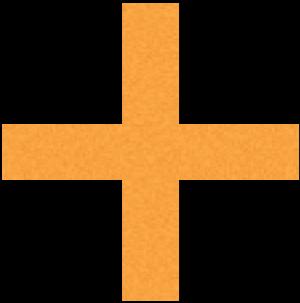
This problem presents itself as a network flow problem even though it might not be obvious at first. Take a moment and attempt to set up a flow graph that can solve this problem.

Questions to ask yourself:

Q: Is there a way that this problem can be simplified as a bipartite graph?

Q: How do I detect an impossible set of pairs?





# Elementary Math



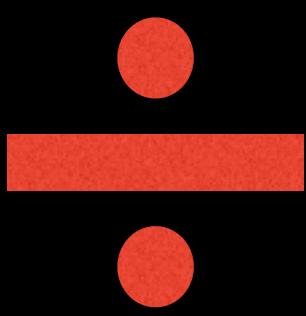
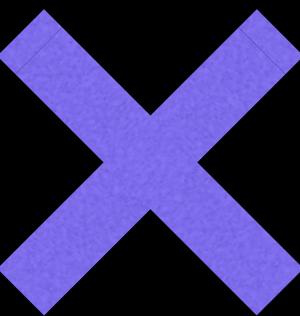
This problem presents itself as a network flow problem even though it might not be obvious at first. Take a moment and attempt to set up a flow graph that can solve this problem.

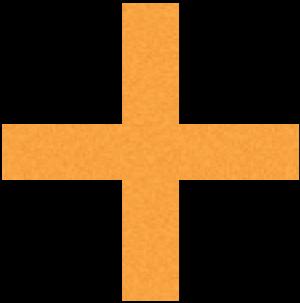
Questions to ask yourself:

Q: Is there a way that this problem can be simplified as a bipartite graph?

Q: How do I detect an impossible set of pairs?

Q: How do I handle multiple repeated pairs?





# Elementary Math



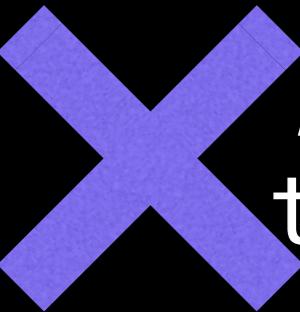
This problem presents itself as a network flow problem even though it might not be obvious at first. Take a moment and attempt to set up a flow graph that can solve this problem.

Questions to ask yourself:

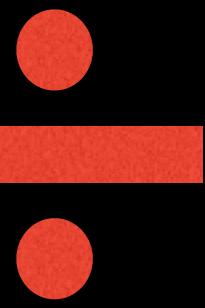
Q: Is there a way that this problem can be simplified as a bipartite graph?

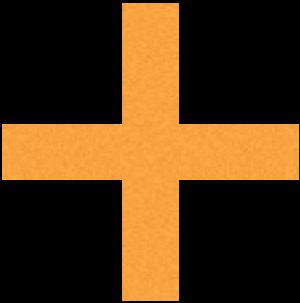
Q: How do I detect an impossible set of pairs?

Q: How do I handle multiple repeated pairs?



A: This slide deck explains the first two, the third is left as an exercise.

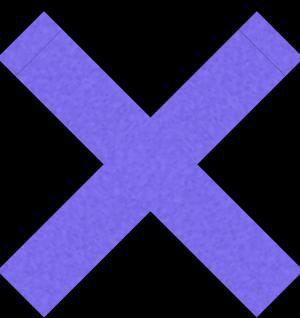
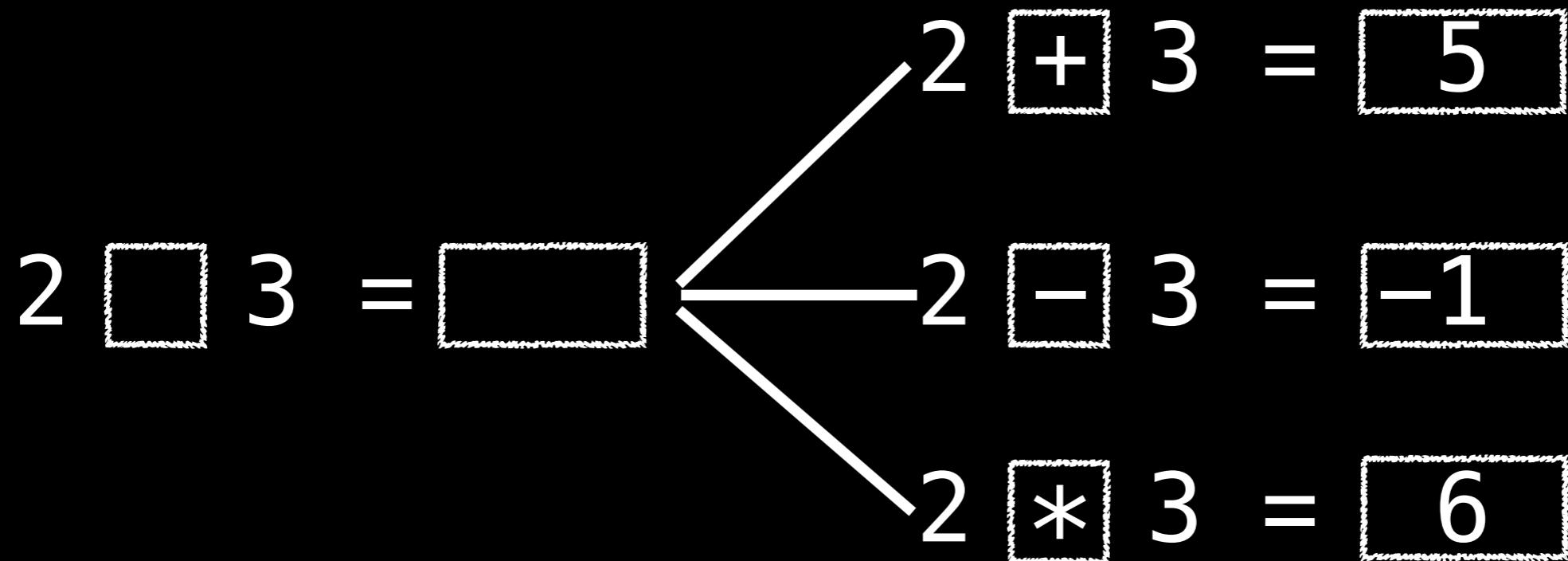




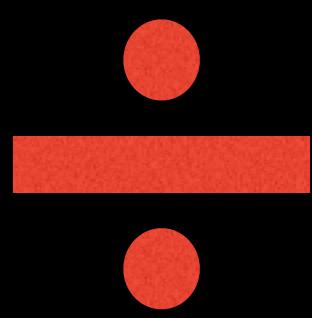
# Elementary Math

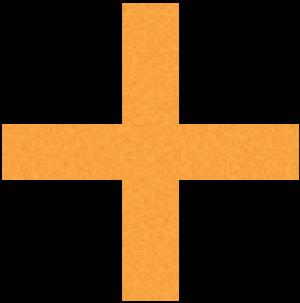


Key realization: For every pair, at most three unique solutions are produced.



This makes it easy to formulate the flow graph as a bipartite graph with input pairs on one side and solutions on the other.





# Elementary Math



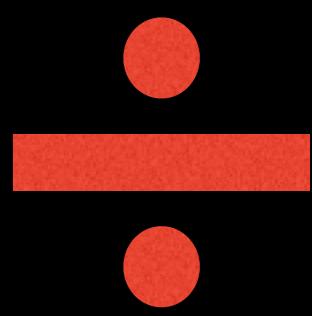
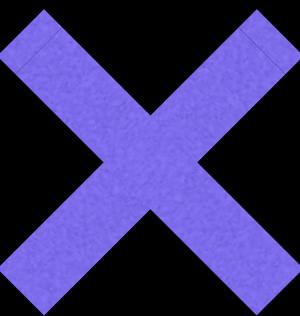
Let's see an example of how to set up the flow graph for the following pairs:

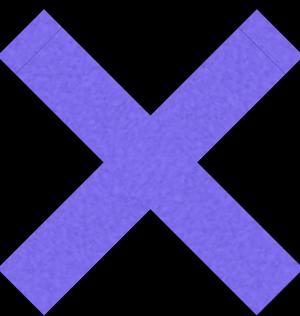
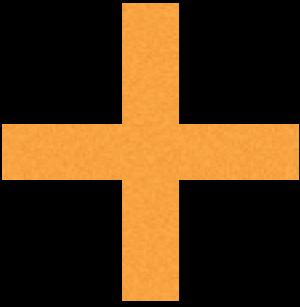
$1 \square 5 = \boxed{\phantom{00}}$

$3 \square 3 = \boxed{\phantom{00}}$

$-1 \square -6 = \boxed{\phantom{00}}$

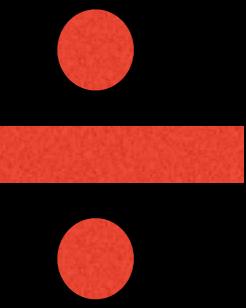
$2 \square 2 = \boxed{\phantom{00}}$



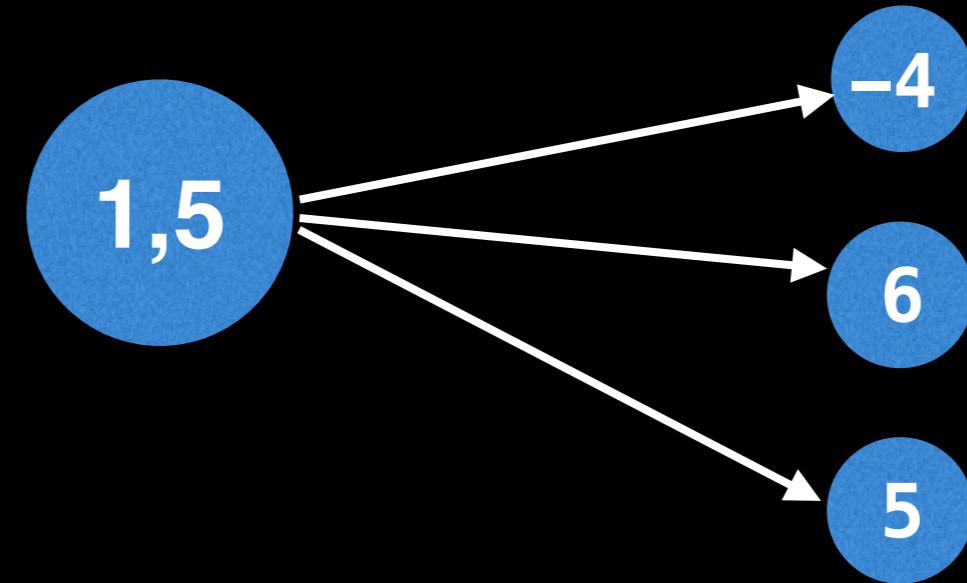


(a, b) pairs

Answers



+



$$1 - 5 = -4$$

$$1 + 5 = 6$$

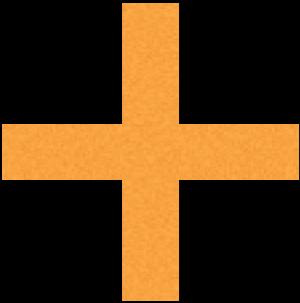
$$1 * 5 = 5$$

X

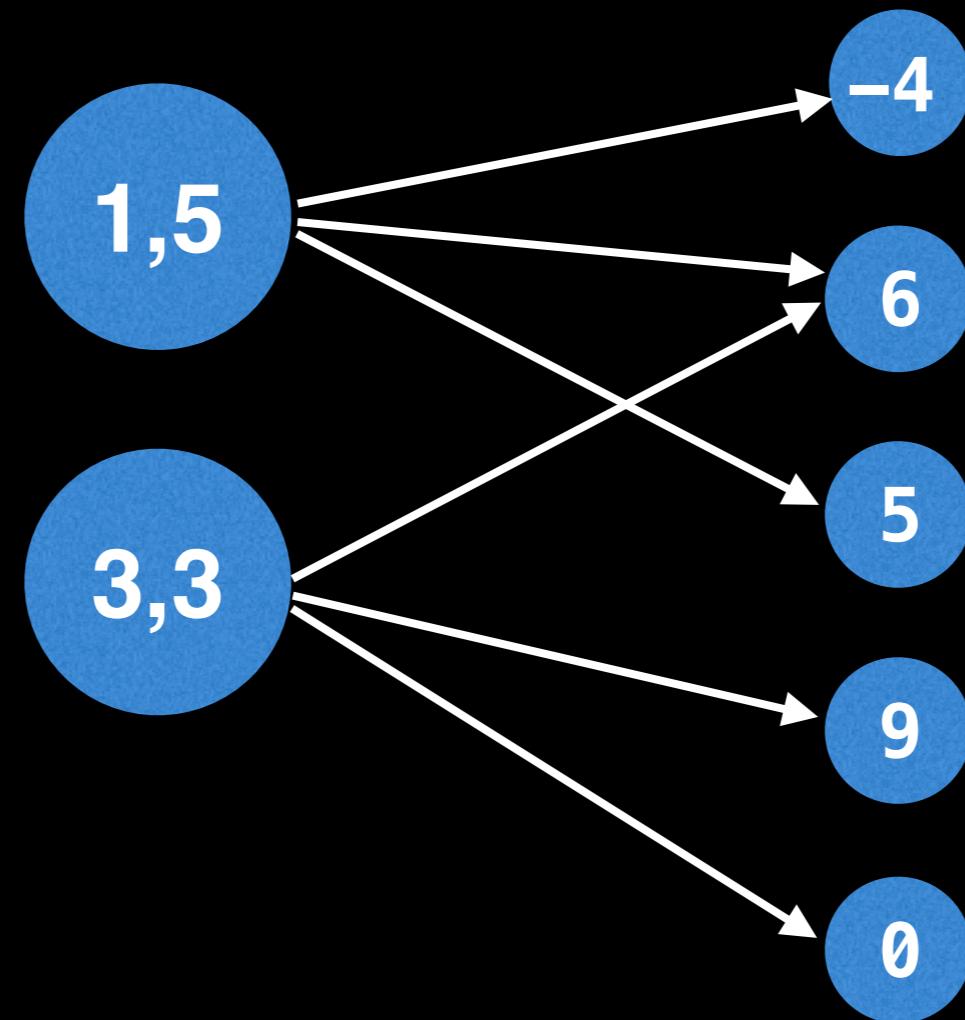
(a, b) pairs

Answers

÷

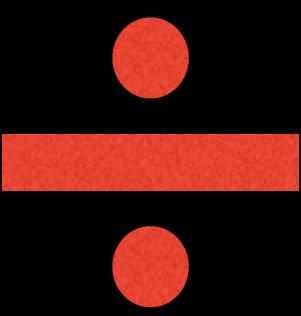
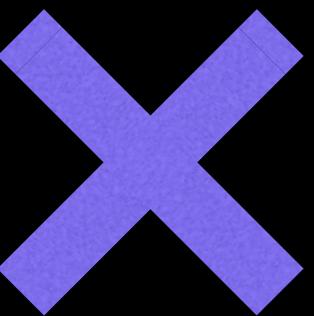


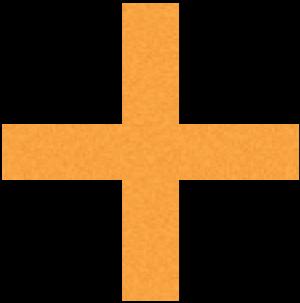
$$\begin{aligned} 3 - 3 &= 0 \\ 3 + 3 &= 6 \\ 3 * 3 &= 9 \end{aligned}$$



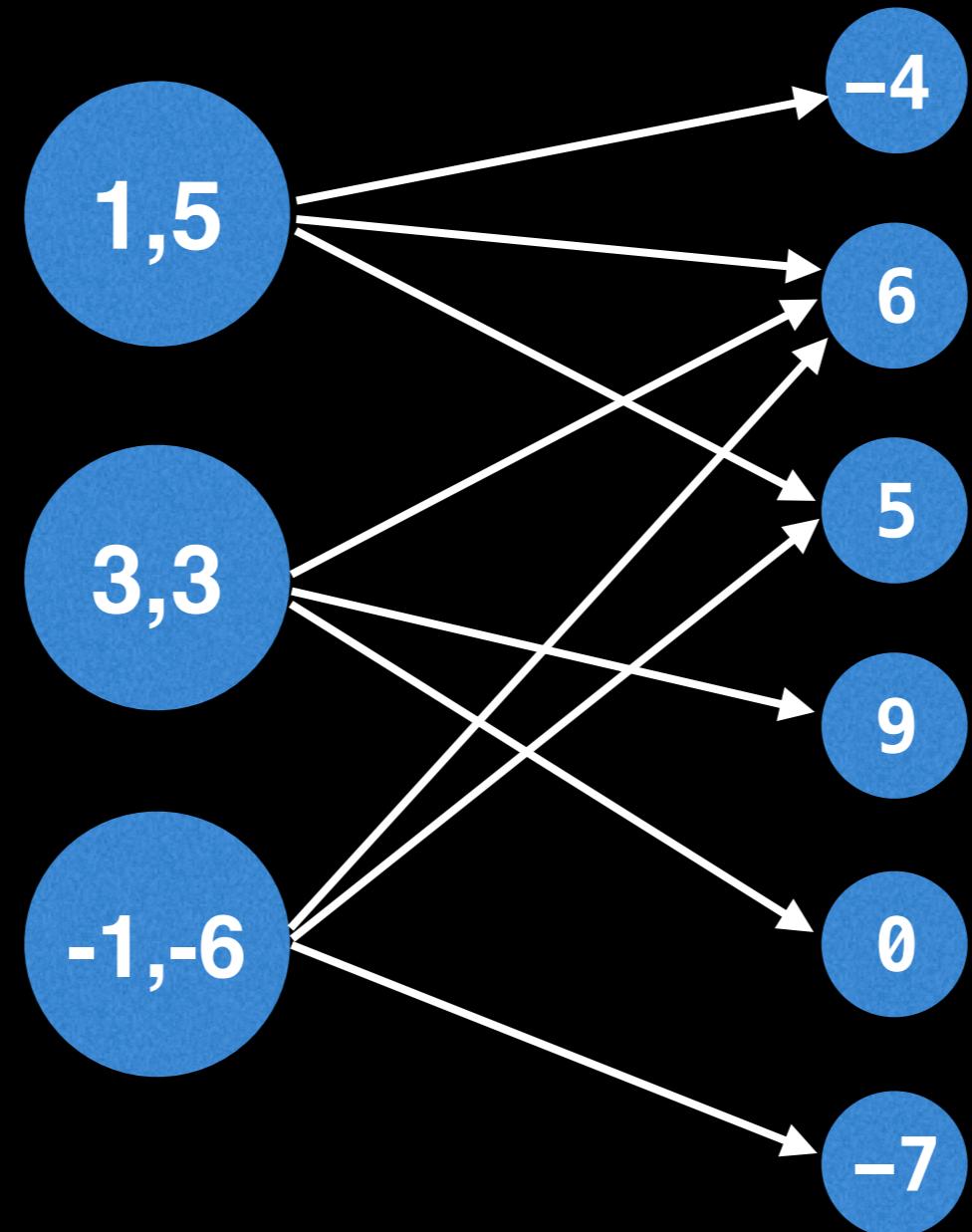
(a, b) pairs

Answers



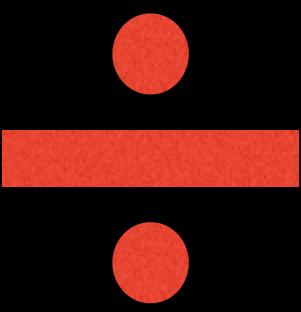
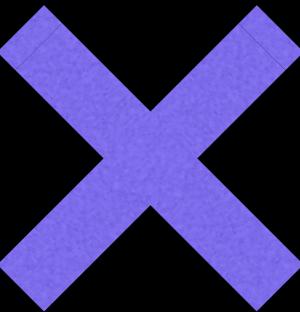


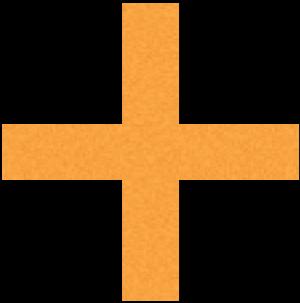
$$\begin{aligned}-1 - -6 &= 5 \\-1 + -6 &= -7 \\-1 * -6 &= 6\end{aligned}$$



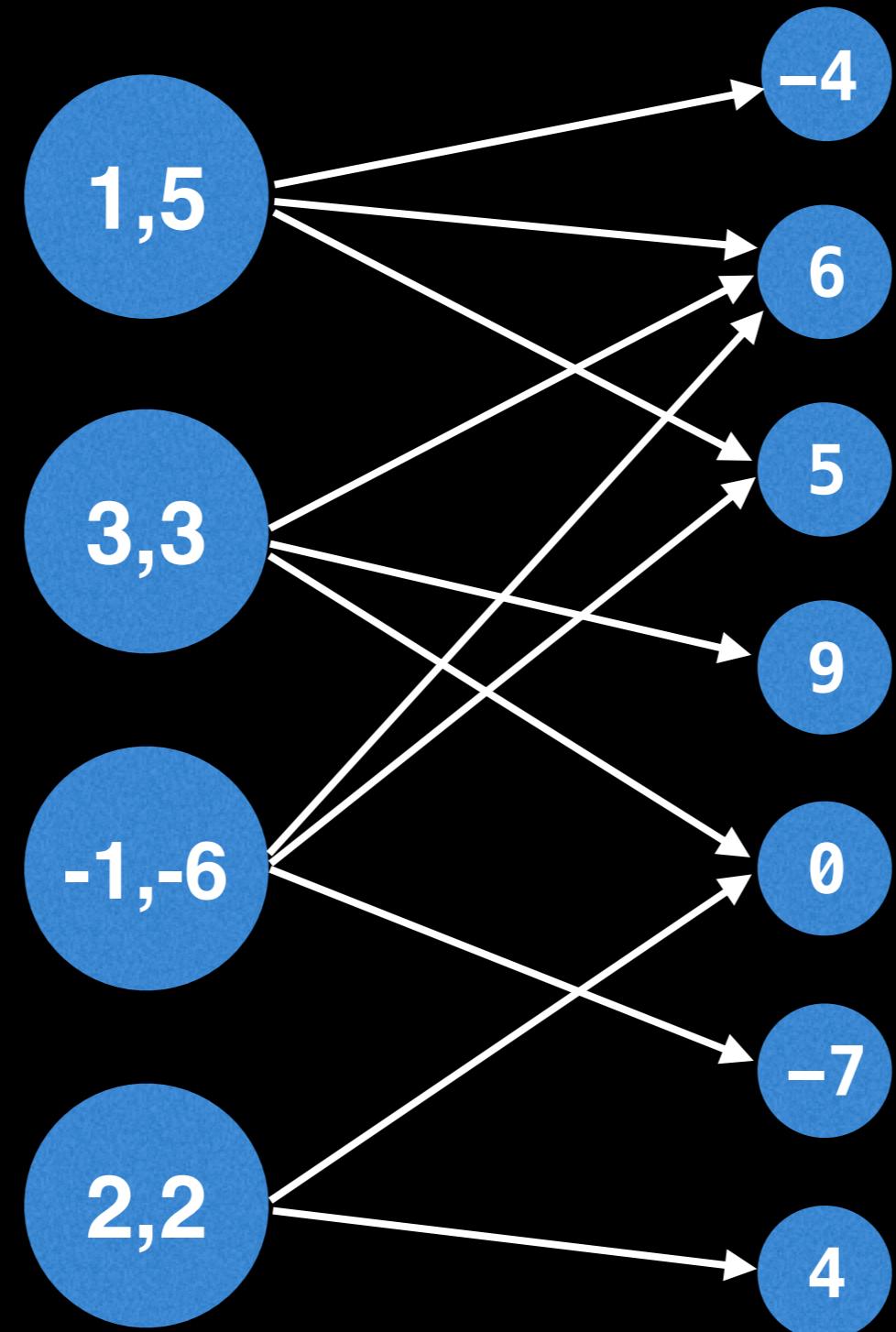
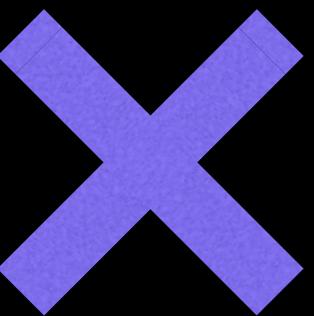
(a, b) pairs

Answers



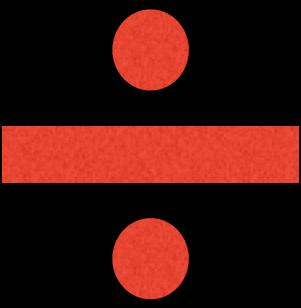


$$\begin{aligned} 2 - 2 &= 0 \\ 2 + 2 &= 4 \\ 2 * 2 &= 4 \end{aligned}$$

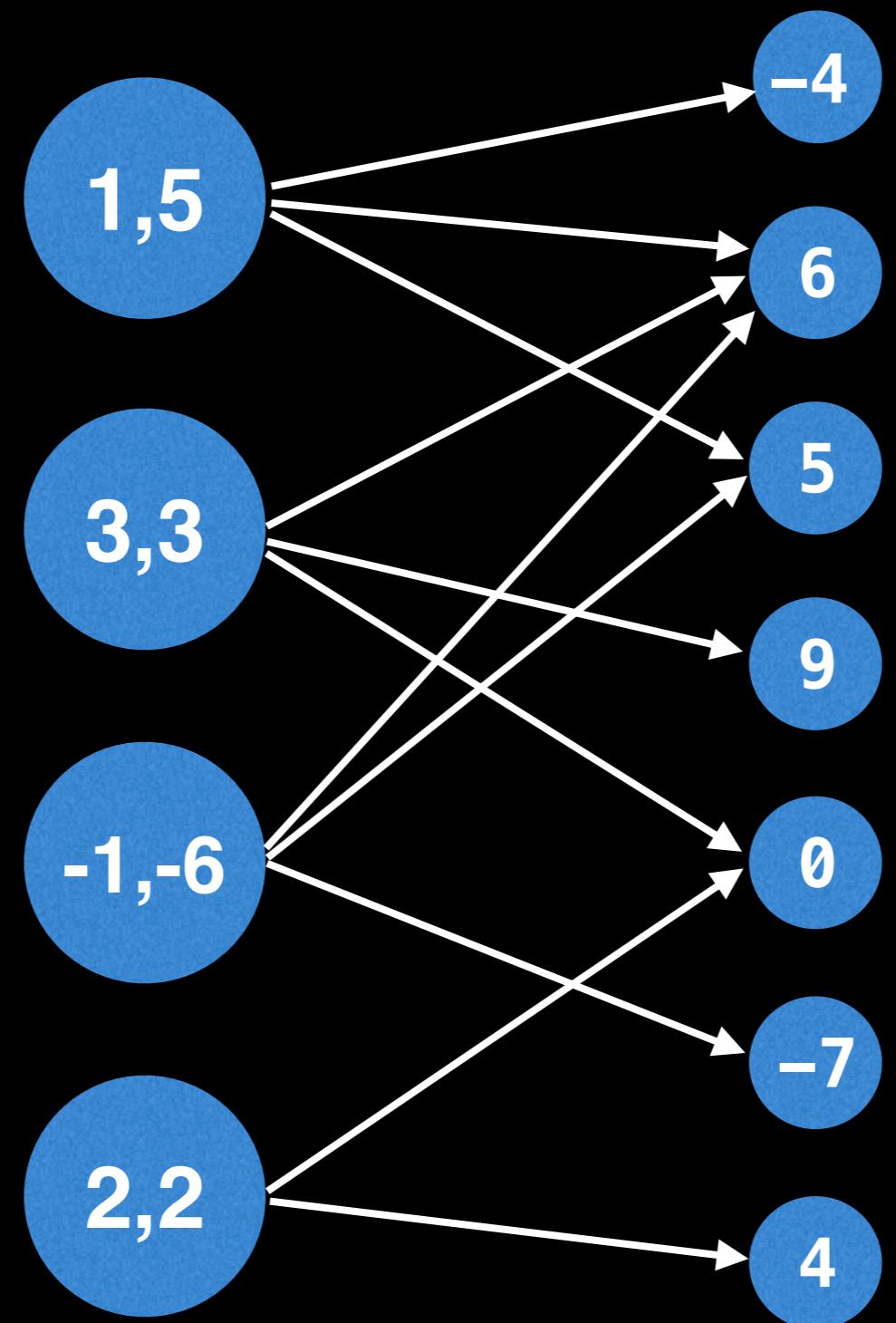


(a, b) pairs

Answers



+

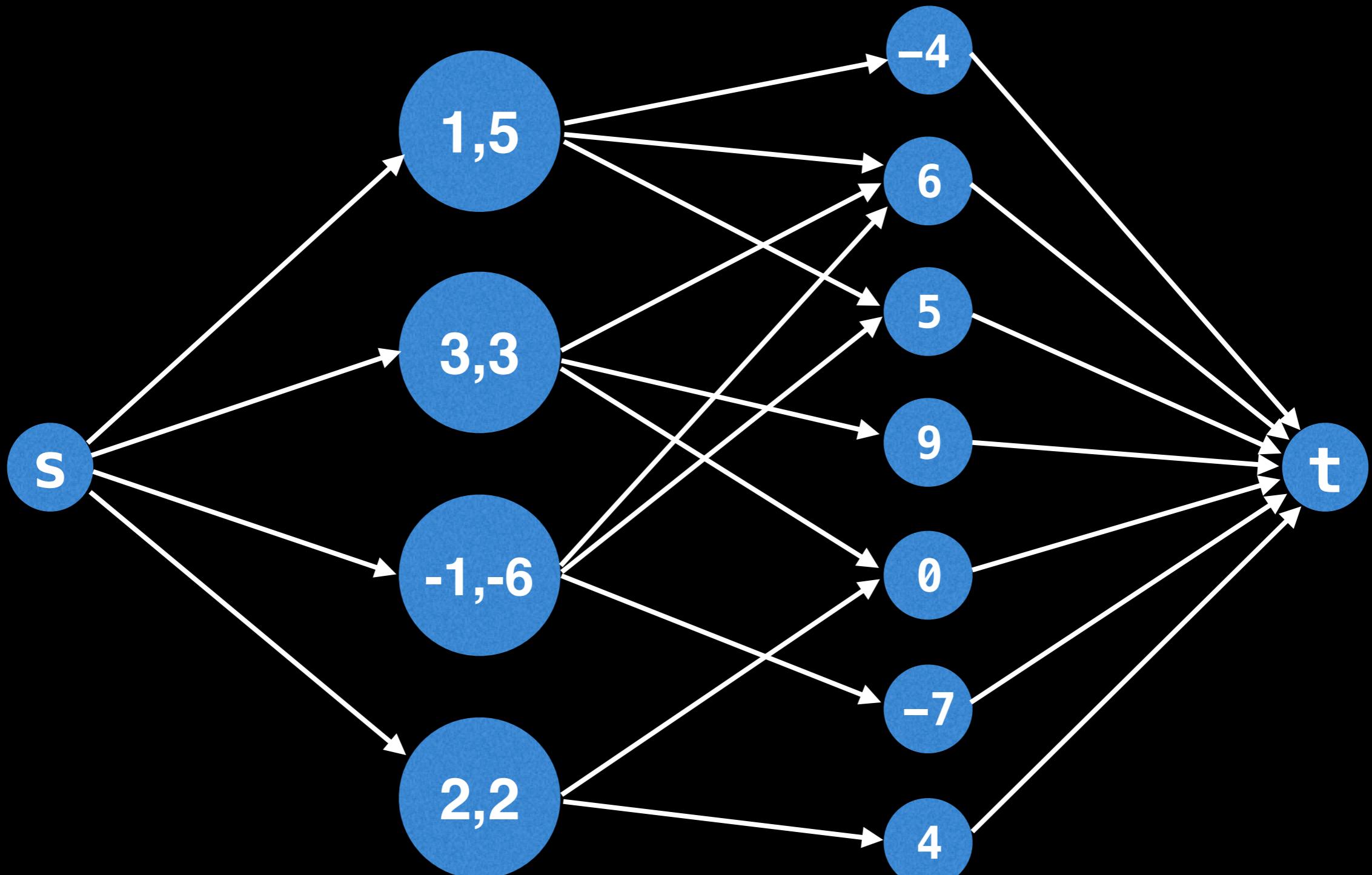


$(a, b)$  pairs

Answers

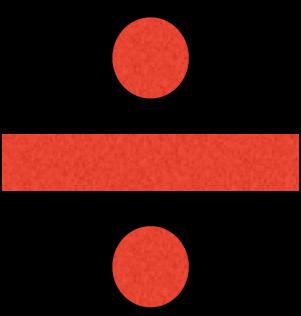
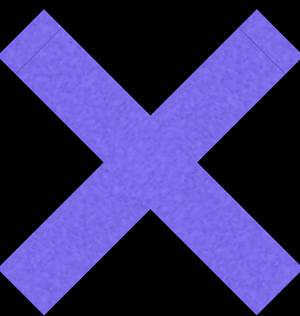
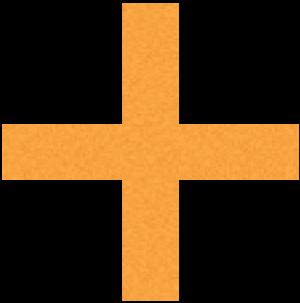
X

÷

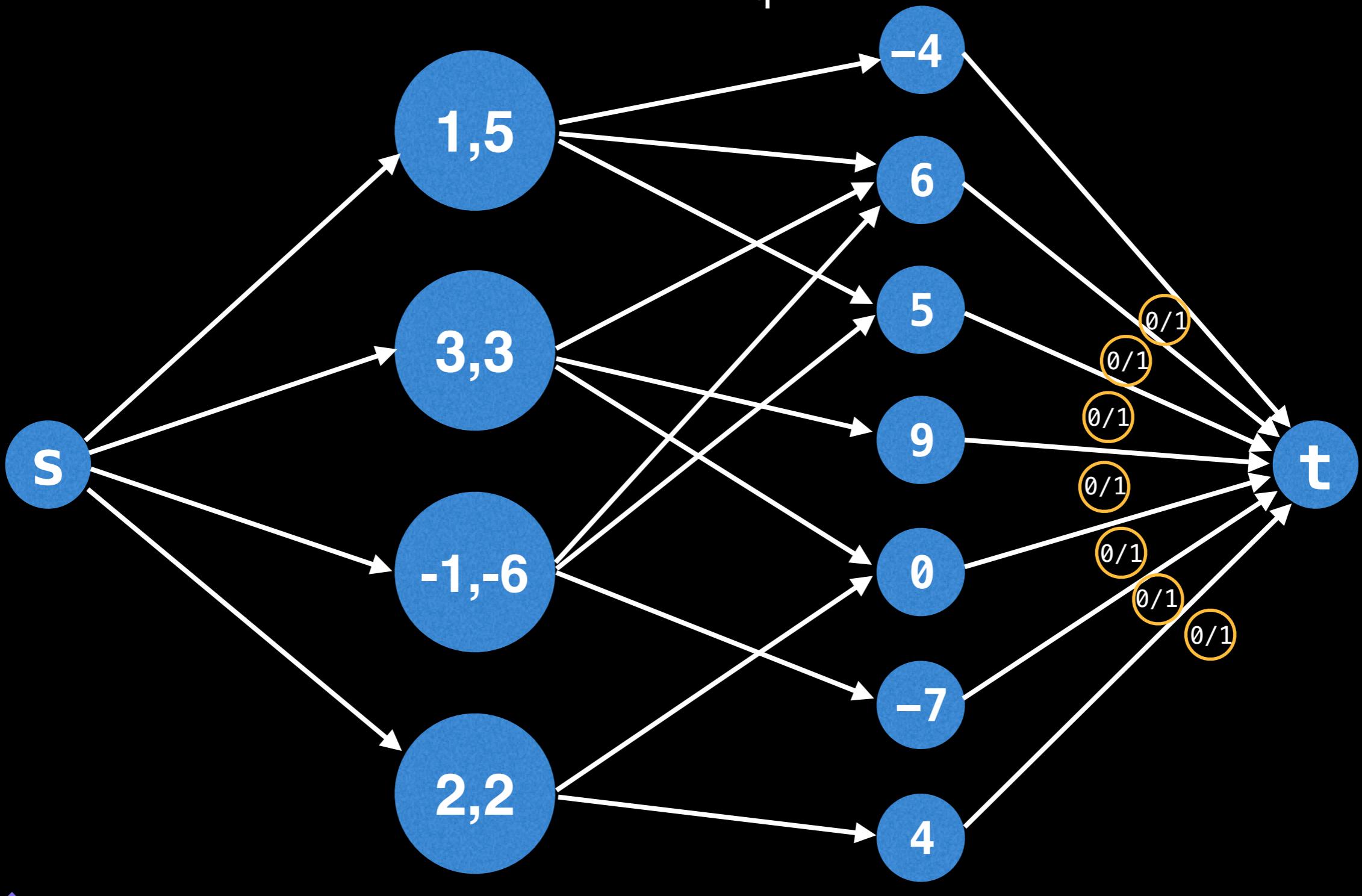


(a, b) pairs

Answers



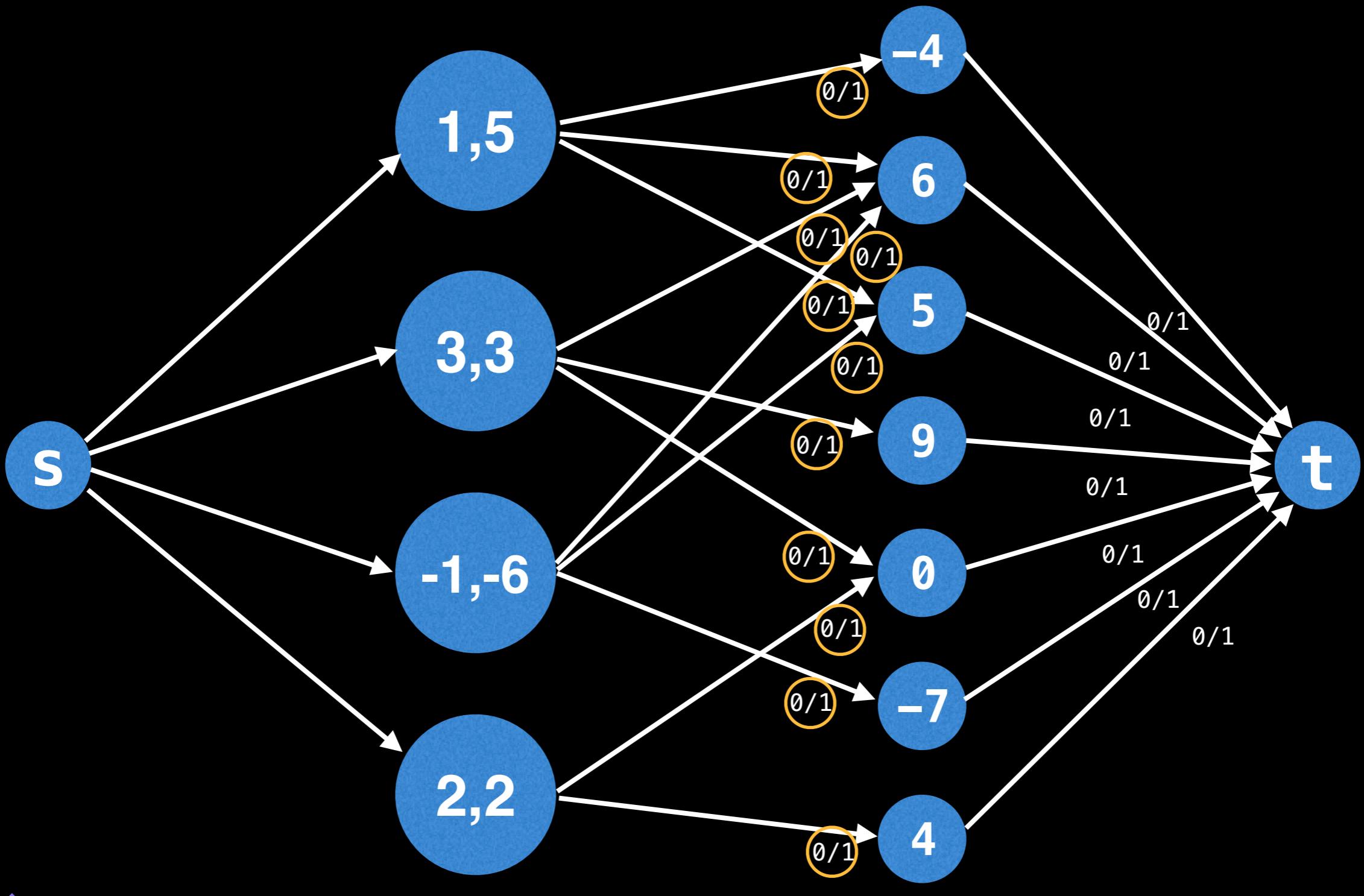
Capacities from answer nodes to sink should have a capacity of one since answers need to be unique.



$(a,b)$  pairs

Answers

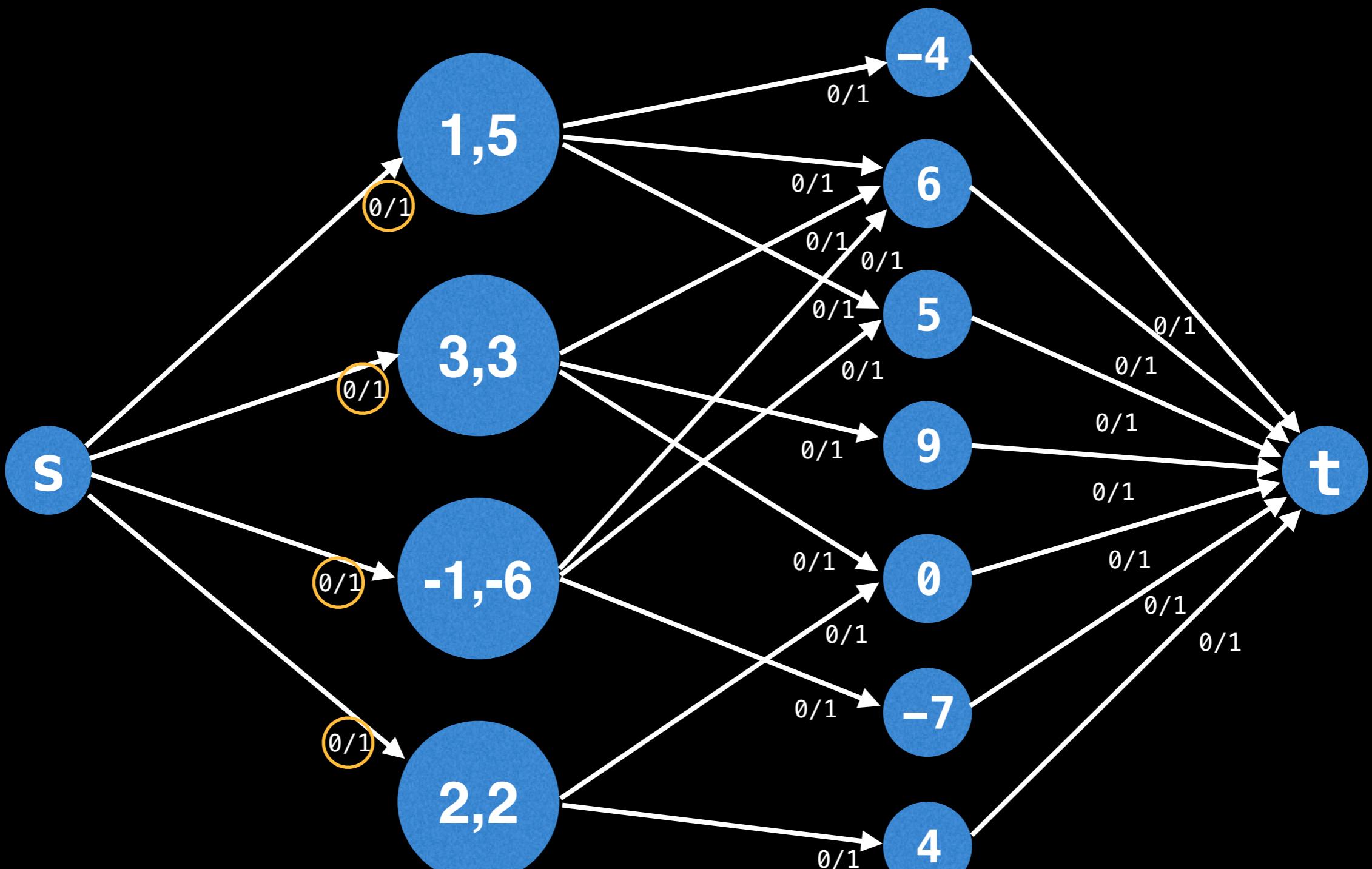
Capacities from input pairs to answers also have capacity one since only one of '+', '-' or '\*' can be matched with an answer.



(a,b) pairs

Answers

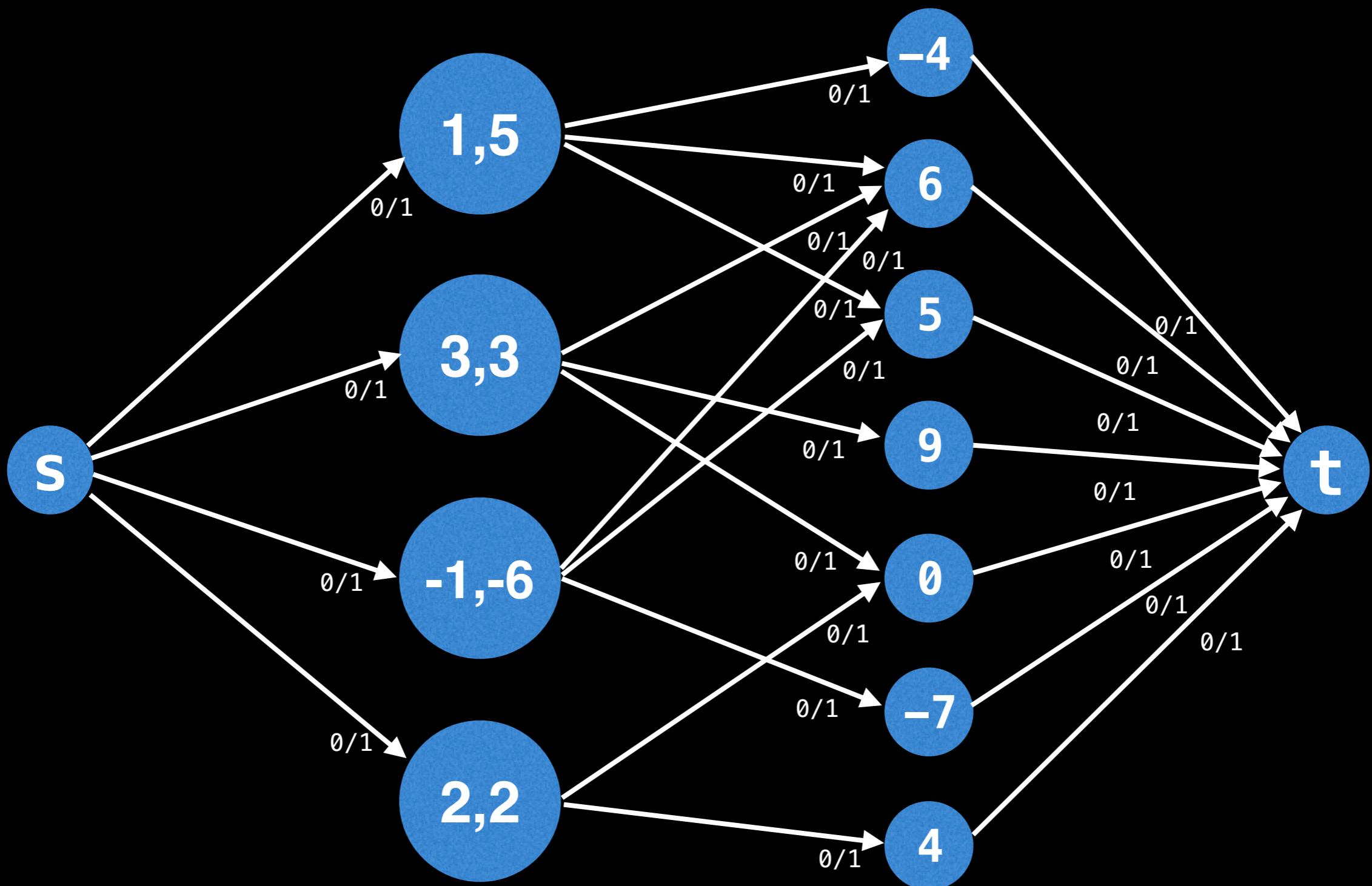
**+** Capacities from source to input pairs should reflect the **frequency of the input pair**. In this example, all frequencies are 1, but as we know, this isn't always the case.



(a, b) pairs

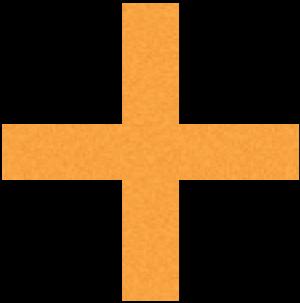
Answers

Once the flow graph is set up, run a max-flow algorithm on it!

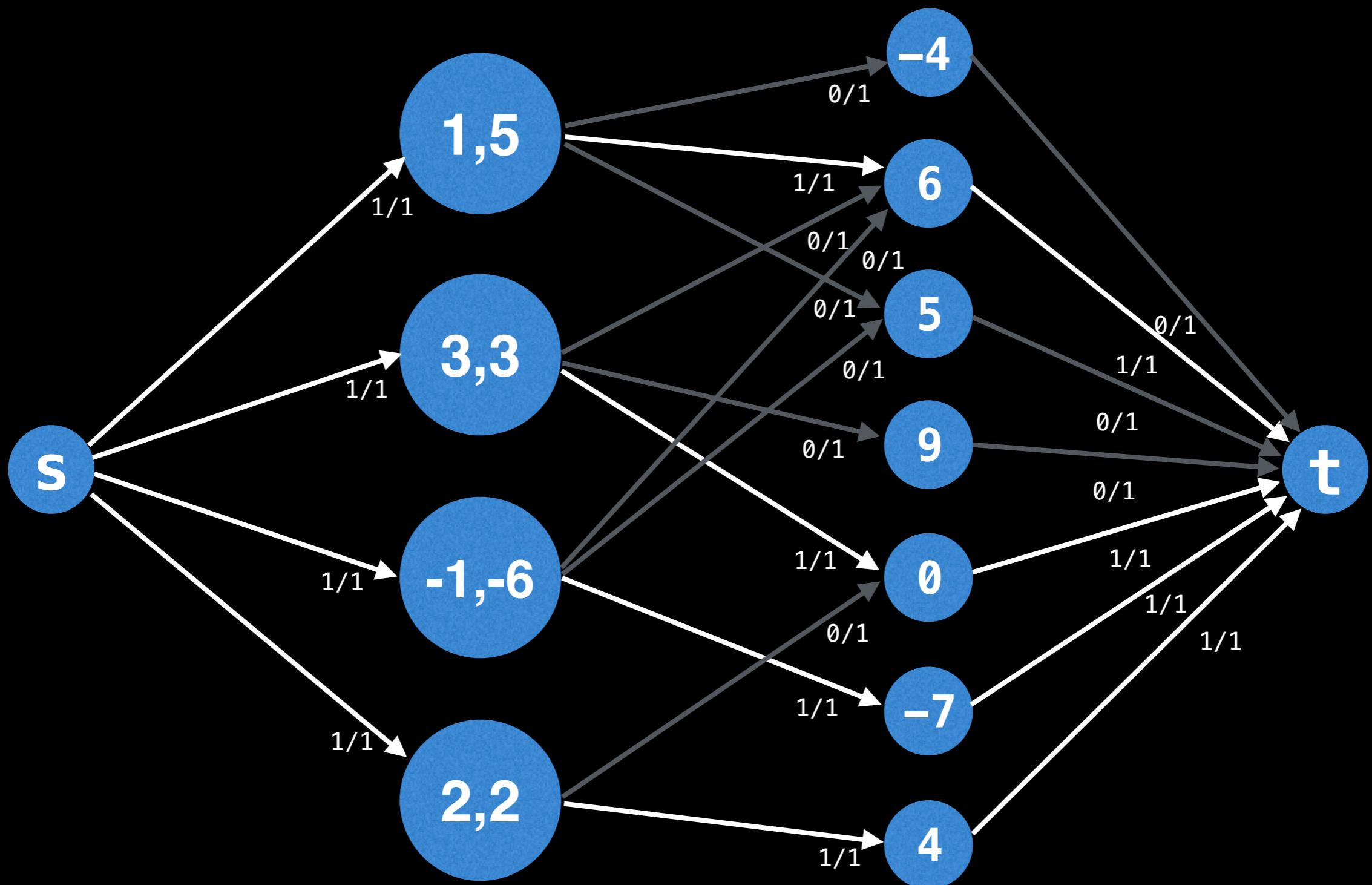


( $a, b$ ) pairs

Answers

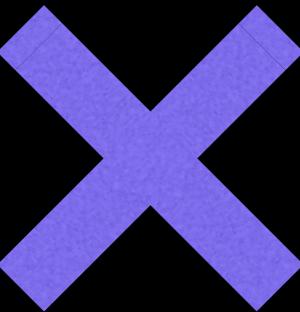


Once the flow graph is set up, run a max-flow algorithm on it!

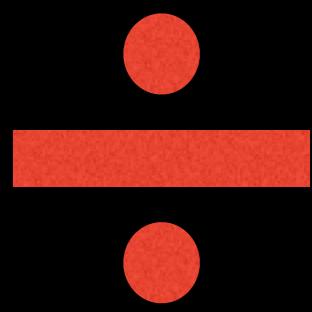


( $a, b$ ) pairs

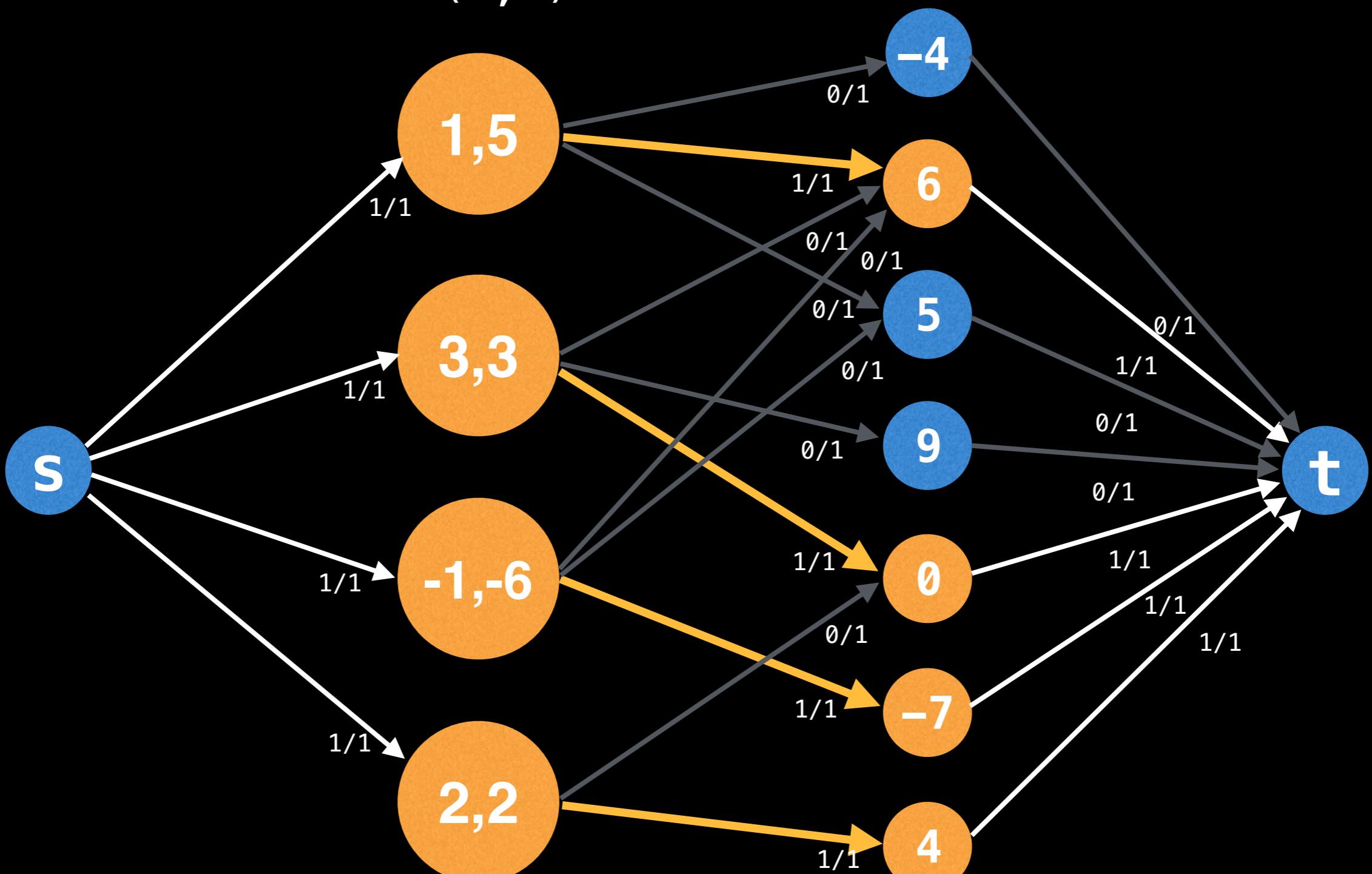
Answers



NOTE: For simplicity, the residual edges are not shown



Every edge in the middle with one unit of flow represents a matching from a pair  $(a, b)$  to its answer.

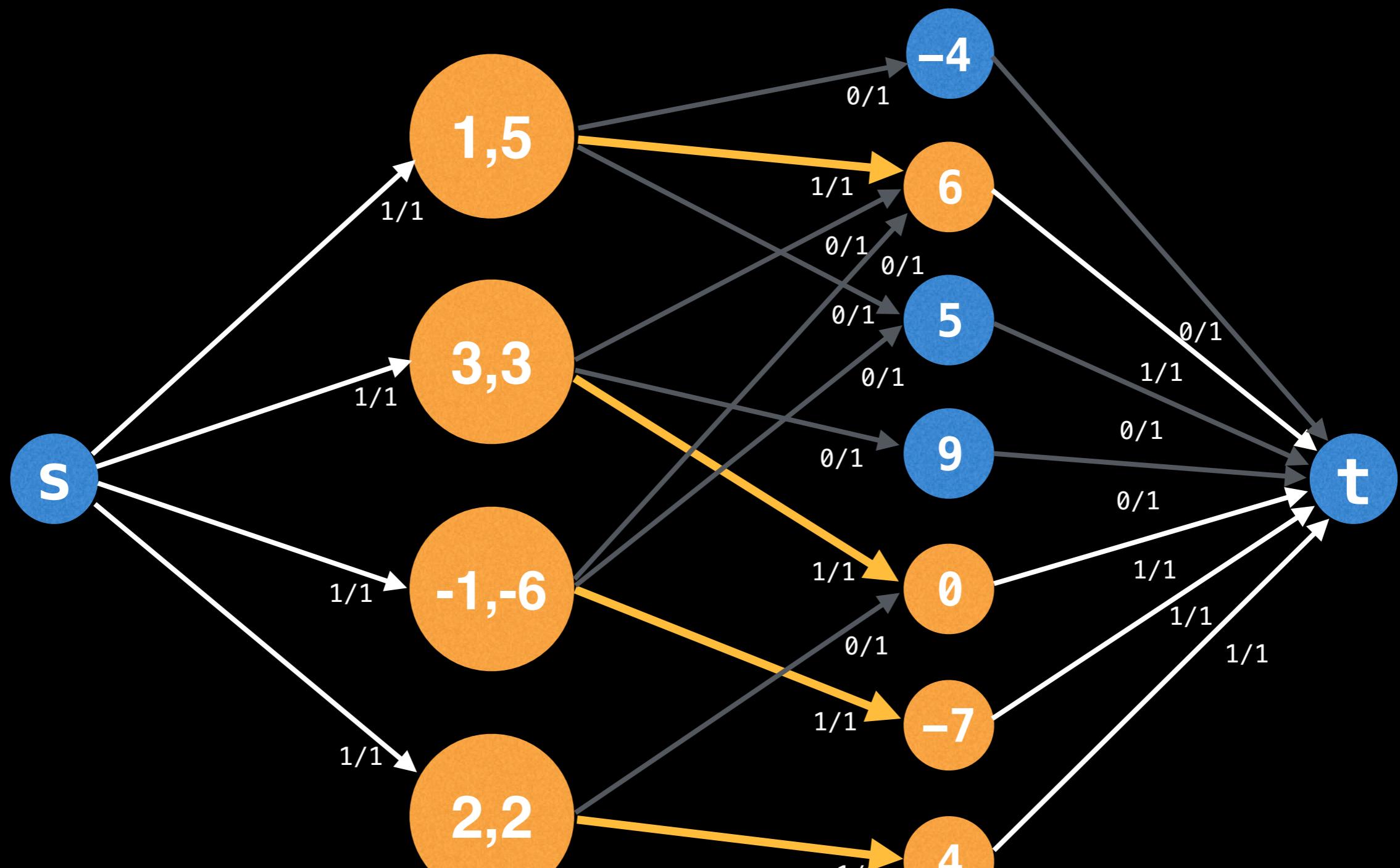


$(a, b)$  pairs

Answers

NOTE: For simplicity, the residual edges are not shown

We can even deduce the operator used for each matching (needed for final output) by trying which of (+, -, \*) results in the answer matched the one on the right.

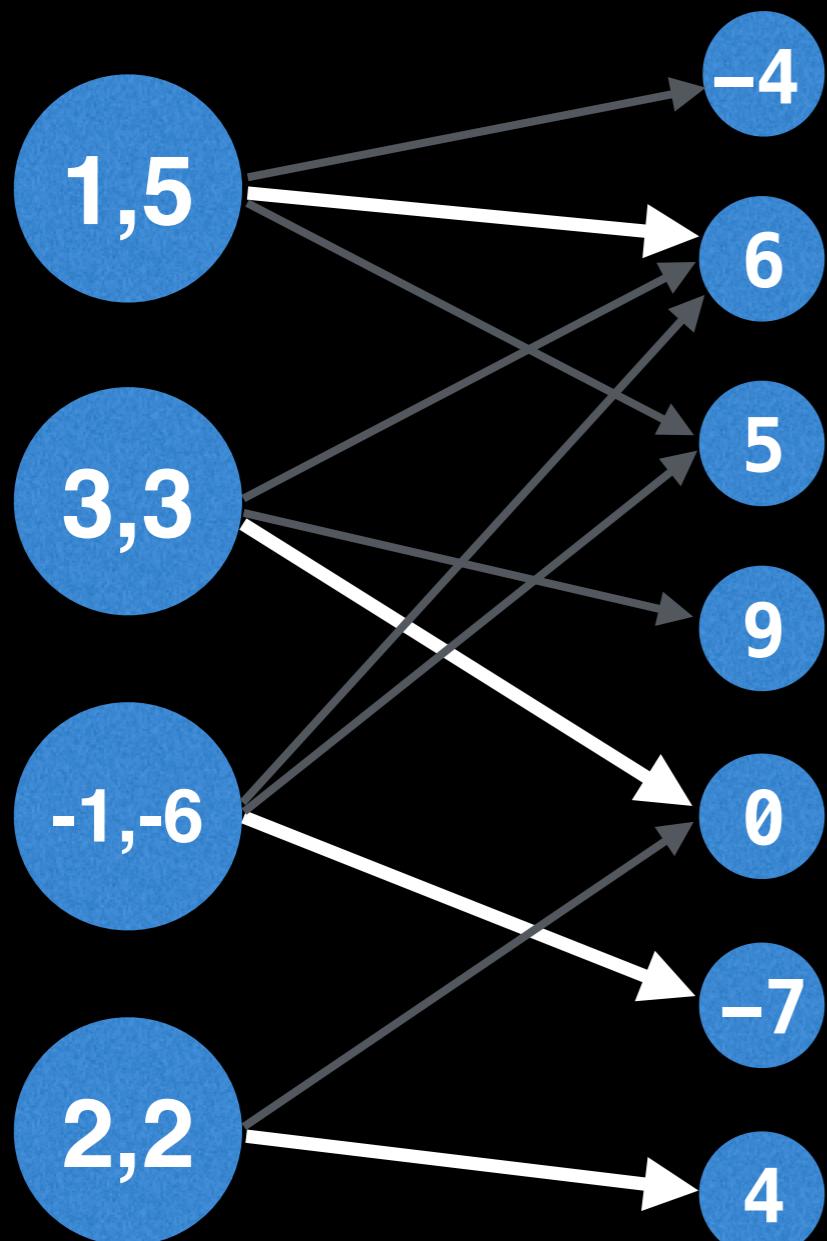


(a, b) pairs

Answers

NOTE: For simplicity, the residual edges are not shown

+



$1 \square 5 = \boxed{\phantom{00}}$

$3 \square 3 = \boxed{\phantom{00}}$

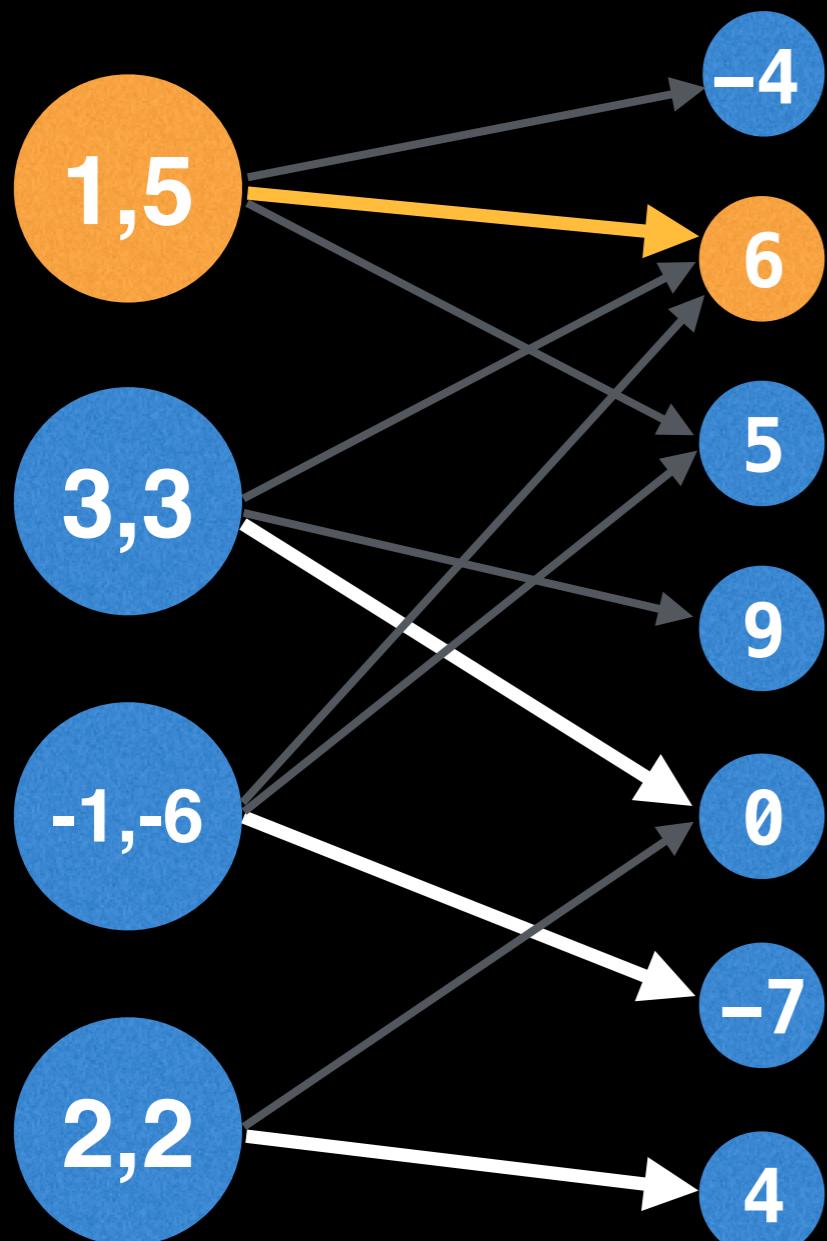
$-1 \square -6 = \boxed{\phantom{00}}$

$2 \square 2 = \boxed{\phantom{00}}$

X

÷

+



$1 \square 5 = \boxed{\phantom{00}}$

$3 \square 3 = \boxed{\phantom{00}}$

$-1 \square -6 = \boxed{\phantom{00}}$

$2 \square 2 = \boxed{\phantom{00}}$

1 + 5 equals 6?

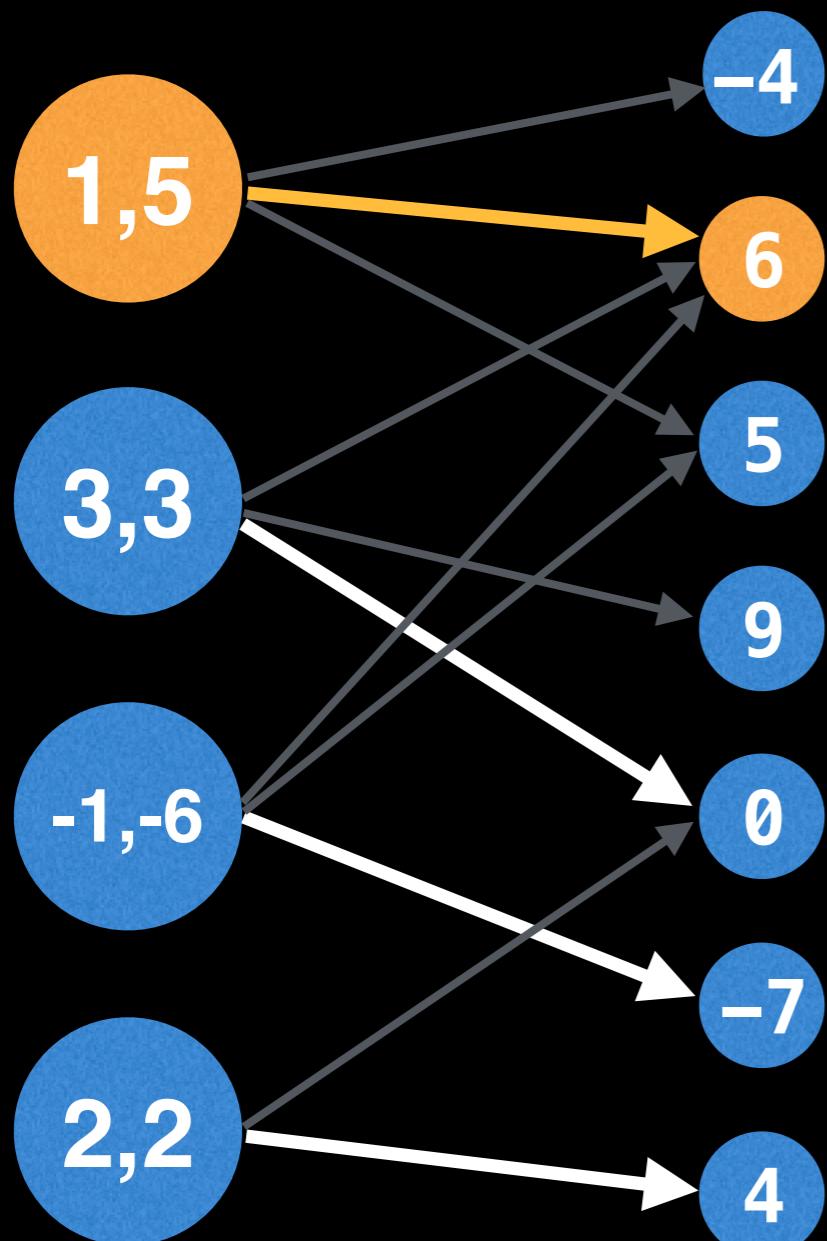
1 - 5 equals 6?

1 \* 5 equals 6?

X

÷

+



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{\phantom{00}}$$

$$-1 \boxed{-} 6 = \boxed{\phantom{00}}$$

$$2 \boxed{*} 2 = \boxed{\phantom{00}}$$

1  5 equals 6?

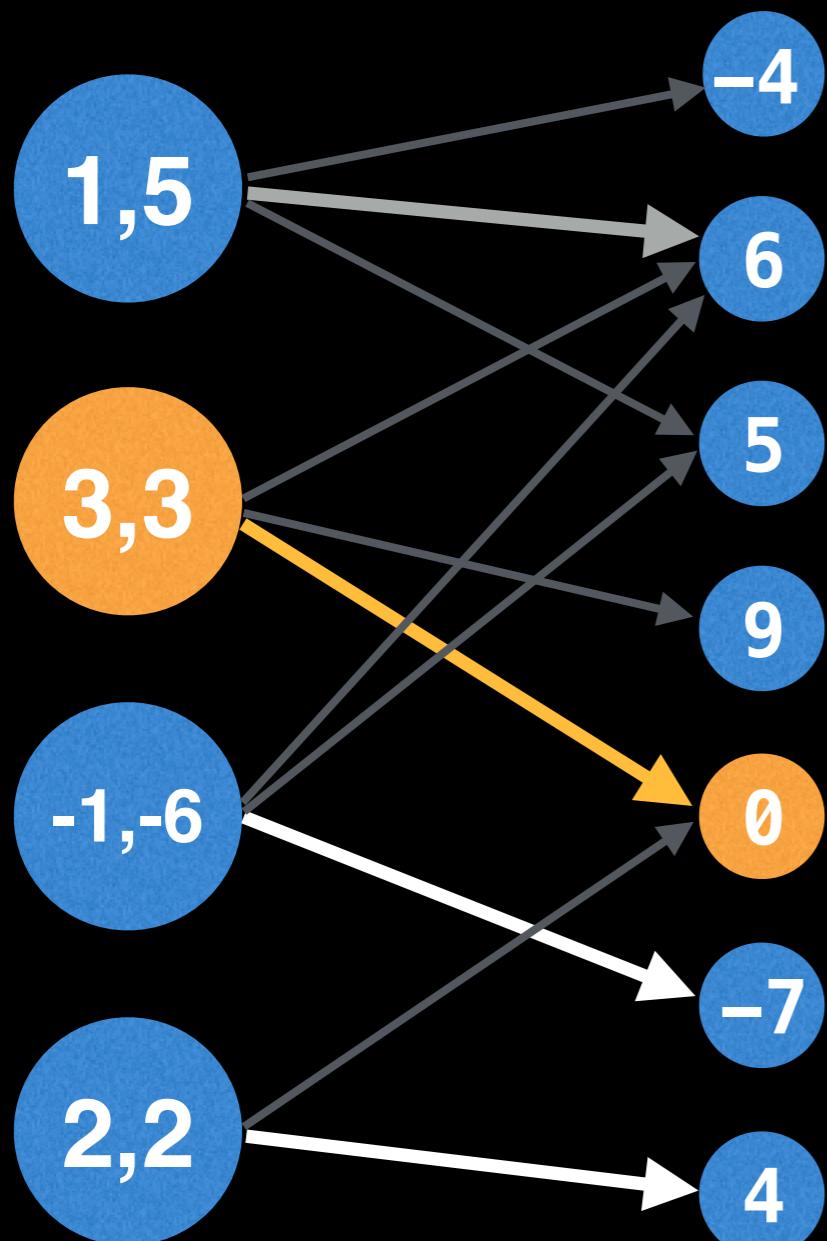
1 - 5 equals 6?

1 \* 5 equals 6?

X

÷

+



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{\phantom{00}}$$

$$-1 \boxed{-} 6 = \boxed{\phantom{00}}$$

$$2 \boxed{*} 2 = \boxed{\phantom{00}}$$

3 + 3 equals 0?

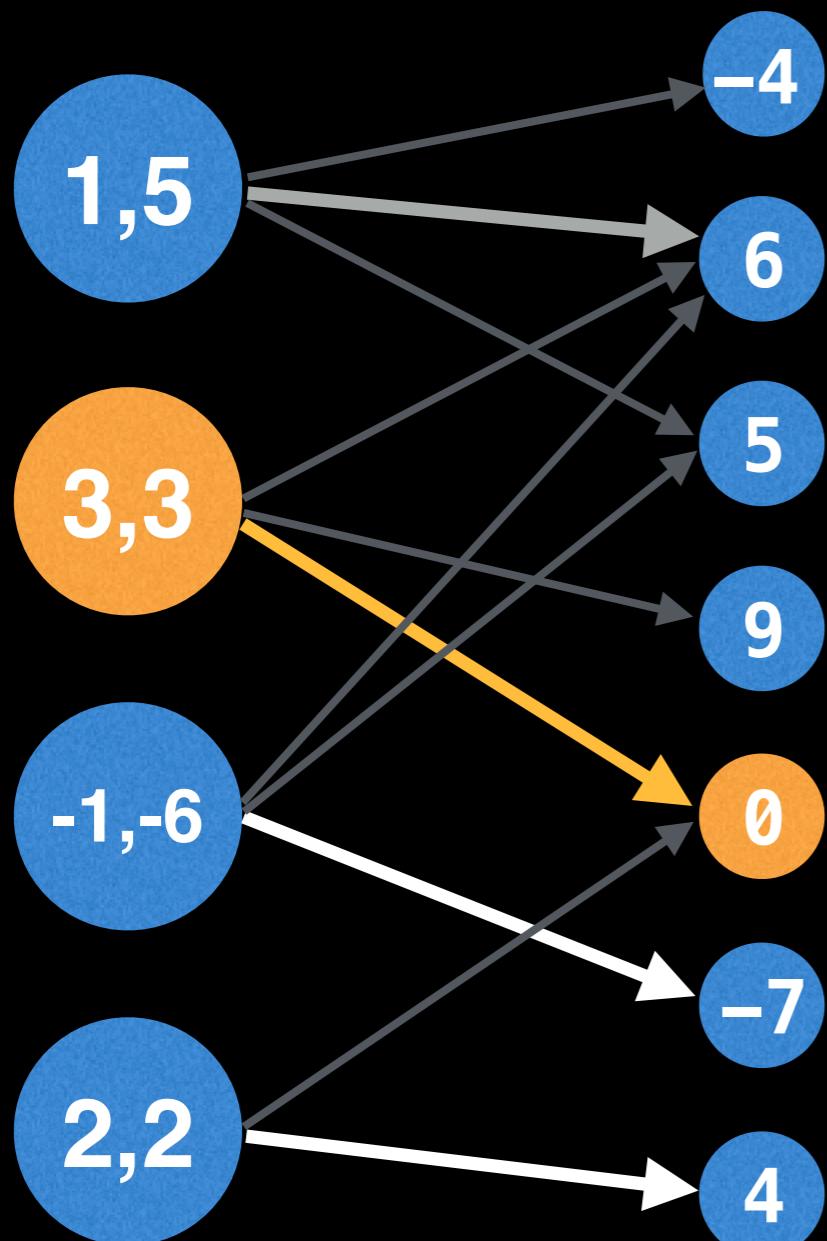
3 - 3 equals 0?

3 \* 3 equals 0?

X

÷

+



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{0}$$

$$-1 \boxed{-} 6 = \boxed{\quad}$$

$$2 \boxed{-} 2 = \boxed{\quad}$$

$3 + 3$  equals  $0$ ?

$3 \boxed{-} 3$  equals  $0$ ?

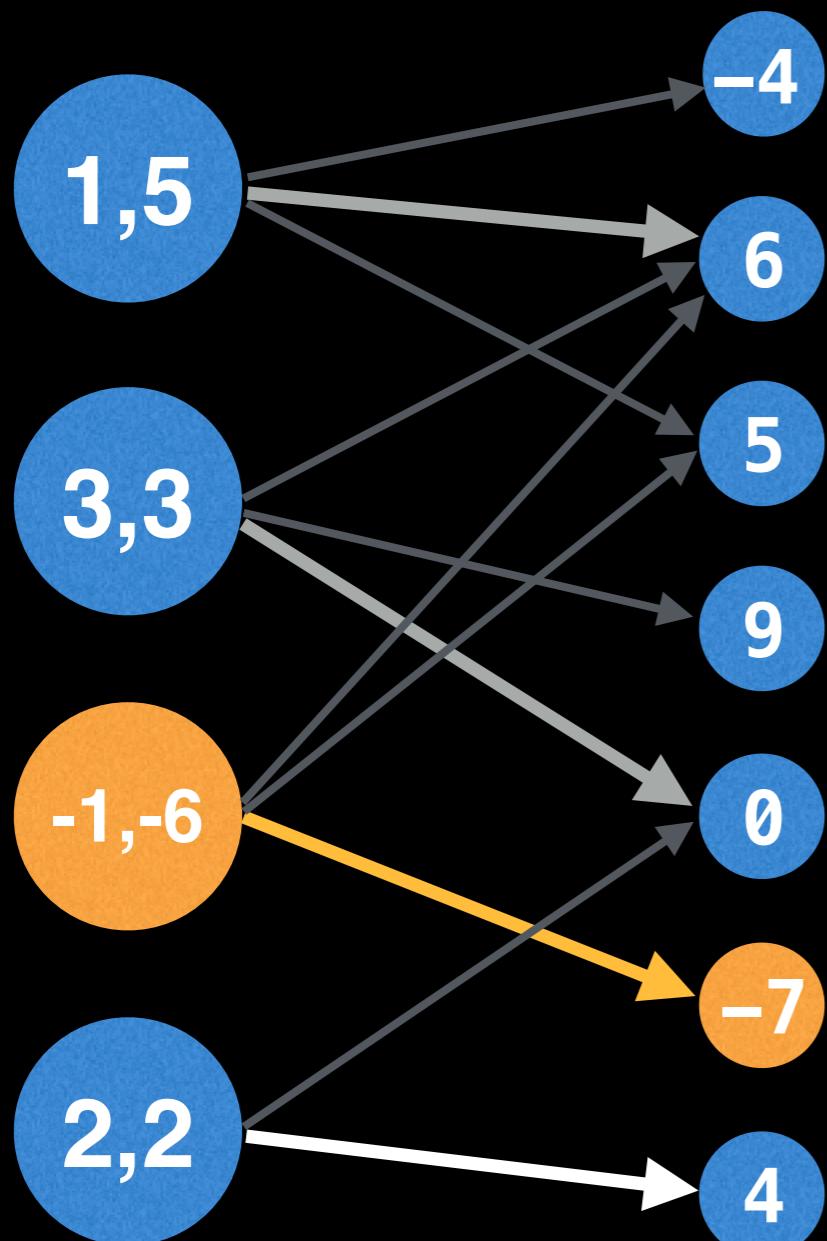
$3 * 3$  equals  $0$ ?

X

÷



+



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{0}$$

$$-1 \boxed{-} 6 = \boxed{\quad}$$

$$2 \boxed{-} 2 = \boxed{\quad}$$

$-1 + -6$  equals  $-7$ ?

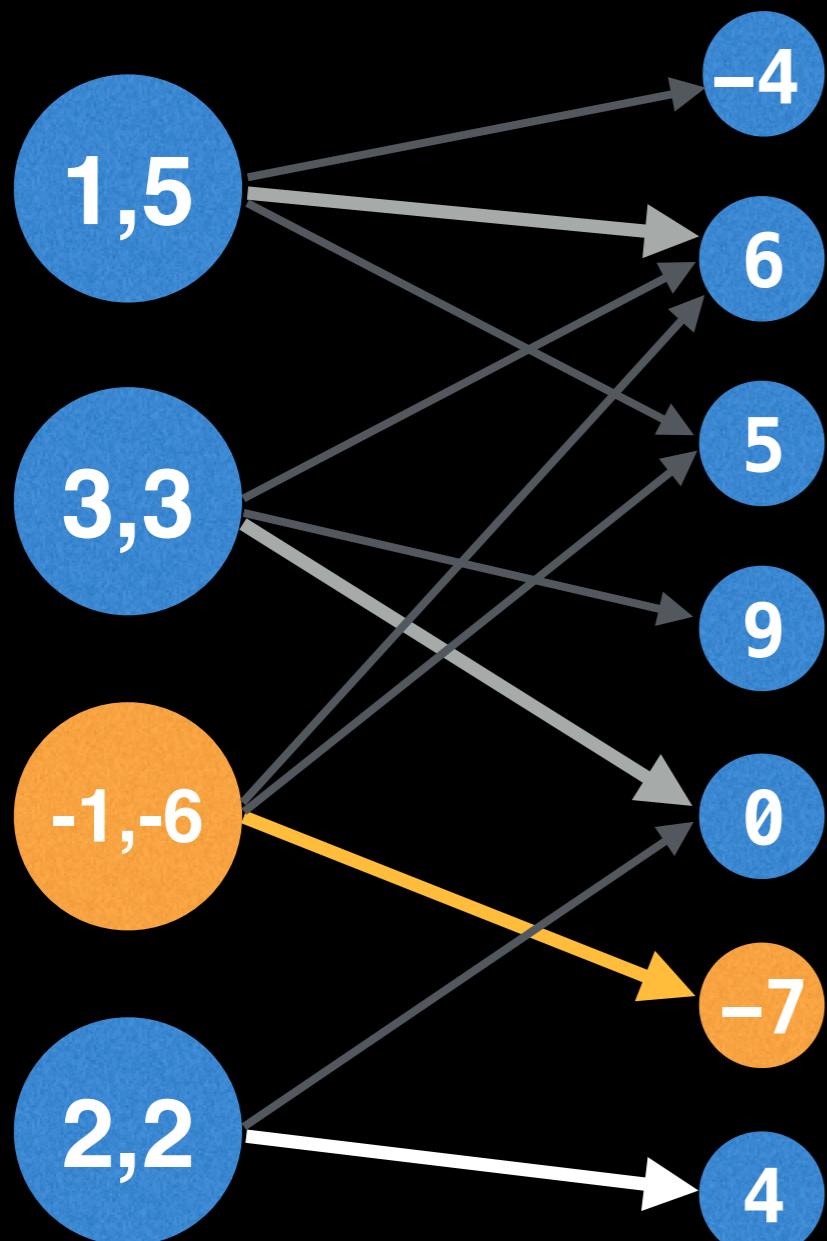
$-1 - -6$  equals  $-7$ ?

$-1 * -6$  equals  $-7$ ?

X

÷

+



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{0}$$

$$-1 \boxed{+} -6 = \boxed{-7}$$

$$2 \boxed{-} 2 = \boxed{\phantom{0}}$$

**-1  -6 equals -7?**

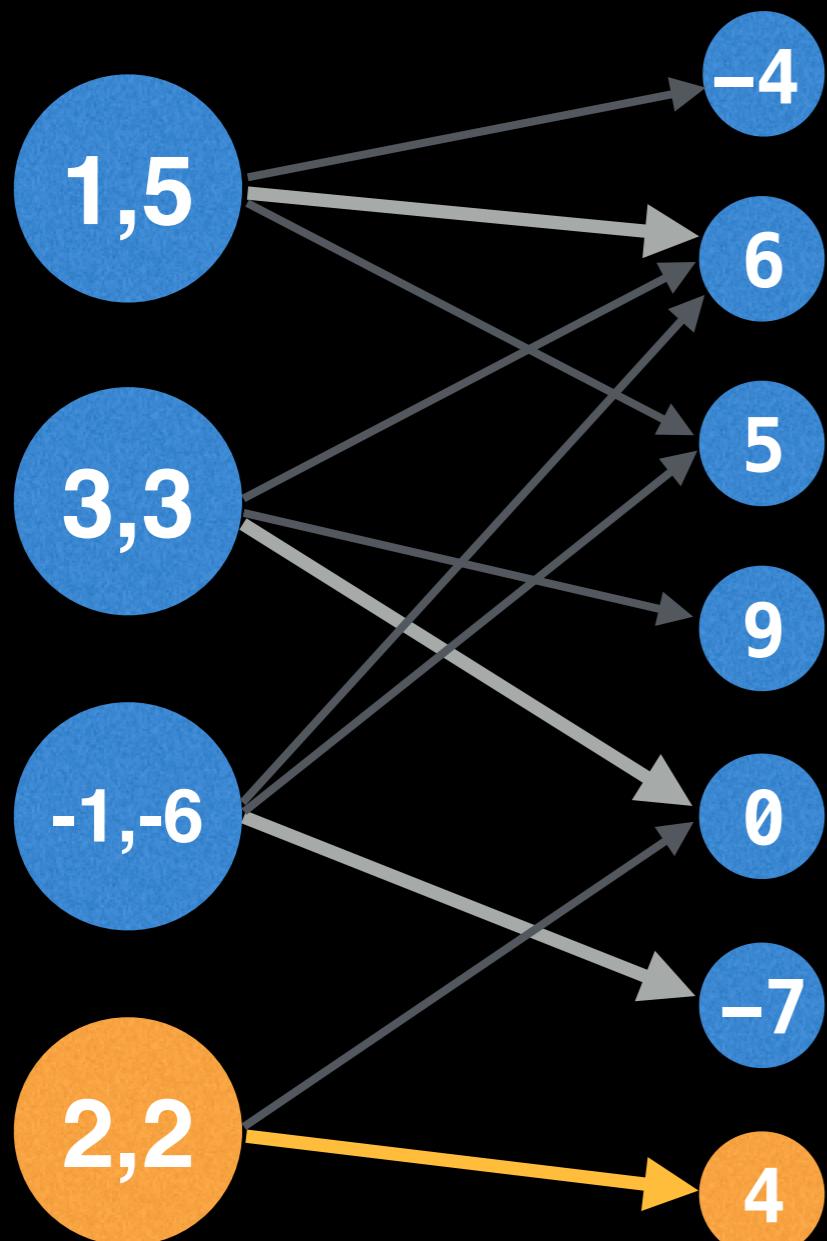
-1 - -6 equals -7?

-1 \* -6 equals -7?

X

÷

+



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{0}$$

$$-1 \boxed{+} -6 = \boxed{-7}$$

$$2 \boxed{\square} 2 = \boxed{\square}$$

2 + 2 equals 4?

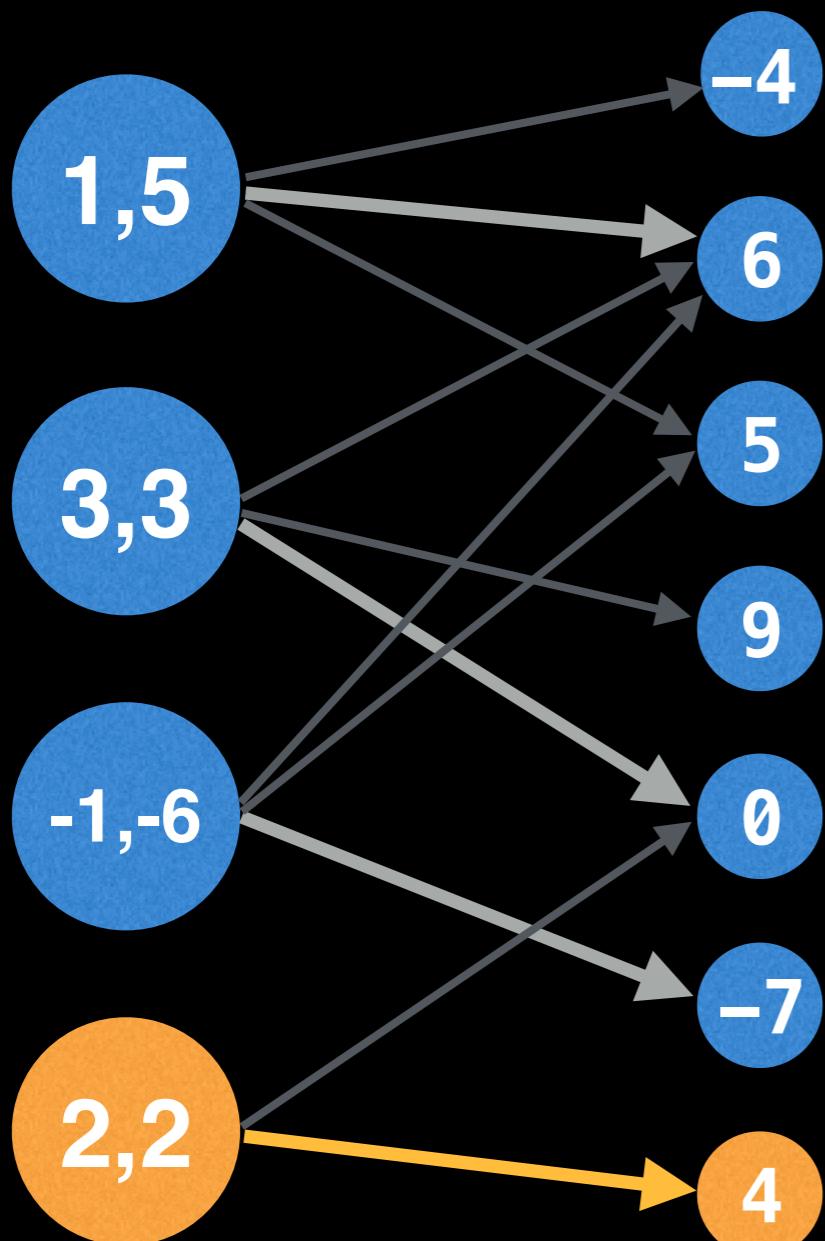
2 - 2 equals 4?

2 \* 2 equals 4?

X

÷

+



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{0}$$

$$-1 \boxed{+} -6 = \boxed{-7}$$

$$2 \boxed{\square} 2 = \boxed{\square}$$

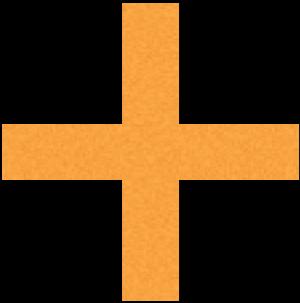
2 + 2 equals 4?

2 - 2 equals 4?

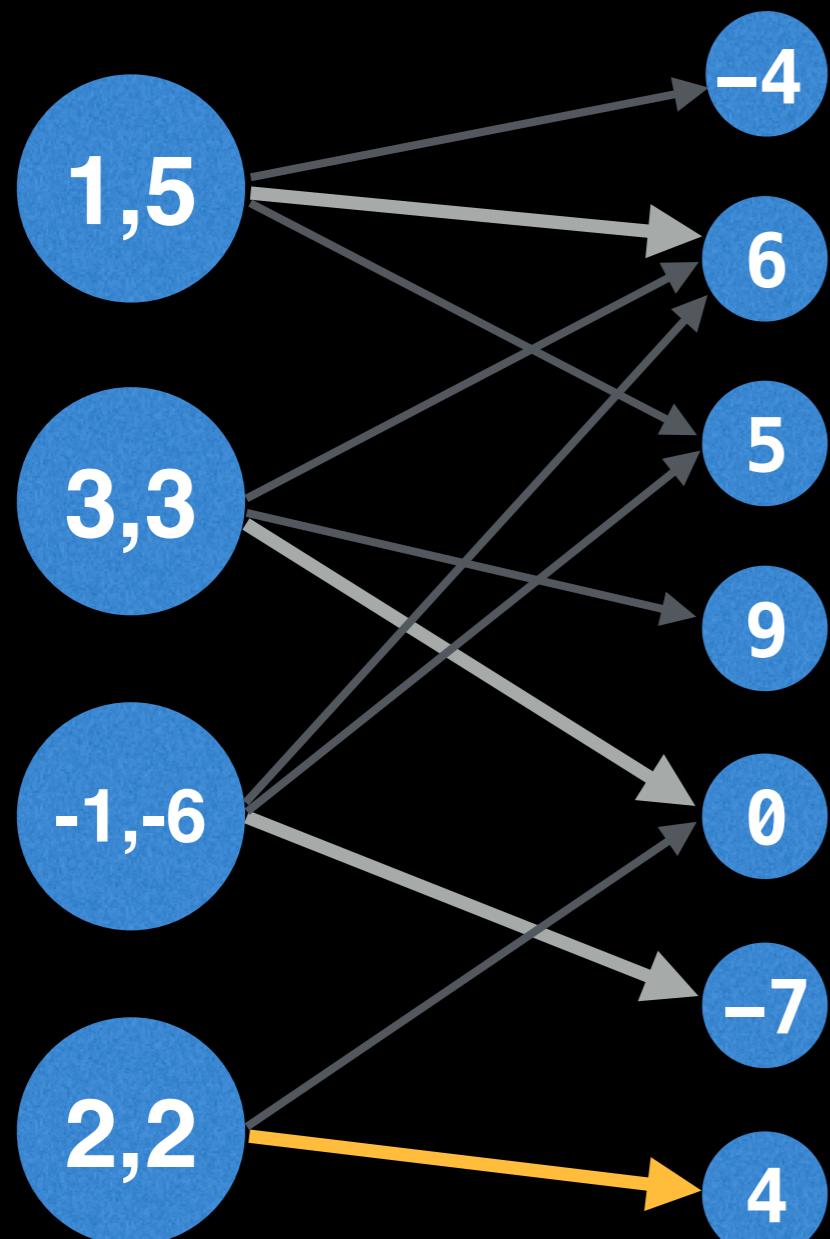
2 \* 2 equals 4?

X

÷



You can pick either operator if multiple of them work.



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{0}$$

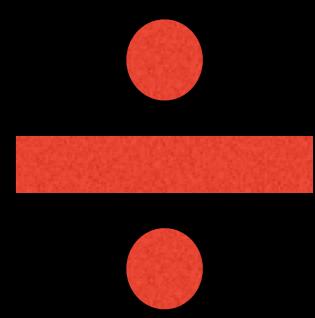
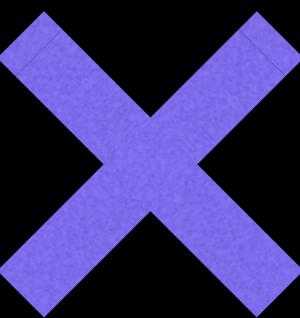
$$-1 \boxed{+} -6 = \boxed{-7}$$

$$2 \boxed{*} 2 = \boxed{4}$$

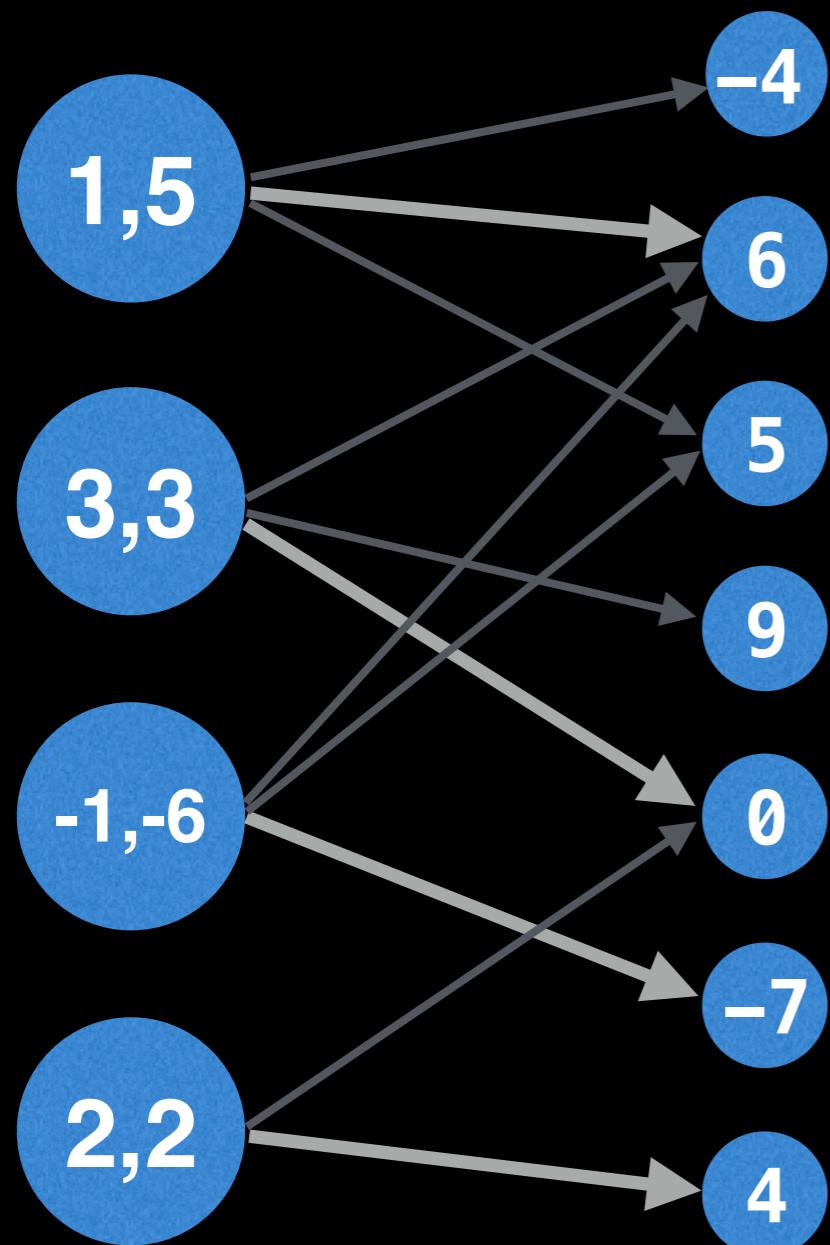
2 + 2 equals 4?

2 - 2 equals 4?

2 2 equals 4?



+



$$1 \boxed{+} 5 = \boxed{6}$$

$$3 \boxed{-} 3 = \boxed{0}$$

$$-1 \boxed{+} -6 = \boxed{-7}$$

$$2 \boxed{*} 2 = \boxed{4}$$

X

÷

+

$$1 + 5 = 6$$

$$3 - 3 = 0$$

$$-1 + -6 = -7$$

$$2 * 2 = 4$$

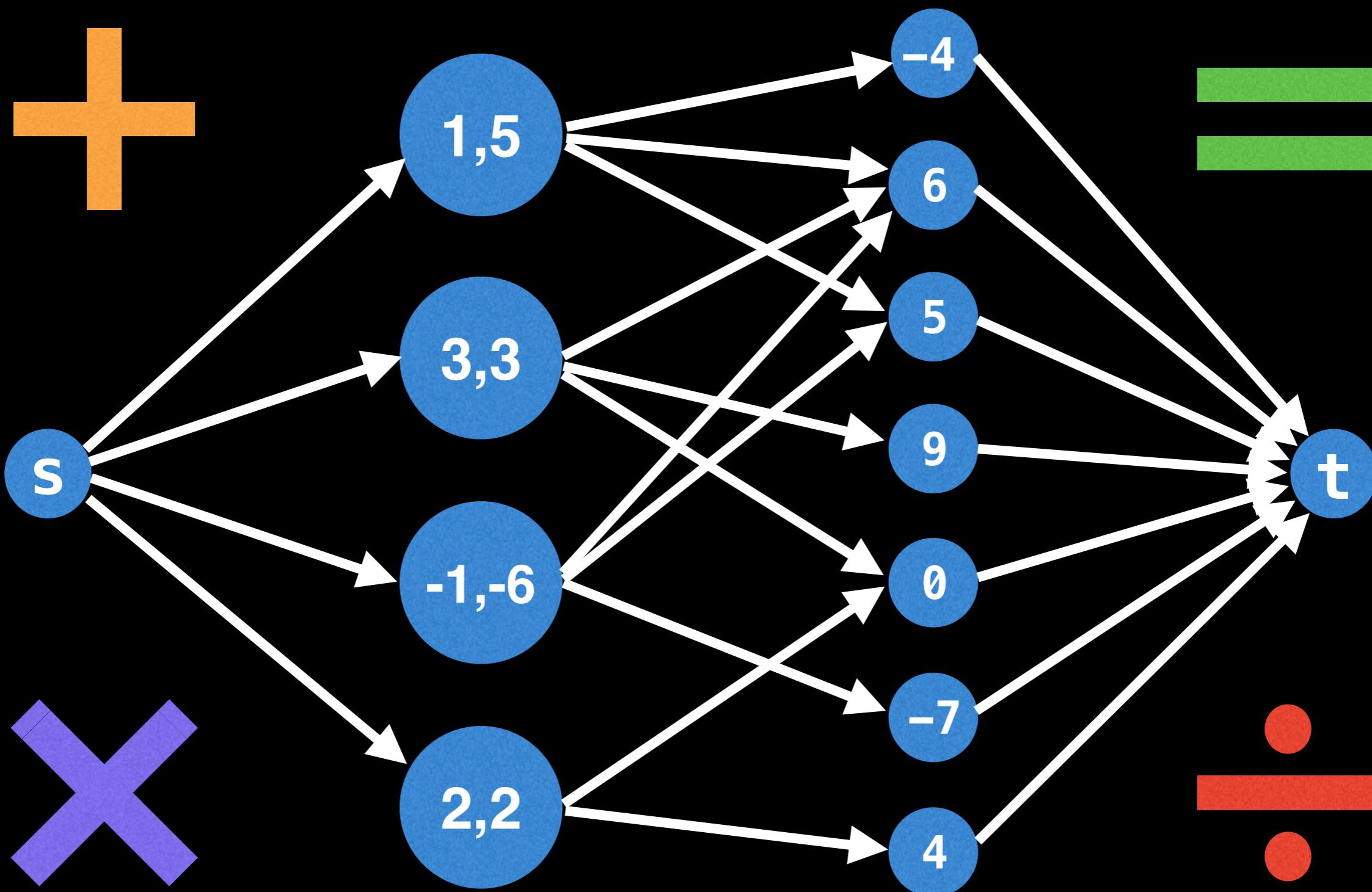


X

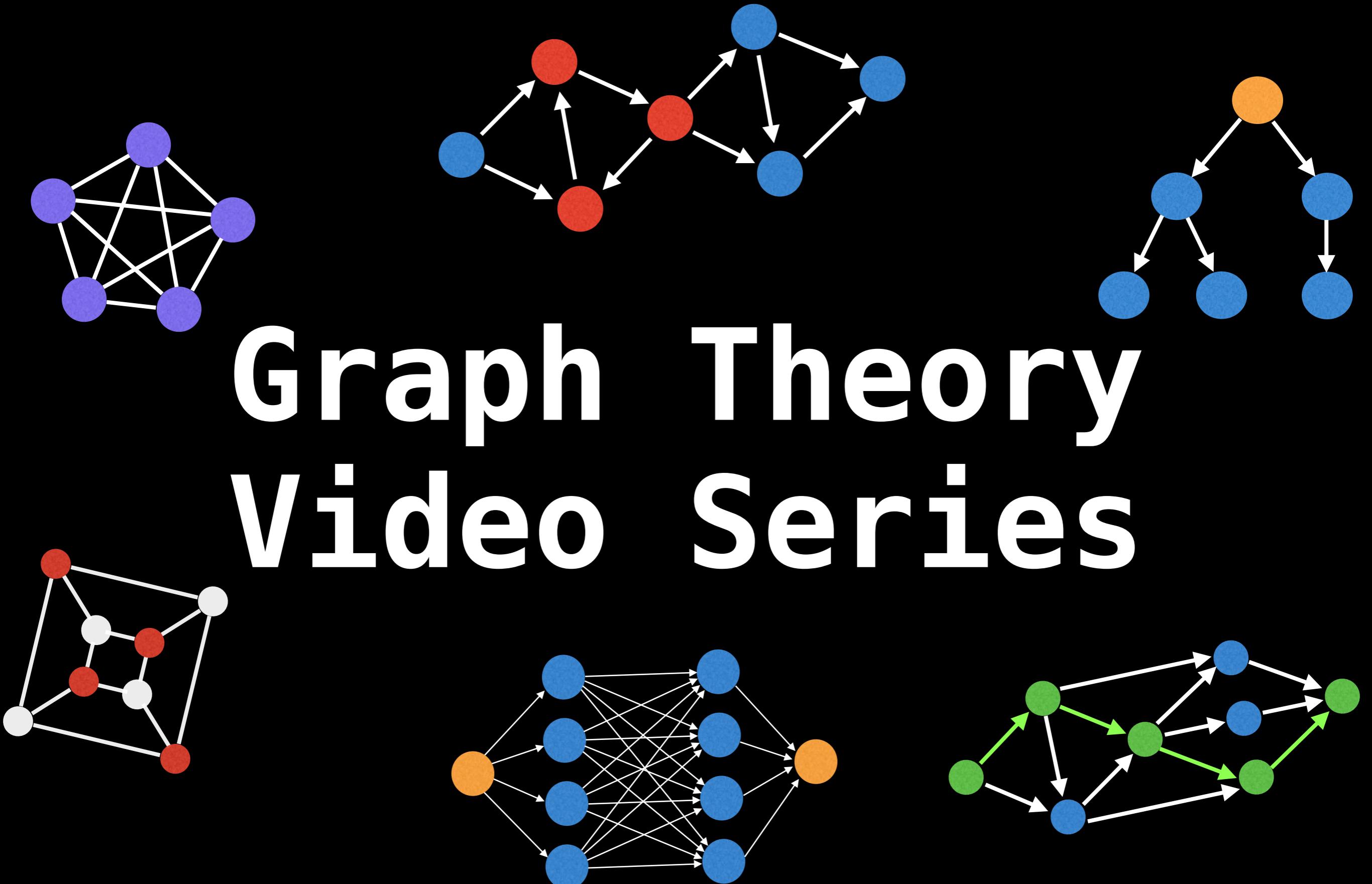
÷



# Network Flow: Elementary Math Problem



# Graph Theory Video Series

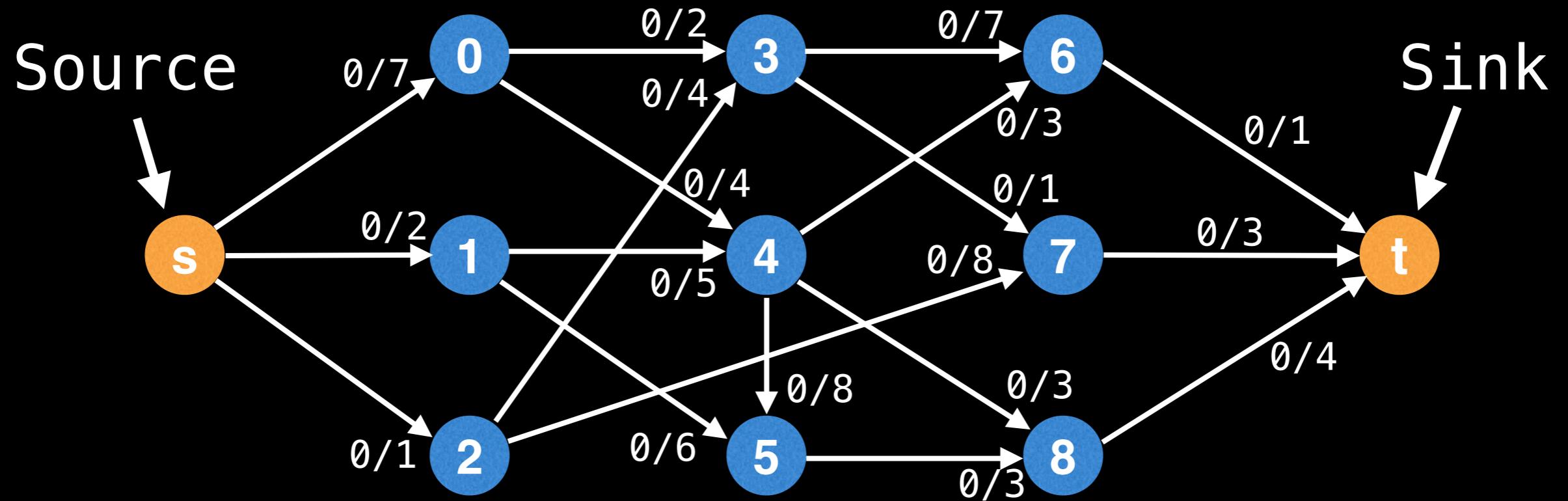


# Network Flow: Edmonds-Karp Algorithm

William Fiset

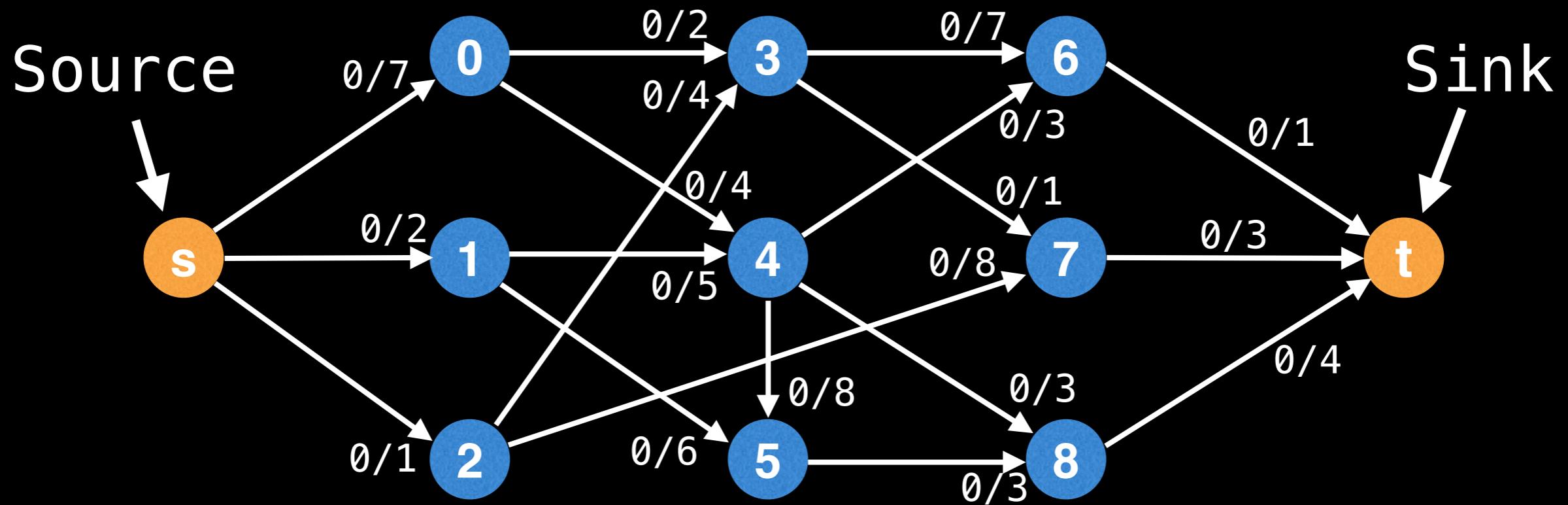
# Ford-Fulkerson overview

The Ford–Fulkerson method is a common technique used to find the maximum flow.



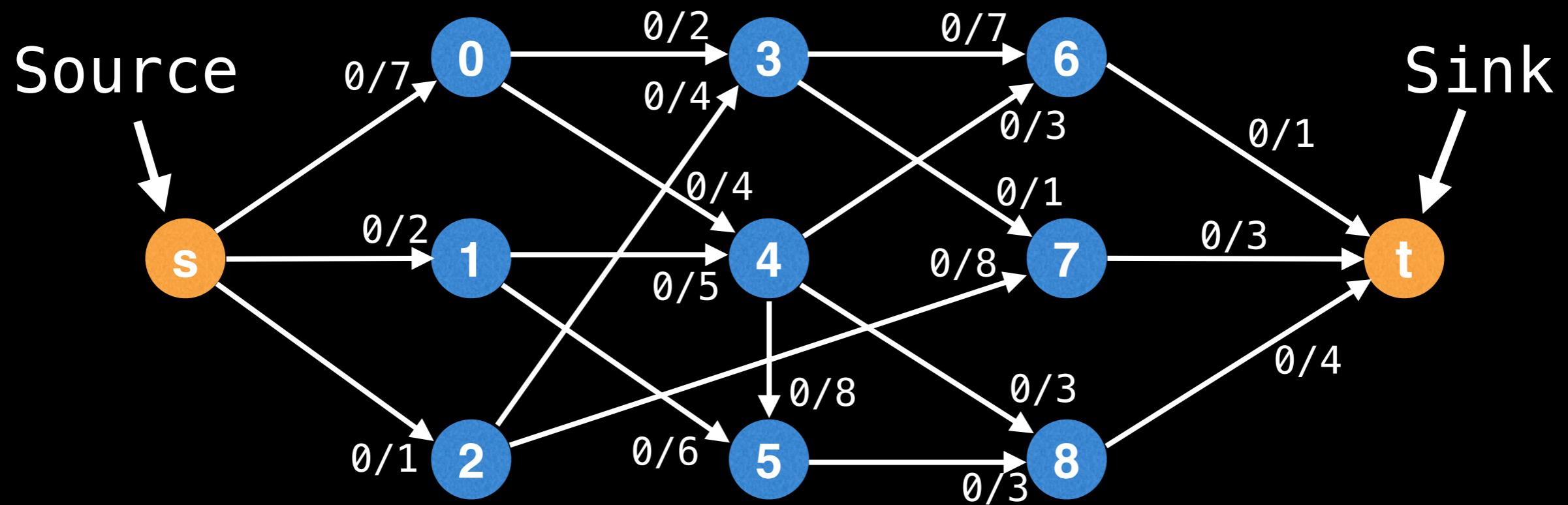
# Ford-Fulkerson overview

At a high level, Ford-Fulkerson says that all we want to do is repeatedly find augmenting paths from  $s \rightarrow t$  in the flow graph, augment the flow and repeat until no more paths exist.



# Ford-Fulkerson overview

The key takeaway here is that the Ford-Fulkerson method does not specify how to actually find augmenting paths. This is where optimizations come into play.



# Edmonds-Karp

The Ford–Fulkerson method using a DFS to find augmenting paths takes  $O(Ef)$  where  $E$  is the number of edges and  $f$  is the max flow.

**ANIMATION HERE**

# Edmonds–Karp

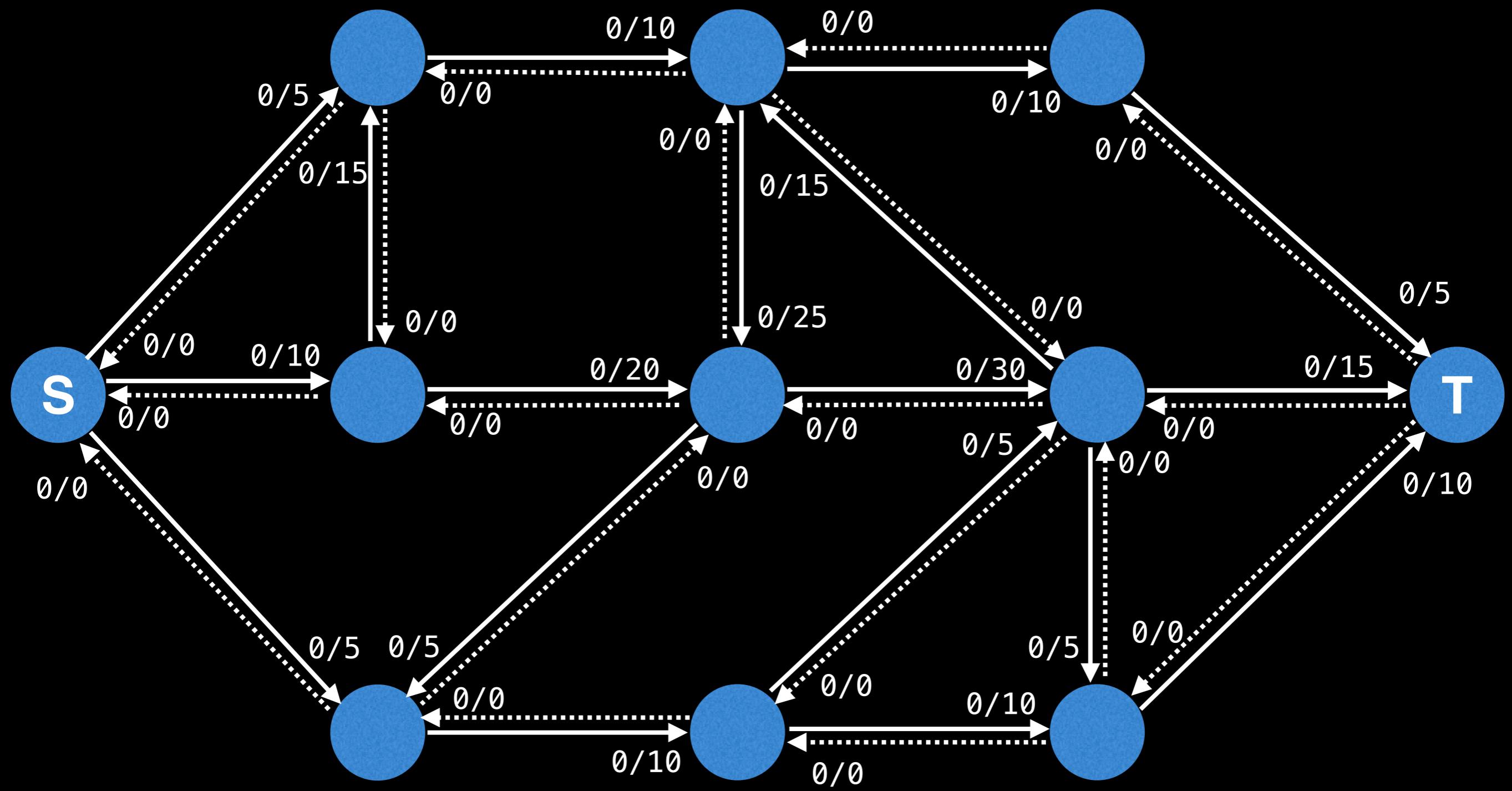
The Ford–Fulkerson method using a DFS to find augmenting paths takes  $O(Ef)$  where  $E$  is the number of edges and  $f$  is the max flow.

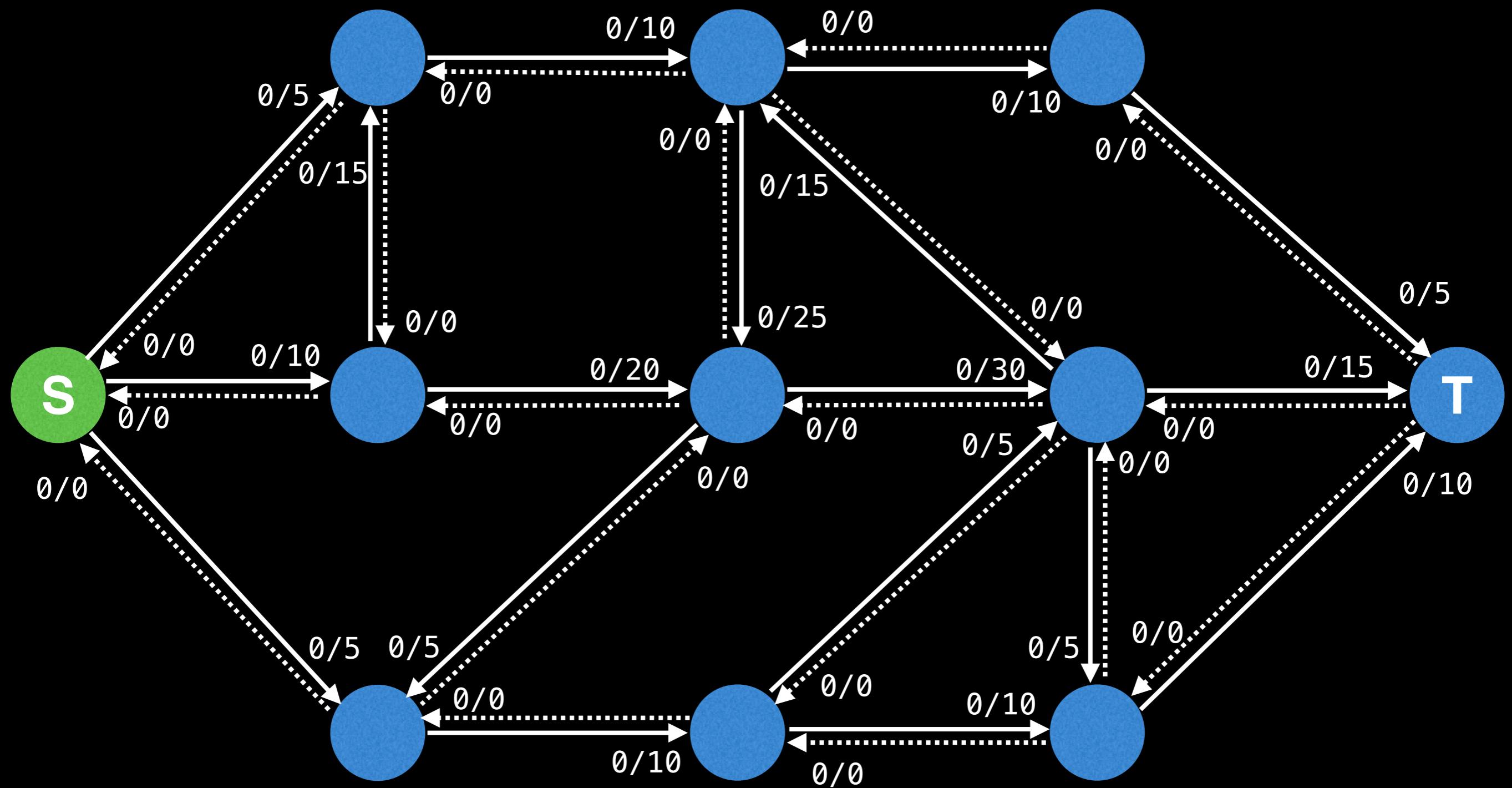
The Edmonds–Karp algorithm uses a **Breadth First Search (BFS)** to find augmenting paths which yields an arguably better time complexity of  $O(VE^2)$ . The major difference in this approach is that the time complexity no longer depends on the capacity value of any edge!

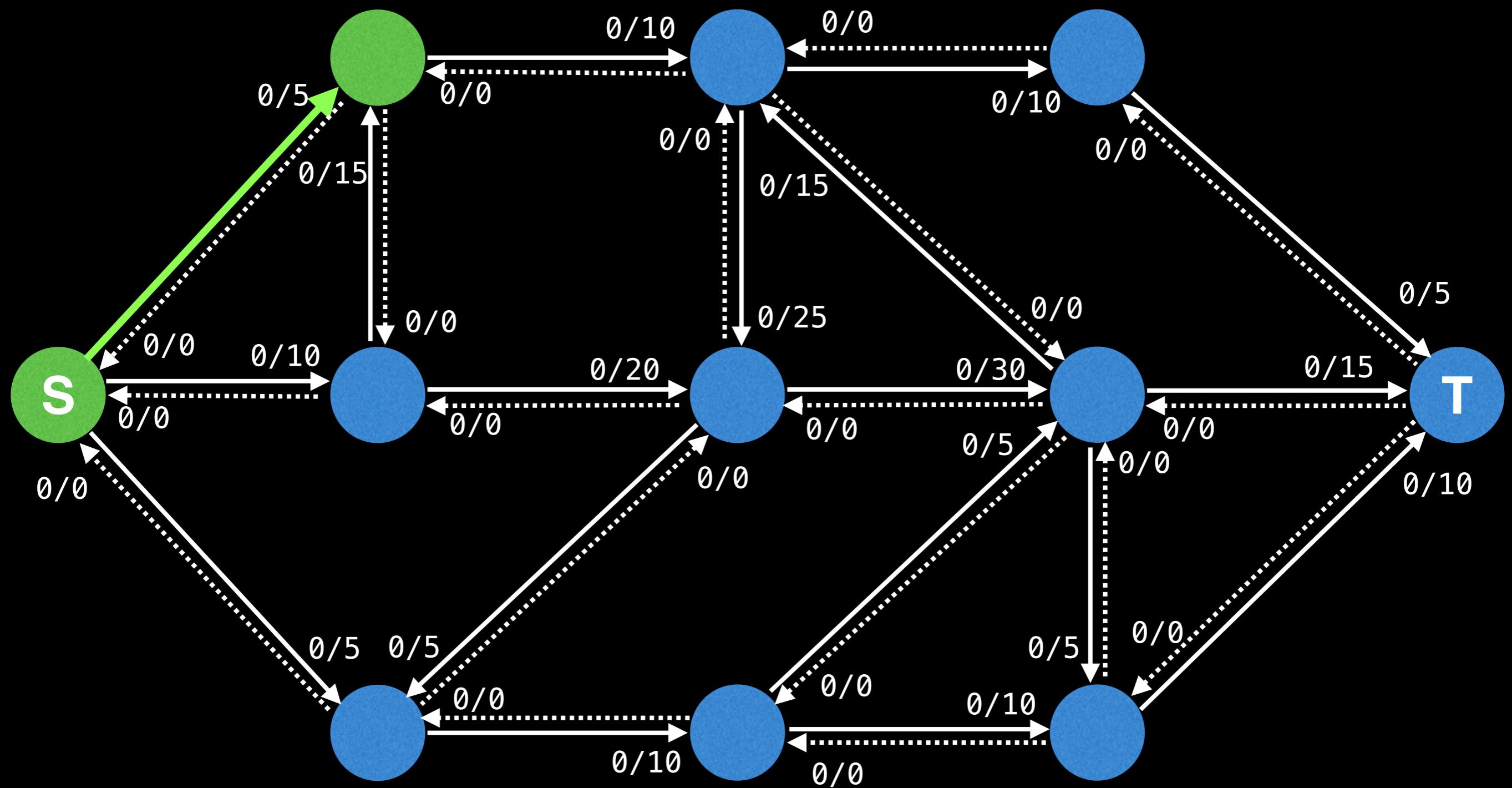
# Shortest augmenting path

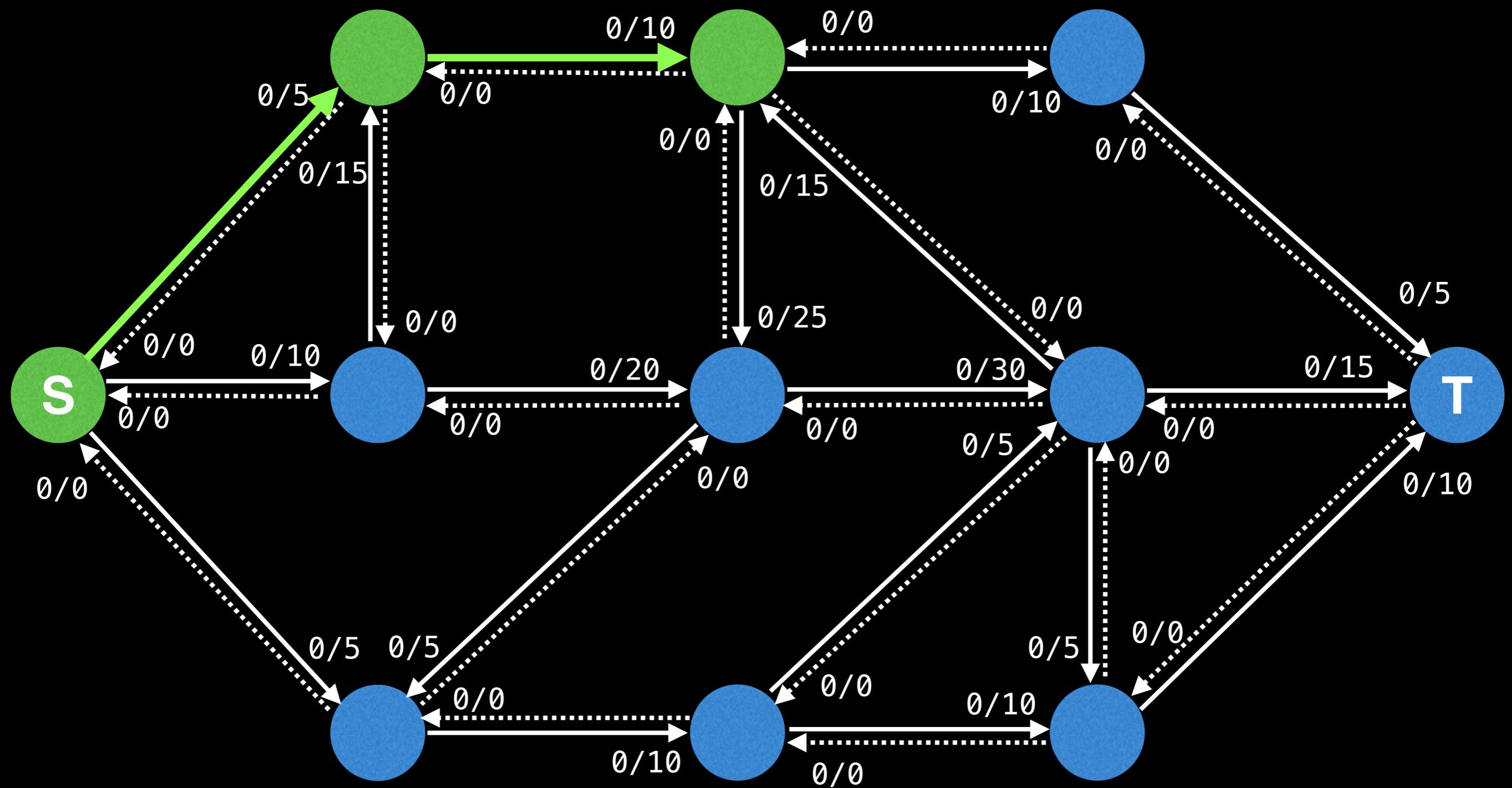
The **Edmonds–Karp algorithm** can also be thought of as a method of augmentation which repeatedly **finds the shortest augmenting path** from  $s \rightarrow t$  in terms of the number of edges used each iteration.

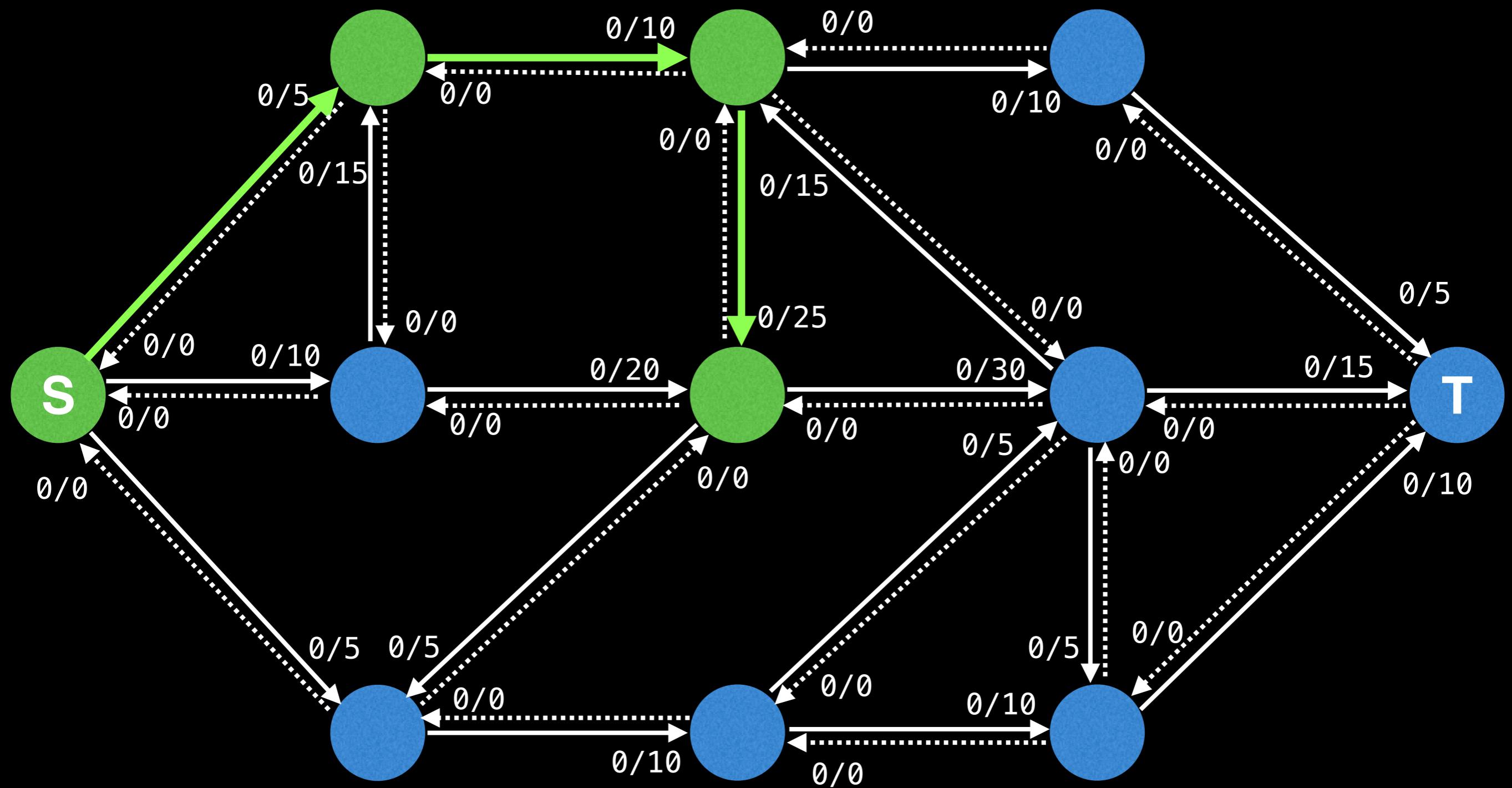
Using a BFS to find augmenting paths ensures that the shortest path from  $s \rightarrow t$  is found every iteration.

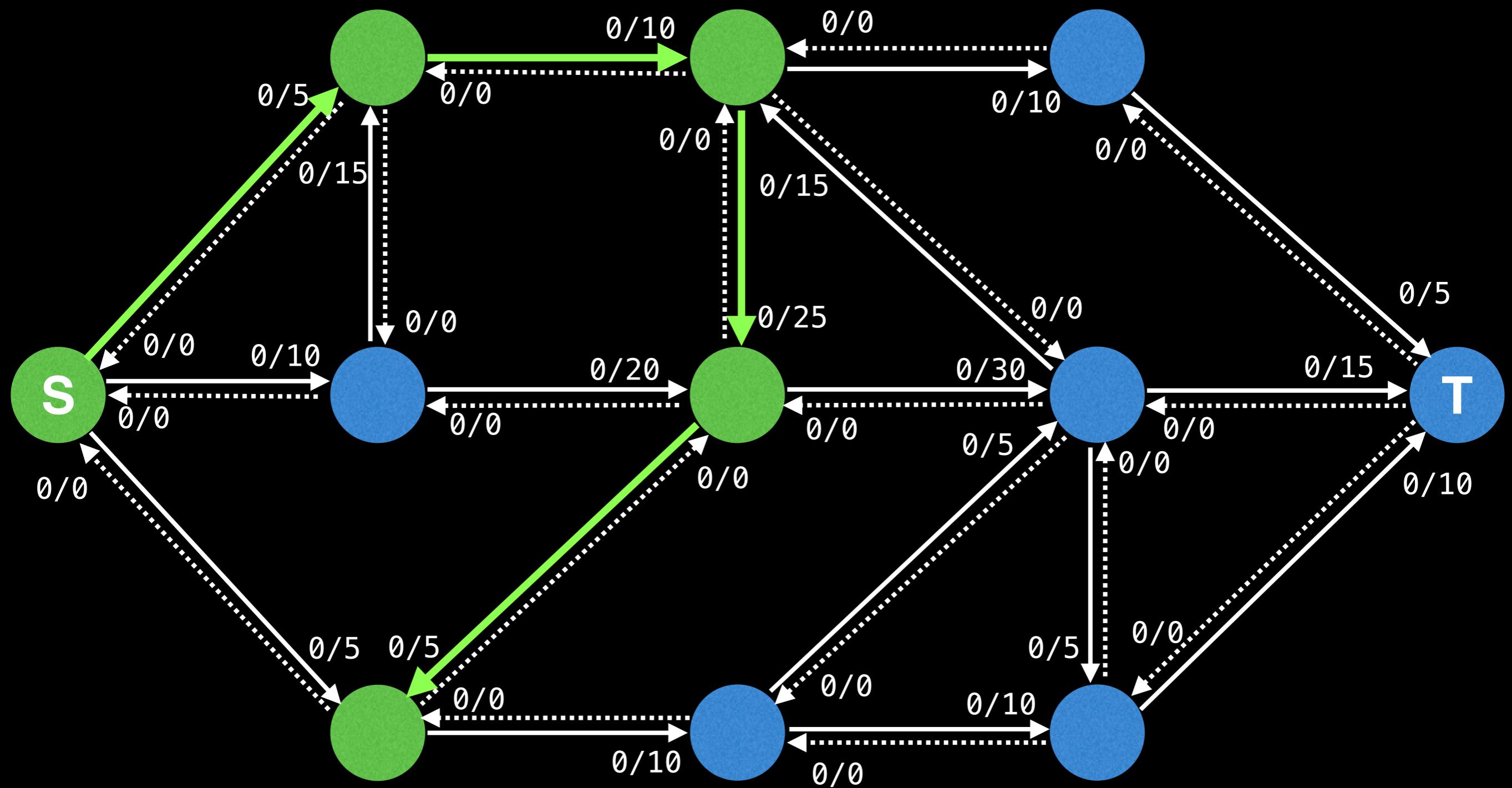


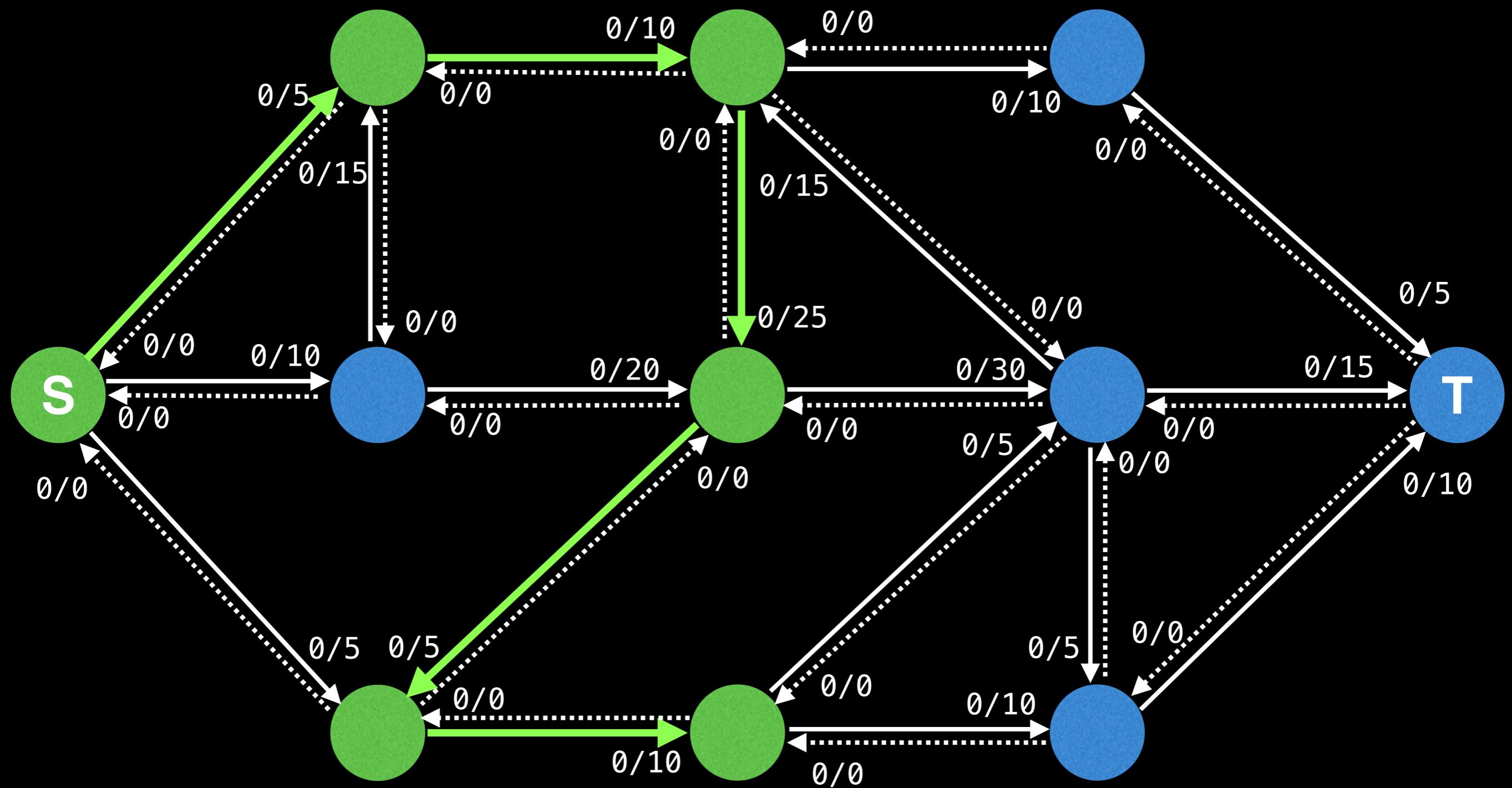


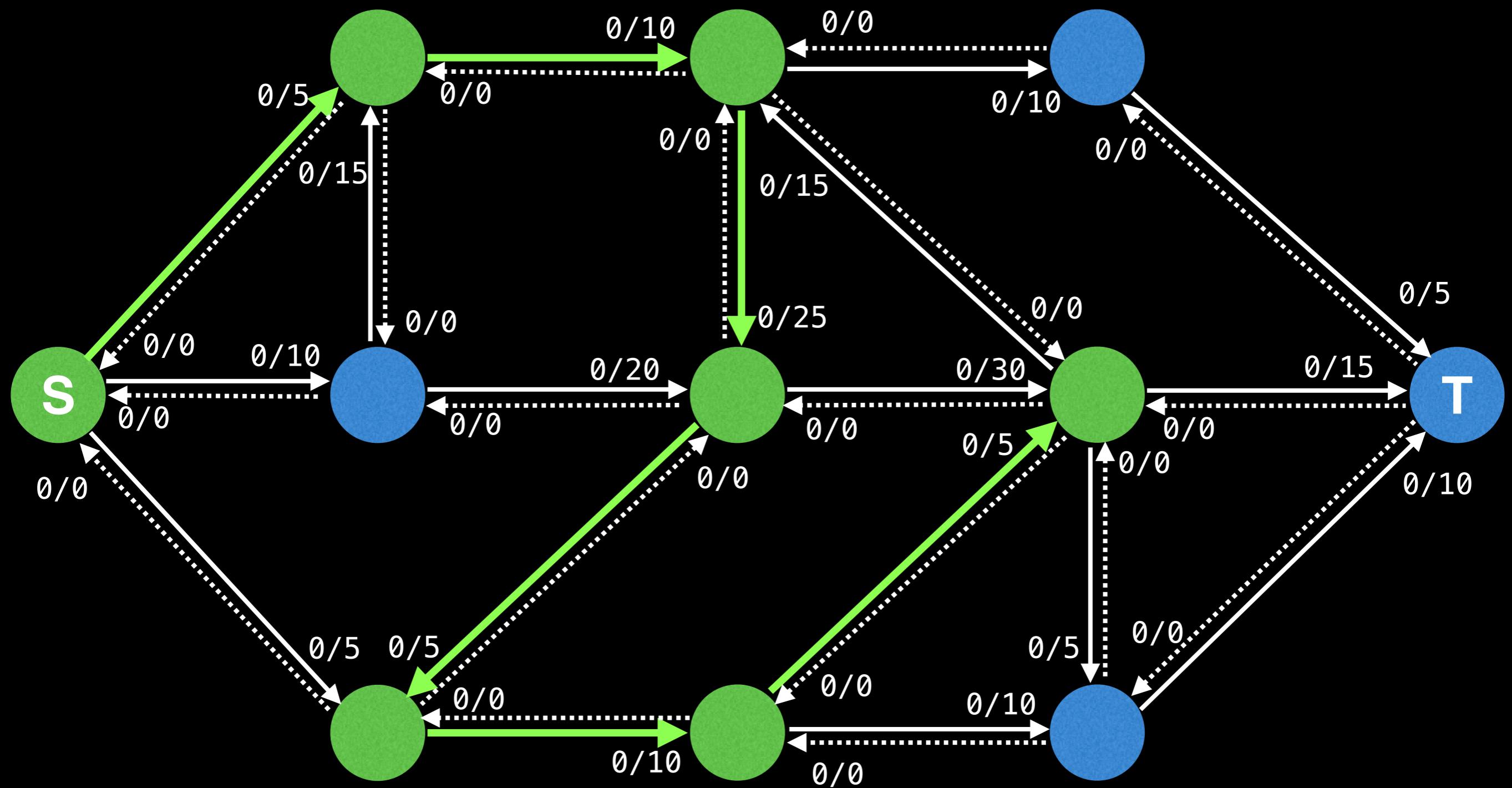




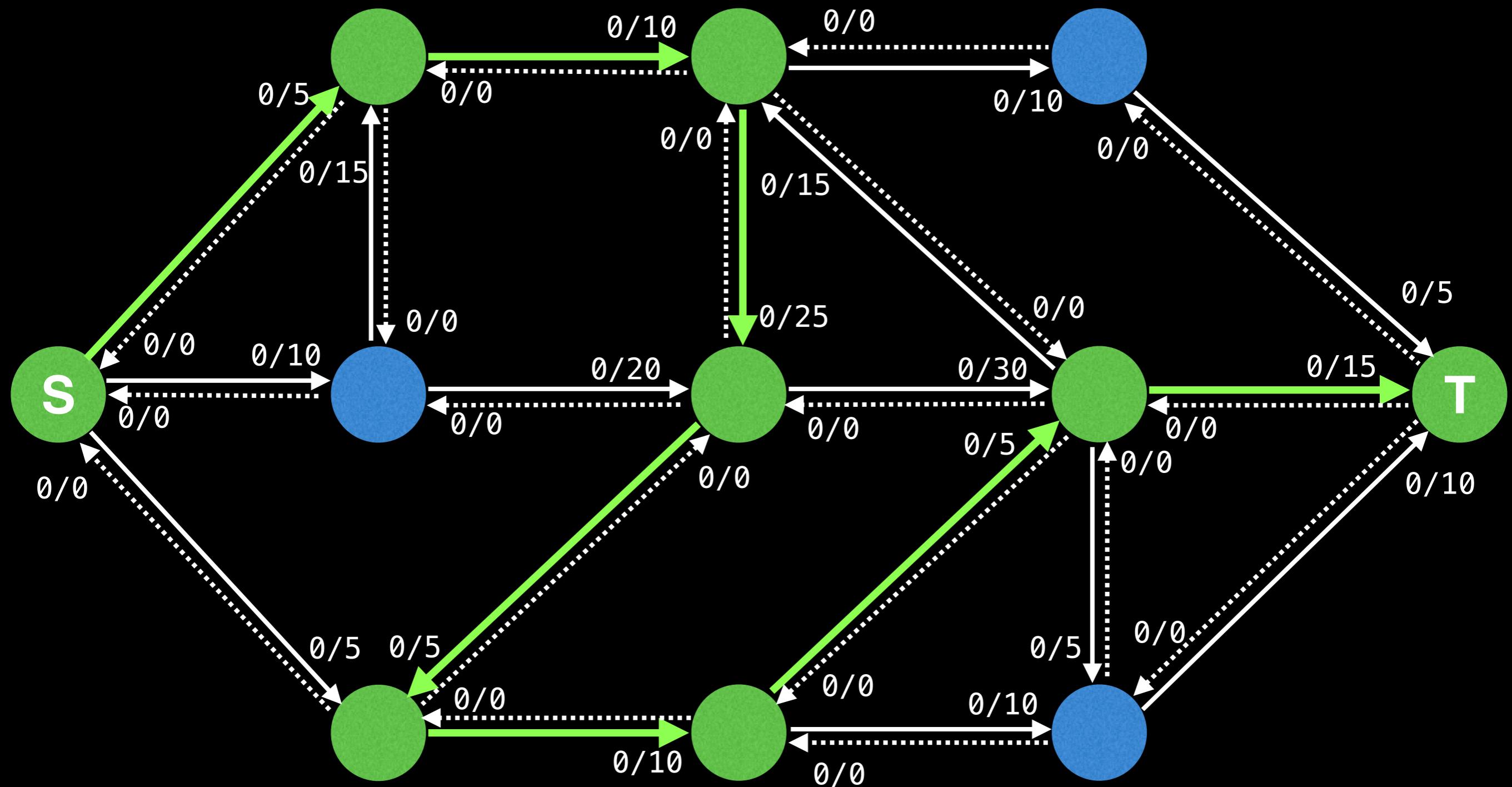




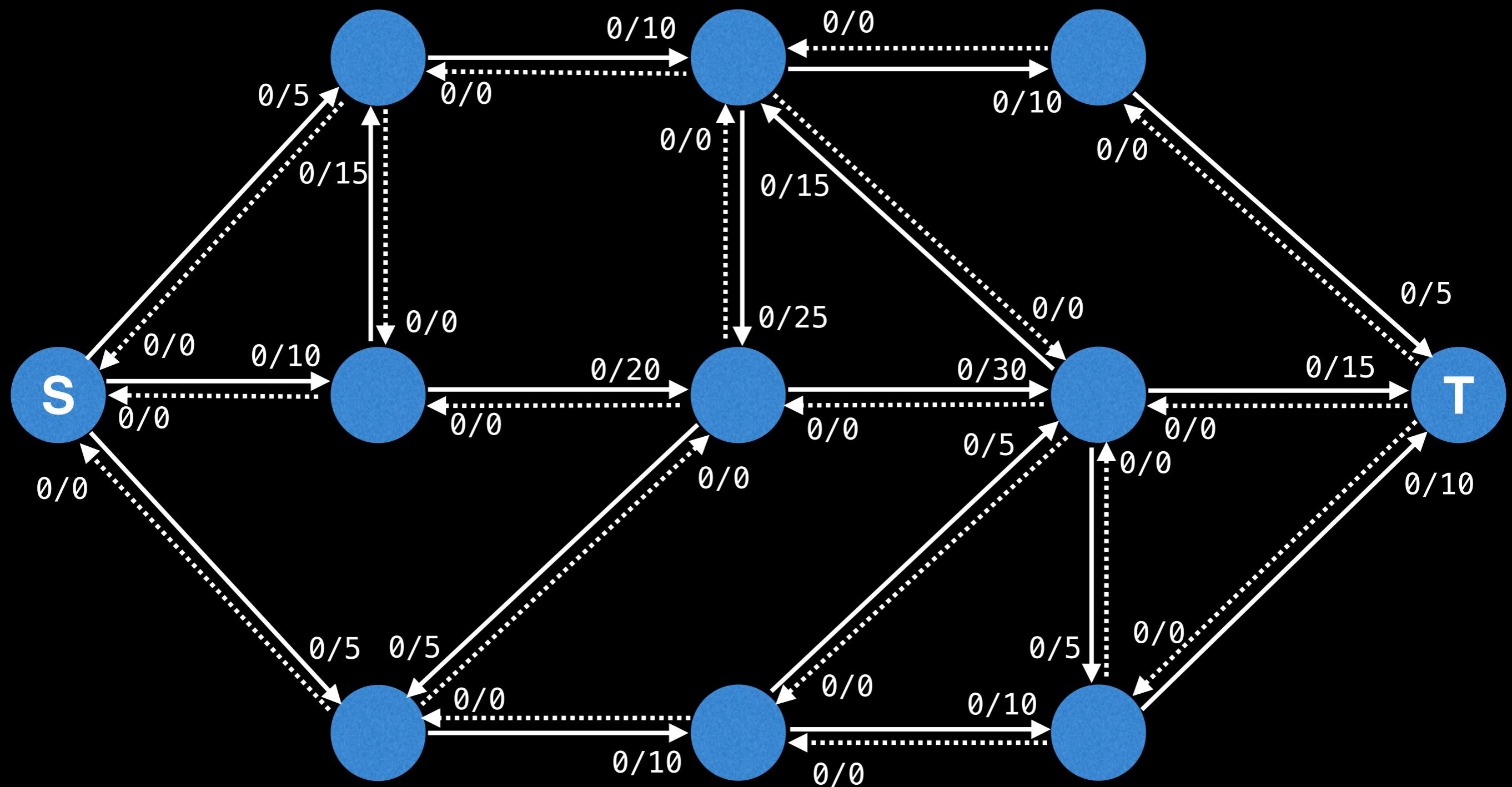




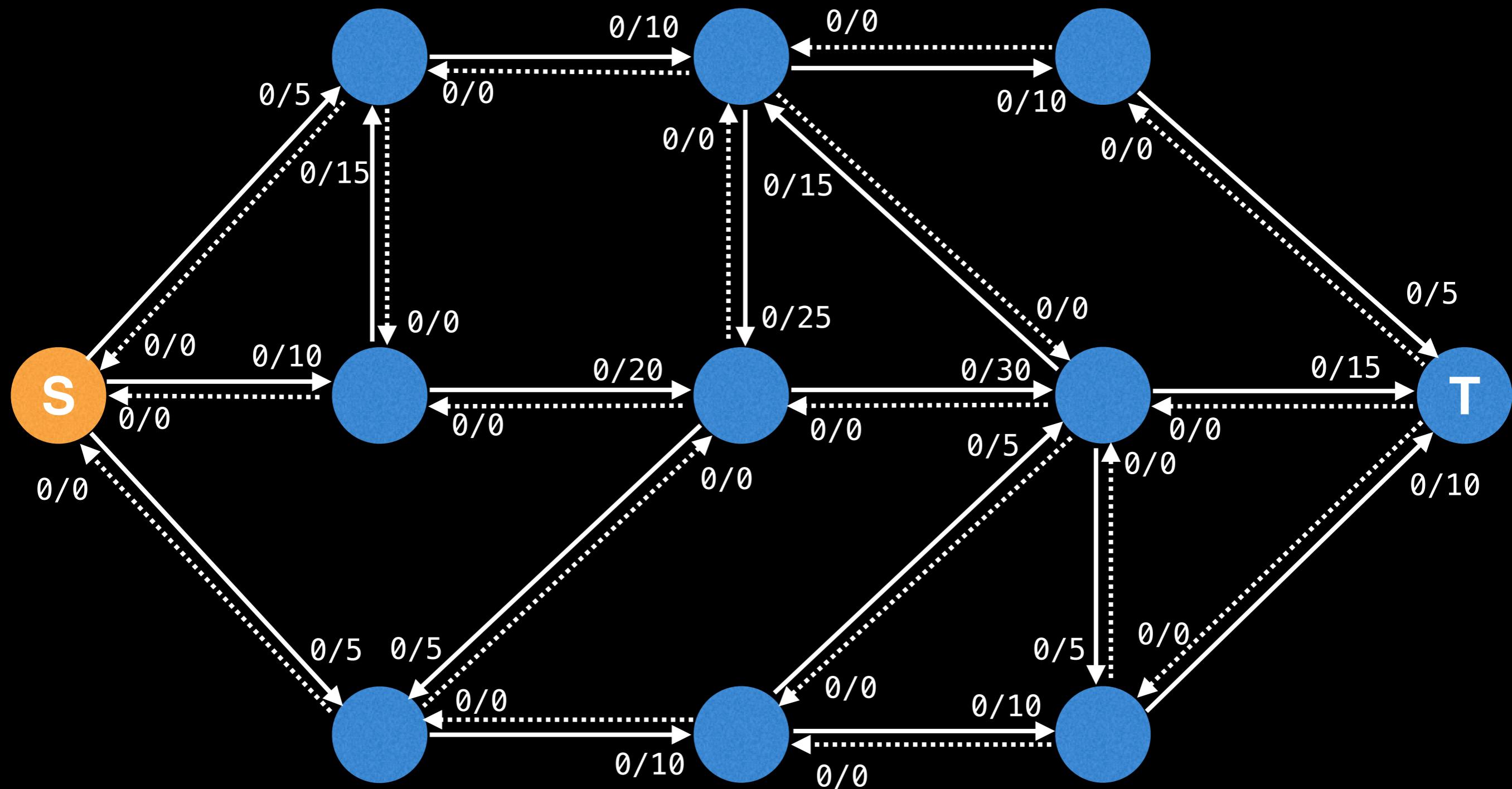
Using a DFS can lead to zigzagging through the flow graph to find the sink which can cause longer augmenting paths. Longer paths are generally undesirable because the longer the path, the higher the chance for a small bottleneck value which results in a longer runtime.



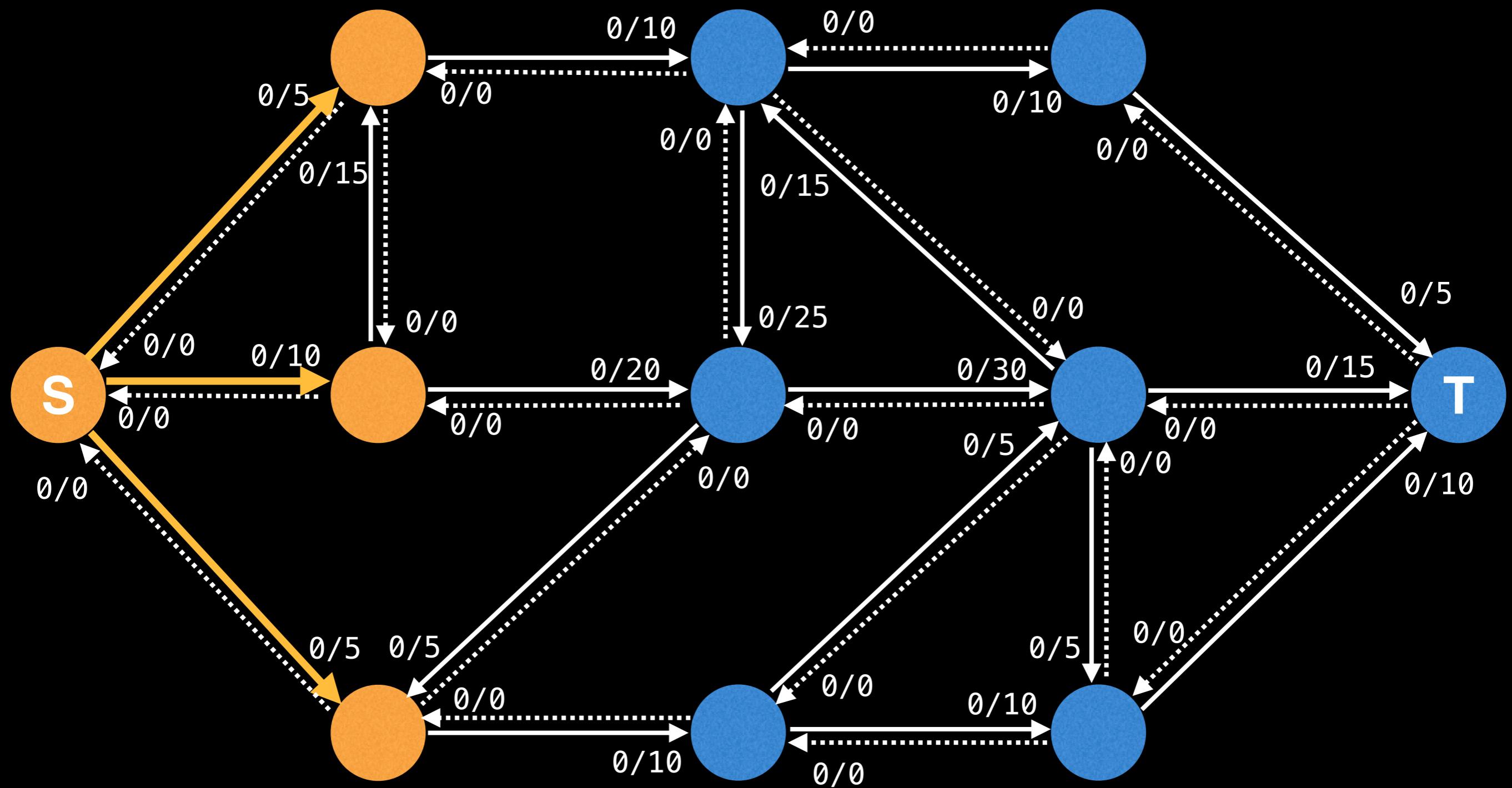
Using the shortest path (in terms of the number of edges) as an augmenting path is a great approach to avoid the DFS worse case and reduce the length of augmenting paths.



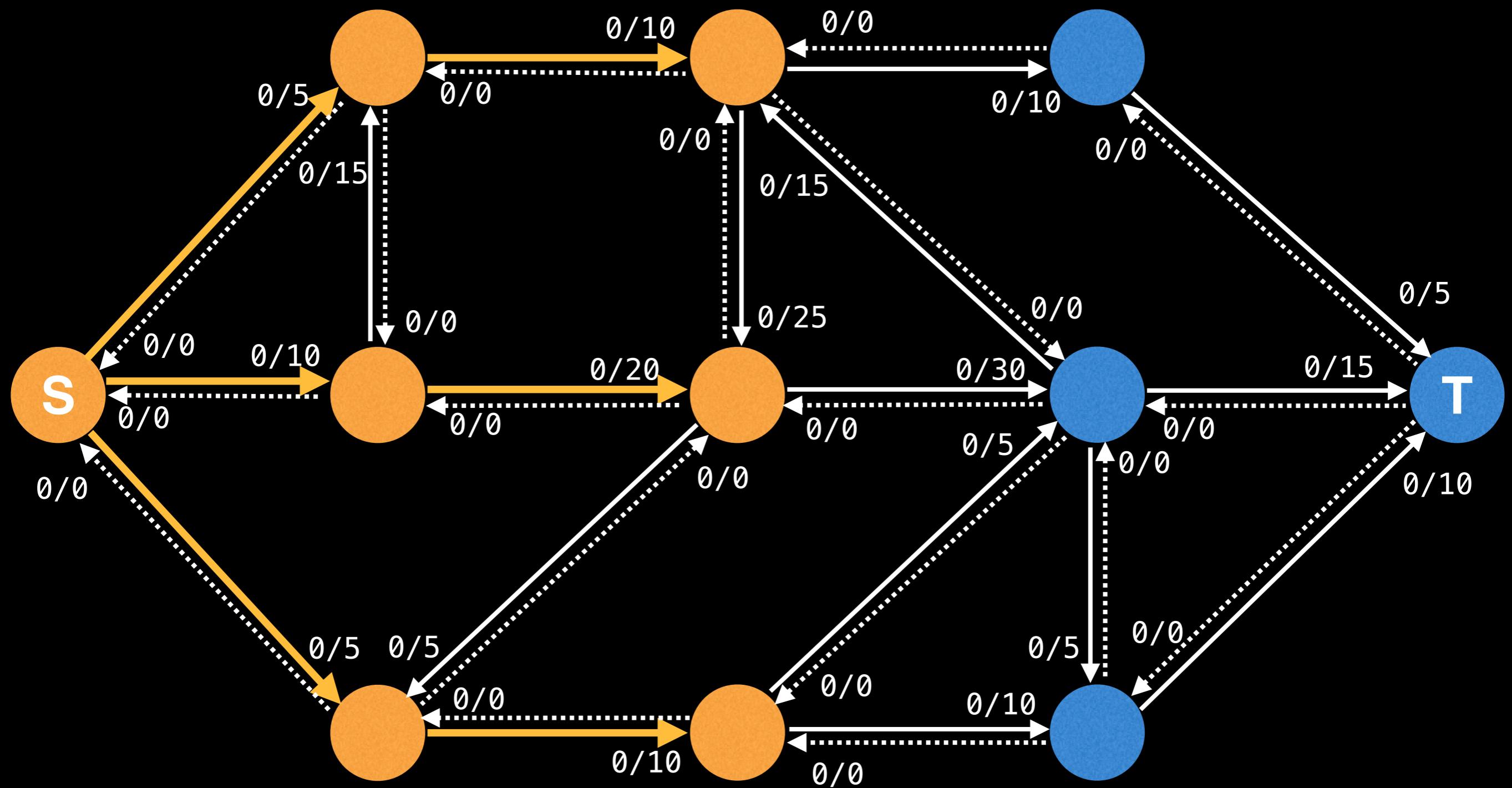
Do a Breadth First Search (BFS) starting at the source and ending at the sink. While exploring the flow graph, remember that we can only reach a node if the capacity of the edge to get there is greater than 0.



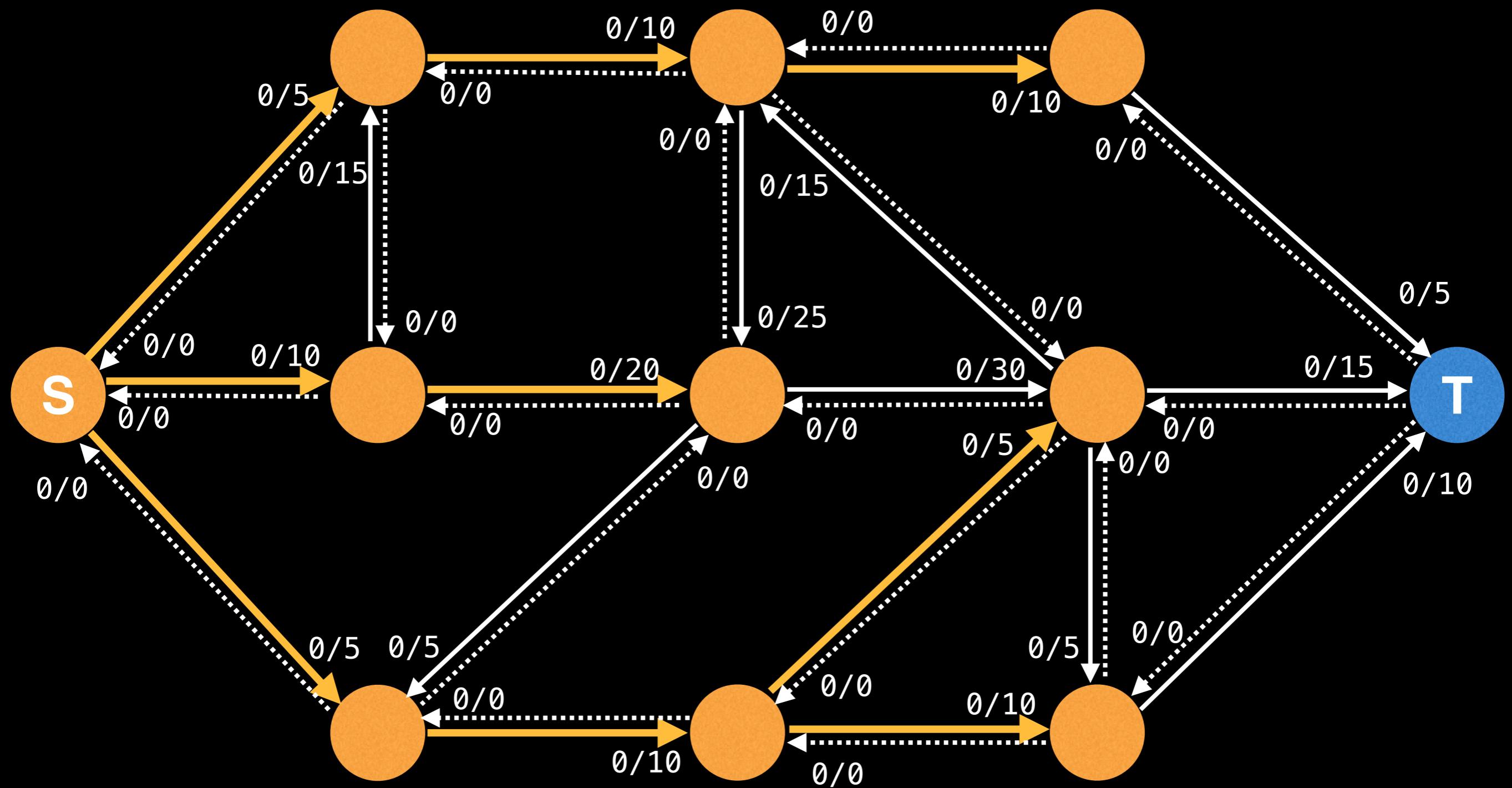
Add all reachable neighbours to the queue and proceed...



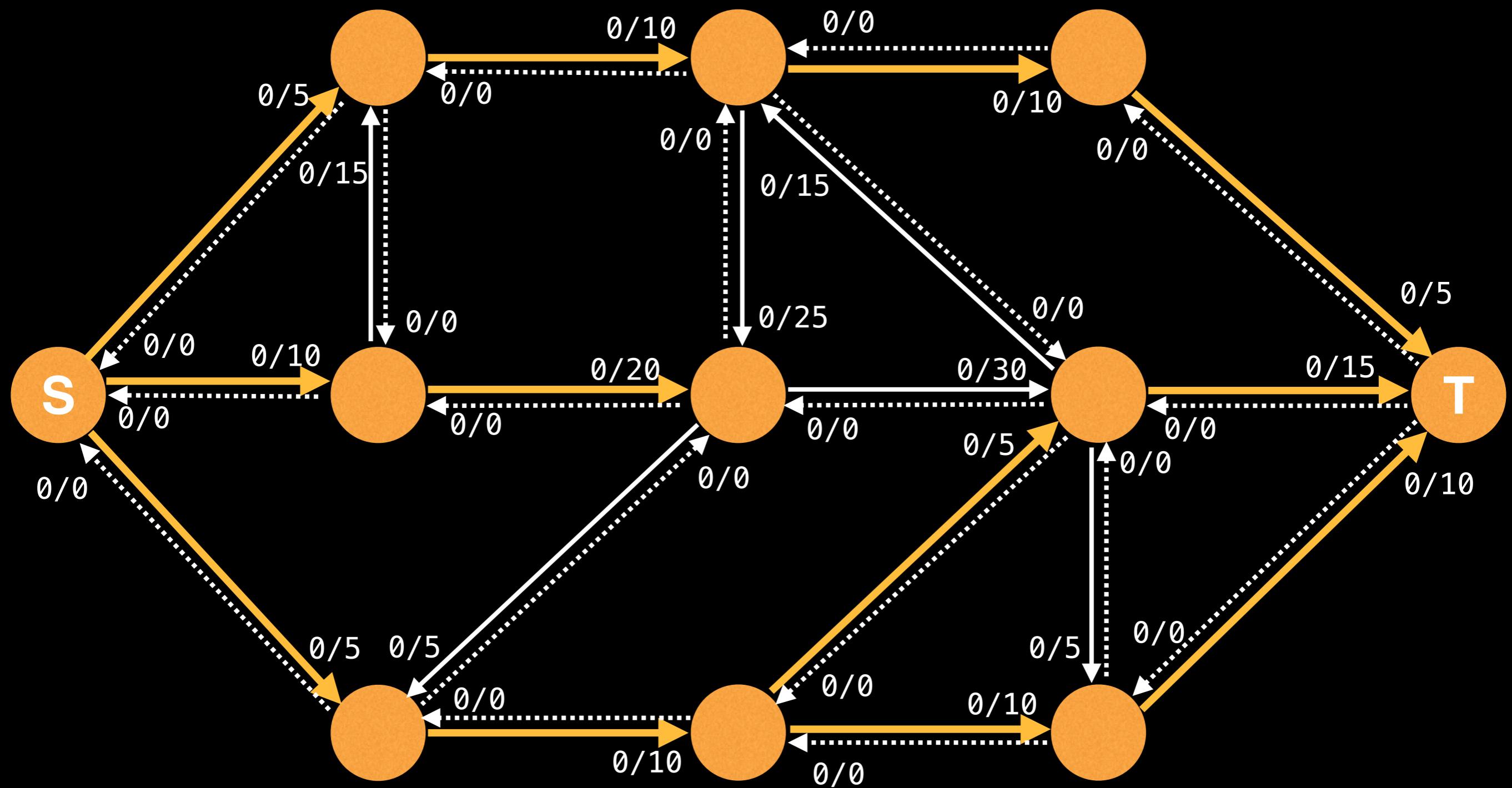
Add all reachable neighbours to the queue and proceed...



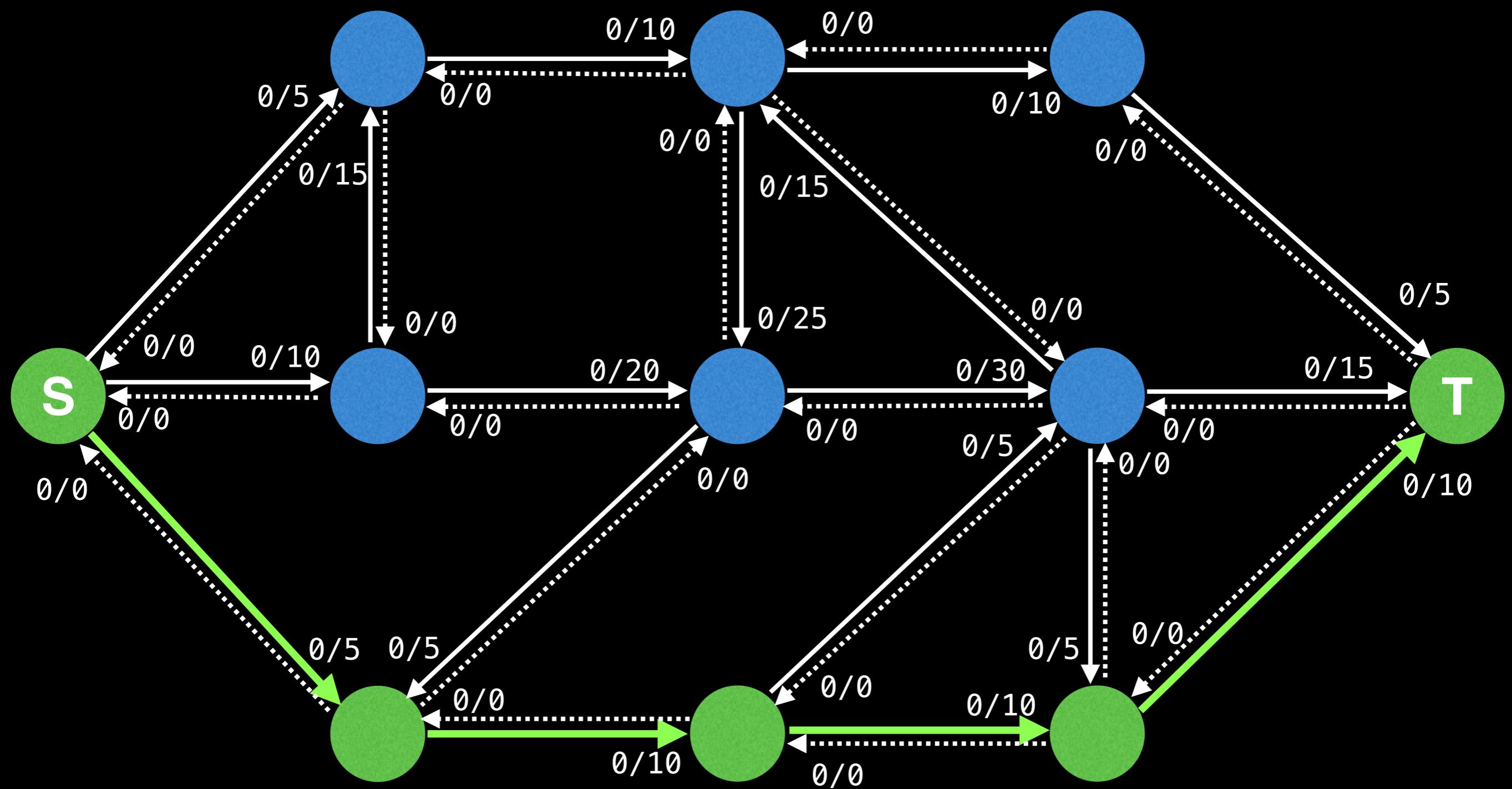
Add all reachable neighbours to the queue and proceed...



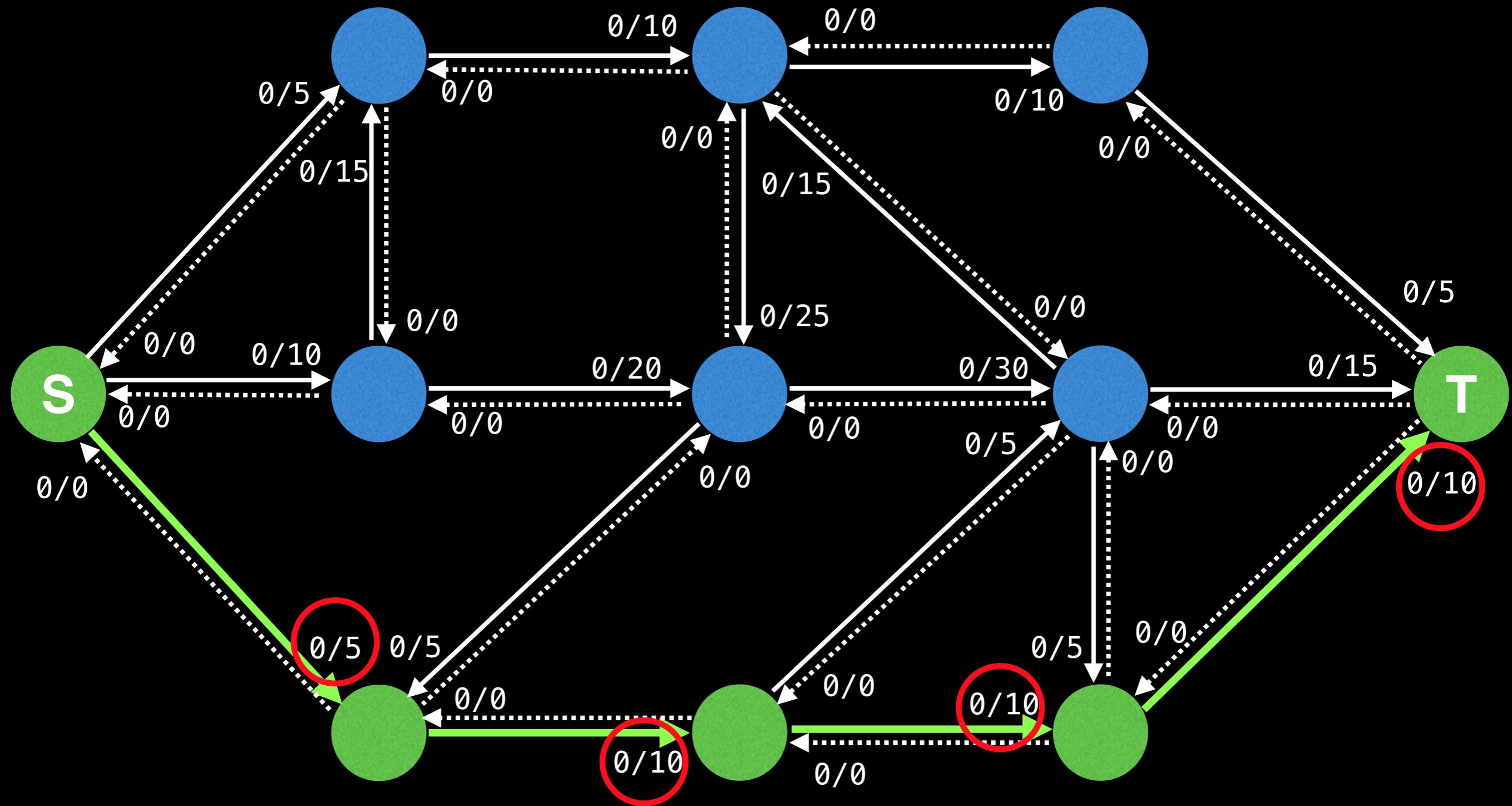
Add all reachable neighbours to the queue and proceed...



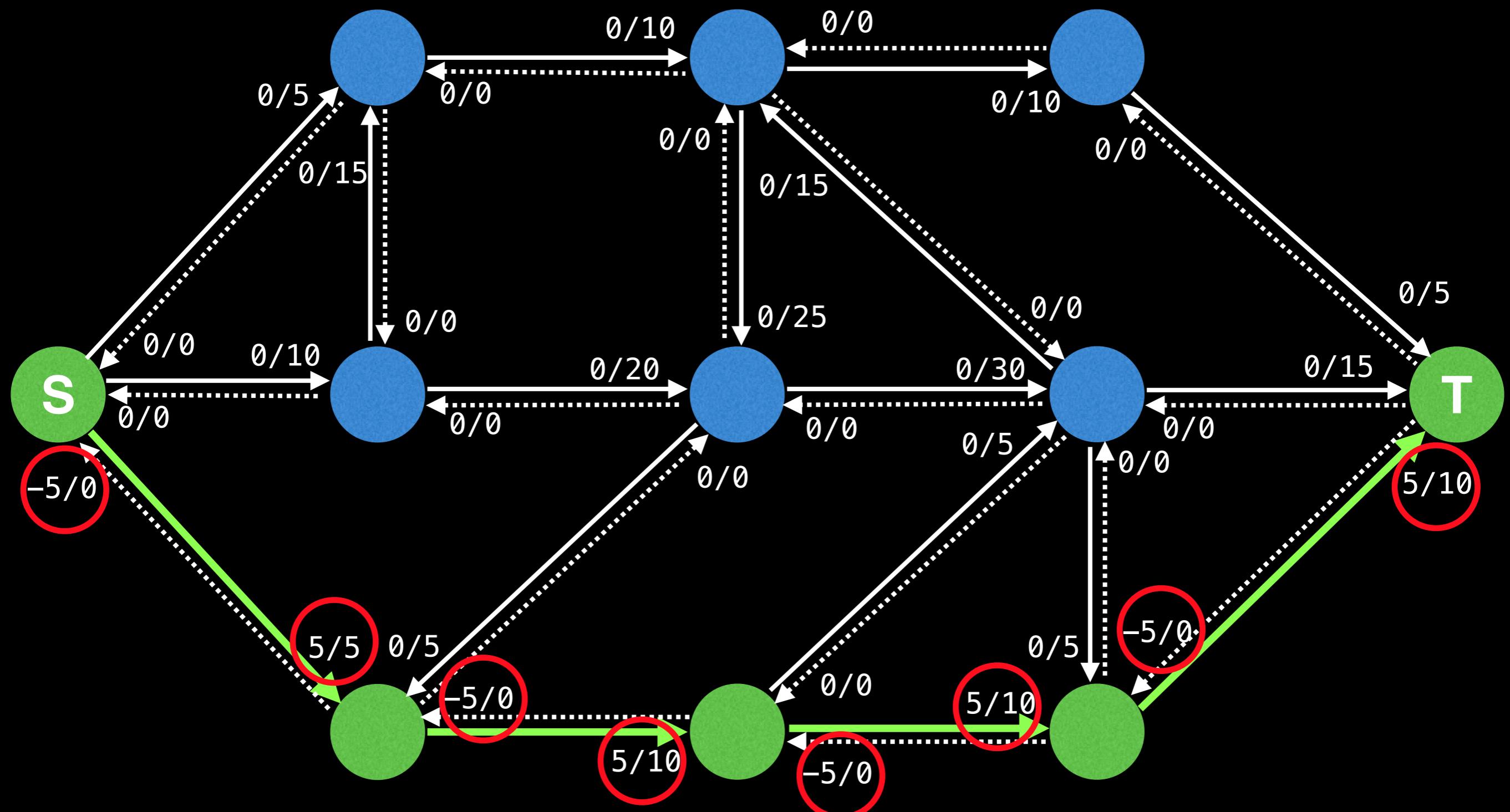
When a complete path from  $s \rightarrow t$  is found, get the augmenting path.



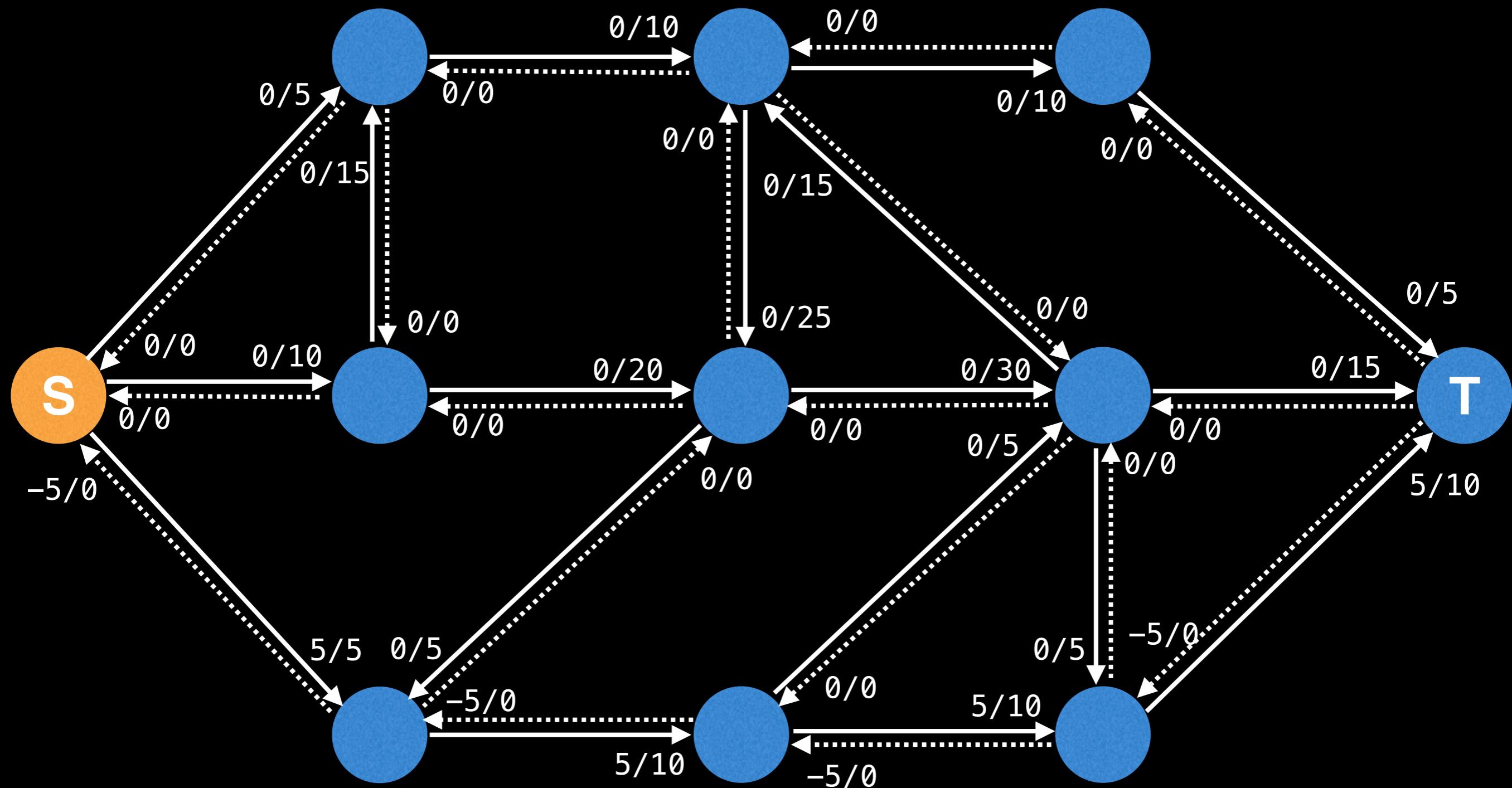
Find the bottleneck value:  
 $\min(5-0, 10-0, 10-0, 10-0) = 5$



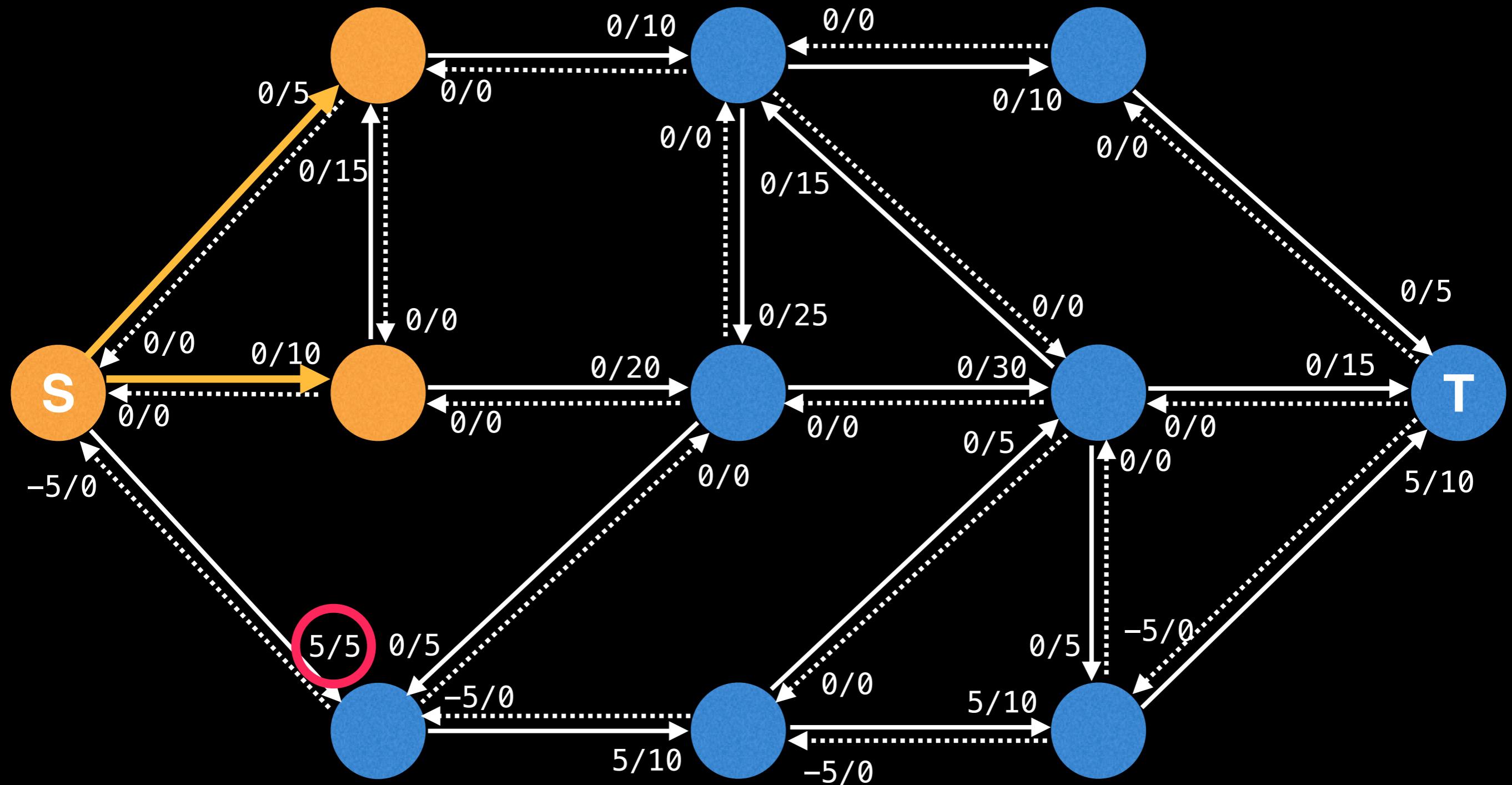
Update (augment) the flow of the forward and residual edges along the path by the bottleneck value (5).



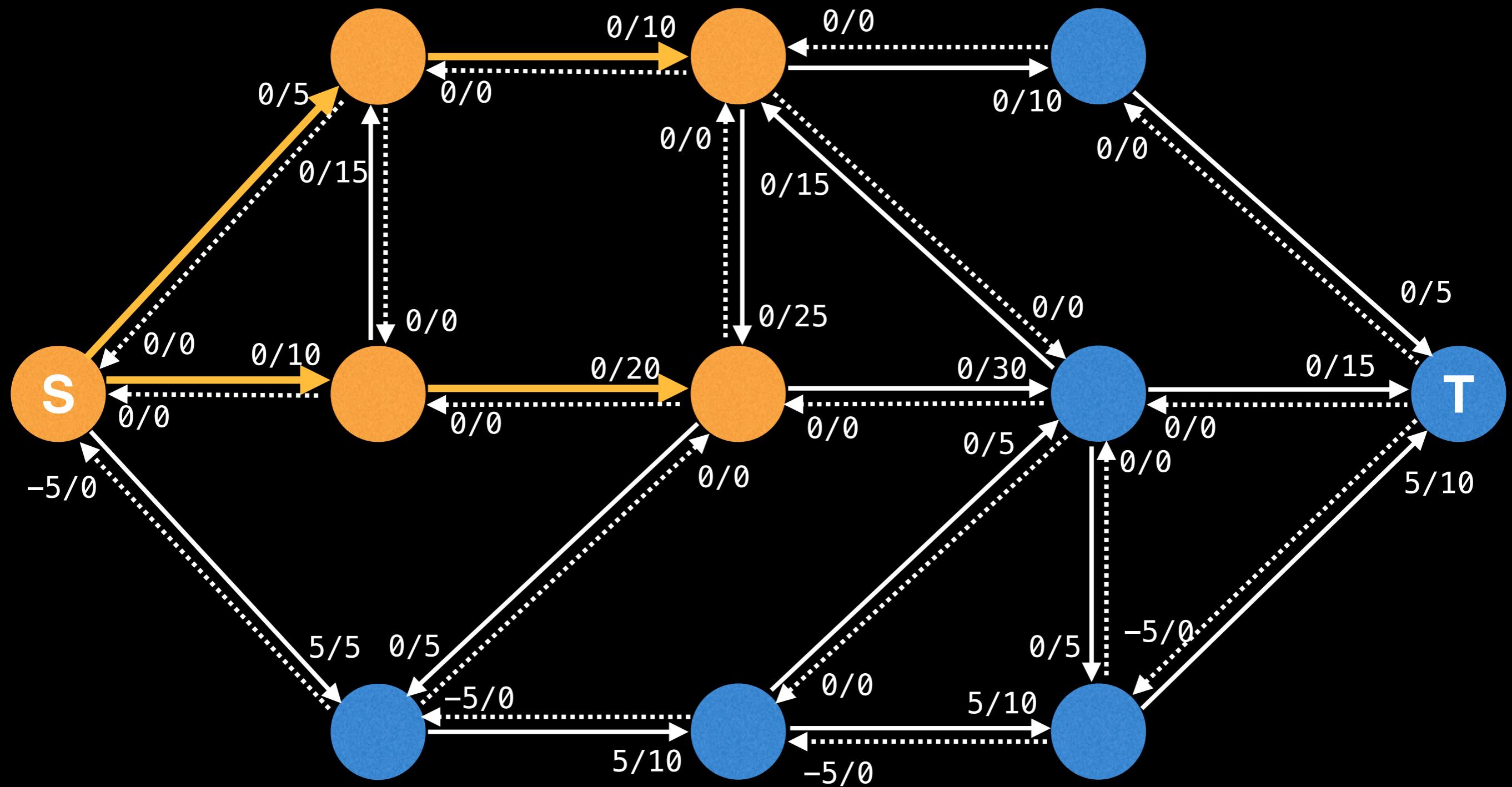
Do a Breadth First Search (BFS) starting at the source and ending at the sink. While exploring the flow graph remember that we can only reach a node if the capacity of the edge to get there is greater than 0



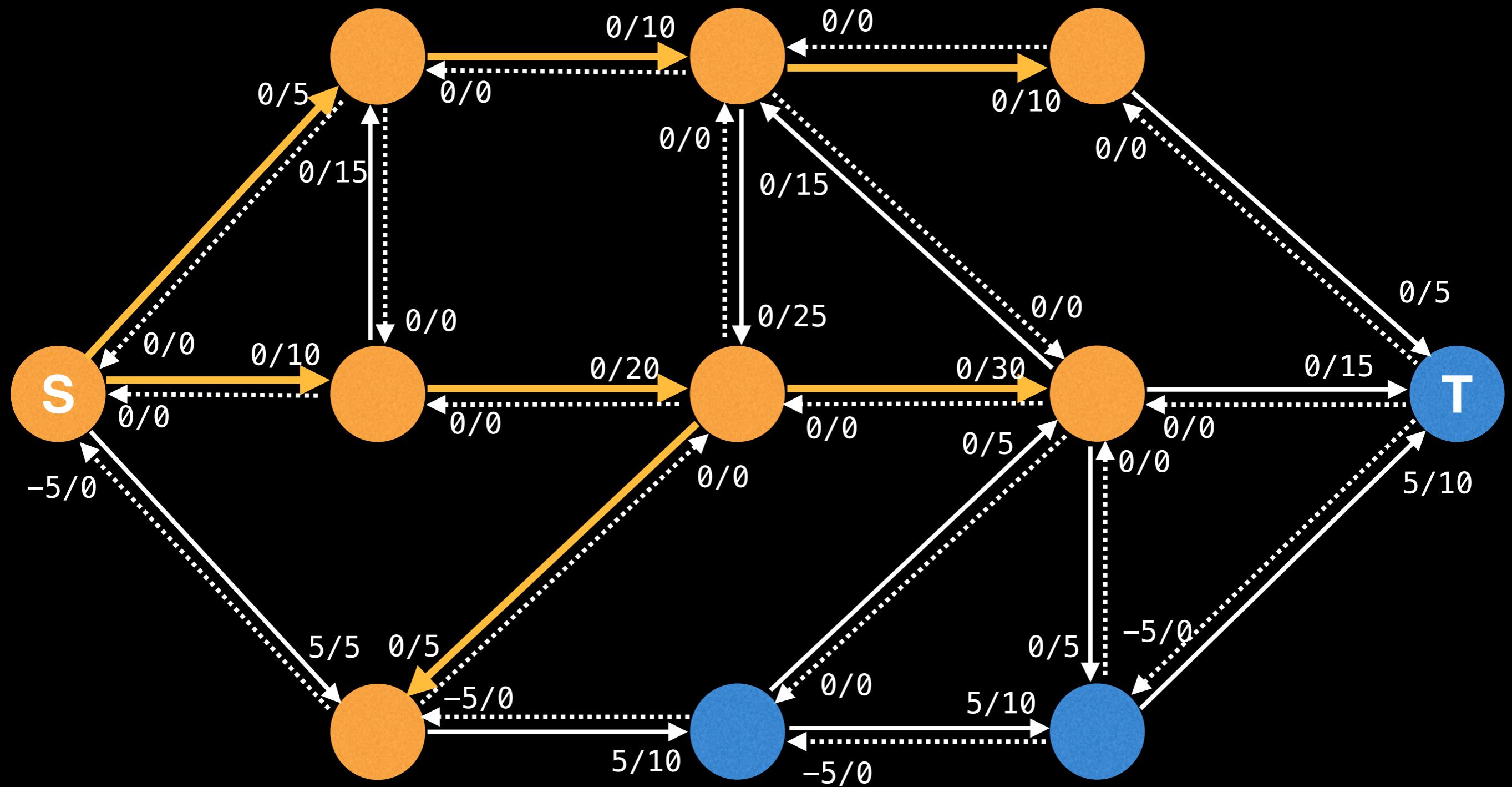
Add all **reachable neighbours** to the queue and proceed...  
The bottom left node here is unreachable because the edge from the source leading to it has a capacity of 0.



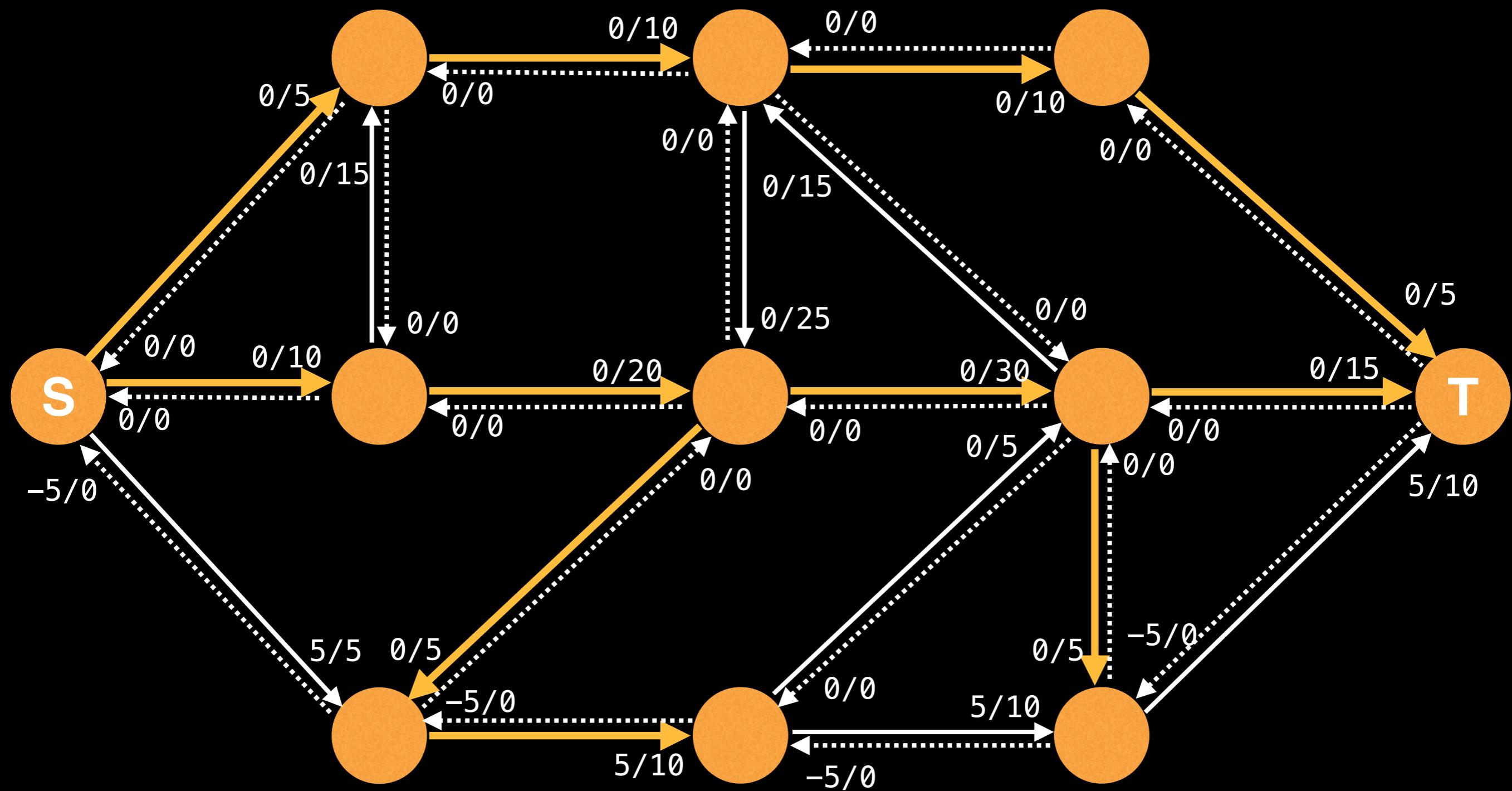
Add all reachable neighbours to the queue and proceed...



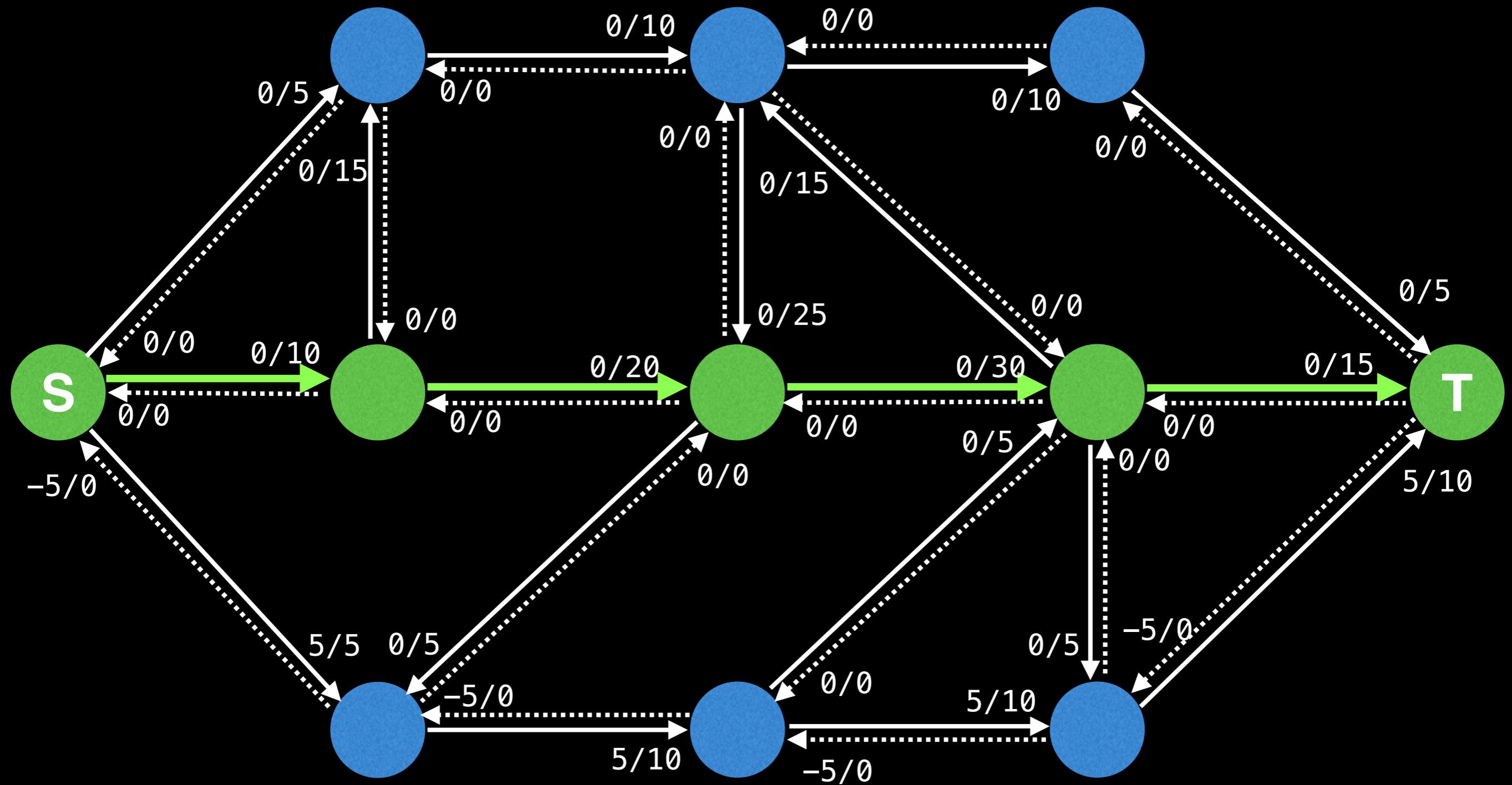
Add all reachable neighbours to the queue and proceed...



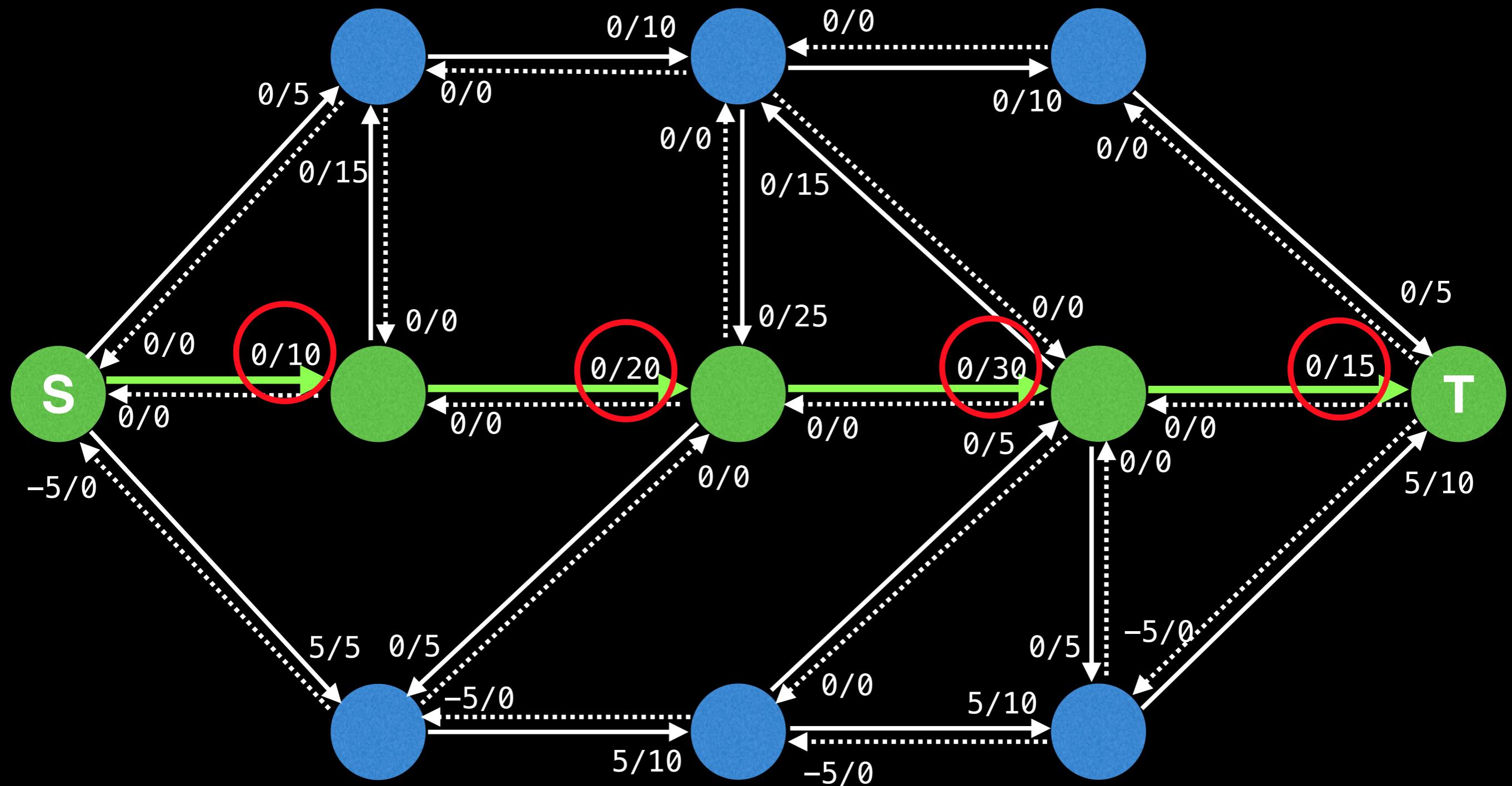
Add all reachable neighbours to the queue and proceed...



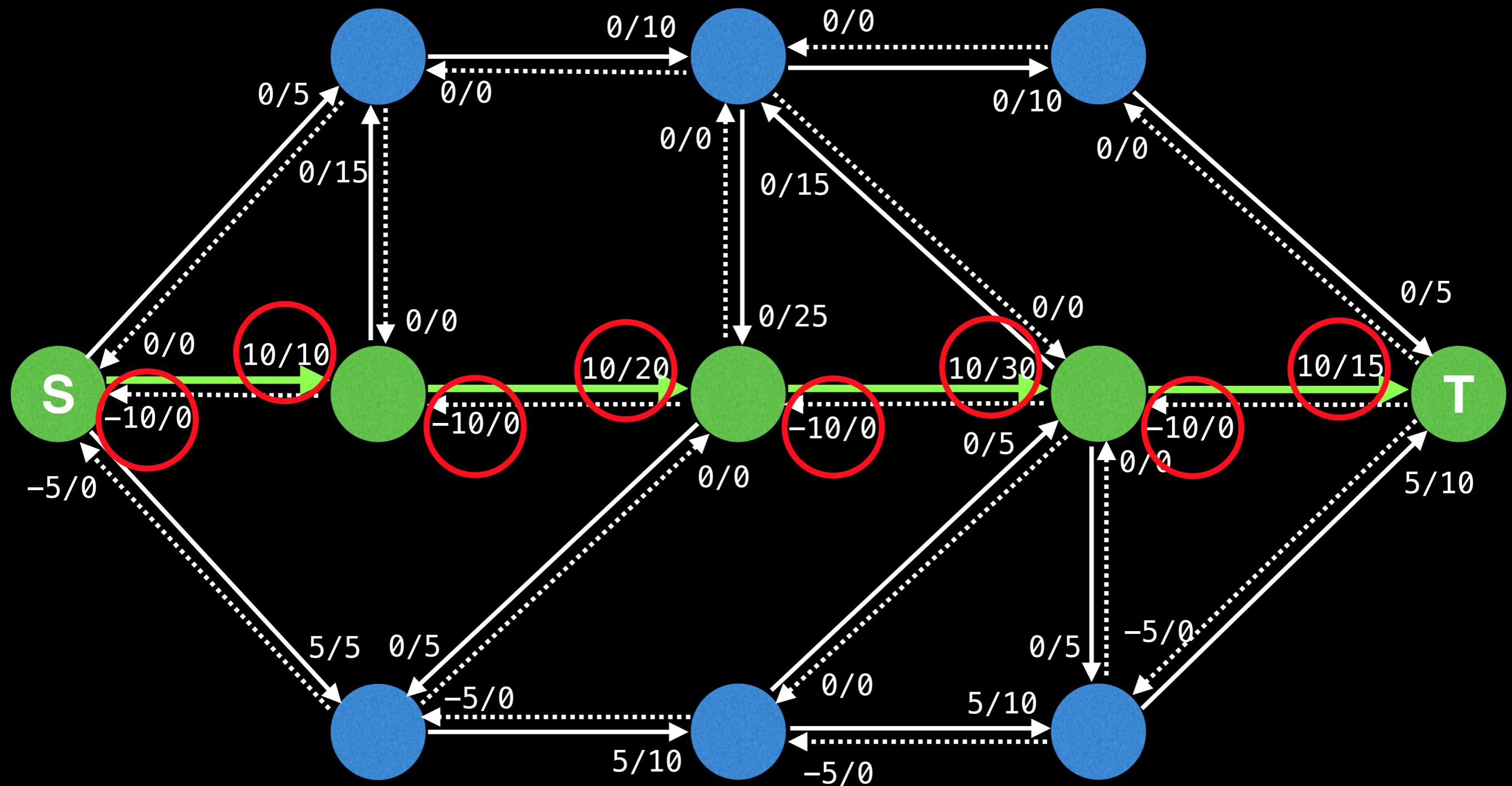
# Reconstruct the augmenting path

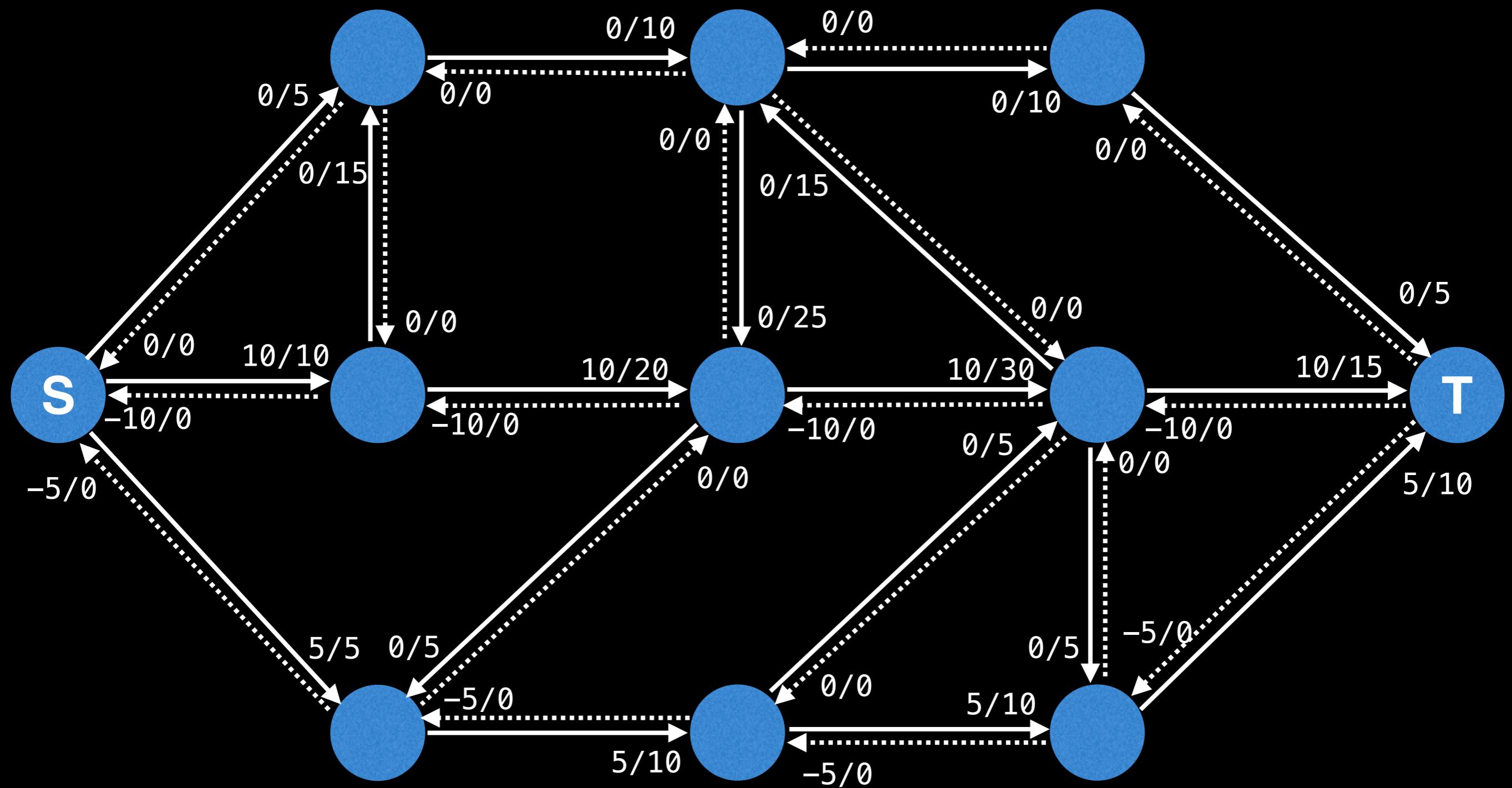


Find the bottleneck value:  
**min**(10-0, 20-0, 30-0, 15-0) = 10

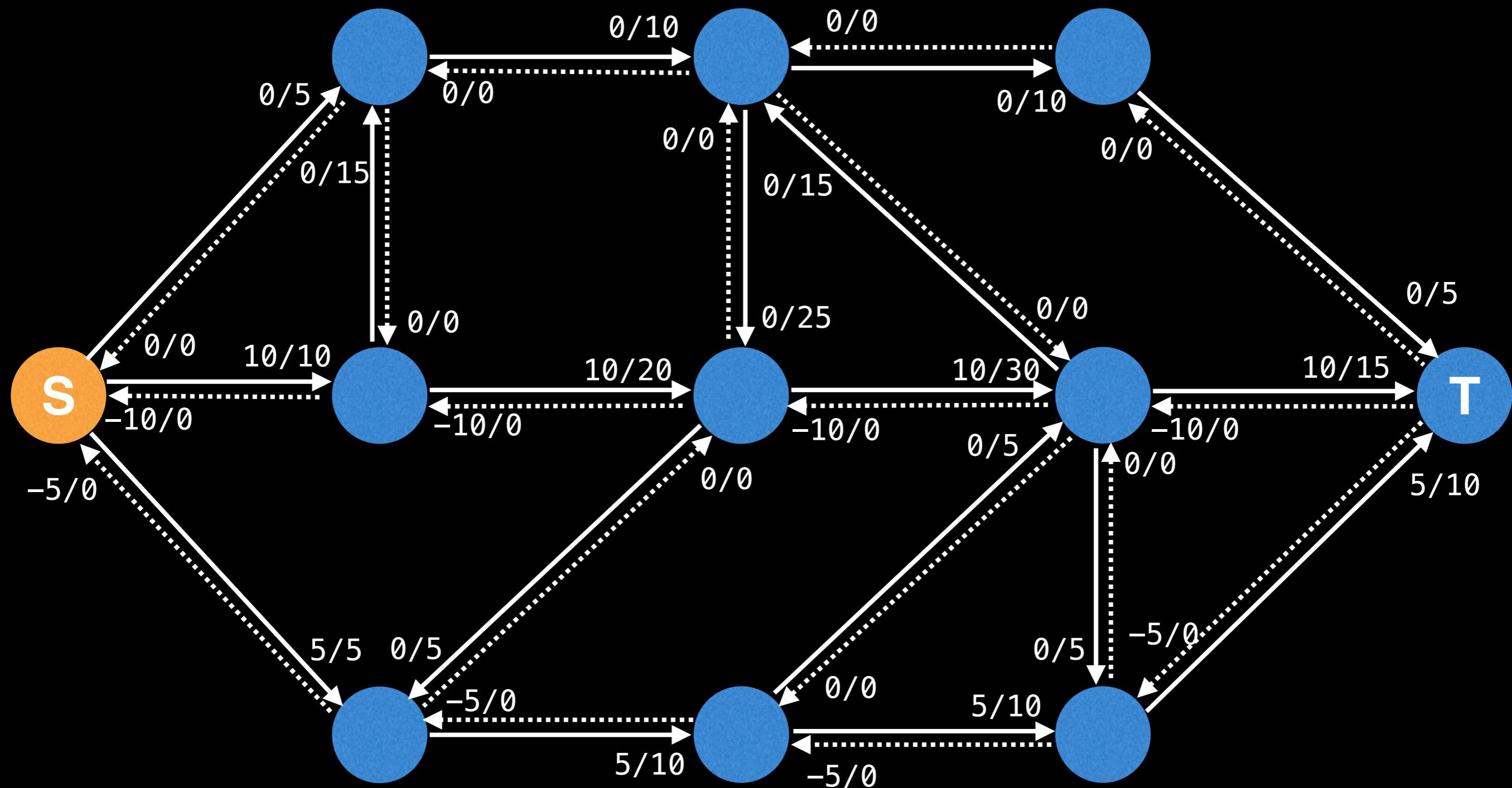


Update (augment) the flow of the forward and residual edges along the path by the bottleneck value (10).

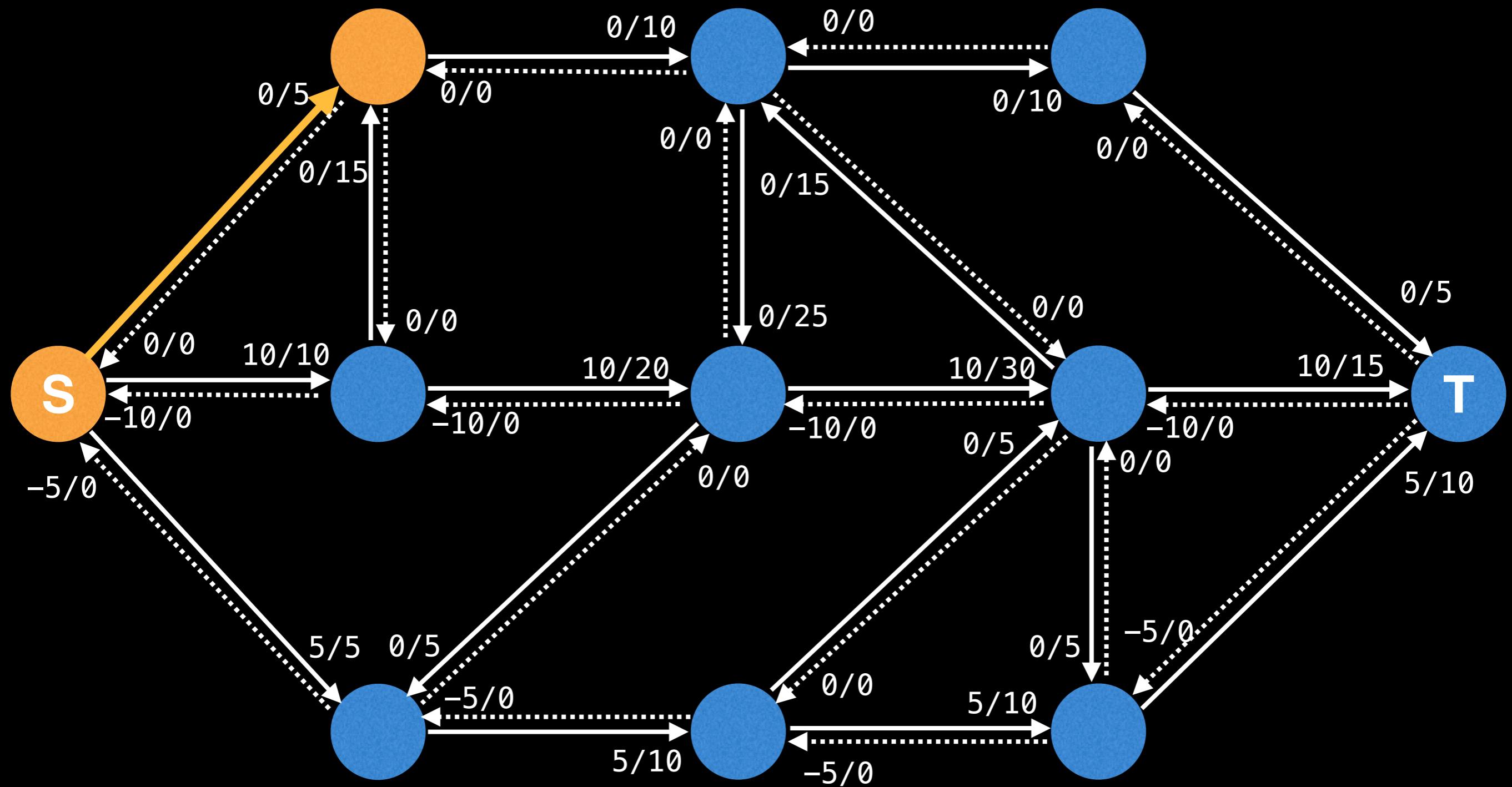




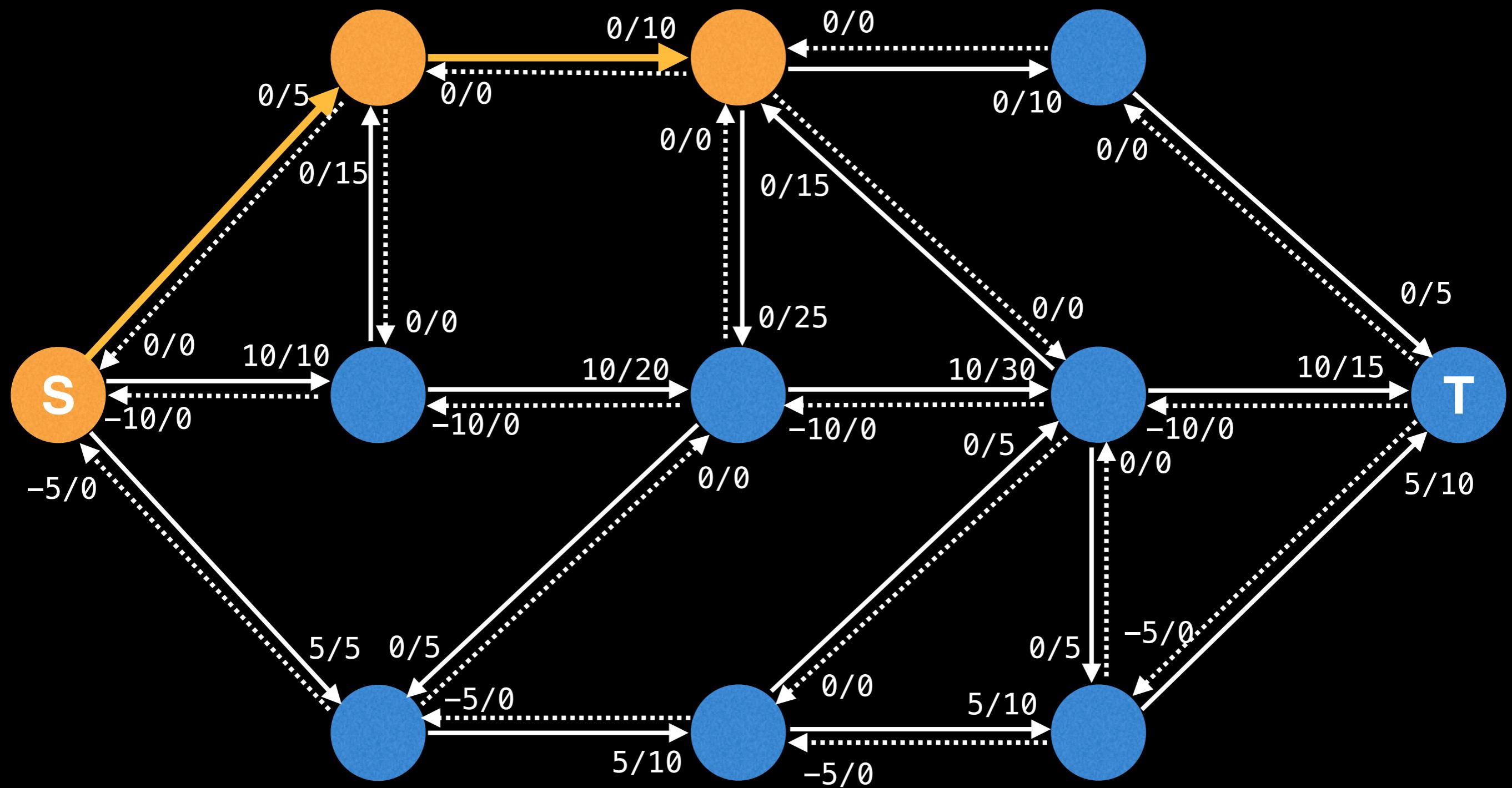
Do a Breadth First Search (BFS) starting at the source and ending at the sink. While exploring the flow graph remember that we can only reach a node if the capacity of the edge to get there is greater than 0



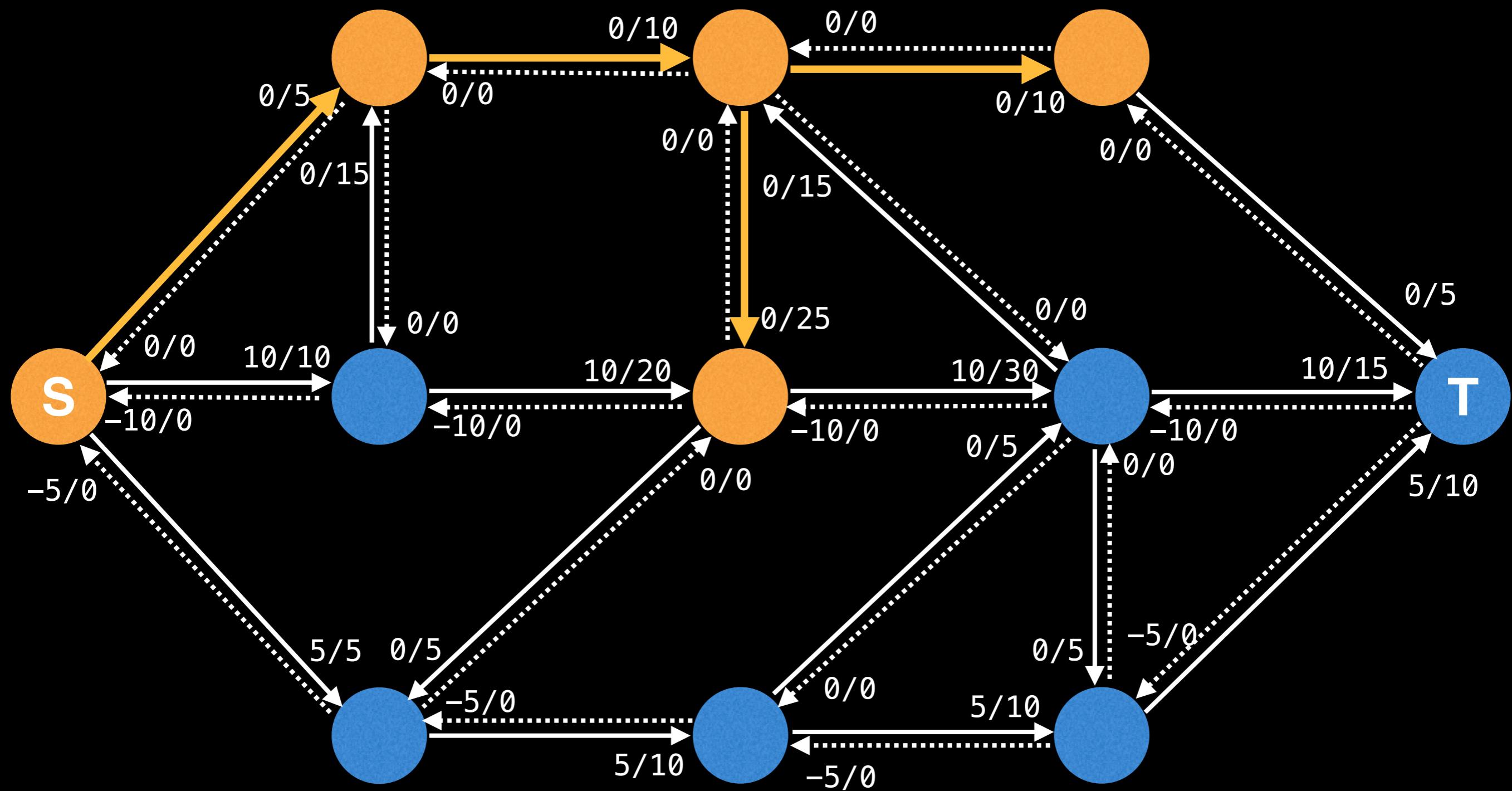
Add all reachable neighbours to the queue and proceed...



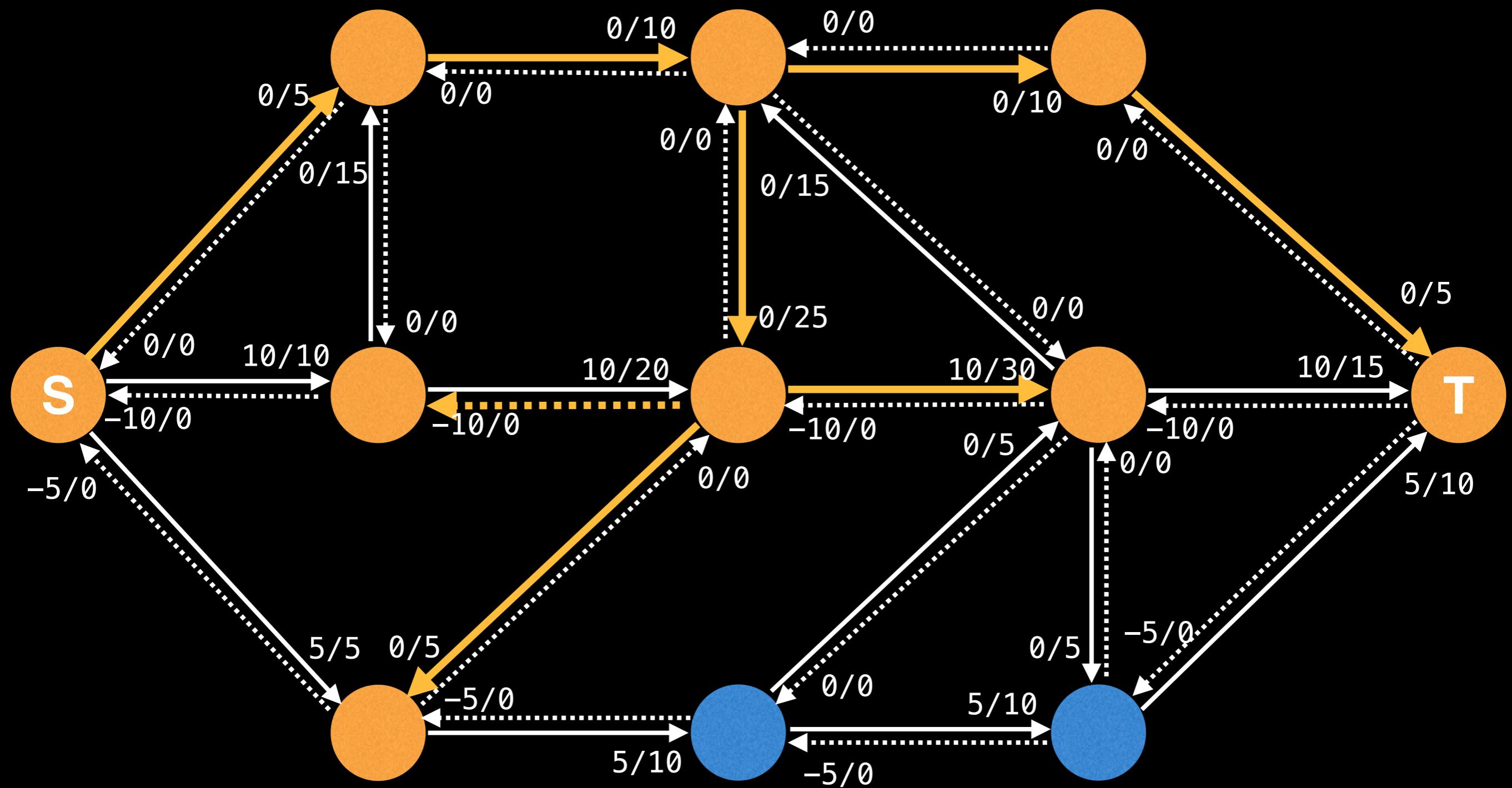
Add all reachable neighbours to the queue and proceed...



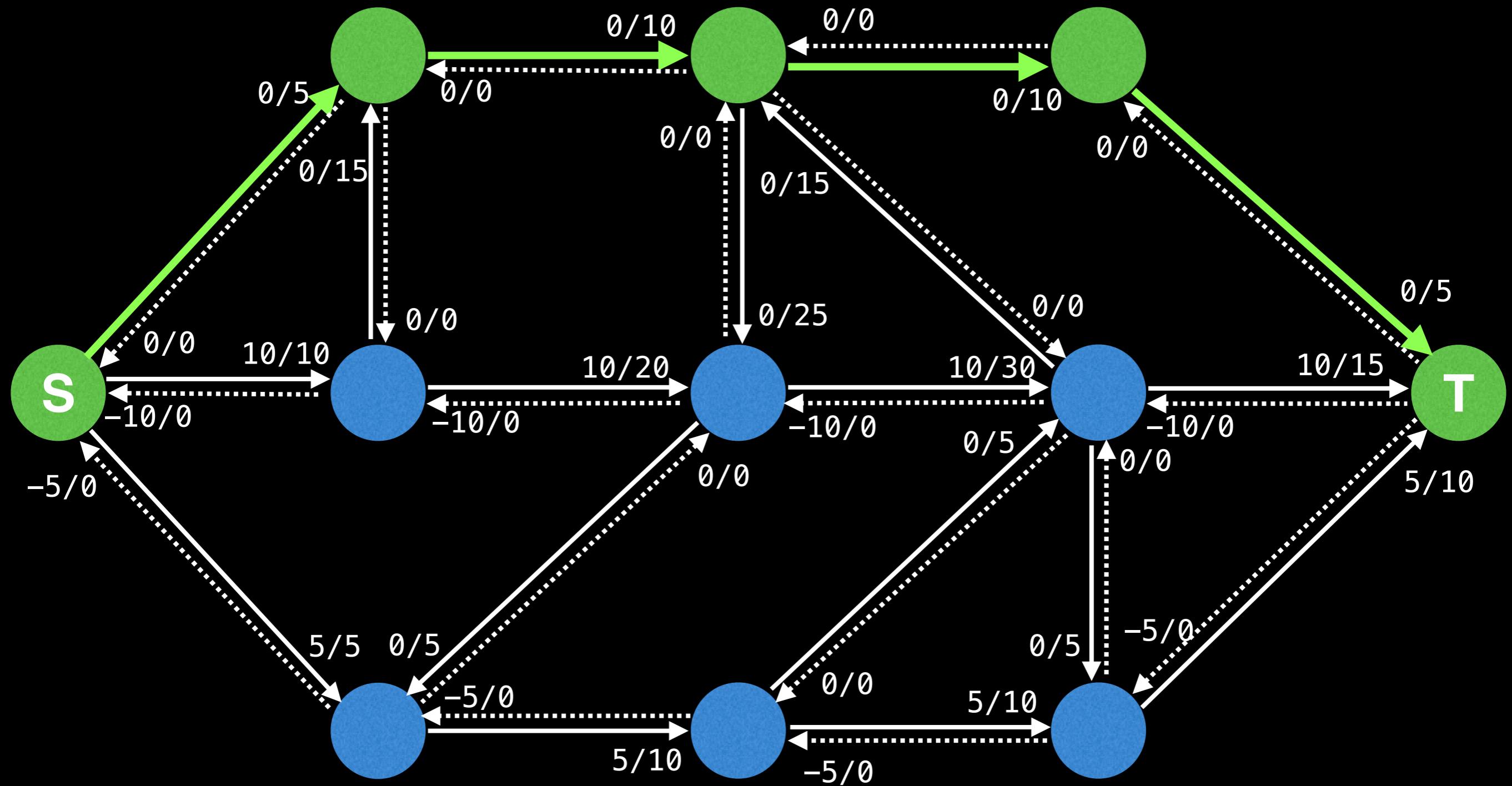
Add all reachable neighbours to the queue and proceed...



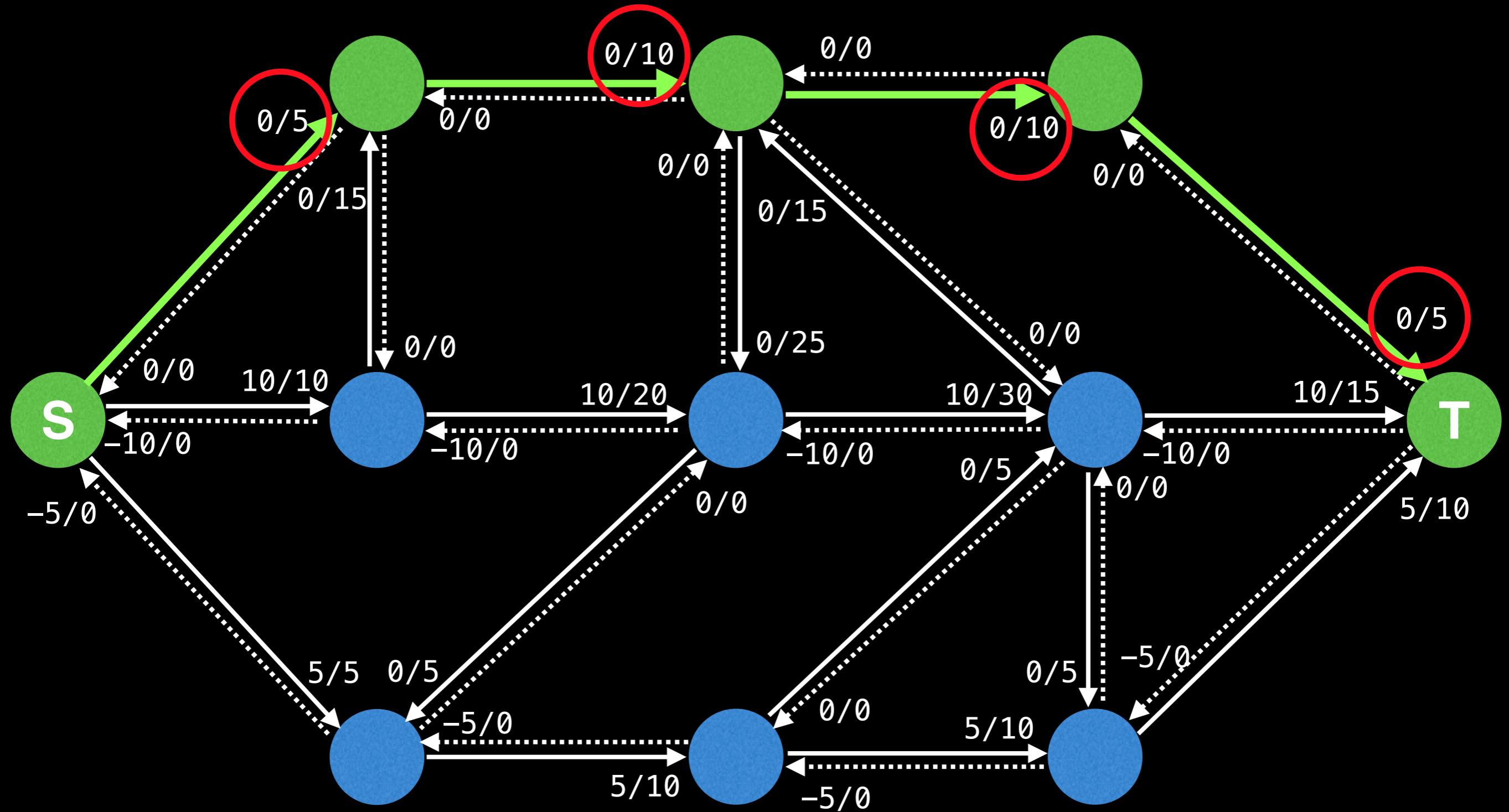
Add all reachable neighbours to the queue and proceed...



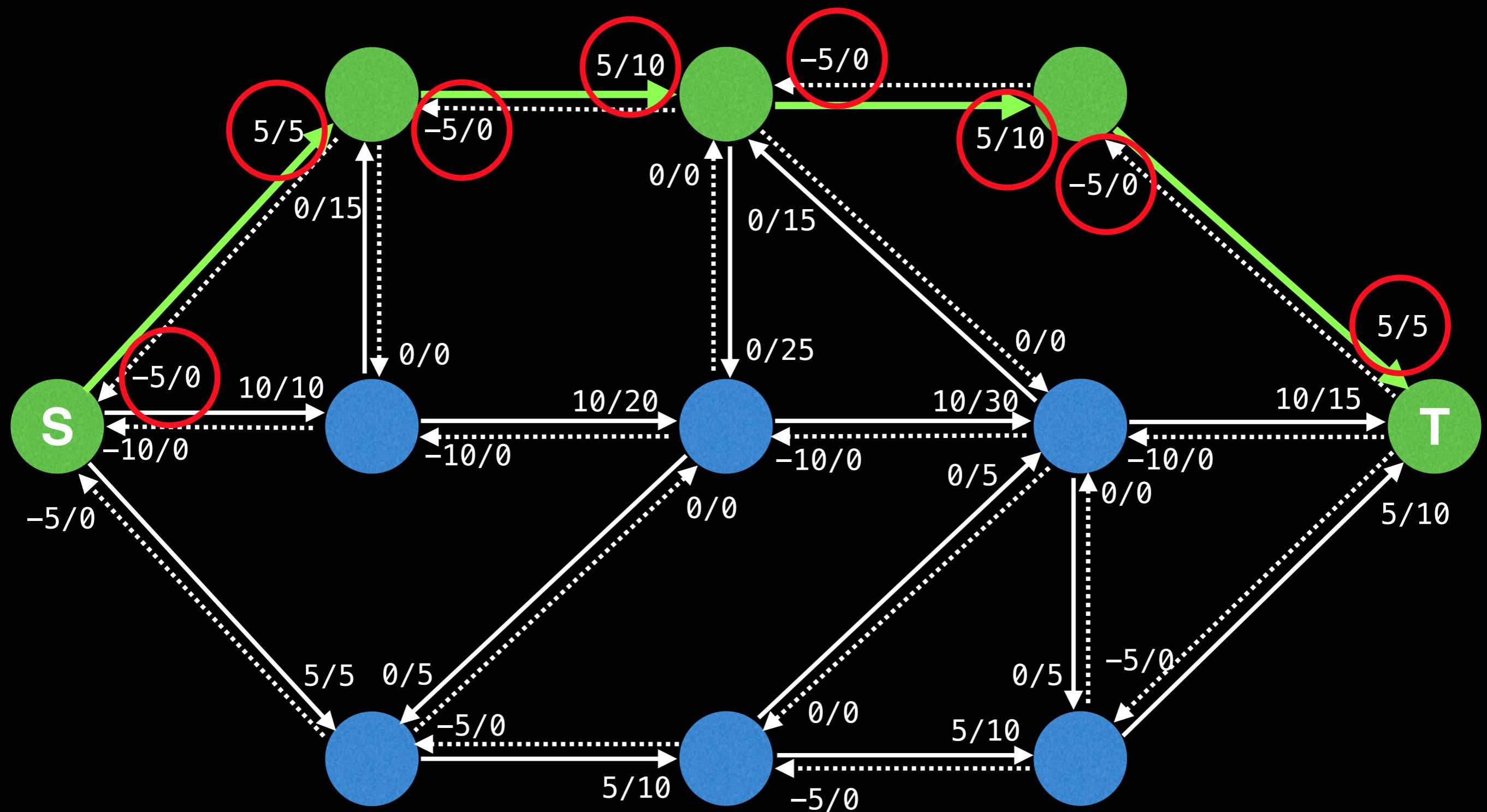
# Reconstruct the augmenting path



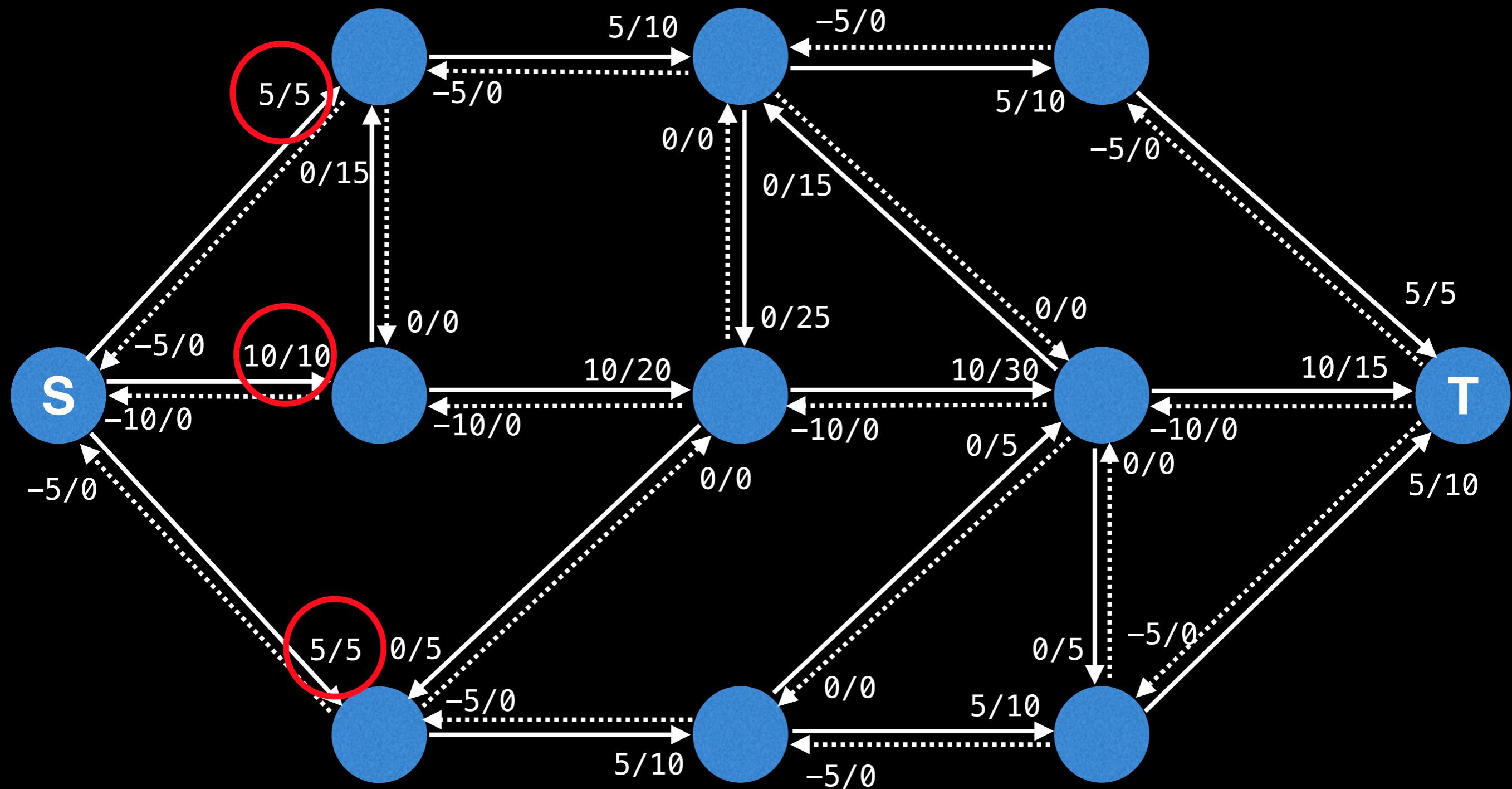
Find the bottleneck value:  
 $\min(5-0, 10-0, 10-0, 5-0) = 5$



Update (augment) the flow of the forward and residual edges along the path by the bottleneck value (5).

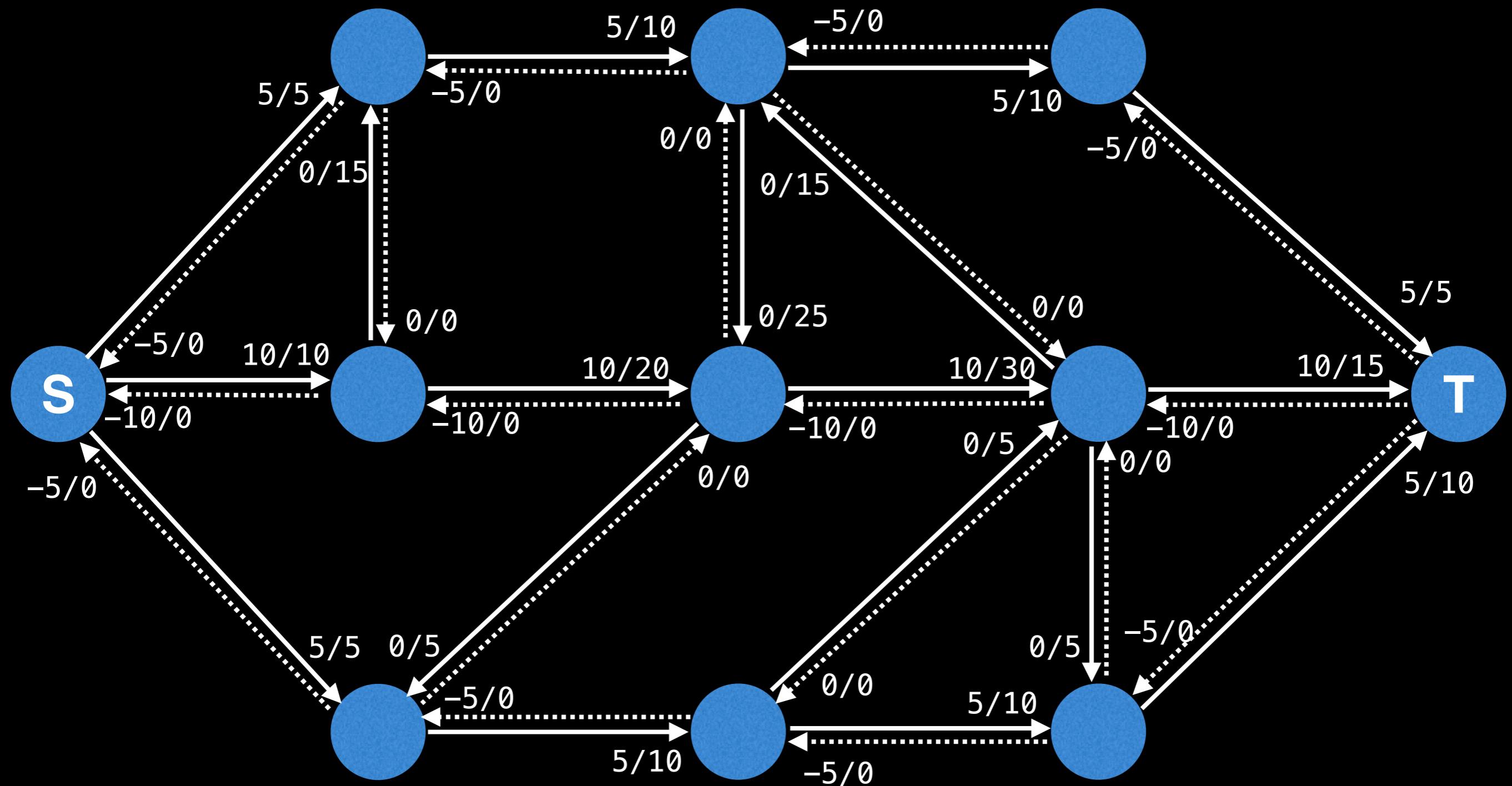


There are no more augmenting paths left to be found because all the edges leading outwards from the source have a remaining capacity of 0. However, more generally we know to stop when there are no more augmenting paths from  $s \rightarrow t$



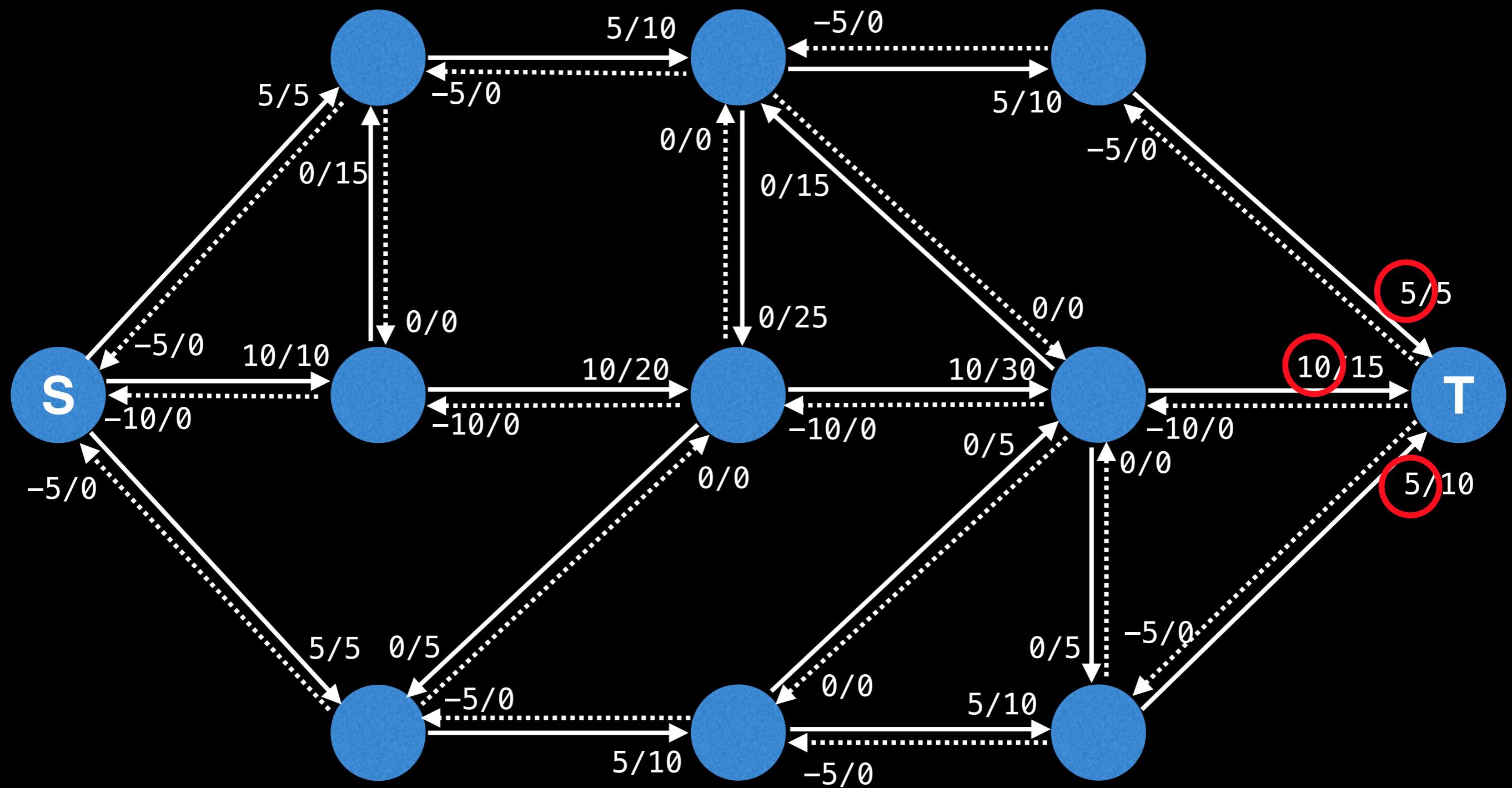
The **max flow** from running Edmonds–Karp is the sum of the bottleneck values:

$$\text{Max flow} = 5 + 10 + 5 = \mathbf{20}$$



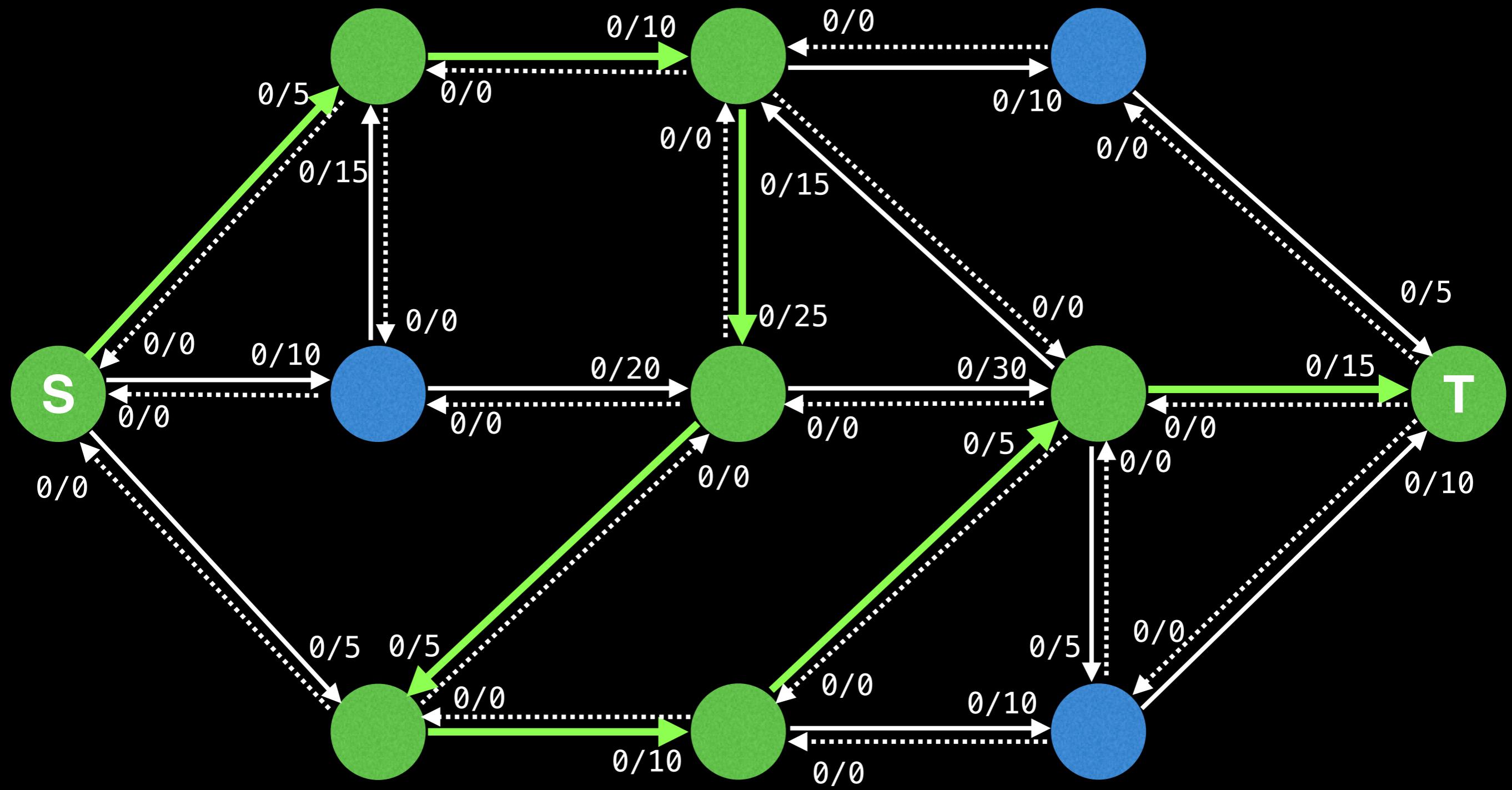
You can also get the **max flow** by summing the capacity values going into the sink.

$$\text{Max flow} = 5 + 10 + 5 = \mathbf{20}$$



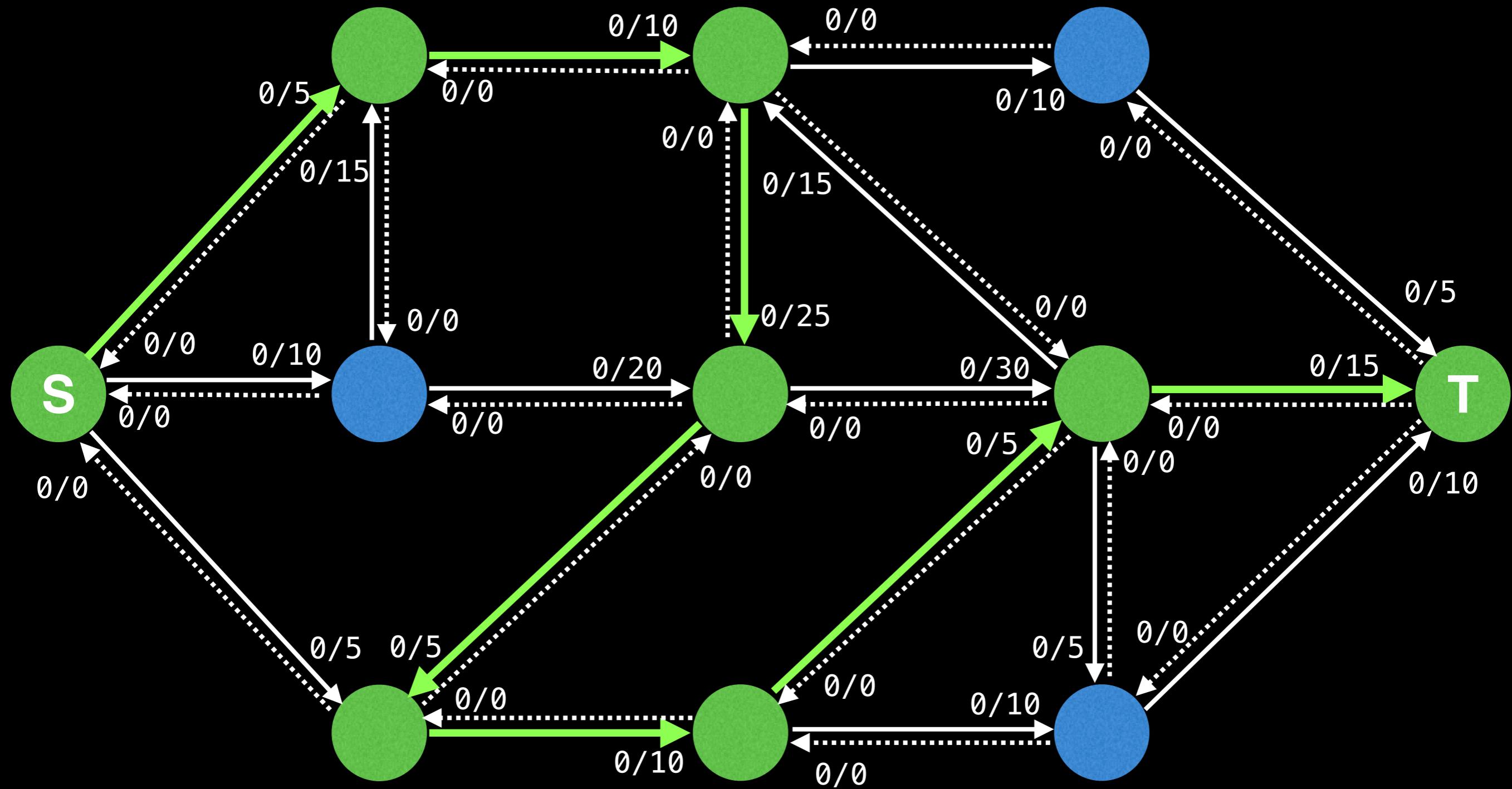
# Summary

A depth first search will sometimes find long windy paths from the source to the sink.



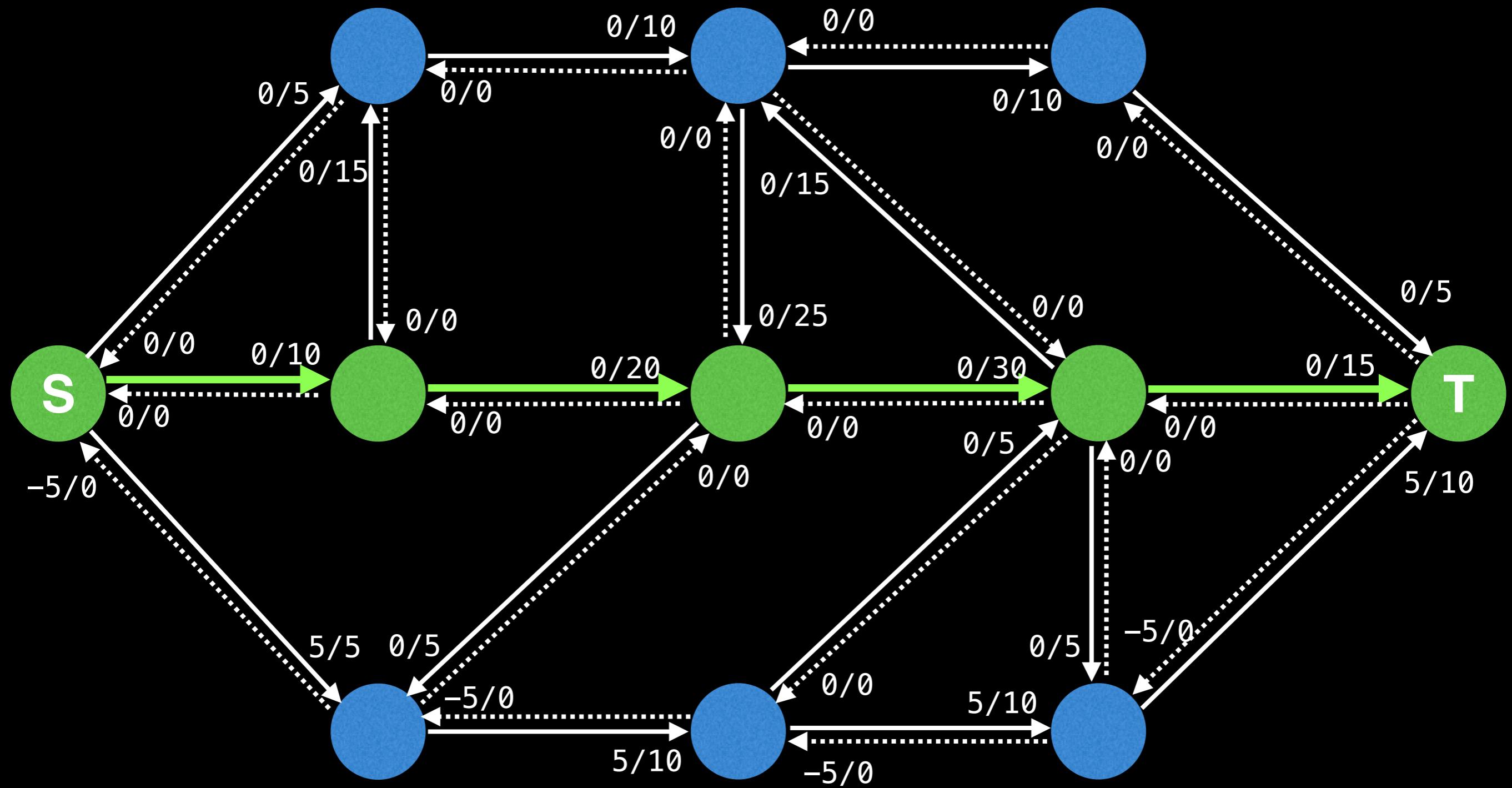
# Summary

This is usually undesirable because the longer the path, the smaller the bottleneck value, which results in a longer runtime.



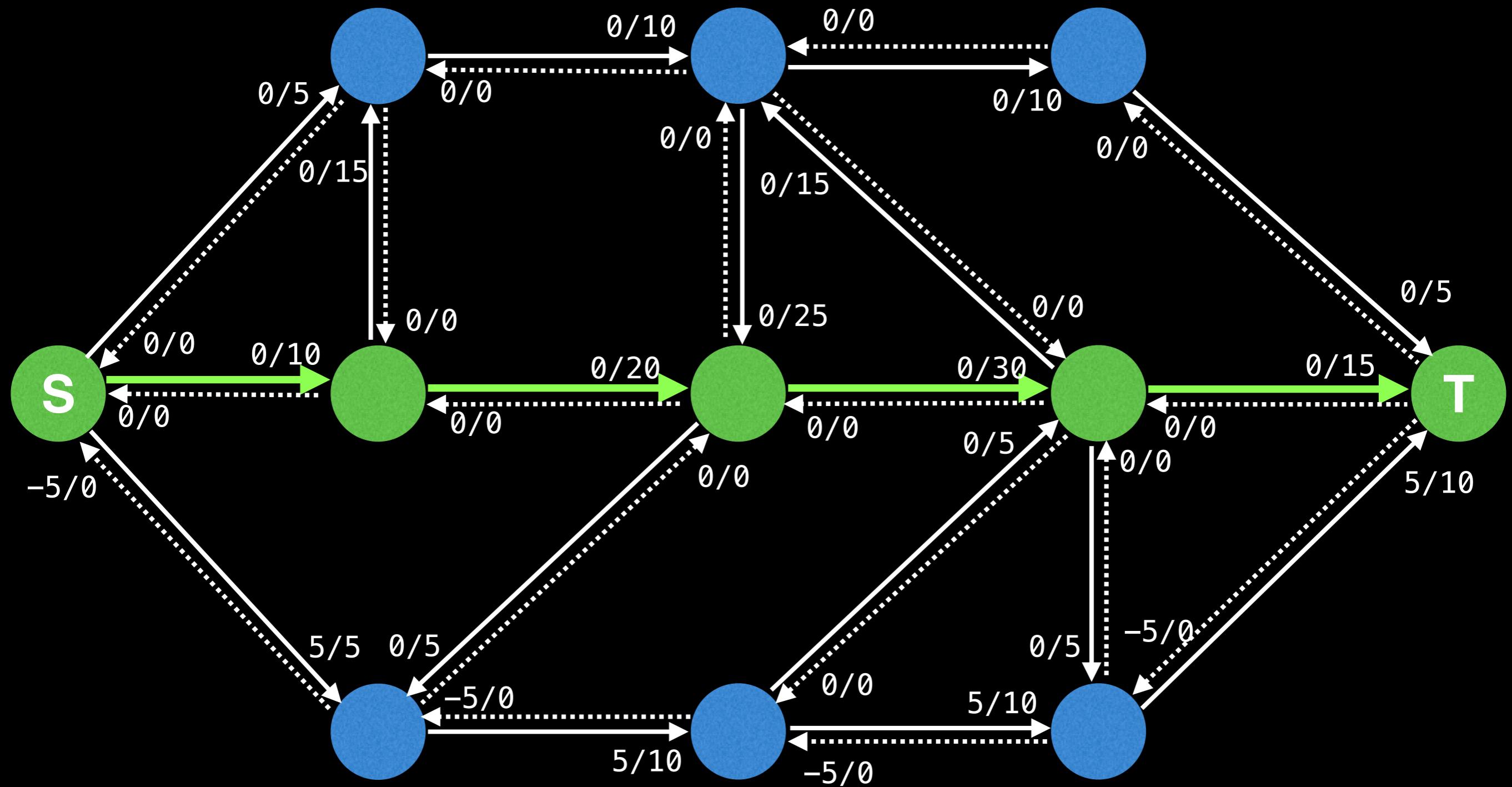
# Summary

Edmonds-Karp tries to resolve this problem by finding the shortest path from the source to the sink using a breadth first search.



# Summary

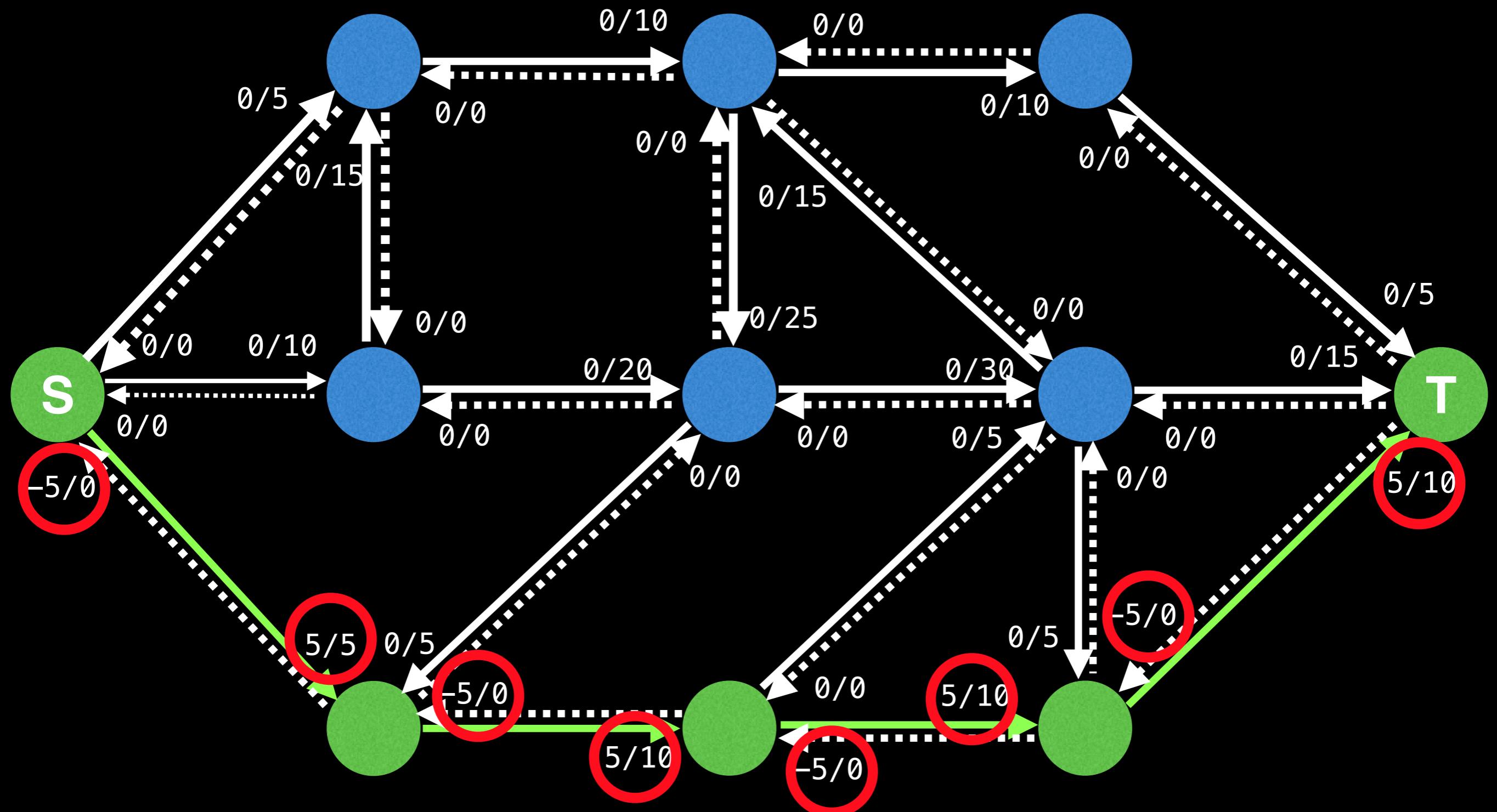
More importantly, the big achievement of Edmonds–Karp is that its time complexity of  $O(VE^2)$  is **independent** of the max flow.



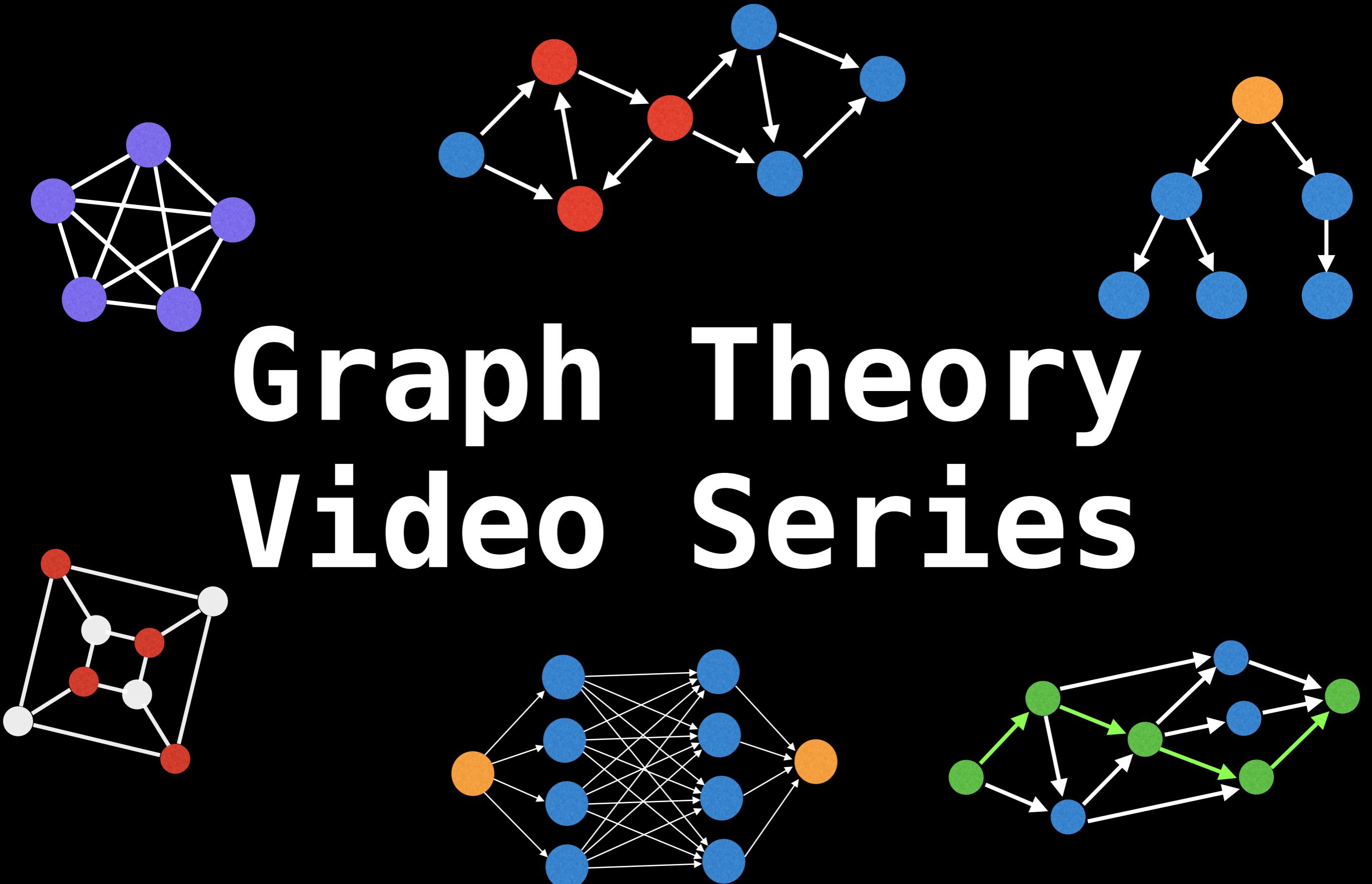
Next Video: Edmonds-Karp Source Code



# Network Flow: Edmonds-Karp



# Graph Theory Video Series



# Network Flow: Edmonds-Karp source code

William Fiset

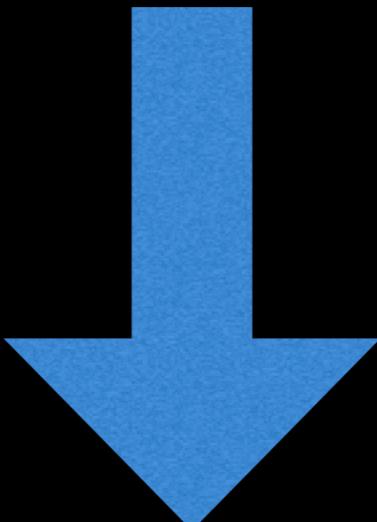
Previous video explaining  
Edmonds-Karp

# Source Code Link

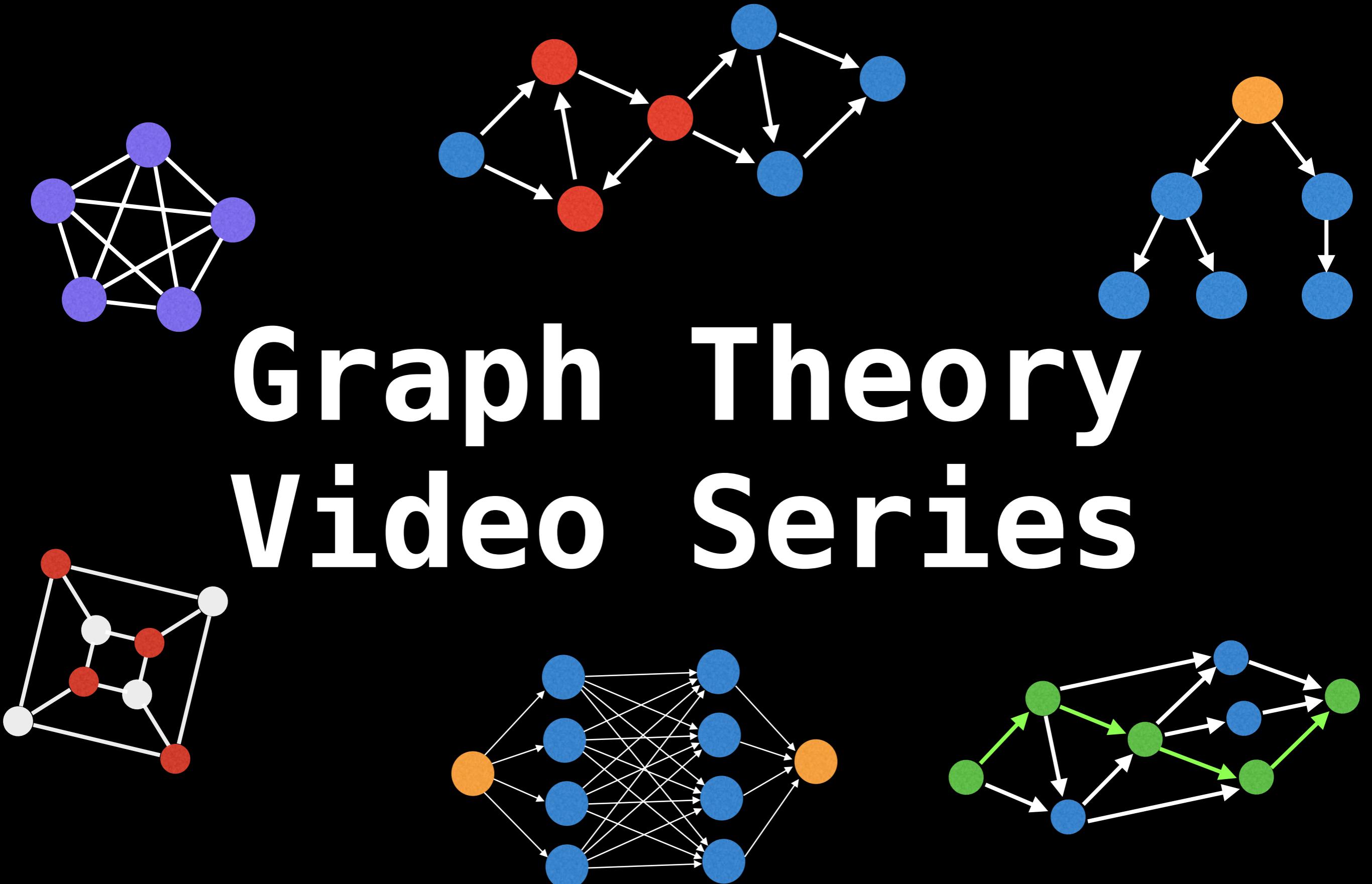
Implementation source code can  
be found at the following link:

[github.com/williamfiset/algorithms](https://github.com/williamfiset/algorithms)

Link in the description:



# Graph Theory Video Series

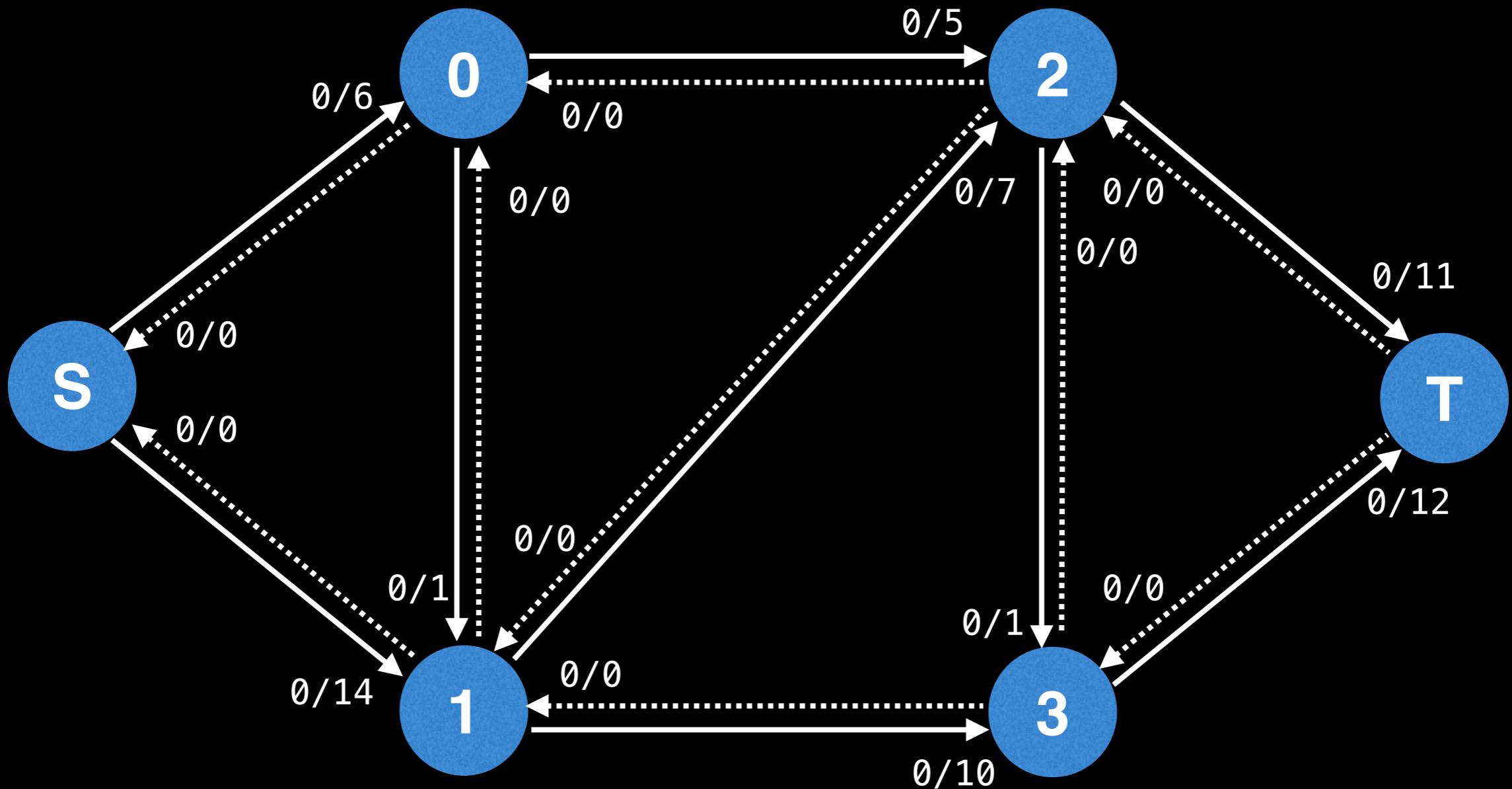


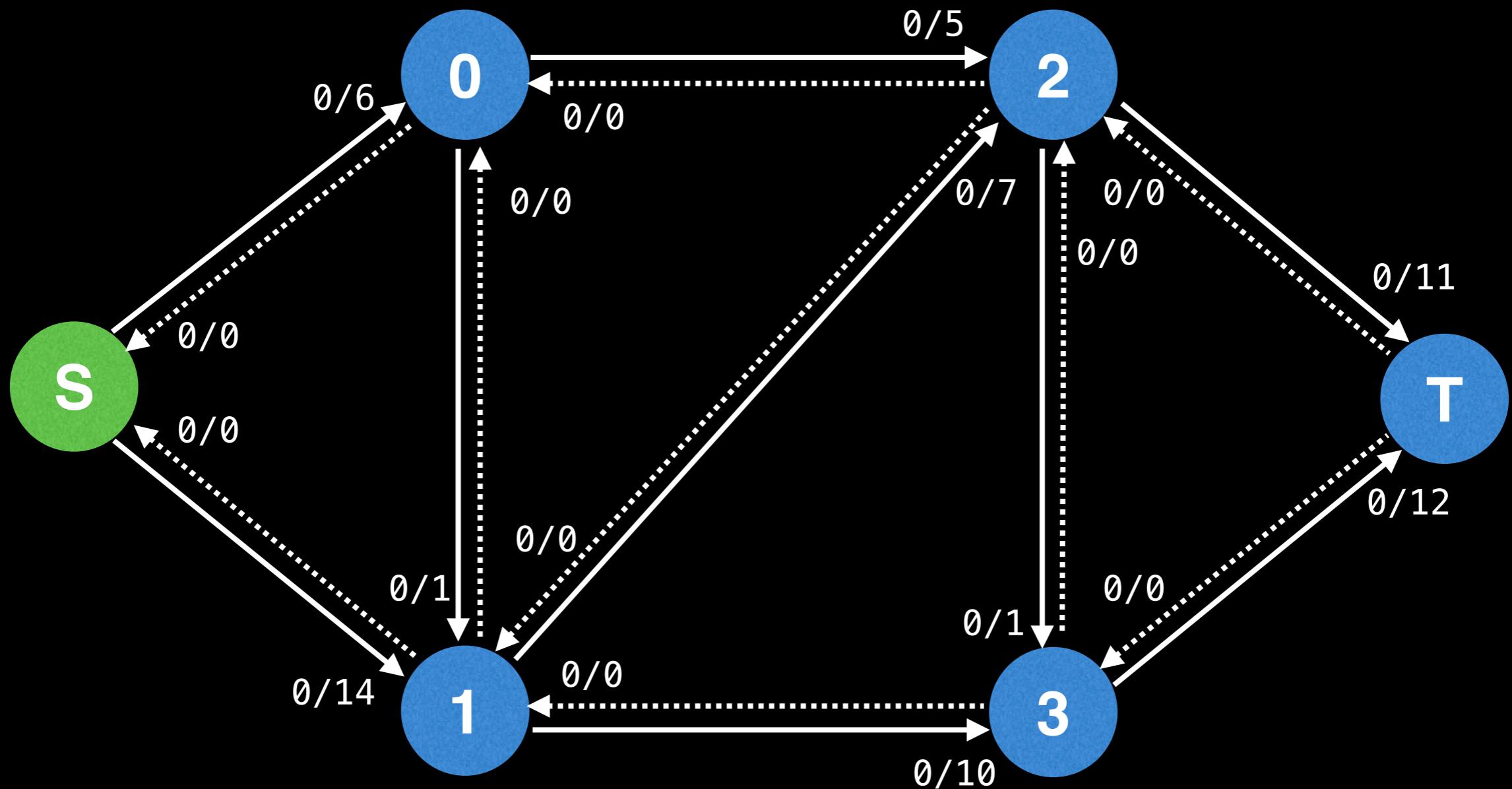
# Network Flow: Capacity Scaling

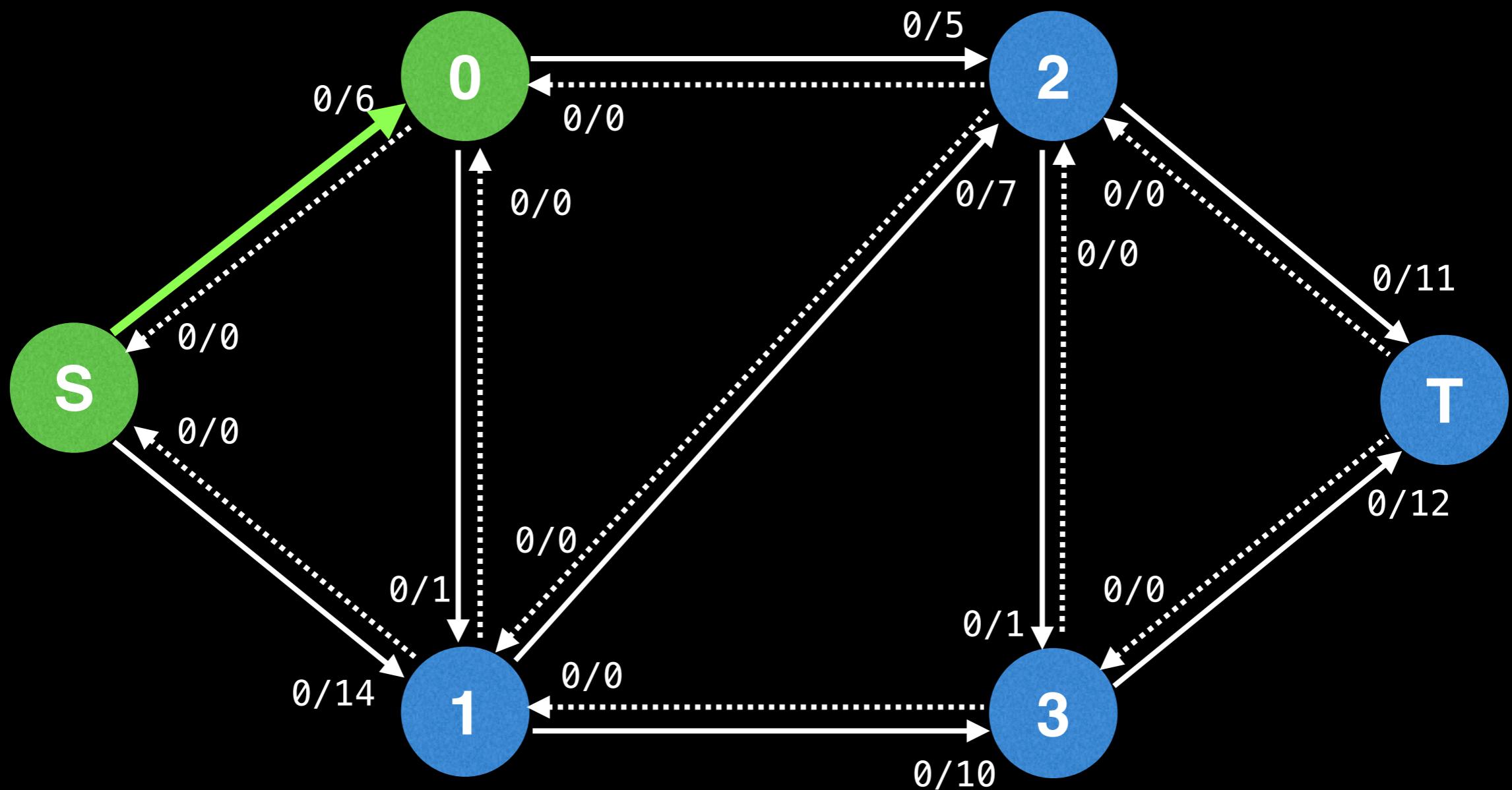
William Fiset

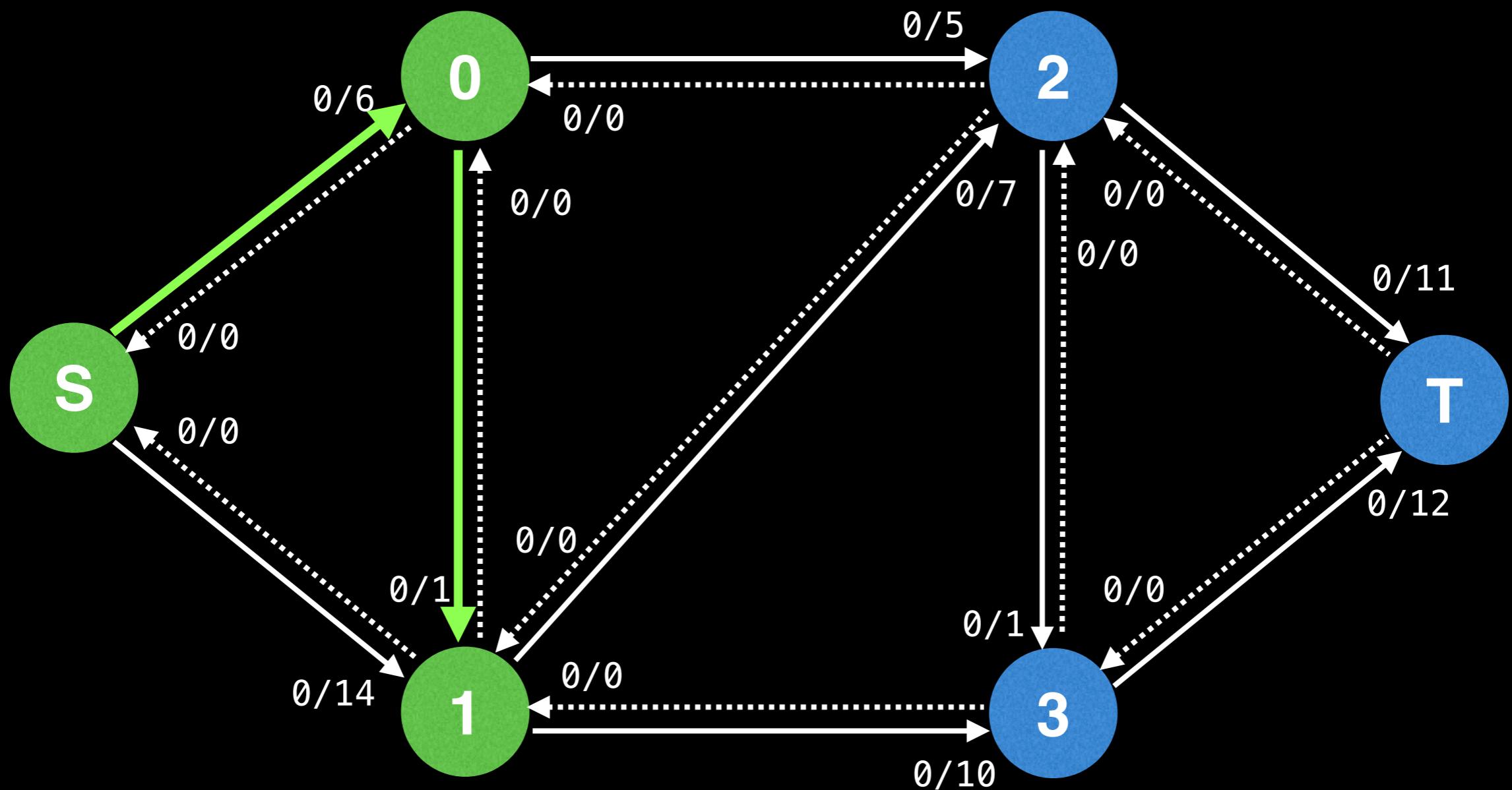


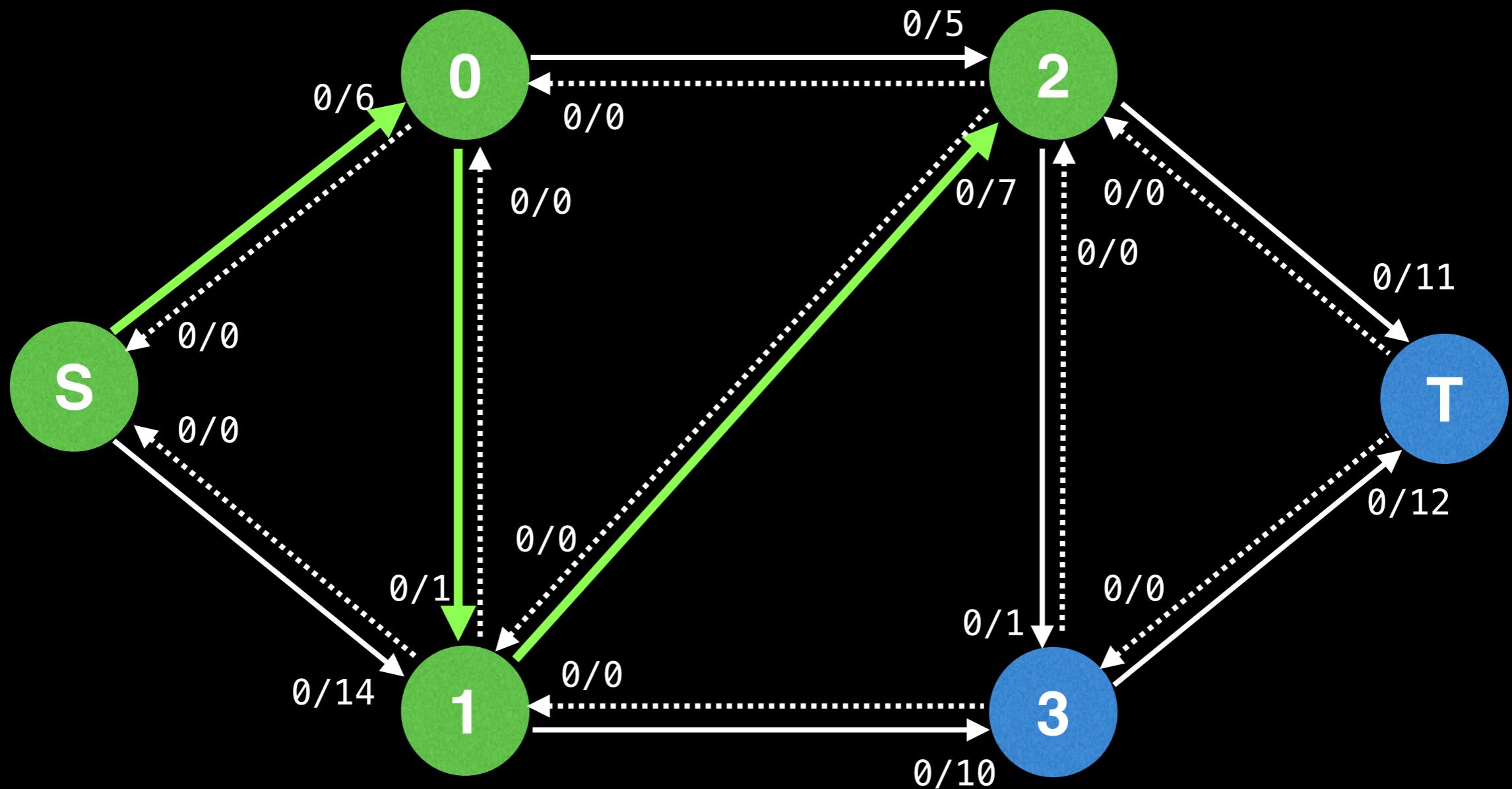
Let's quickly revisit finding the maximum flow using a DFS.

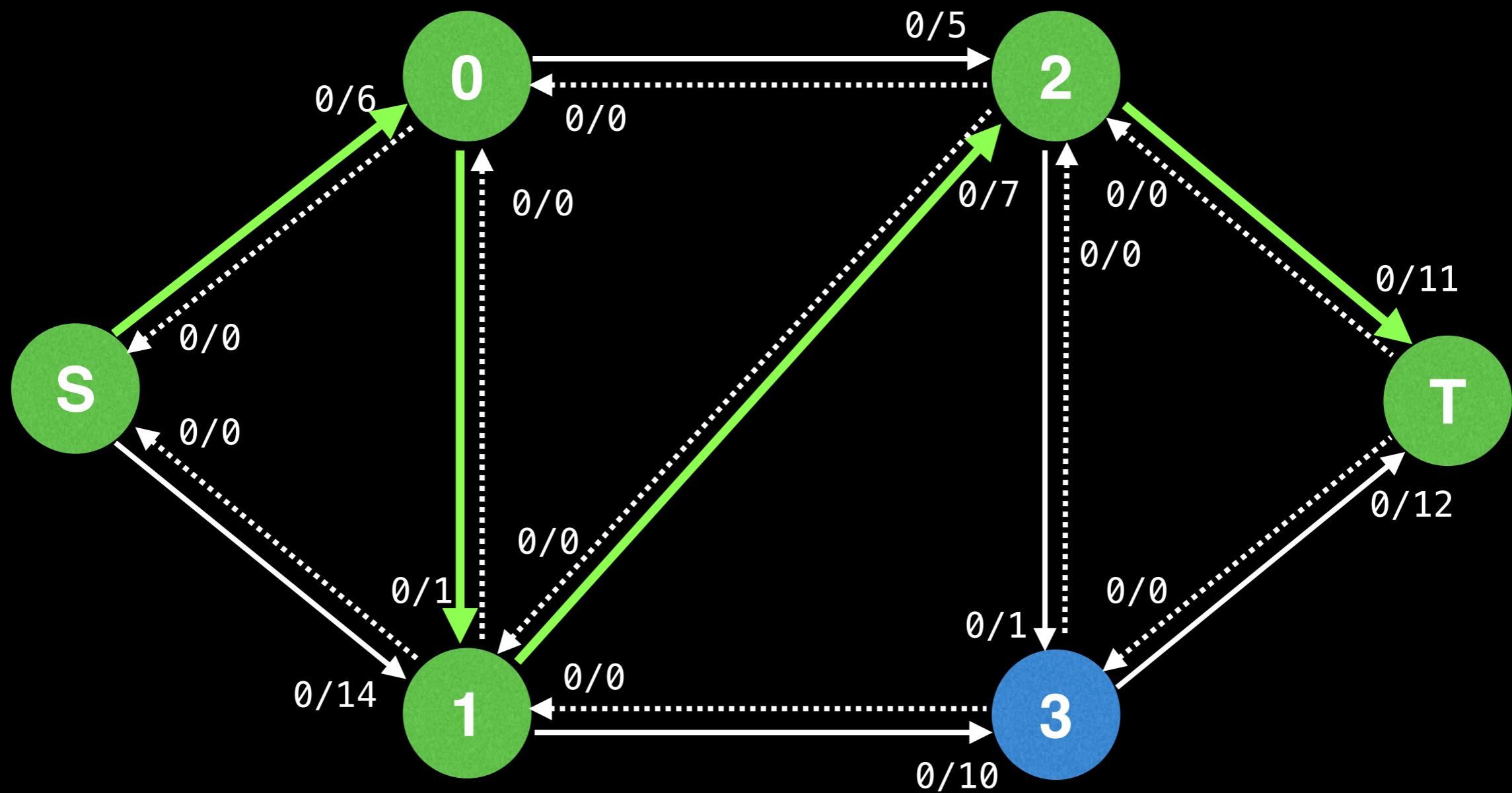






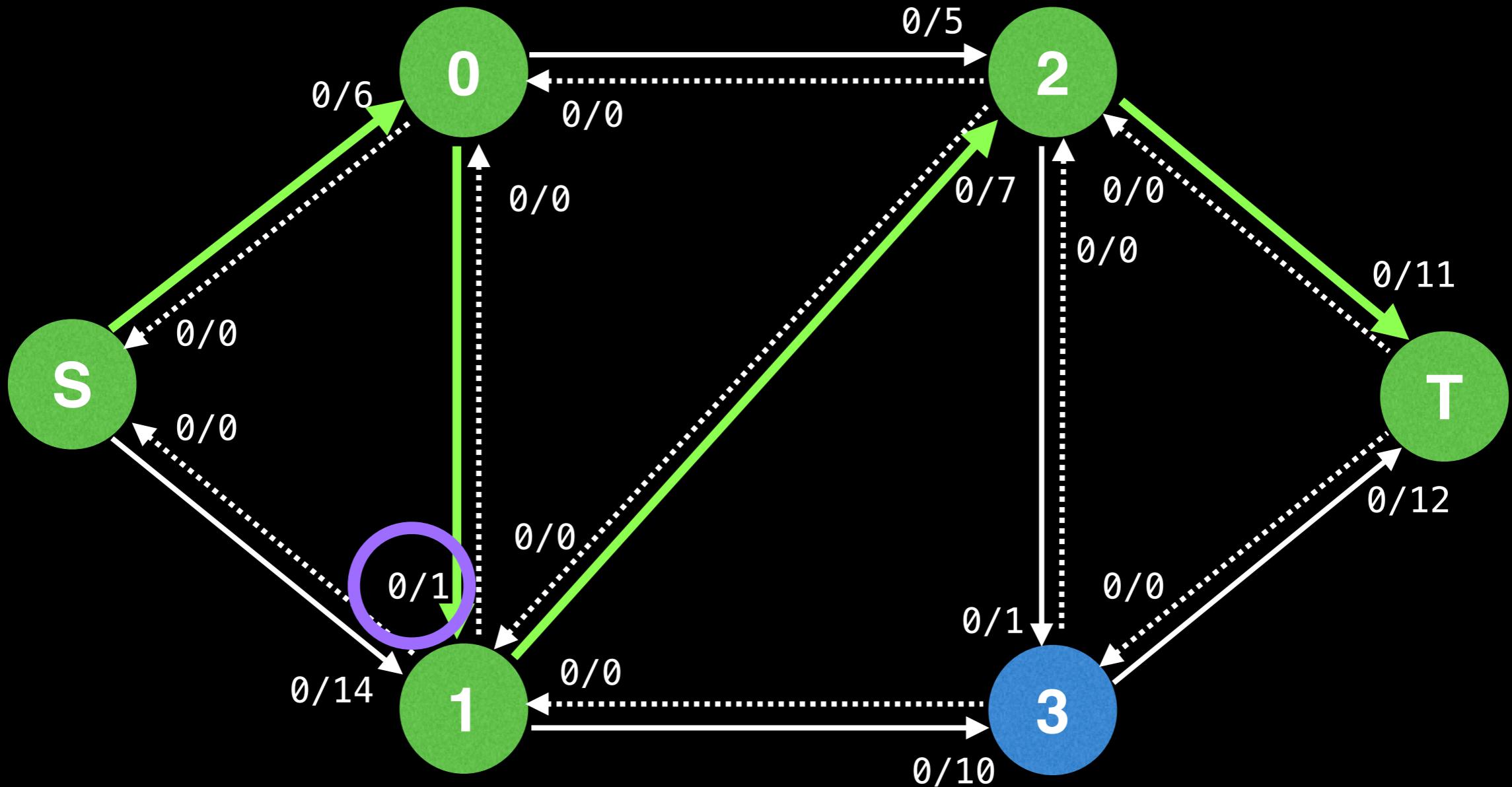






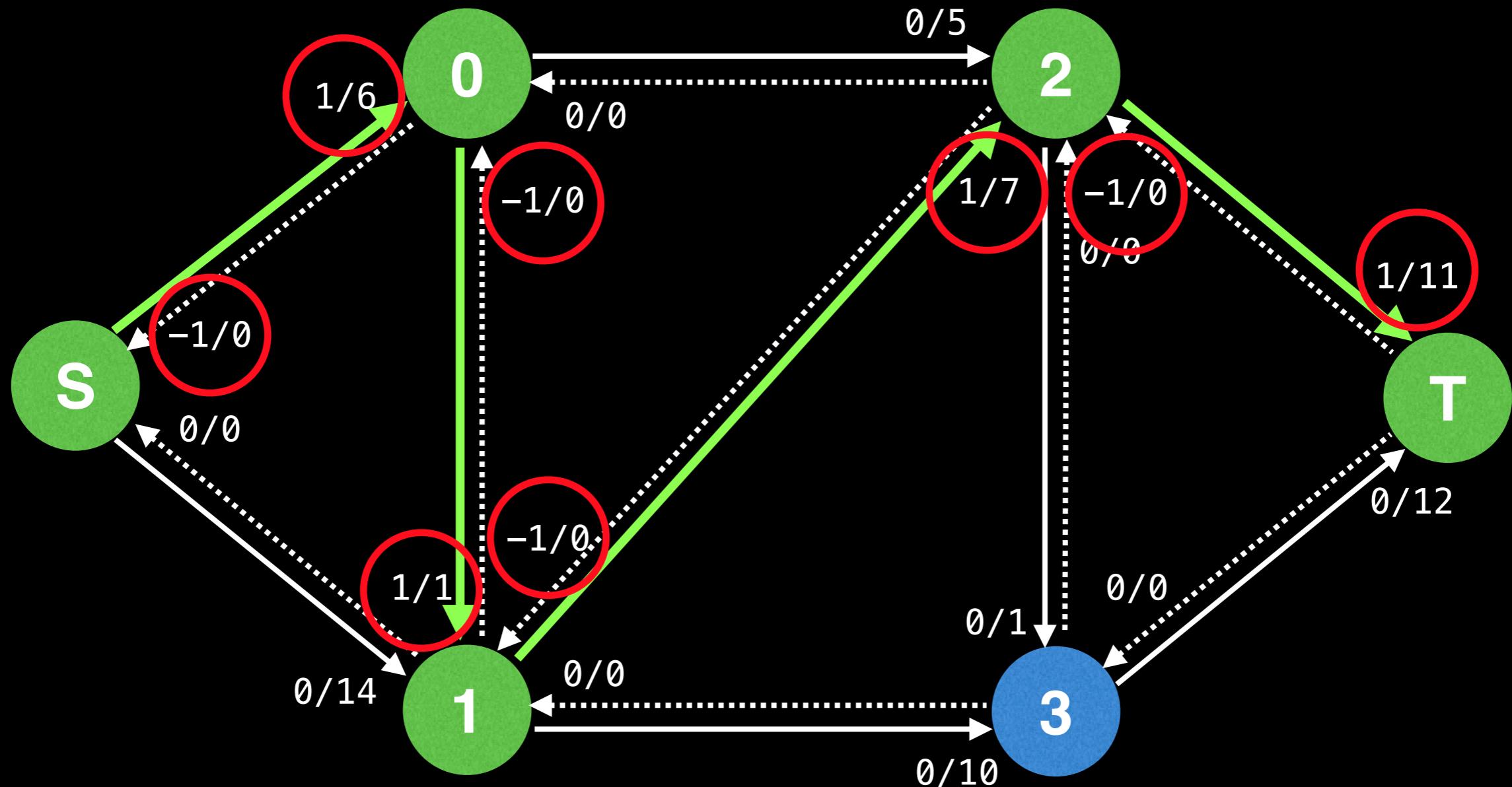
Find the bottleneck (edge with smallest remaining capacity) along the augmenting path.

$$\min(6-0, 1-0, 7-0, 11-0) = 1$$

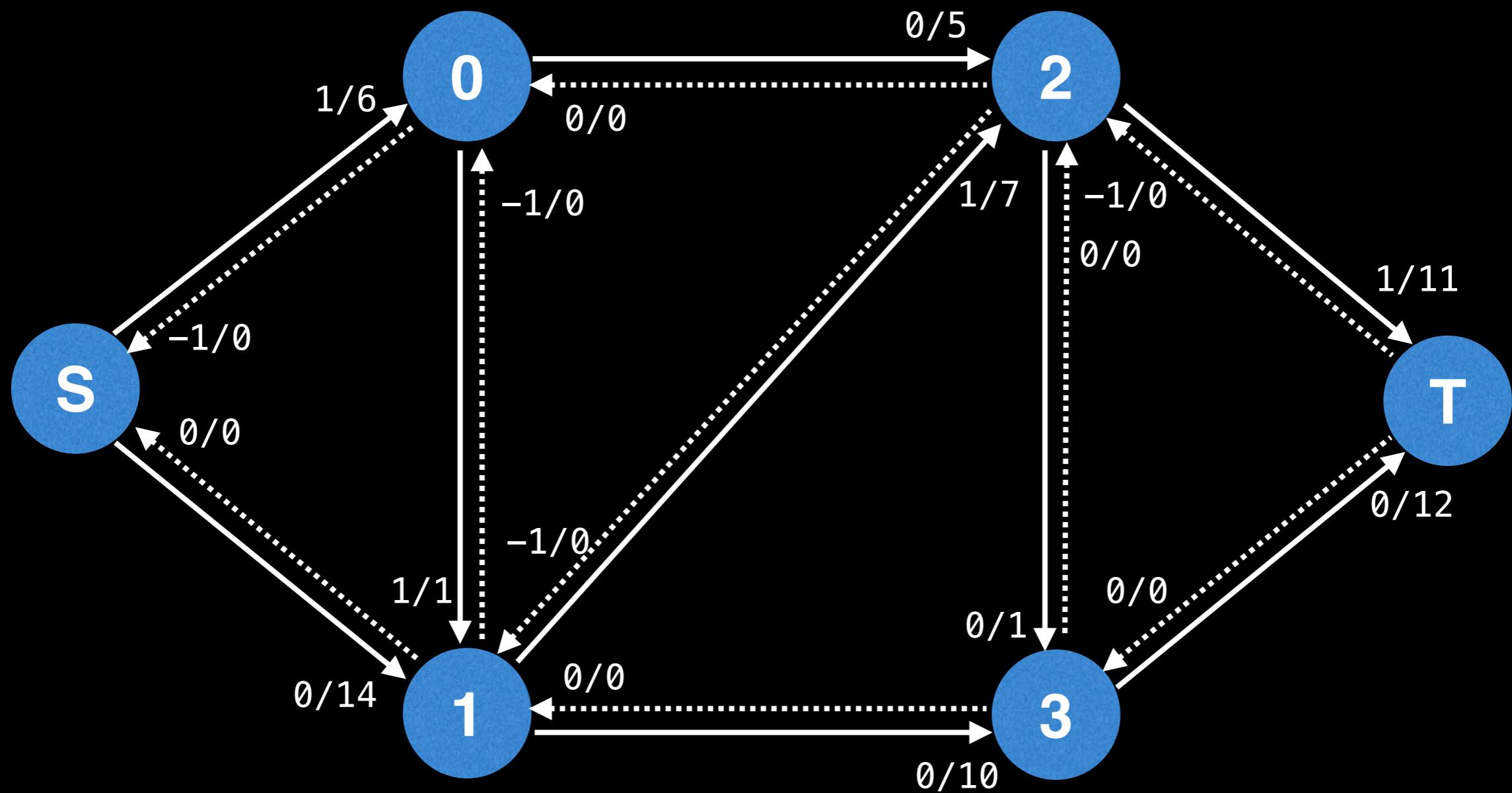


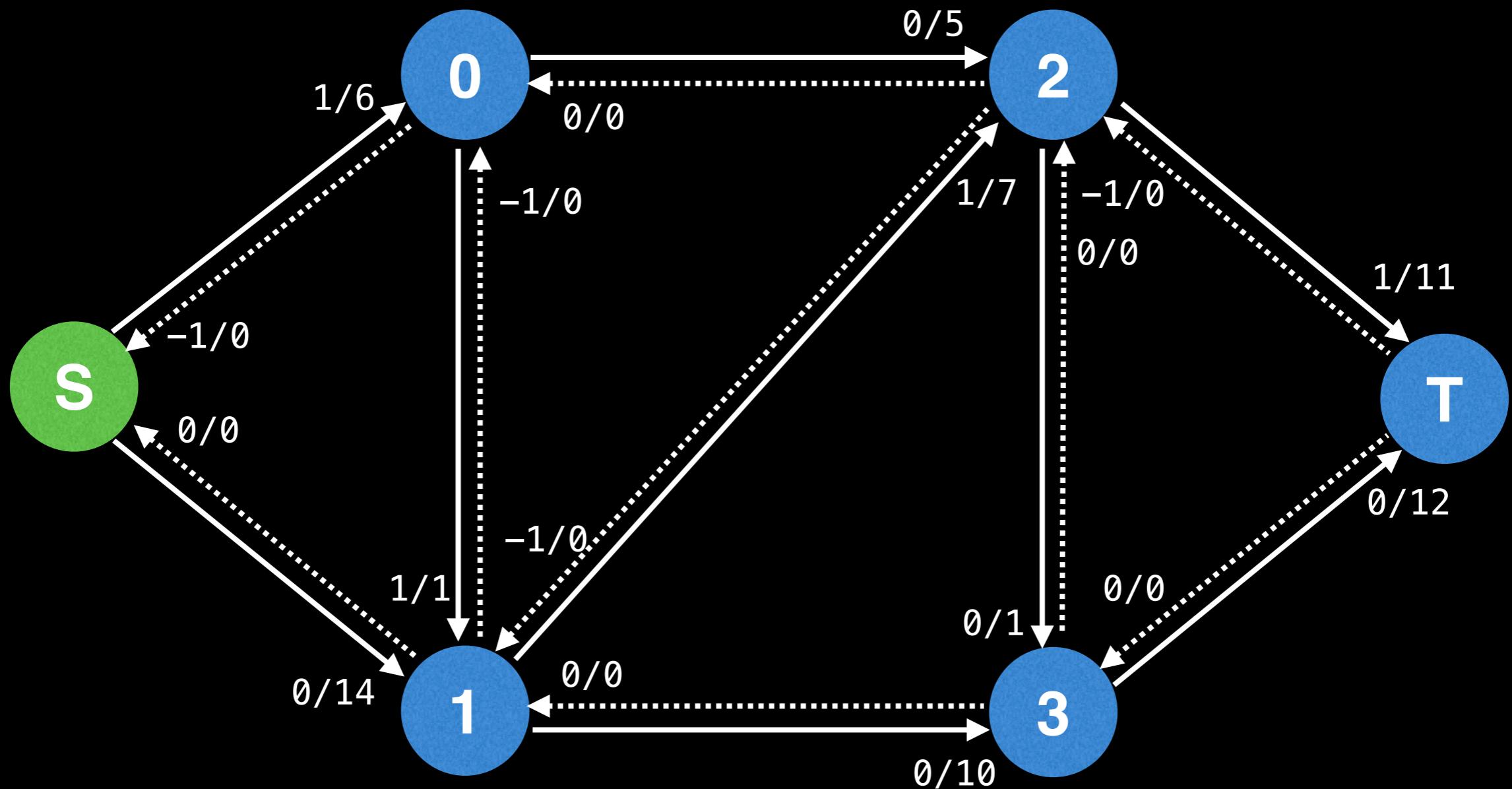
bottleneck is 1 unit of flow!

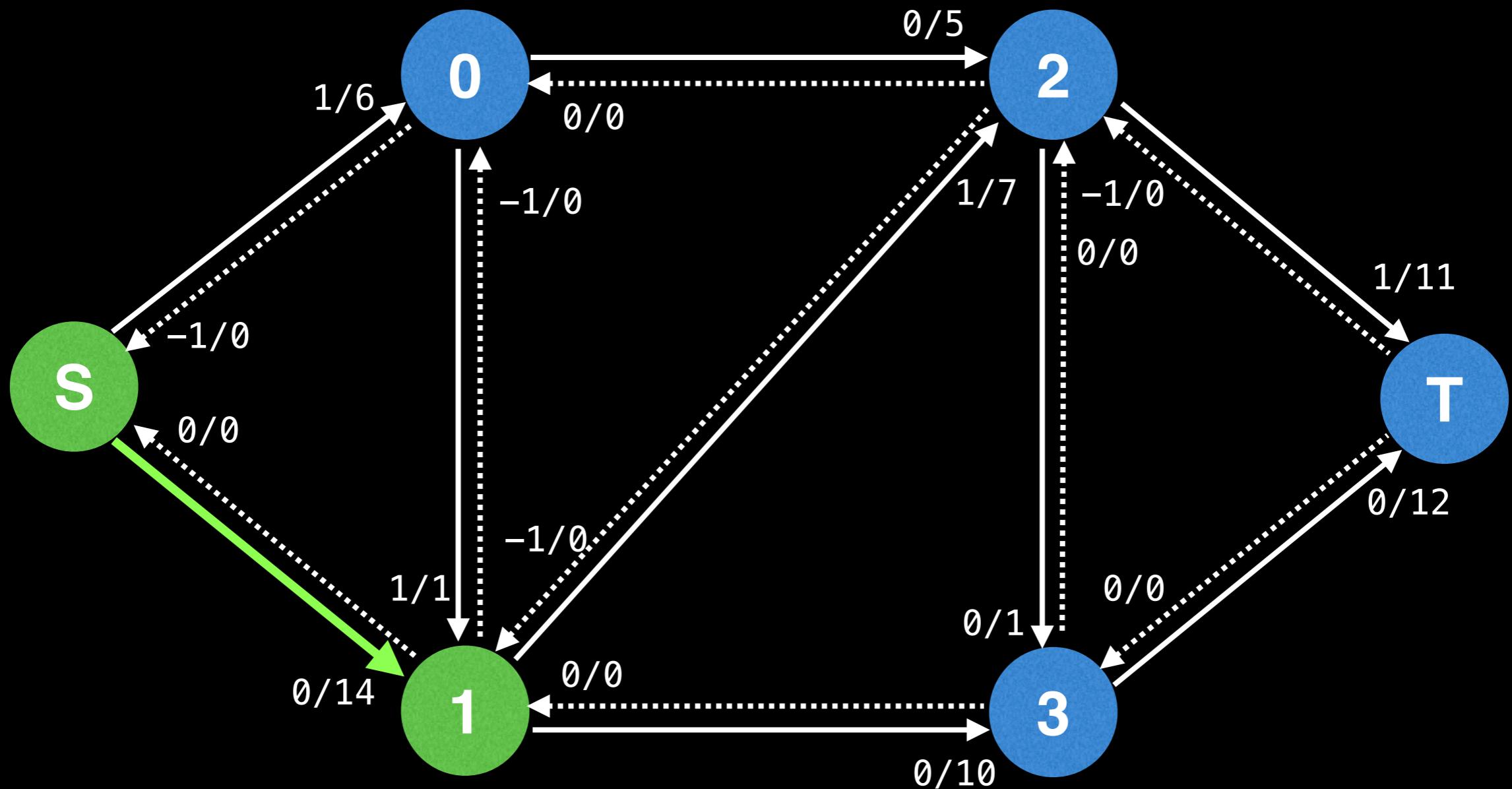
Augment (update) the flow by adding the bottleneck value to the flow along the forward edges and subtracting flow by the bottleneck value along the residual edges.

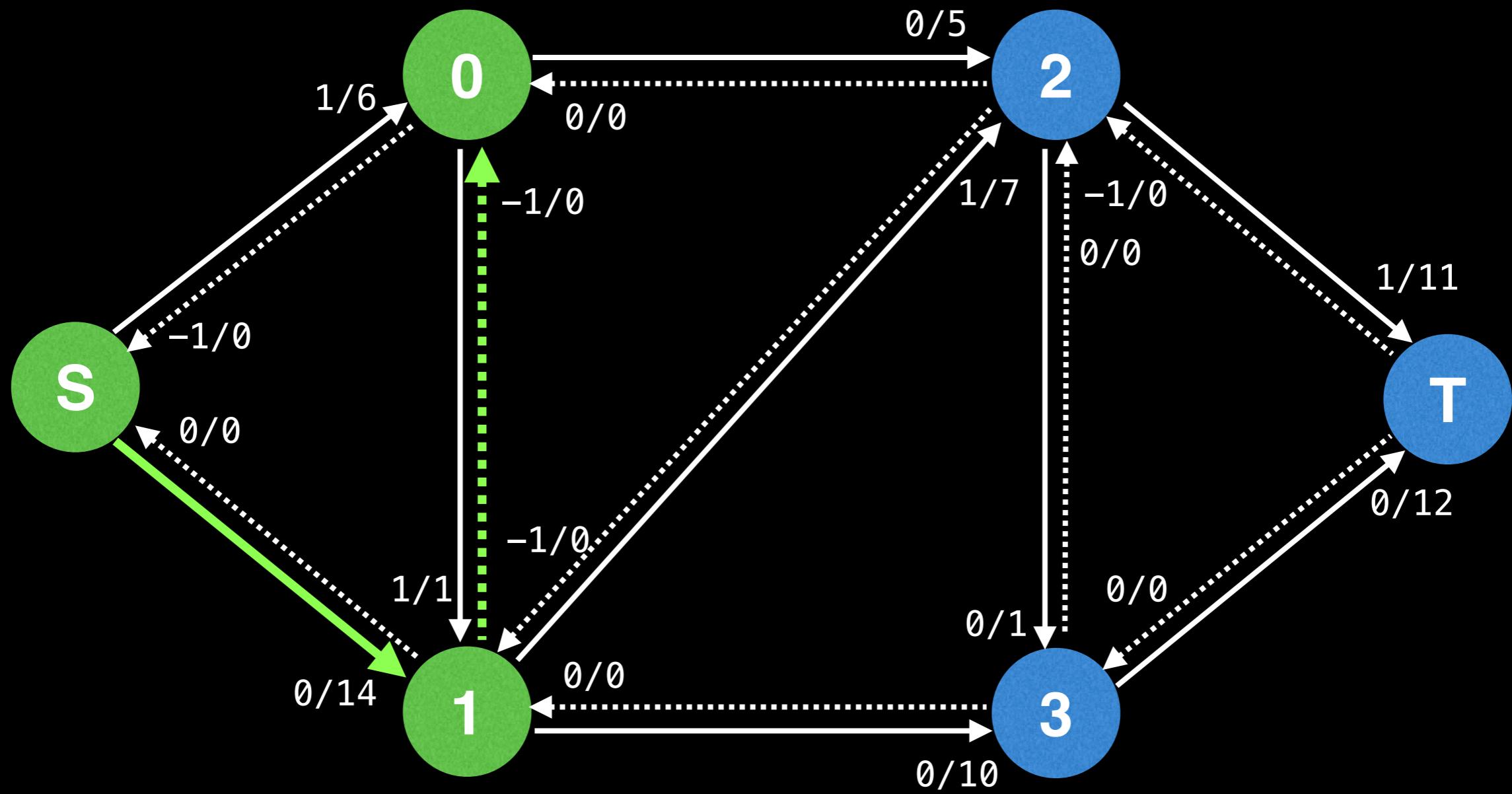


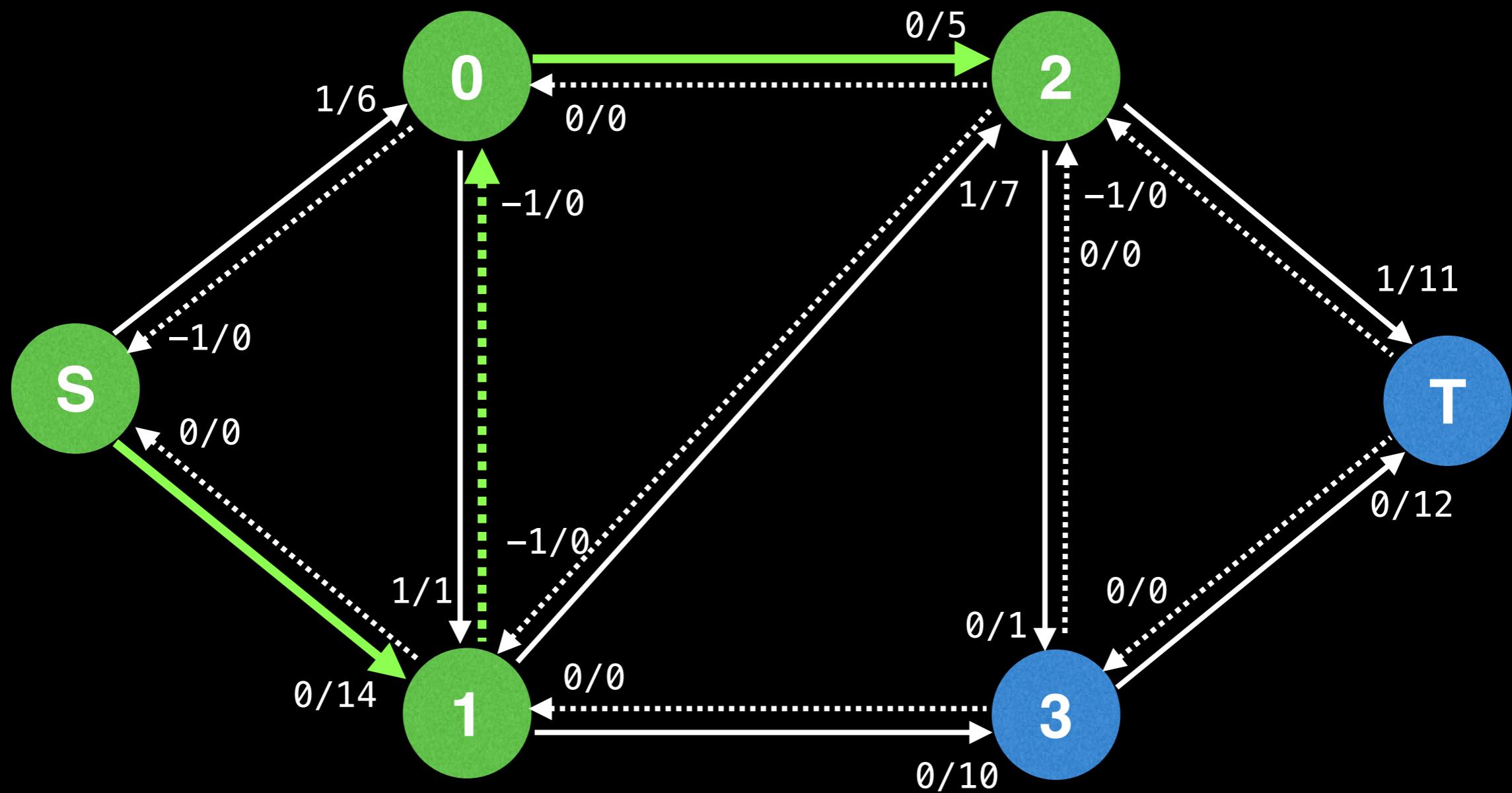
Unfortunately, we are only able to augment the flow along the path by one unit :/

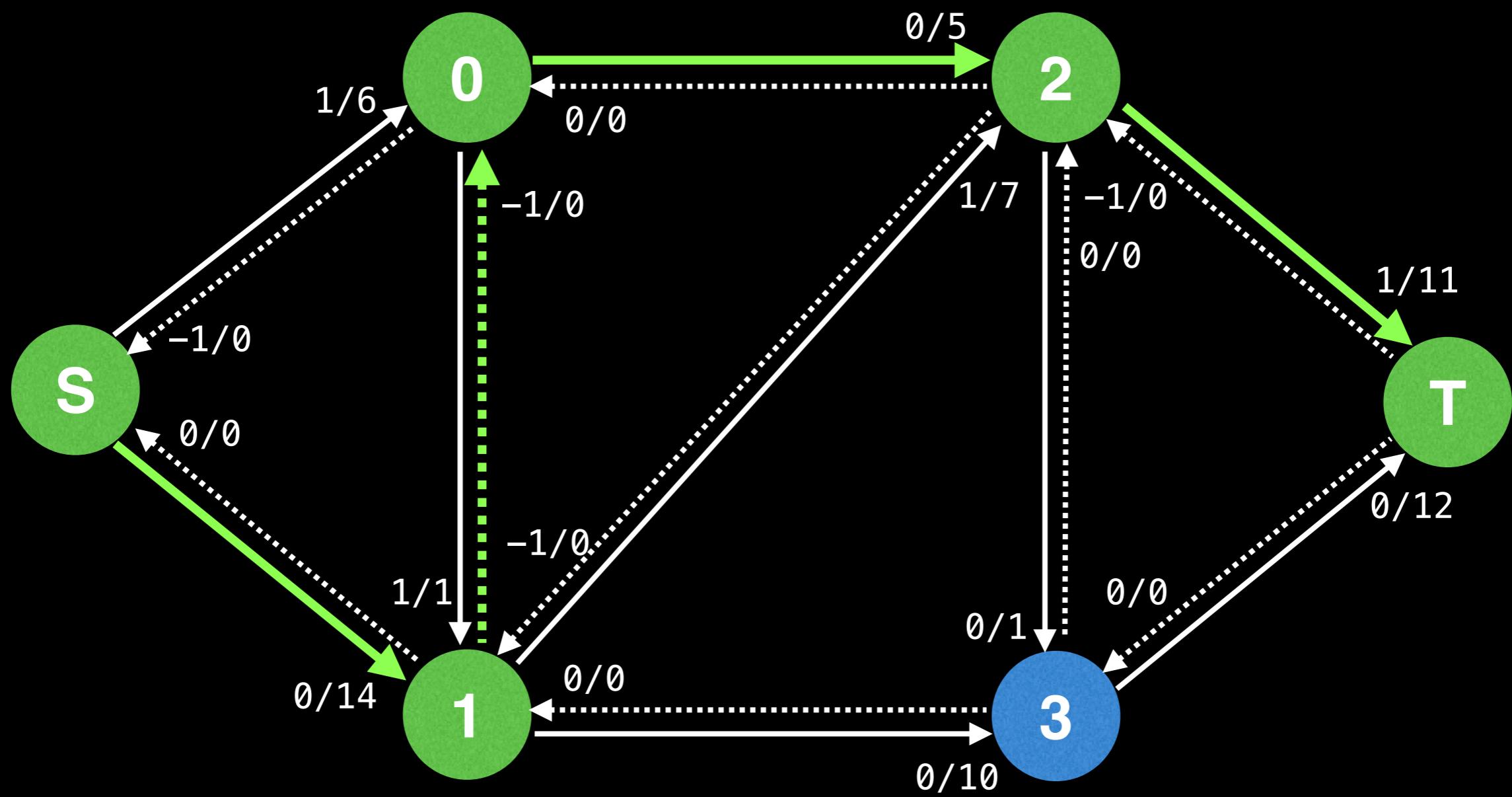




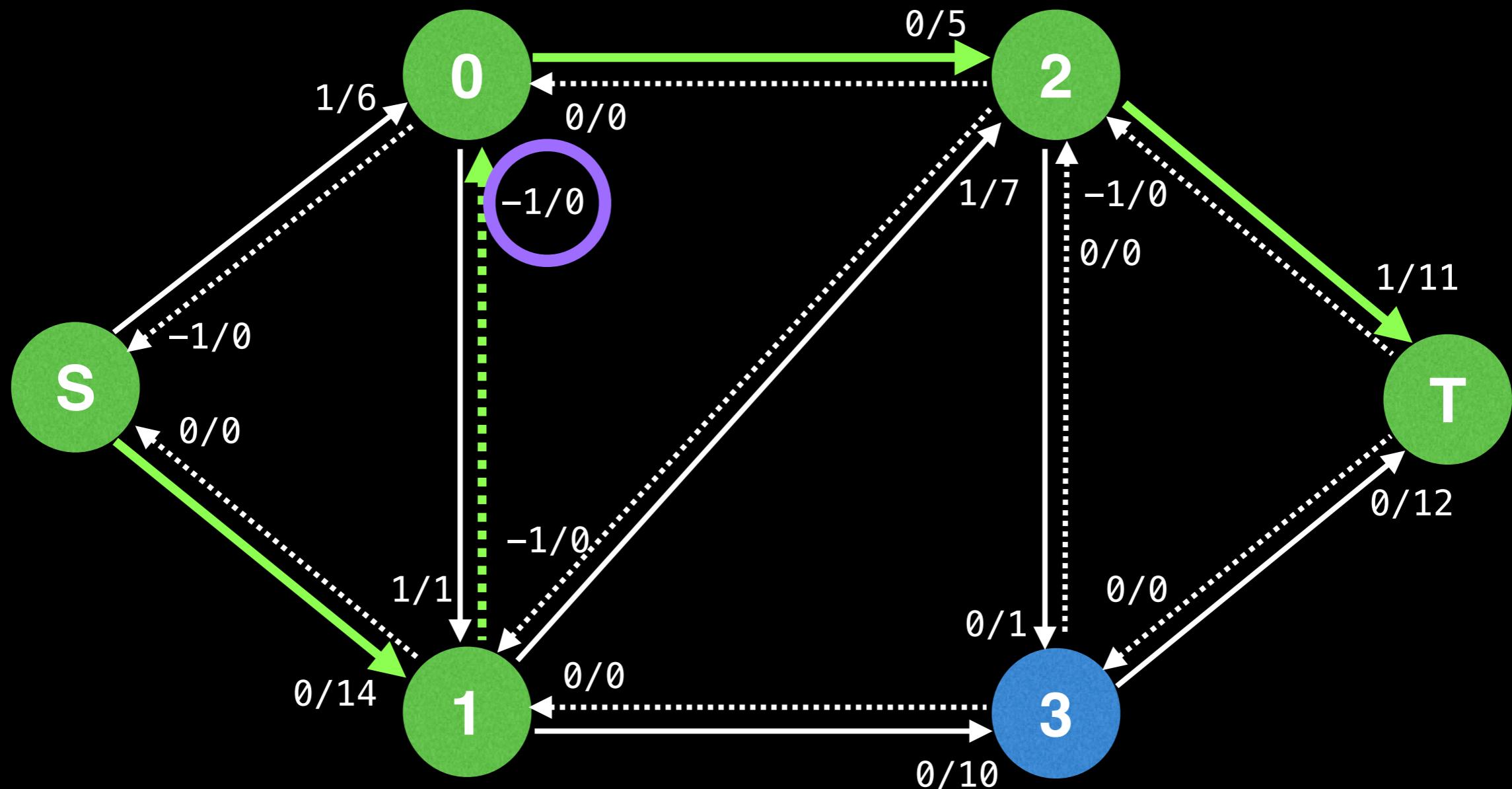






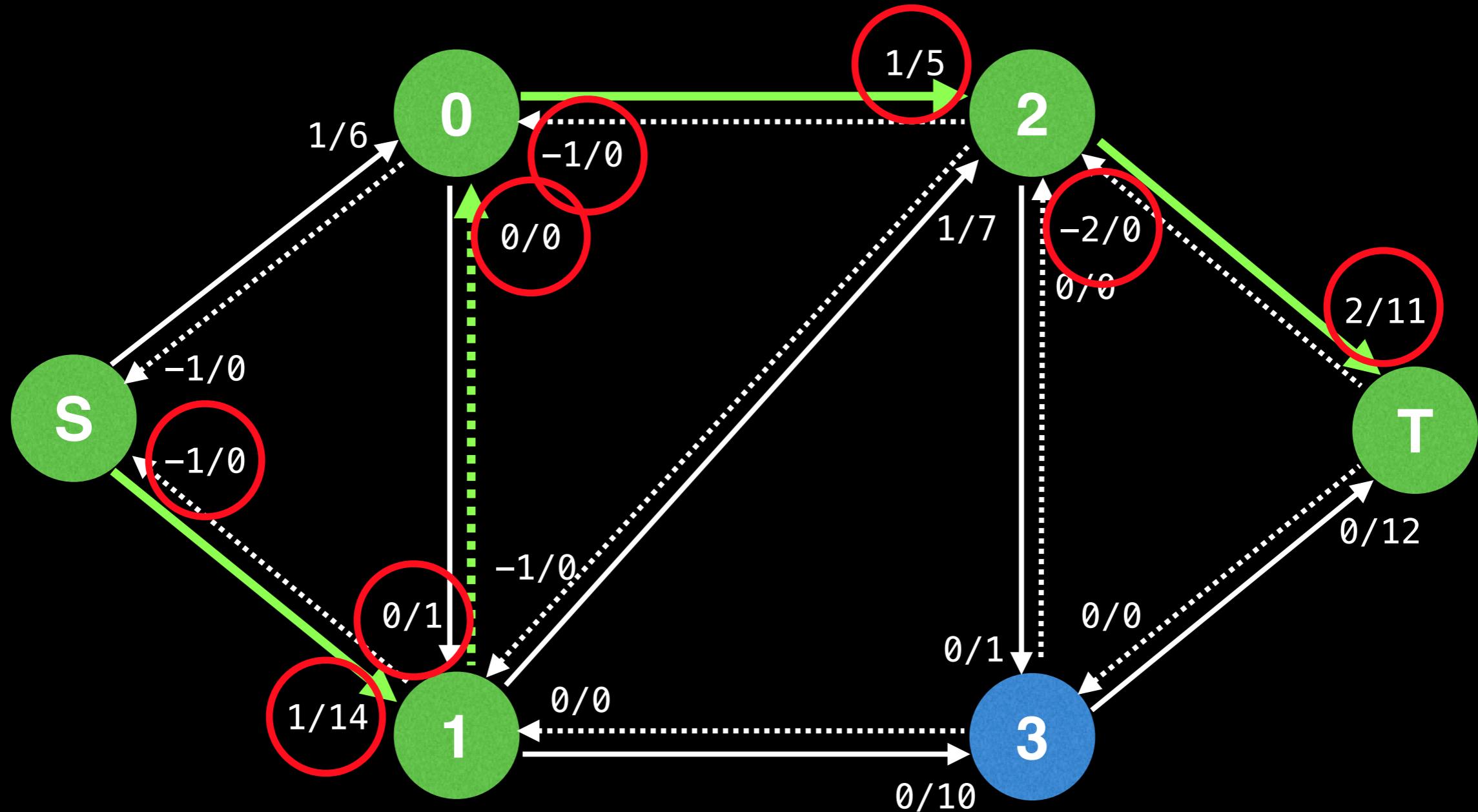


Recall that the **remaining capacity** of an edge is calculated as: capacity - flow. This allows residual edges with negative flow to have a positive remaining capacity.  
 $\min(14-0, 0-(-1), 5-0, 11-1) = 1$

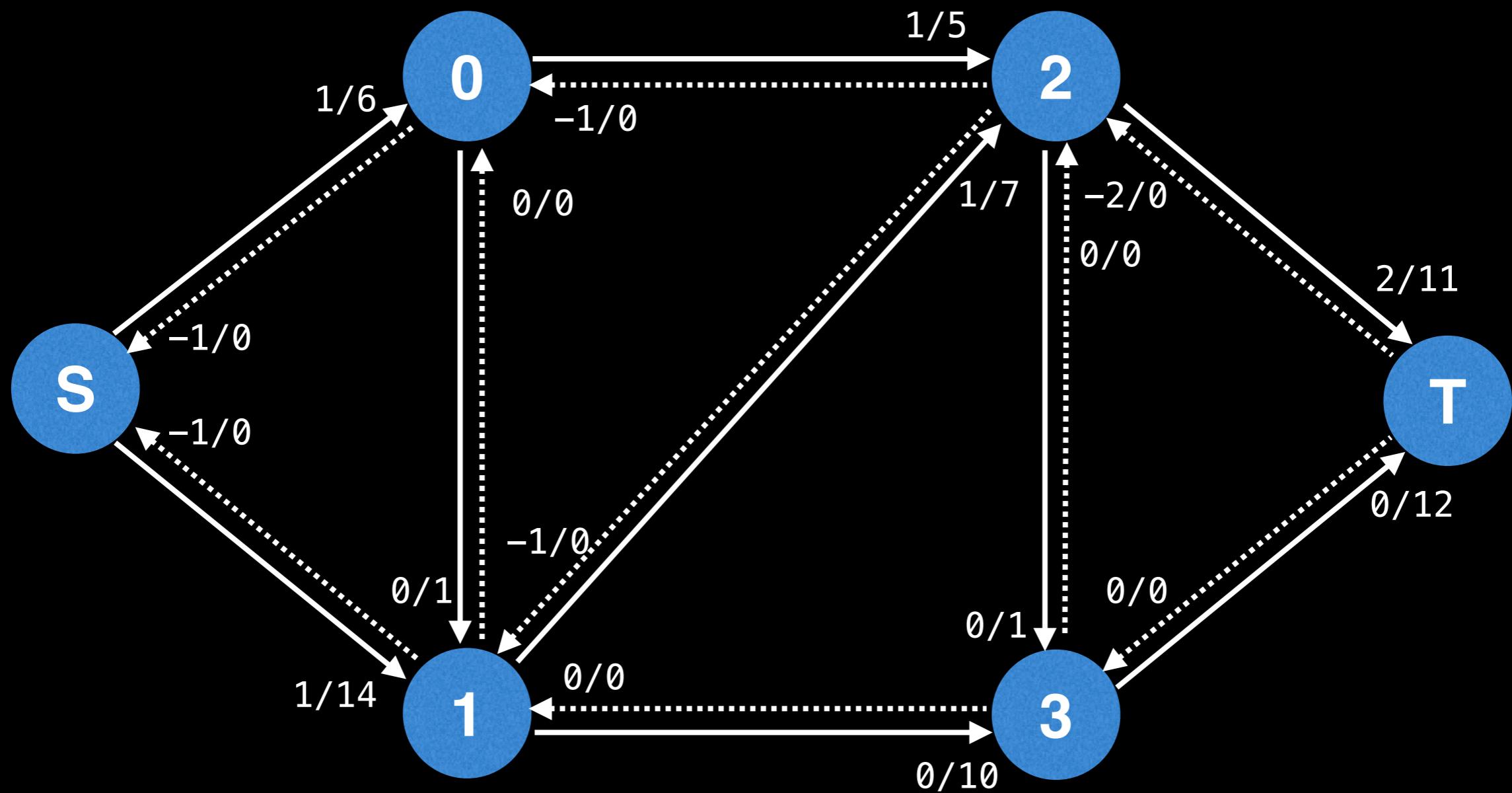


The bottleneck is still one unit of flow going through the residual edge!

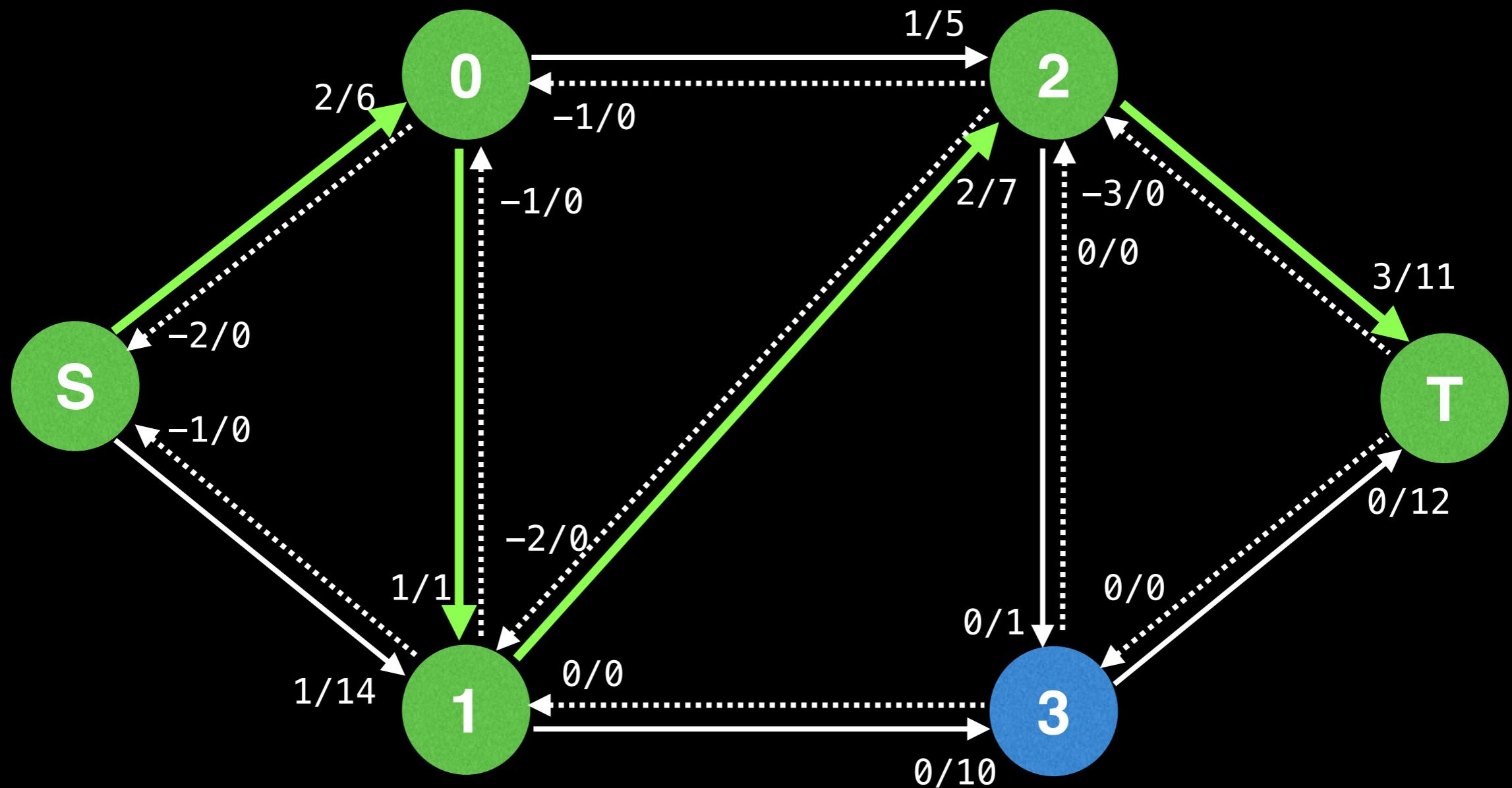
Augment (update) the flow by adding the bottleneck value to the flow along the forward edges and subtracting flow by the bottleneck value along the residual edges.



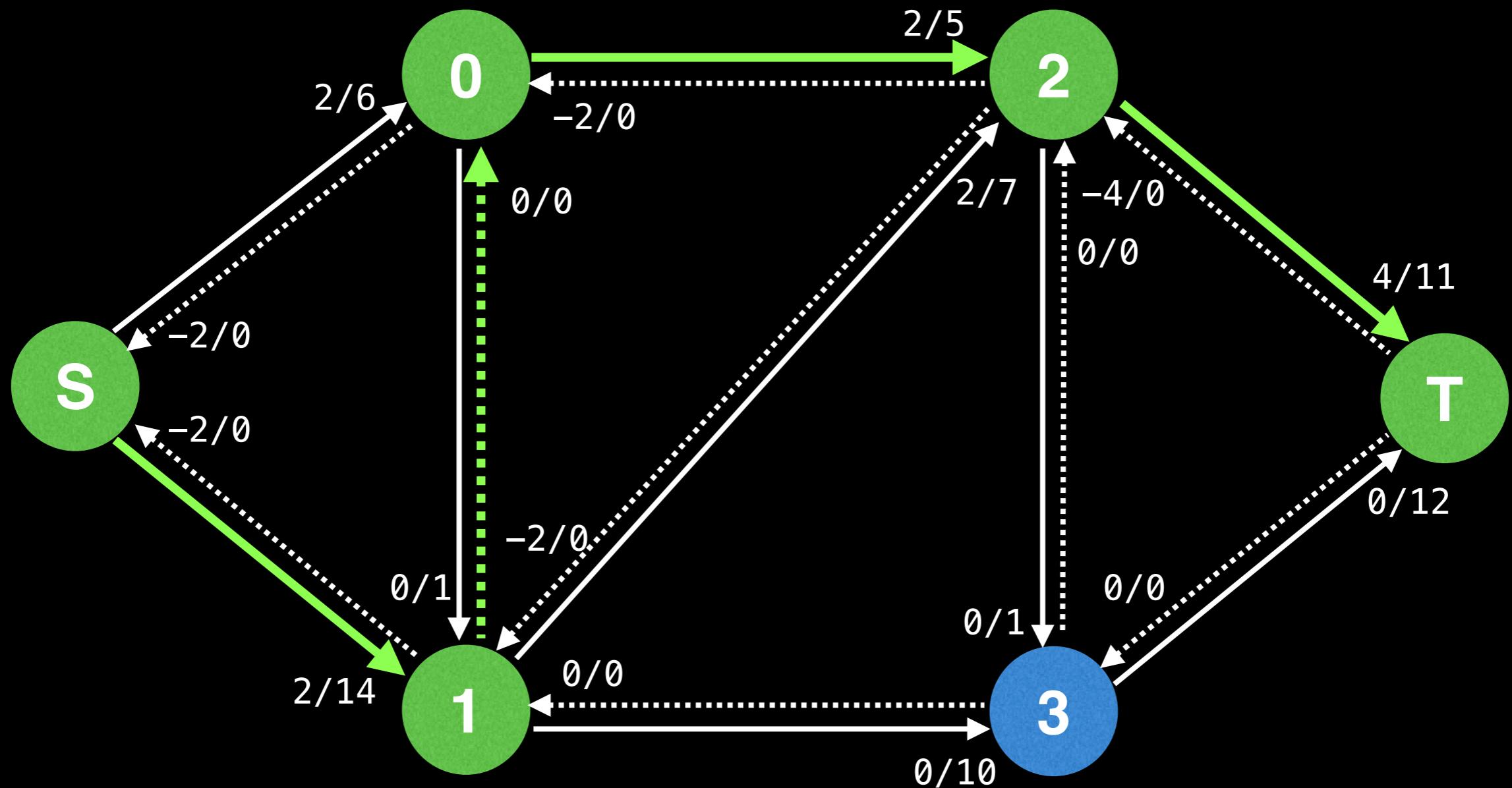
Unfortunately, we are only able to augment the flow along the path by one unit :/



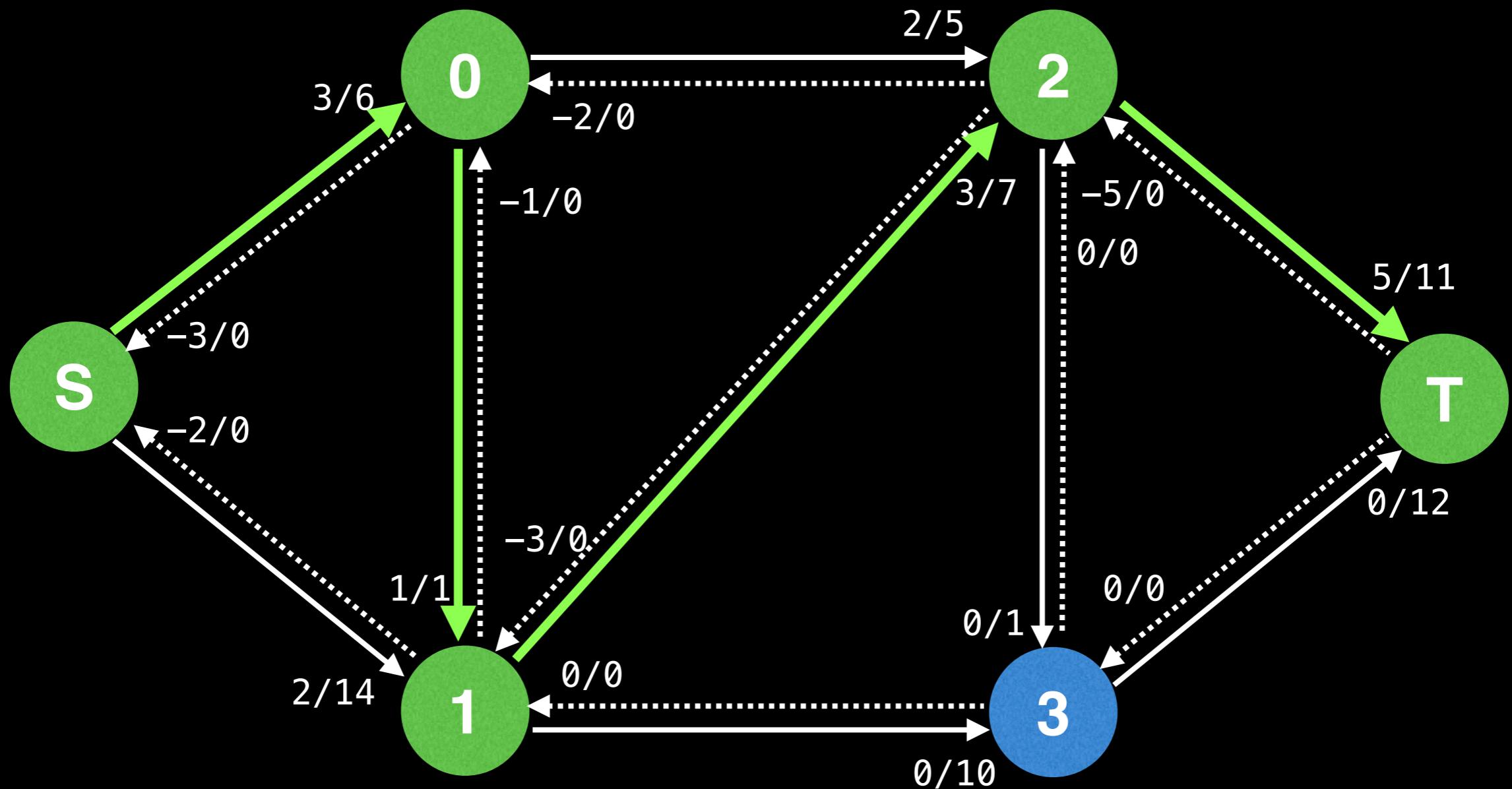
You could imagine a DFS algorithm repeatedly taking an edge with a capacity of 1, ultimately limiting how much flow we can push through the network with each iteration.



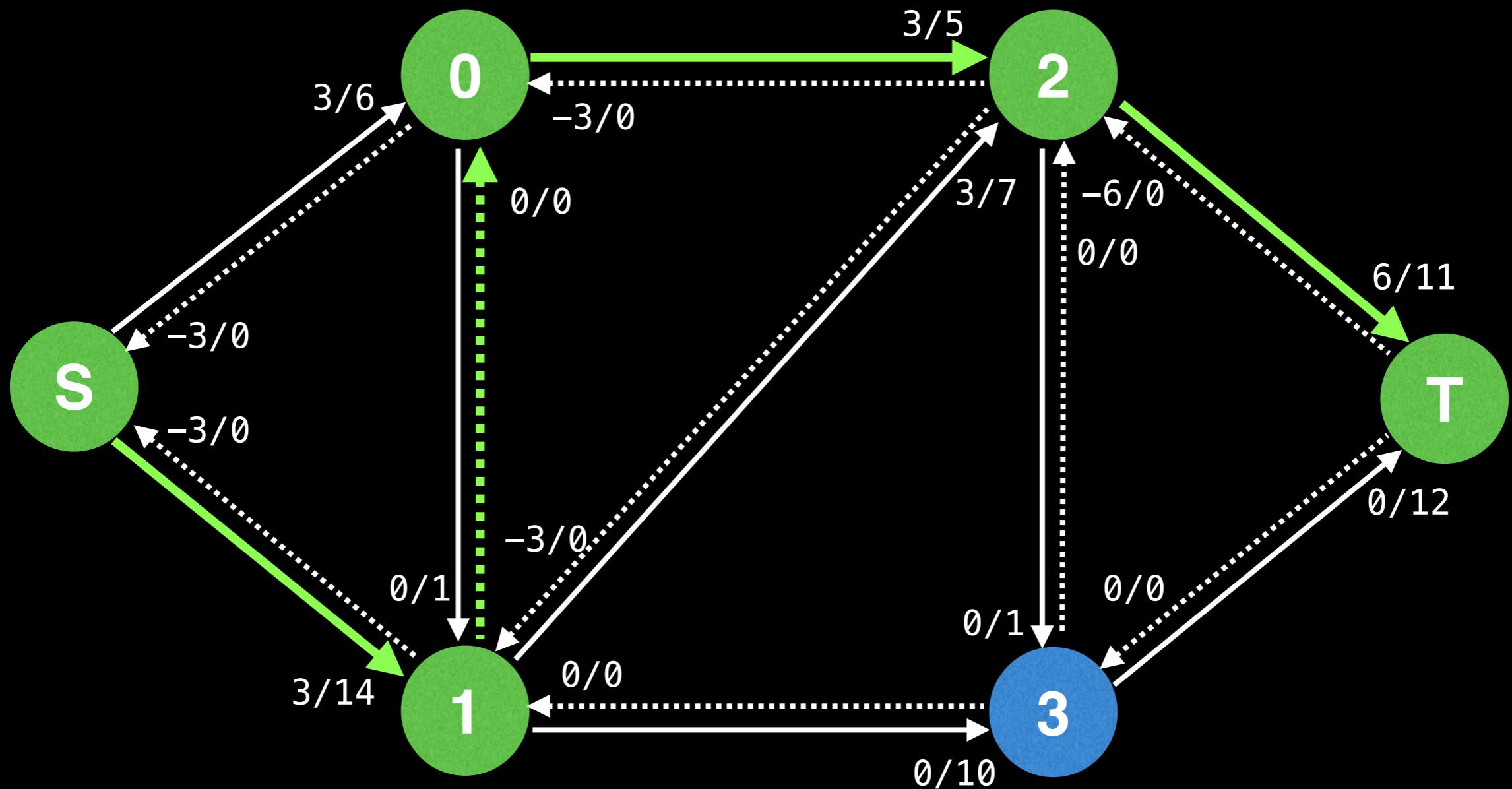
You could imagine a DFS algorithm repeatedly taking an edge with a capacity of 1, ultimately limiting how much flow we can push through the network with each iteration.



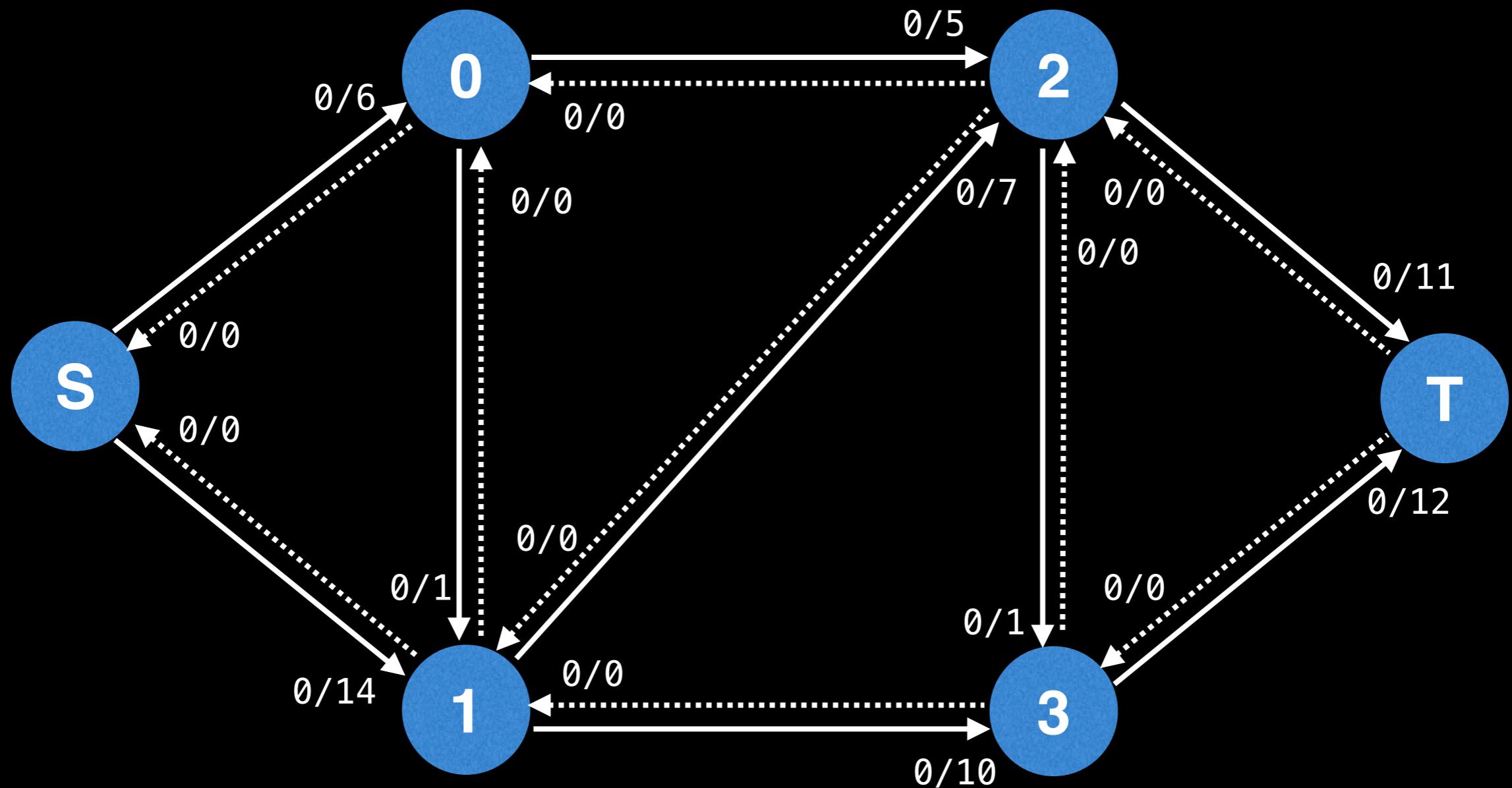
You could imagine a DFS algorithm repeatedly taking an edge with a capacity of 1, ultimately limiting how much flow we can push through the network with each iteration.



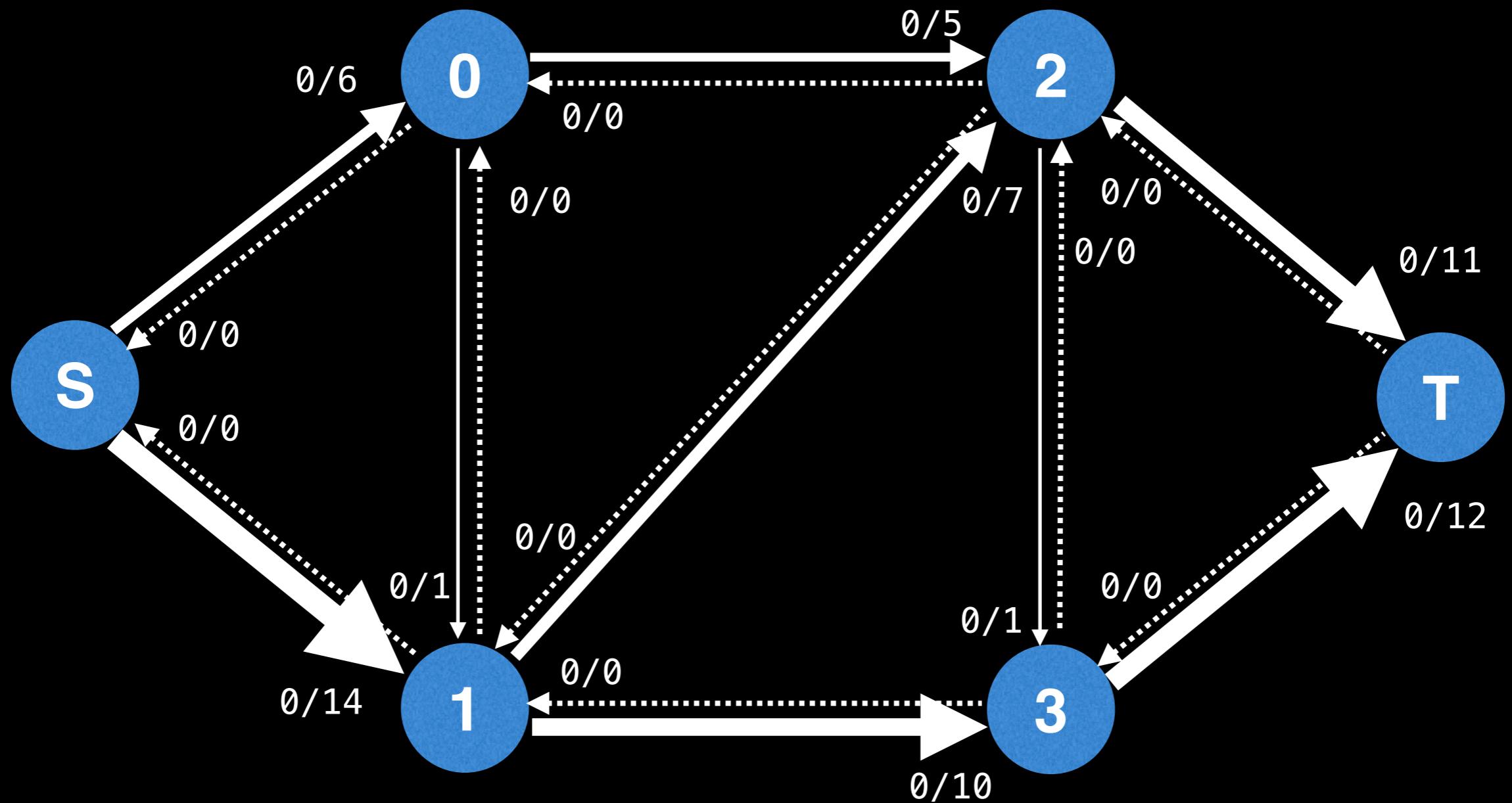
You could imagine a DFS algorithm repeatedly taking an edge with a capacity of 1, ultimately limiting how much flow we can push through the network with each iteration.



Capacity scaling is the idea that we should prioritize taking edges with larger capacities to avoid ending up with a path that has a small bottleneck.



If we adjust the size of each edge based on its capacity value, then we can more easily visualize which edges we should give more attention to.



# Capacity Scaling Algorithm

Let  $U$  = the value of the largest edge capacity in the initial flow graph.

Let  $\Delta$  = the largest power of 2 less than or equal to  $U$ .

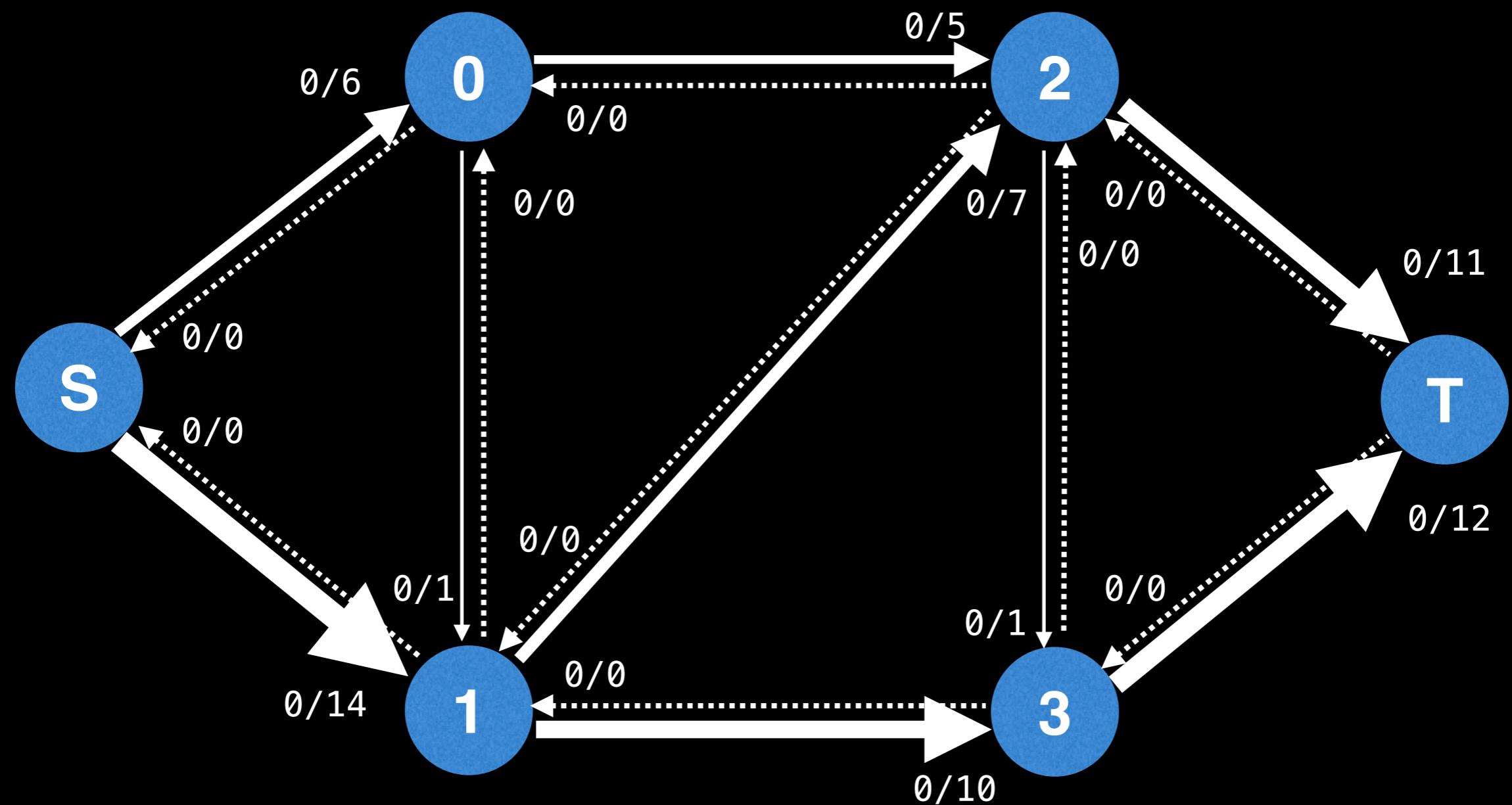
The **capacity scaling heuristic** says that you should only take edges whose remaining capacity is  $\geq \Delta$  in order to achieve a better runtime.

# Capacity Scaling Algorithm

The algorithm repeatedly finds augmenting paths with remaining capacity  $\geq \Delta$  until no more paths satisfy this criteria, then decrease the value of delta by dividing it by 2 ( $\Delta = \Delta / 2$ ) and repeat while  $\Delta > 0$ .

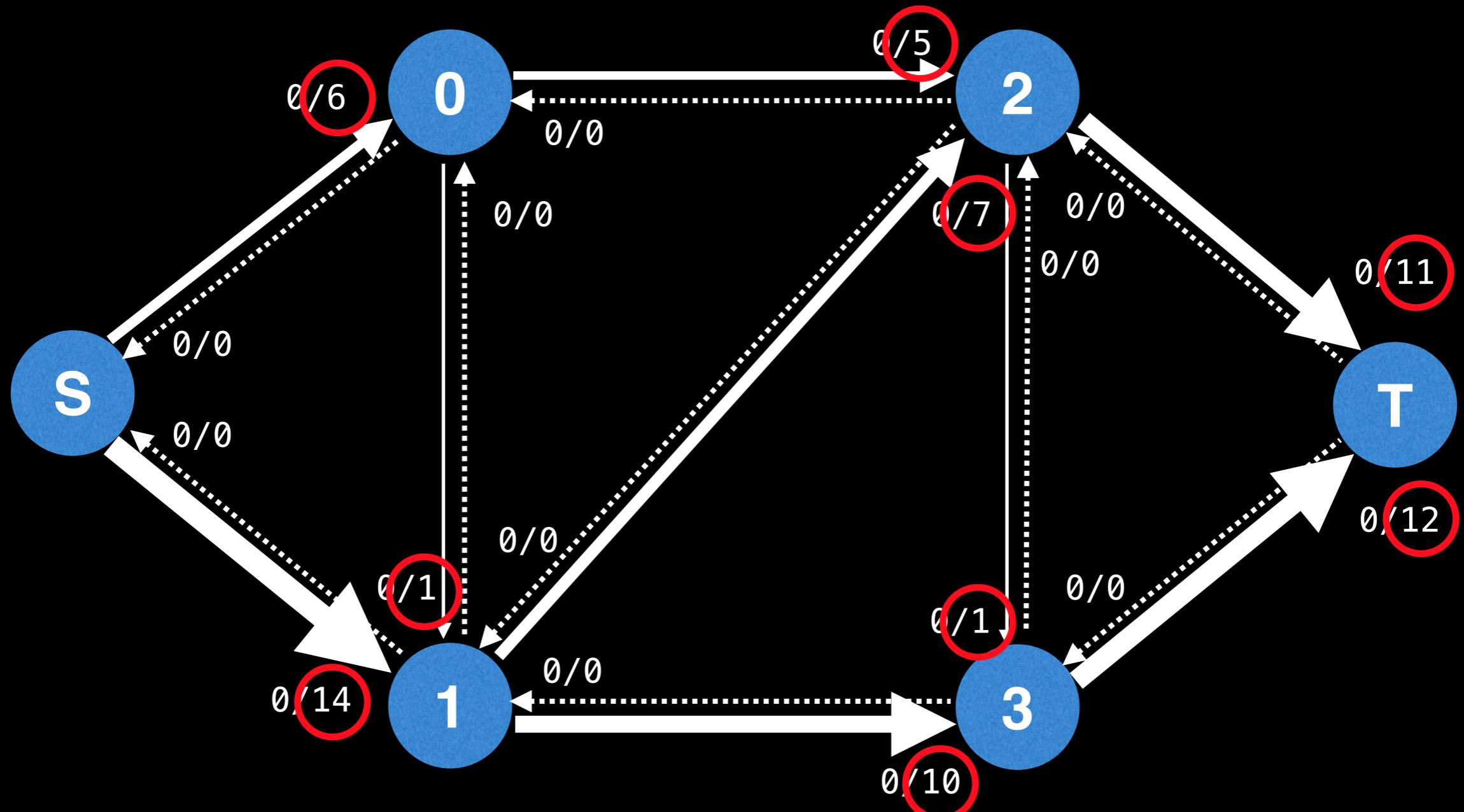
The capacity scaling **works very well in practice**. In terms of time complexity, capacity scaling with a DFS is bounded by  **$O(E^2 \log(U))$**  or  **$O(EV \log(U))$**  if the shortest augmenting path (like Edmonds-Karp) is used (although I've found that the latter seems much slower in practice).

Let's use capacity scaling to find the max flow of the following flow graph!



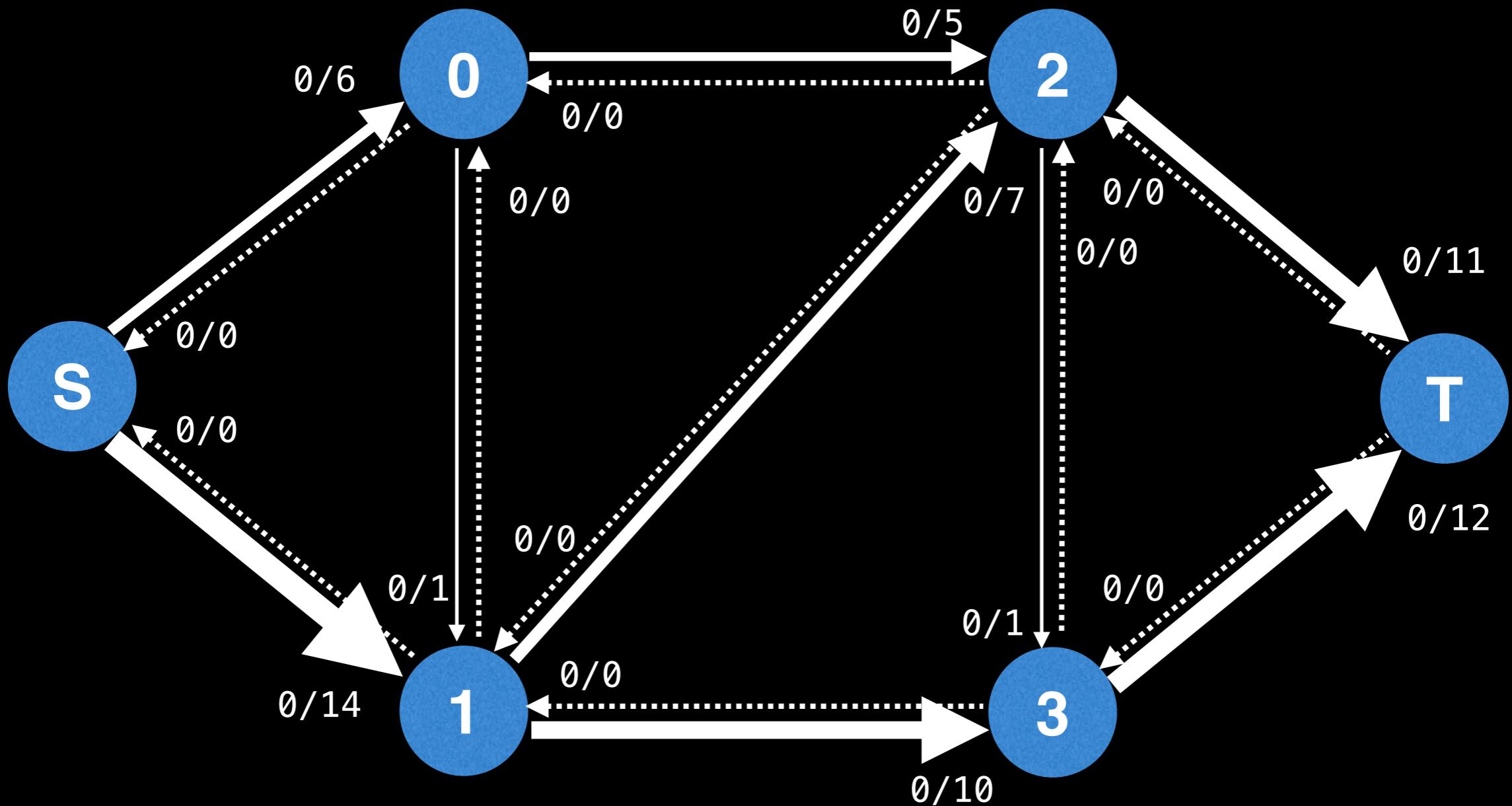
First, compute  $U$  as the maximum of all initial capacities.

$$U = \max(6, 14, 1, 5, 7, 10, 1, 11, 12) = 14$$

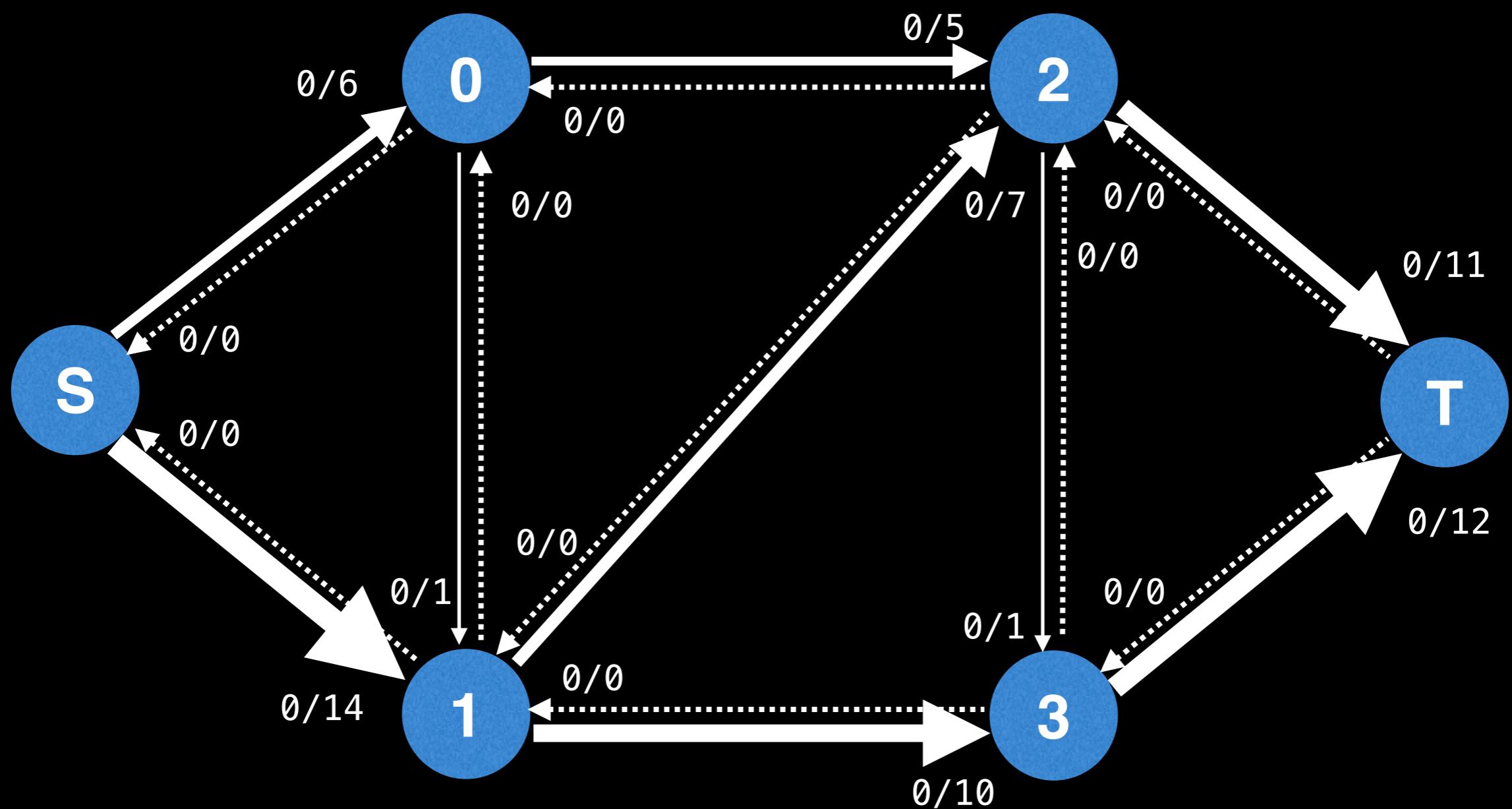


Next, compute the starting value for  $\Delta$  which is the smallest power of 2 less than or equal to  $U (=14)$ .

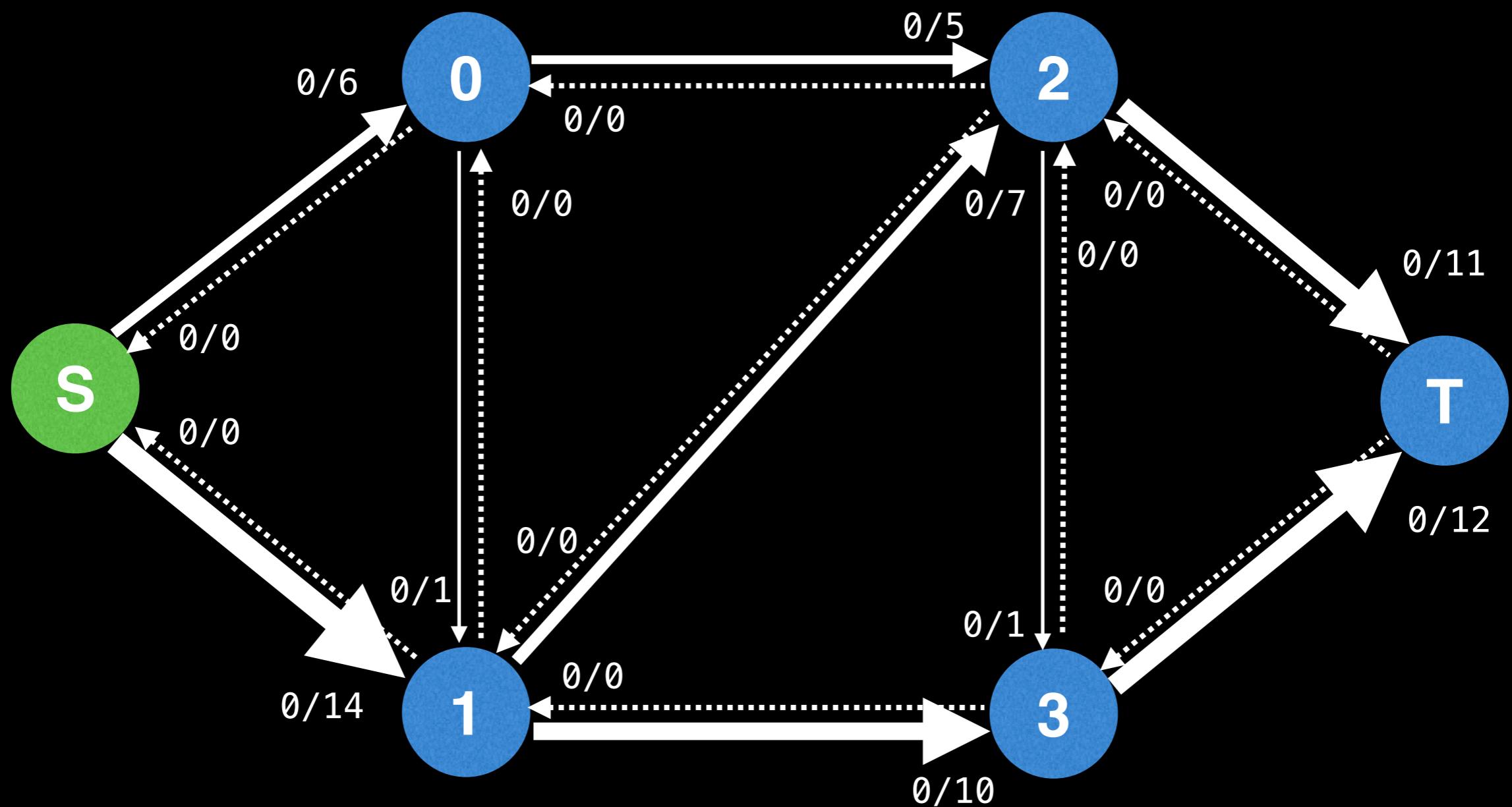
$\Delta = 2^3 = 8$ , since the next power of 2 is  $2^4 = 16 > U$



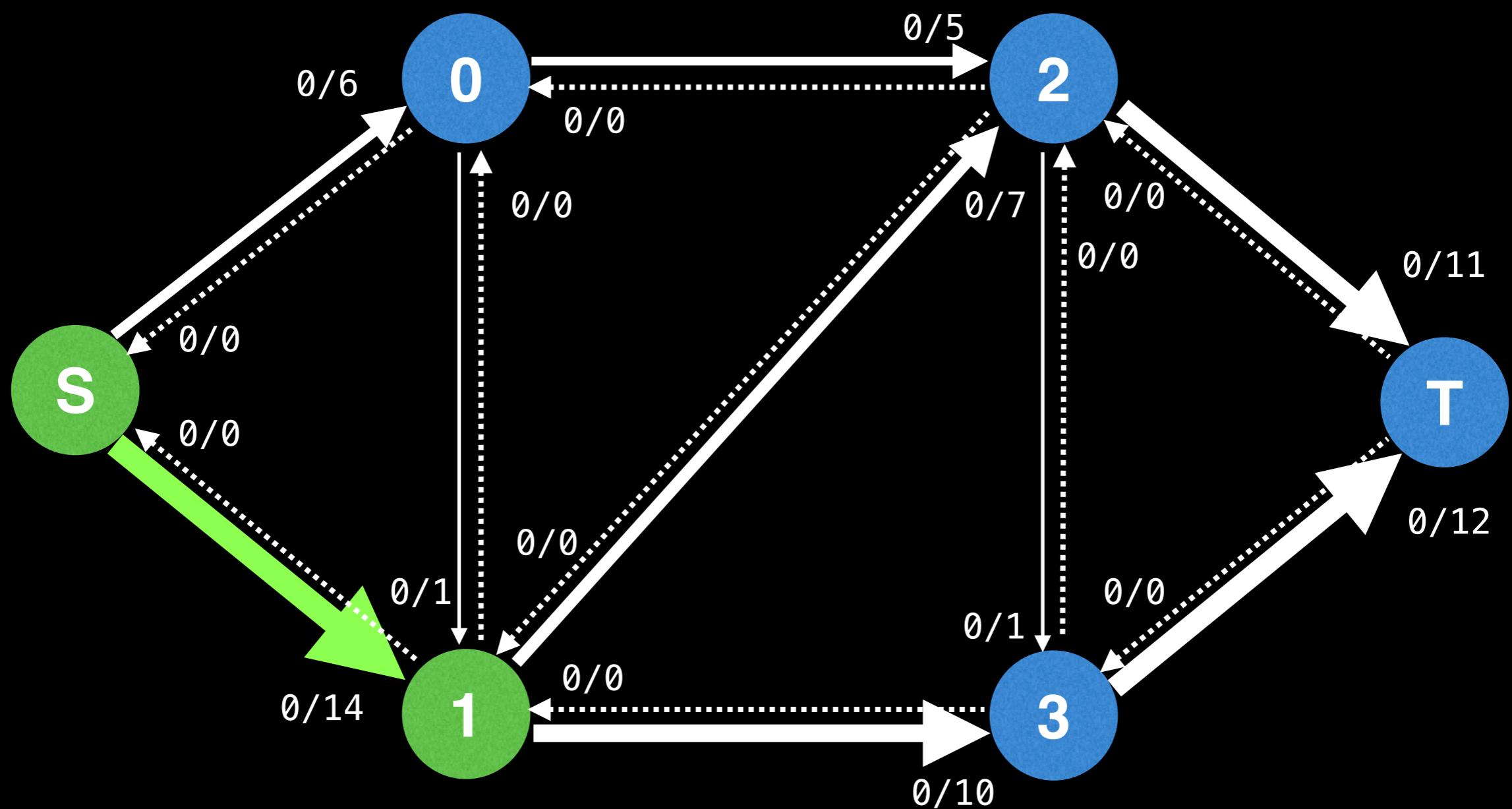
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 8$ .



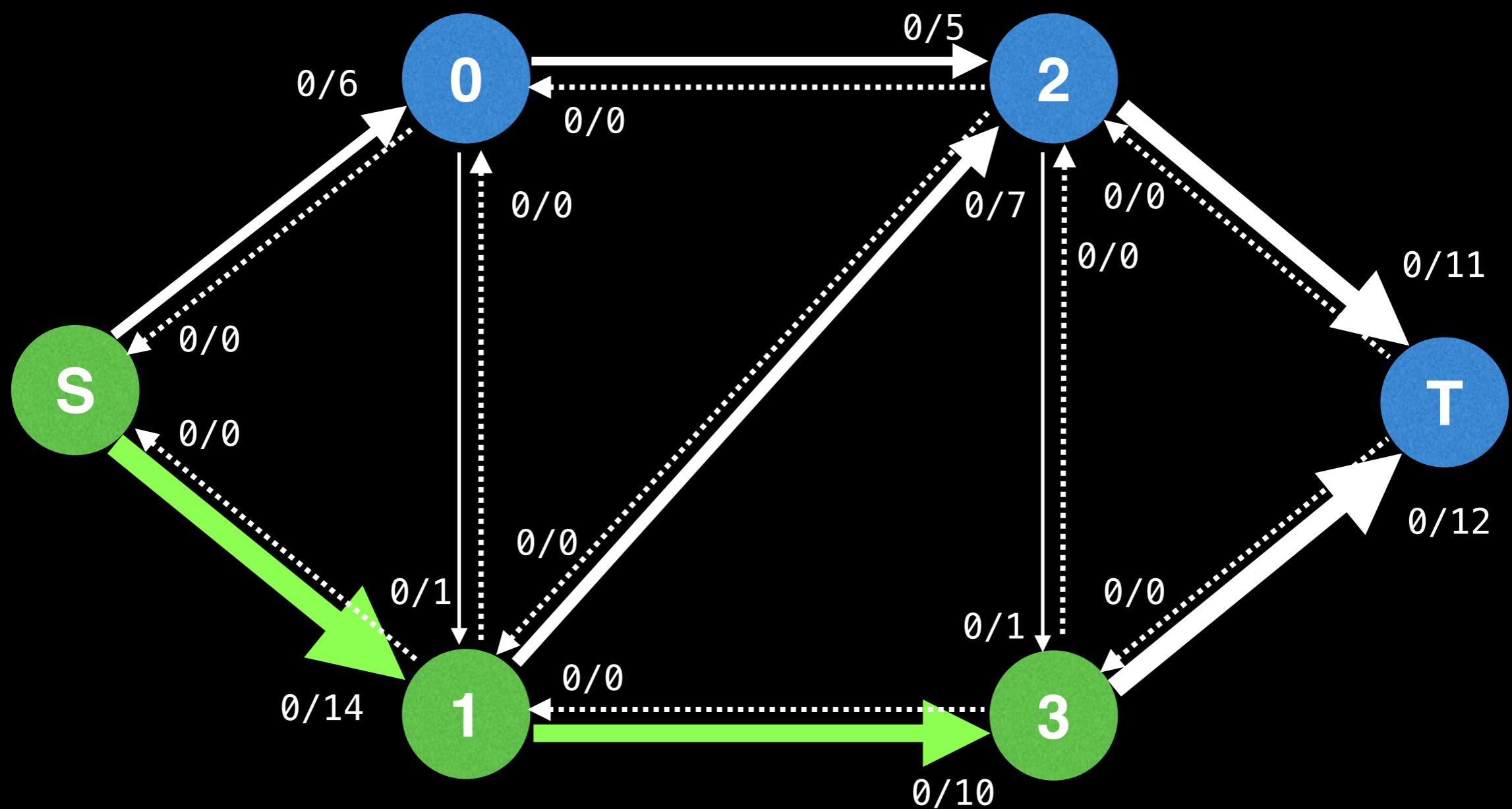
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 8$ .



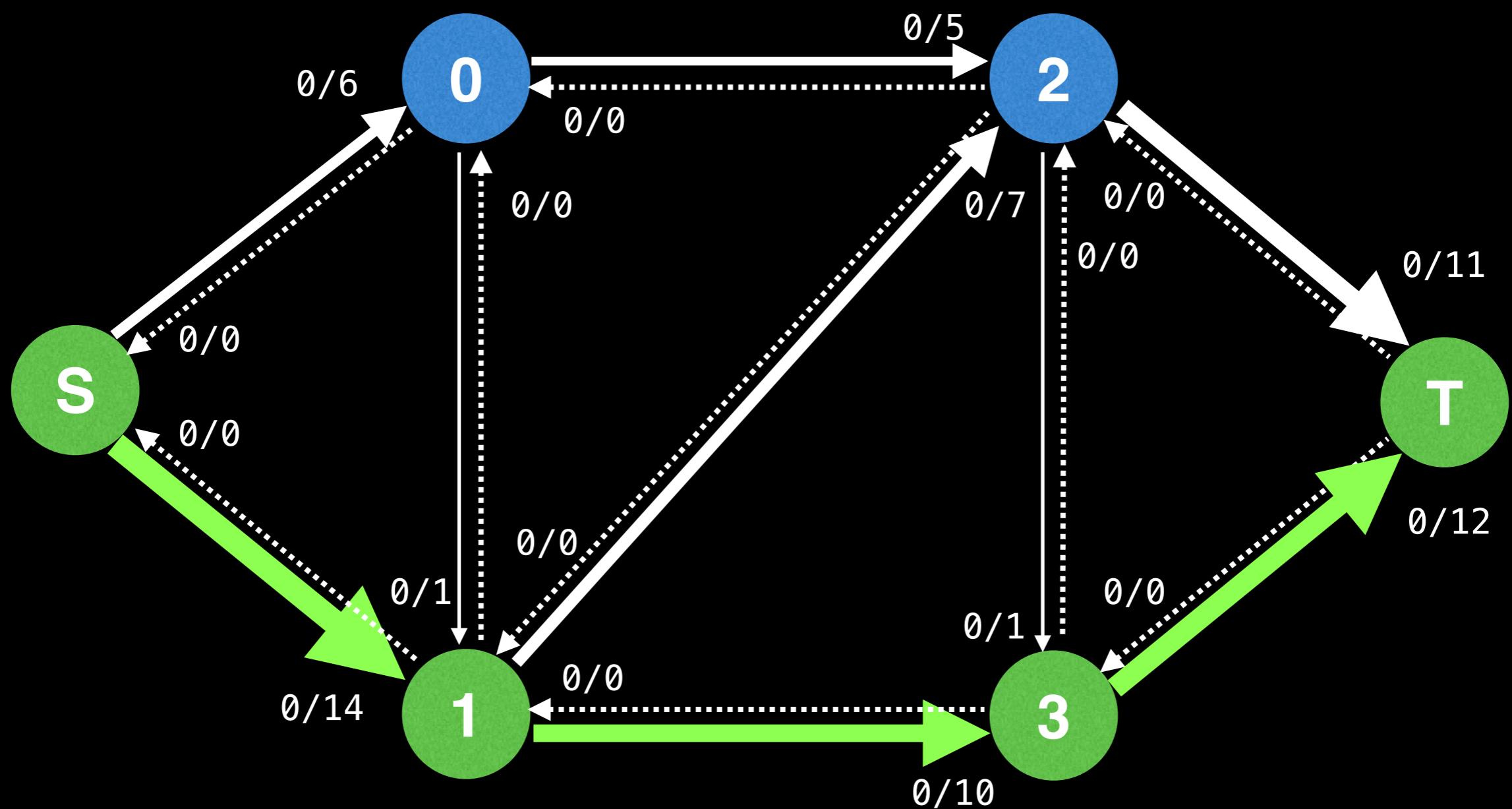
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 8$ .



Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 8$ .

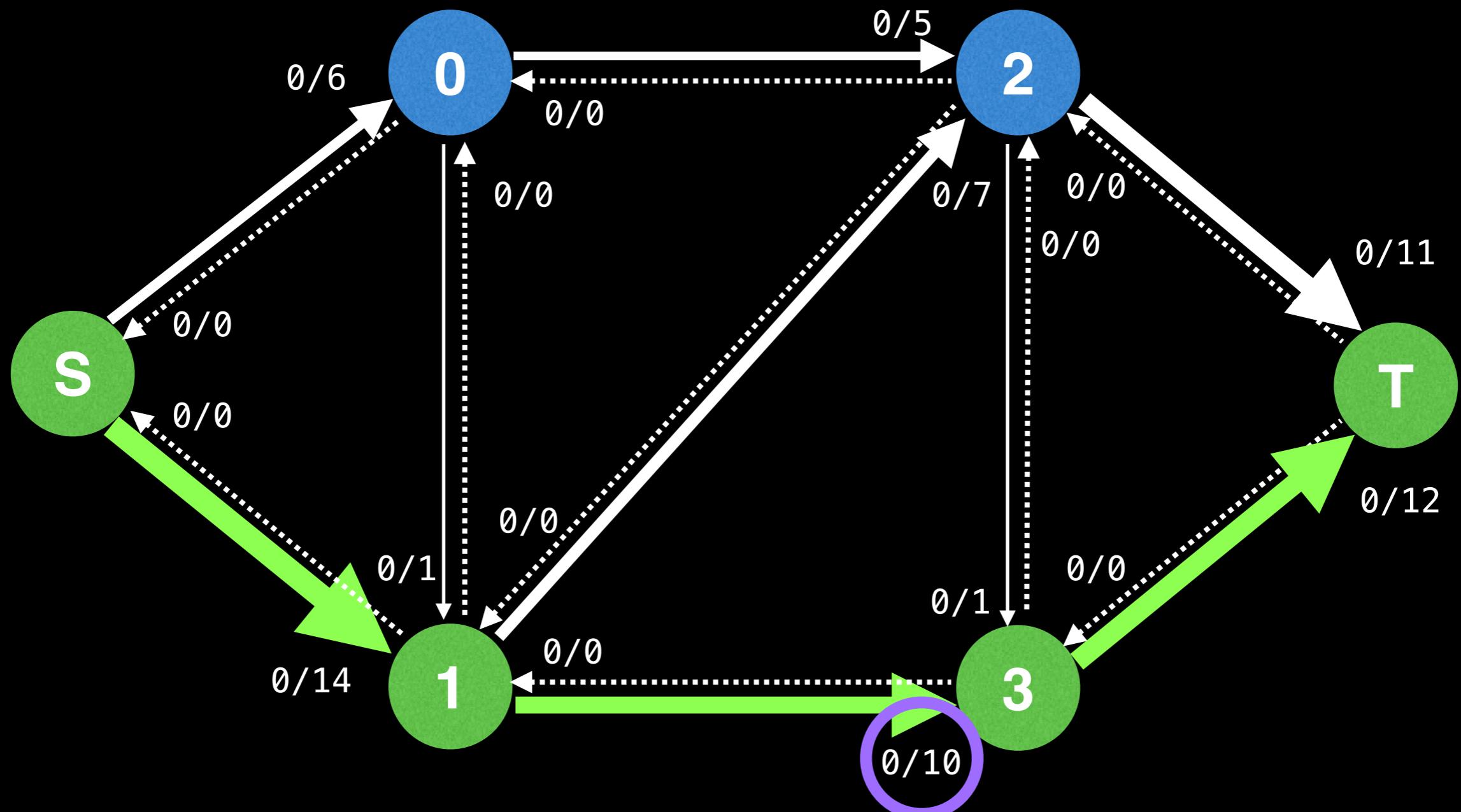


Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 8$ .



Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 8$ .

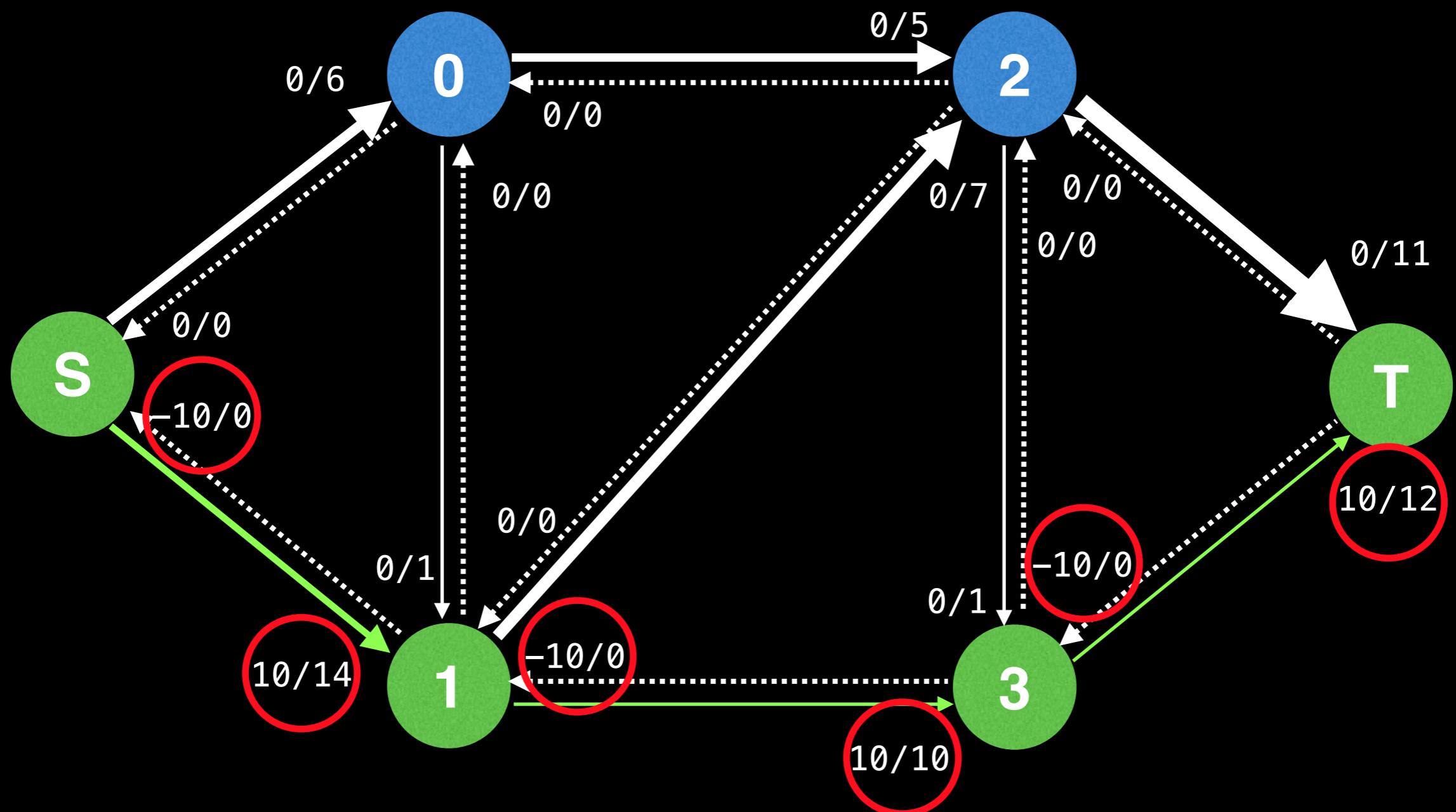
The bottleneck value is 10, because  $\min(14-0, 10-0, 12-0) = 10$  is the smallest remaining capacity along the path.

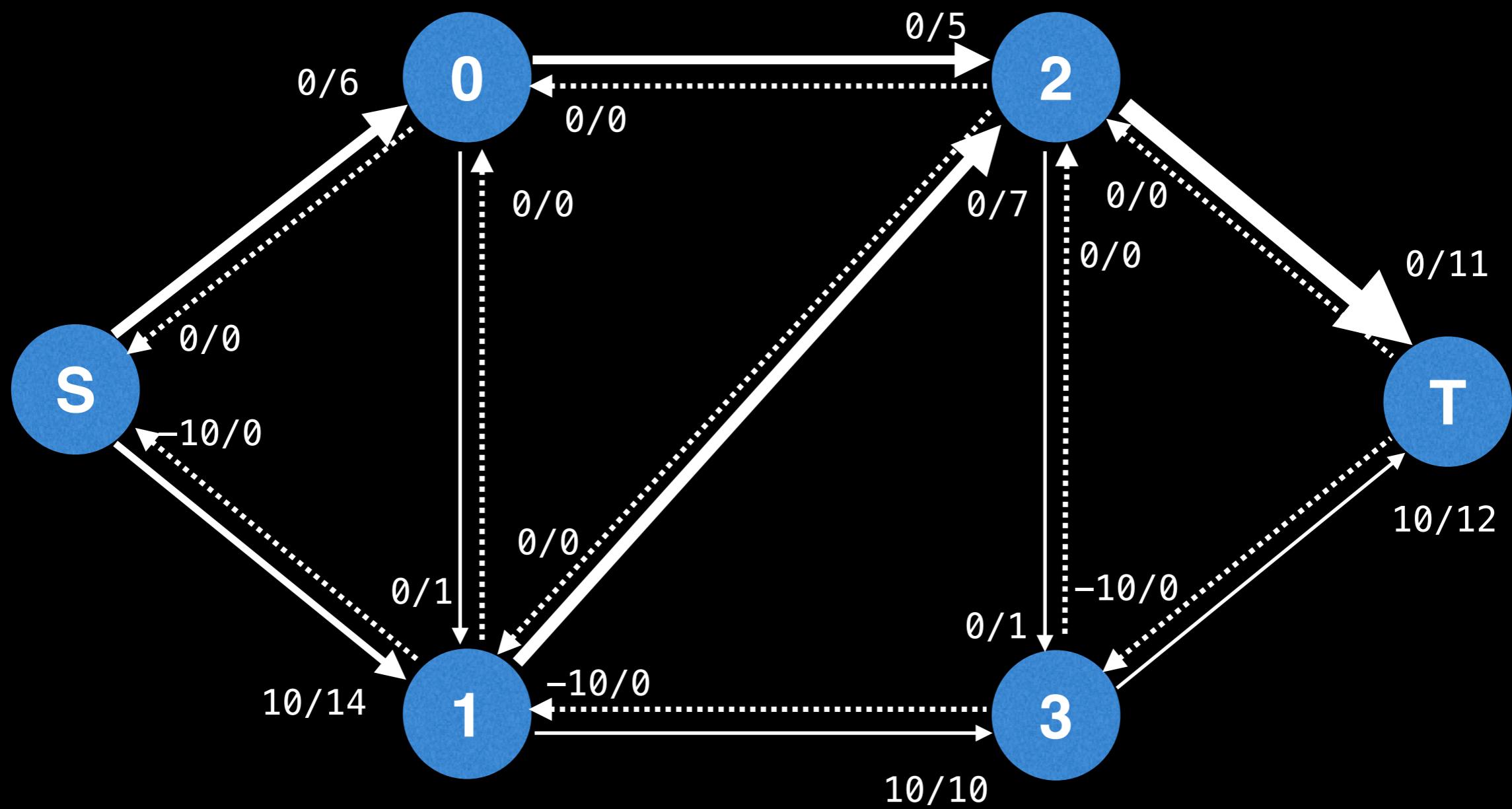


The bottleneck value will always be greater than or equal to  $\Delta$ .

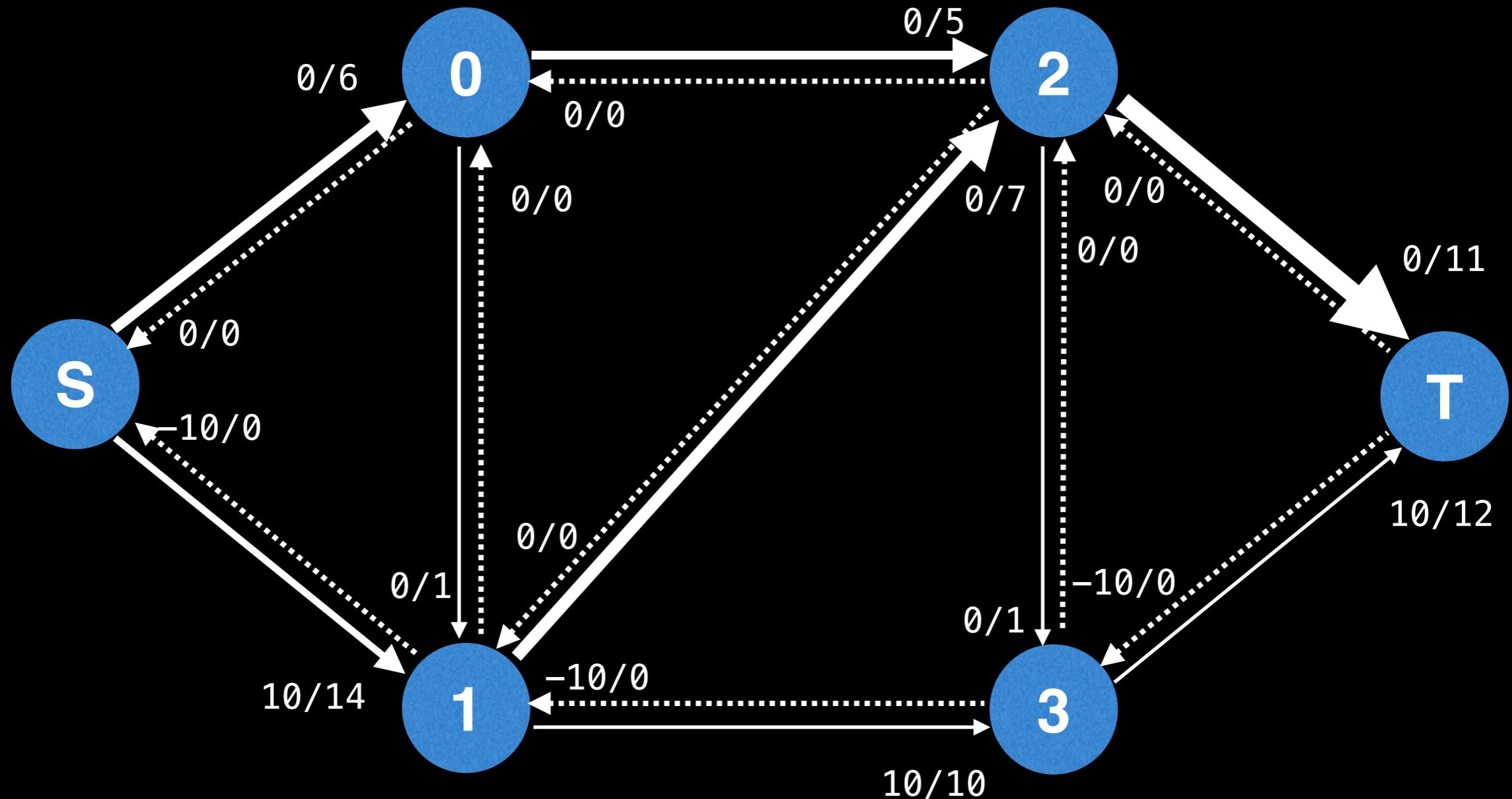
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 8$ .

The bottleneck value is 10, because  $\min(14-0, 10-0, 12-0) = 10$  is the smallest remaining capacity along the path.

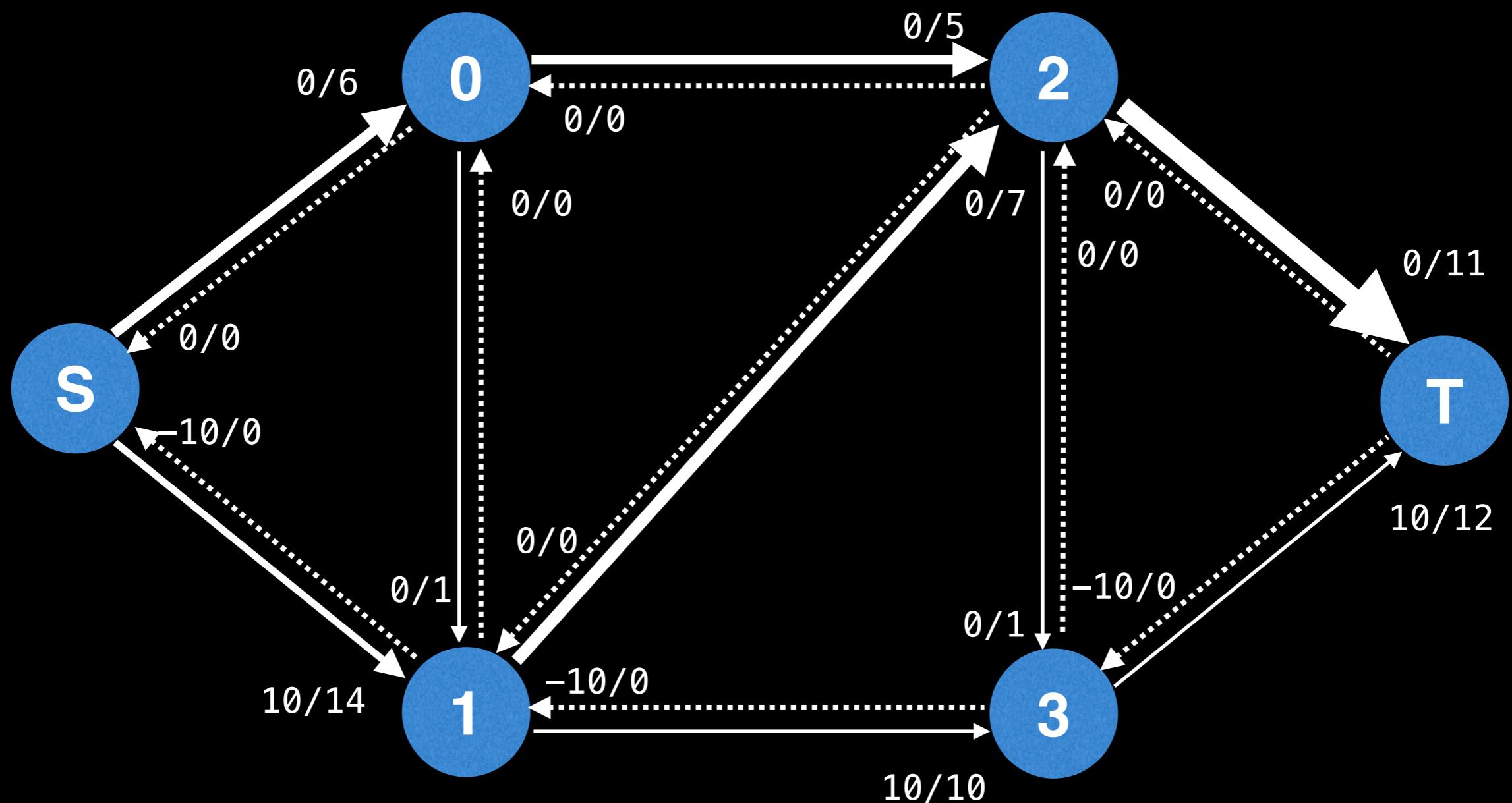




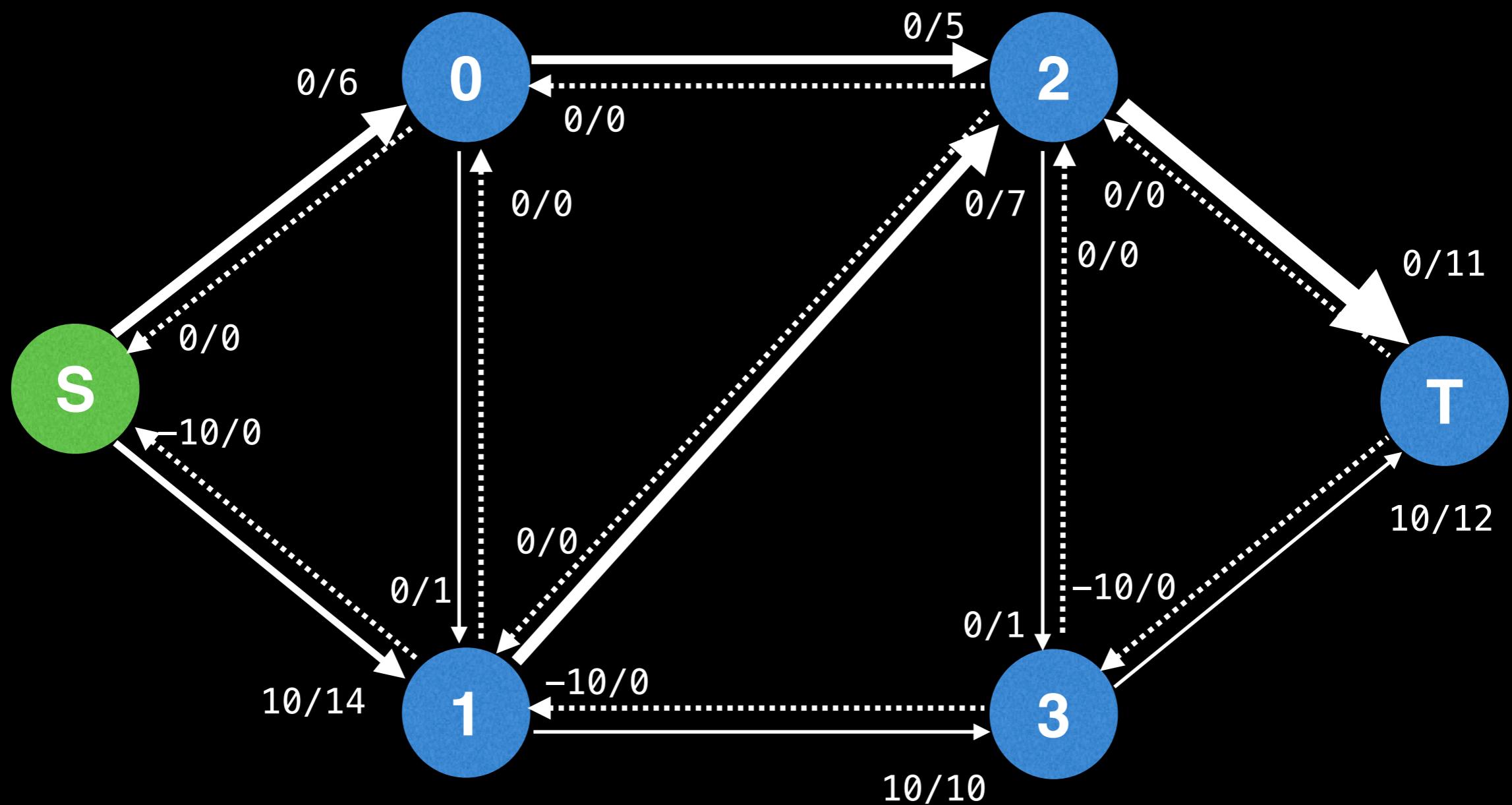
There are no more augmenting paths from  $s \rightarrow t$  which have a remaining capacity greater than or equal to  $\Delta$  ( $=8$ ), so the new  $\Delta$  is:  $\Delta = \Delta / 2 = 4$ .



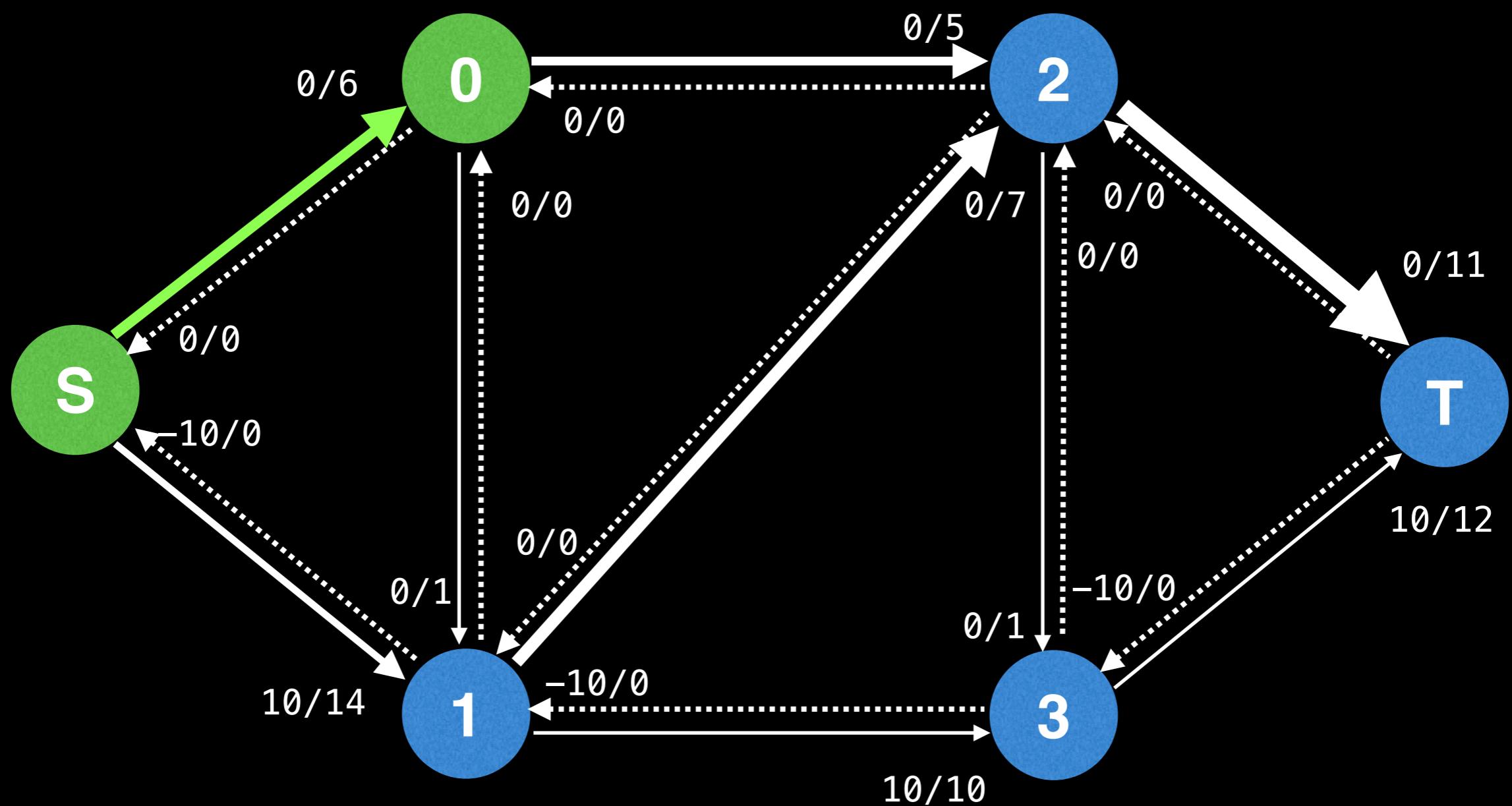
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .



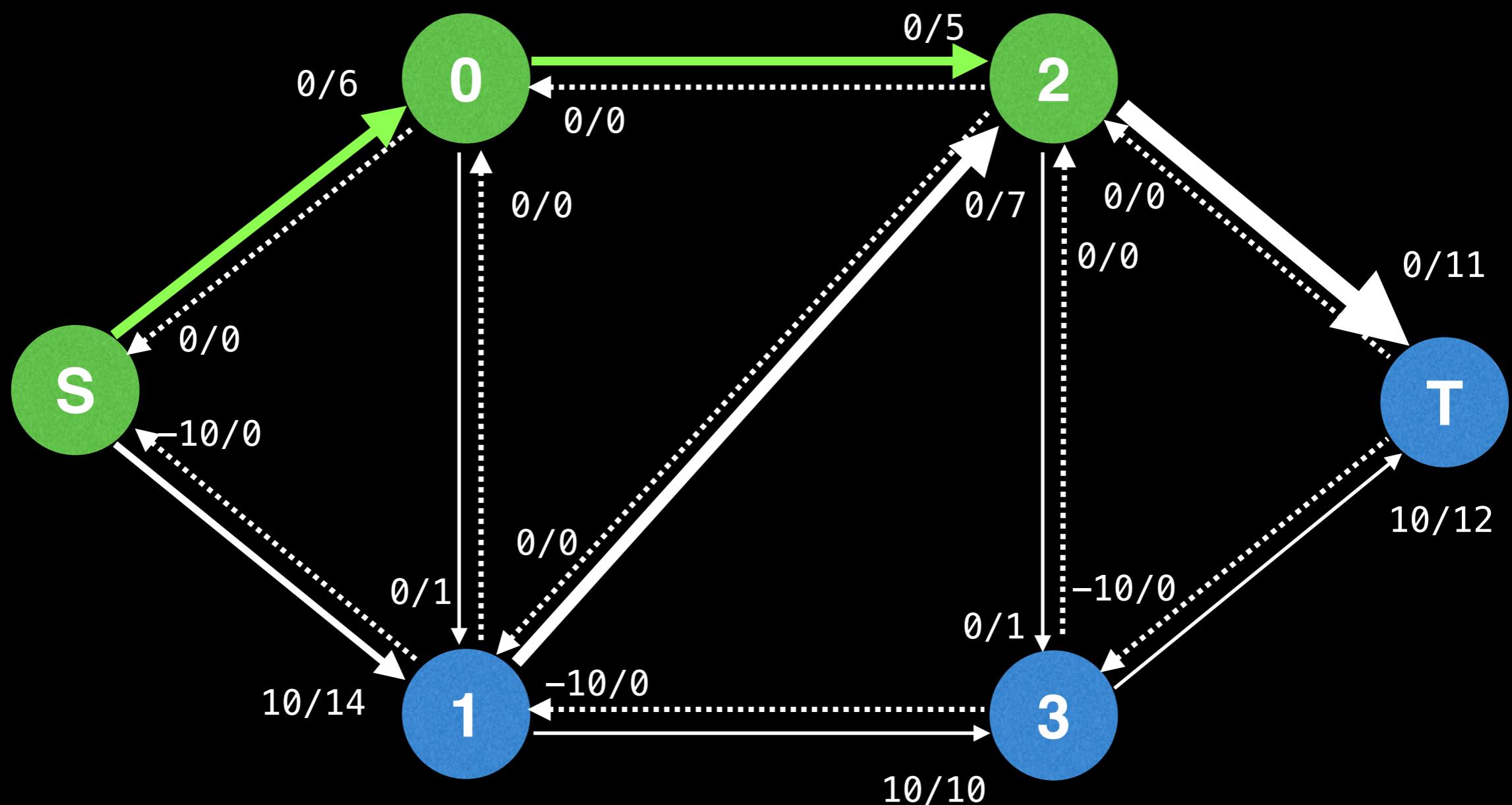
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .



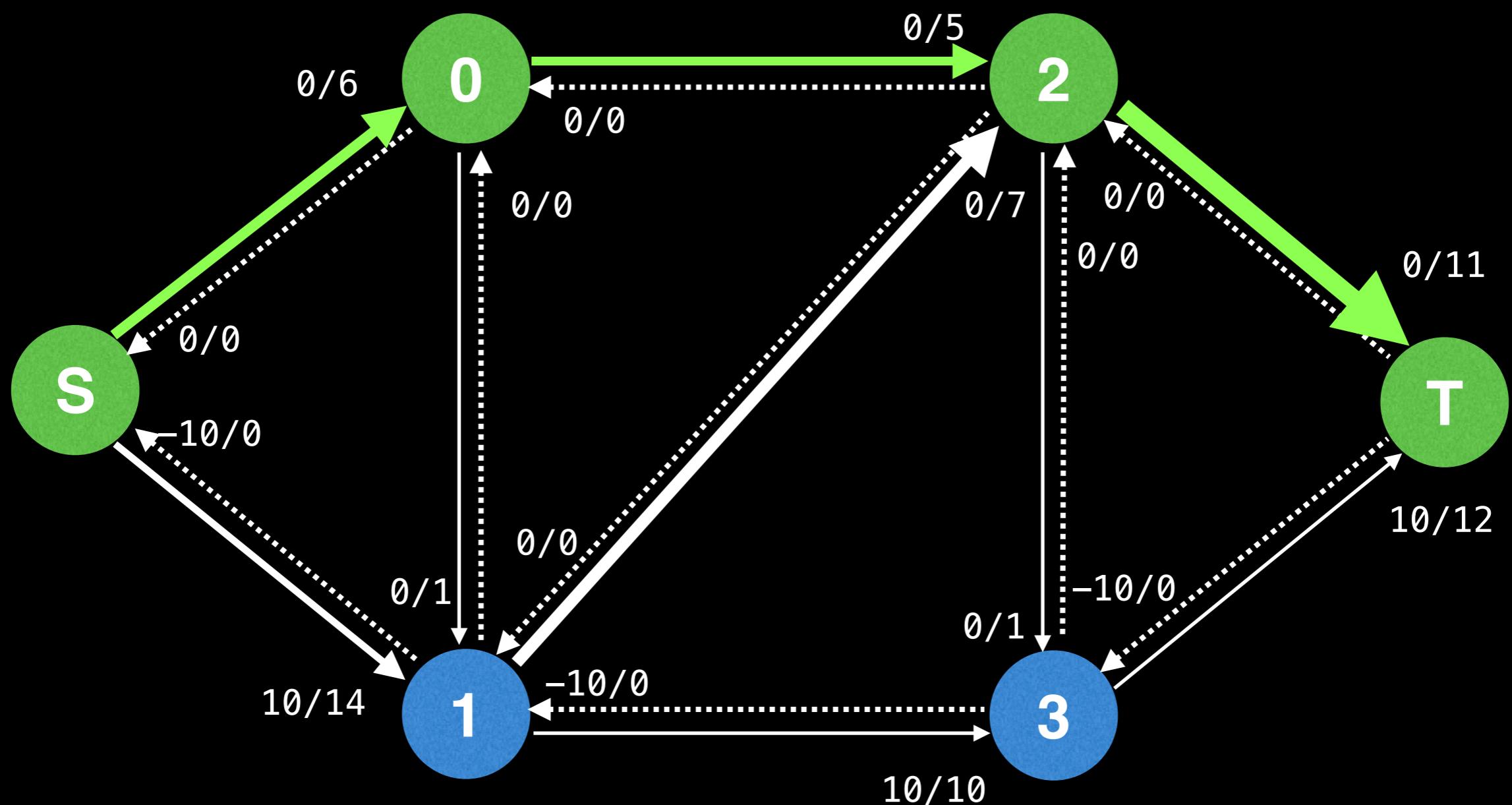
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .



Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .

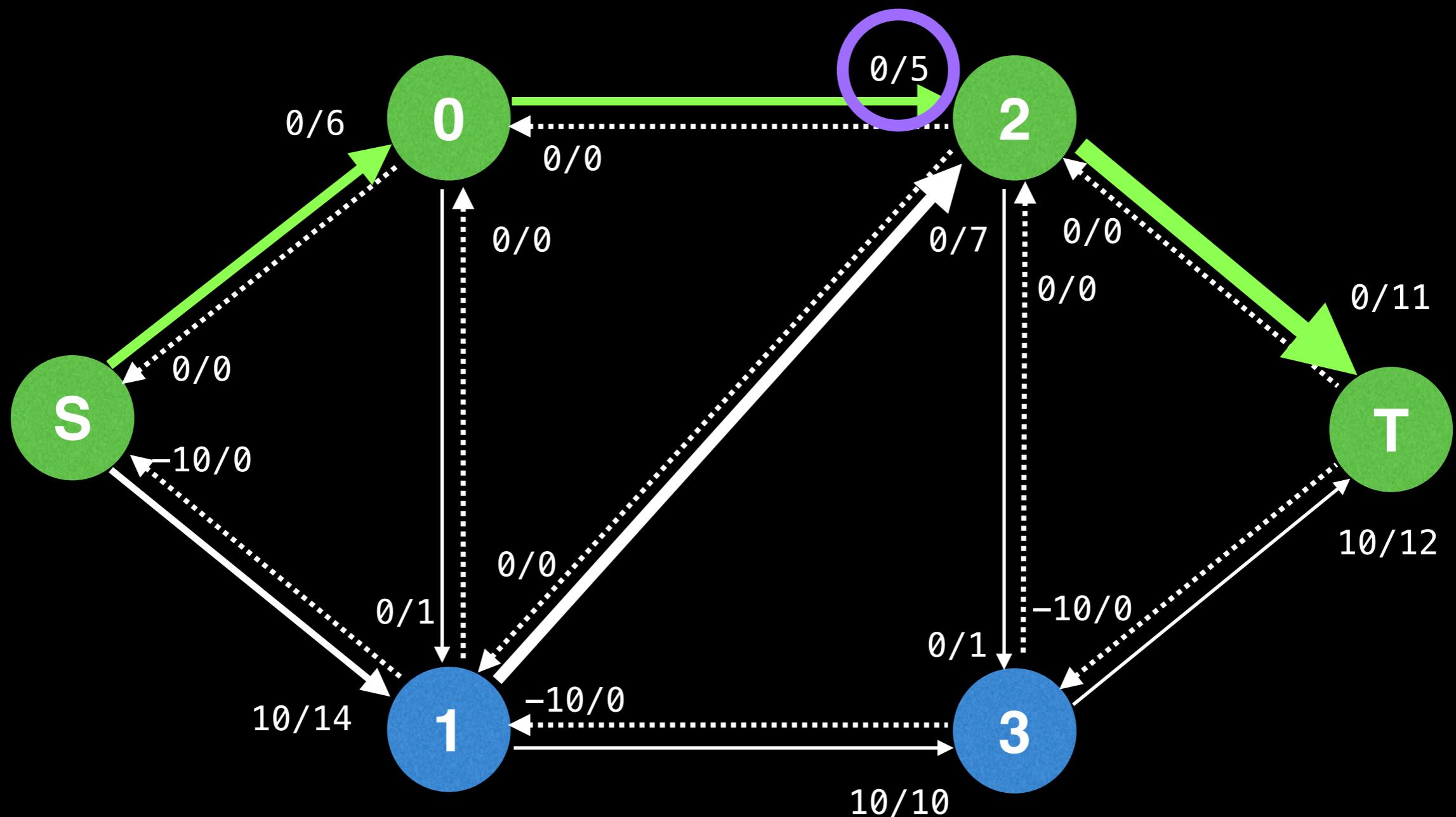


Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .



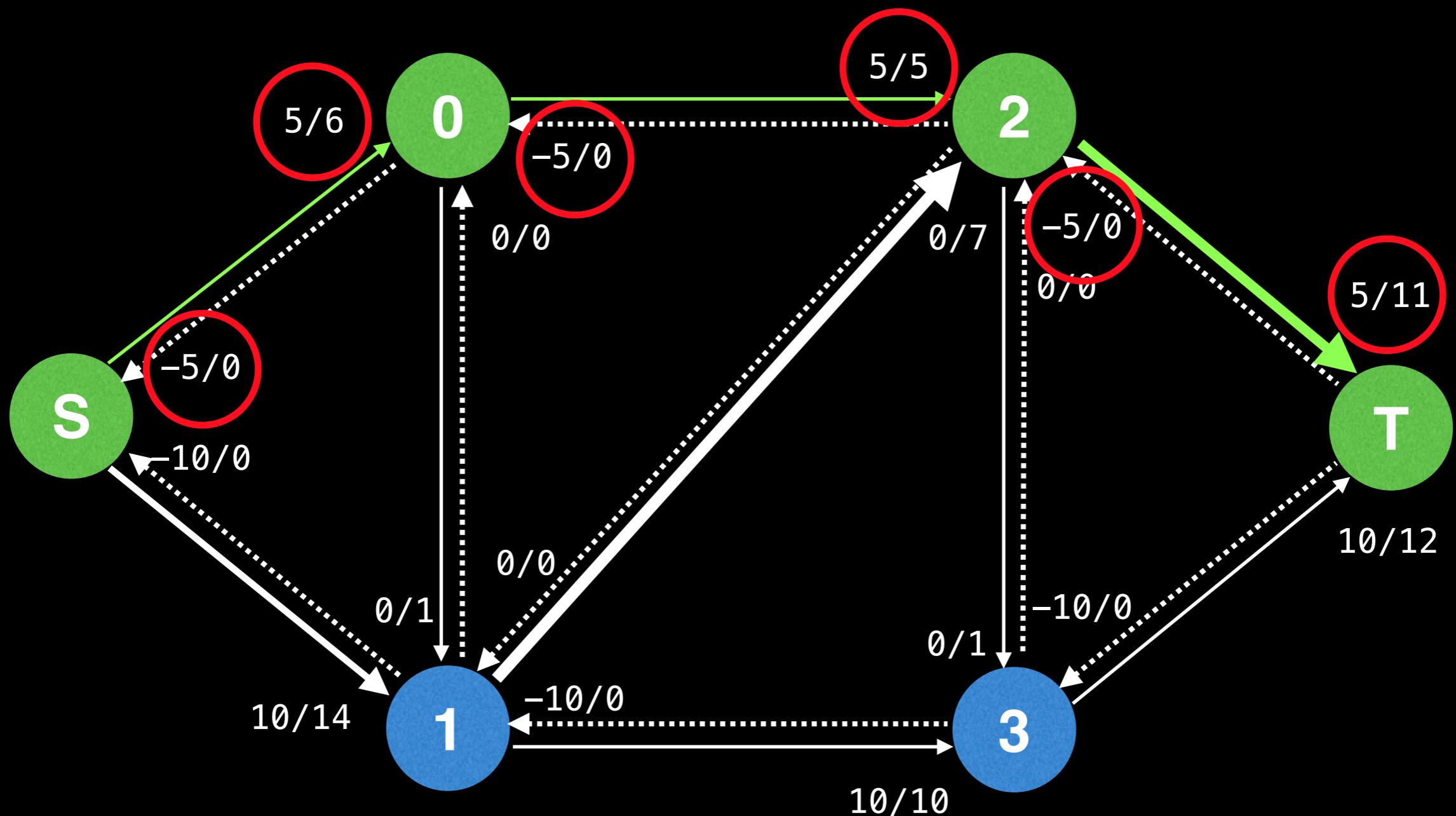
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .

The bottleneck value is 5, because  $\min(6-0, 5-0, 11-0) = 5$  is the smallest remaining capacity along the path.

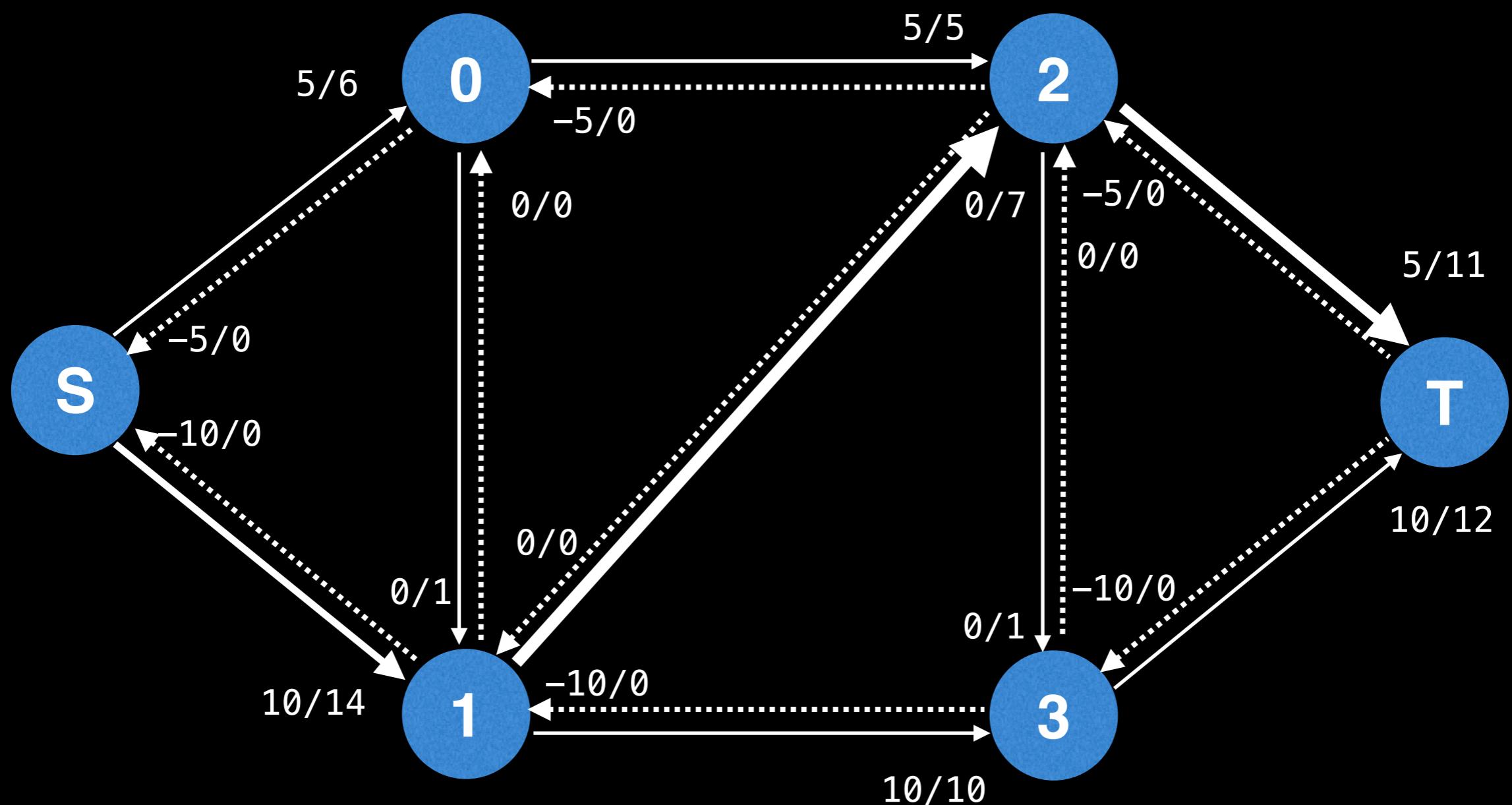


Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .

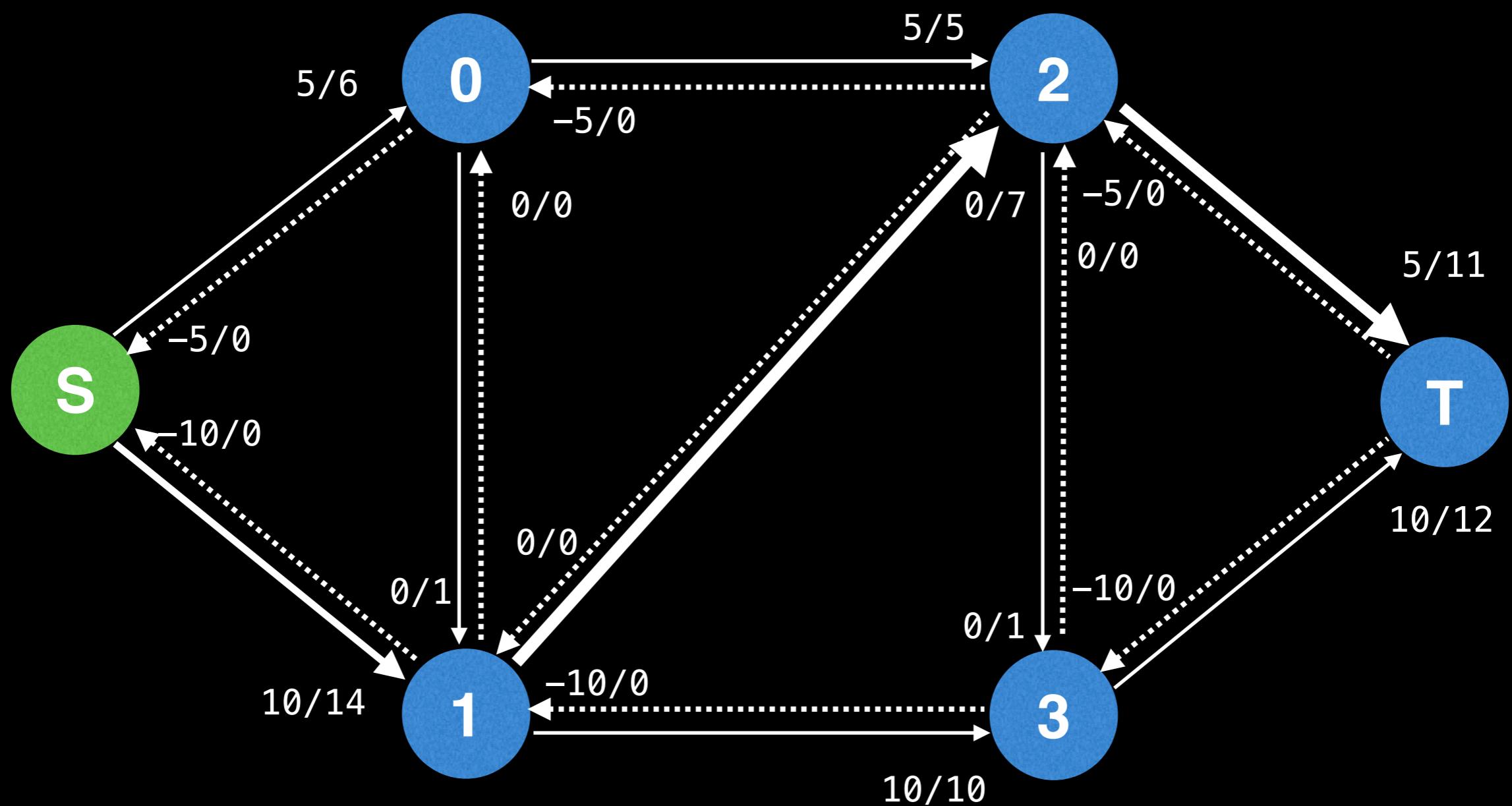
The bottleneck value is 5, because  $\min(6-0, 5-0, 11-0) = 5$  is the smallest remaining capacity along the path.



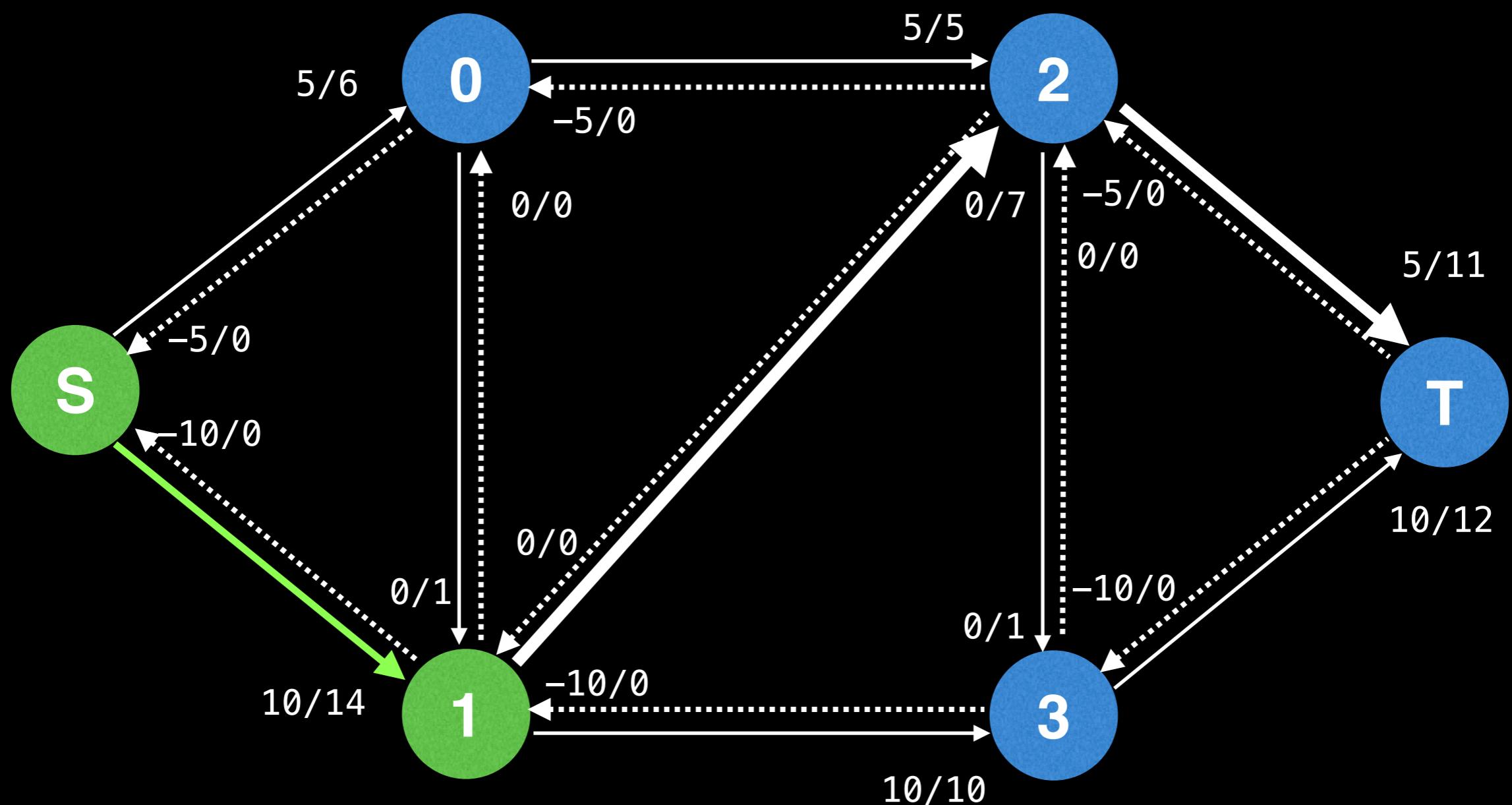
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .



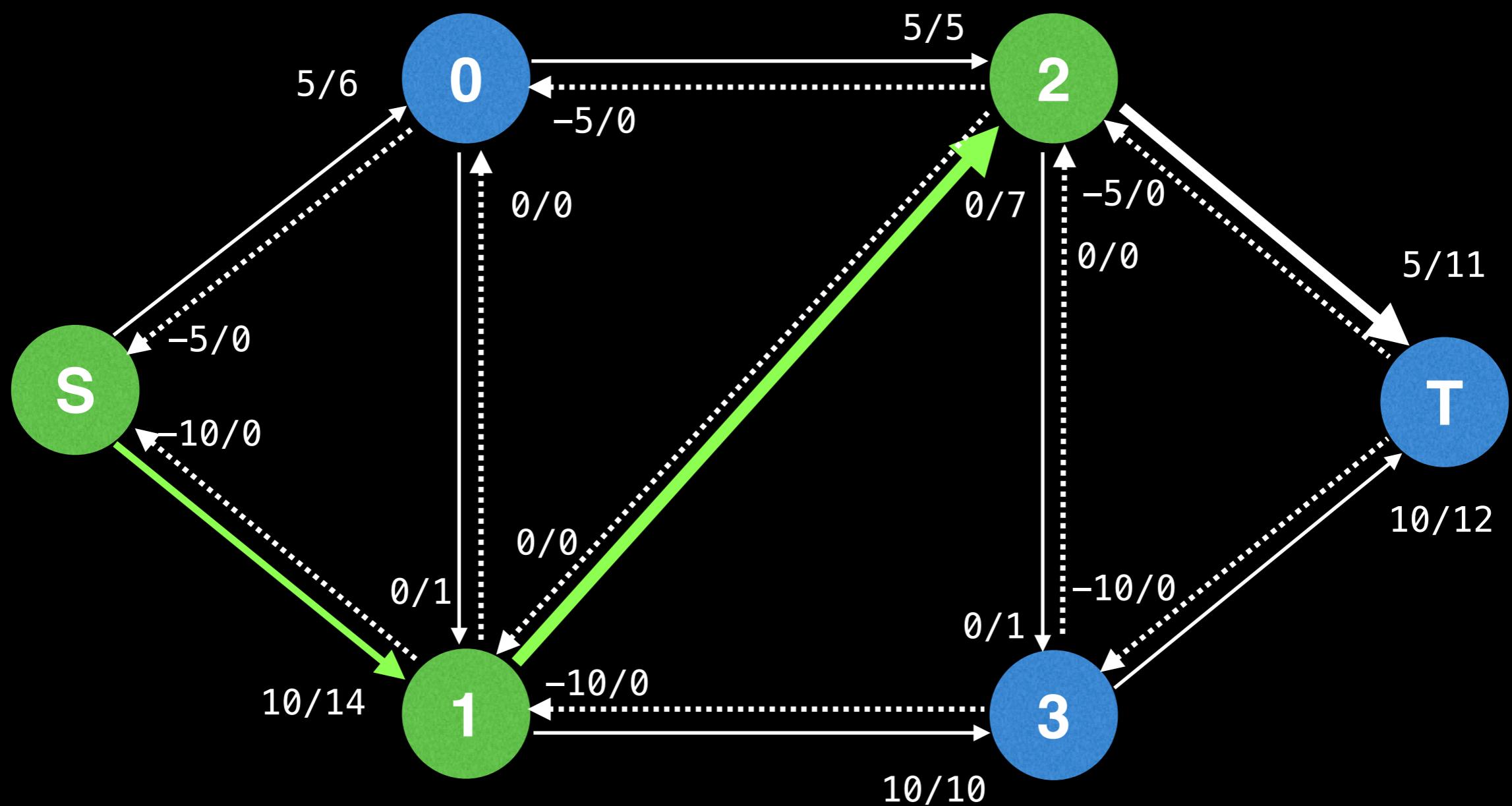
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .



Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .

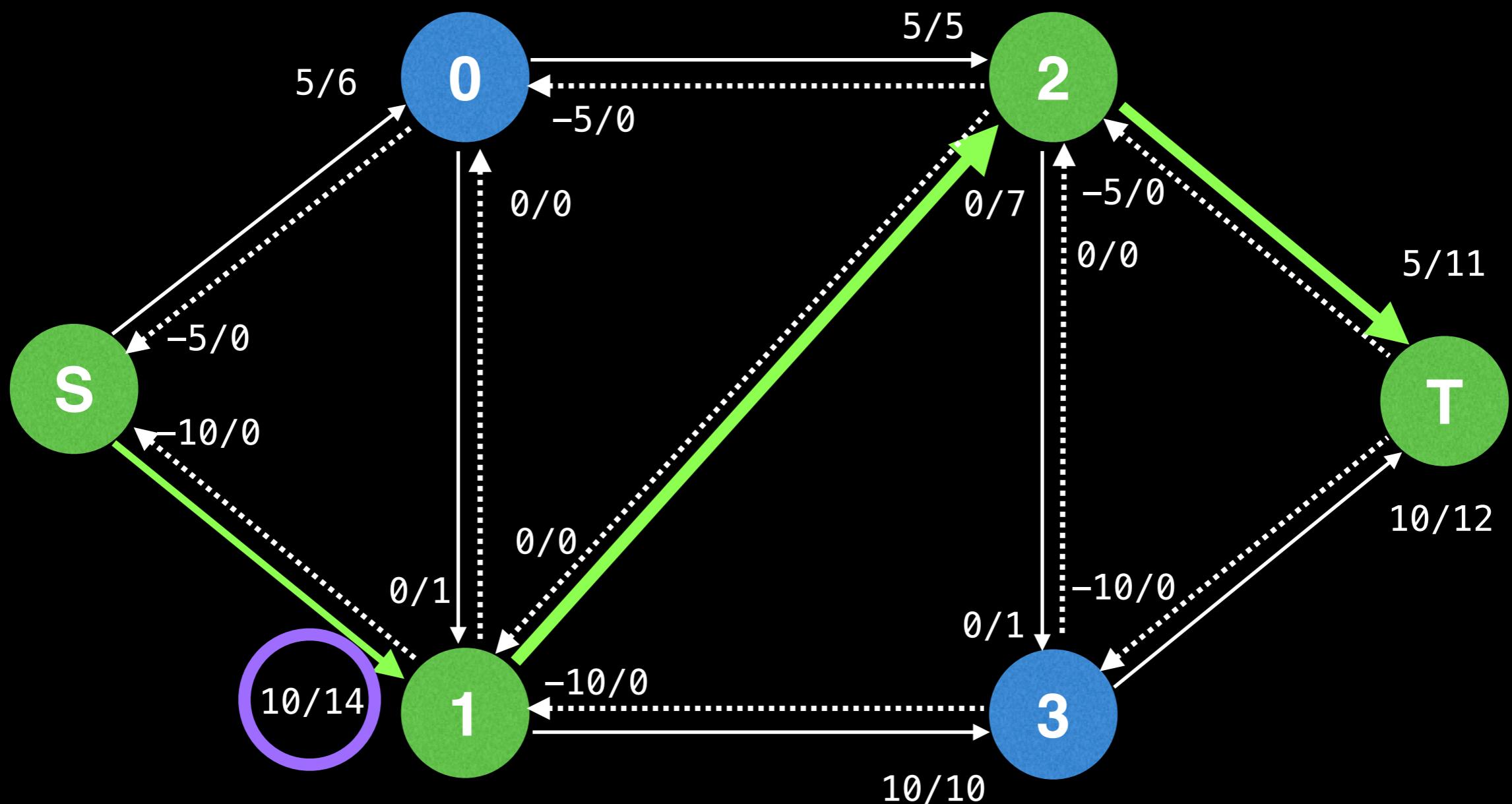


Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .



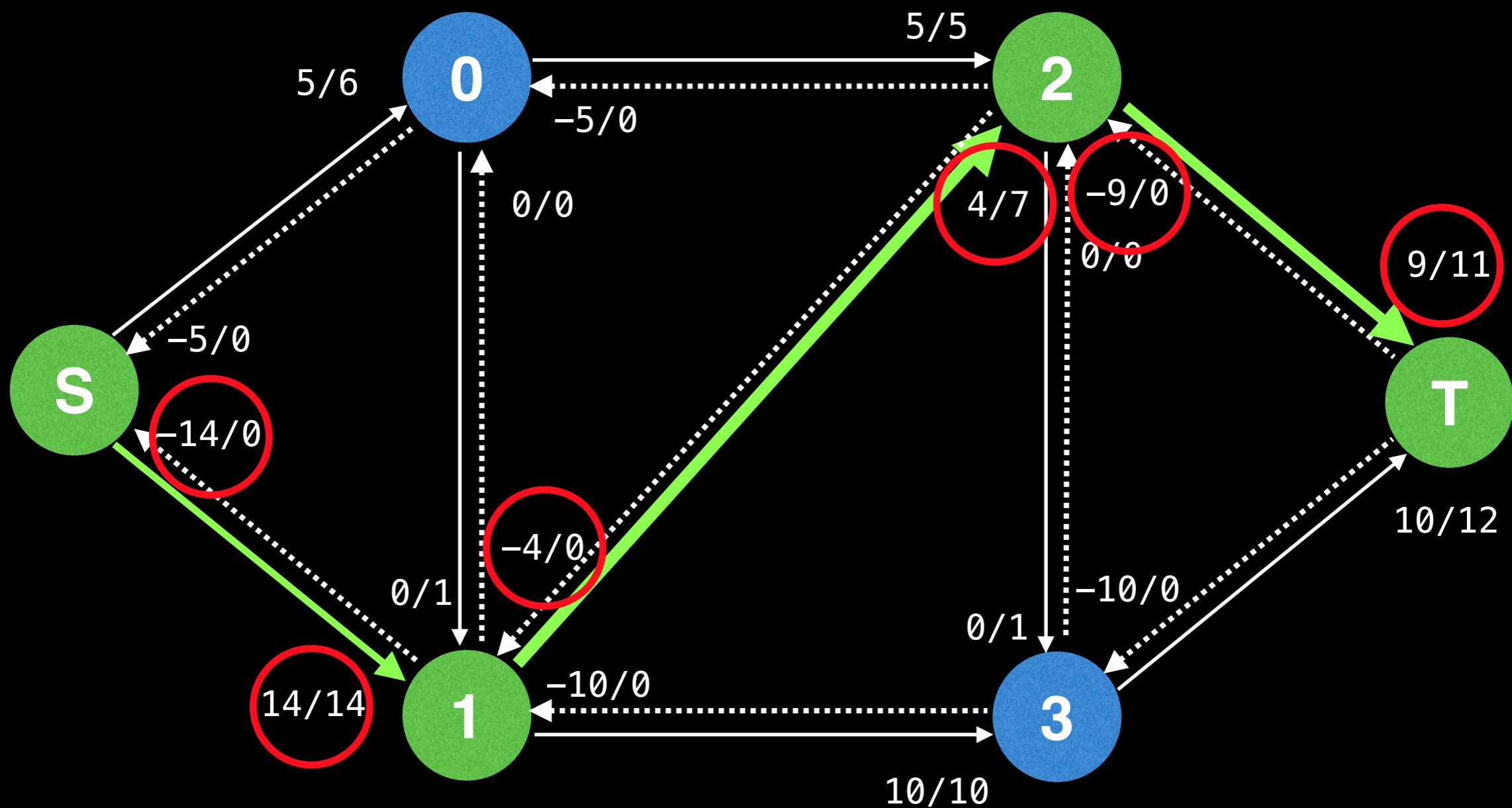
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .

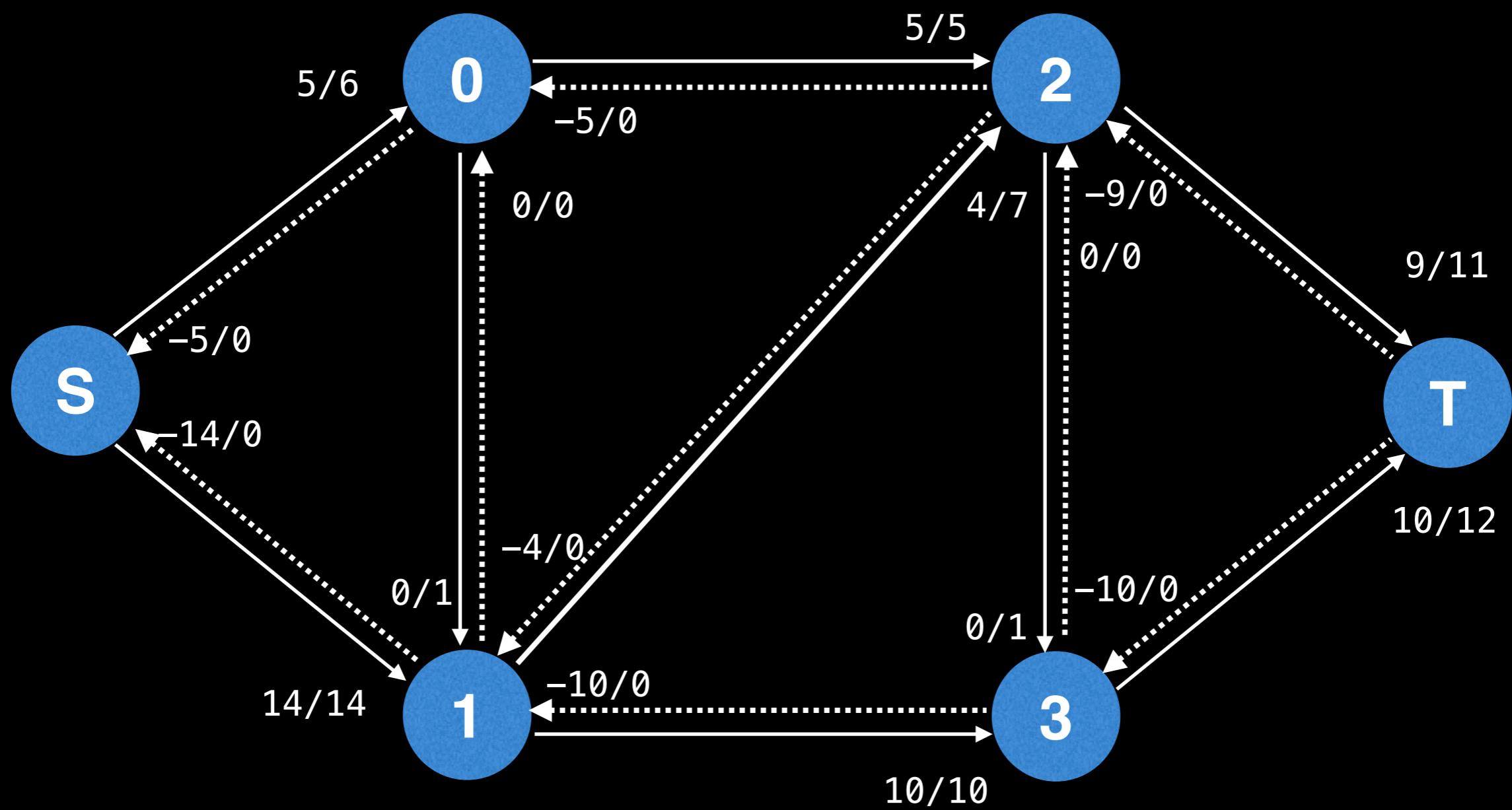
The bottleneck value is 4, because  $\min(14-10, 7-0, 11-5) = 4$  is the smallest remaining capacity along the path.



Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 4$ .

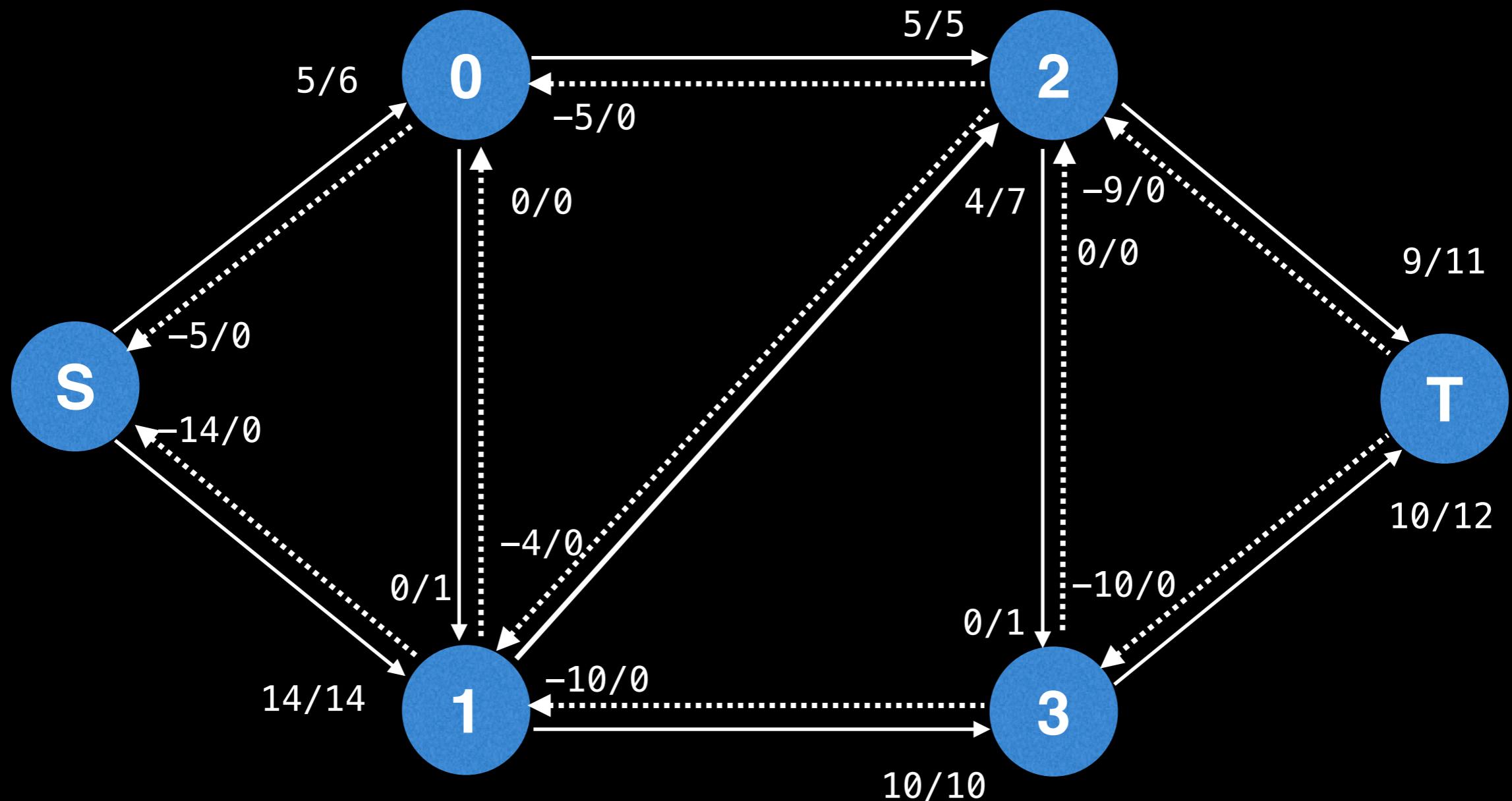
The bottleneck value is 4, because  $\min(14-10, 7-0, 11-5) = 4$  is the smallest remaining capacity along the path.



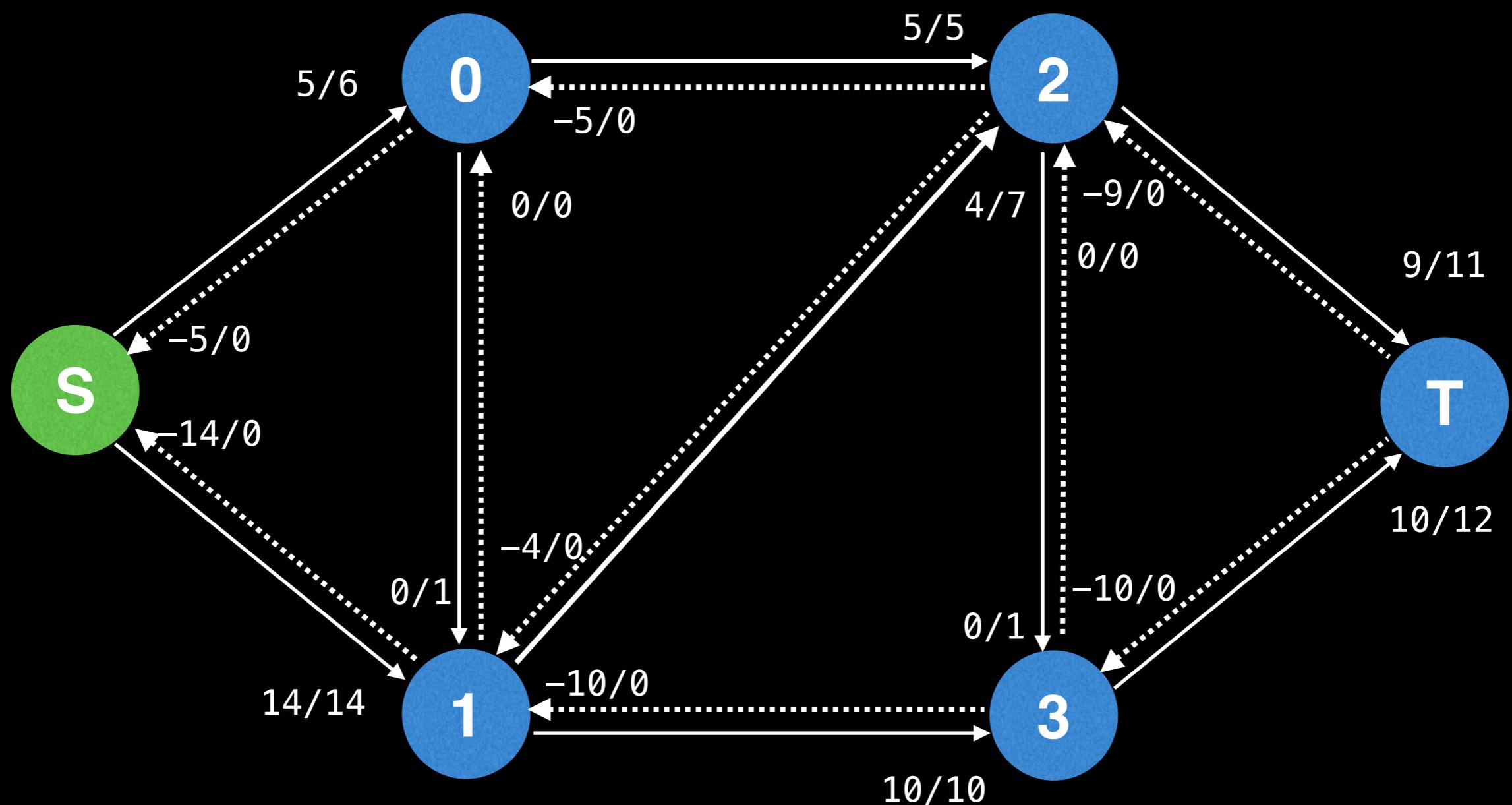


There are no more augmenting paths from  $s \rightarrow t$  which have a remaining capacity greater than or equal to  $\Delta (=4)$ , so the new  $\Delta$  is:  $\Delta = \Delta / 2 = 2$ .

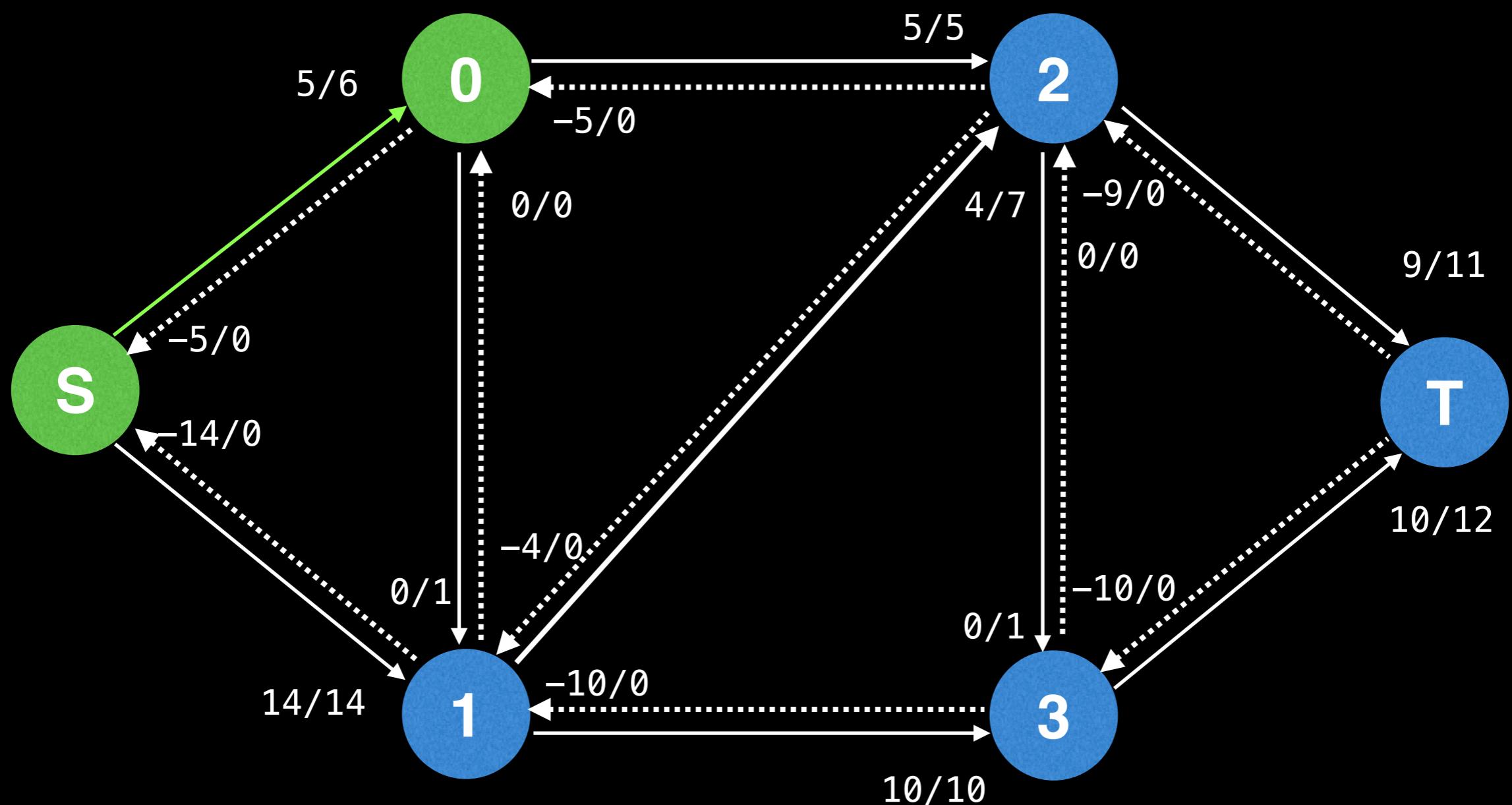
However, there are also no more augmenting paths from  $s \rightarrow t$  which have a remaining capacity greater than or equal to  $\Delta (=2)$ , so the new  $\Delta$  is:  $\Delta = \Delta / 2 = 1$ .



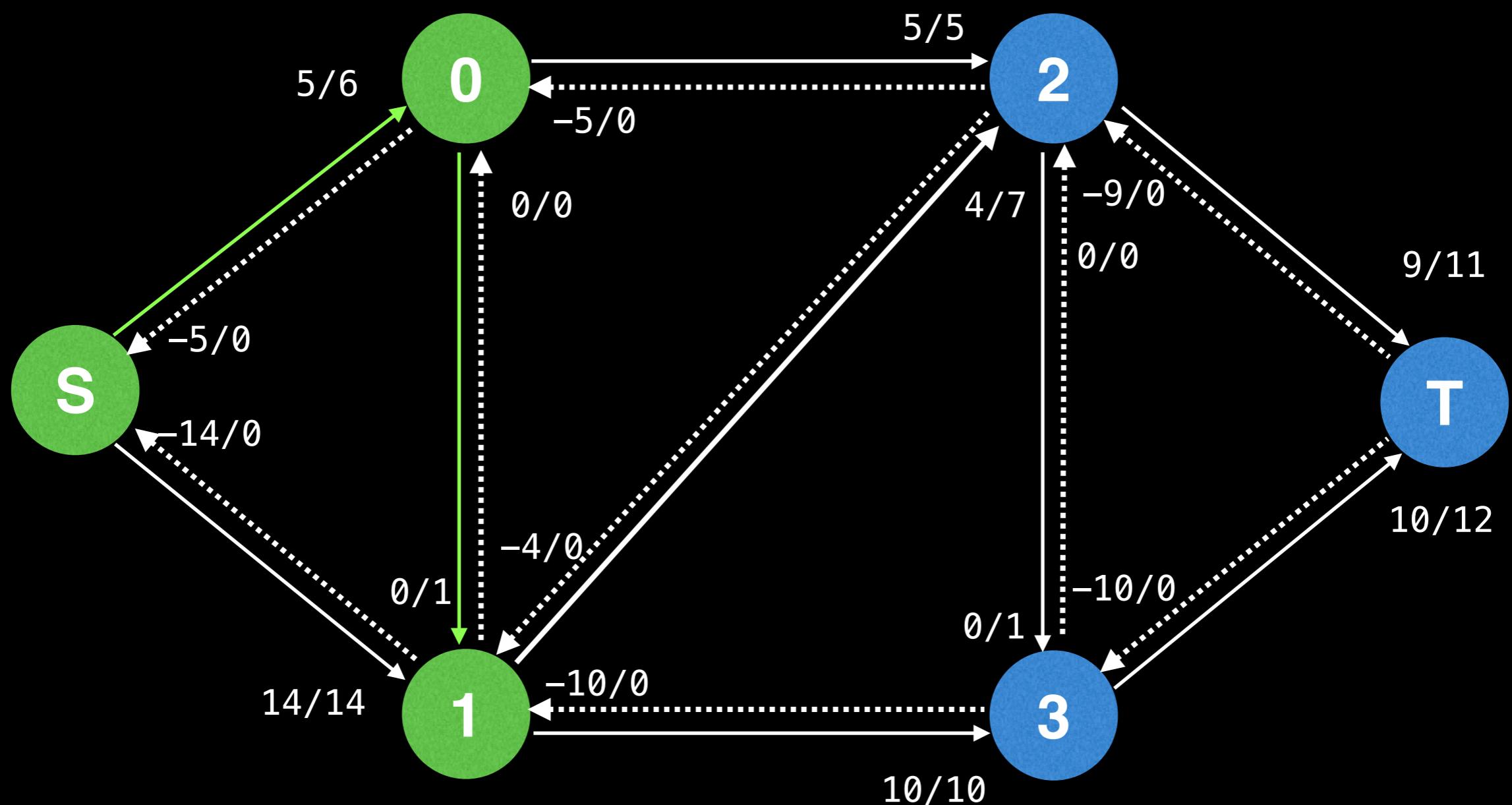
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 1$ .



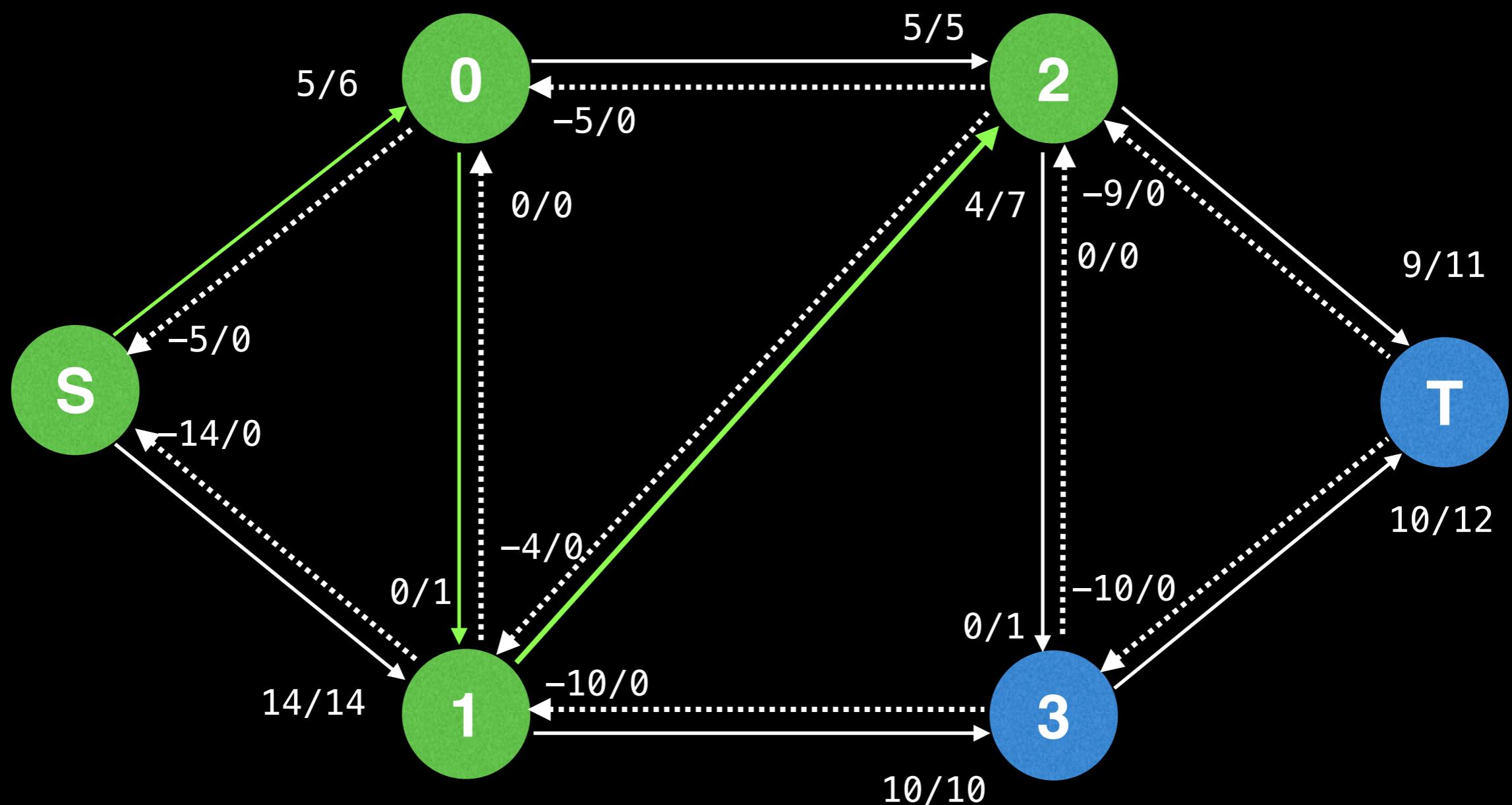
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 1$ .



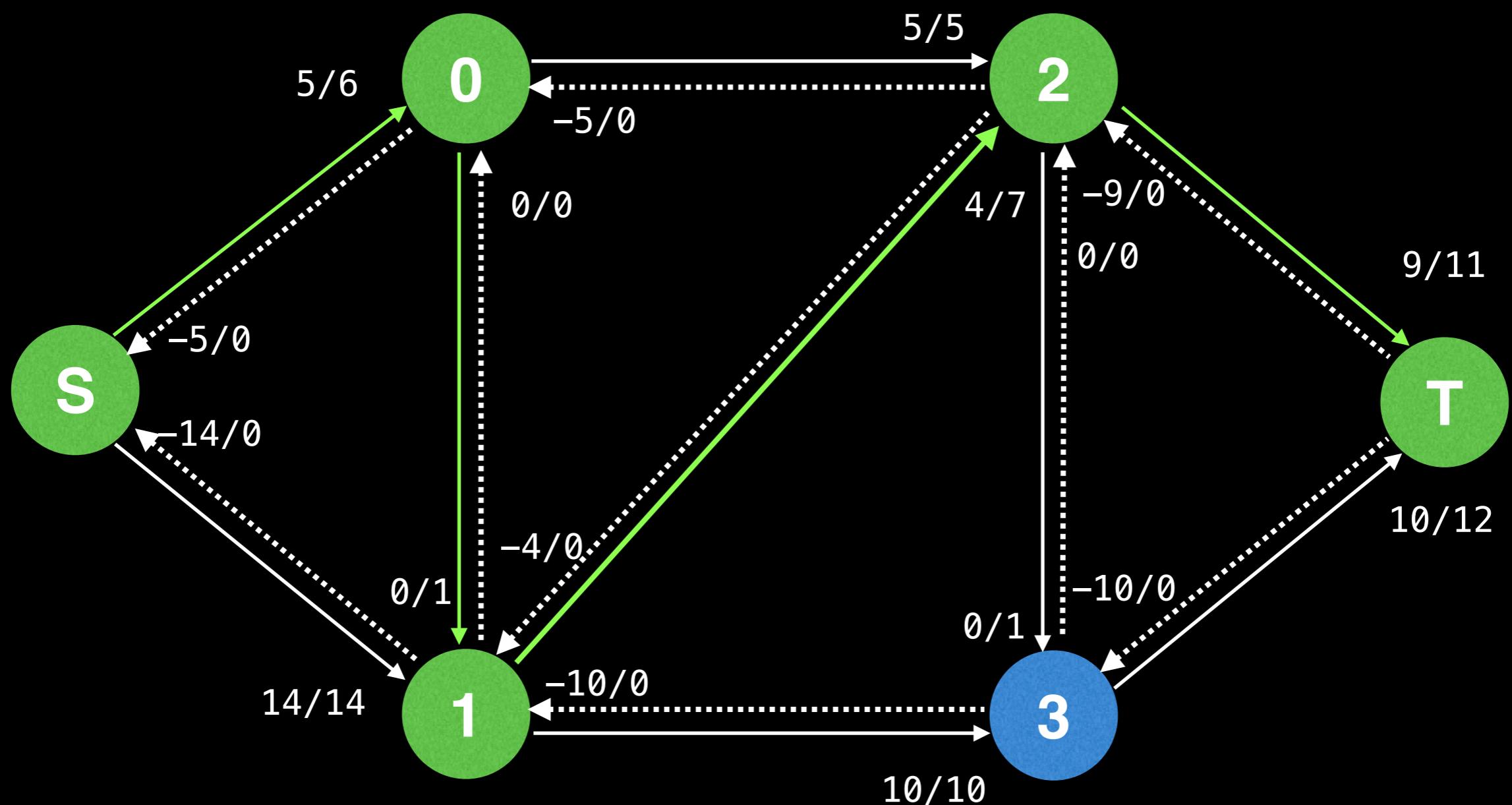
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 1$ .



Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 1$ .

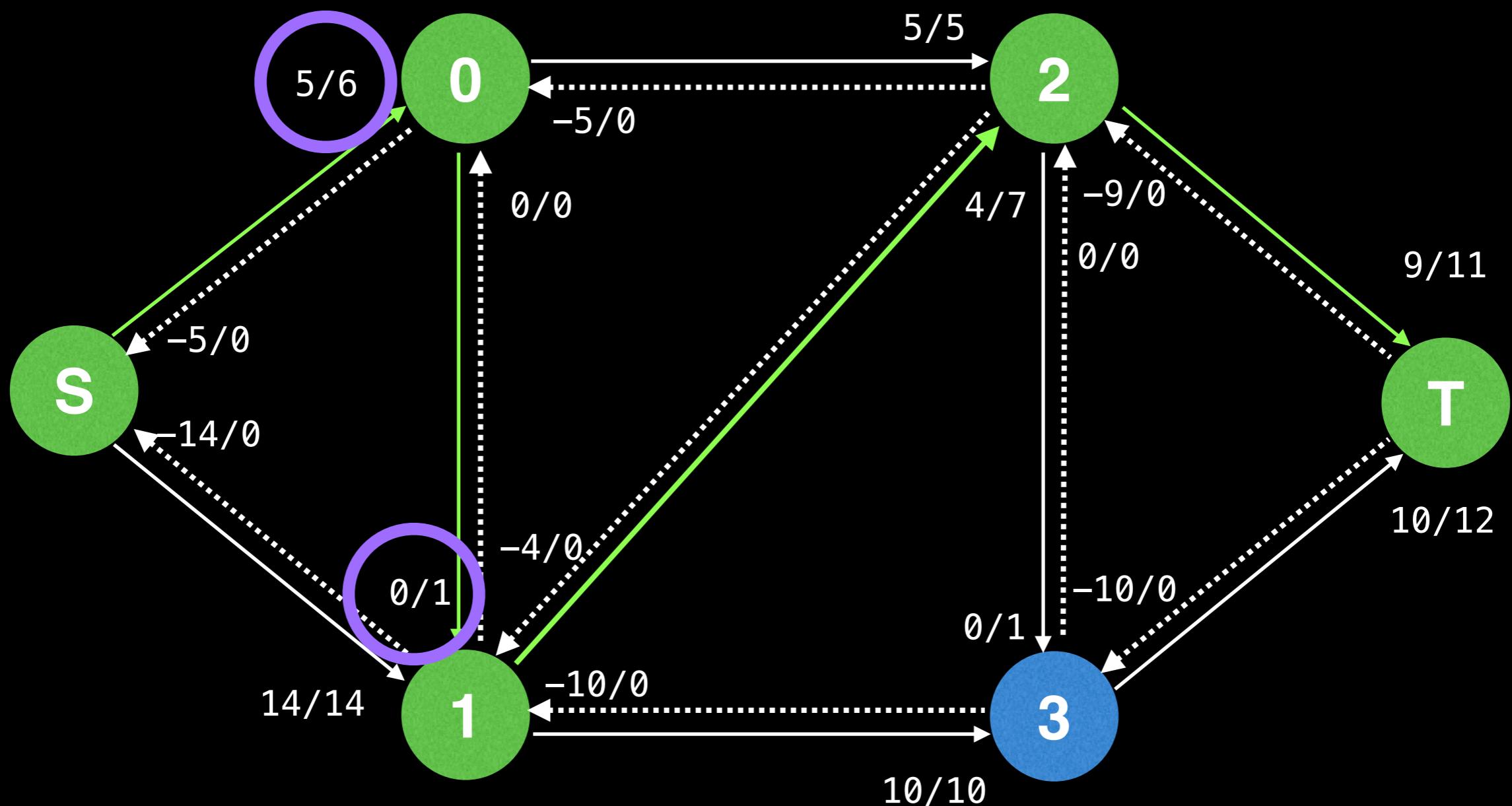


Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 1$ .



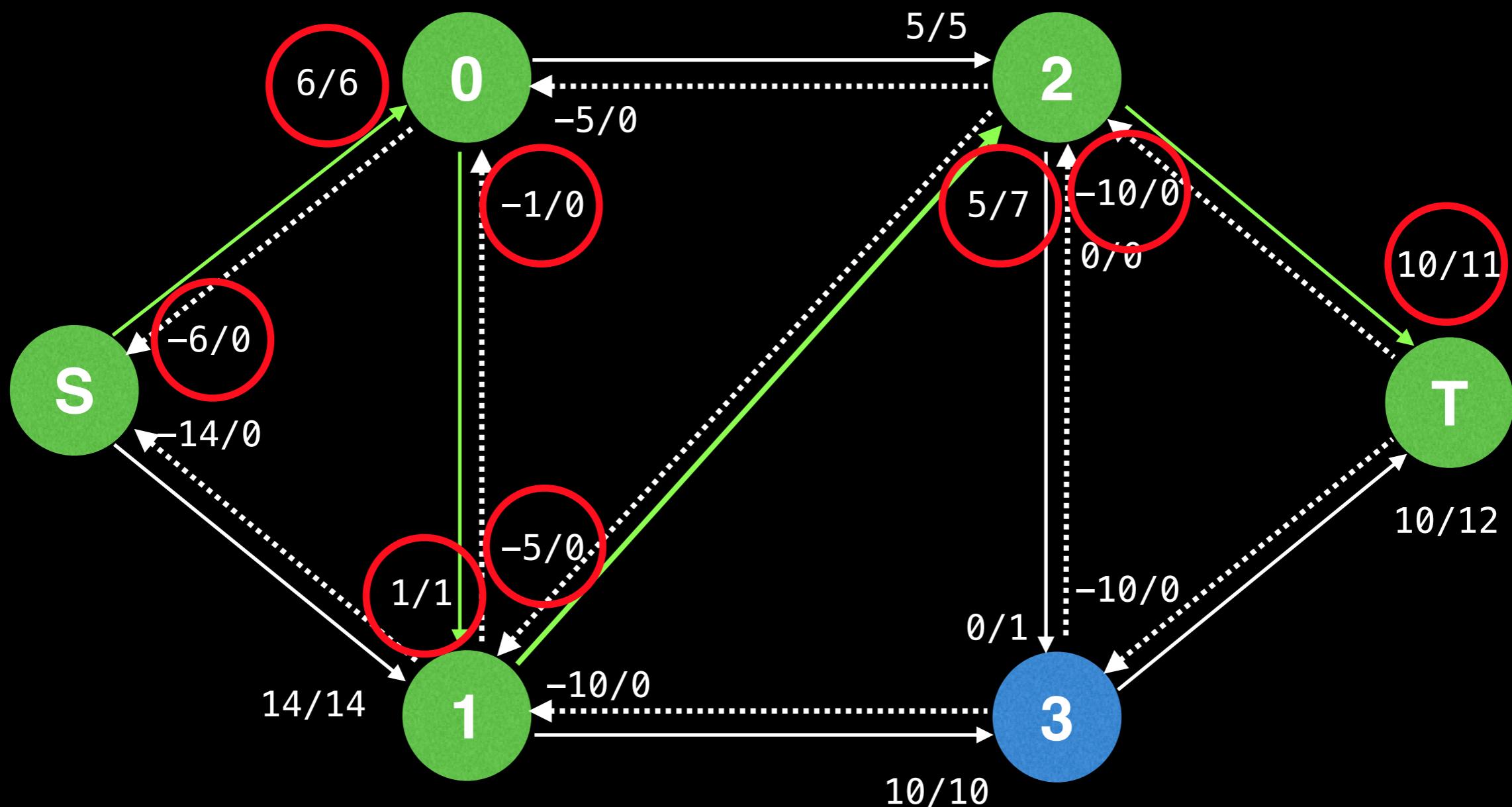
Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 1$ .

The bottleneck value is 1, because  $\min(6-5, 1-0, 7-4, 11-9) = 1$  is the smallest remaining capacity along the path.

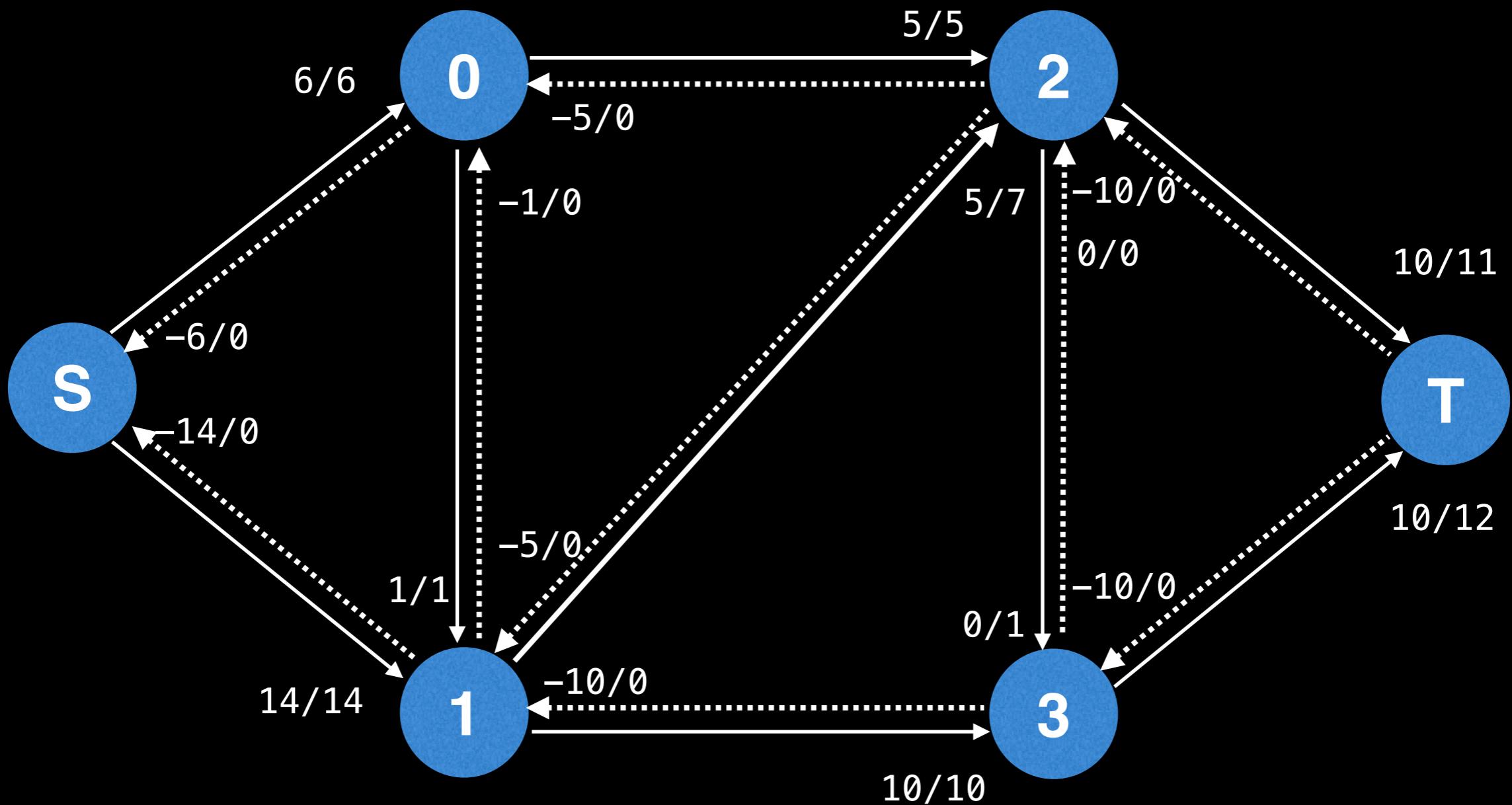


Let's find an augmenting path which has all edges with a remaining capacity greater than or equal to  $\Delta = 1$ .

The bottleneck value is 1, because  $\min(6-5, 1-0, 7-4, 11-9) = 1$  is the smallest remaining capacity along the path.

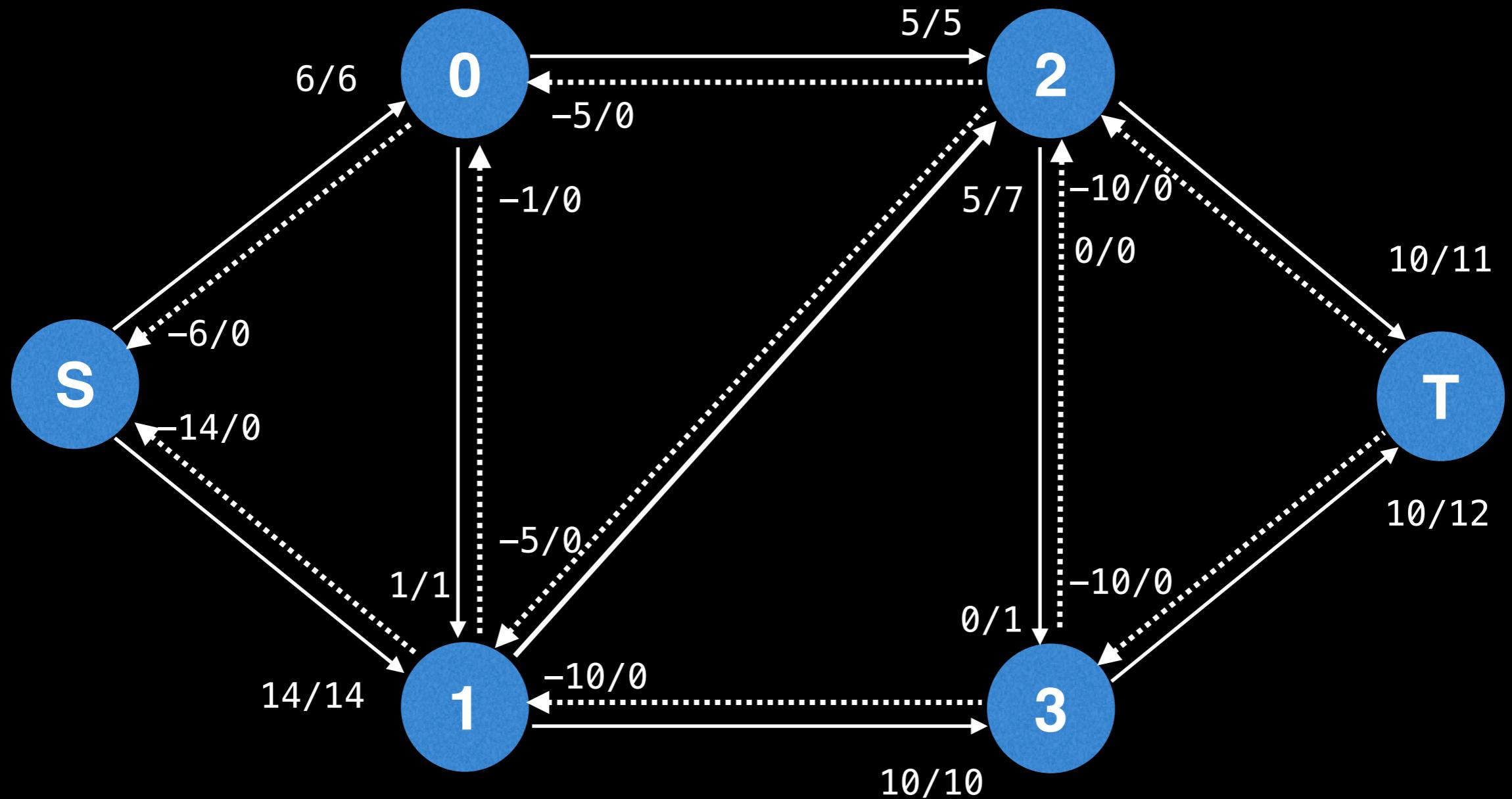


There are no more augmenting paths from  $s \rightarrow t$  which have a remaining capacity greater than or equal to  $\Delta (=1)$ , so the new  $\Delta$  is:  $\Delta = \Delta / 2 = 0$ , which terminates the algorithm.



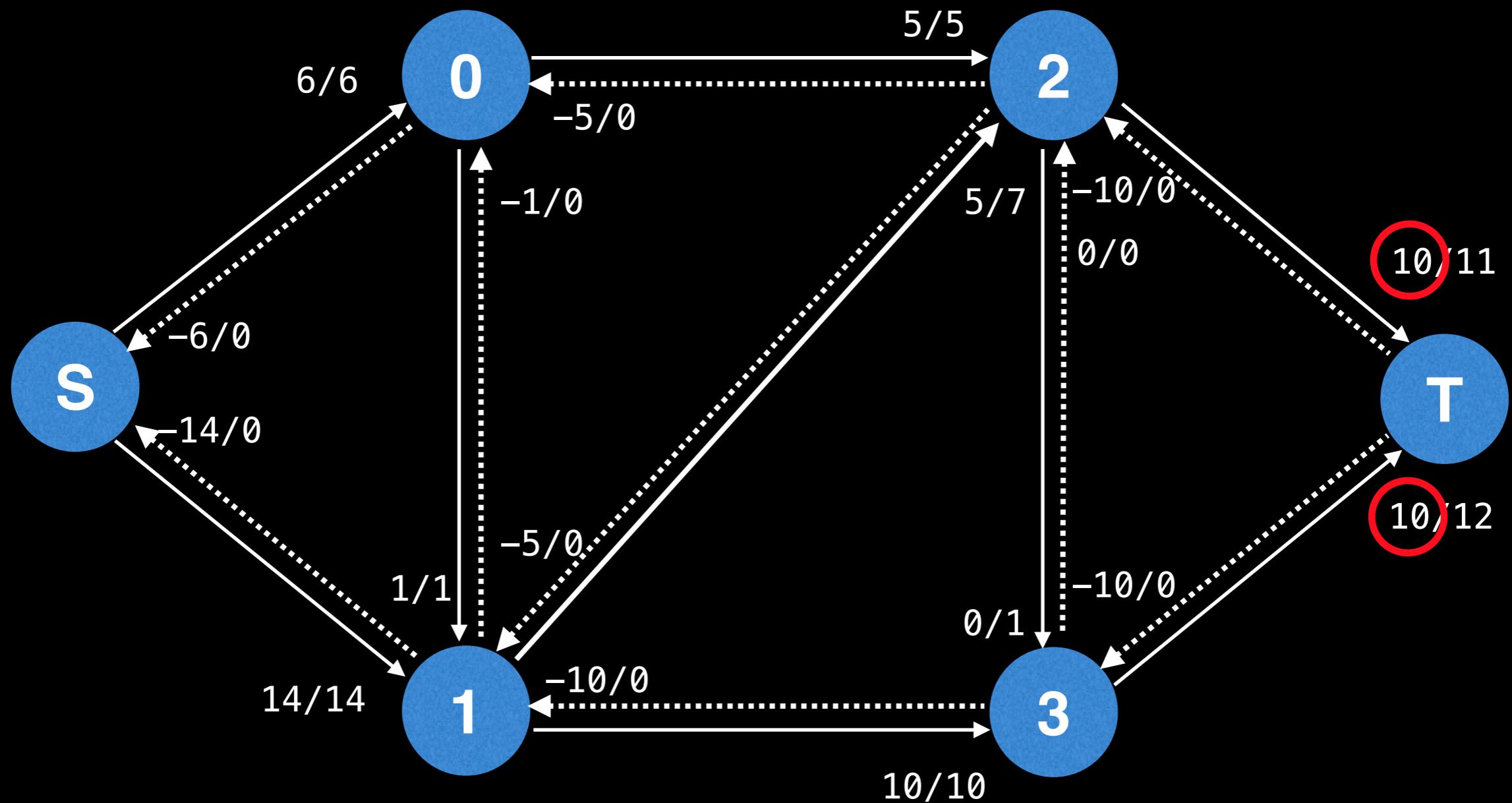
The **max flow** value is the sum of the bottleneck values found during each iteration:

$$\text{Max flow} = 10 + 5 + 4 + 1 = \mathbf{20}$$



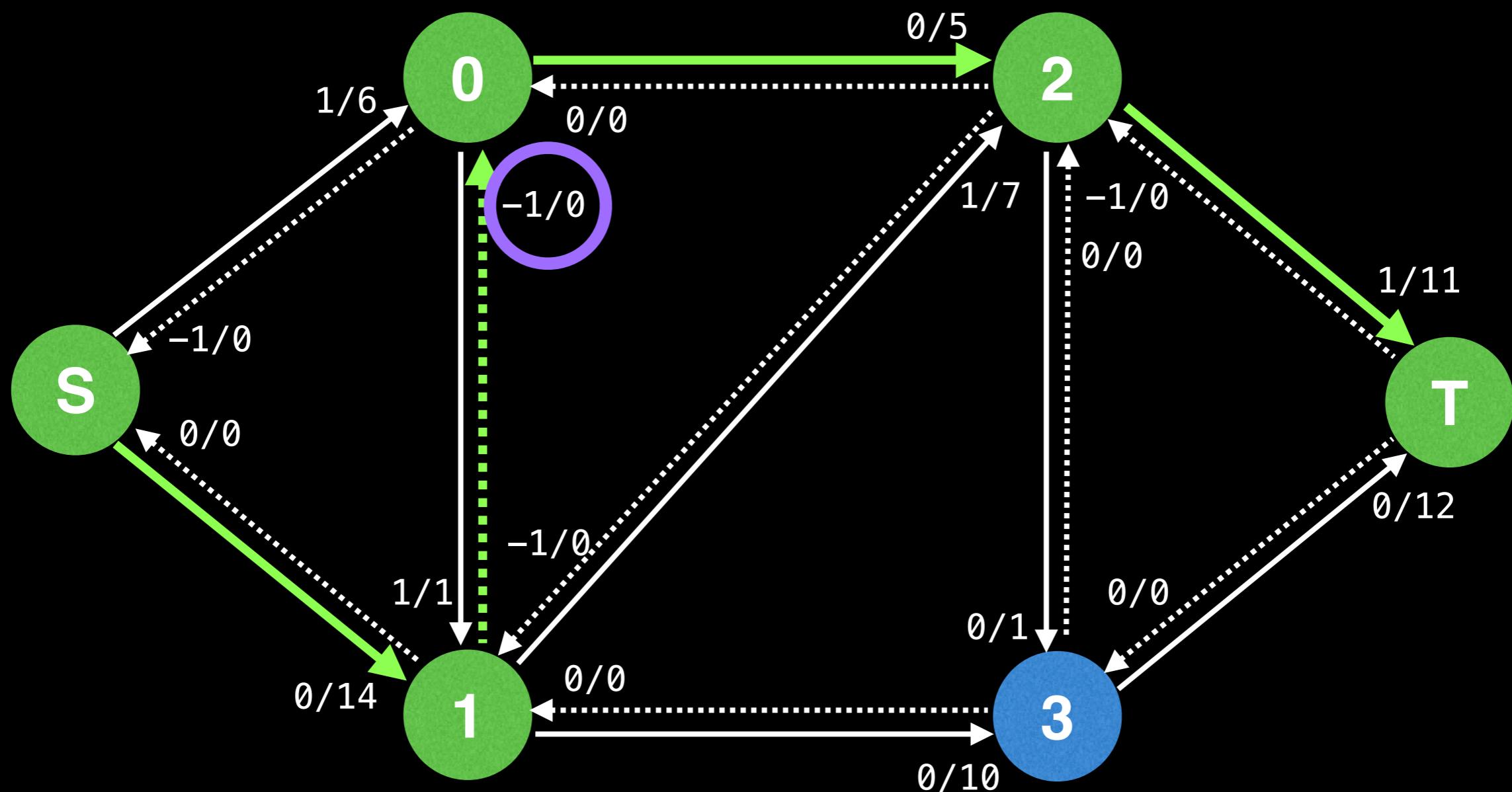
You can also get the **max flow** by summing the flow values going into the sink

$$\text{Max flow} = 10 + 10 = 20$$



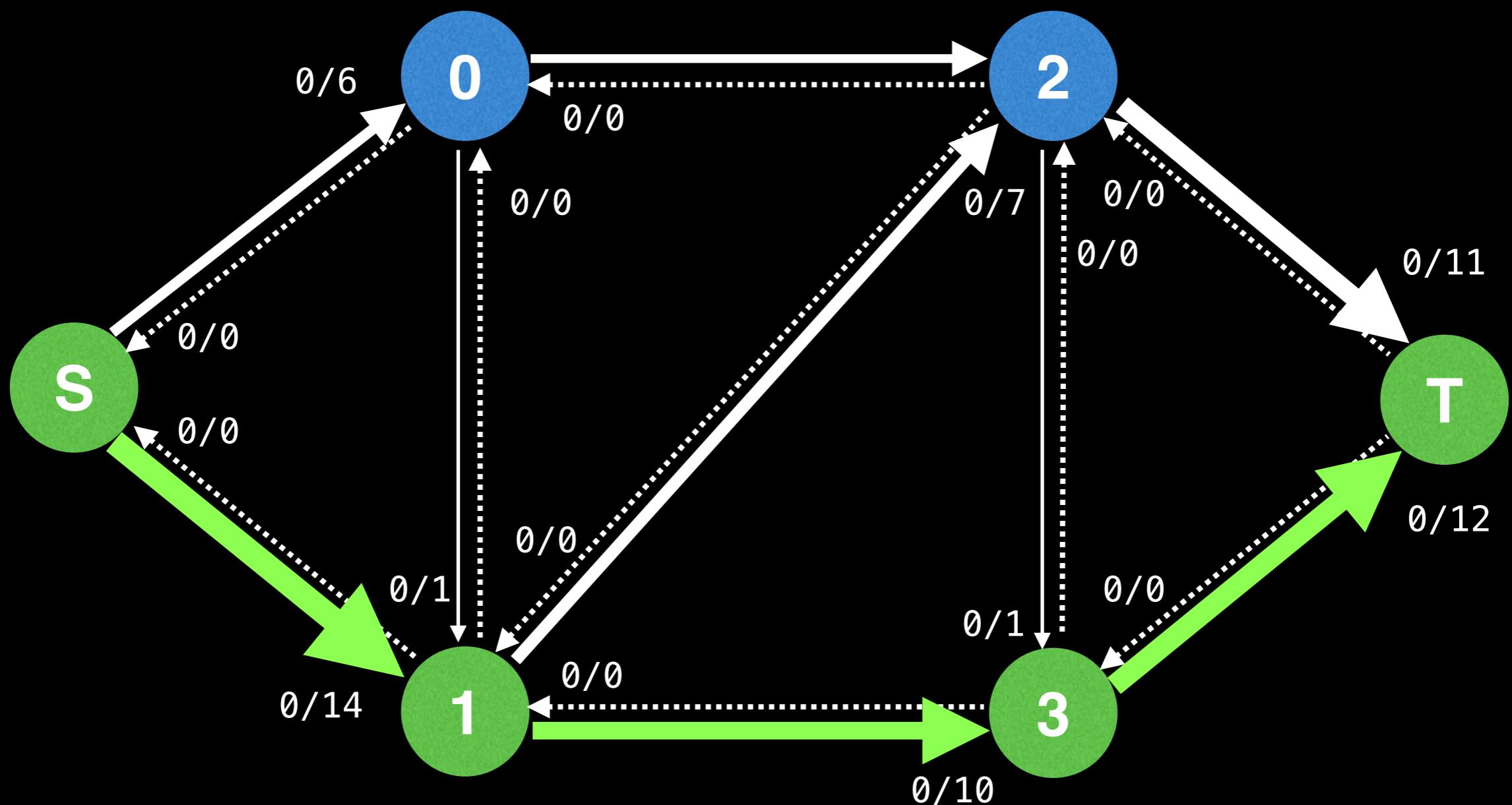
# Summary

Ford–Fulkerson implemented with a DFS can result in having a bottleneck value of 1 every iteration for a complexity of **0(fE)**.



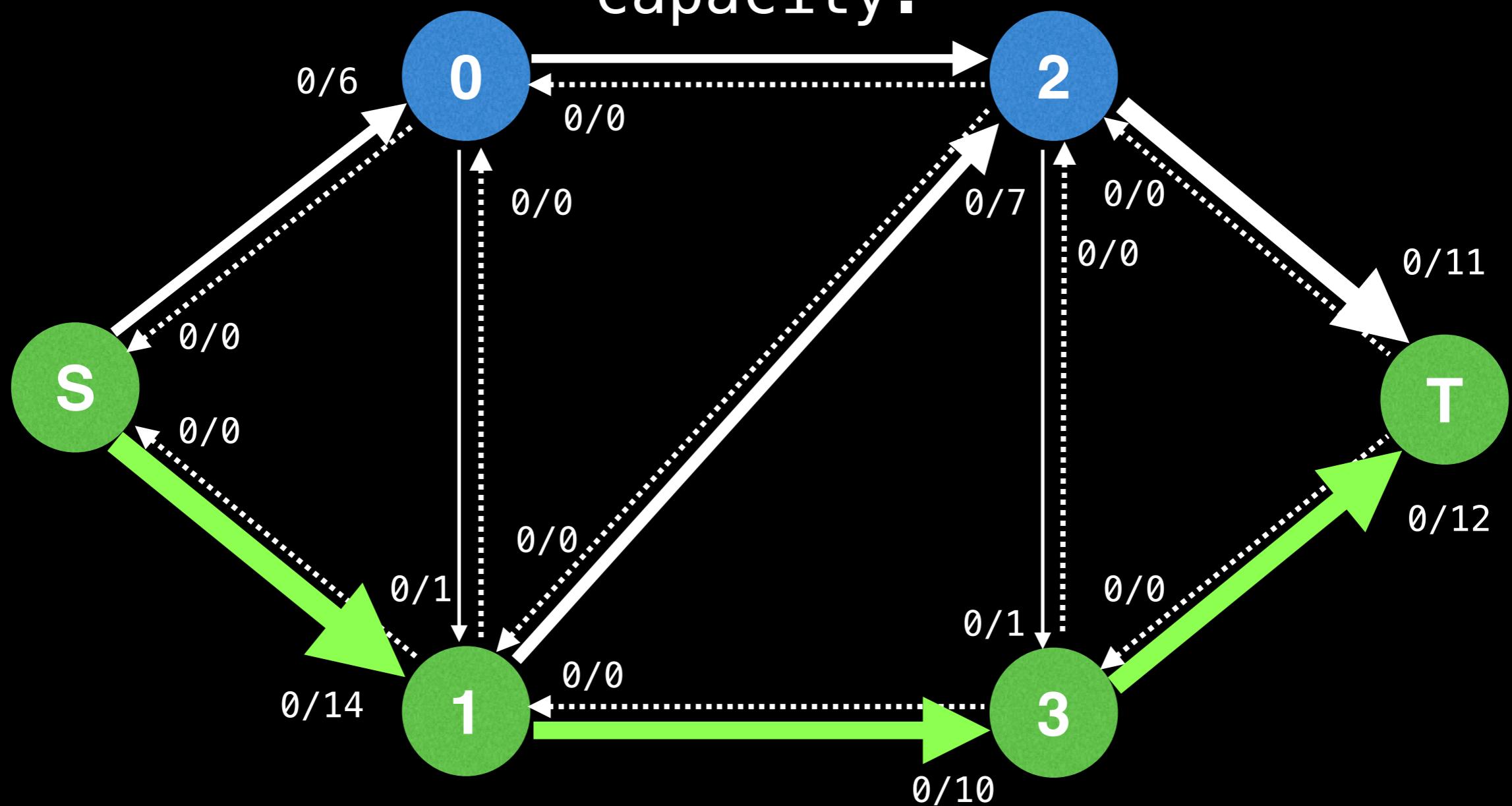
# Summary

Capacity scaling is when we push flow through the larger edges first to achieve a better runtime.



# Summary

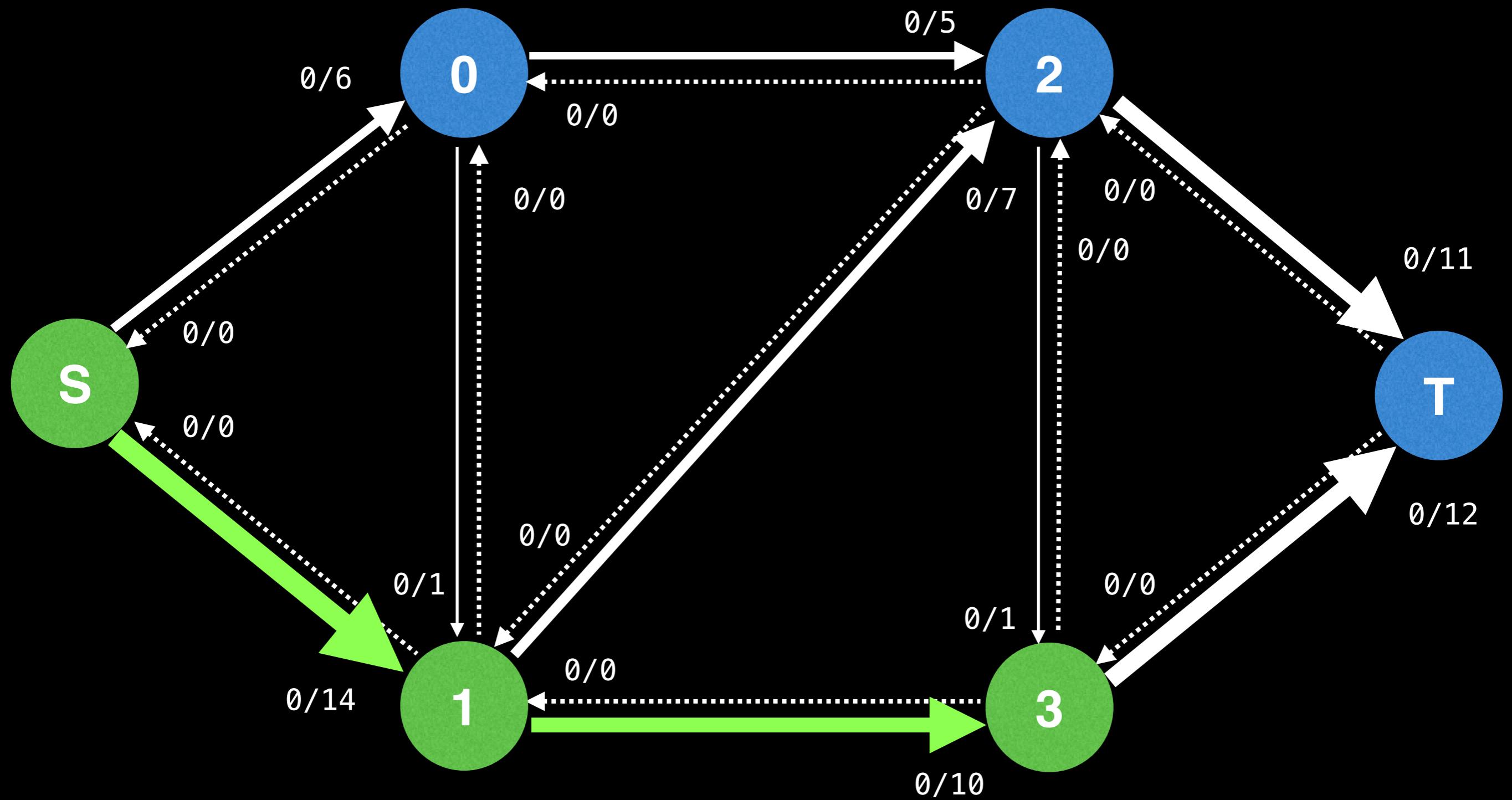
One approach to capacity scaling is to maintain a decreasing parameter  $\Delta$  (i.e by halving) which acts as a threshold for which edges should be accepted or rejected based on their remaining capacity.



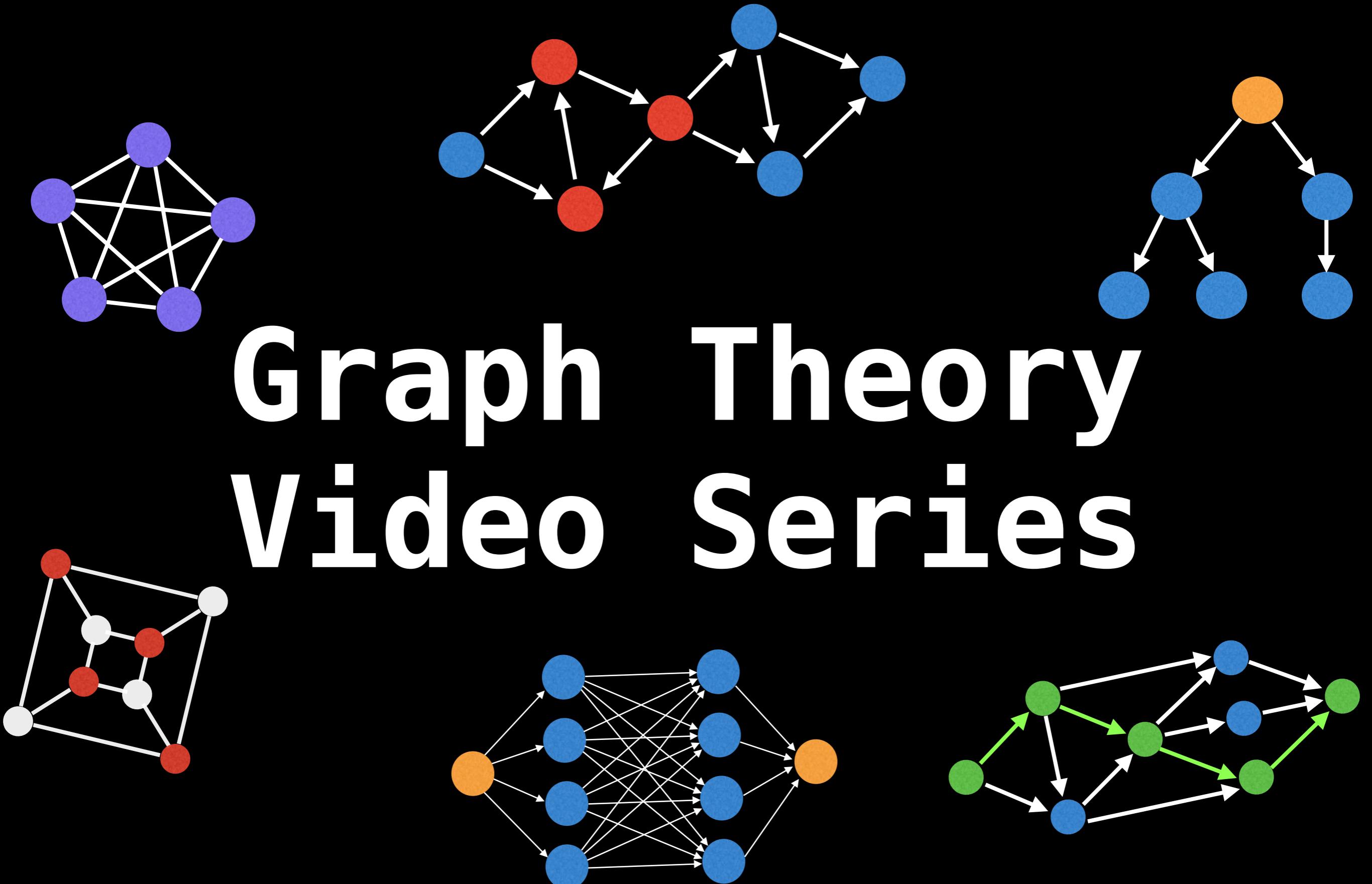
# Next Video: Capacity Scaling Source Code



# Network Flow: Capacity Scaling



# Graph Theory Video Series



# Network Flow Capacity Scaling source code

William Fiset

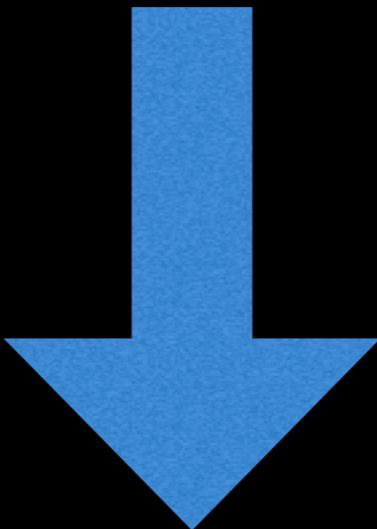
Previous video explaining  
capacity scaling:

# Source Code Link

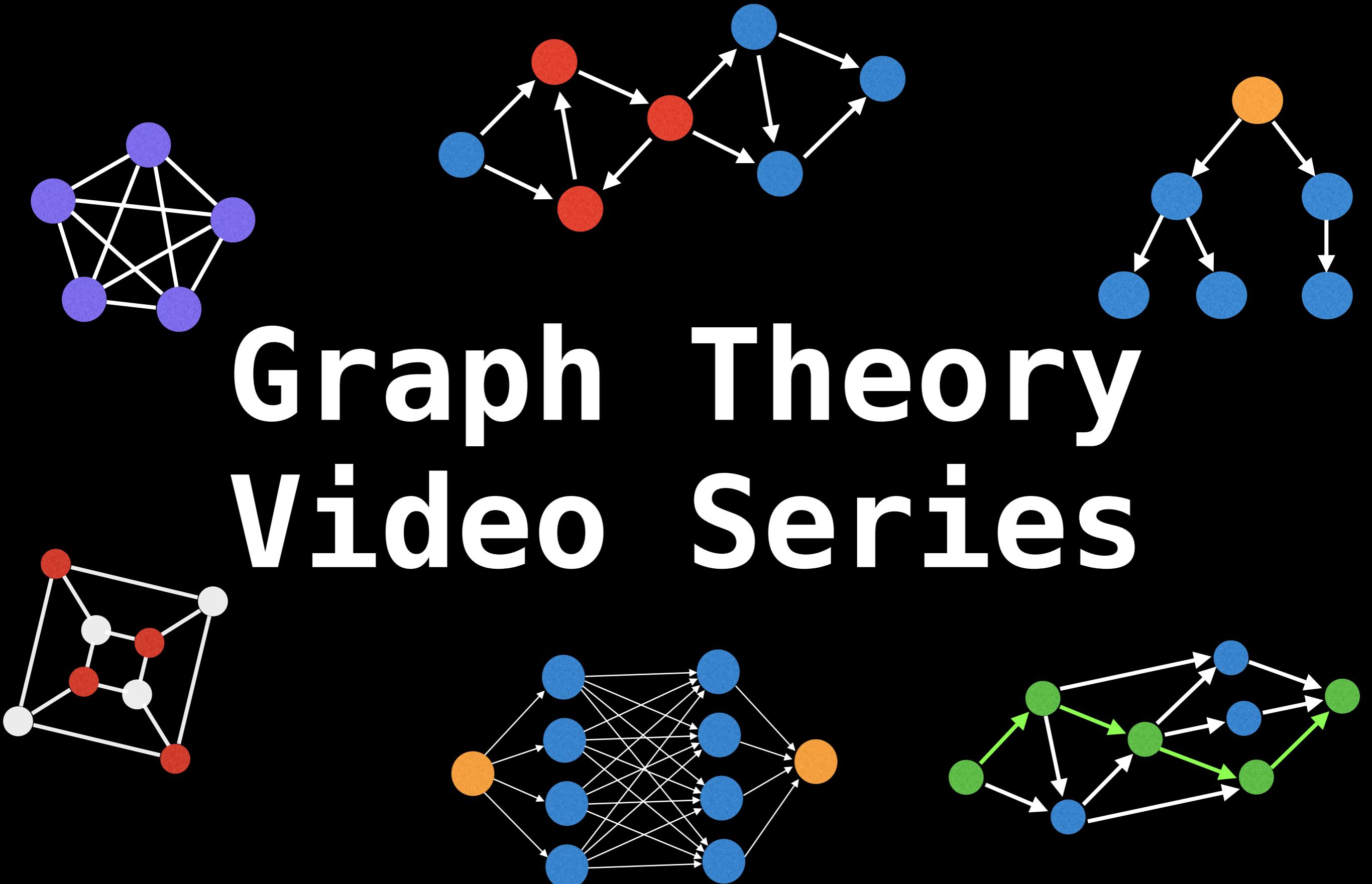
Implementation source code can  
be found at the following link:

[github.com/williamfiset/algorithms](https://github.com/williamfiset/algorithms)

Link in the description:



# Graph Theory Video Series



# Network Flow Dinic's Algorithm

William Fiset

# Dinic's Algorithm

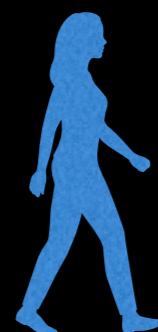
Dinic's is a strongly polynomial maximum flow algorithm with a runtime of  $O(V^2E)$ .

It is extremely fast in practice and works even better on **bipartite graphs** giving a time complexity of  $O(\sqrt{VE})$  due to the algorithm's reduction to **Hopcroft-Karp**.

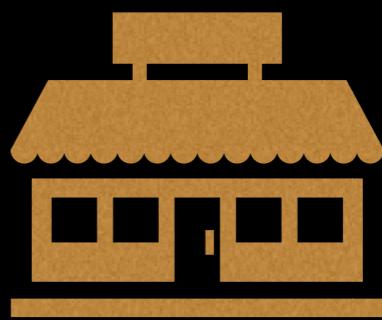
The algorithm was originally invented by Yefim Dinitz in 1969 and published in 1970. The algorithm was later modified slightly and popularized by Shimon Even mispronouncing “Dinitz’s algorithm” as “Dinic’s algorithm”.

# Dinic's Intuition Analogy

Suppose you and a friend planned to meet up at a **coffee shop** a **few streets east** of where you are. You have never been to this coffee shop and you don't exactly know where it is, but you know it's somewhere east, how would you get there?



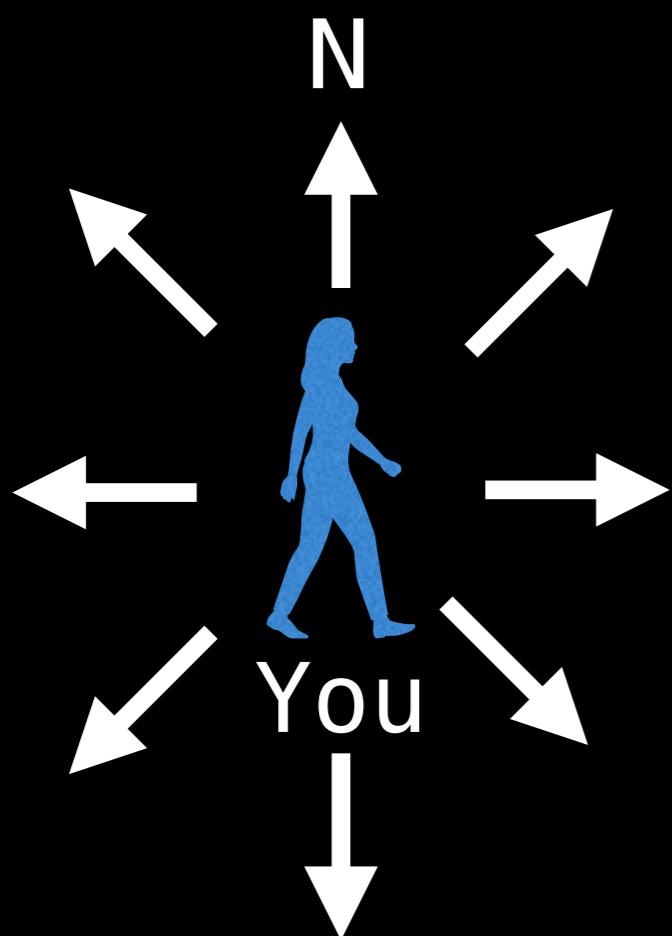
You



Coffee shop

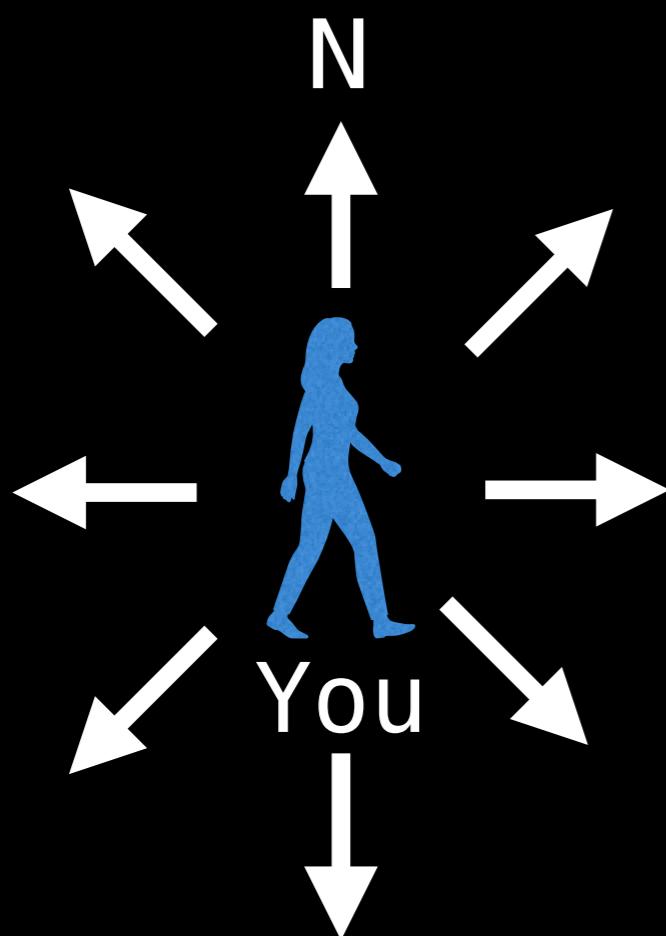
# Dinic's Intuition Analogy

Q: With the information you have, would it make sense to head south? What about north west?



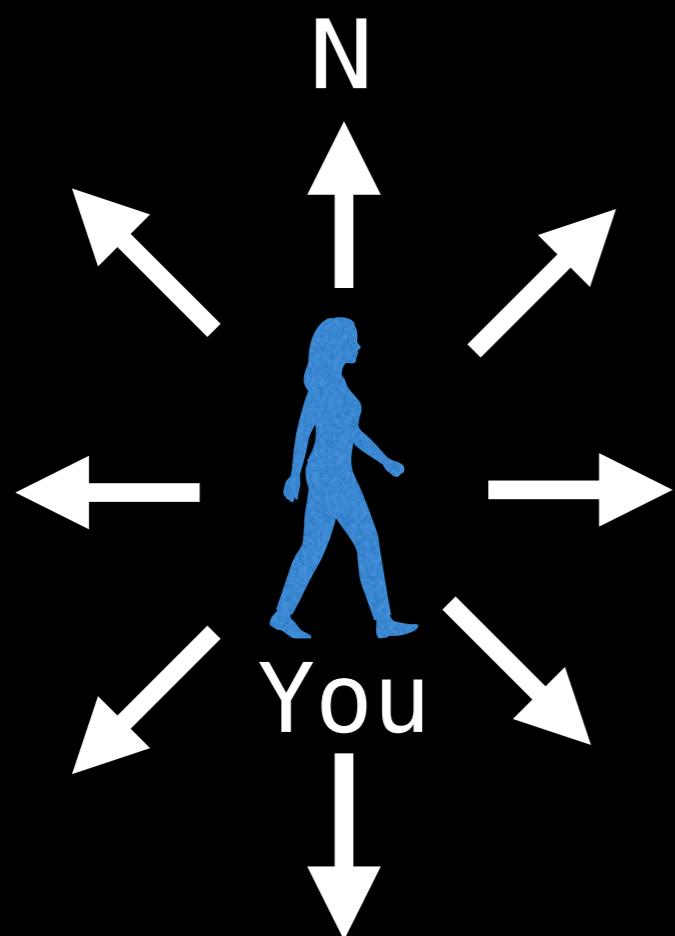
# Dinic's Intuition Analogy

The only sensible directions are: east, north east and south east, this is because you know that those directions guarantee **positive progress towards** the coffee shop.



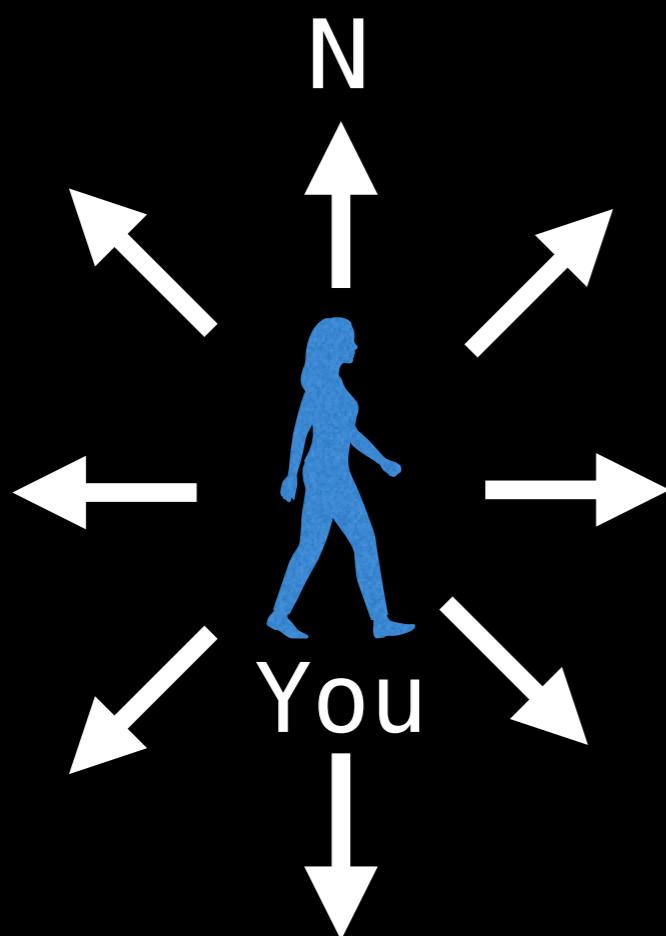
# Dinic's Intuition Analogy

This form of **heuristic** ensures that we **continuously make progress** towards whatever place of interest we desire to go, how can we apply this to solving maximum flow?

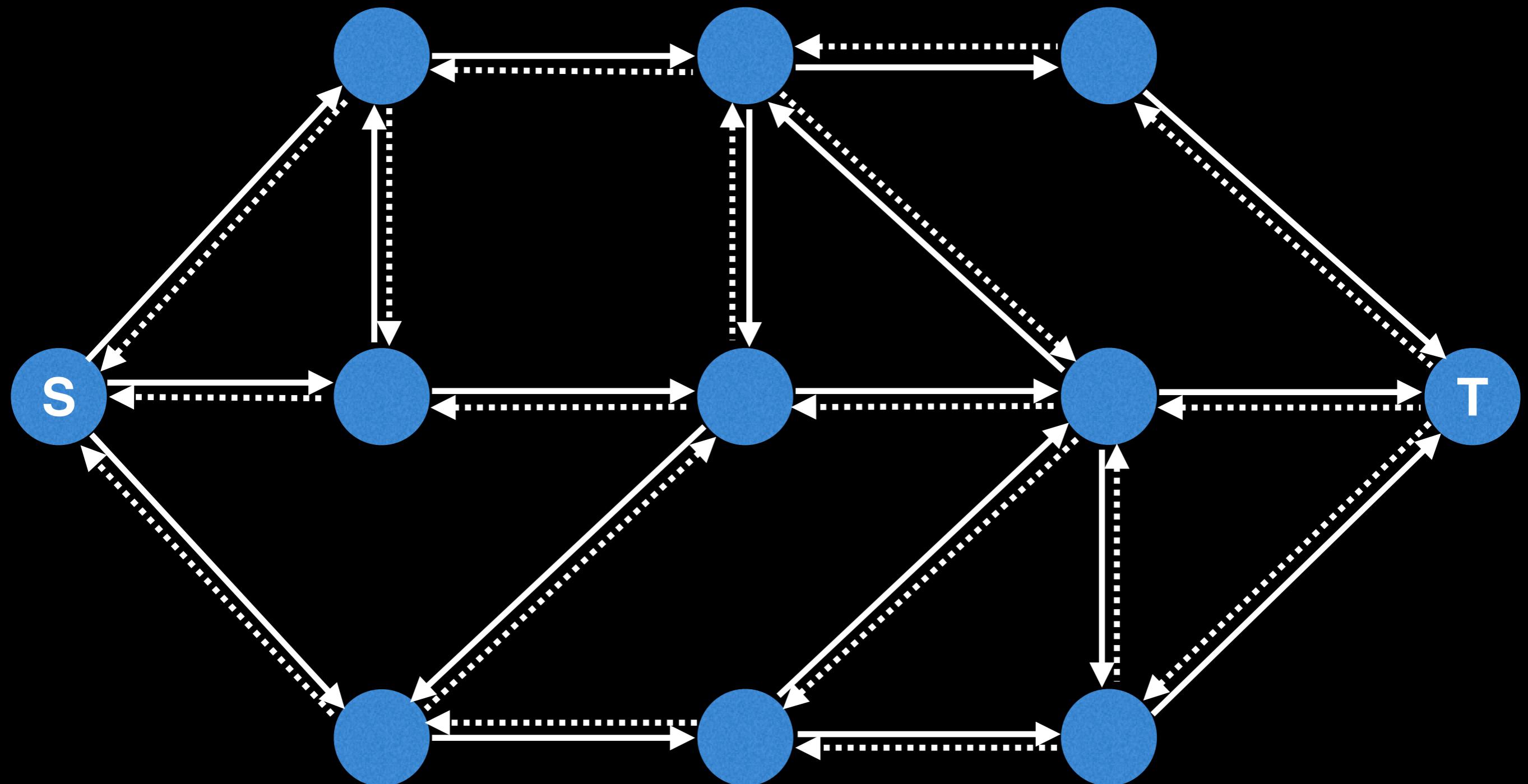


# Dinic's Intuition Analogy

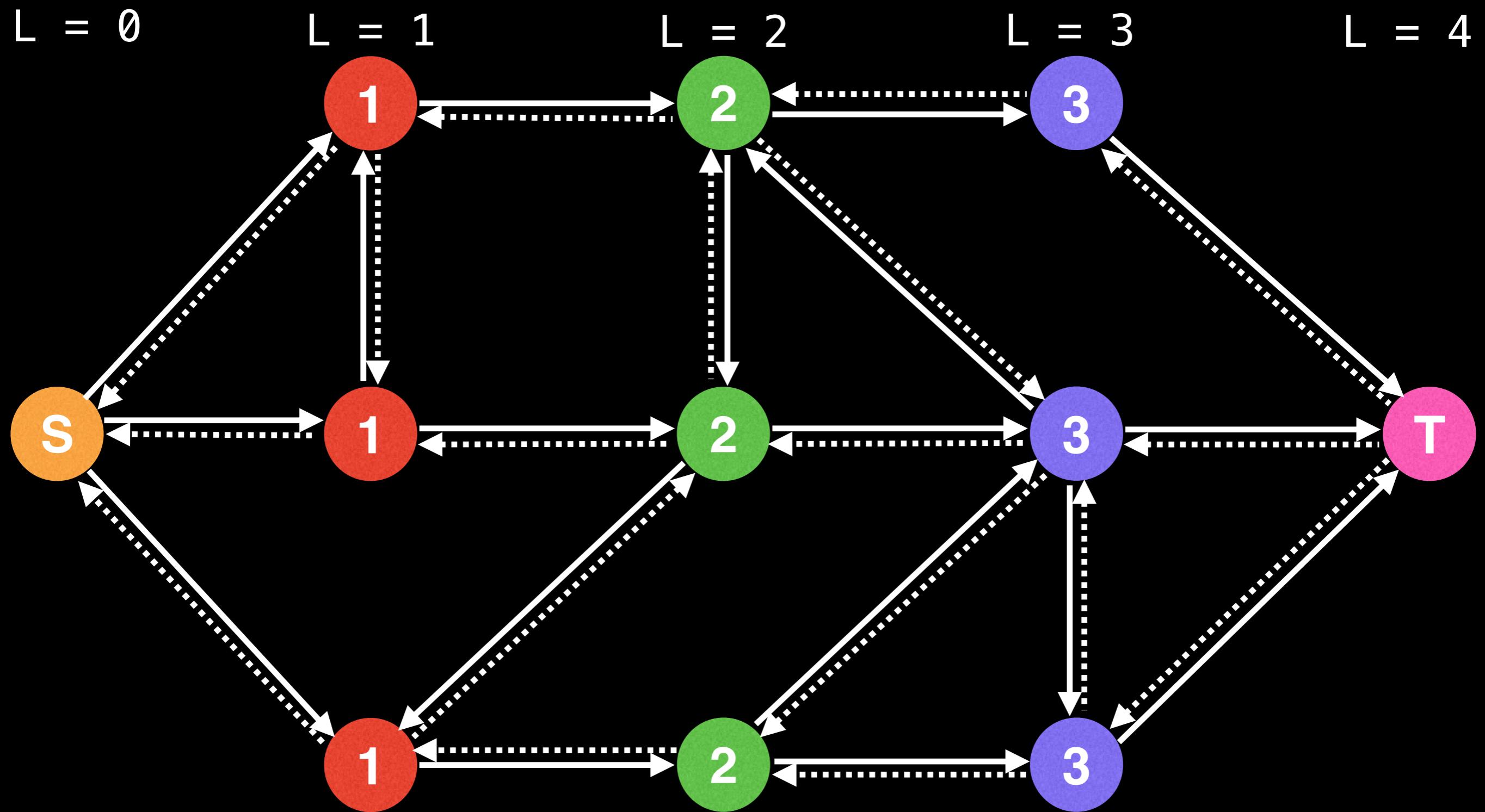
In this analogy, **you are the source** and the **coffee shop is the sink**. The main idea behind Dinic's algorithm is to guide augmenting paths from  $s \rightarrow t$  using a **level graph**, and in doing so greatly reducing the runtime.



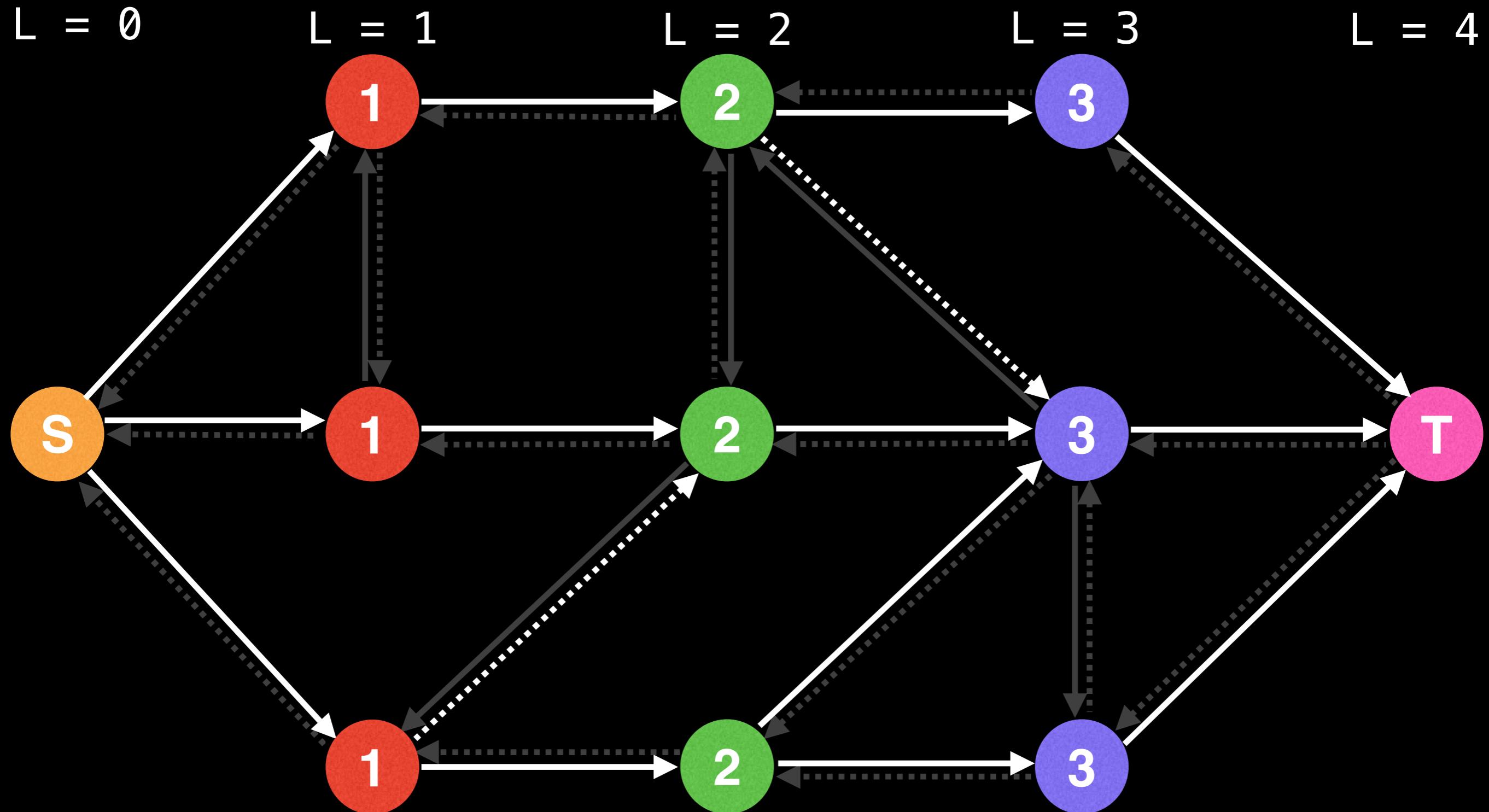
The way Dinic's determines which edges make progress towards the sink T and which do not is by building a **level graph**.



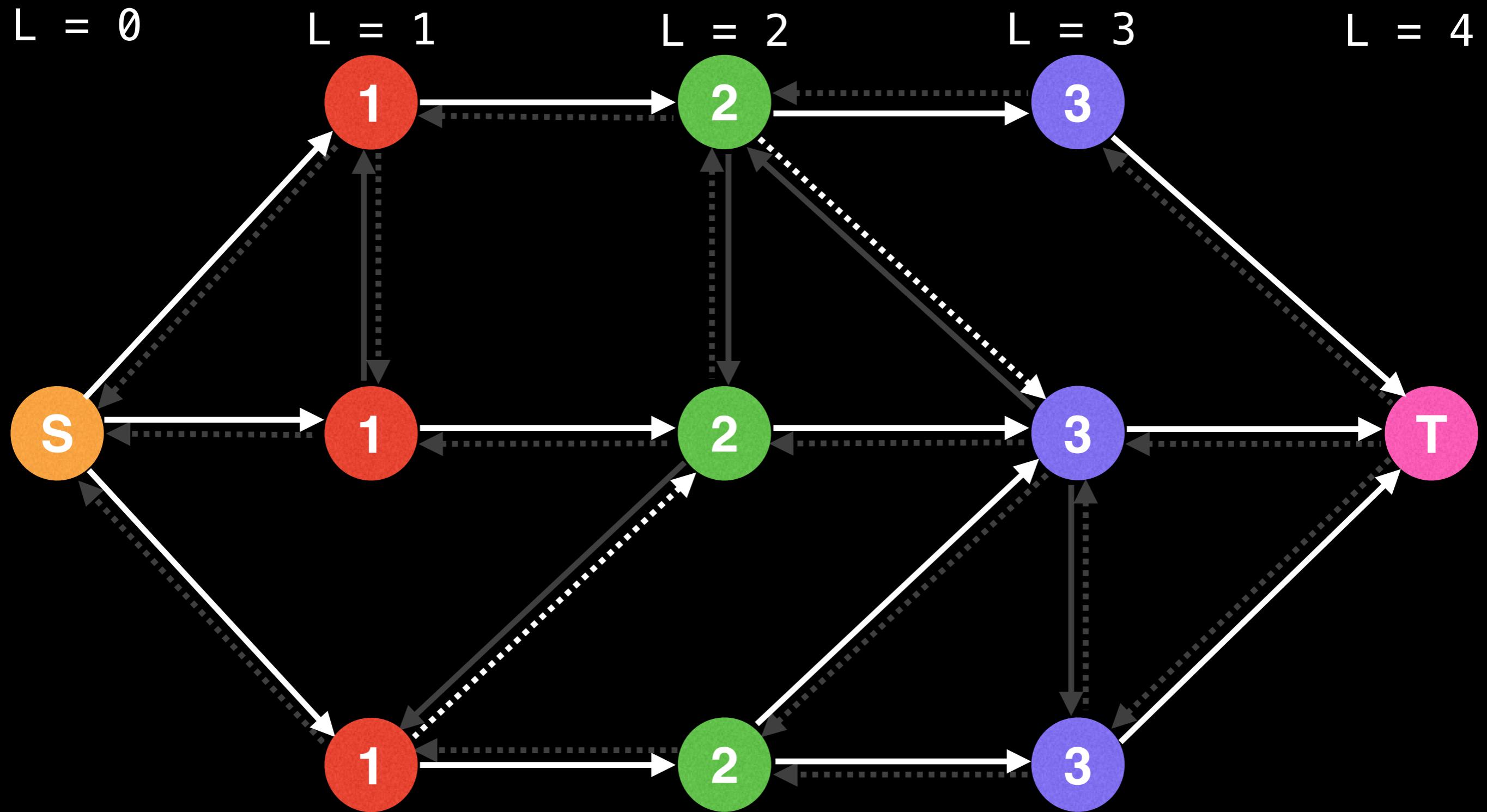
The levels of the graph are those obtained by doing a BFS from the source.



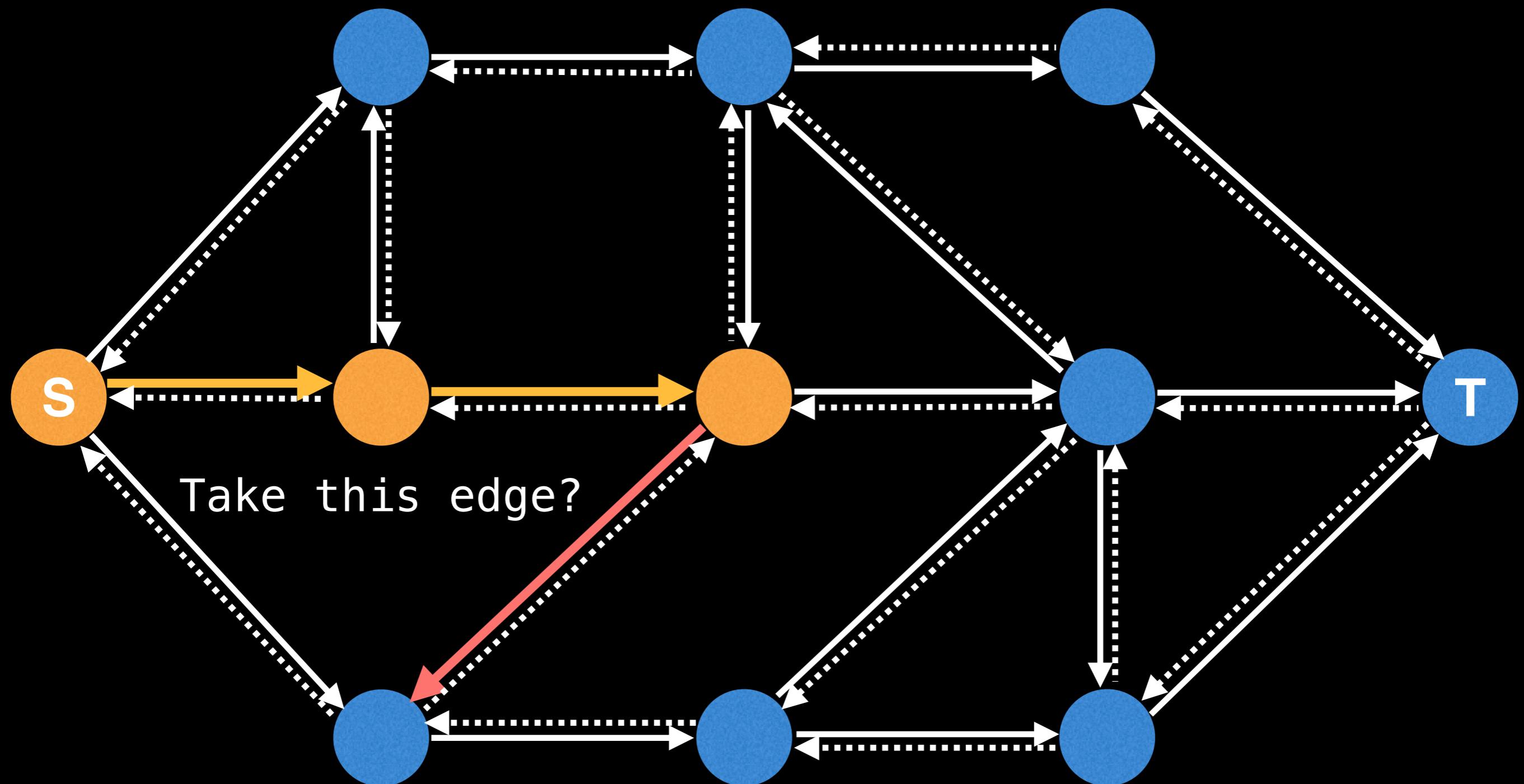
Furthermore, an edge is only part of the level graph if it makes progress towards the sink. That is, the edge must go from a node at level  $L$  to another at level  $L+1$ .



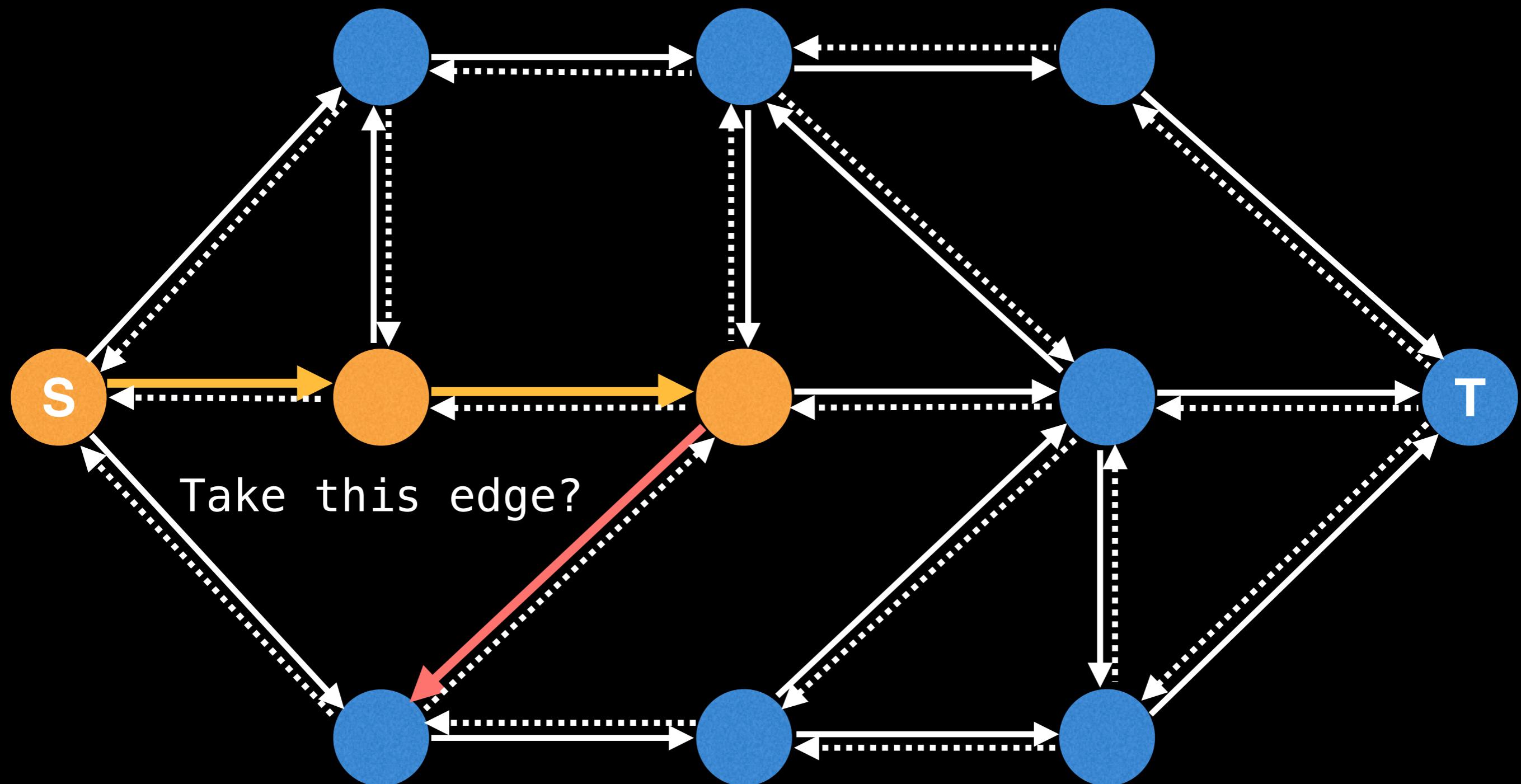
The requirement that edges must go from  $L$  to  $L+1$  prunes backwards and “sideways” edges (grey edges in this slide).



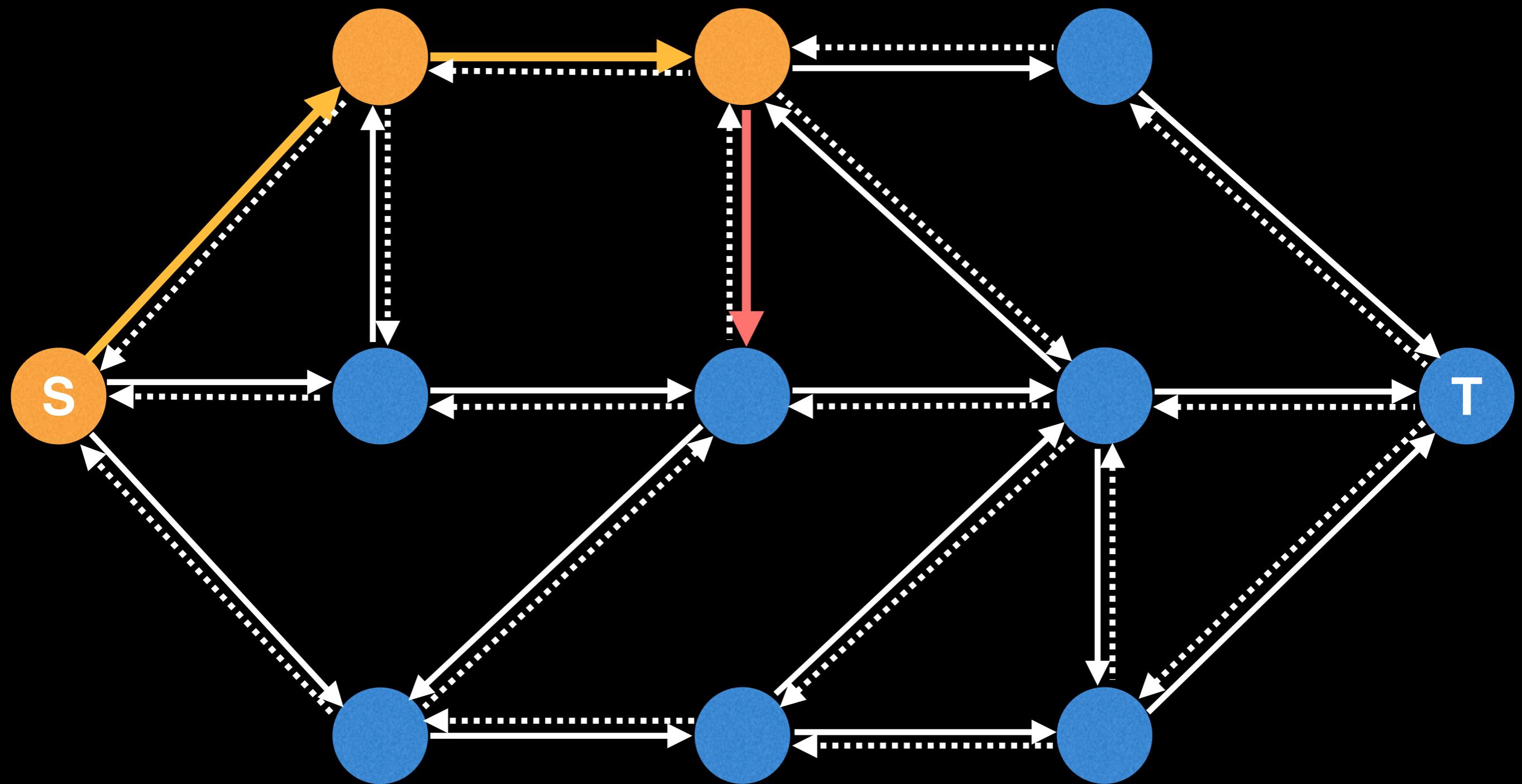
Q: If we're trying to go from  $s \rightarrow t$  as quickly as possible, does it make sense to take the red edge going in the backwards direction?



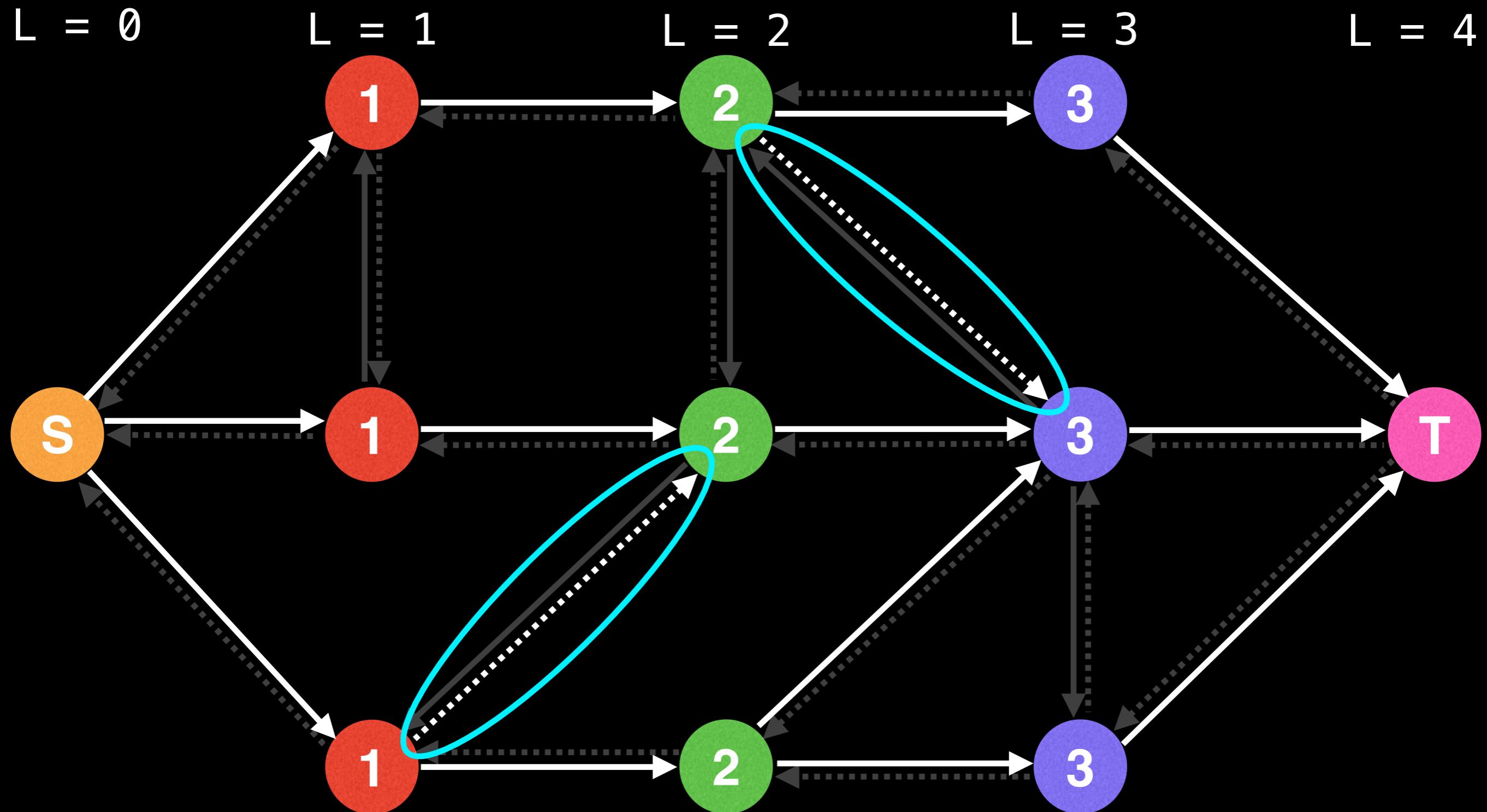
A: No, taking the red edge doesn't bring you closer to the sink, so it should only be taken if a detour is required. This is why backwards edges are omitted in the level graph.



The same thing can be said about edges which cut across sideways to the sink as no progress is made.



It's also worth mentioning that residual (dotted) edges can be made part of the level graph, but they must have a **remaining capacity** (capacity - flow)  $> 0$ .



# Dinic's Algorithm Steps

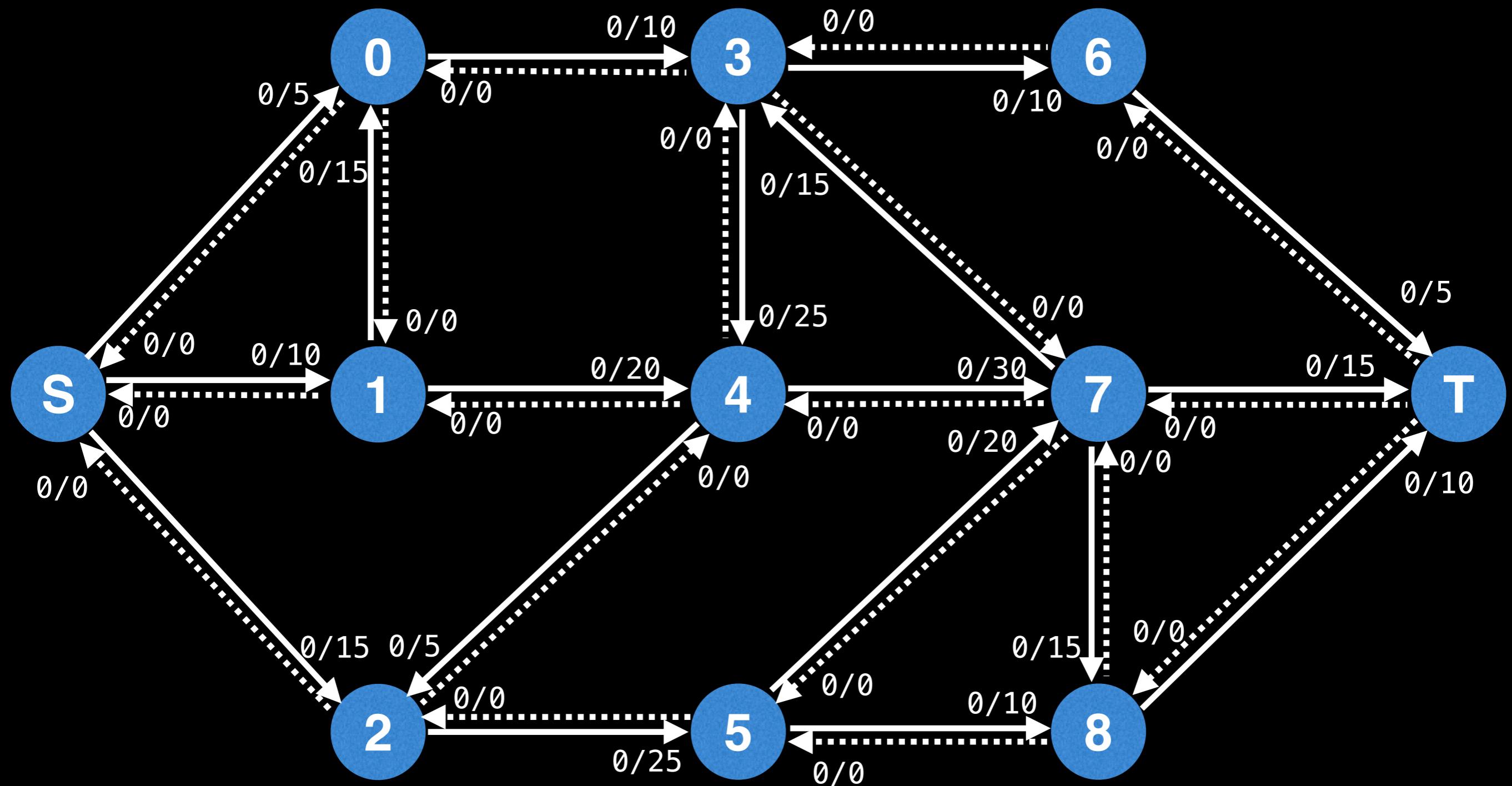
**Step 1:** Construct a level graph by doing a BFS from the source to label all the levels of the current flow graph.

**Step 2:** If the sink was never reached while building the level graph, then stop and return the max flow.

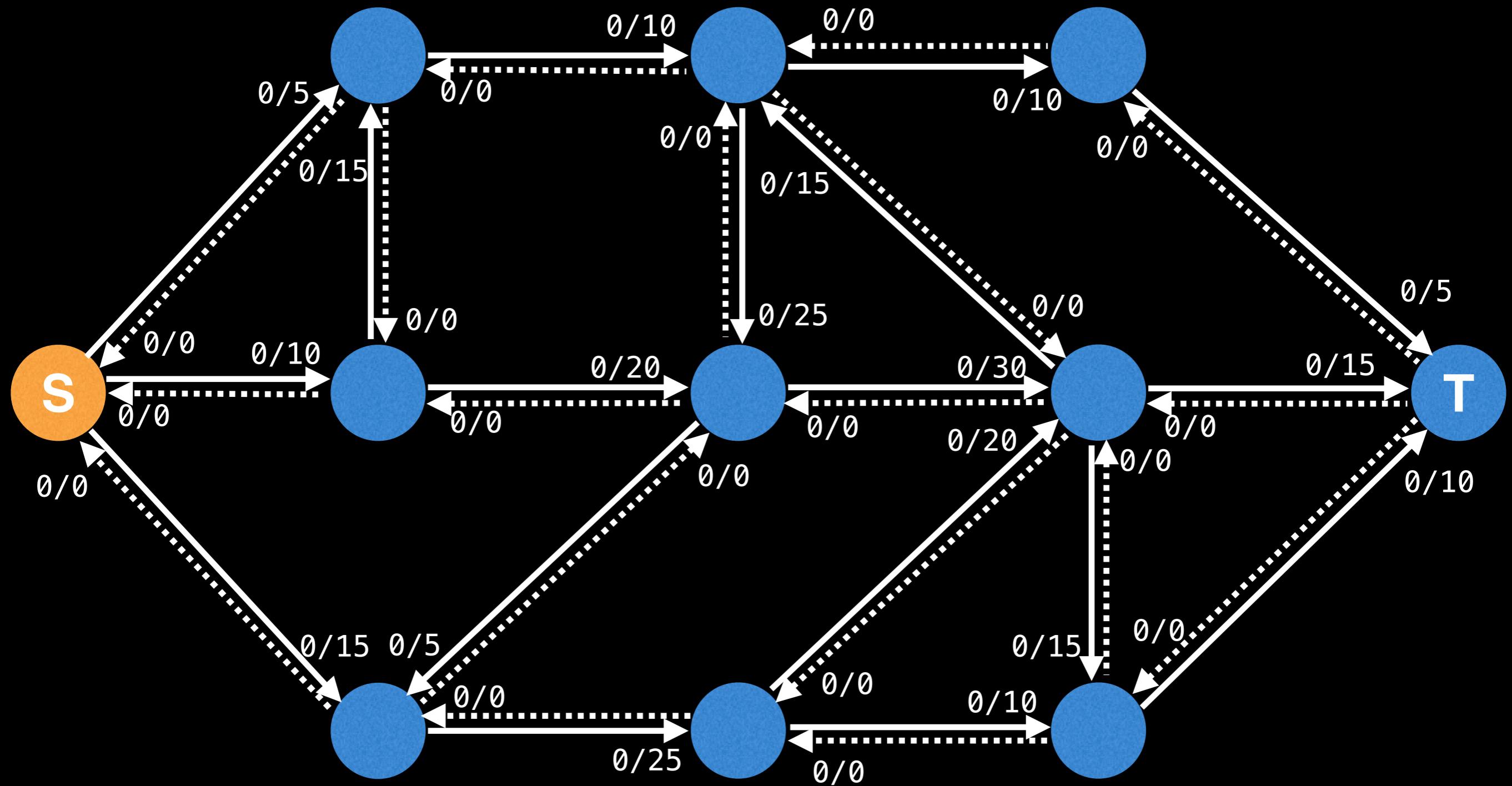
**Step 3:** Using only valid edges in the level graph, do multiple DFSs from  $s \rightarrow t$  until a **blocking flow** is reached, and sum over the bottleneck values of all the augmenting paths found to calculate the max flow.

Repeat Steps 1 to 3

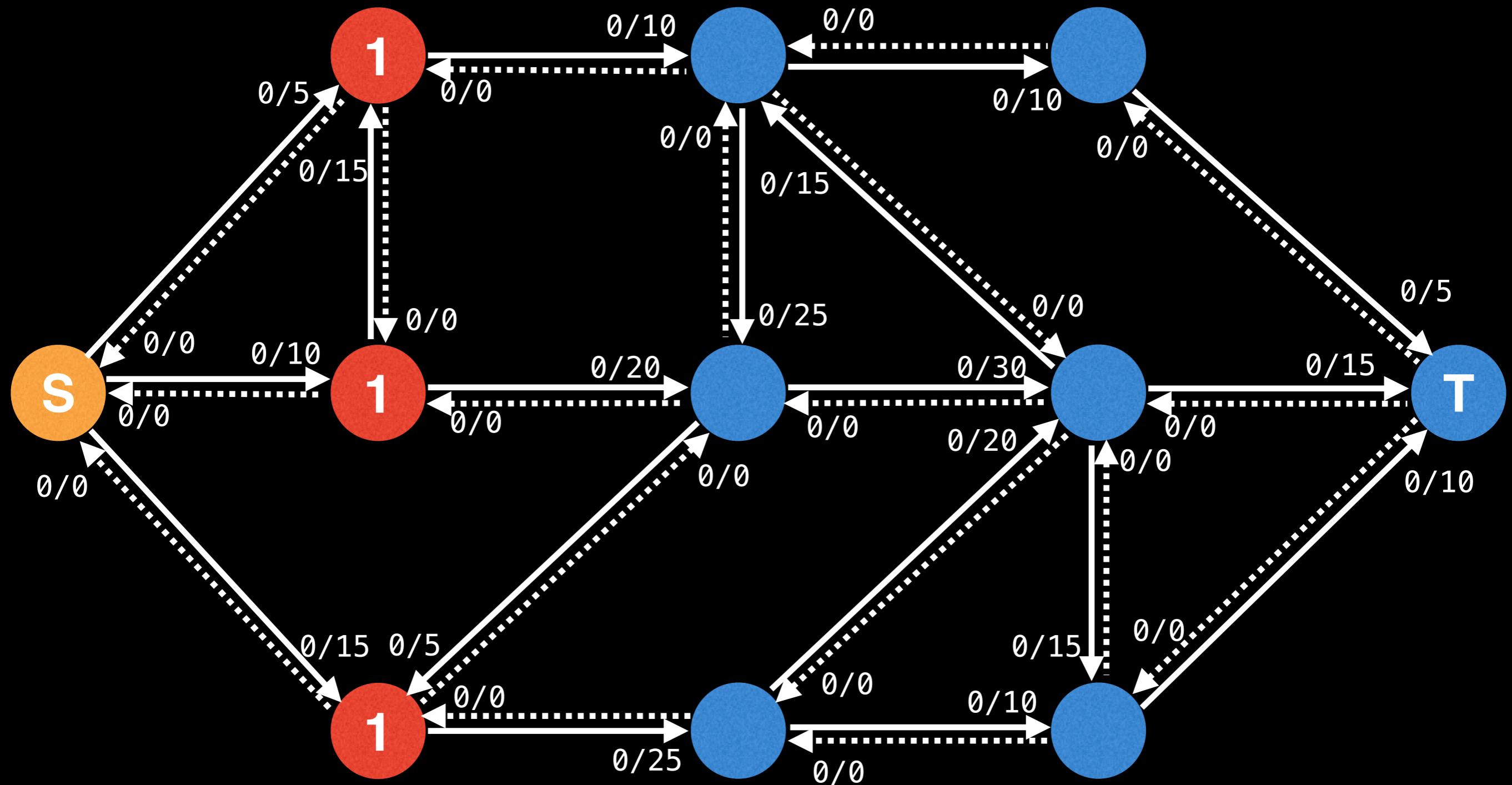
Let's run through an example of using Dinic's algorithm to find the maximum flow on the following flow graph.



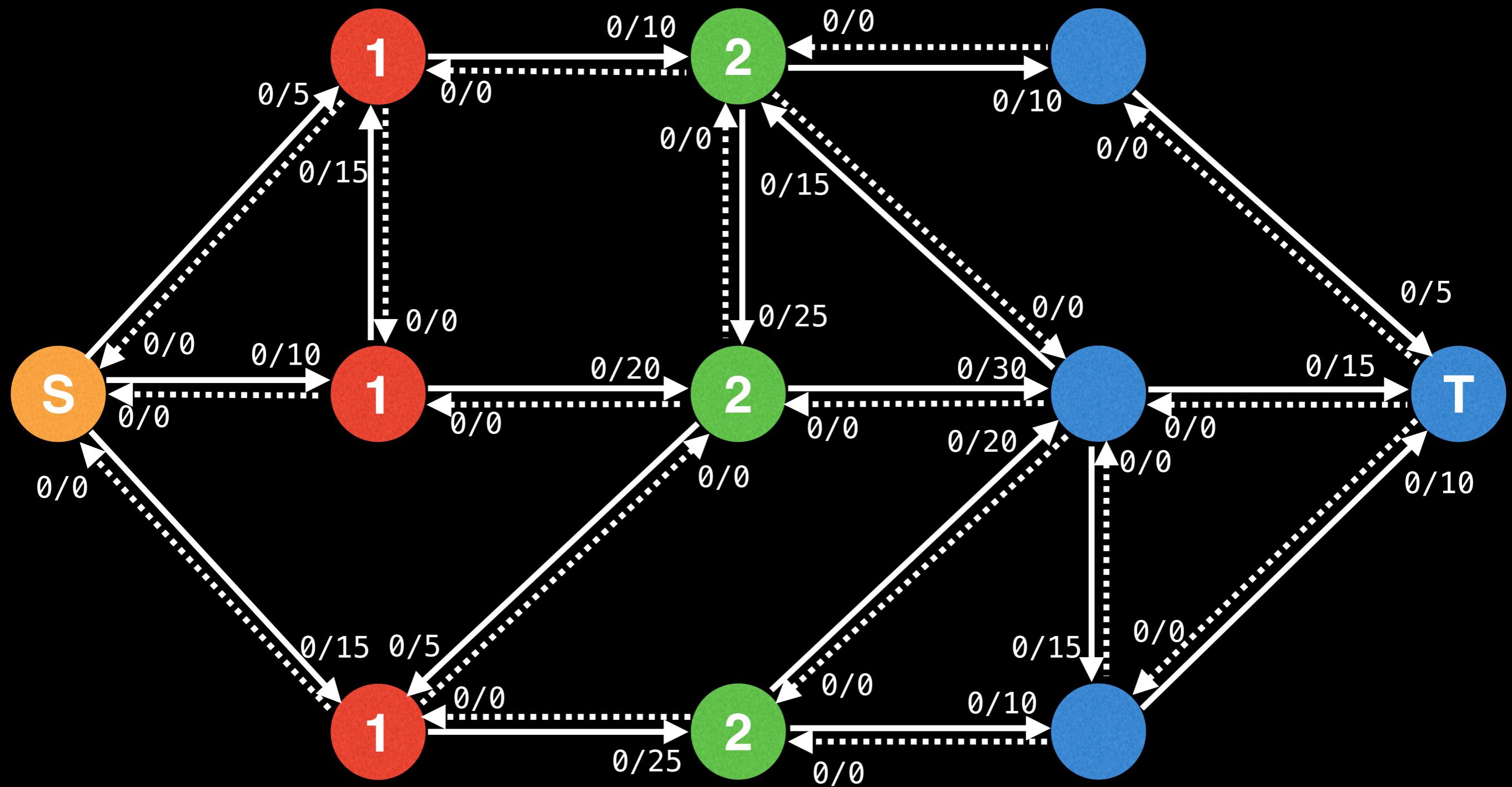
Begin by constructing the level graph for the current flow graph.



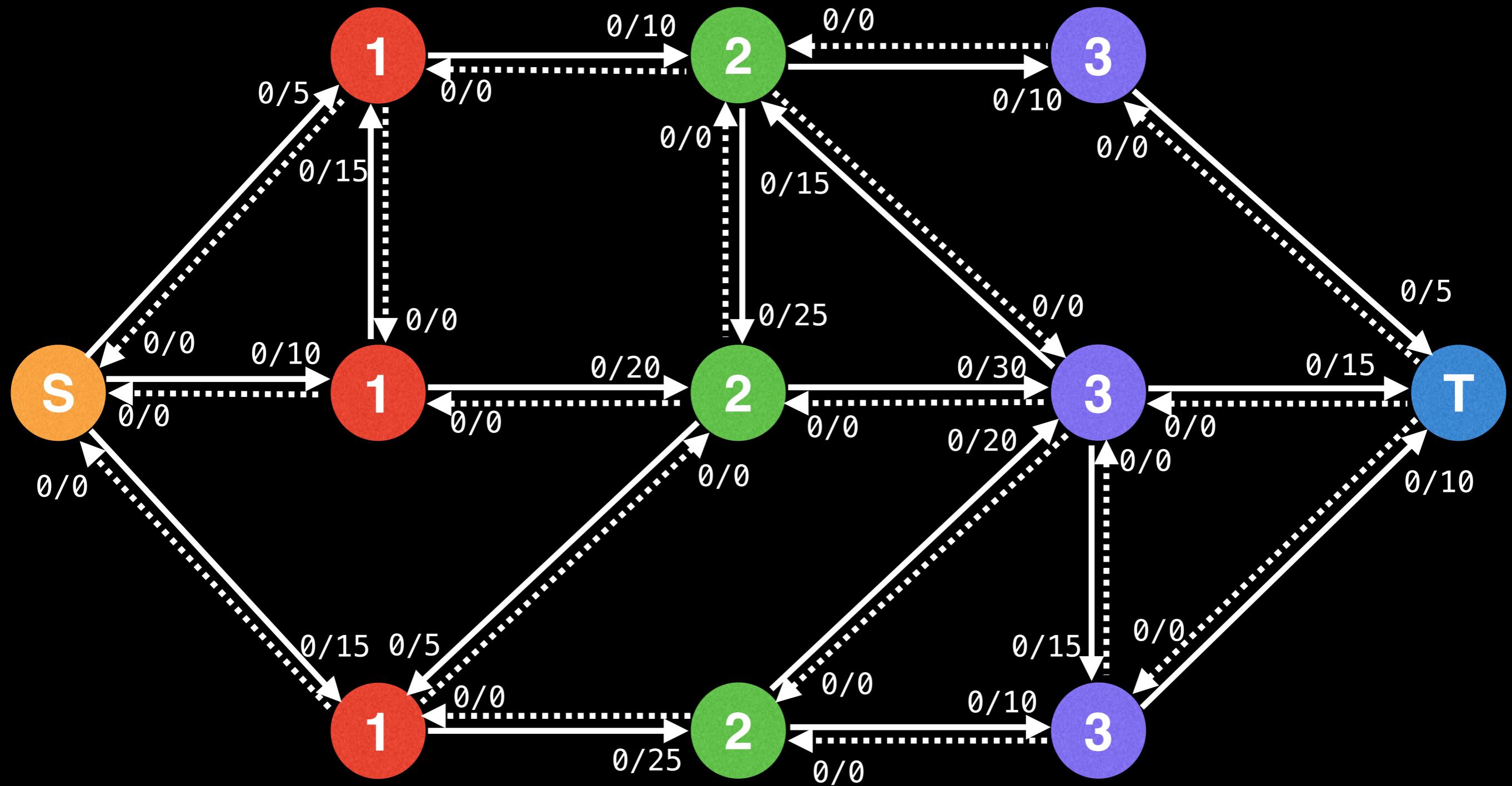
Begin by constructing the level graph for the current flow graph.



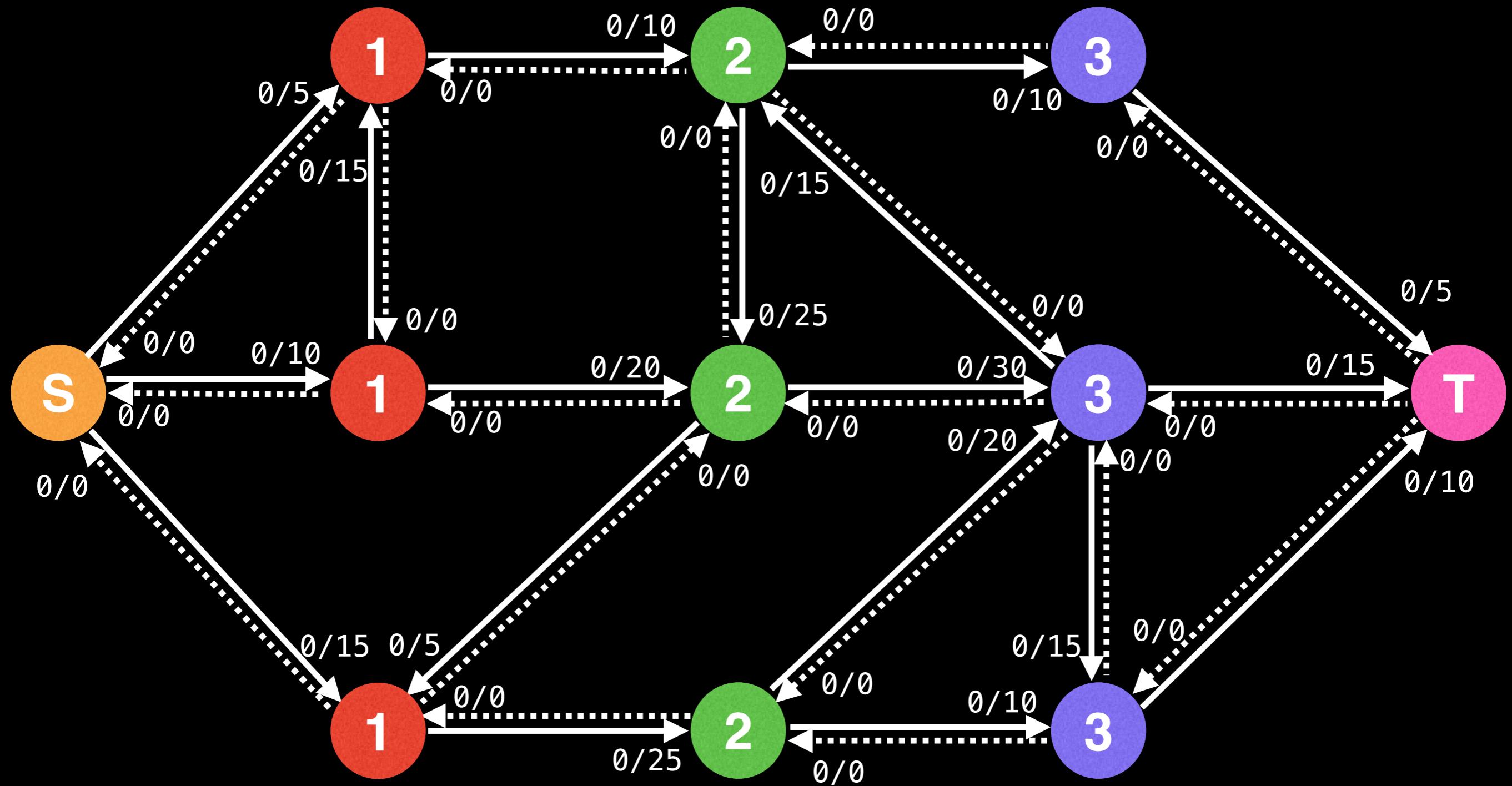
Begin by constructing the level graph for the current flow graph.



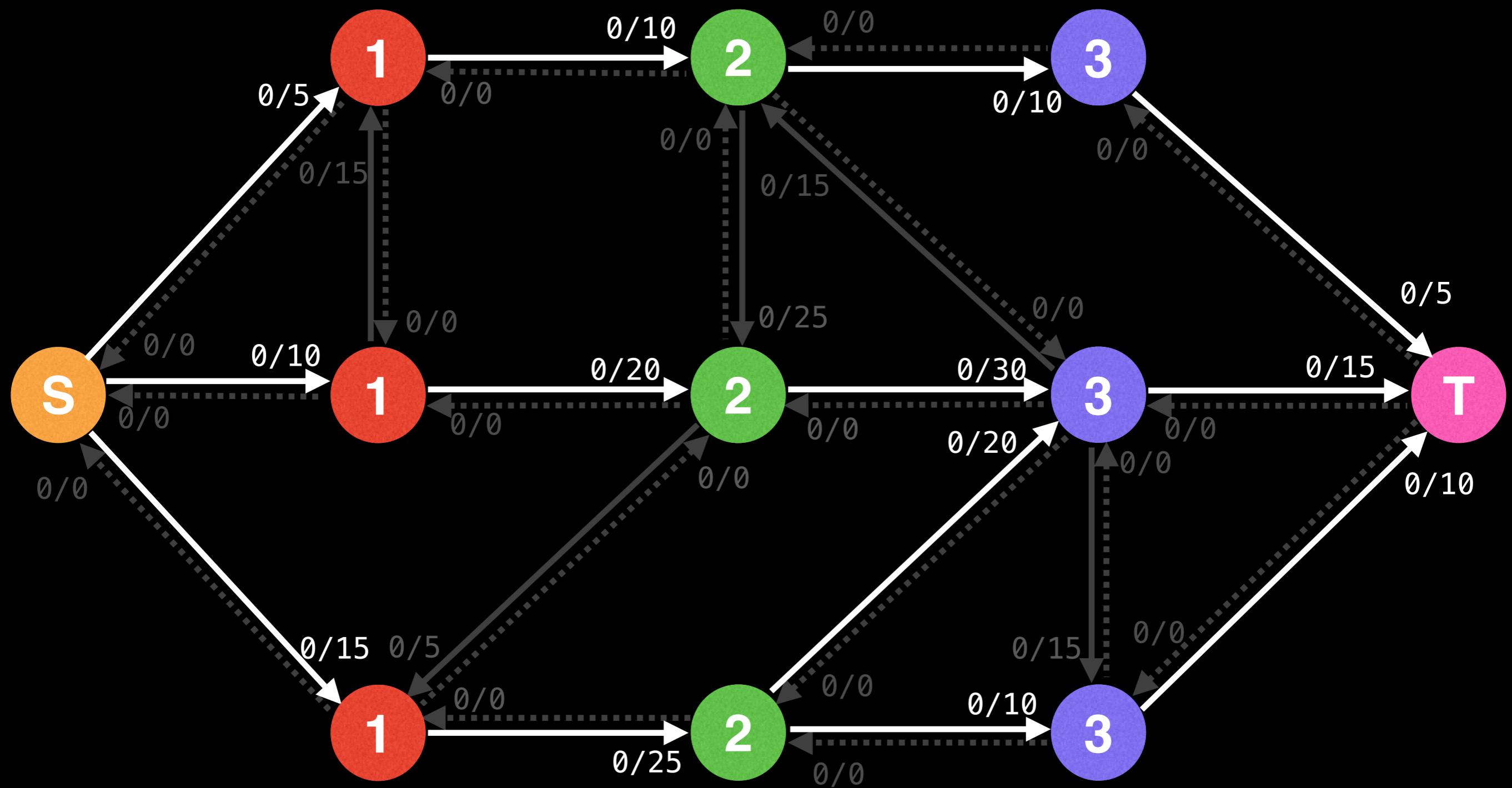
Begin by constructing the level graph for the current flow graph.



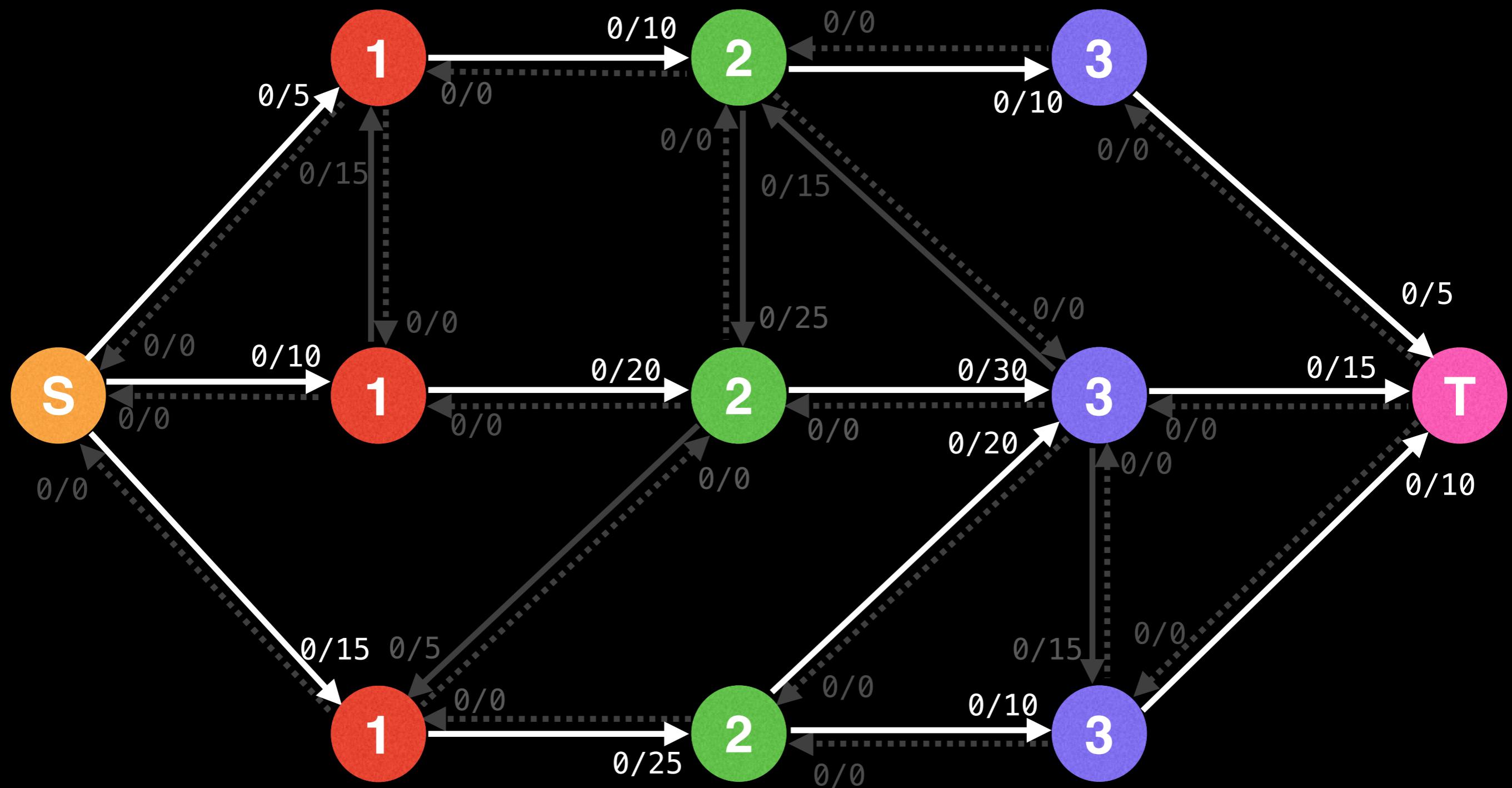
Begin by constructing the level graph for the current flow graph.



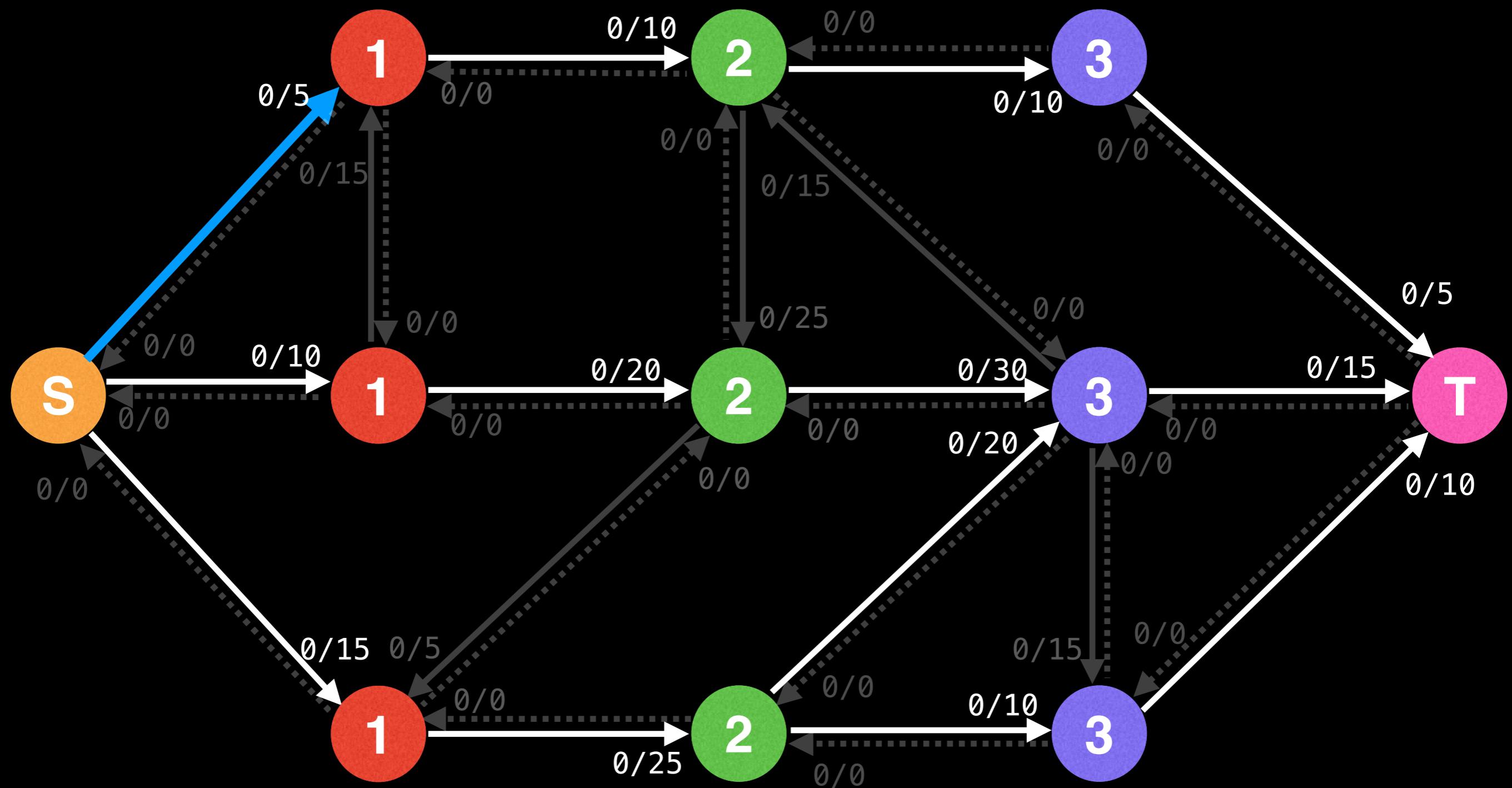
The level graph consists of all edges which go from L to L+1 in level and have remaining capacity > 0.



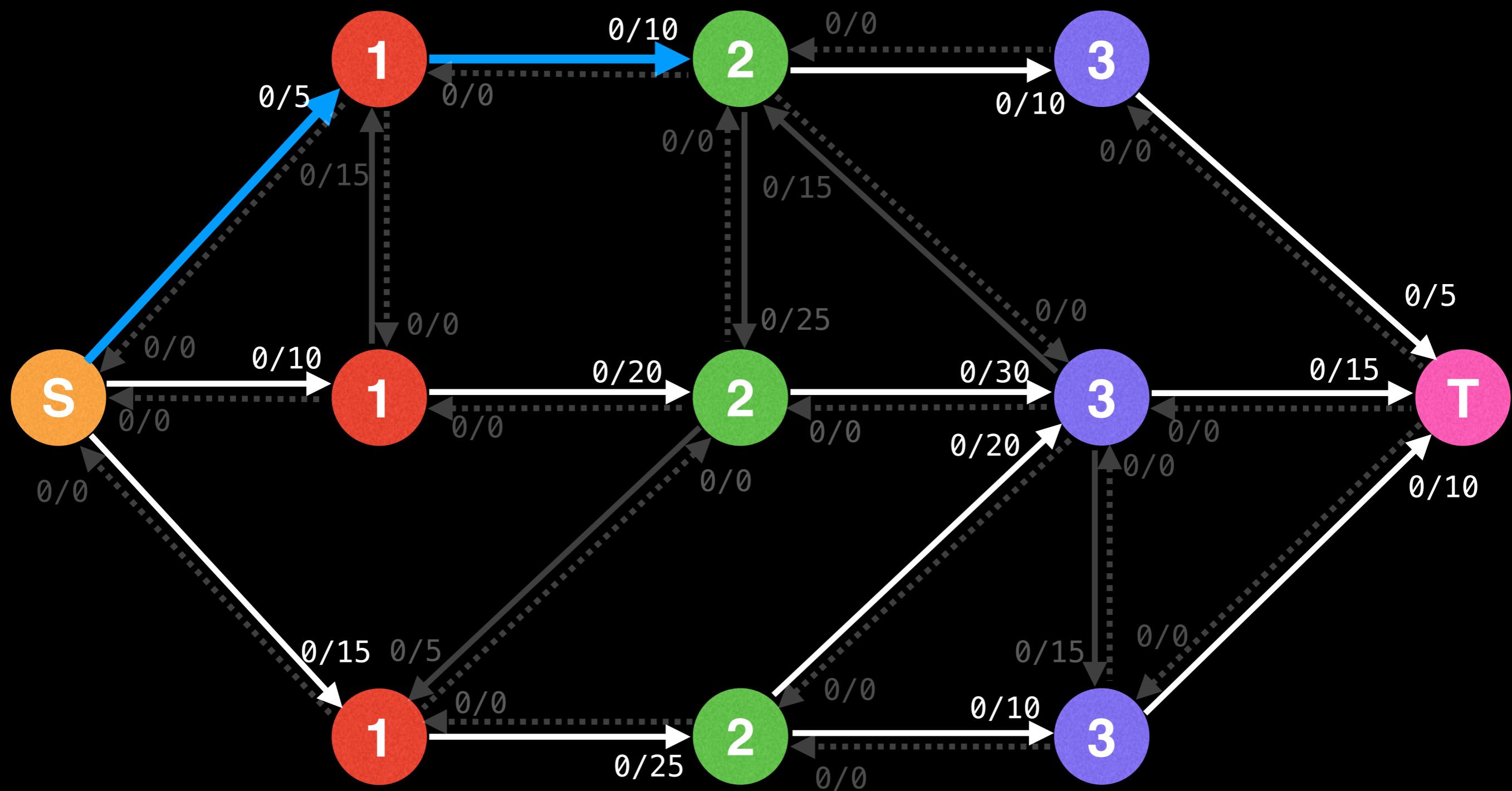
Find paths from  $s \rightarrow t$  until a blocking flow is reached (i.e. you cannot find any more paths).



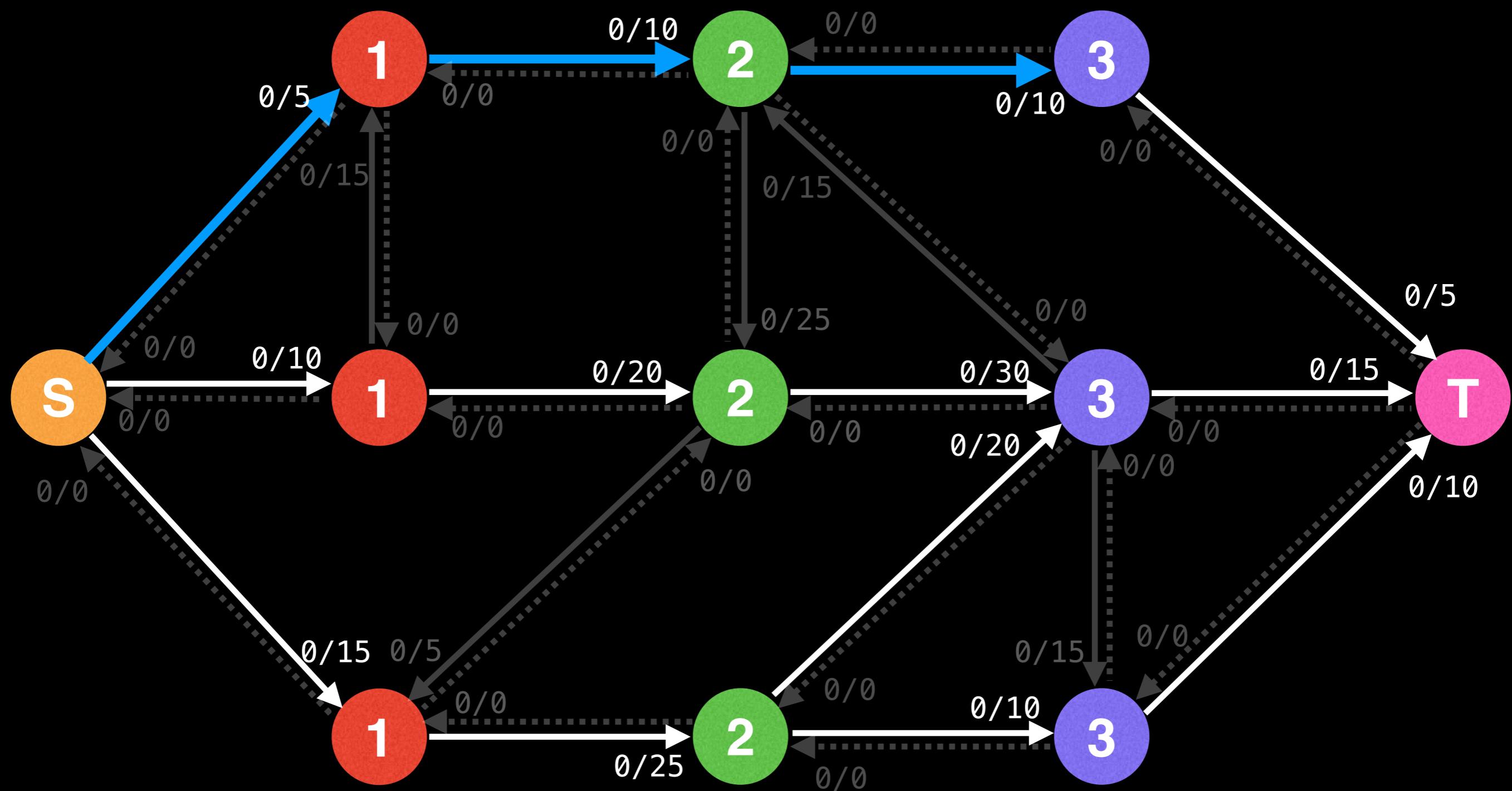
Find paths from  $s \rightarrow t$  until a blocking flow is reached (i.e. you cannot find any more paths).



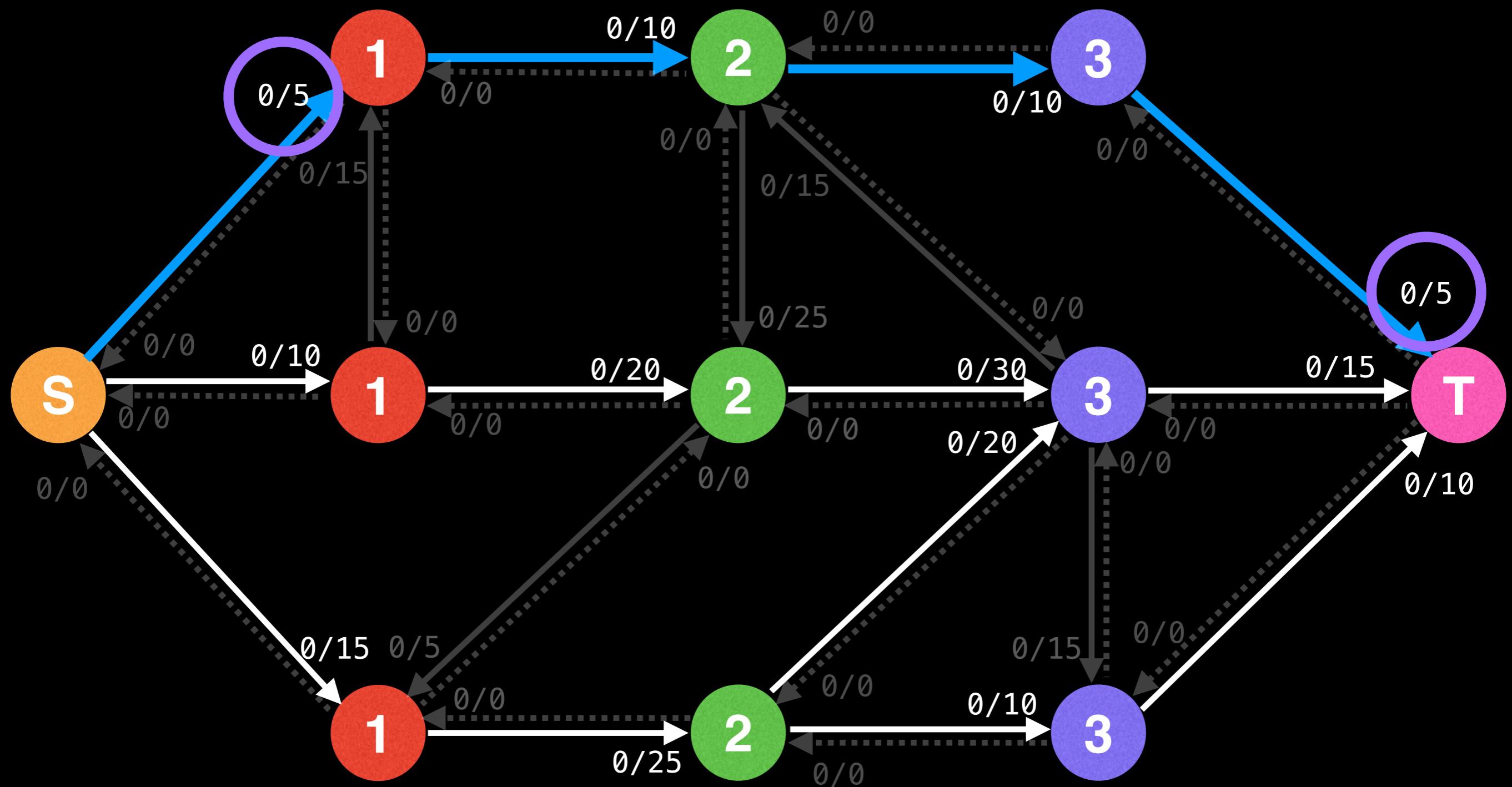
Find paths from  $s \rightarrow t$  until a blocking flow is reached (i.e. you cannot find any more paths).



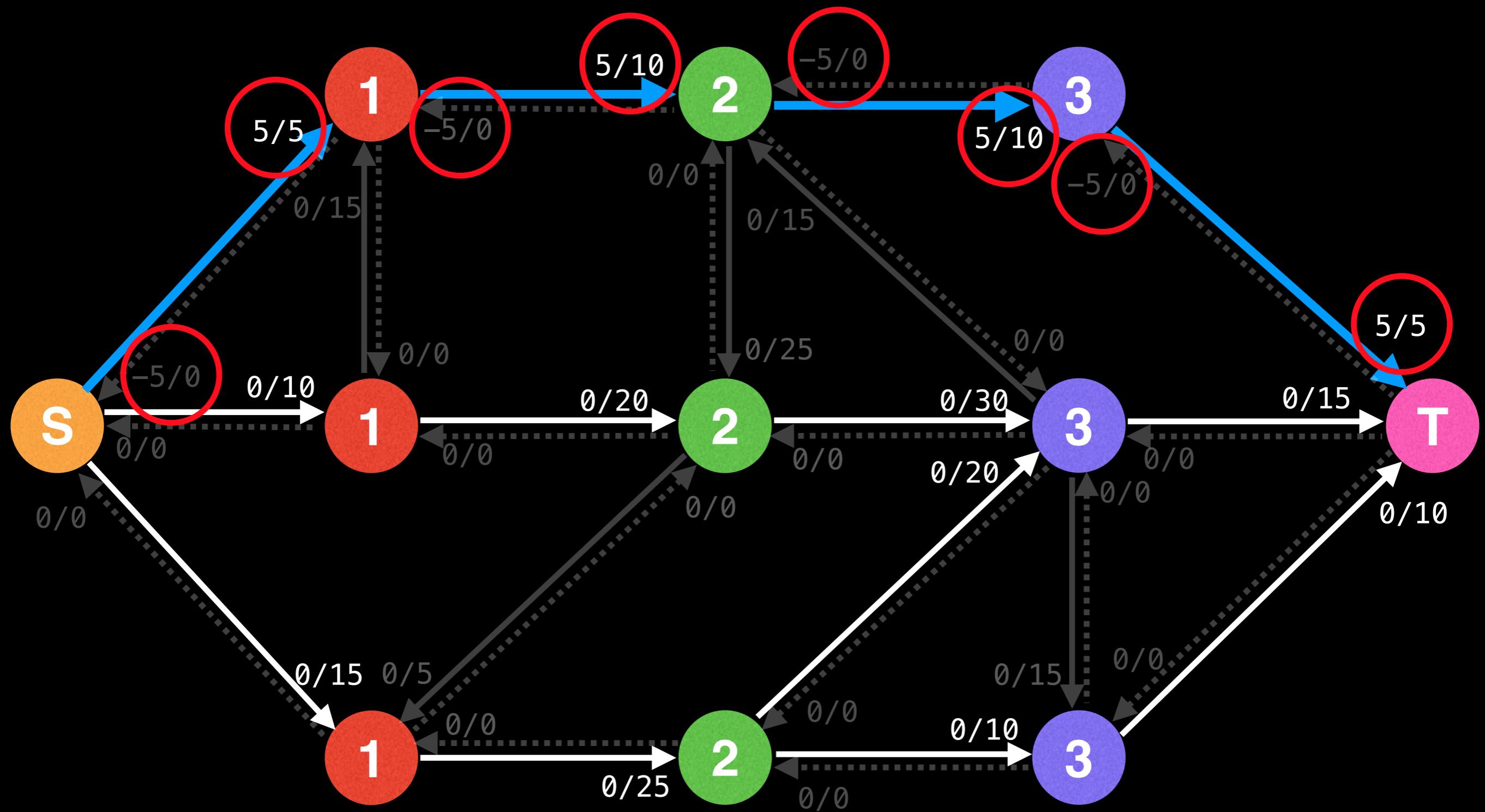
Find paths from  $s \rightarrow t$  until a blocking flow is reached (i.e. you cannot find any more paths).



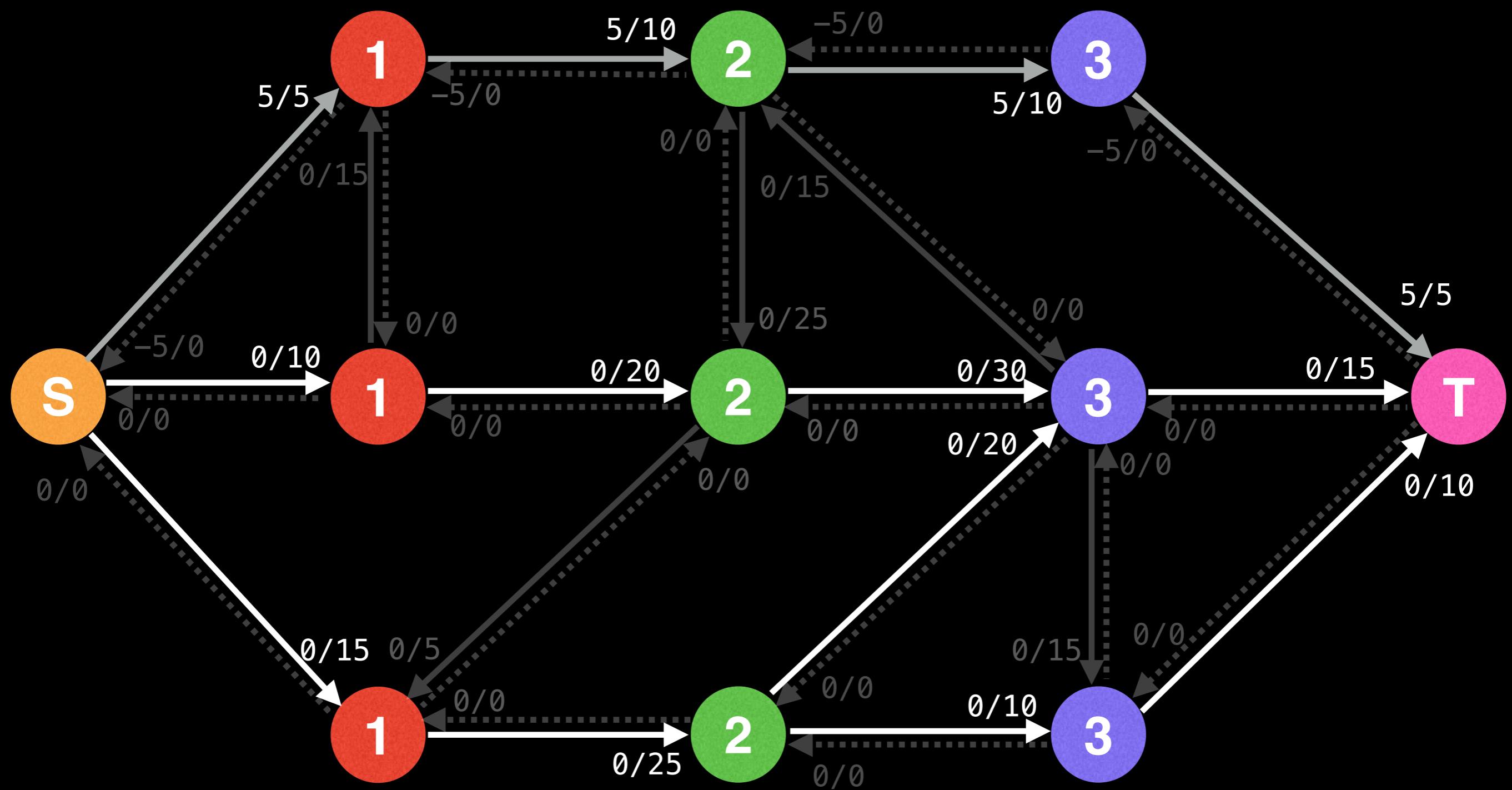
The current path has a bottleneck value of 5  
since  $\min(5-0, 10-0, 10-0, 5-0) = 5$ .

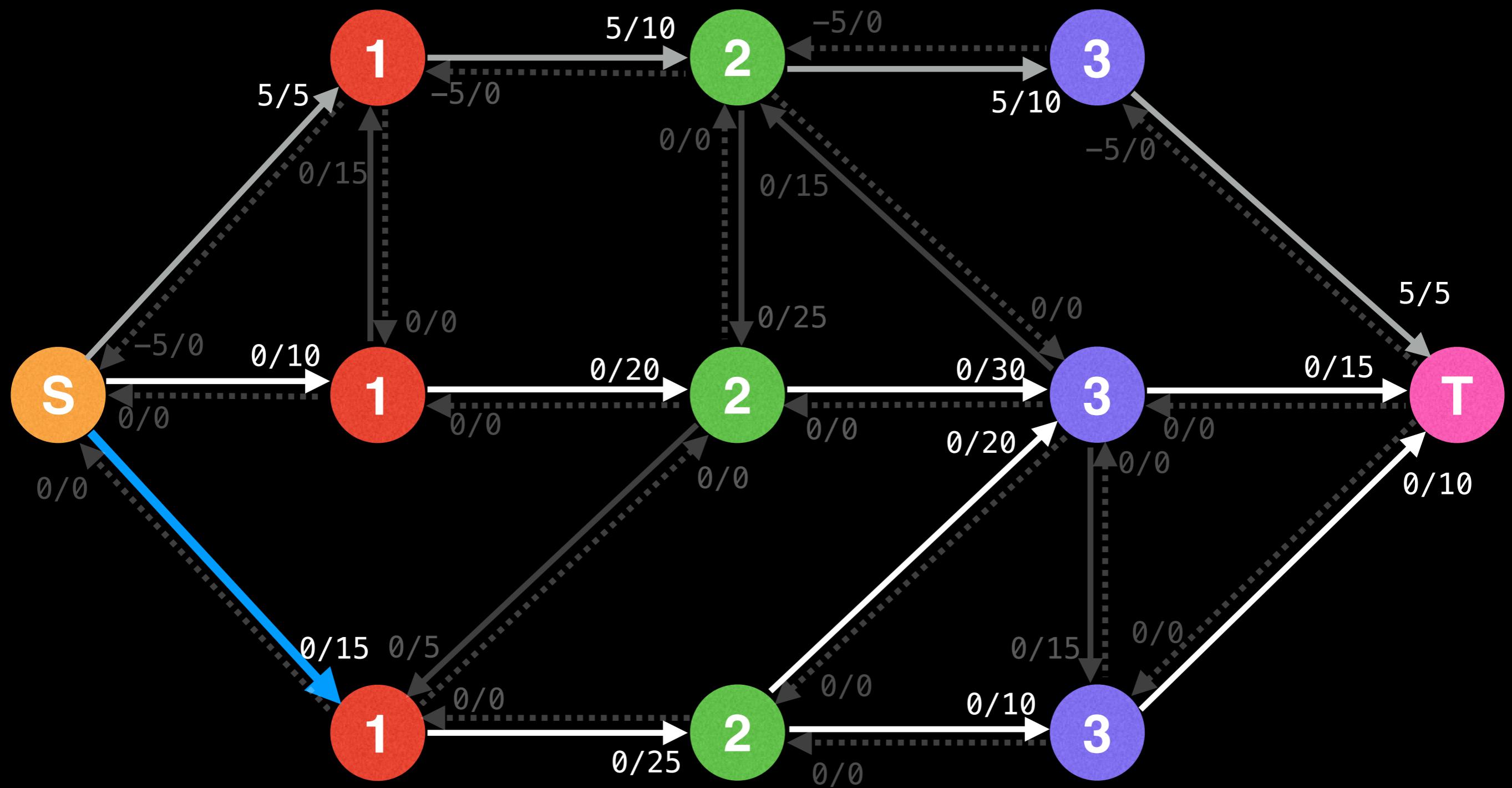


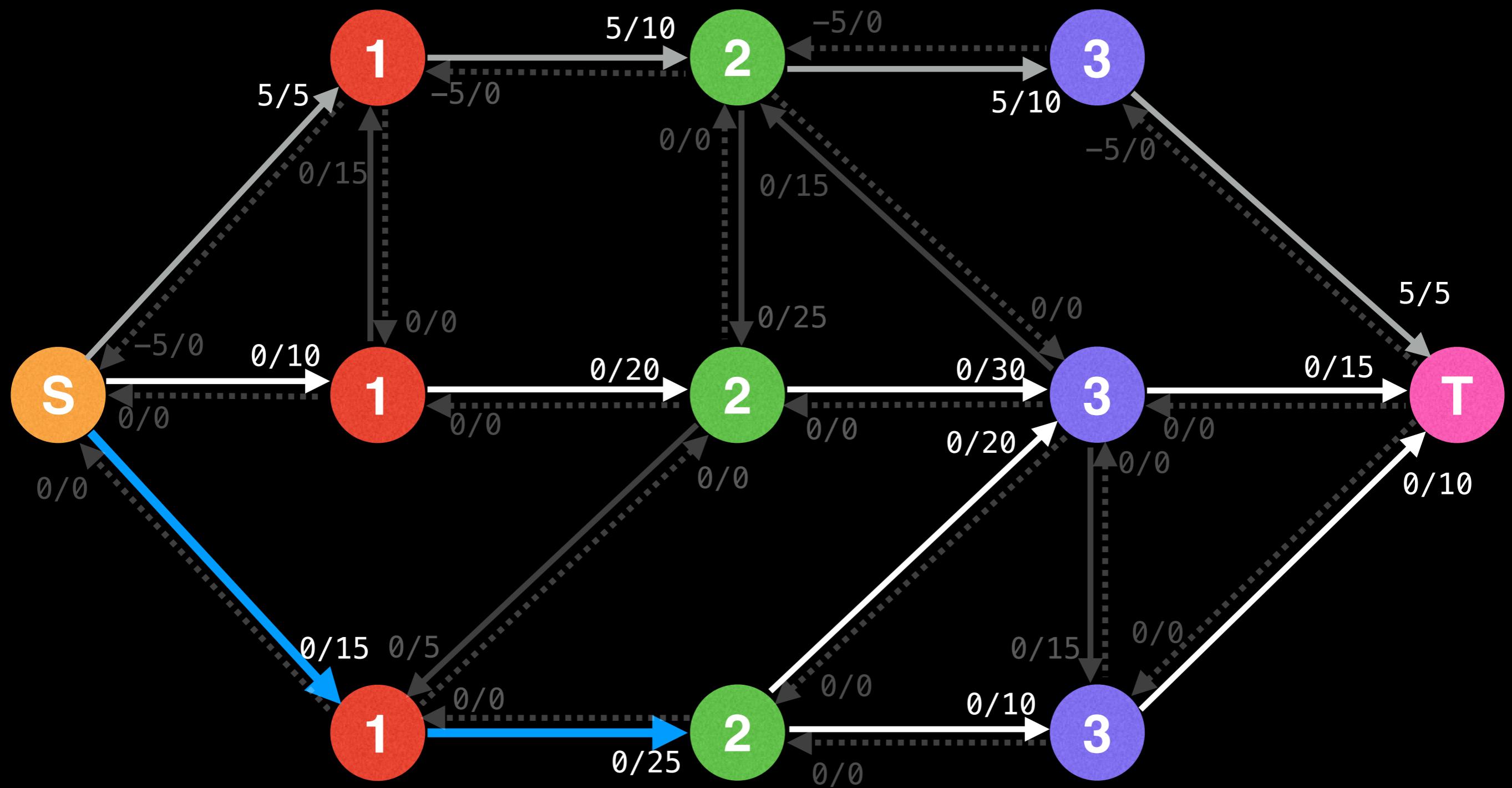
Augment the flow values along the path by 5.

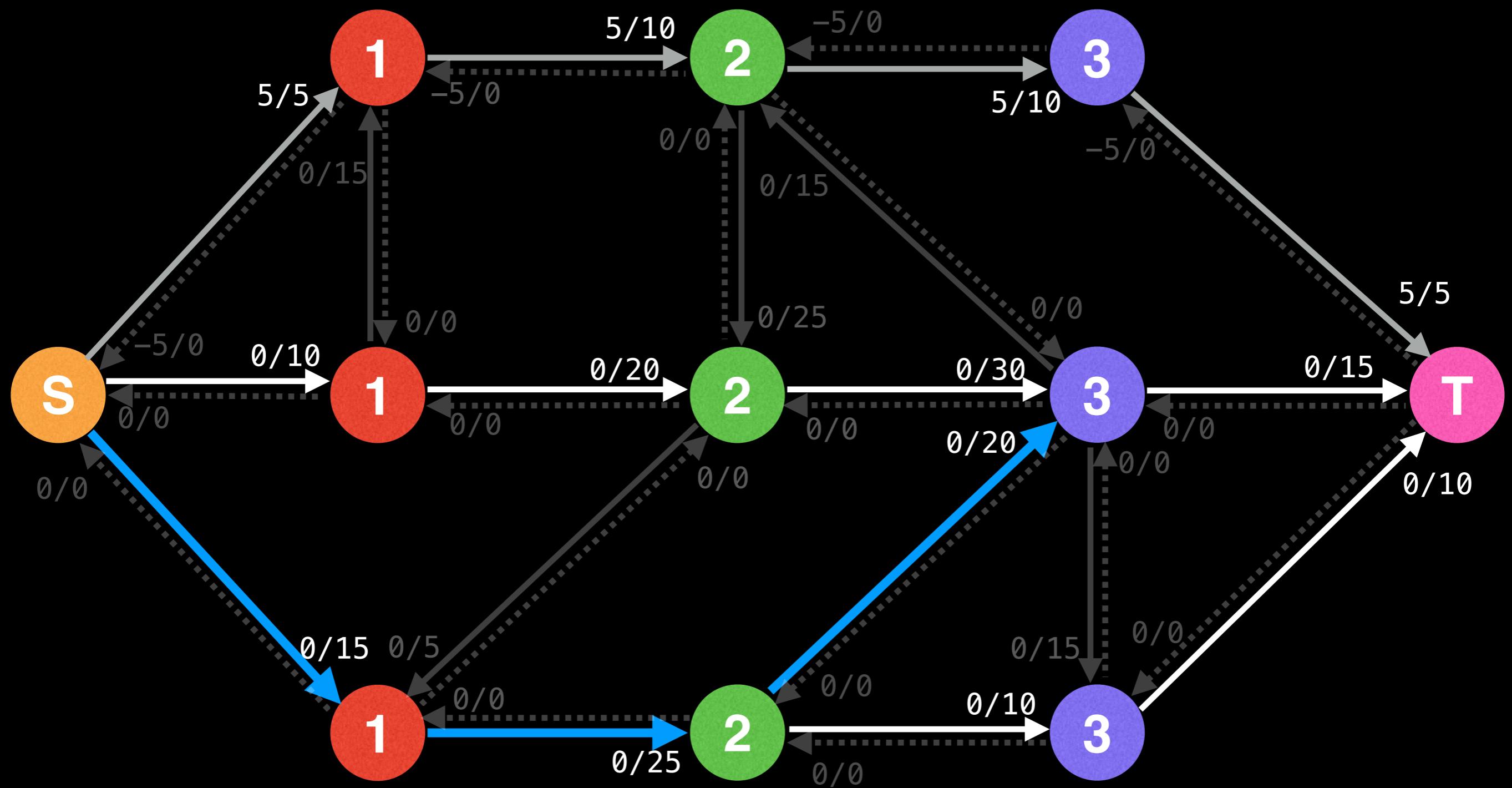


The blocking flow has not yet been reached since there still exists paths from  $s \rightarrow t$ .

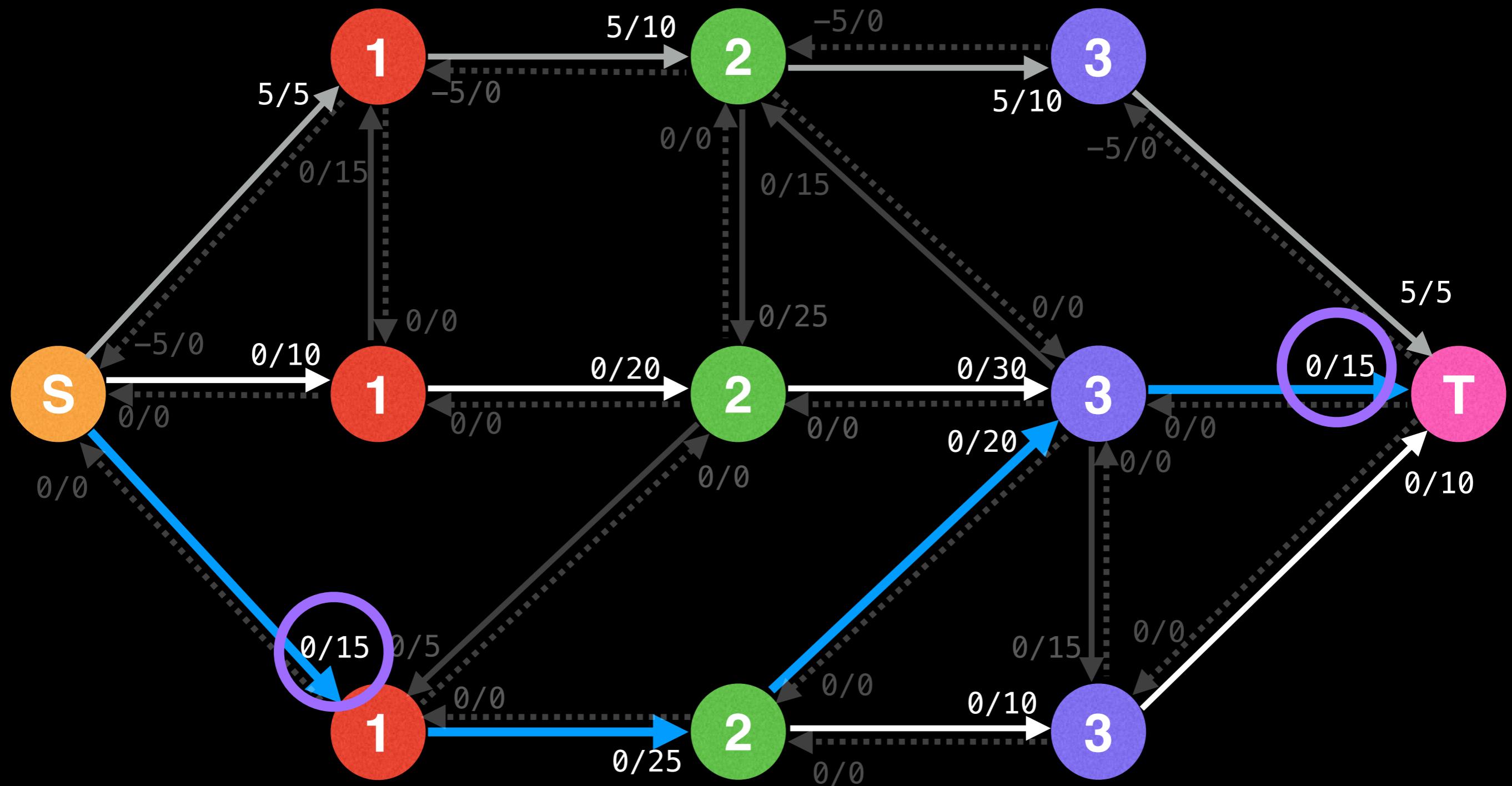




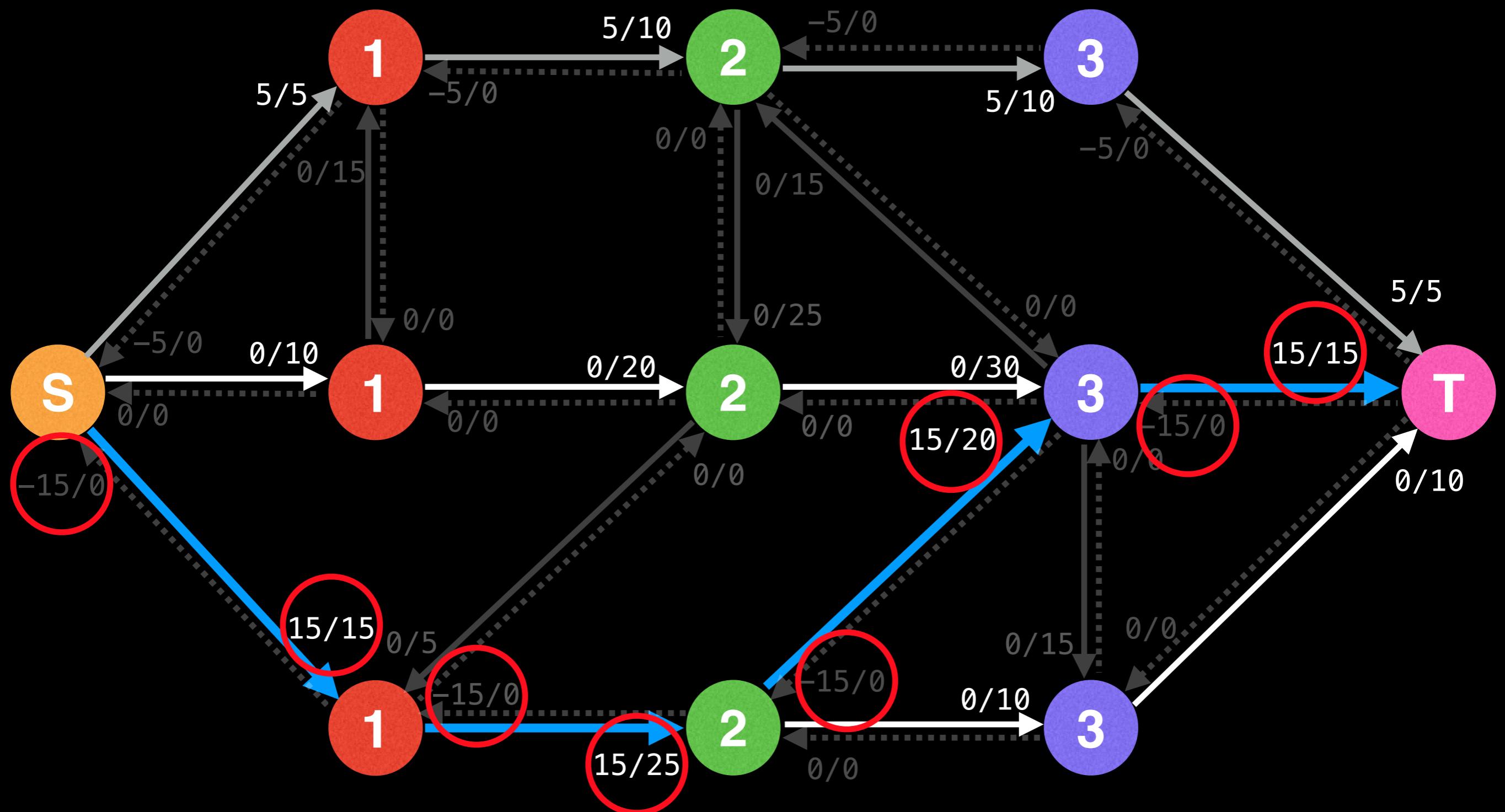




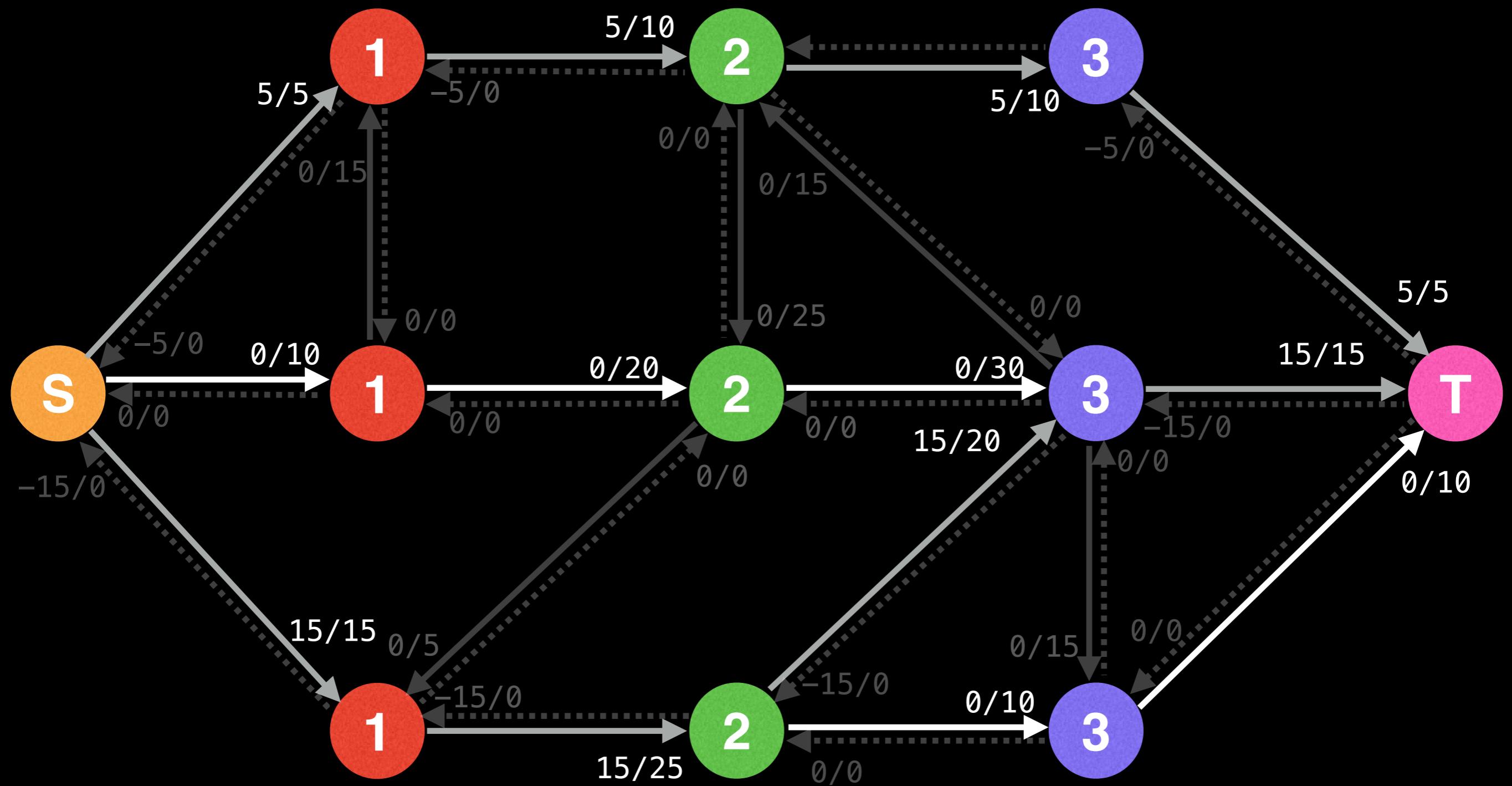
The current path has a bottleneck value of 15  
since  $\min(15-0, 25-0, 20-0, 15-0) = 15$ .

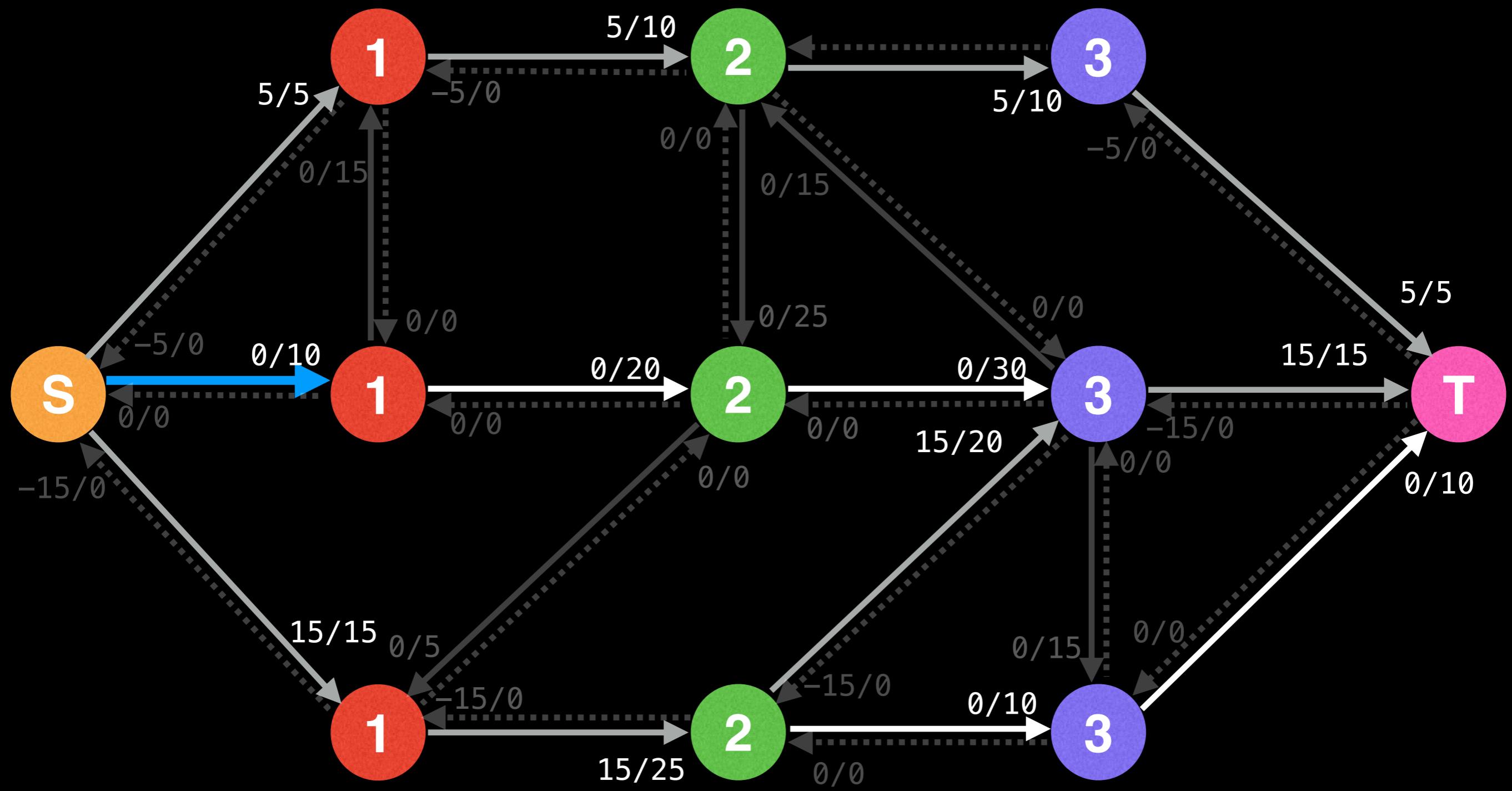


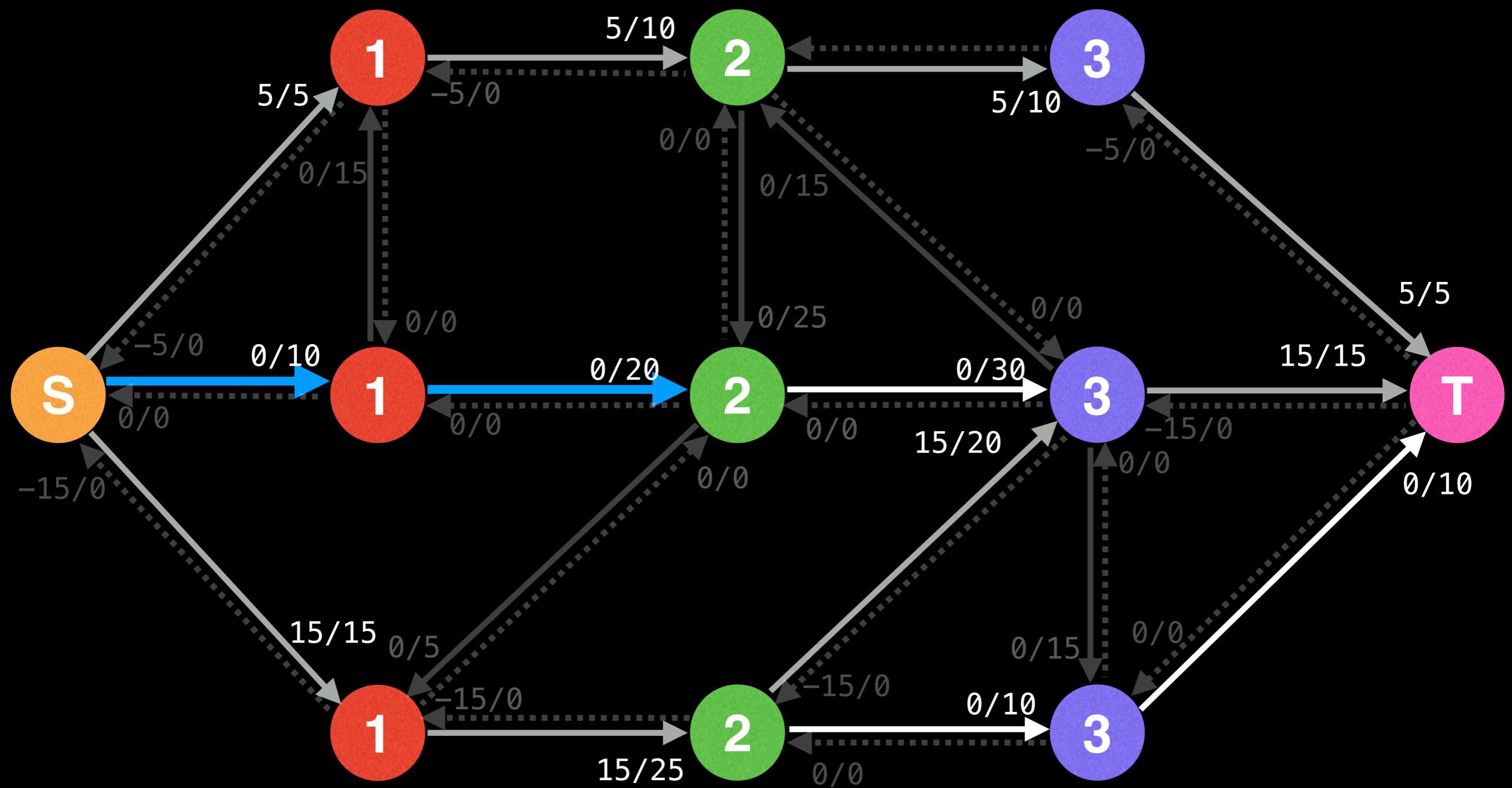
Augment the flow values along the path by 15.

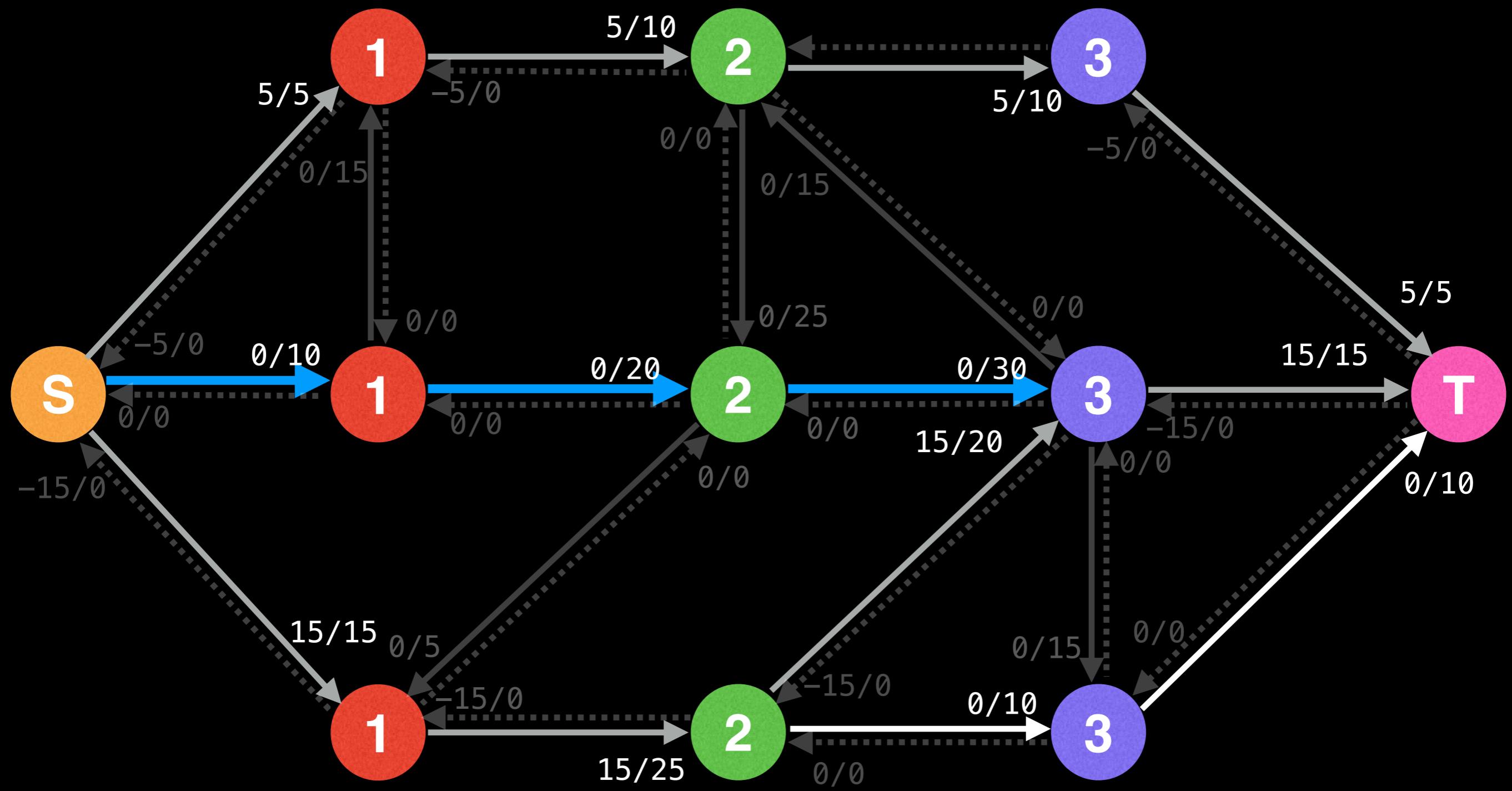


Let's try and find another path from  $s \rightarrow t$ .

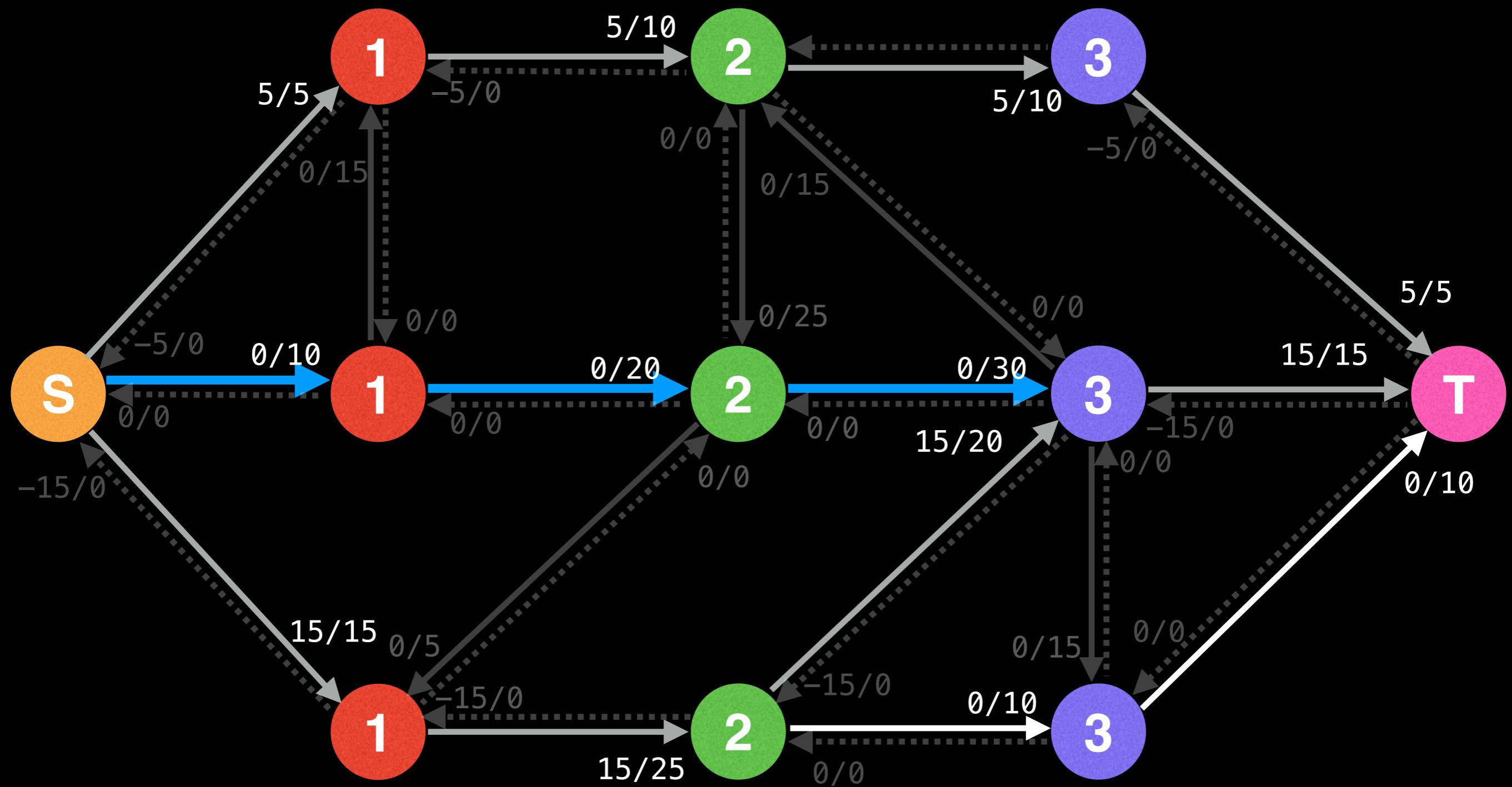


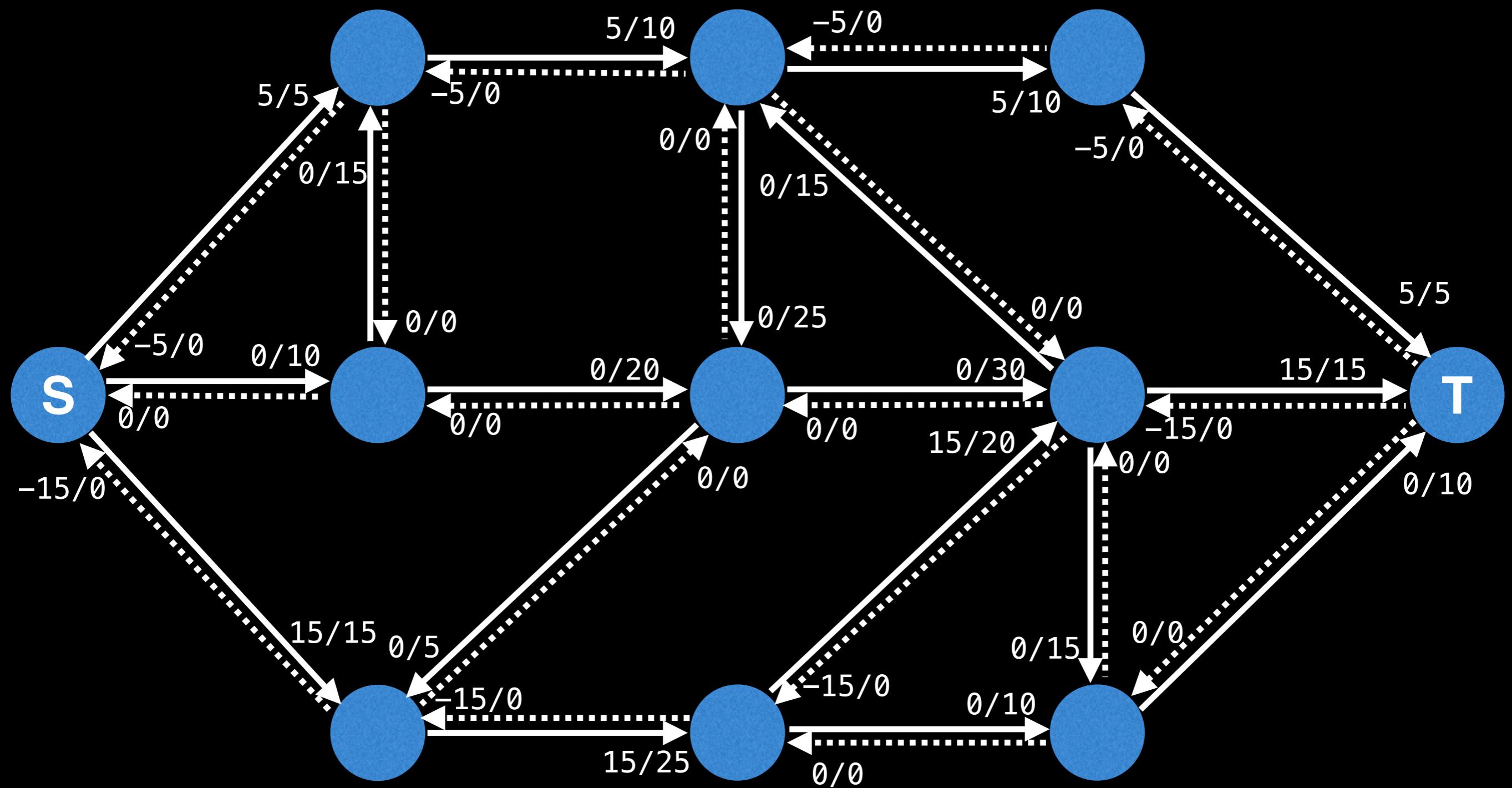




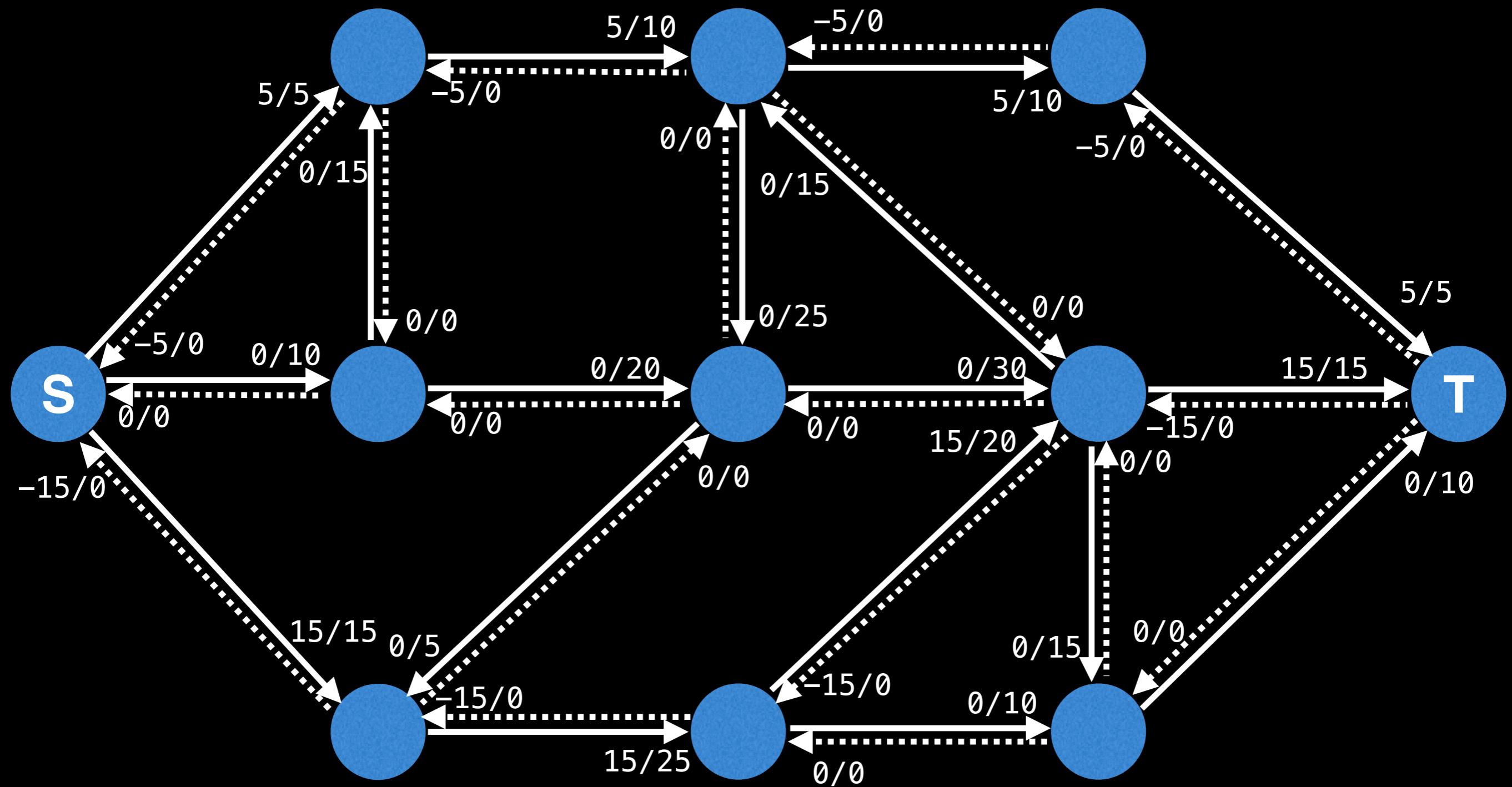


We get stuck finding a path, so the blocking flow has been reached.

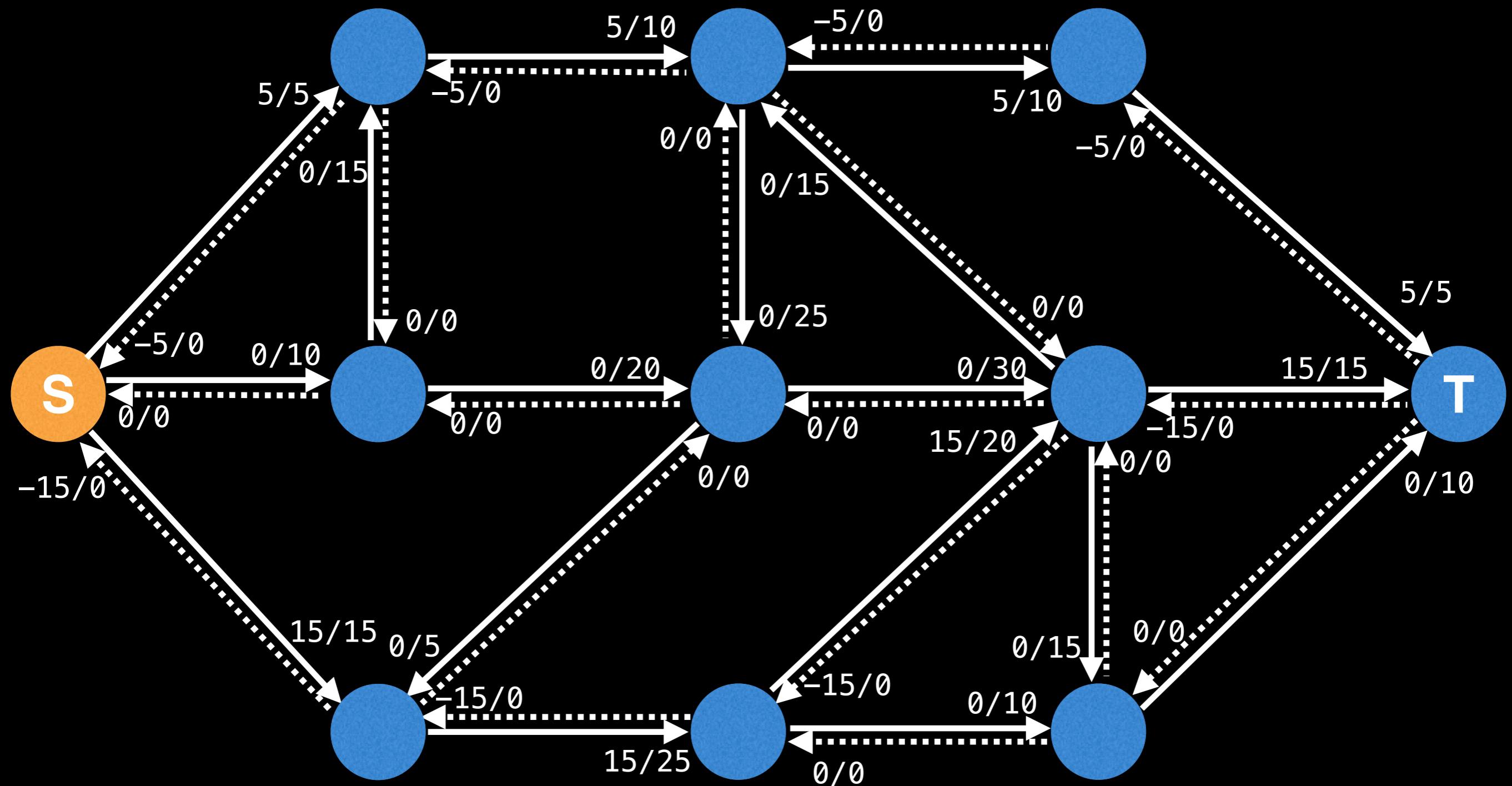




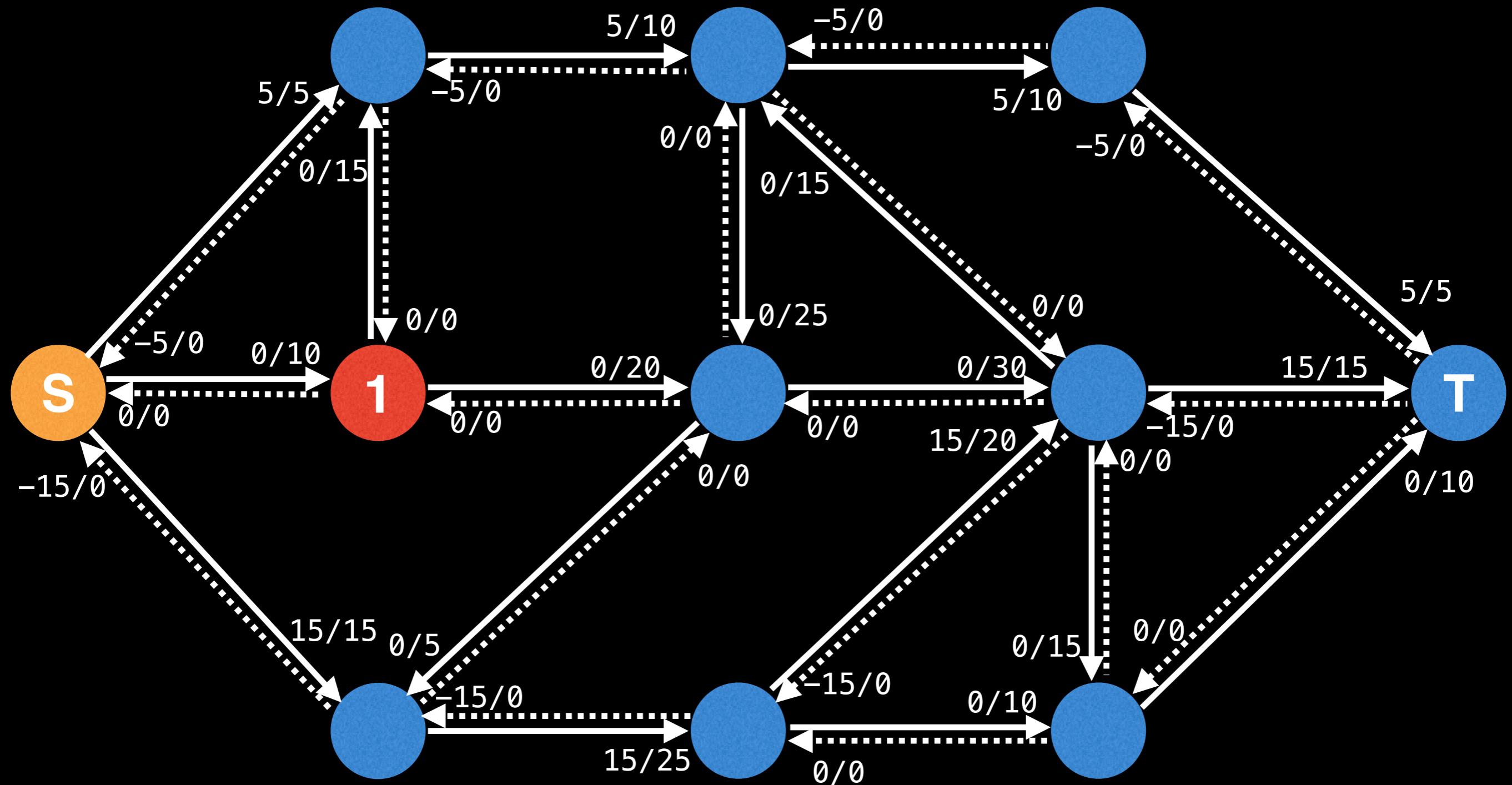
Rebuild the level graph. This time it should look different because the remaining capacities of multiple edges has changed.



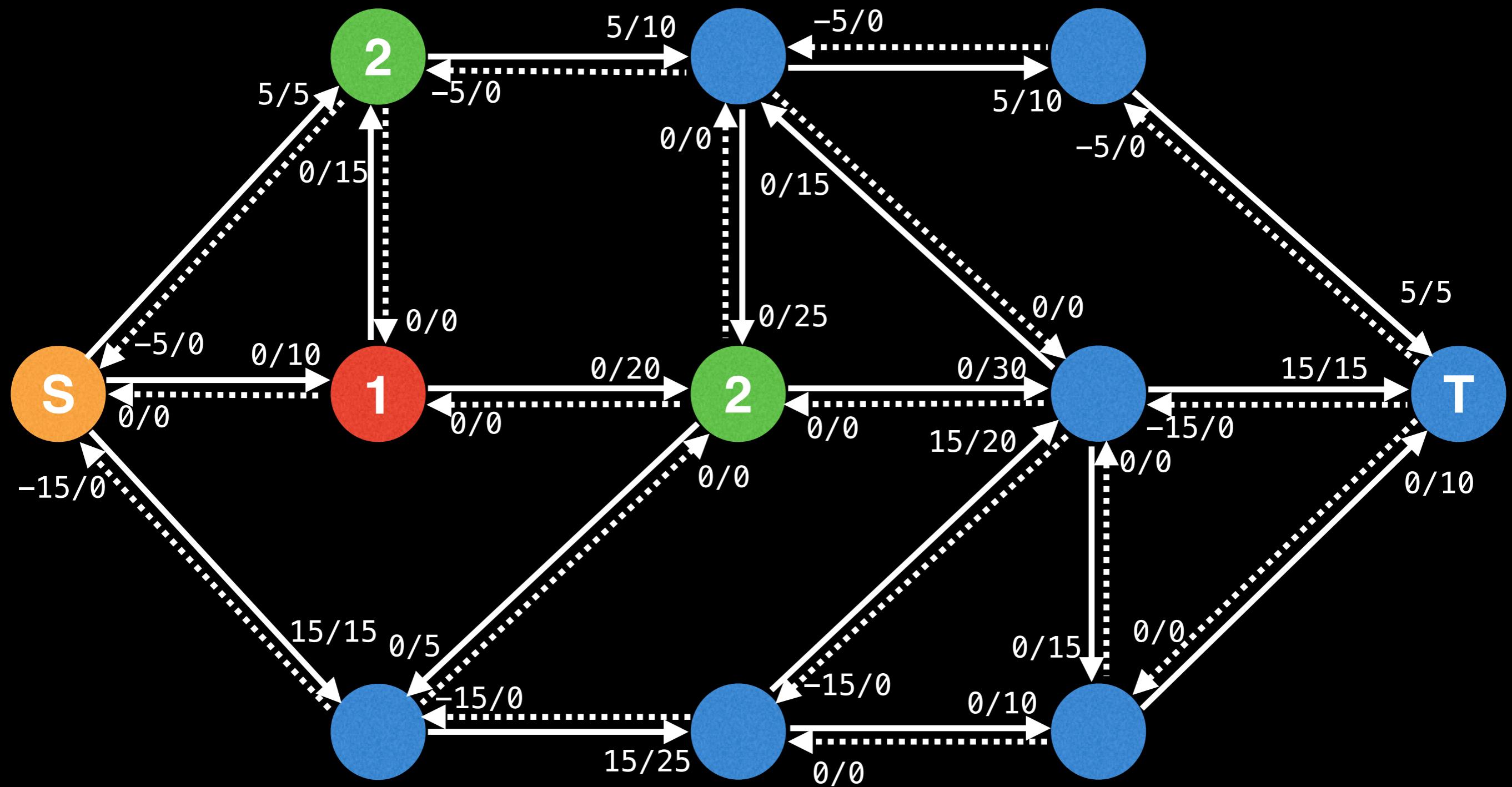
Rebuild the level graph. This time it should look different because the remaining capacities of multiple edges has changed.



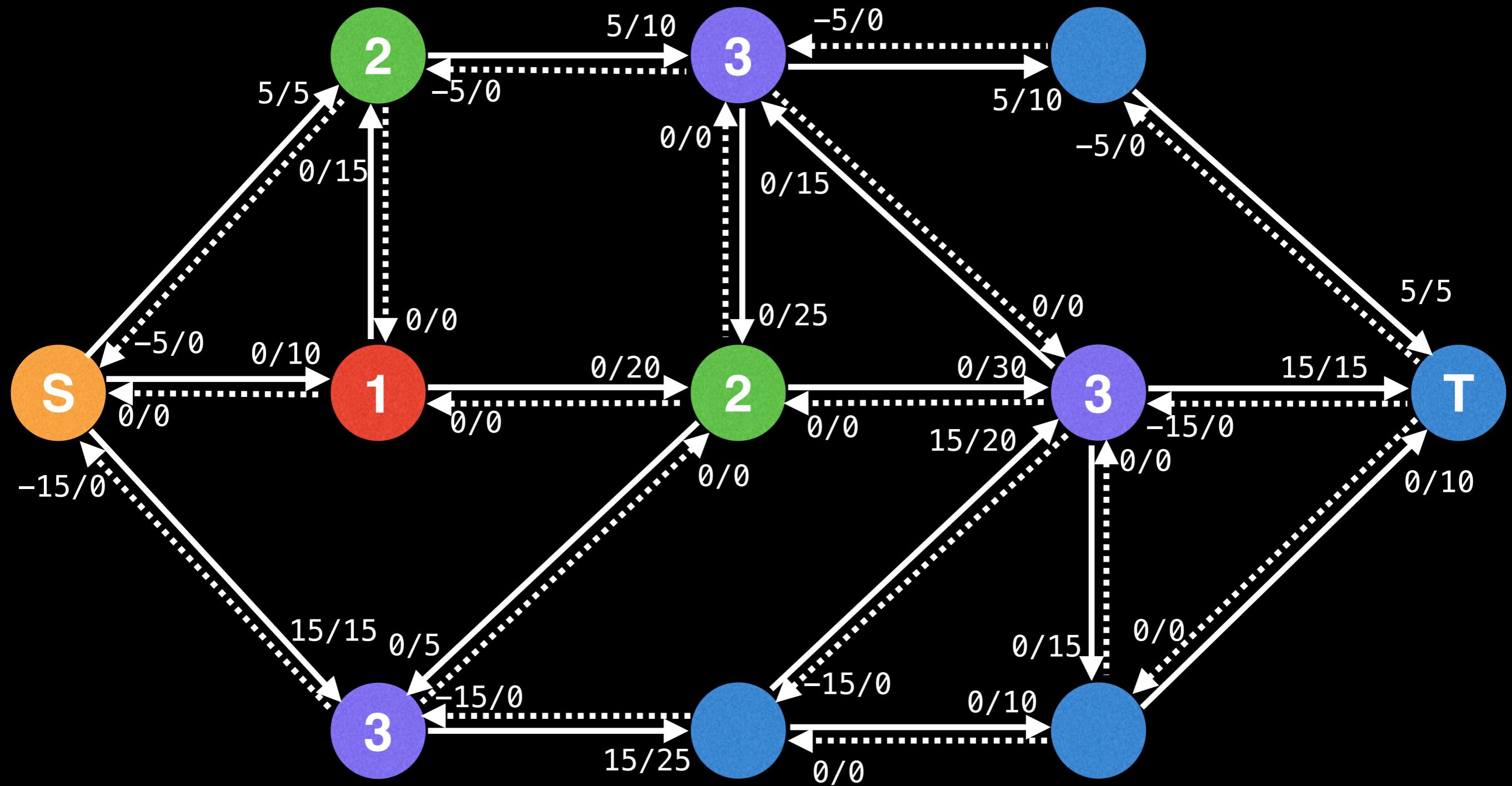
Rebuild the level graph. This time it should look different because the remaining capacities of multiple edges has changed.



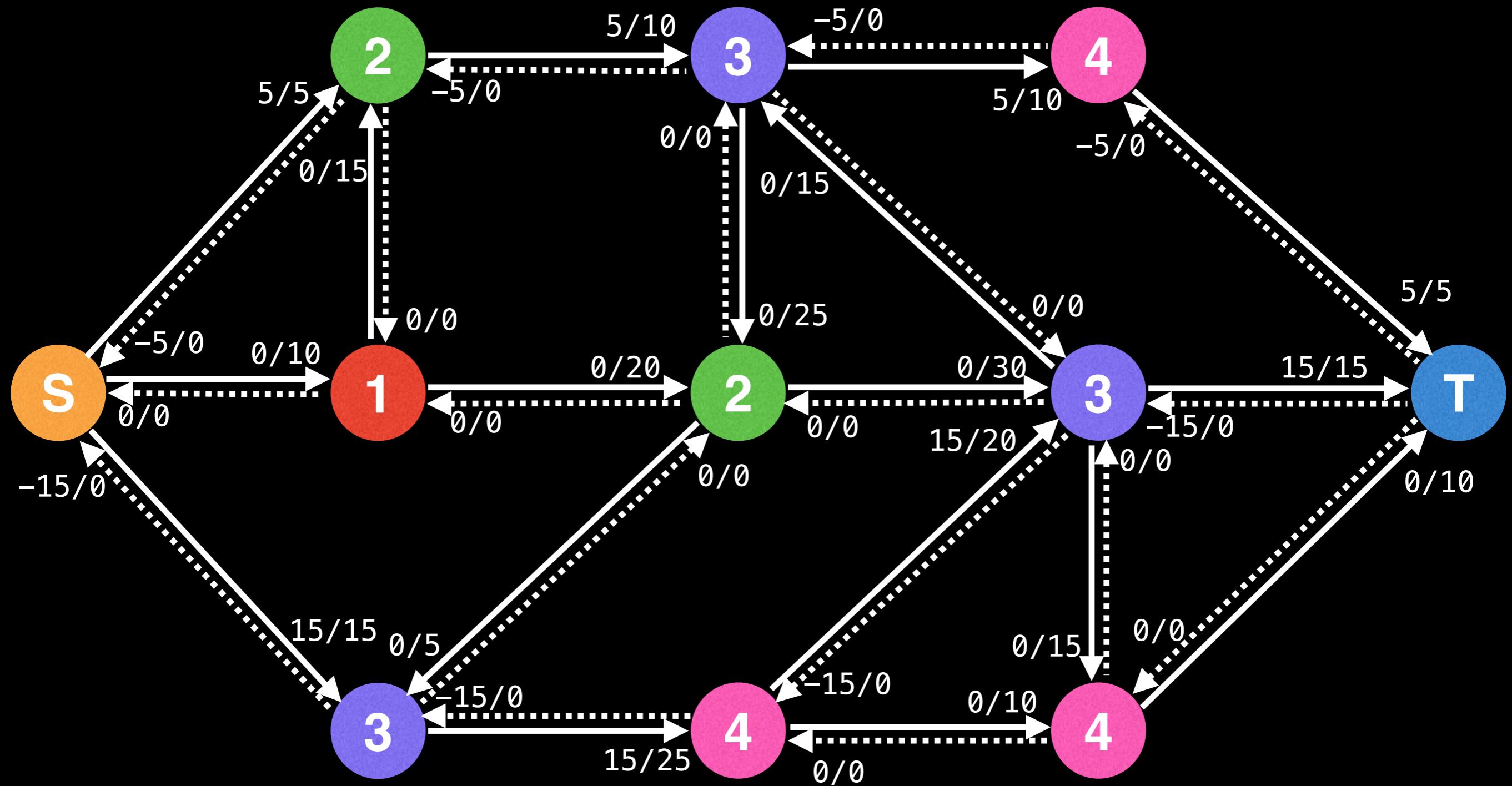
Rebuild the level graph. This time it should look different because the remaining capacities of multiple edges has changed.



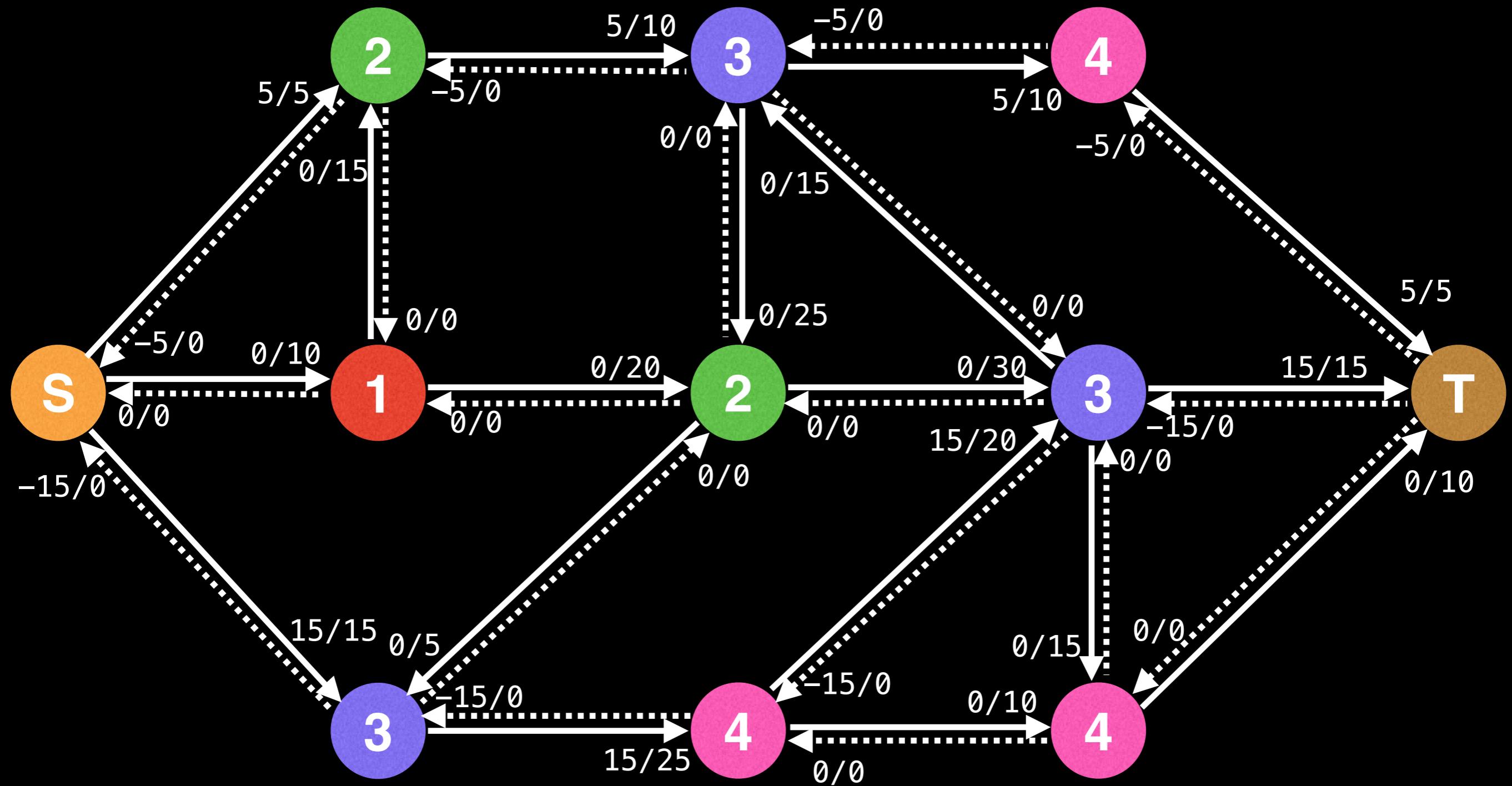
Rebuild the level graph. This time it should look different because the remaining capacities of multiple edges has changed.



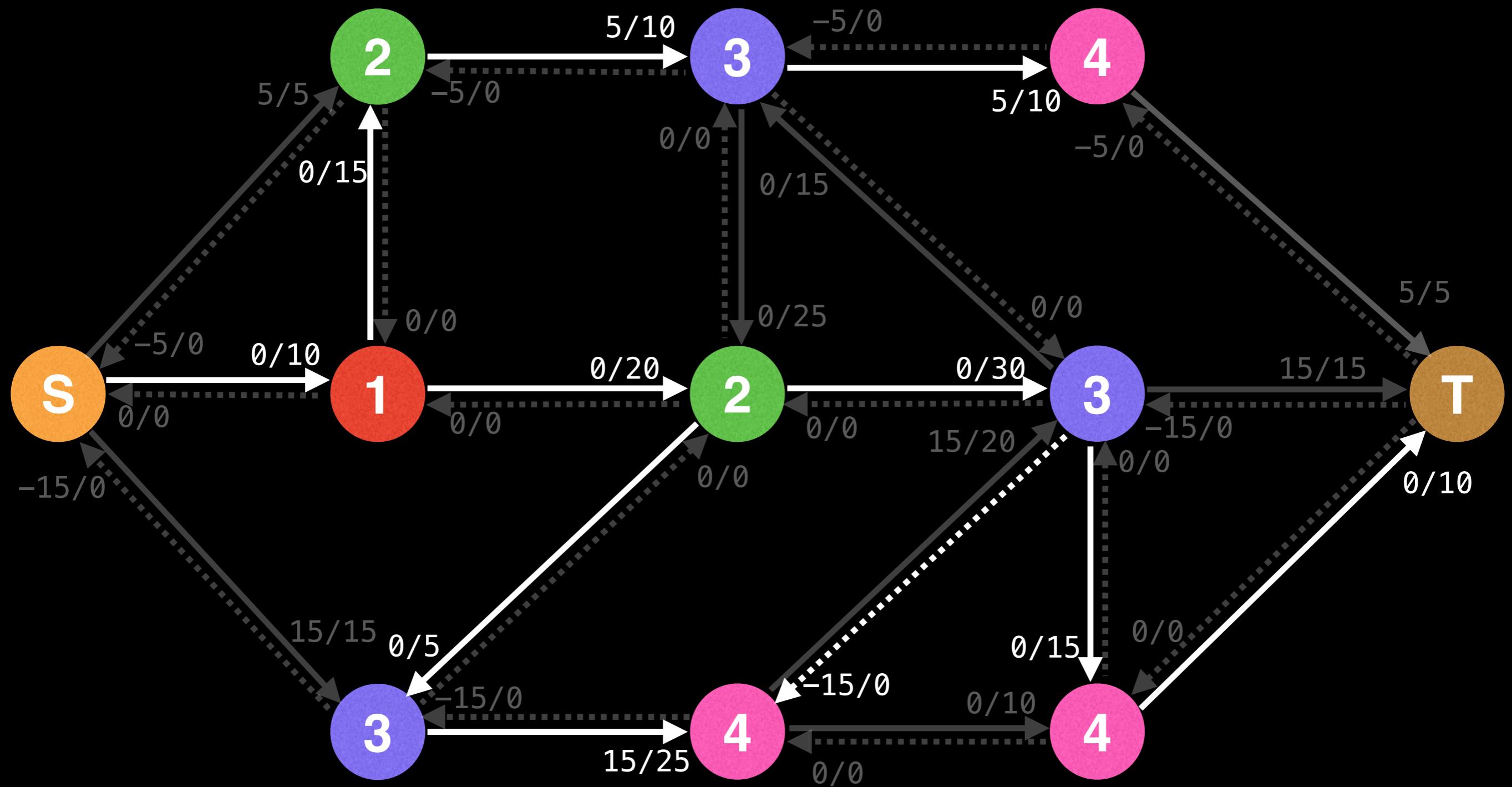
Rebuild the level graph. This time it should look different because the remaining capacities of multiple edges has changed.



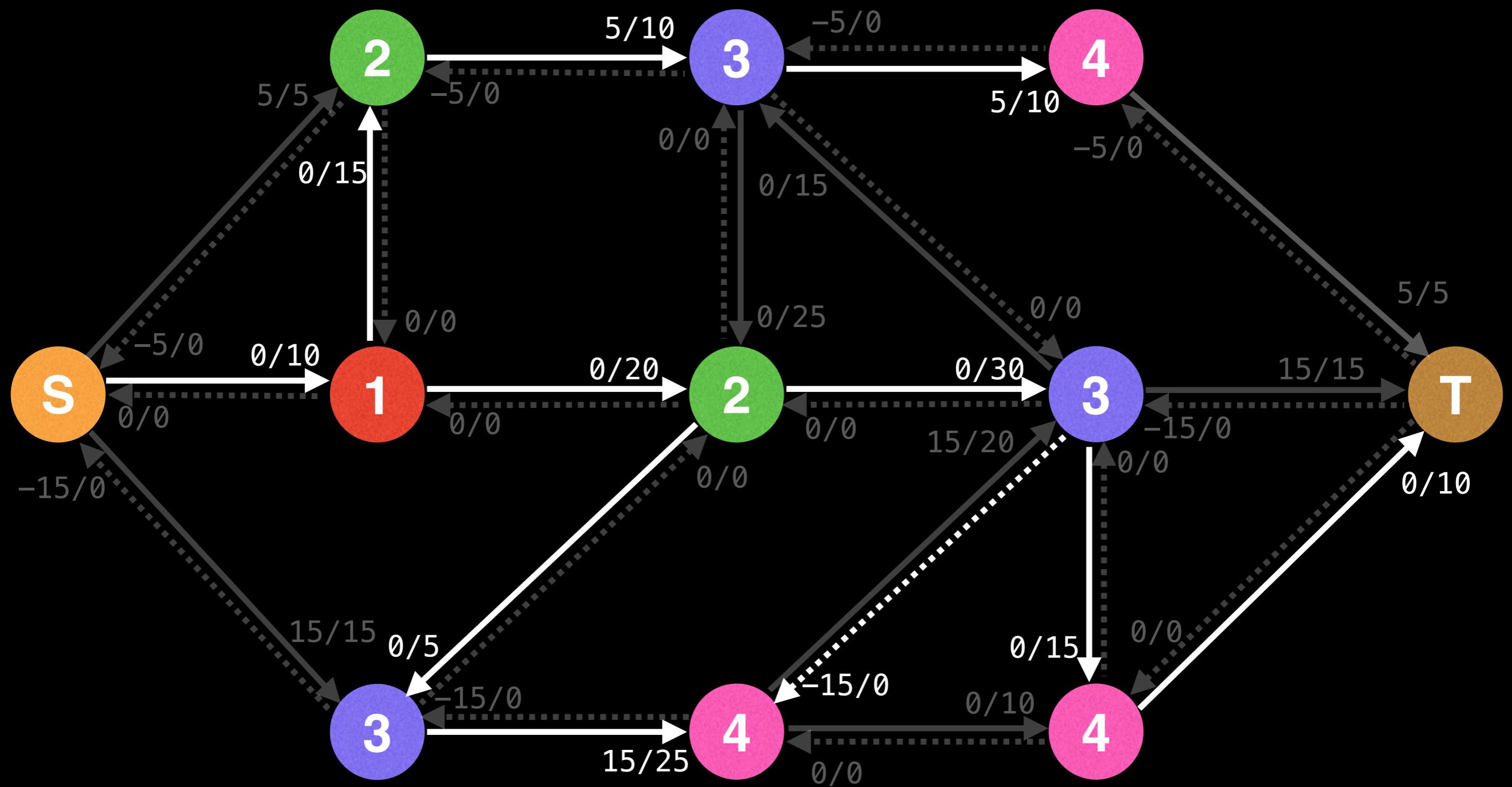
Rebuild the level graph. This time it should look different because the remaining capacities of multiple edges has changed.



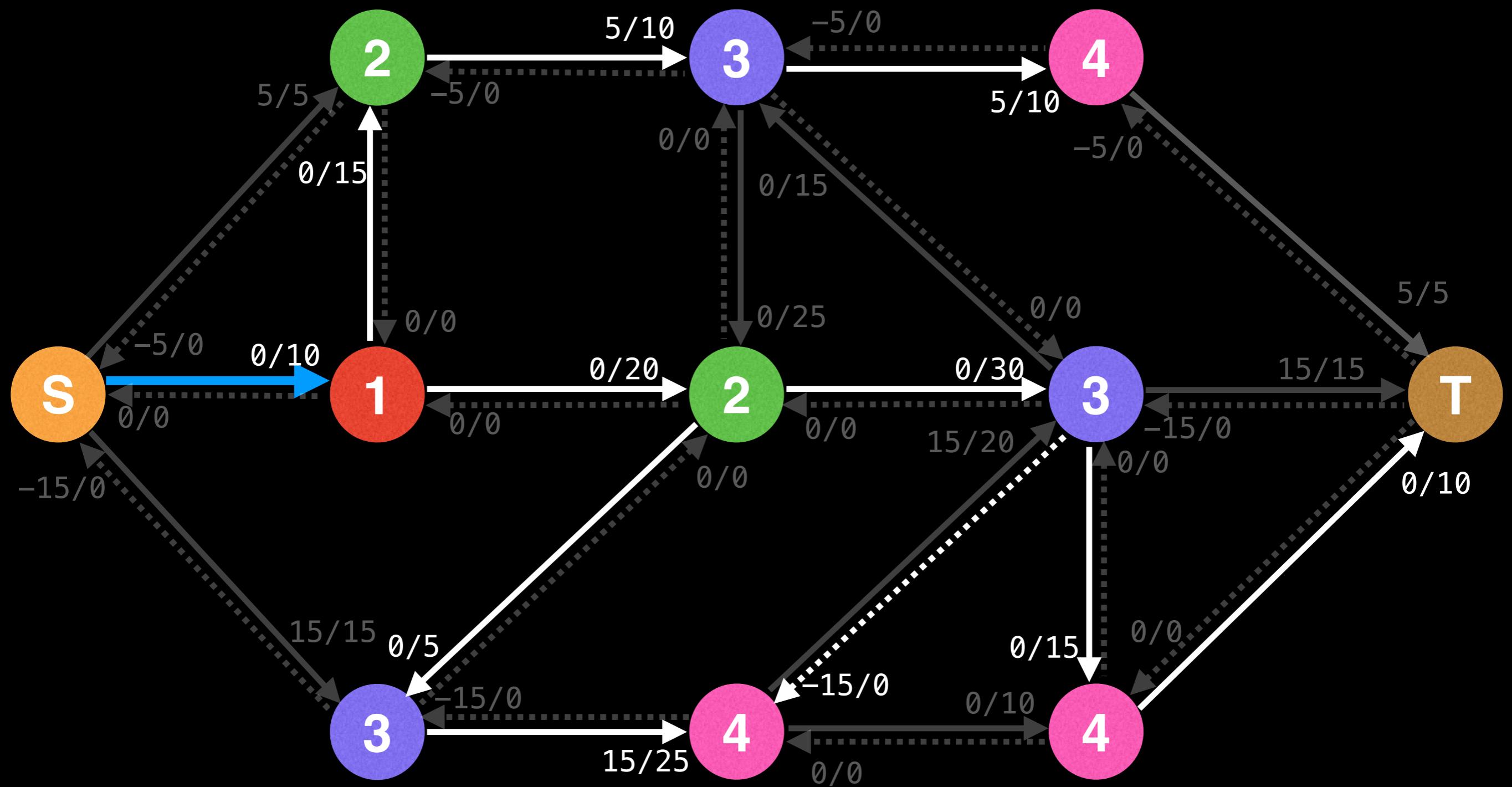
The level graph consists of all edges which go from L to L+1 in level and have remaining capacity > 0.



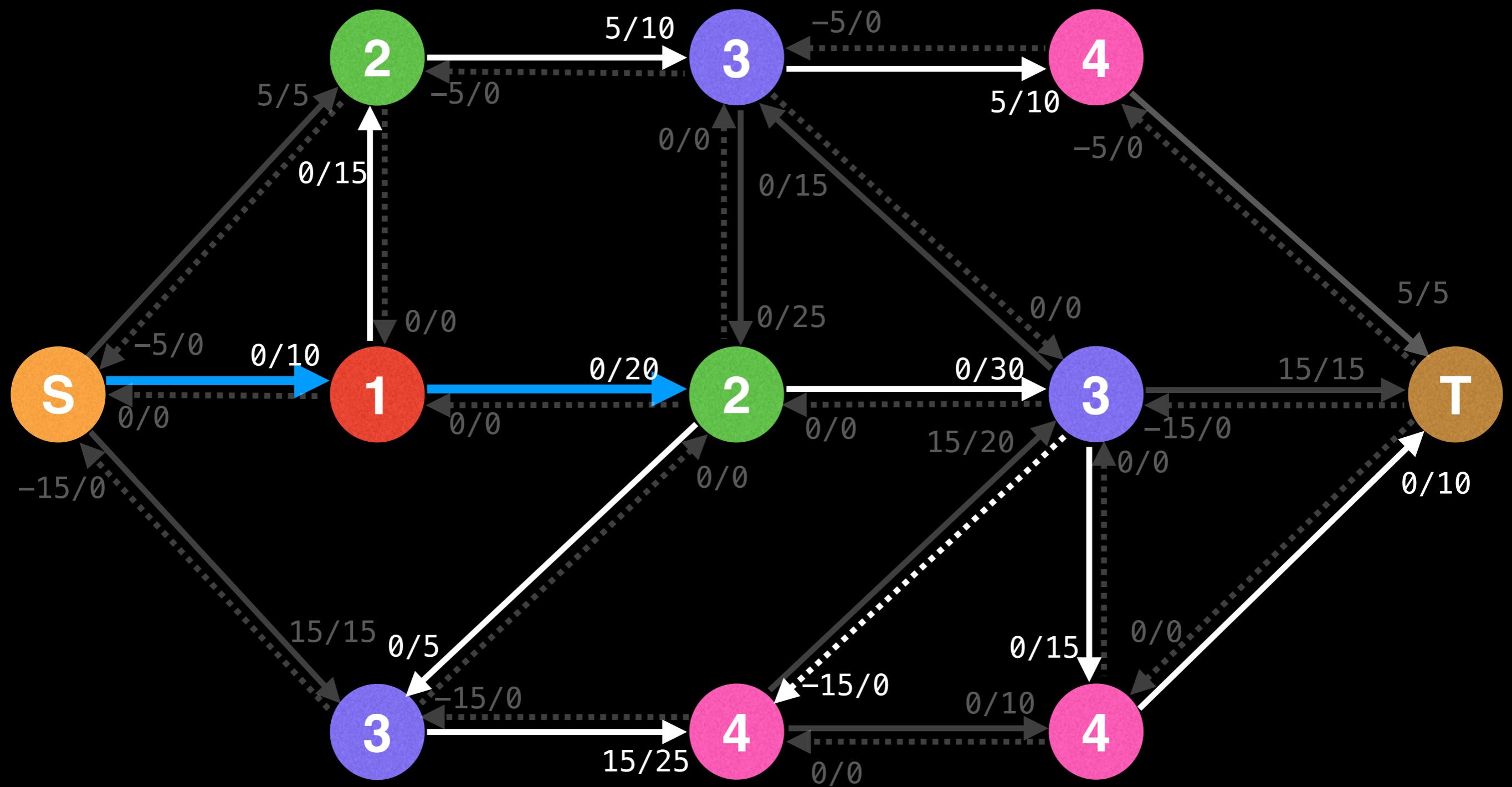
Let's try and find a path from  $s \rightarrow t$ .



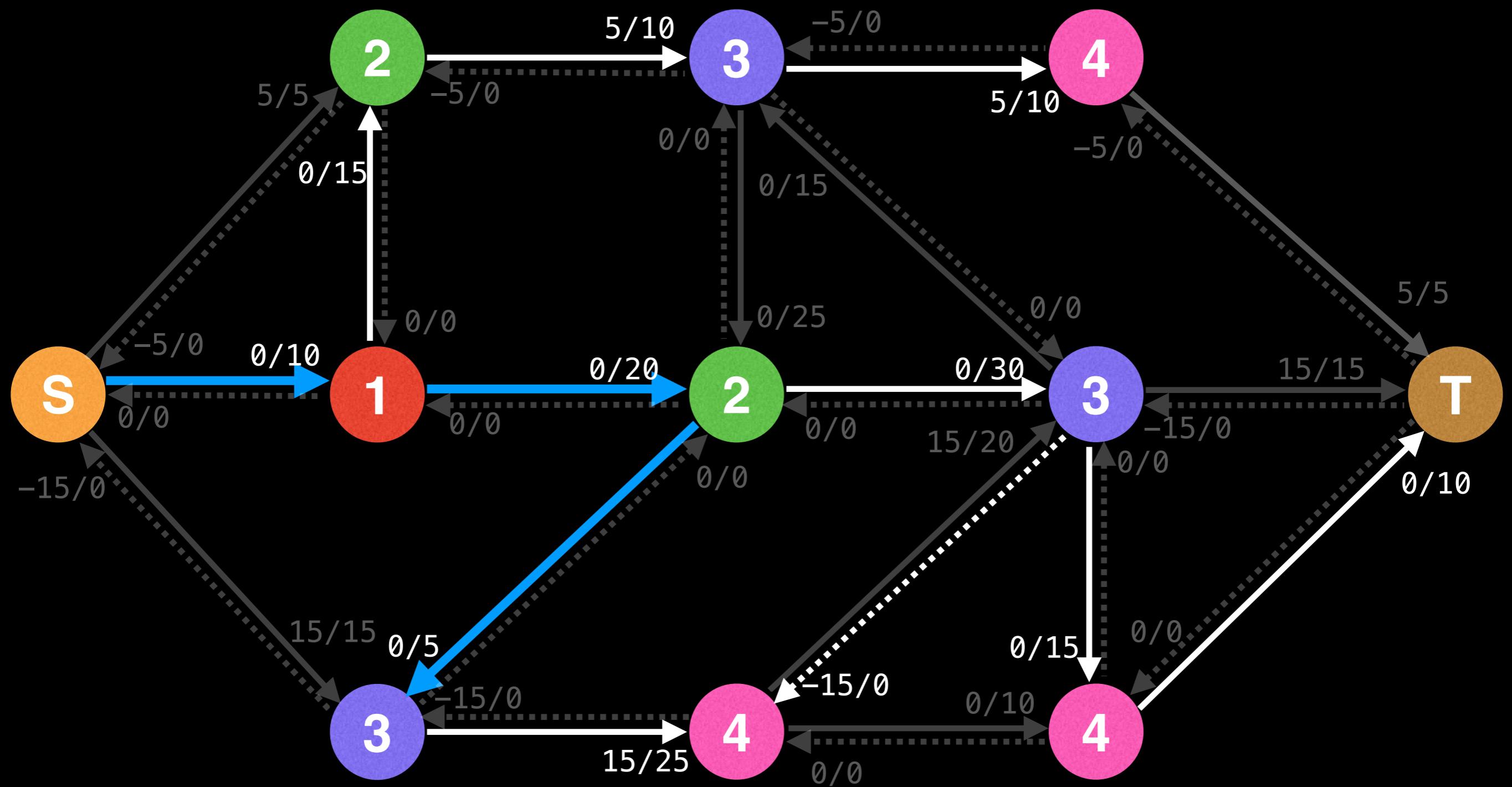
Let's try and find a path from  $s \rightarrow t$ .



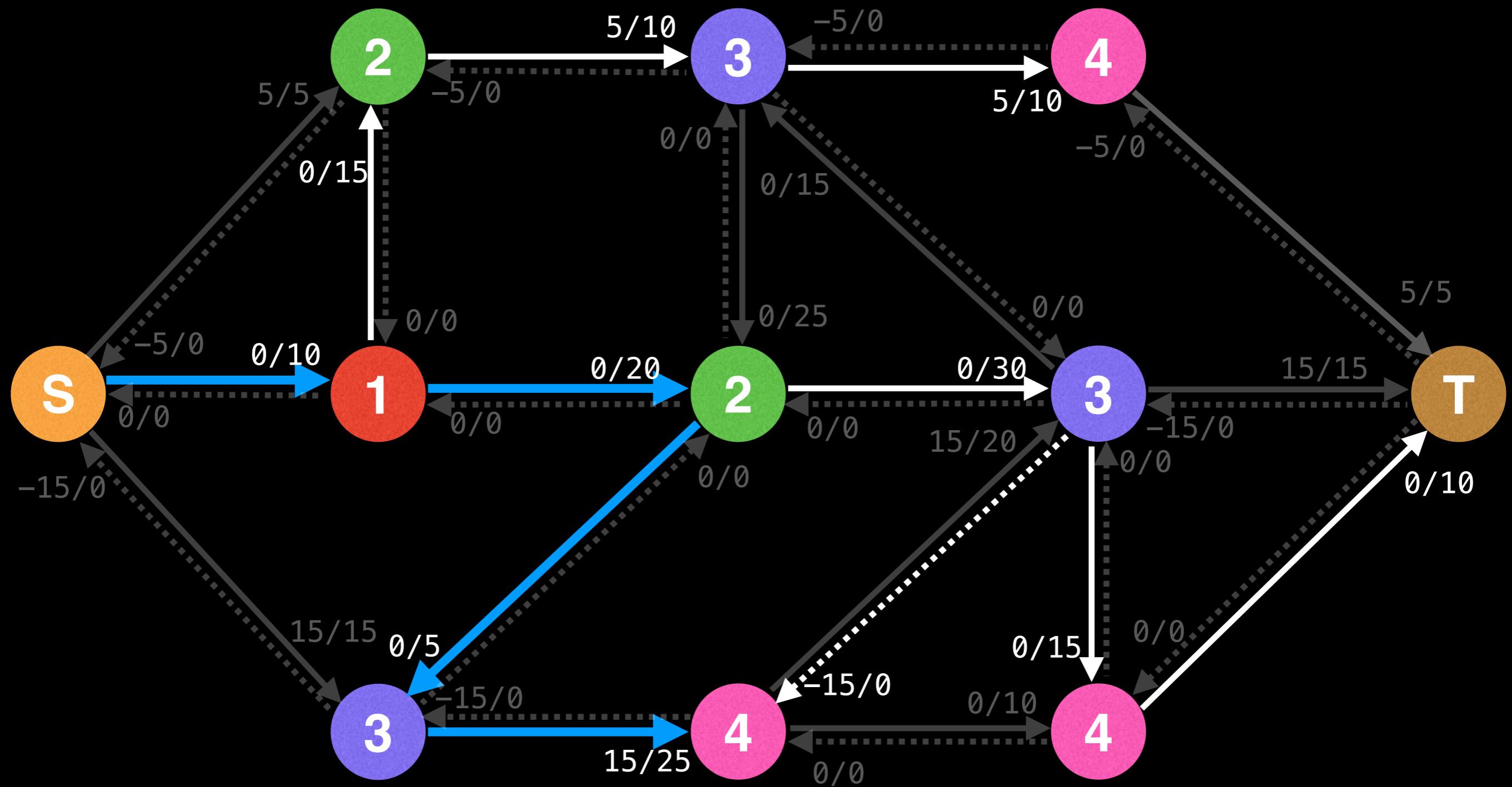
Let's try and find a path from  $s \rightarrow t$ .

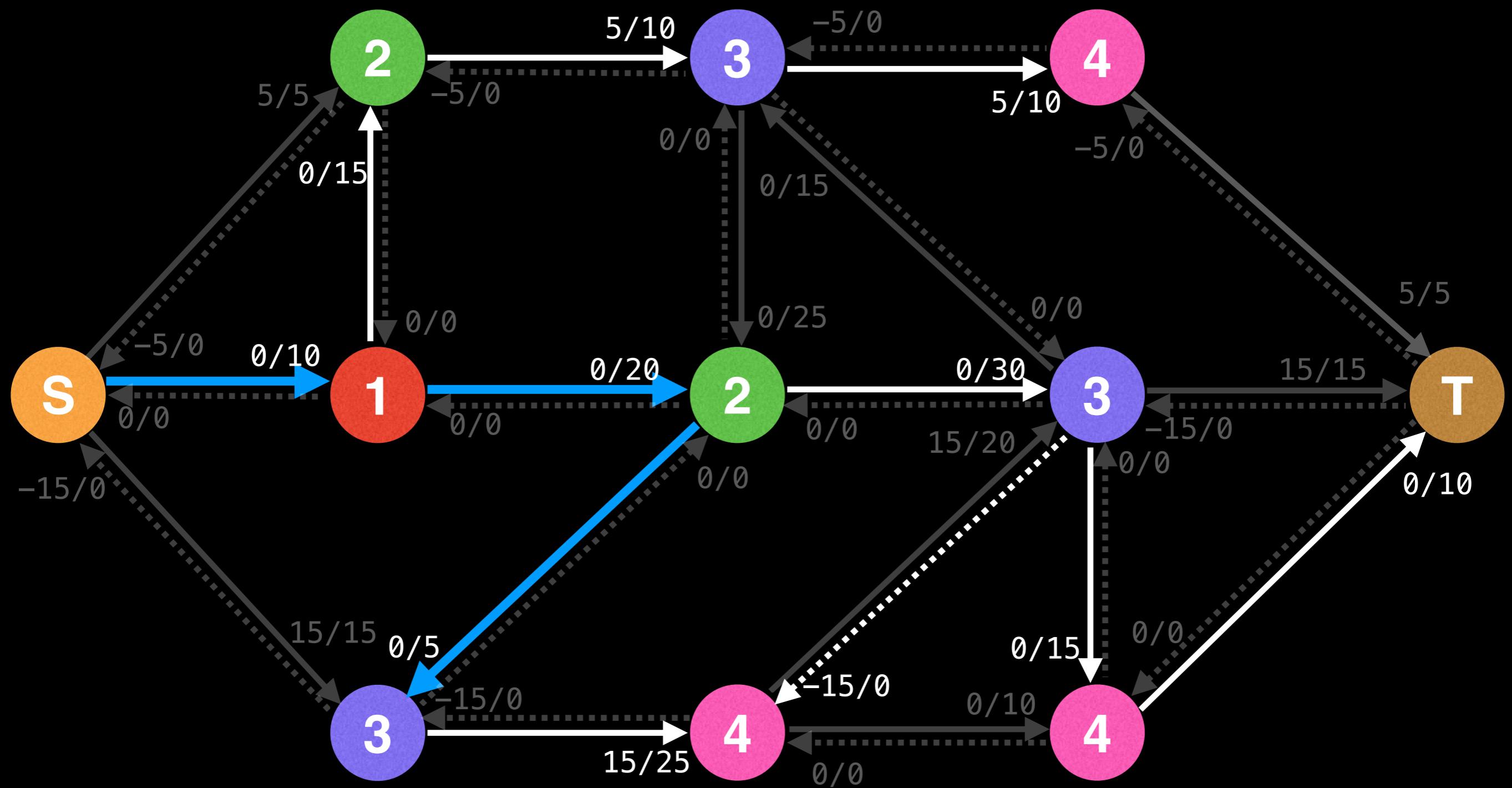


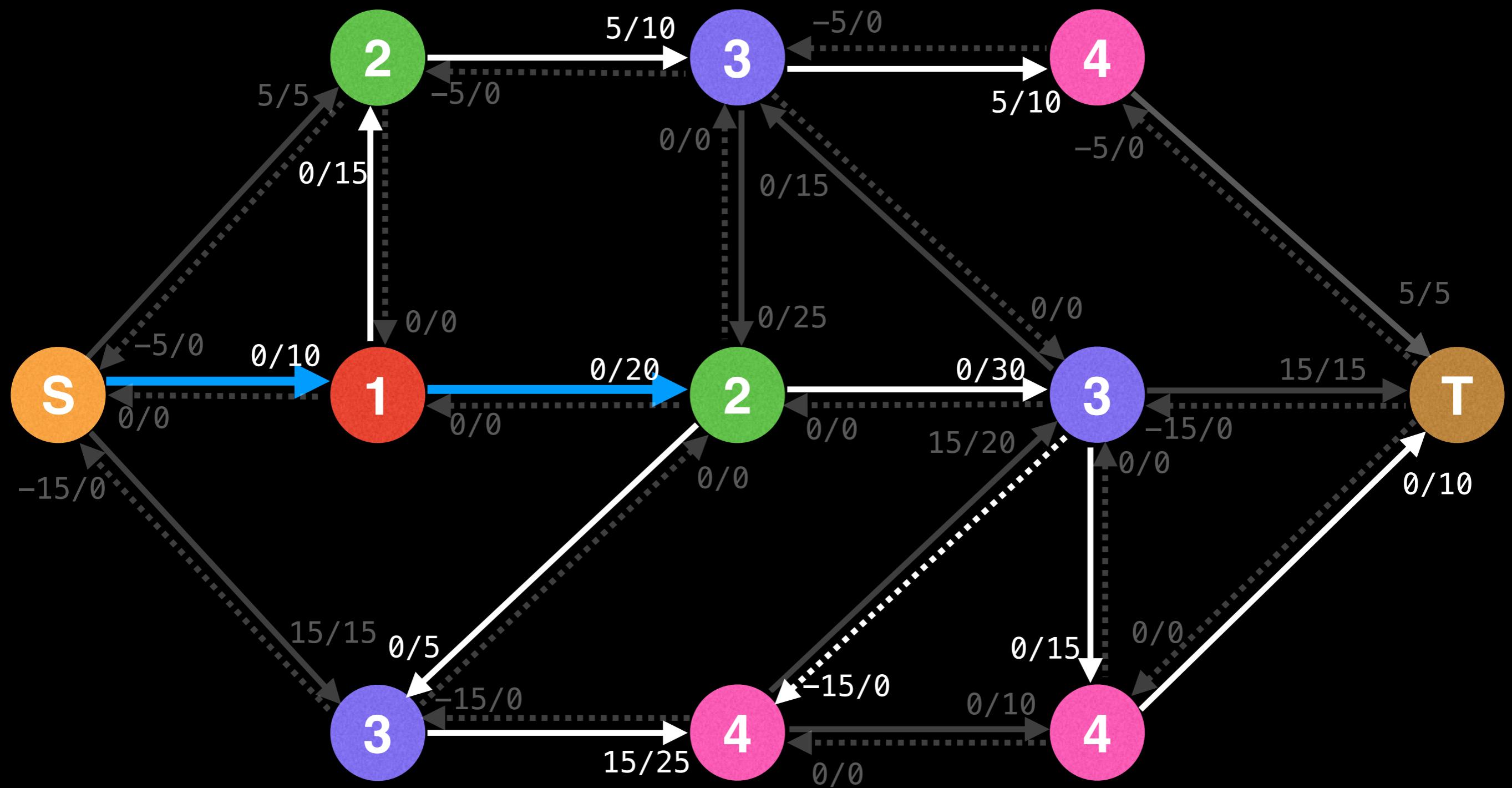
Let's try and find a path from  $s \rightarrow t$ .

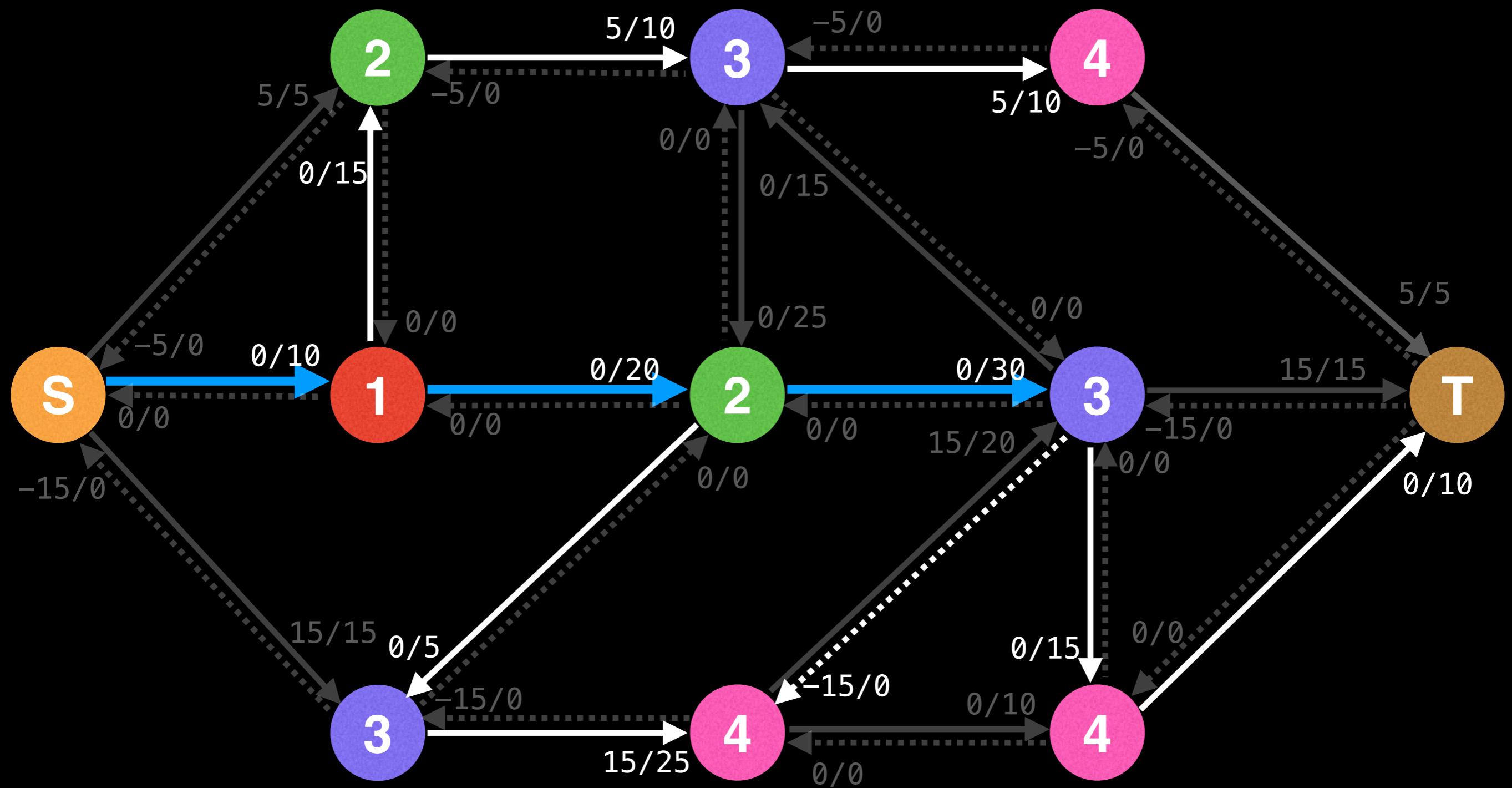


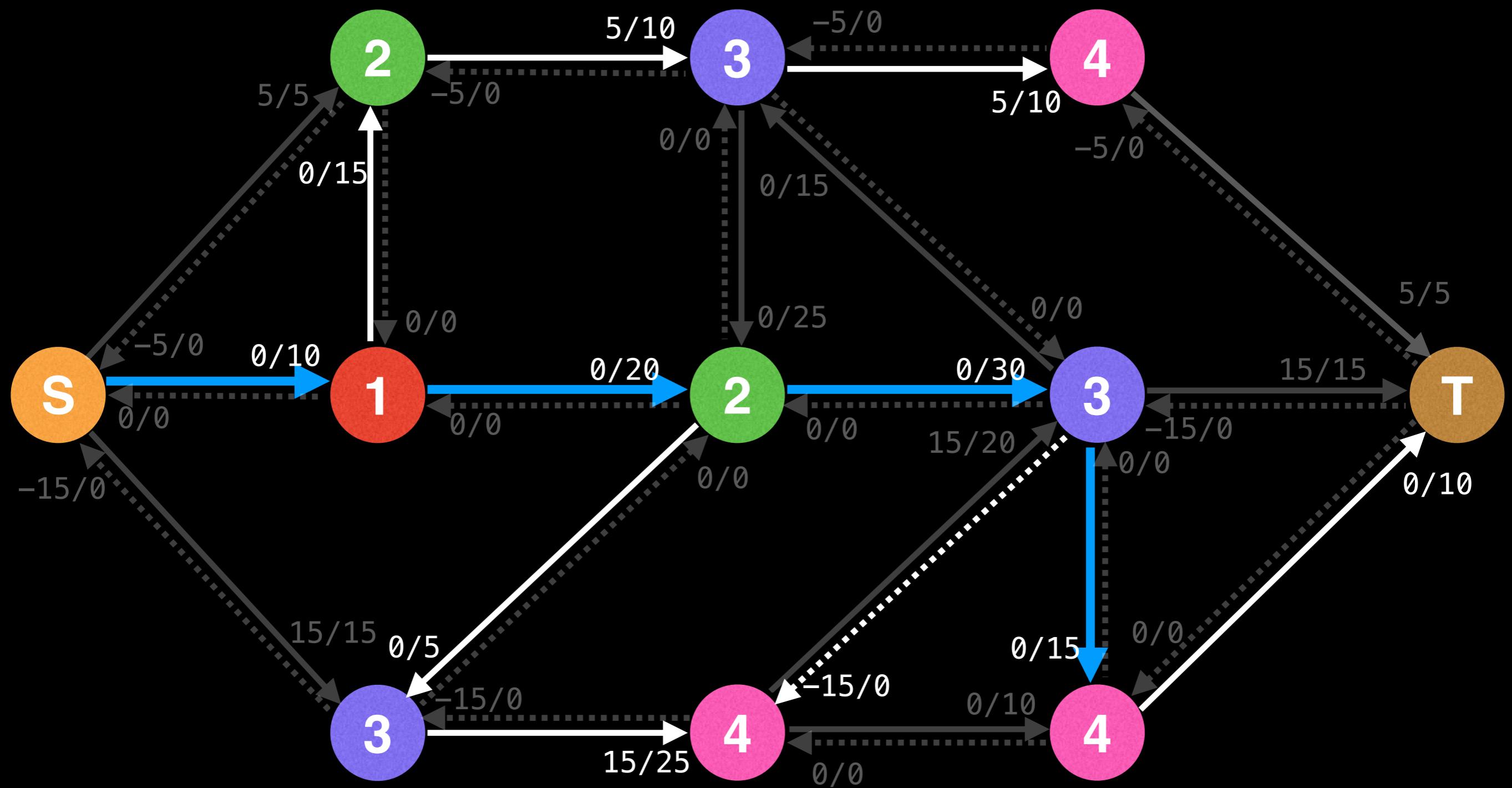
Oops, we reached a **dead end** in our DFS, so backtrack and keep going until we reach the sink.

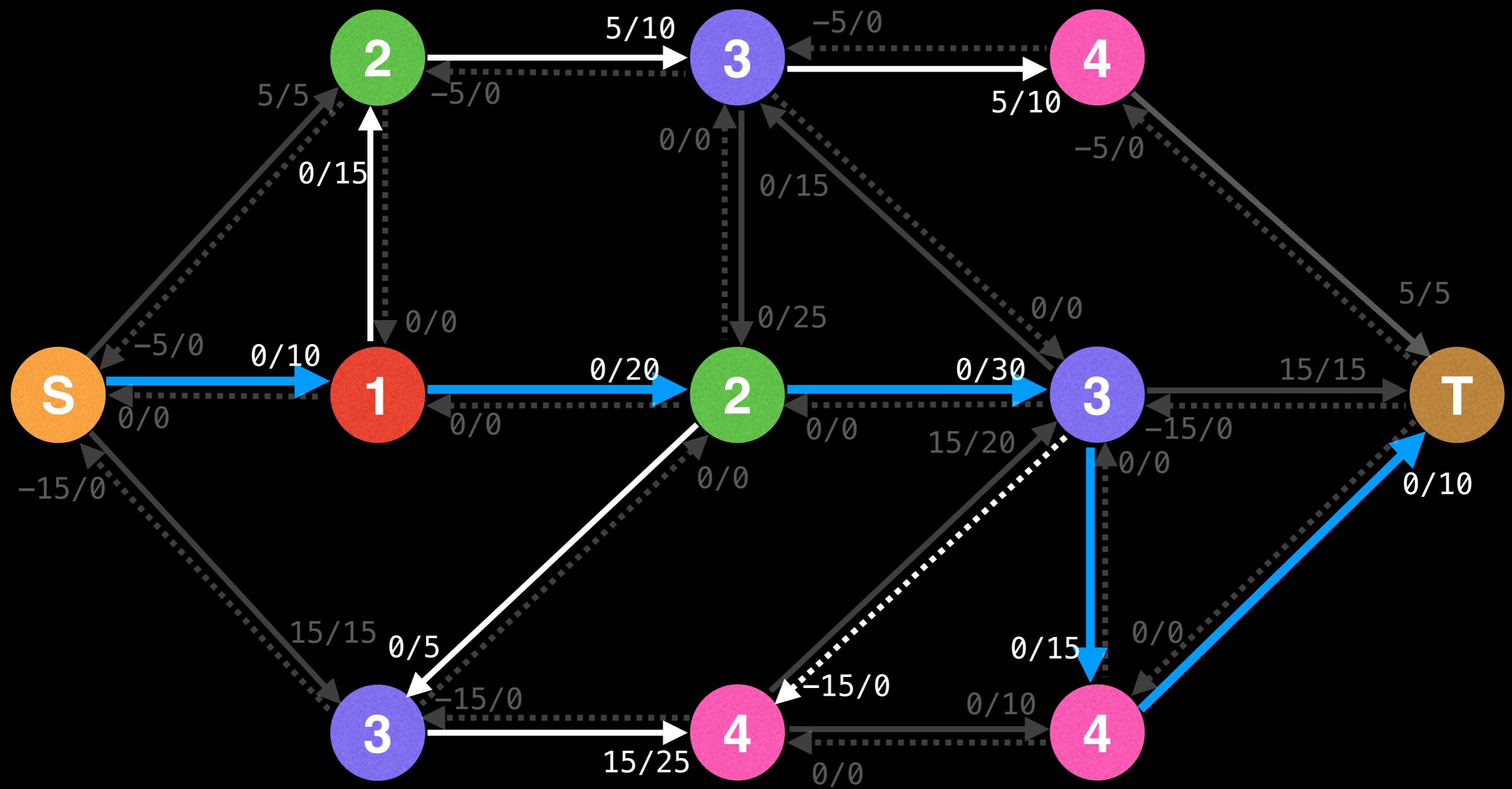




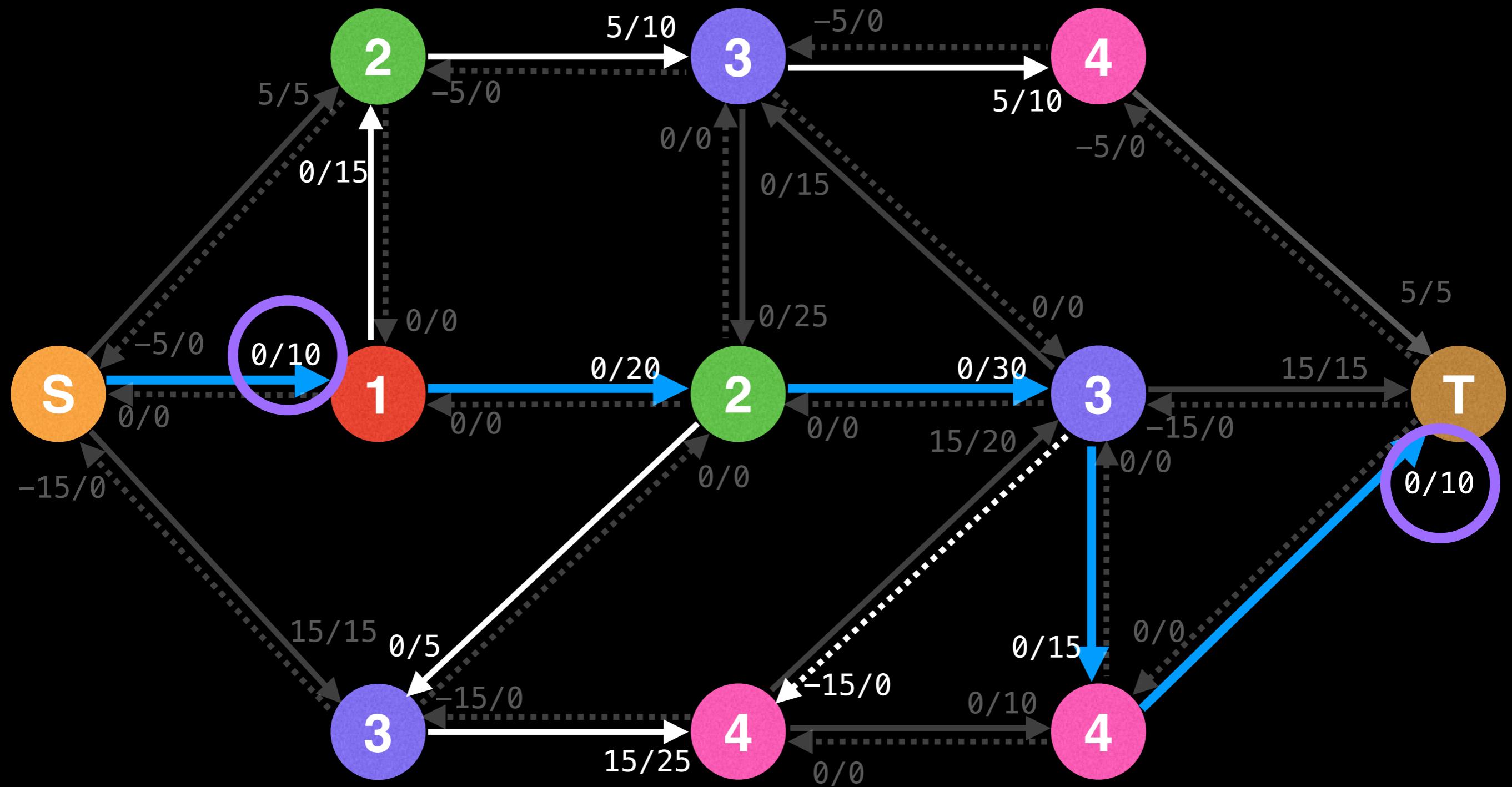




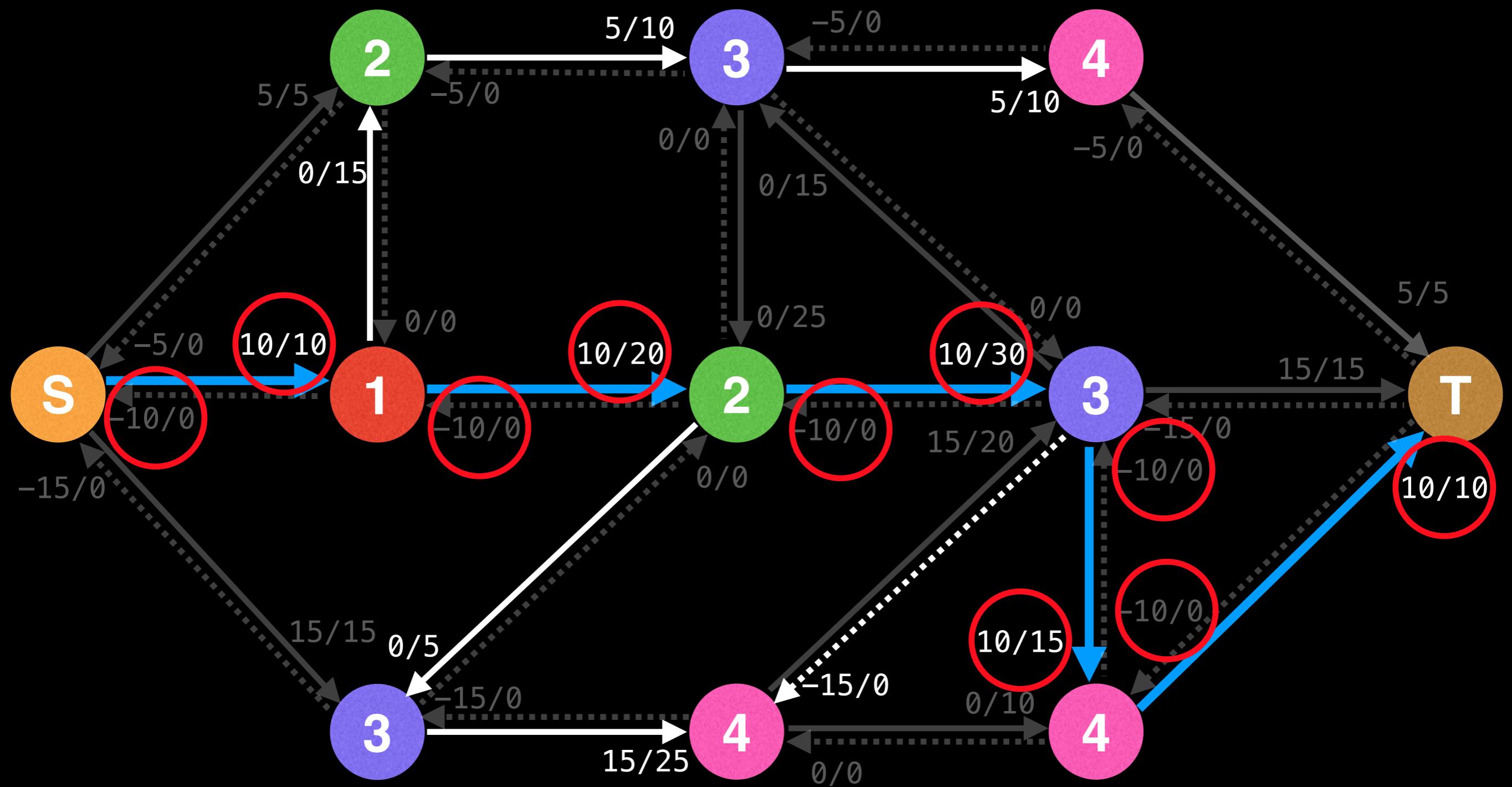


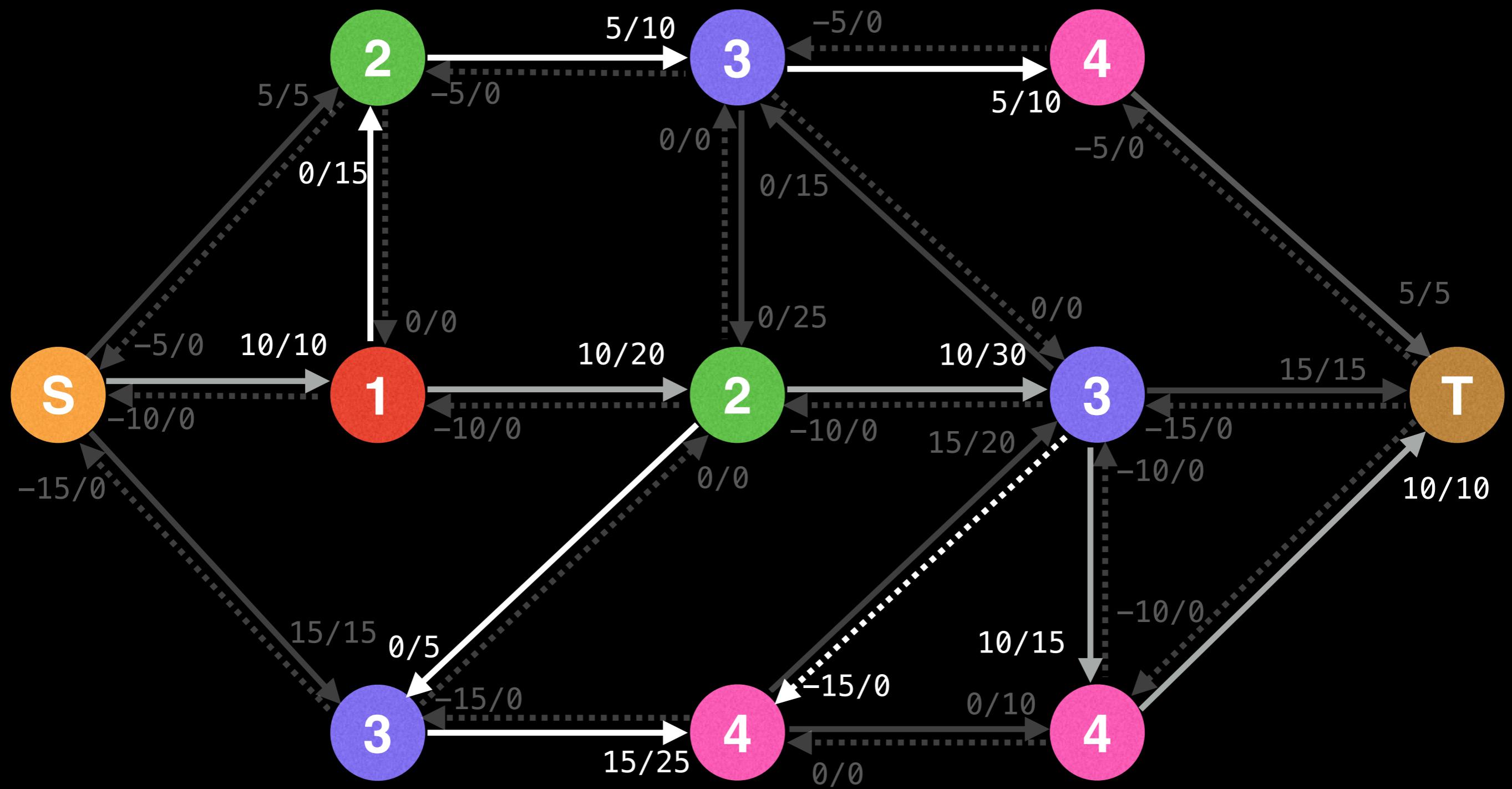


The current path has a bottleneck value of 10  
since  $\min(10-0, 20-0, 30-0, 15-0, 10-0) = 10$ .

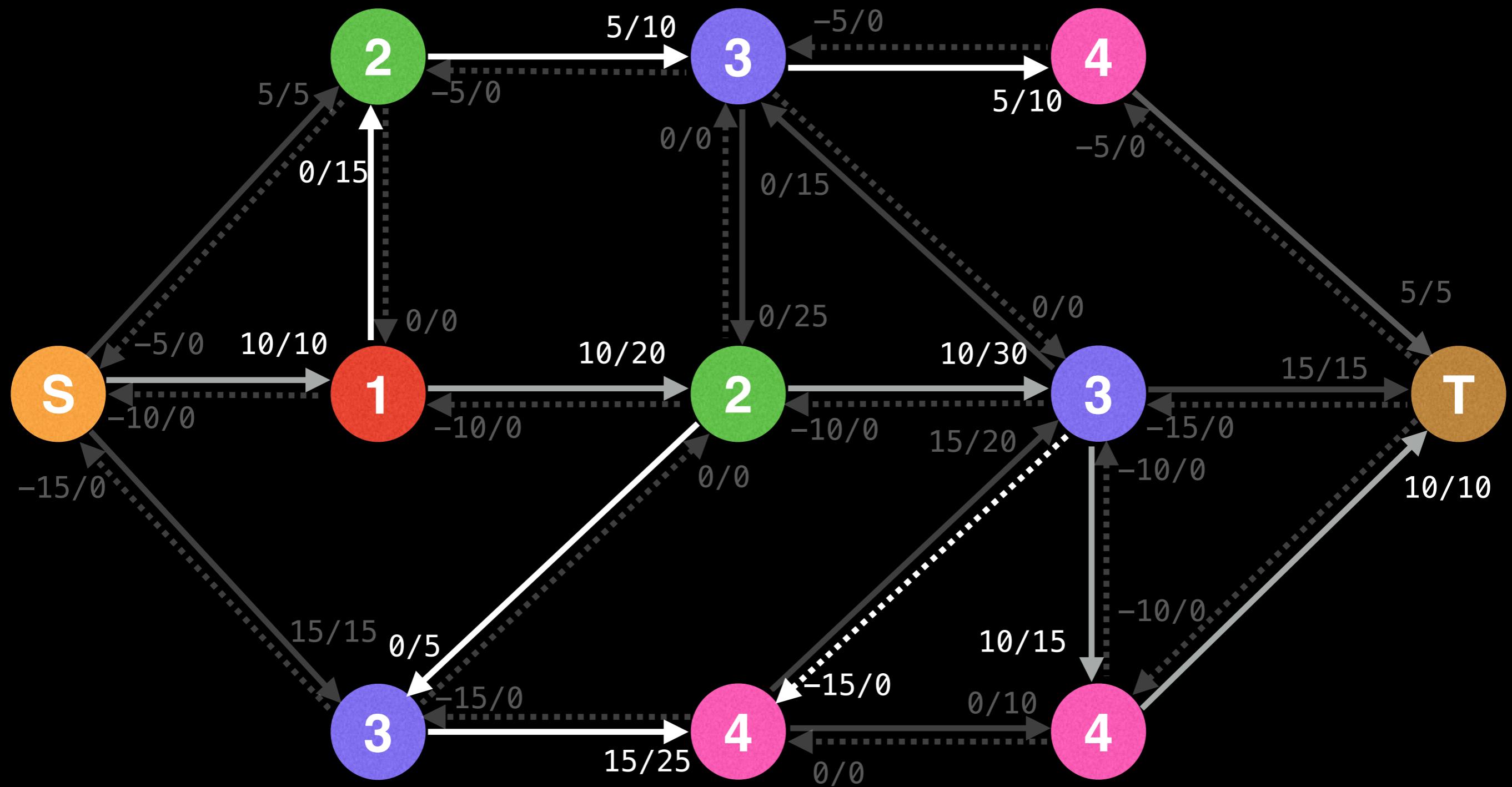


Augment the flow values along the path by 10



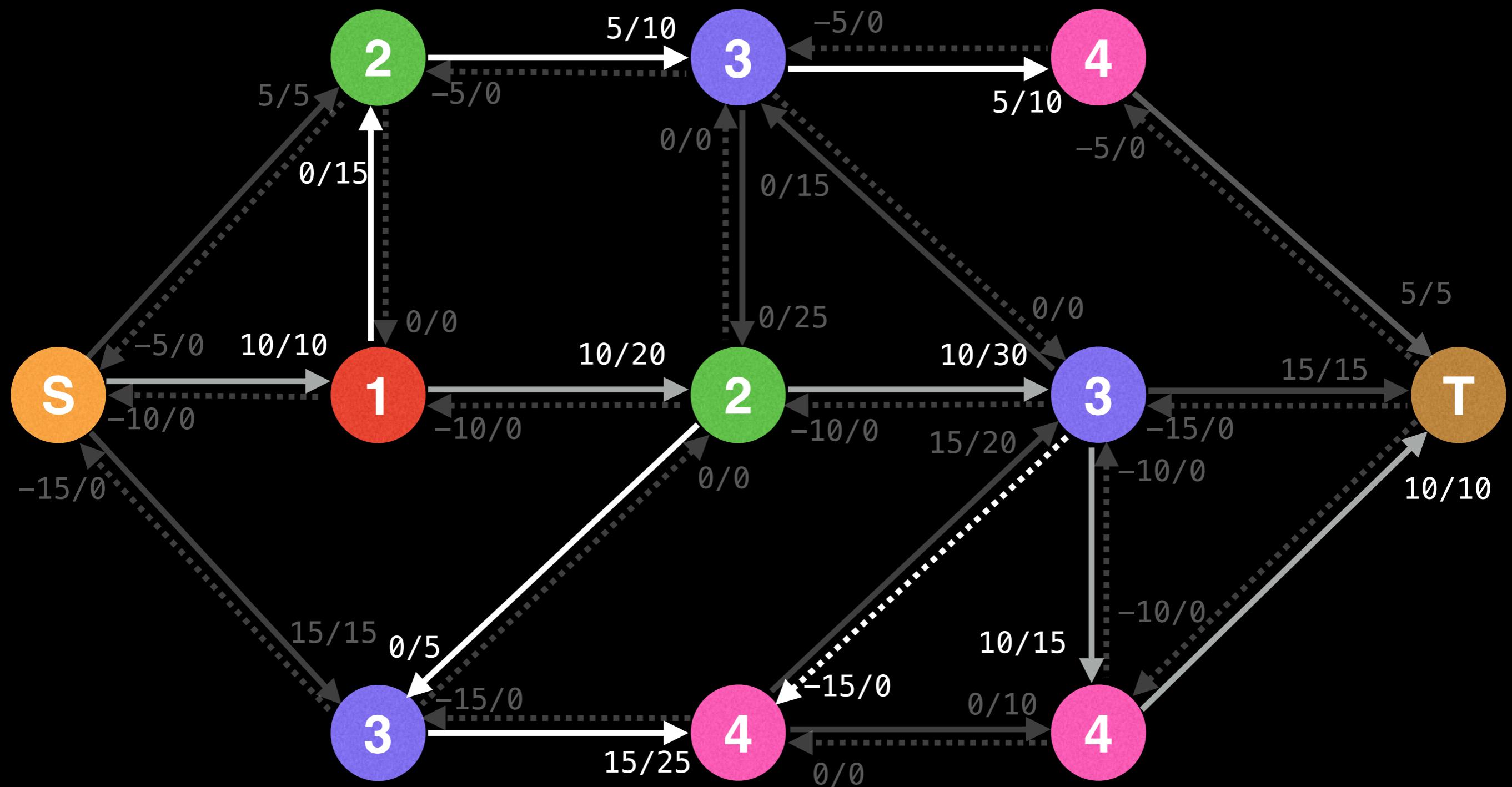


The blocking flow has been reached and no more flow can be pushed through the network, so the algorithm terminates.



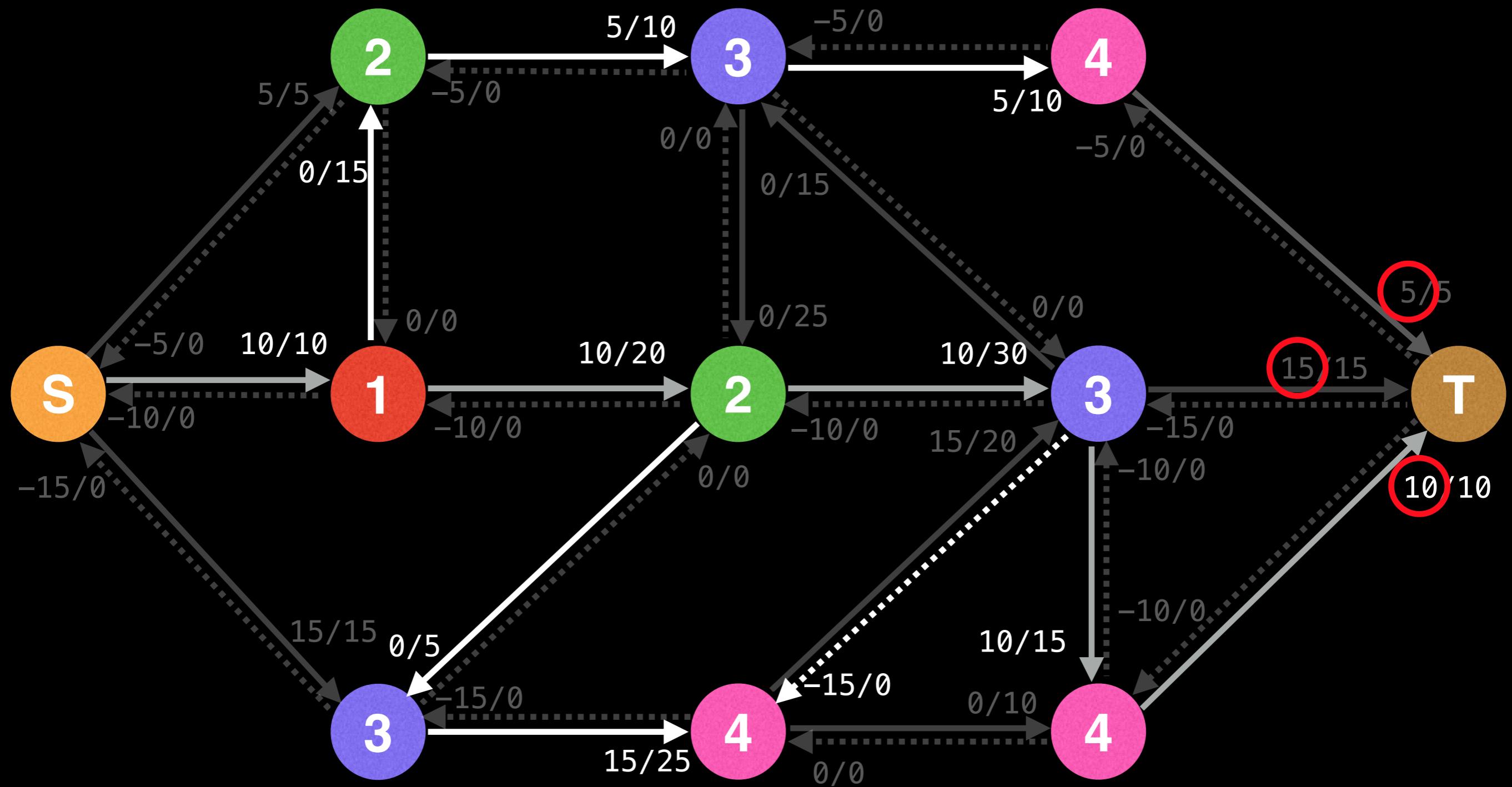
The max flow is the sum of the bottleneck values:

$$\text{Max flow} = 5 + 15 + 10 = 30$$



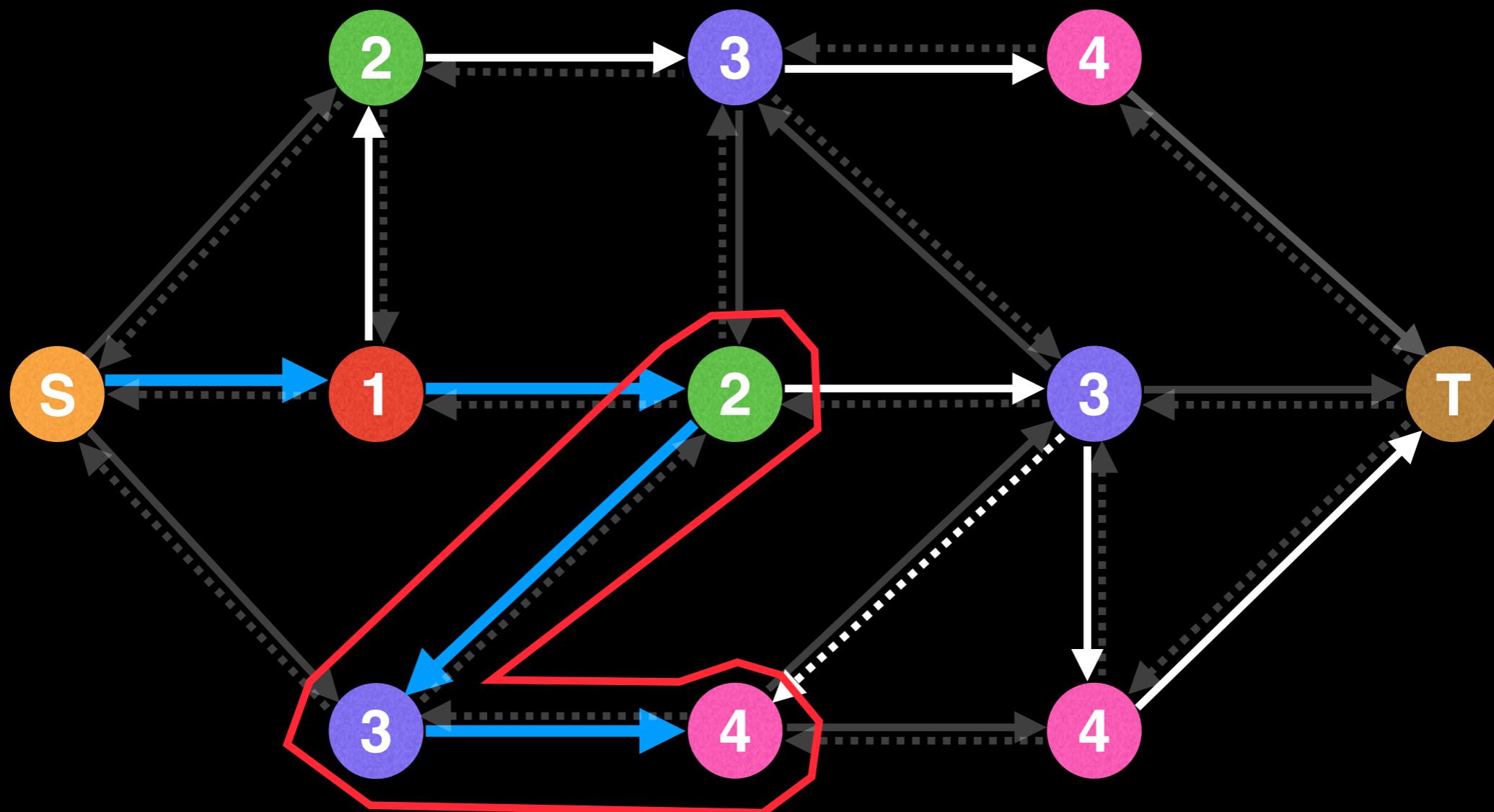
The max flow can also be calculated by looking at the flow values of the edges leading into the sink.

$$\text{Max flow} = 5 + 15 + 10 = 30$$



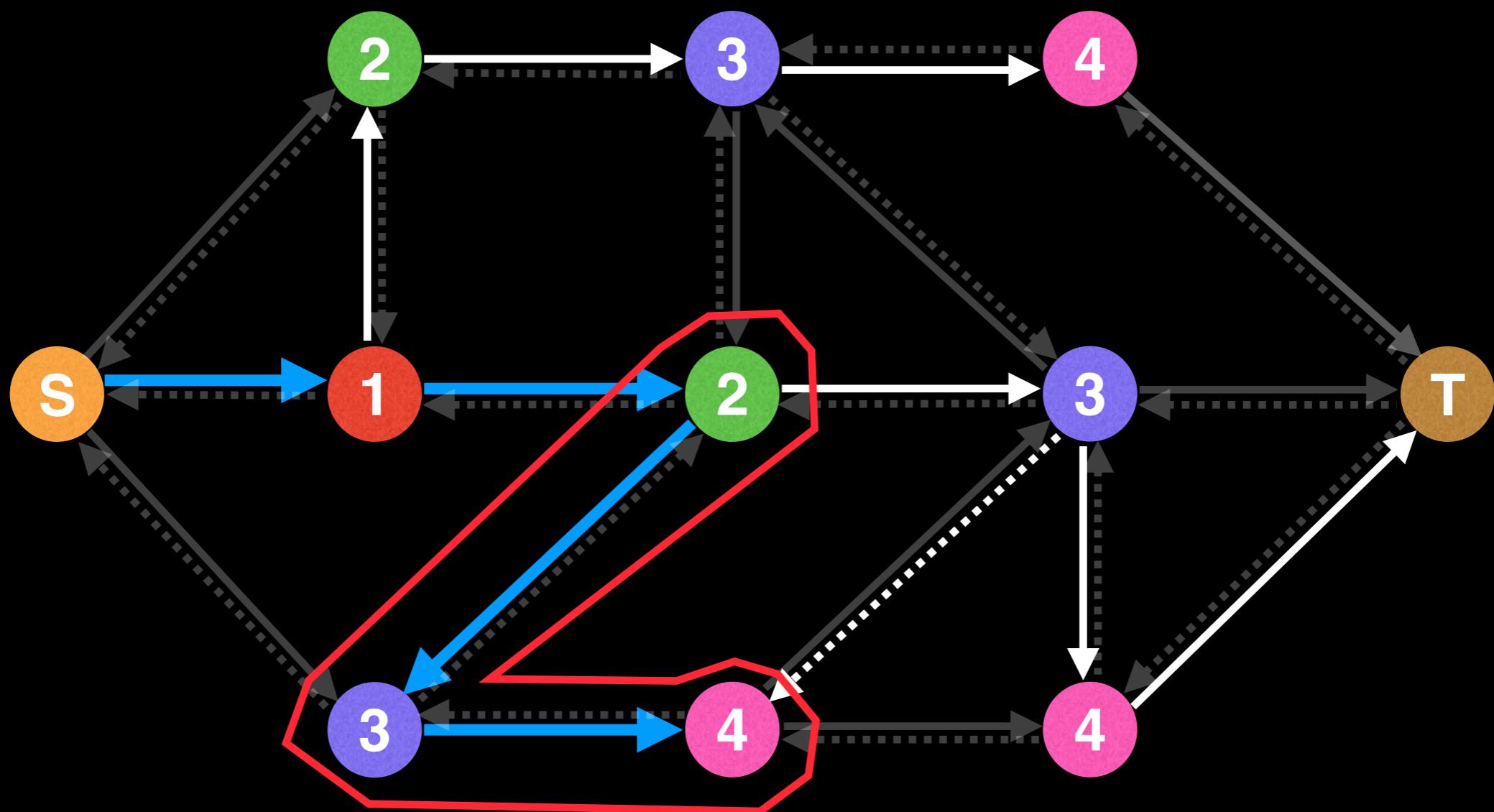
# History & Optimizations

One of the pitfalls of the current implementation of Dinic's algorithm at the moment is that the algorithm may encounter multiple **dead ends** during the DFS phase.



# History & Optimizations

Dead ends are highly undesirable because they extend the length of the journey from the source to the sink, especially if the same dead end is taken multiple times during a blocking flow iteration.



# History & Optimizations

In Dinitz's original paper, he suggested "cleaning" the level graph ridding it of all dead ends before each blocking flow phase.

In 1975, Shimon Even and Alon Itai suggested pruning dead ends when backtracking during the DFS phase.

This trick greatly speeds up (and simplifies) the algorithm because dead ends are only ever encountered once.

## Dinitz' Algorithm: The Original Version and Even's Version

Yefim Dinitz

Dept. of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel  
dinitz@cs.bgu.ac.il

**Abstract.** This paper is devoted to the max-flow algorithm of the author: to its original version, which turned out to be unknown to non-Russian readers, and to its modification created by Shimon Even and Alon Itai; the latter became known worldwide as 'Dinic's algorithm'. It also presents the origins of the Soviet school of algorithms, which remain unknown to the Western Computer Science community, and the substantial influence of Shimon Even on the fortune of this algorithm.

### 1 Introduction

The reader may be aware of the so called "Dinic's algorithm" [4]<sup>1</sup>, which is one of the first (strongly) polynomial max-flow algorithms, while being both one of the easiest to implement and one of the fastest in practice. This introduction discusses two essential influences on its fortune: the first supported its invention, and the other furthered its publicity, though changing it partly.

The impact of the late Shimon Even on Dinic's algorithm dates back to 1975. You may ask how a person in Israel could influence something in the former USSR, when it was almost impossible both to travel abroad from the USSR and to publish abroad or even to communicate with the West? This question will clear up after first considering the impact made by the early Soviet school of computing and algorithms.

The following anecdote sheds some light on how things were done in the USSR. Shortly after the "iron curtain" fell in 1990, an American and a Russian, who had both worked on the development of weapons, met. The American asked: "When you developed the Bomb, how were you able to perform such an enormous amount of computing with your weak computers?". The Russian responded: "We used better algorithms."

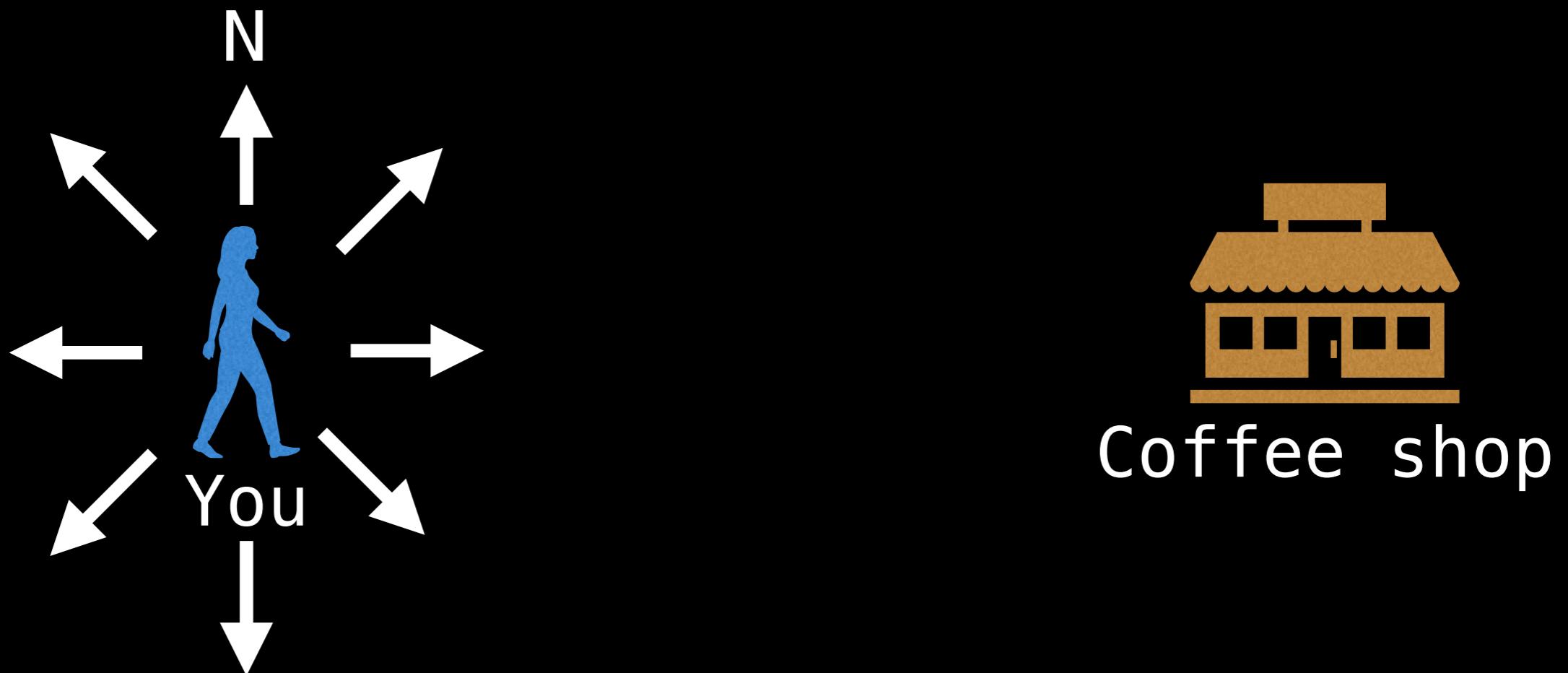
This was really so. Russia had a long tradition of excellence in Mathematics. In addition, the usual Soviet method for attacking hard problems was to combine pressure from the authorities with people's enthusiasm. When Stalin decided to develop the Bomb, many bright mathematicians, e.g., Izra'il Gelfand and my first Math teacher, Alexander Kronrod, put aside their mathematical studies and delved deeply into the novel area of computing. They have assembled teams of talented people, and succeeded. The teams continued to grow and work on the theory and practice of computing.

The supervisor of my M.Sc. thesis was George Adel'son-Vel'sky, one of the fathers of Computer Science. Among the students in his group at that time were M. Kronrod (one of the future "Four Russians", i.e. the four authors of [3]), A. Karzanov (the future author of

<sup>1</sup> In this paper, the references are given in chronological order.

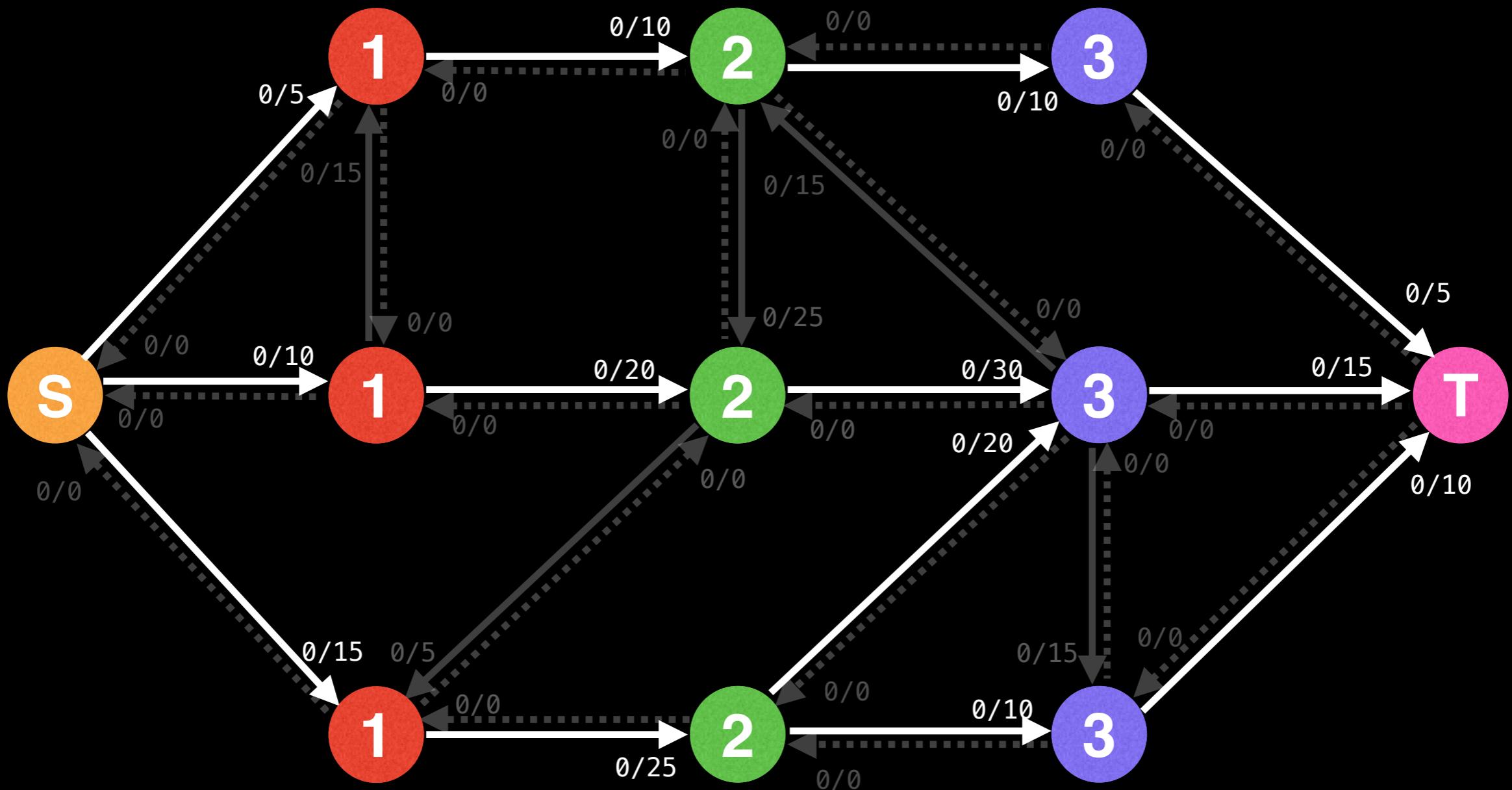
# Summary

A good pathfinding heuristic which ensures that we **continuously make progress** towards a goal will greatly speed up an algorithm.



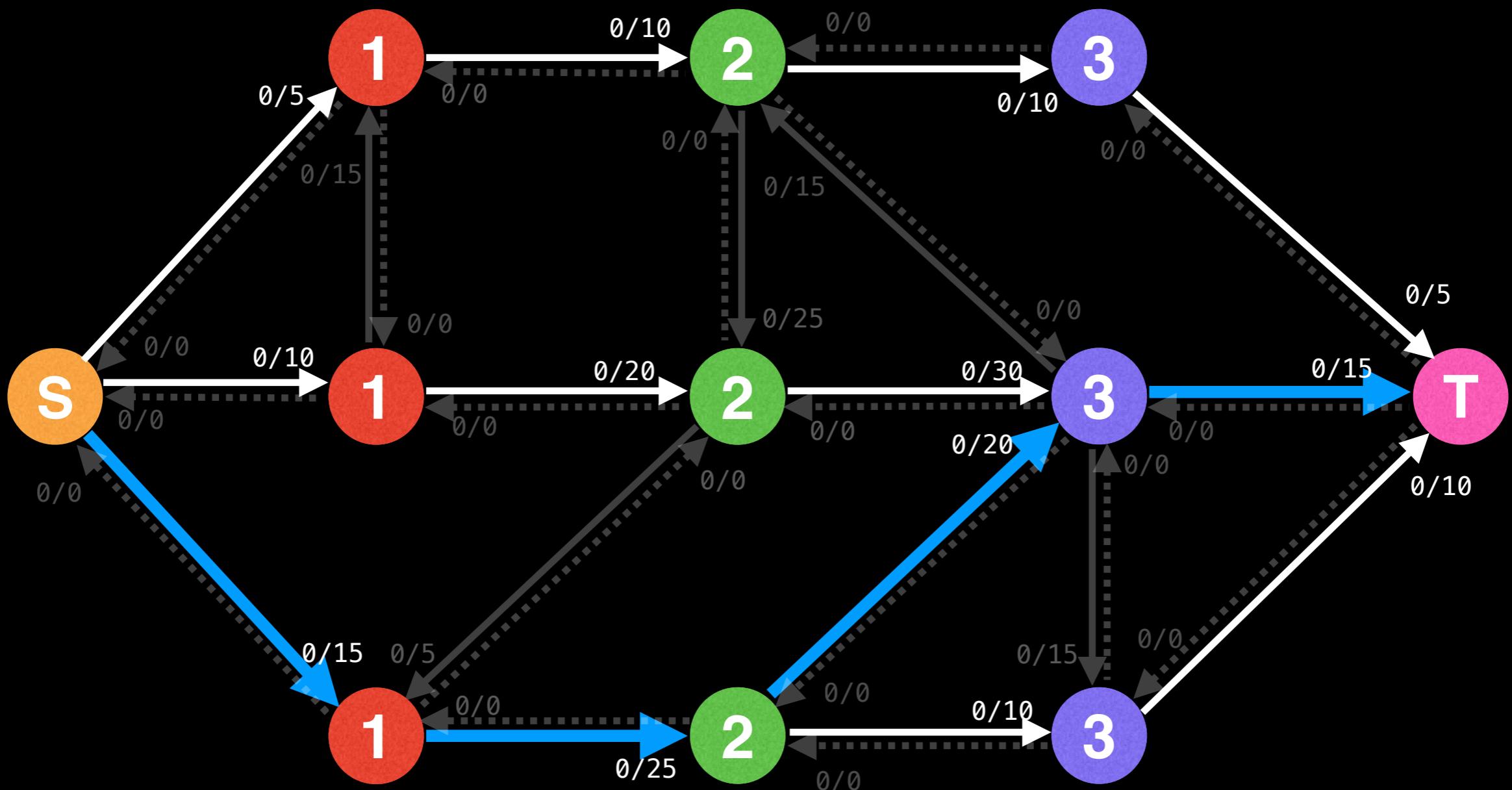
# Summary

Dinic's algorithm uses a BFS to construct a level graph which directs edges towards the sink.



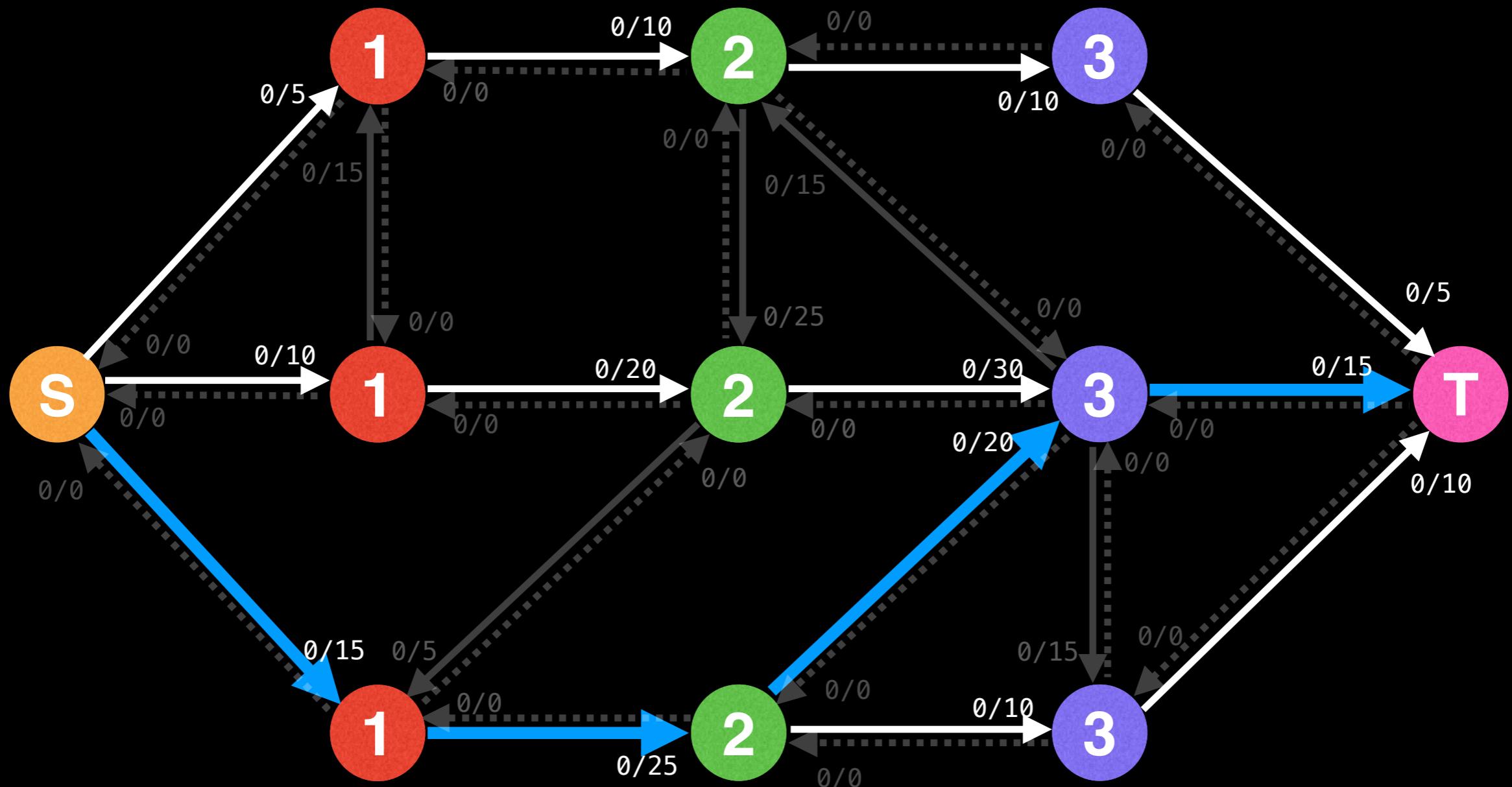
# Summary

In the next phase of the algorithm, multiple DFSs are run on the level graph until a blocking flow is reached.



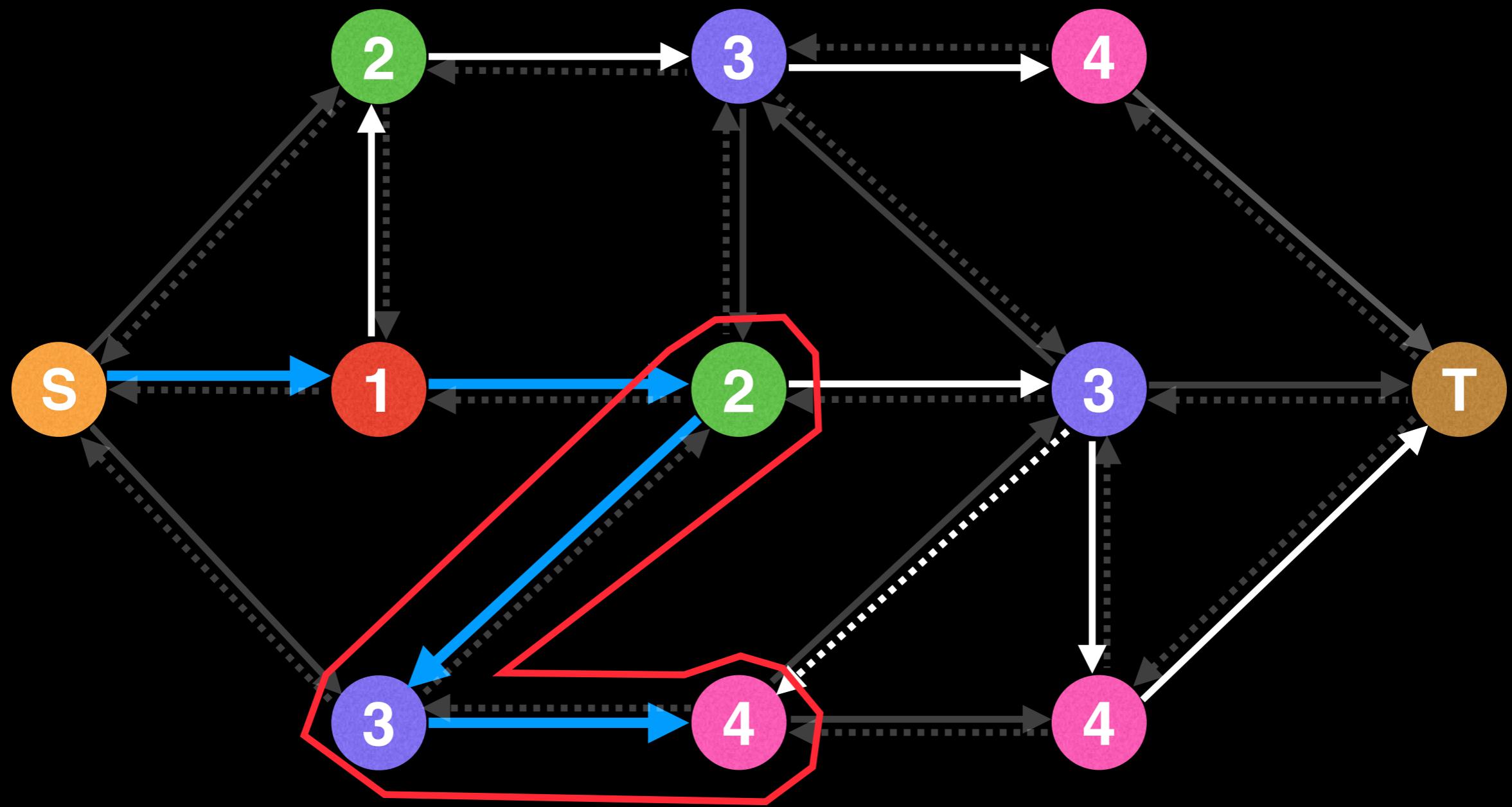
# Summary

The process of rebuilding the level graph and finding the blocking flow is repeated until no more augmenting paths exist and the max flow is found.



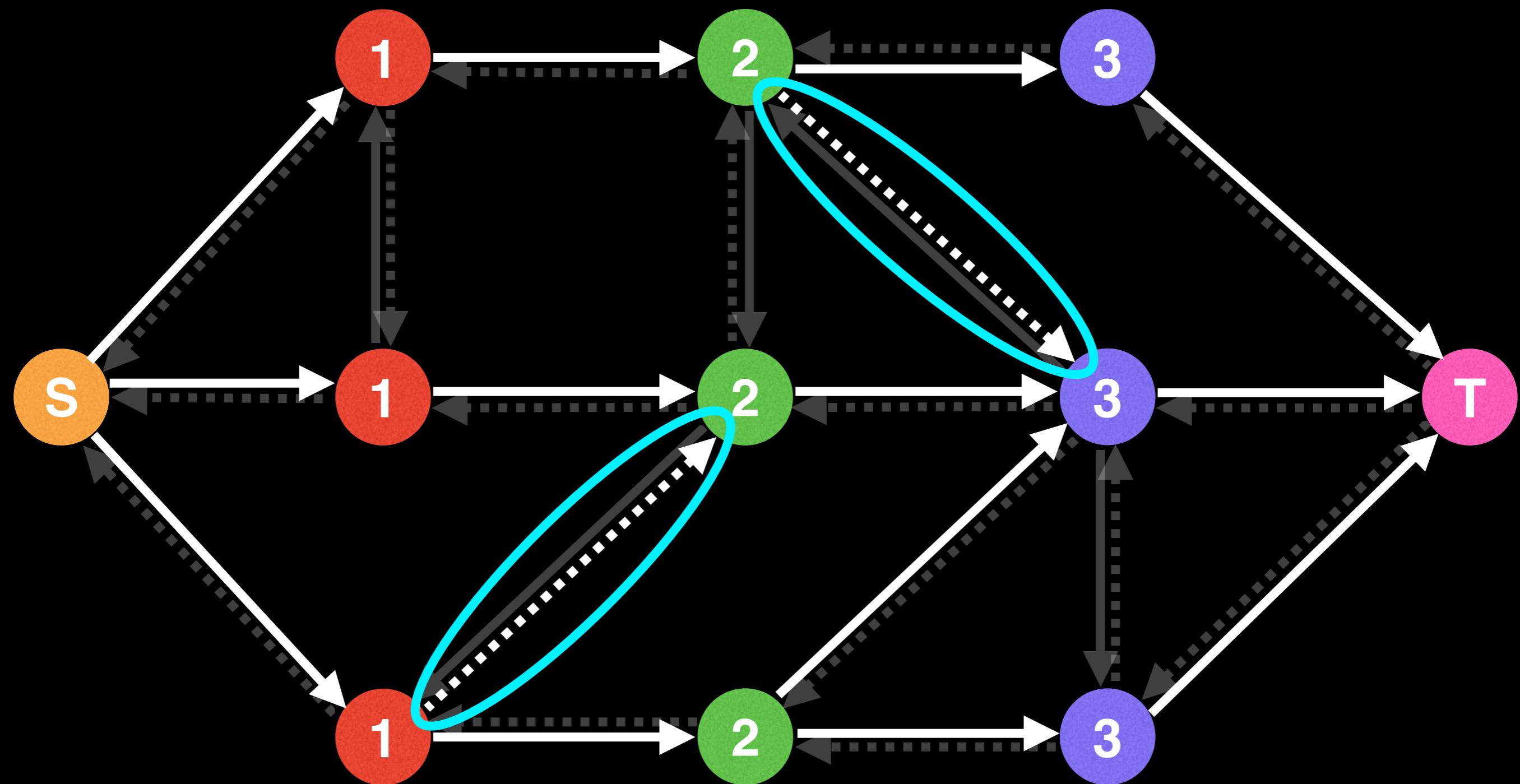
# Summary

A critical optimization of Dinitz's algorithm is pruning dead ends so that you do not encounter them again.

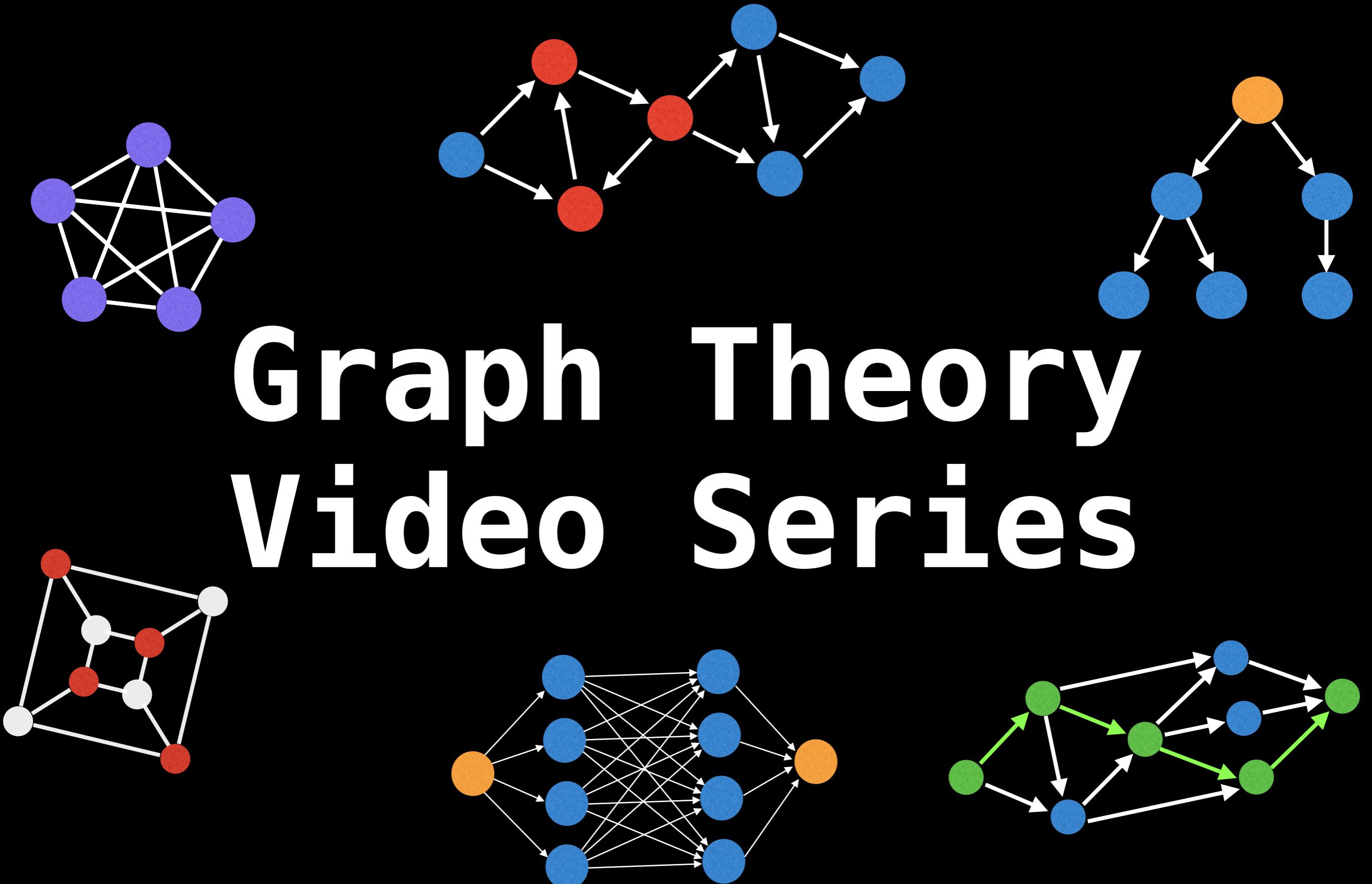




# Network Flow: Dinic's Algorithm



# Graph Theory Video Series



# Network Flow: Dinic's Algorithm source code

William Fiset

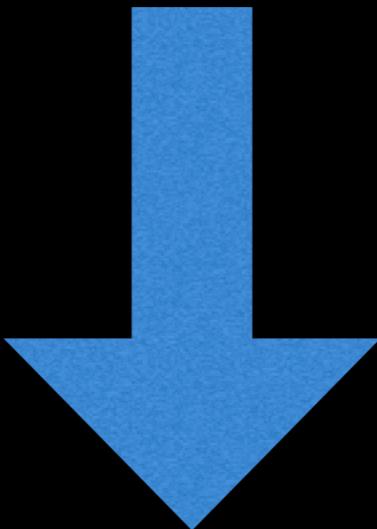
Previous video explaining  
Dinic's algorithm:

# Source Code Link

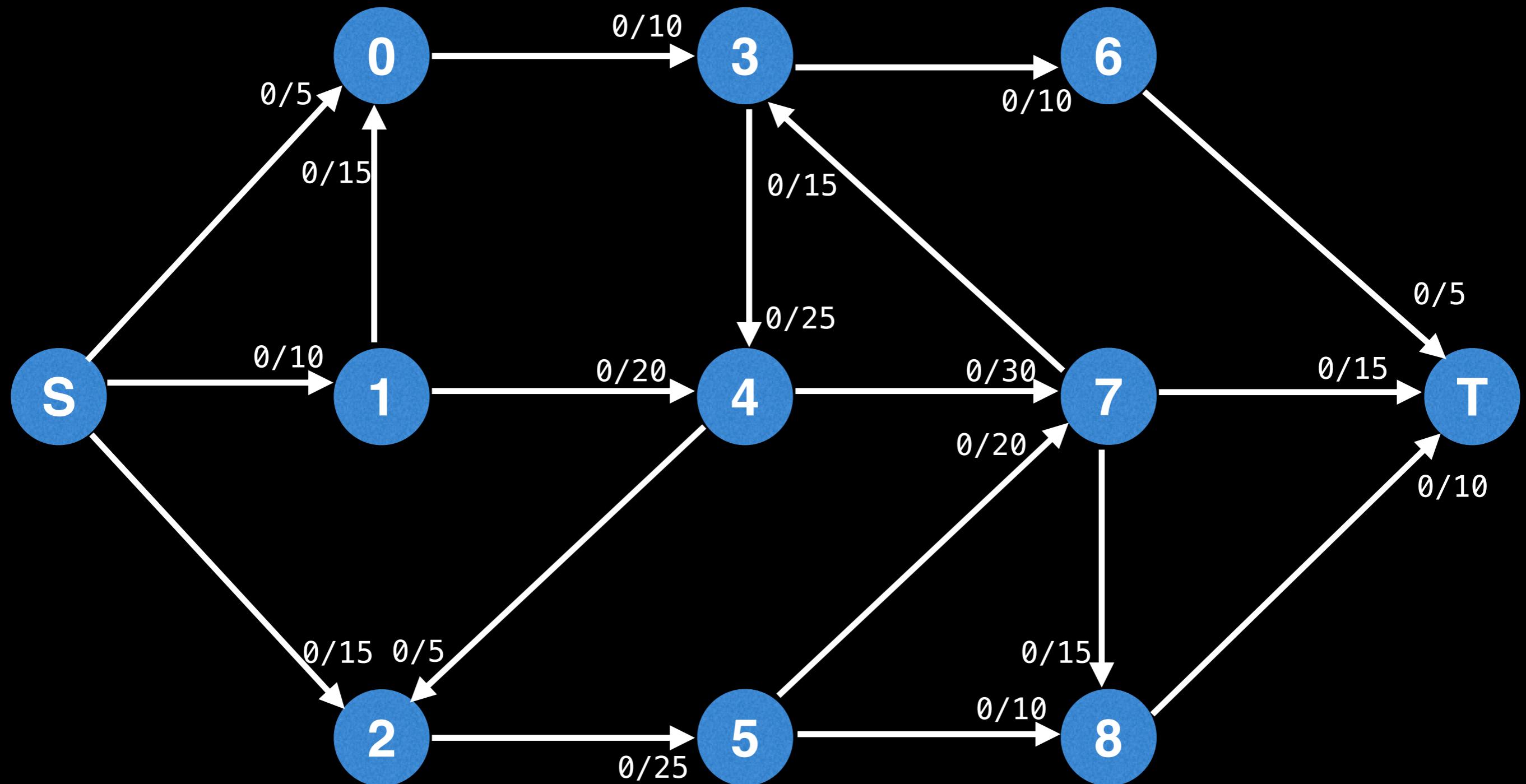
Implementation source code can  
be found at the following link:

[github.com/williamfiset/algorithms](https://github.com/williamfiset/algorithms)

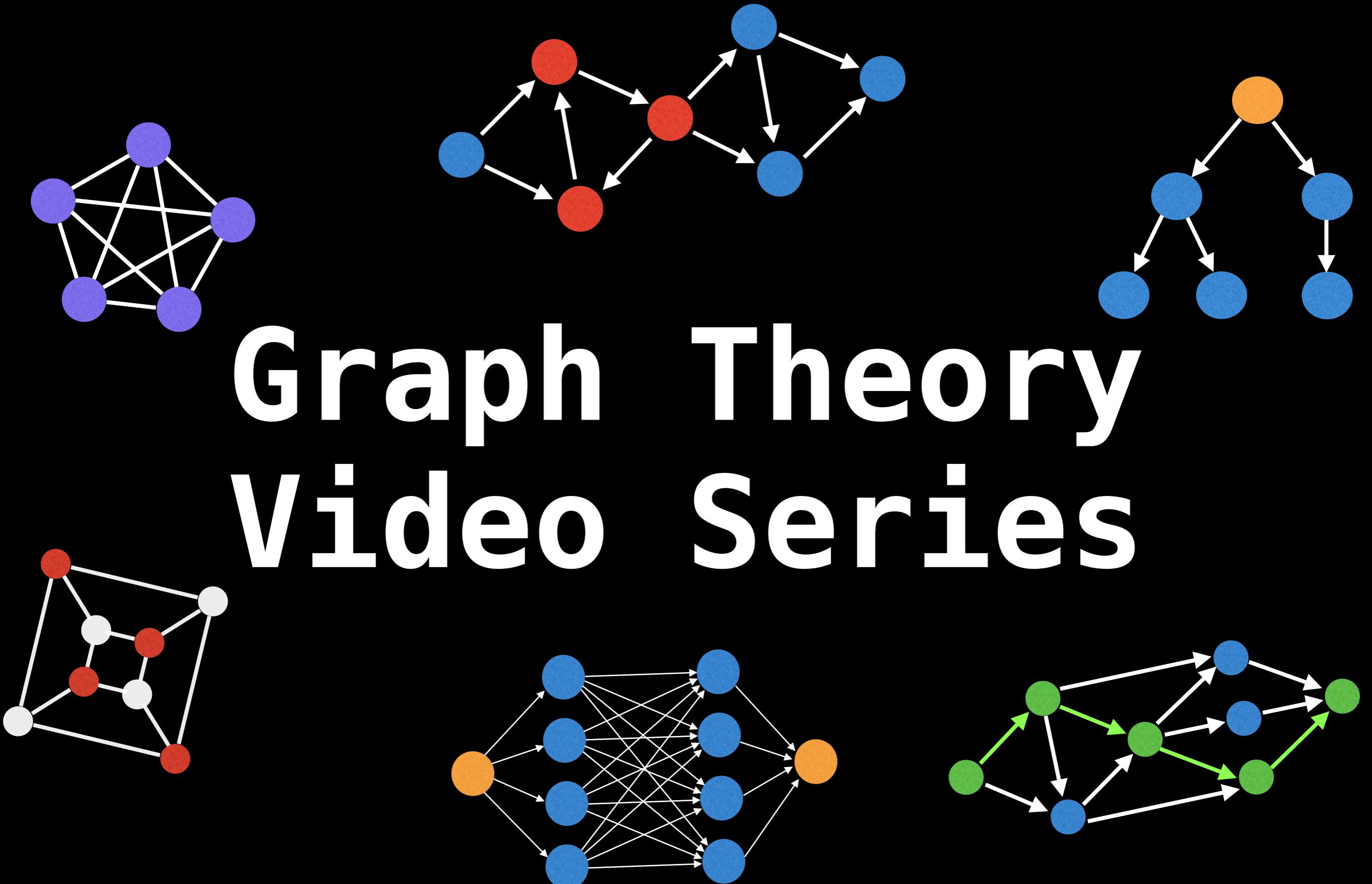
Link in the description:



Let's run through an example of using Dinic's algorithm to find the maximum flow on the following flow graph.



# Graph Theory Video Series



# Network Flow: Mincost Maxflow

William Fiset

# TODO list

What's up with negative cycles?

How to draw an edge with cap and cost?

Mention residual edges have negative cost  
figure out which examples to use.

Implement dummy S->T max flow + negative  
cycle cancelation.

Mention most of the time edge cost are  $> 0$   
and the problem is solved with SSSP  
algorithms.

What is the bigger picture here? What  
general class of problems is MCMF solving?

We have these graphs with cap and cost on the edges. rn I know how to solve this when the cost is  $\geq 0$  because it's a reduction to SSSP which is awesome and really what we care about most of the time. But, when cost  $< 0$  shit goes crazy because now we can have negative cycles – oh shit right. To handle these we need to use Klien's cycle canceling method (analogous to FF for max flow). Basically Kliens finds cycles and augments the flow as per my understanding. Now finding these negative cycles is a hard problem (computationally intensive that is) so some fancy Network Simply algorithm can find and cancel negative cycles on the fly. This is the boss of all graph algorithms.

A mincost flow graph cannot contain a directed negative cycle, otherwise the optimal cost is unbounded.

It is safe to assume all edge costs are non-negative because we don't lose anything by doing so, only need to transform. Also edges must be integral for this to work.

# Klein's Cycle Canceling Method

illustrate that there can be multiple graphs with the same “flow” but different costs?

# TODO list

illustrate that there can be multiple graphs with the same “flow” but different costs?

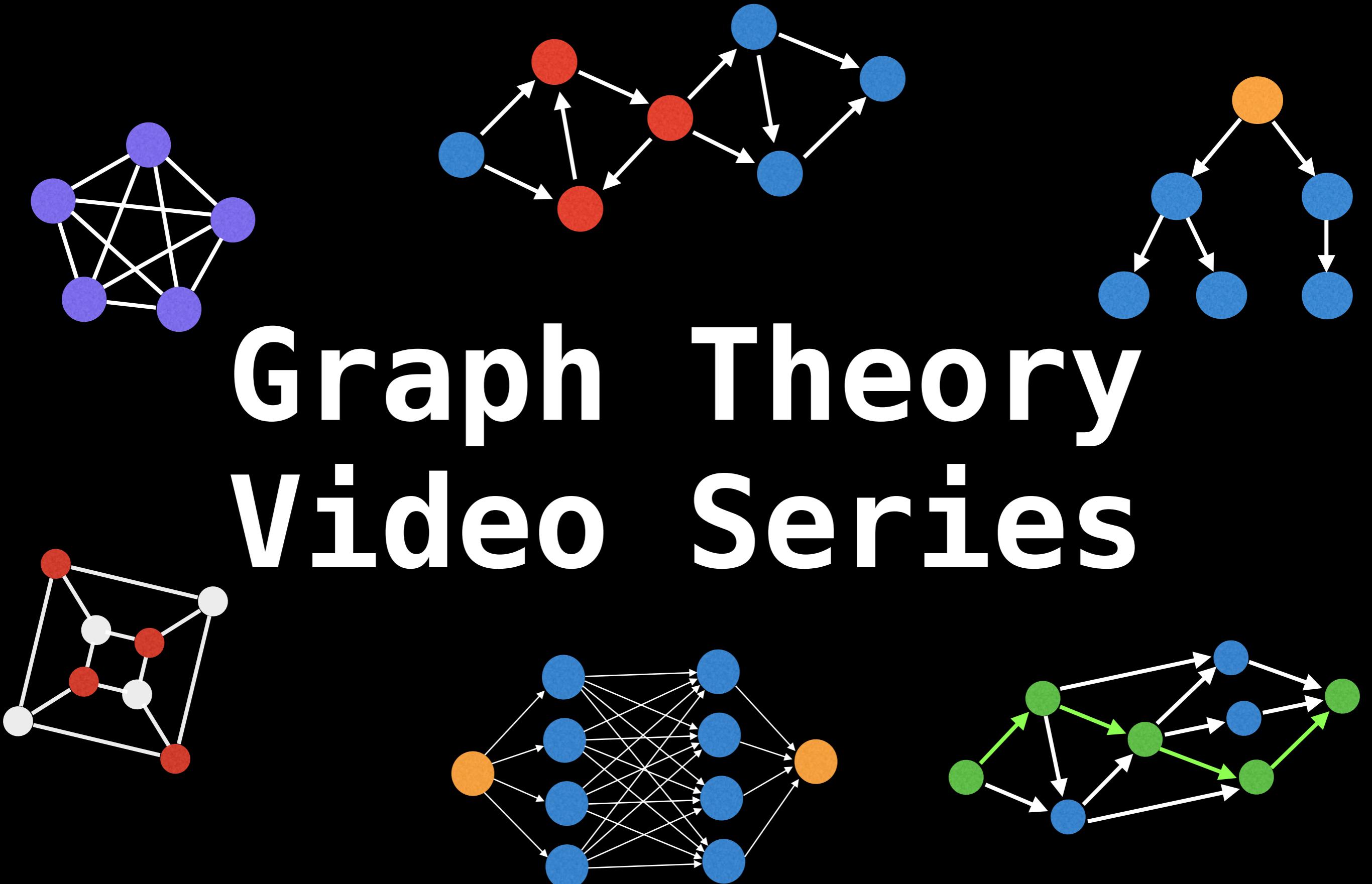
# Negative Cycles

"Because of negative edge costs, these networks can have negative-cost cycles. The concept of negative cycles, which seemed artificial when we first considered it in the context of shortest-path algorithms, plays a critical role in mincost-flow algorithms we now see."

# Negative Cycles

"A maxflow is a mincost maxflow if and only if its residual network contains no negative-cost (directed) cycles."

# Graph Theory Video Series



# Network Flow: Mincost Maxflow Source Code

William Fiset

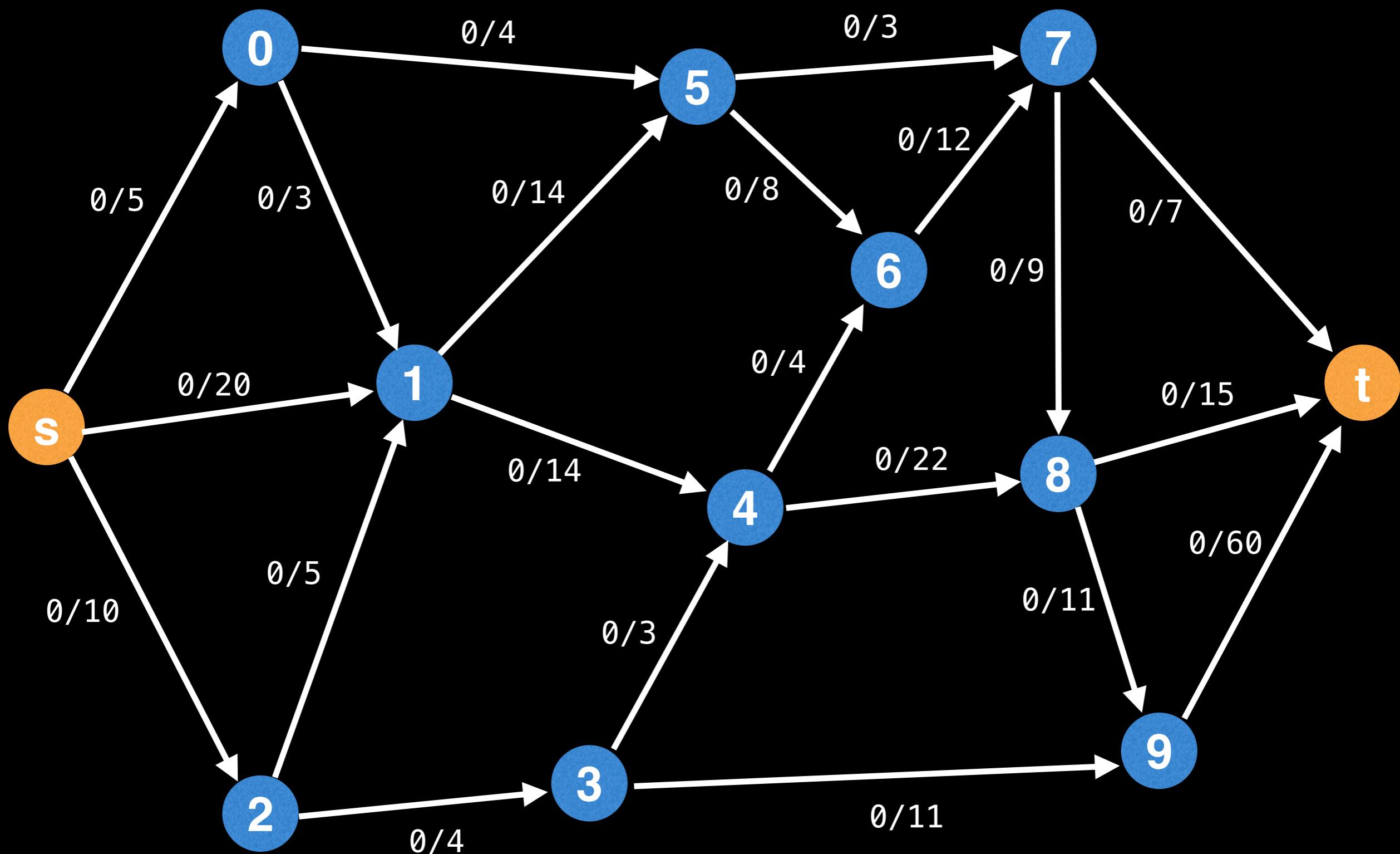
# TODO list

Make sure to show changes to the `NetworkFlowSolverBase`. In particular mention the `addEdge` method and the negative cost edge there and the new cost property added to edges.

# Next Video: Dinic's Source Code

Unused slides

# Max Flow



Would have been a good example but it's too large and might confuse some folks

