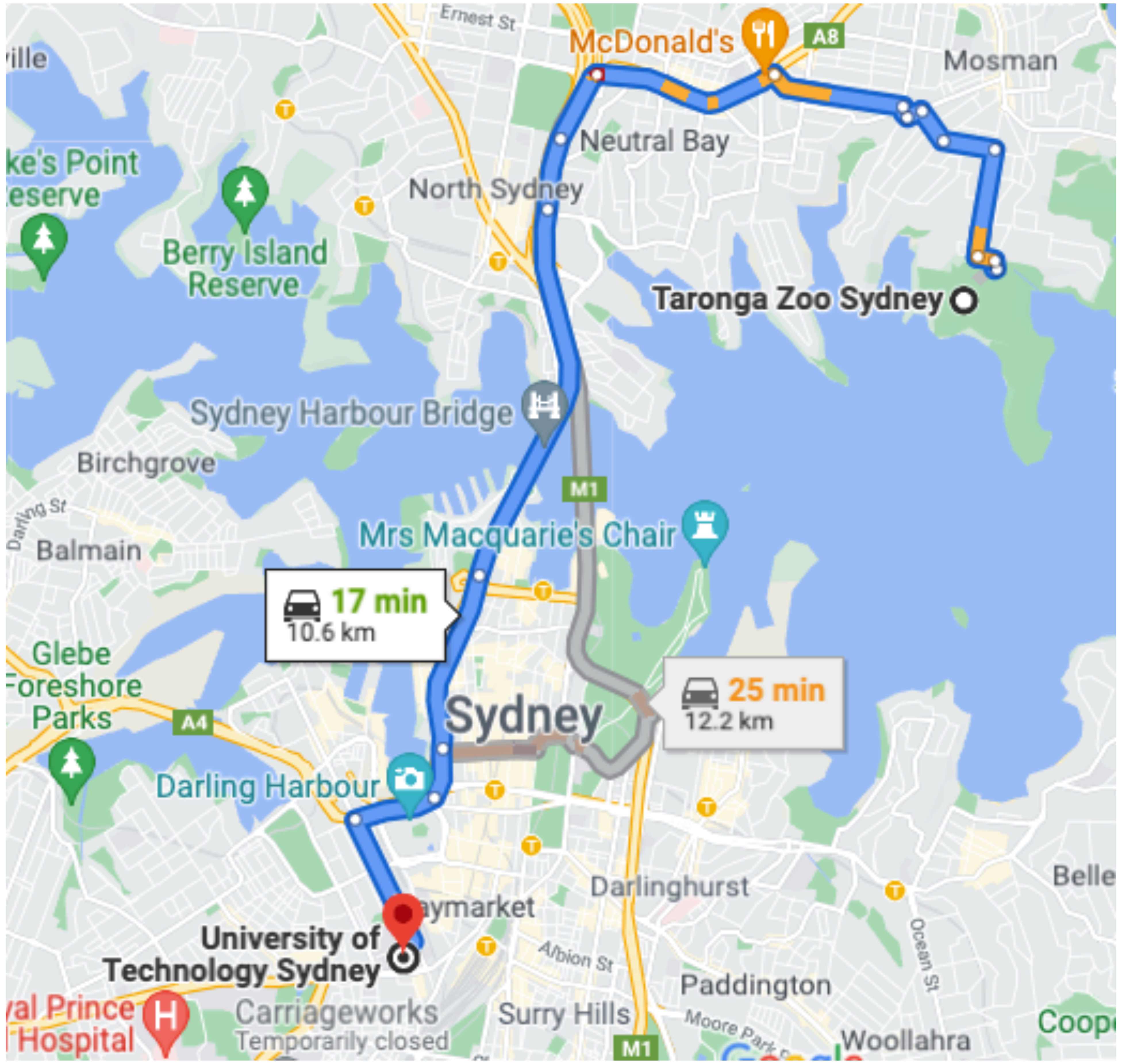
Shortest Paths

Shortest Paths

Today we are going to talk about finding shortest paths in directed and weighted graphs.

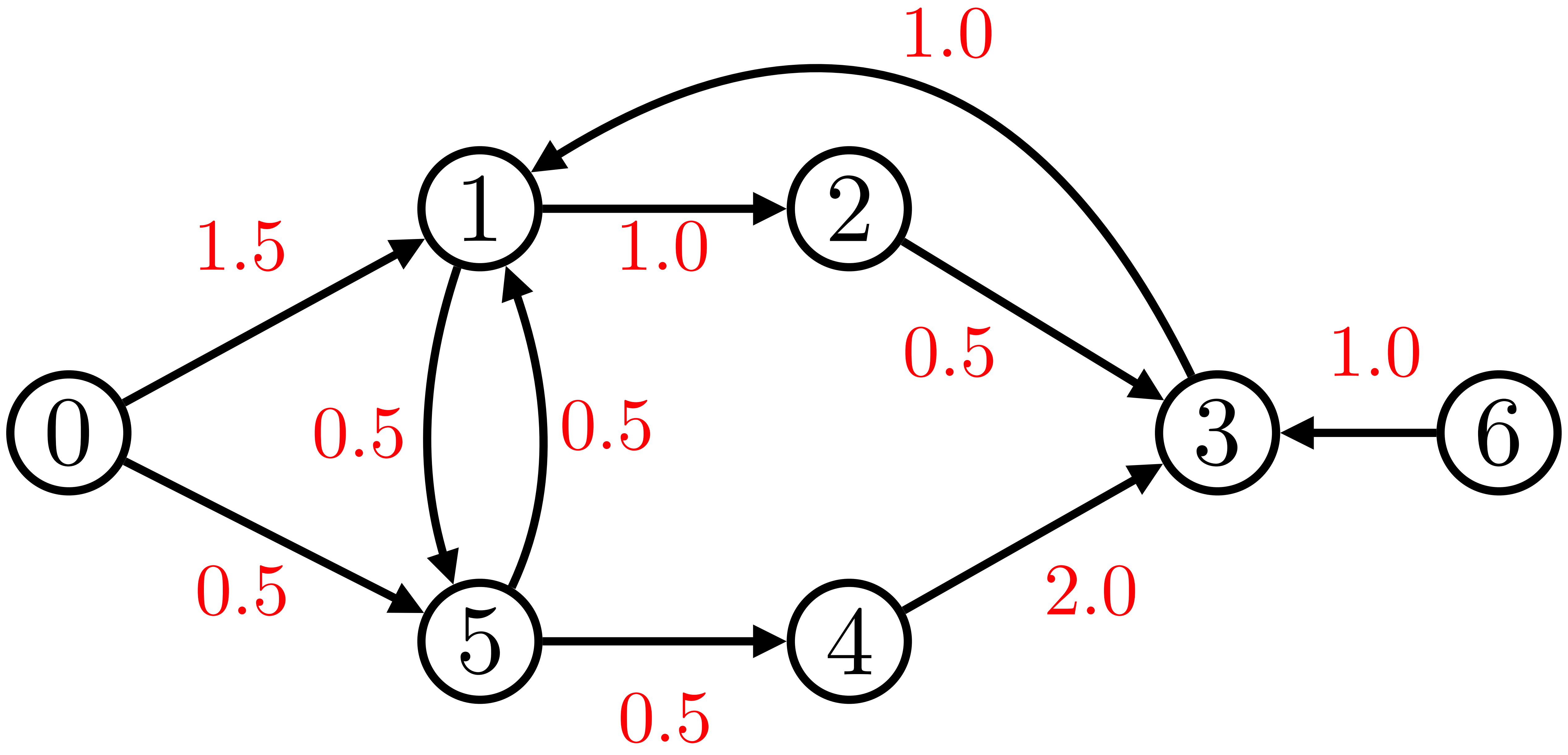


This is a problem that most of us solve every day.

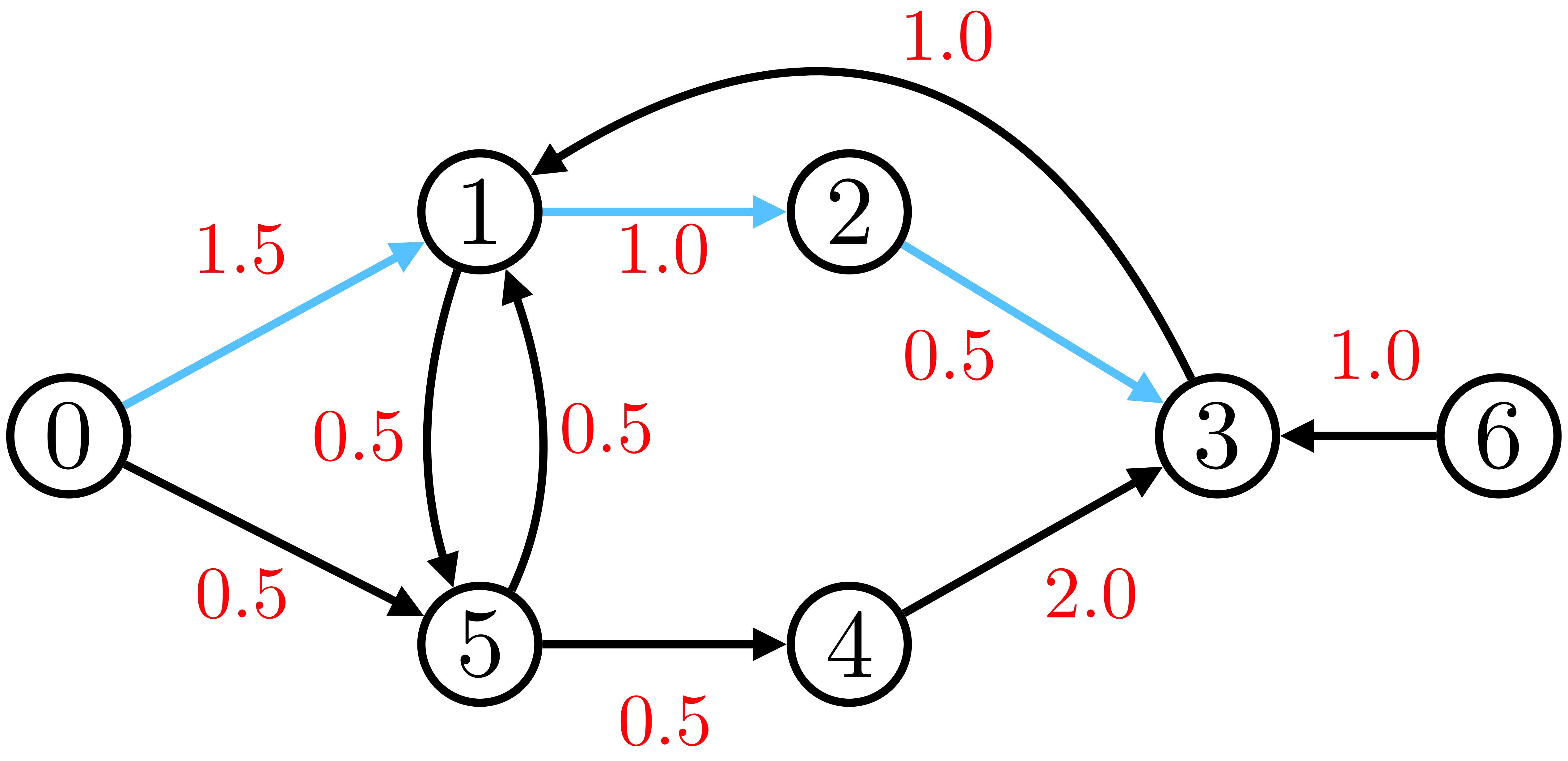
We might need directed graphs because of one-way streets.

Edge weights could be distance, time, fuel, cost, etc.

Example



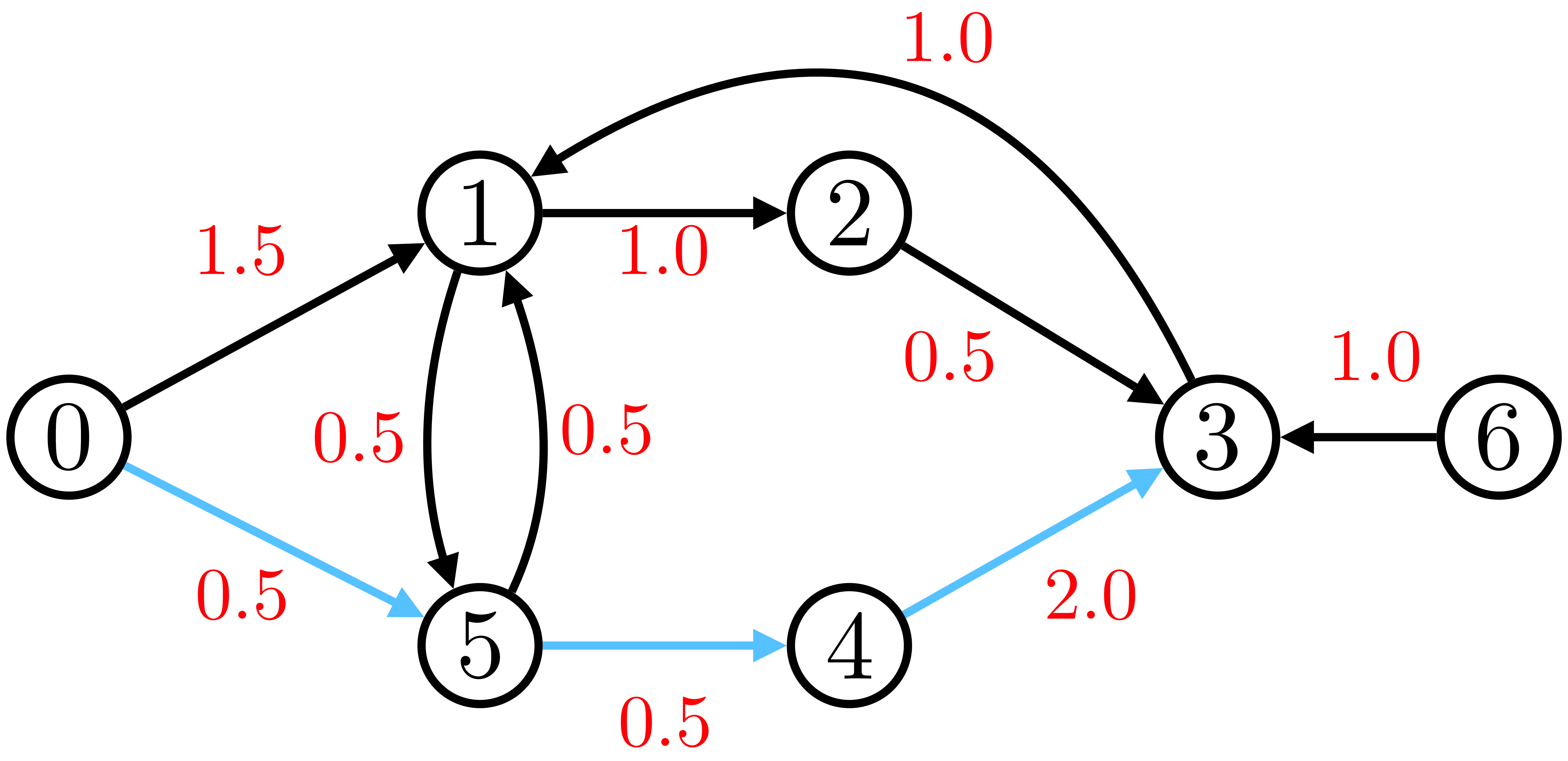
There are several paths from vertex 0 to vertex 3:



There are several paths from vertex 0 to vertex 3:

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3$$

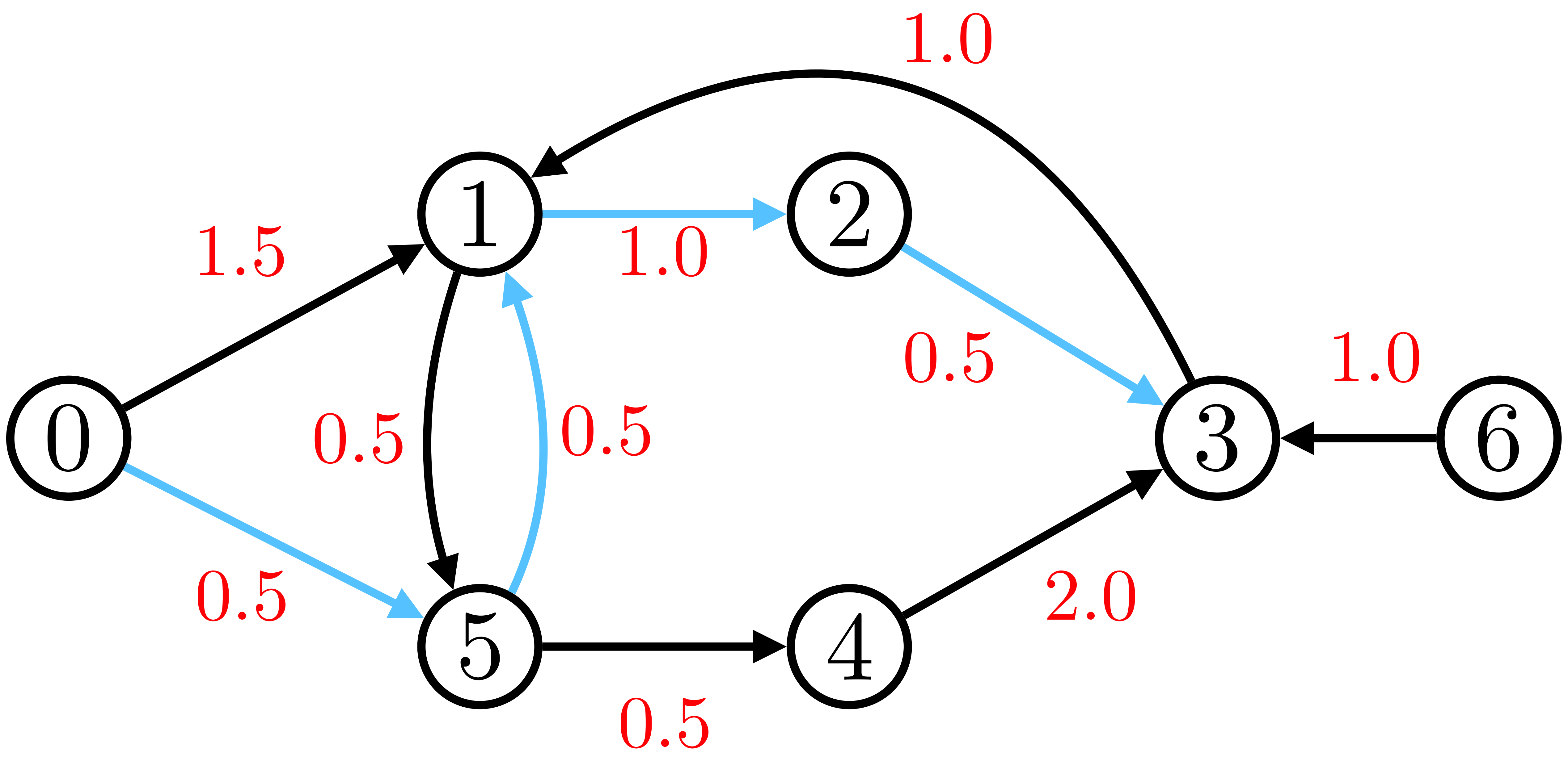
The weight of this path is 3.0.



There are several paths from vertex 0 to vertex 3:

$$0 \to 5 \to 4 \to 3$$

The weight of this path is 3.0.

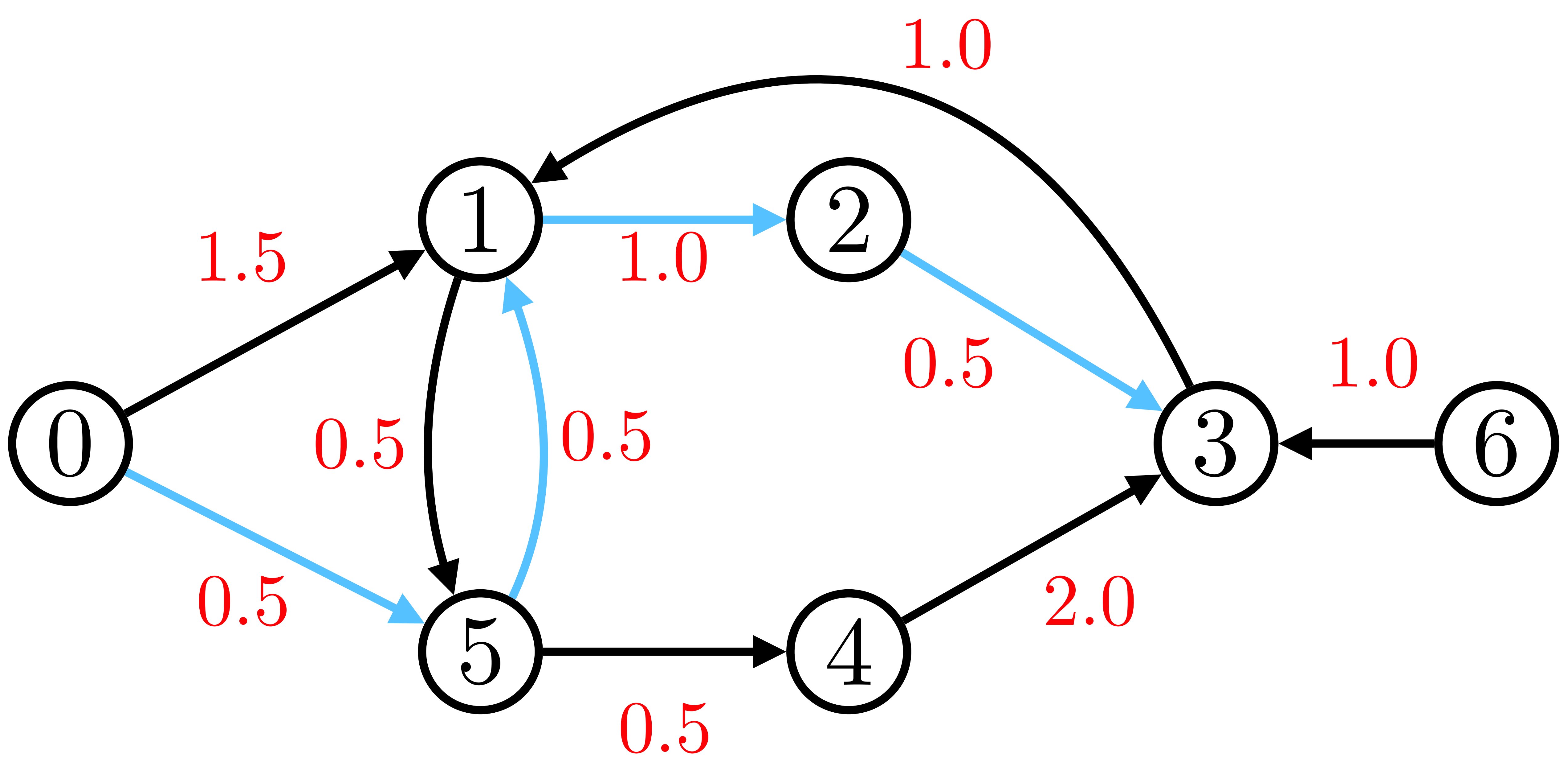


There are several paths from vertex 0 to vertex 3:

$$0 \to 5 \to 1 \to 2 \to 3$$

The weight of this path is 2.5.

This is the shortest path from 0 to 3 in this graph. The total sum of edge weights on the path is the smallest.



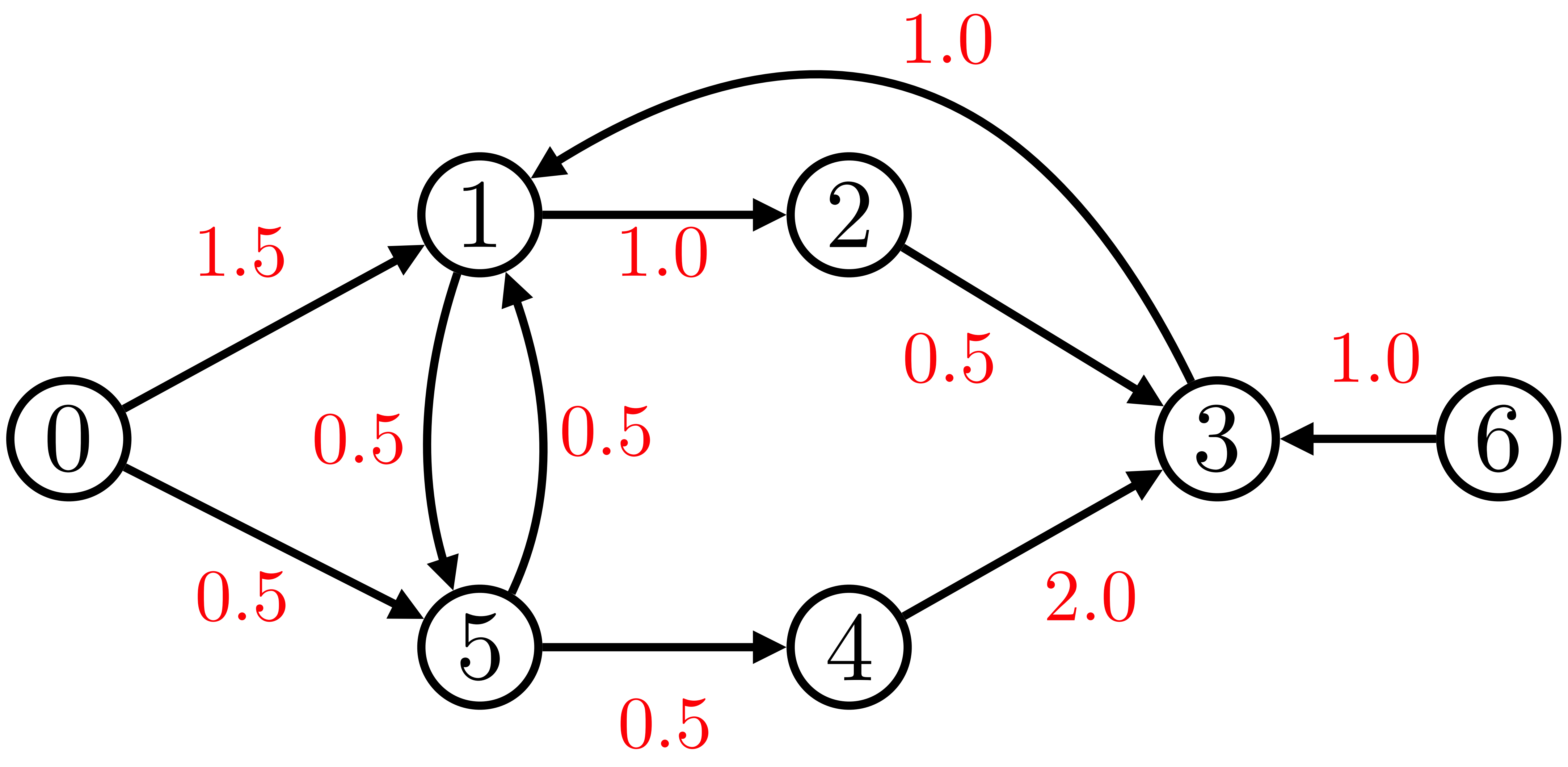
$0 \to 5 \to 1 \to 2 \to 3$

The length of a shortest path from 0 to 3 is 2.5.

We define the distance $d(u, v)$ from vertex u to vertex v to be the length of a shortest path from u to v.

In this graph $d(0, 3) = 2.5$.

Unreachable Vertices

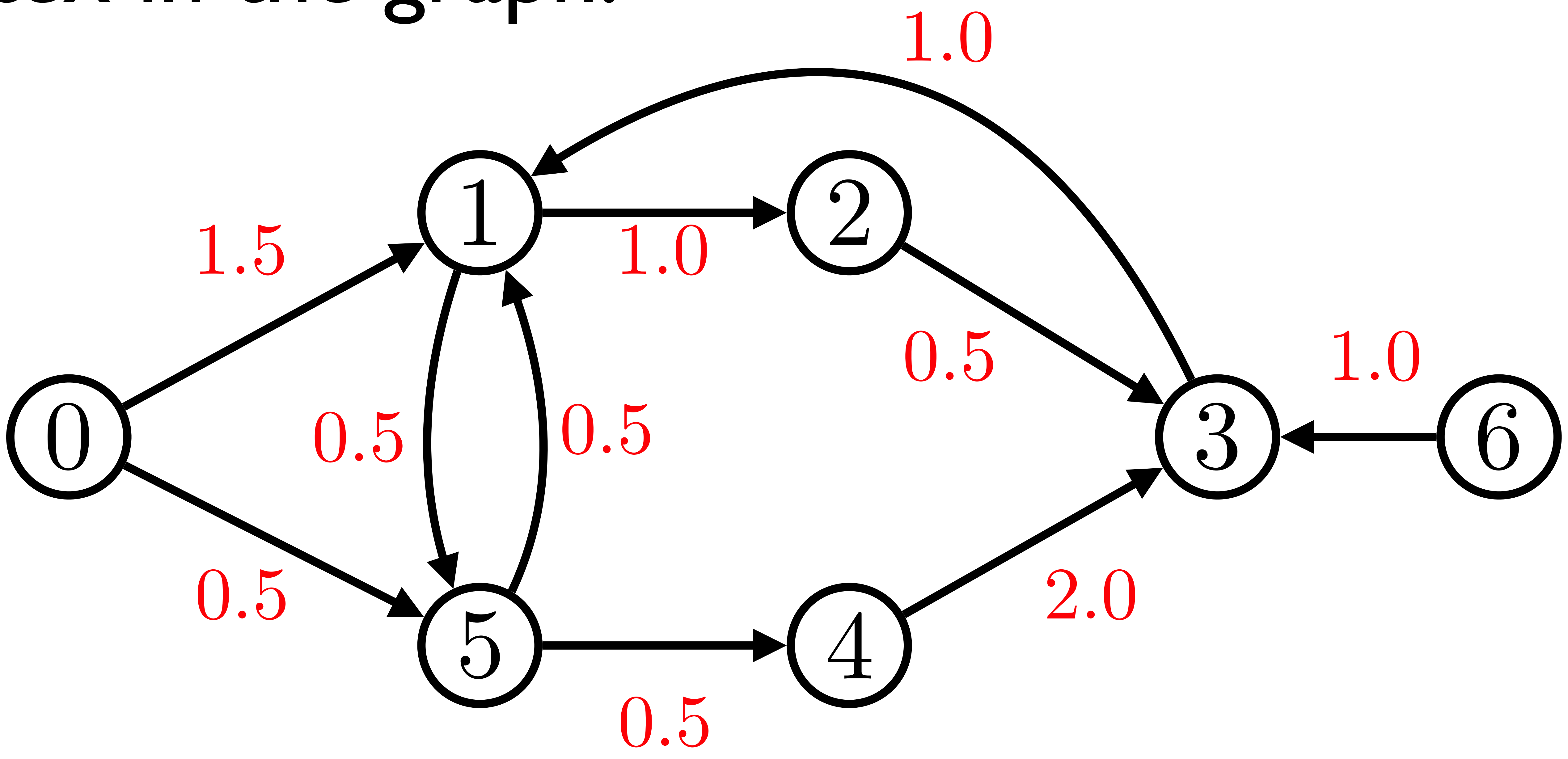


In order to avoid a special case, when v is not reachable from u we define $d(u, v) = \infty$.

In this graph $d(0, 6) = \infty$.

Single-Source Distance

Single-source distance problem: Given a vertex v, find the distance from v to every other vertex in the graph.



output: distance from vertex 0.

0	1	2	3	4	5	6
0	1	2	2.5	1	0.5	∞

$\texttt{dist\_to}[i] = d(0, i)$

Single-Source Shortest Paths

Usually we don't just want to know the distance from vertex v to the other vertices in the graph but also shortest paths.

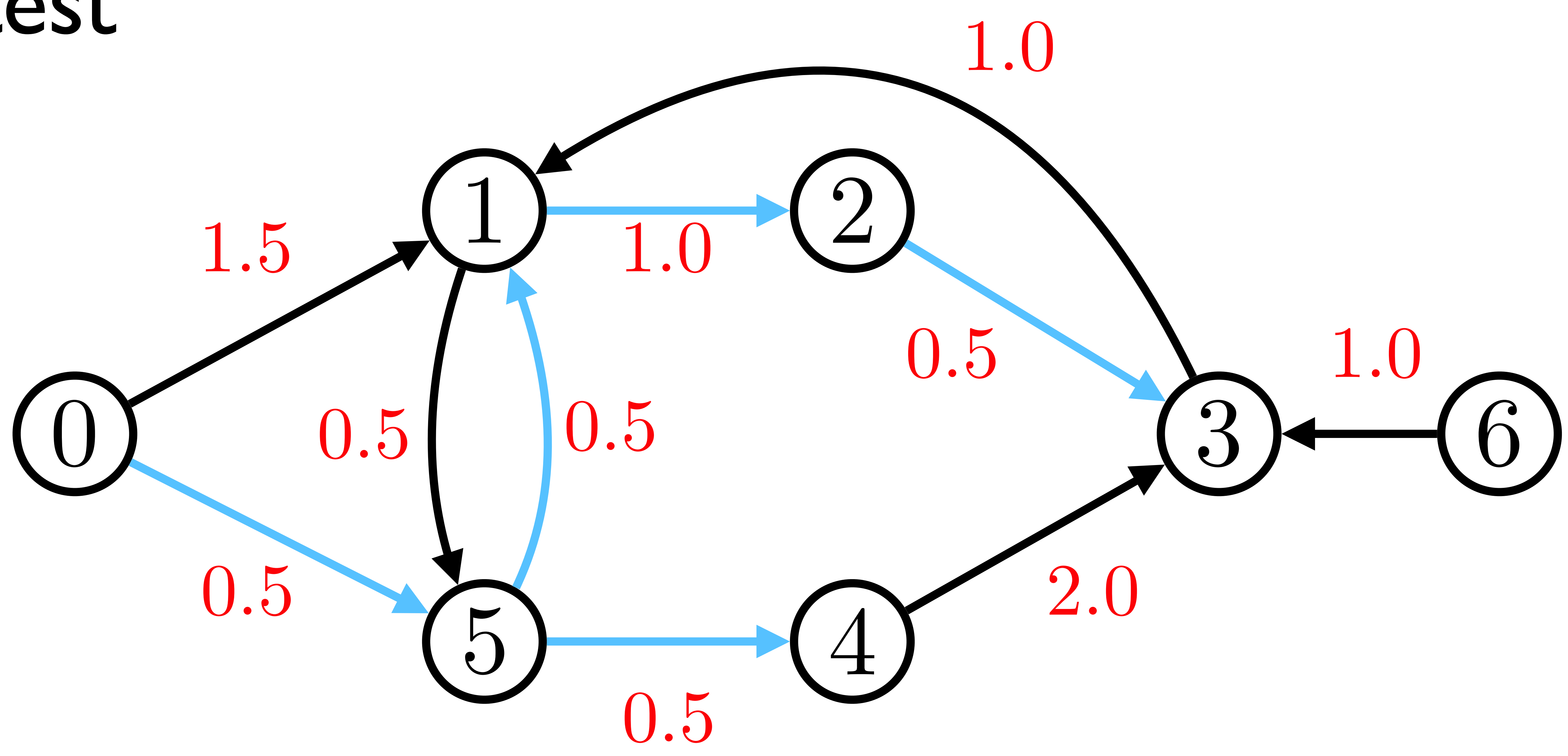
It is generally too expensive to output a shortest path from v to every other vertex in the graph.

But we can output a data structure from which shortest paths can be reconstructed.

We can output array of size n so that for any vertex w we can reconstruct a shortest path from the source v to w in time $O(n)$.

Single-Source Shortest Paths

Following the blue edges gives a shortest path from 0 to every other vertex reachable from 0.



This is the analog of the shortest-path tree we saw in the undirected case.

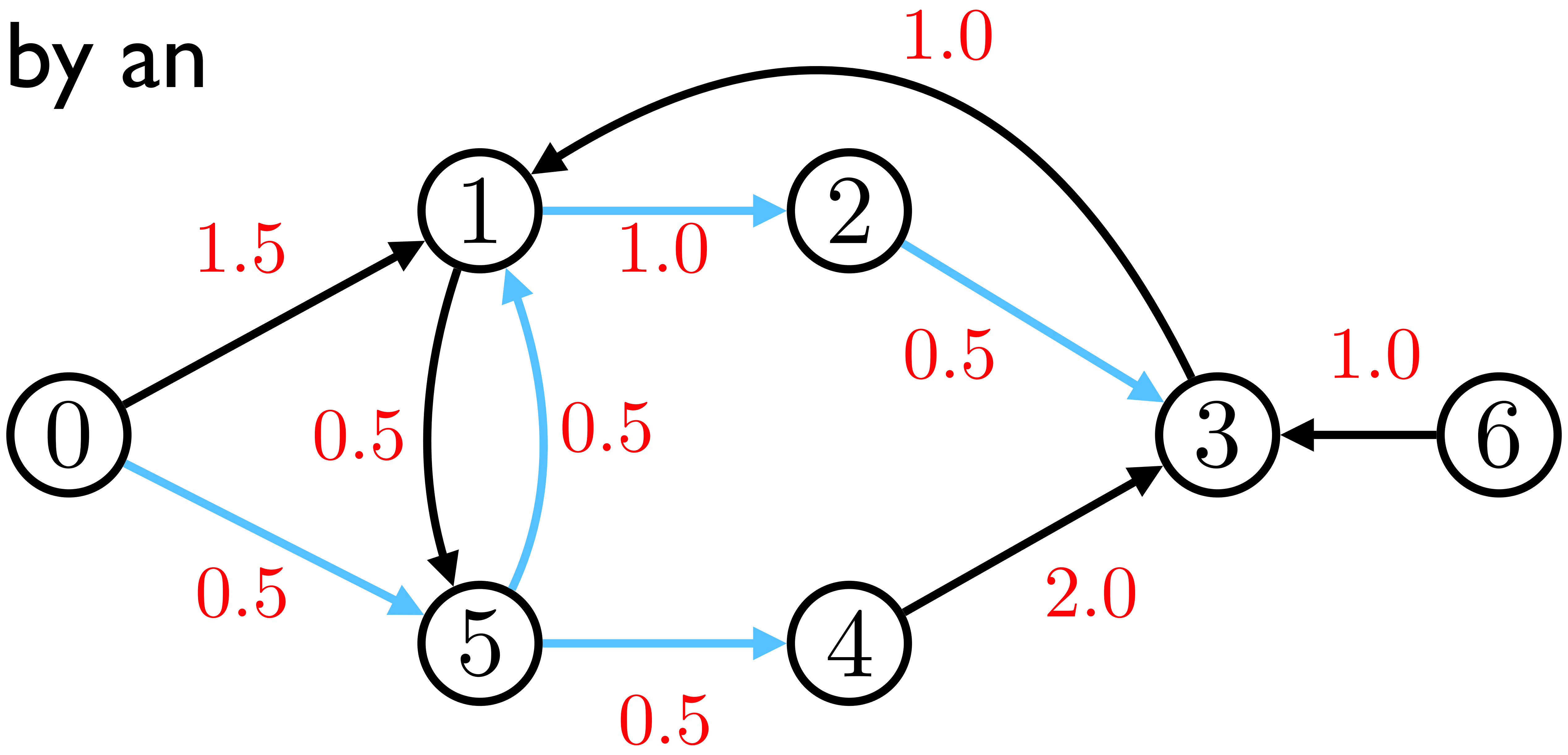
There is a unique blue-edge path from 0 to every vertex reachable from 0.

Single-Source Shortest Paths

We can represent the blue edges by an array of size n .

Every vertex has at most one incoming blue edge.

$\texttt{edge\_to}[i]$ gives the name of the vertex that the blue edge to i comes from, or is -1 if i has no incoming blue edge.



-1	5	1	2	5	0	-1
0	1	2	3	4	5	6

edge_to

Reconstructing Shortest Paths

We can use the `edge_to` array to reconstruct shortest paths from 0.

A shortest path from vertex 0 to vertex 3 is given by:

3

↑

edge_to[3] == 2

↑

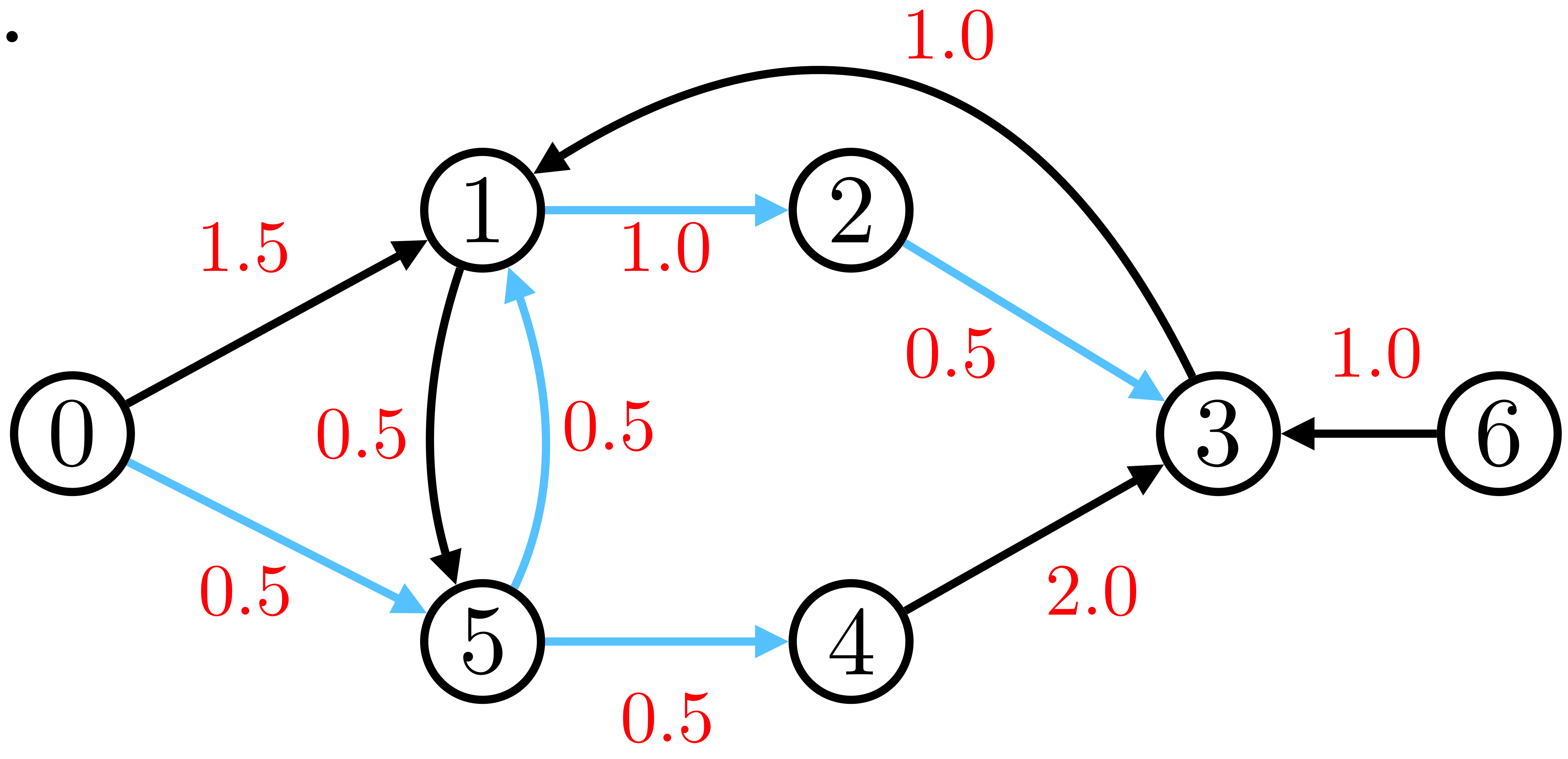
edge_to[2] == 1

↑

edge_to[1] == 5

↑

edge_to[5] == 0



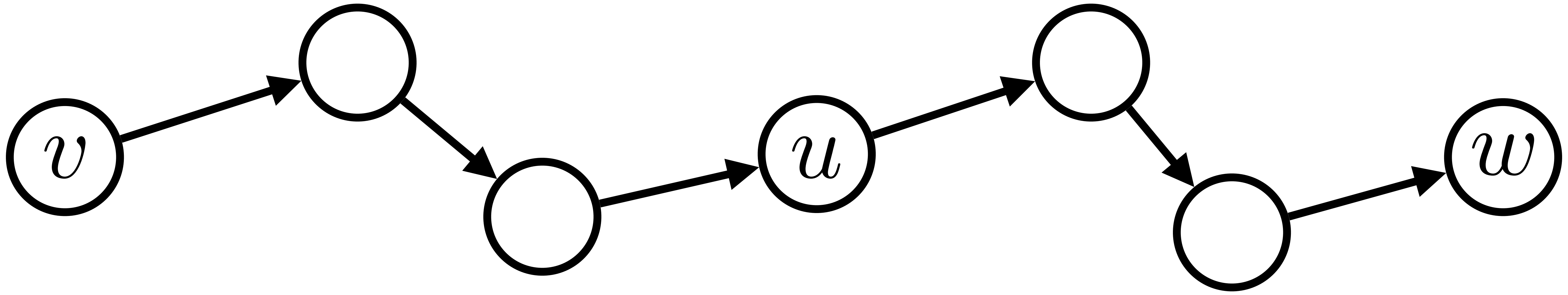
−1	5	1	2	5	0	−1	edge_to
0	1	2	3	4	5	6	

Single-Pair Shortest Path

You may be wondering why we talk about the single-source shortest path problem.

Usually we just ask google maps how to go from point A to point B, not from point A to everywhere else.

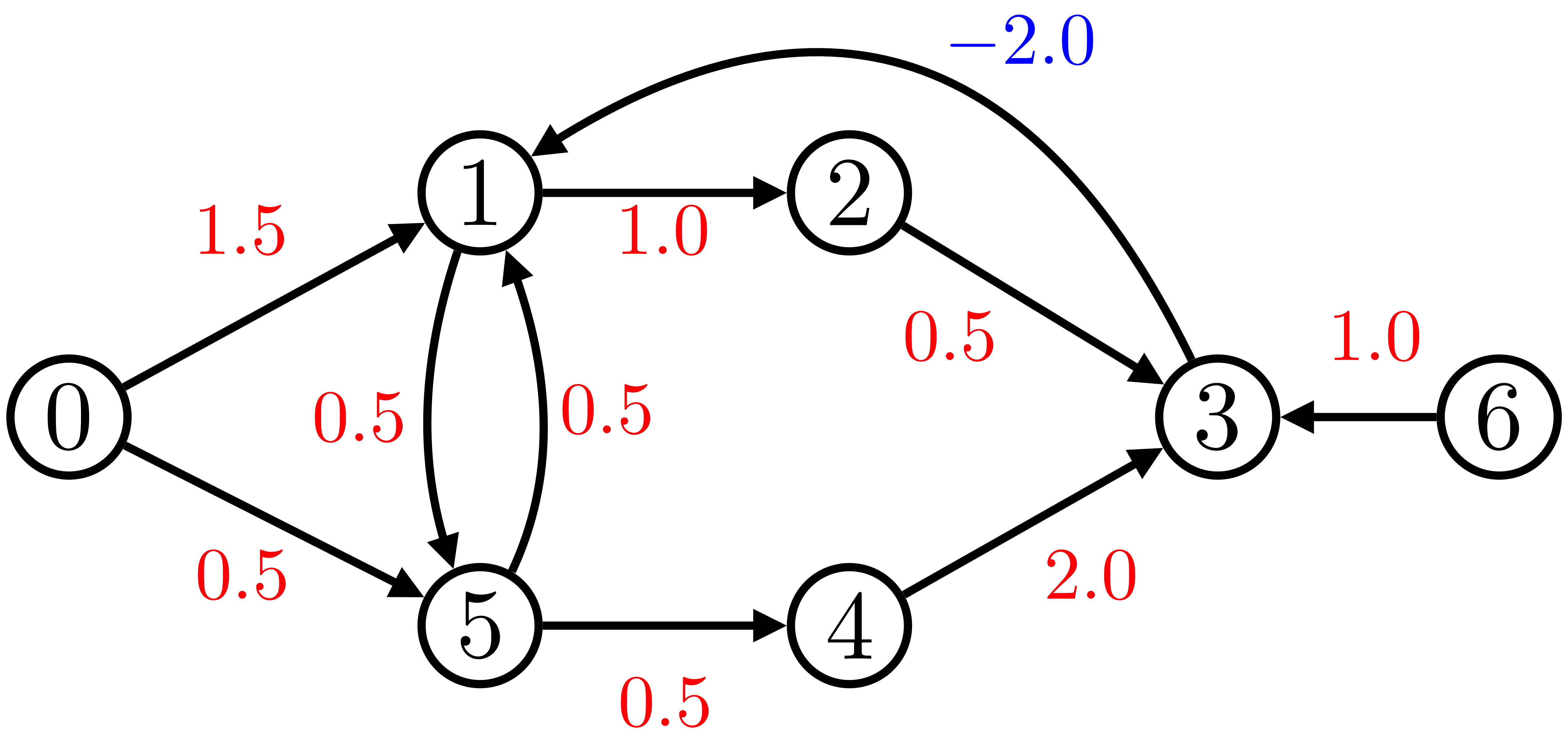
We can consider the single-pair shortest path problem, but we don't have any better algorithms than for the single-source shortest path problem!



If the shortest path from v to w goes through u, it uses a shortest path from v to u.

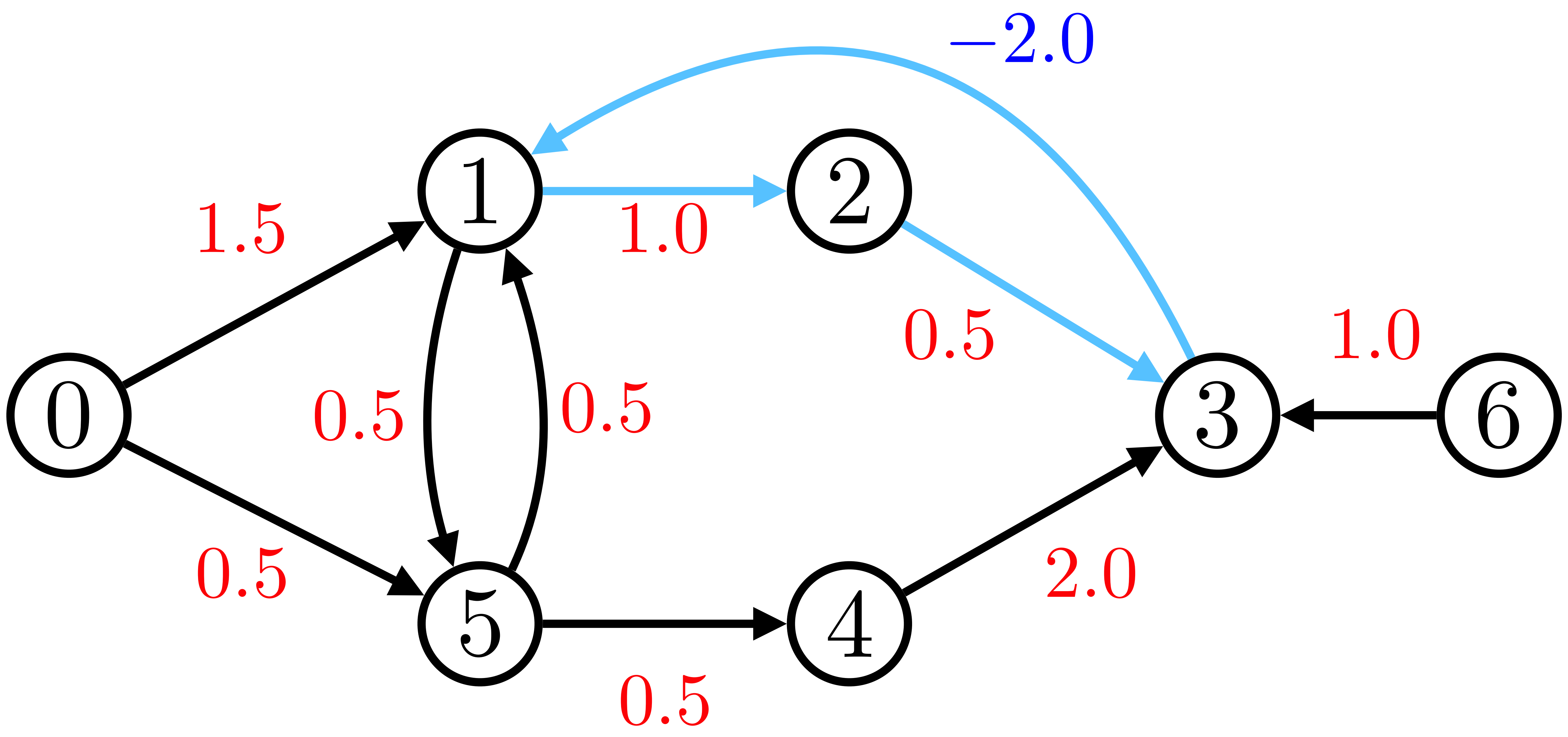
Negative Weight Cycles

Before we get into shortest path algorithms we have to discuss a special case.



I've taken our example graph and changed the weight in blue to instead be -2.

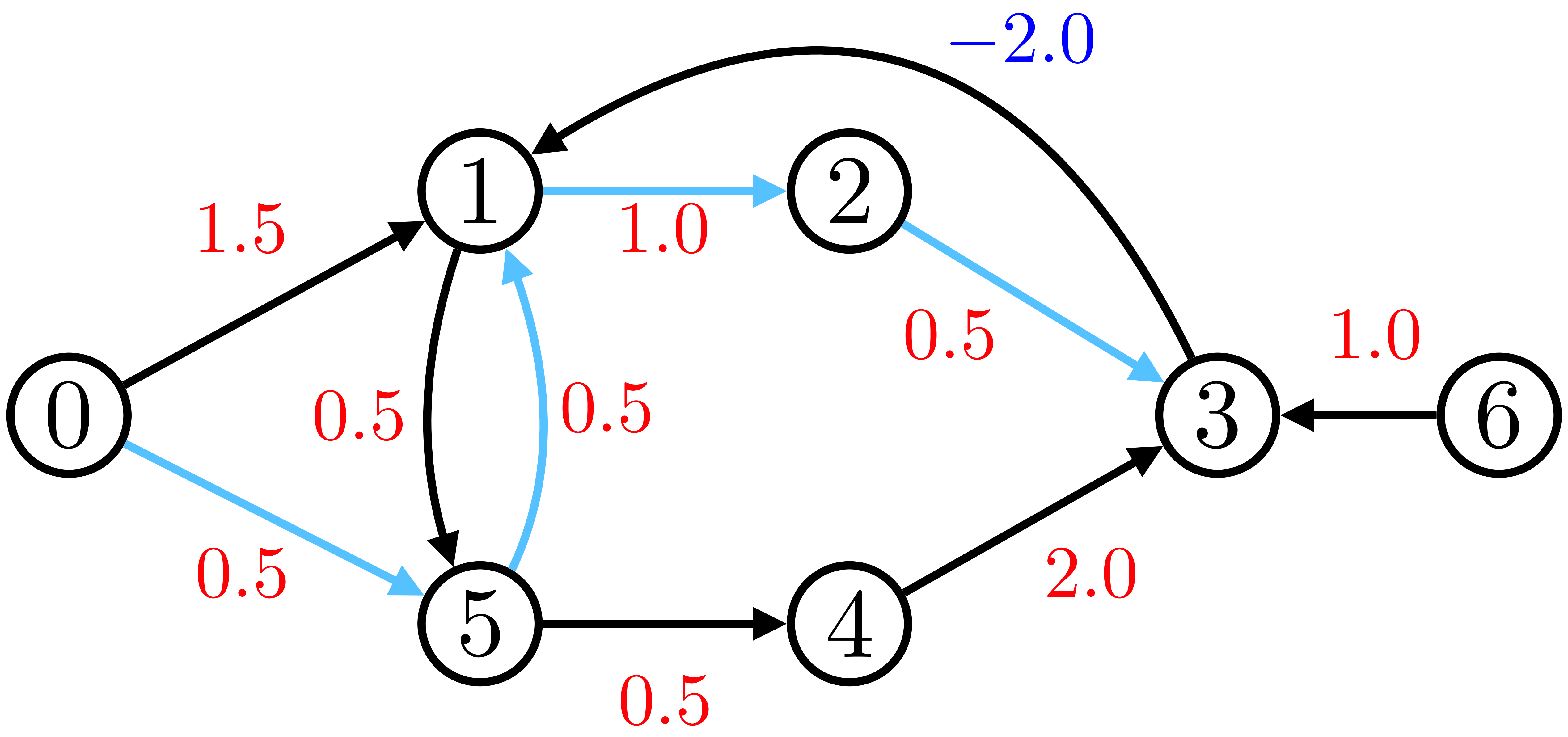
Negative Weight Cycles



The sum of the weights on the blue edge cycle is -0.5.

Now what is the shortest path from vertex 0 to vertex 3?

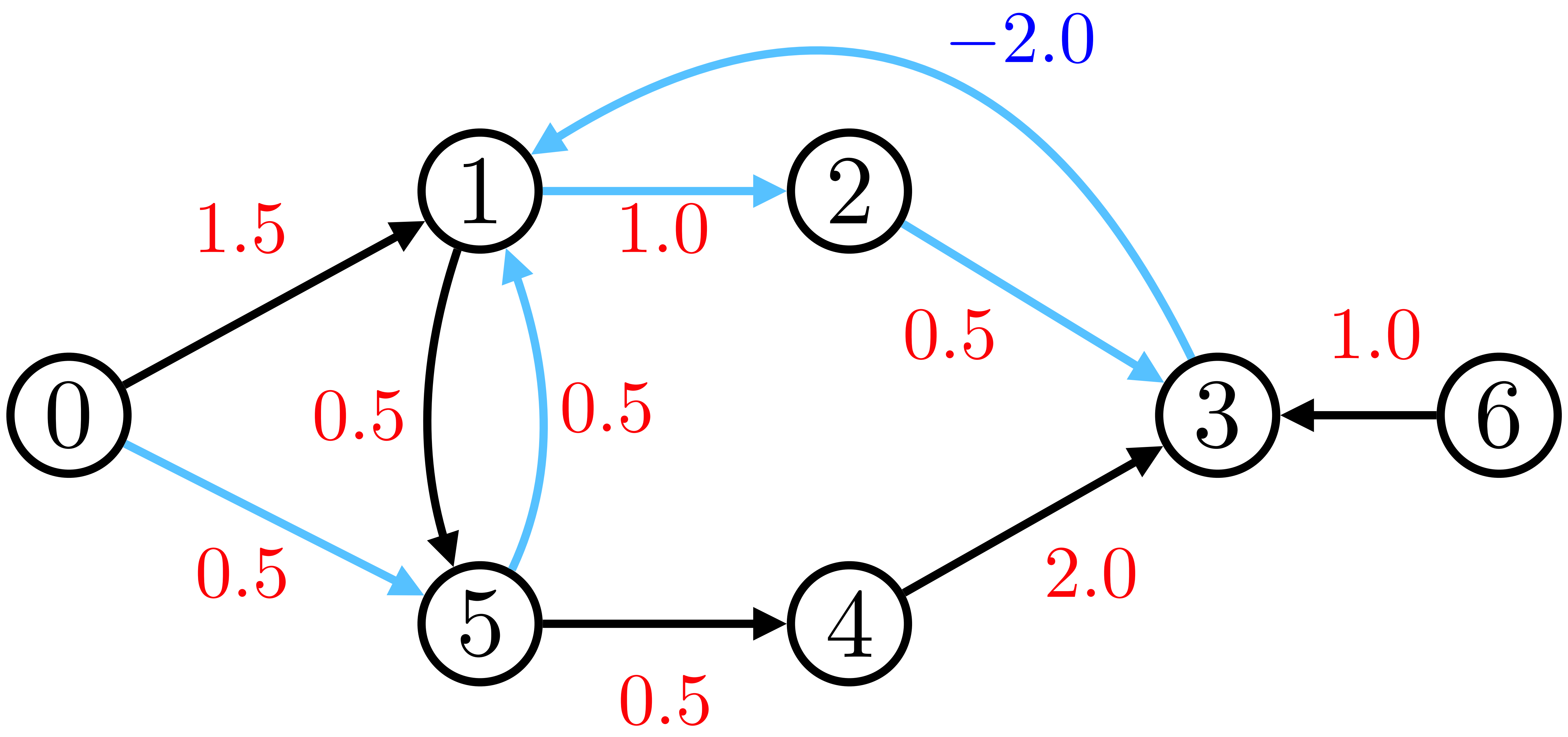
Negative Weight Cycles



This path has length 2.5.

But if we go around the negative weight cycle, we get a path of length 2.0.

Negative Weight Cycles

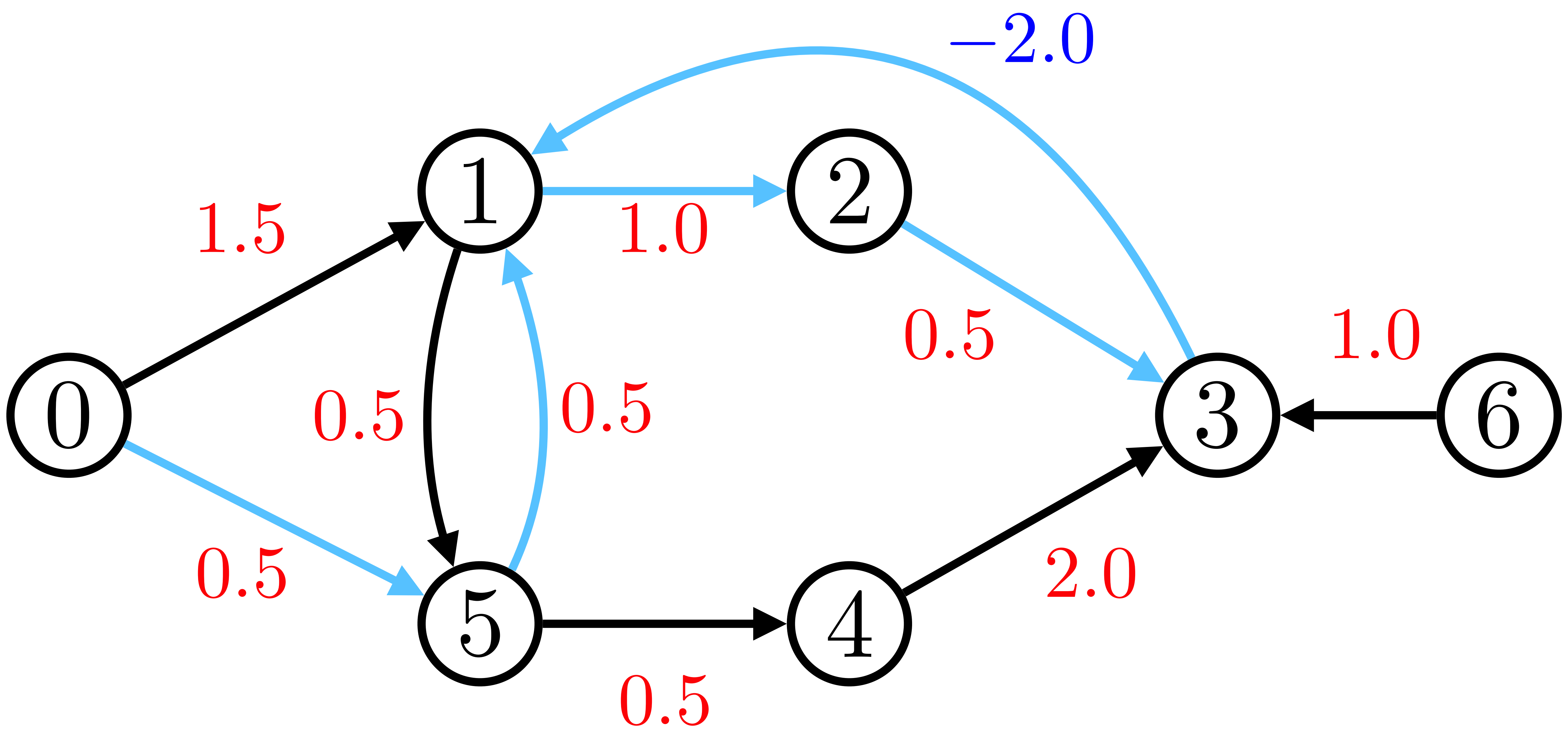


If we go around the negative weight cycle and then stop at vertex 3, we get a path of length 2.0.

If we go around the negative weight cycle twice, we get a path of length 1.5.

Using the negative weight cycle, we can get shorter and shorter paths.

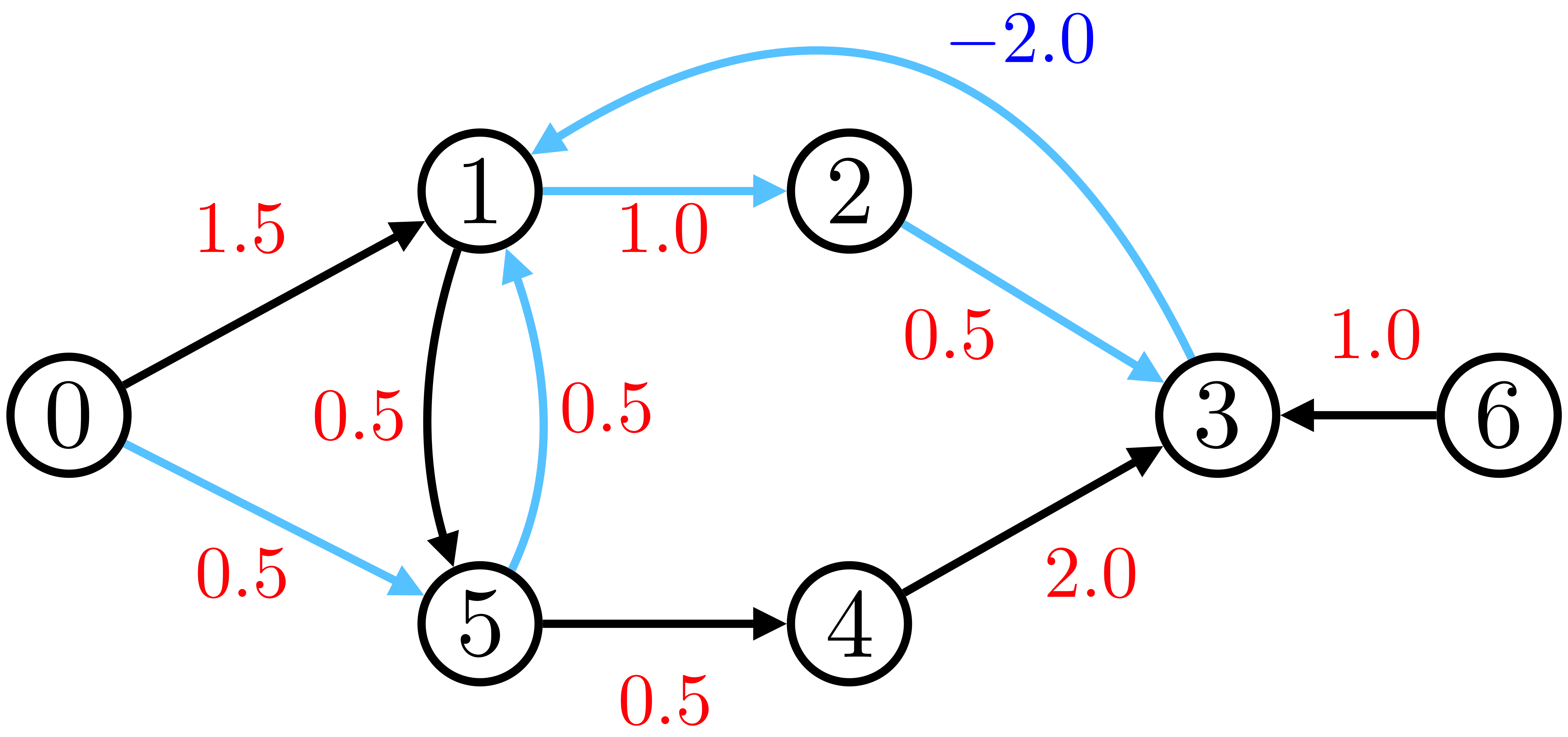
Negative Weight Cycles



In this case we define $d(0, 3) = -\infty$, and the shortest path from vertex 0 to vertex 3 is undefined.

When a negative weight cycle is reachable from the source vertex, the distance to every other vertex reachable from the source vertex is $-\infty$.

Negative Weight Cycles



We would like to detect if there is a negative weight cycle reachable from the source vertex, and output the cycle if so.

We will see that this can be done by the Bellman-Ford shortest path algorithm.

Negative weight cycles have interesting applications to arbitrage in markets.

Simple Shortest Paths

If no negative weight cycle is reachable from the source vertex, then shortest paths from the source vertex do not contain cycles.

If the weight of the cycle is ≥ 0, removing the cycle gives a path that is not longer.

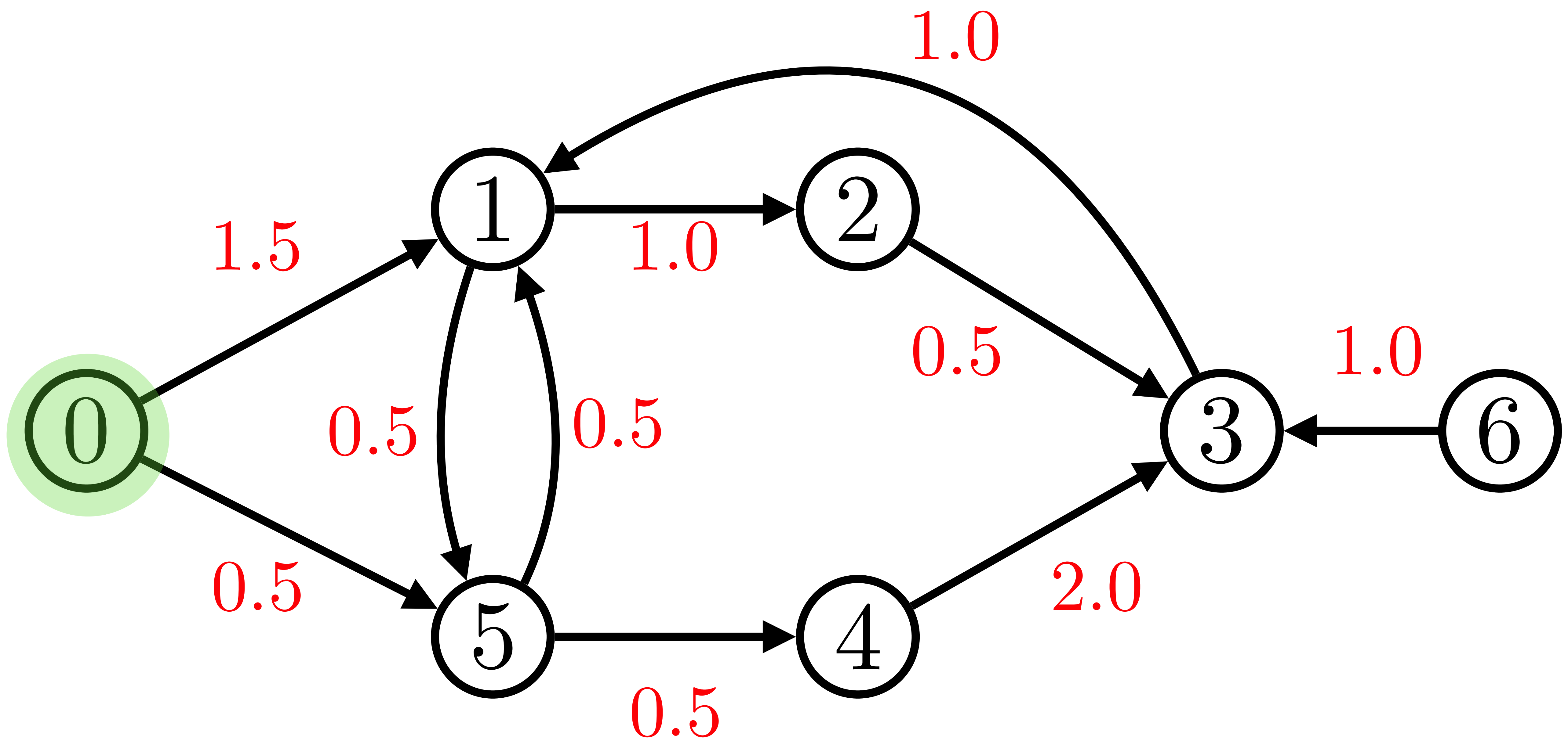
In this case shortest paths do not repeat vertices, they will be simple.

Shortest paths will have at most $n-1$ edges when the graph has n vertices.

Generic Shortest Path Algorithm

Setup

Let's say the source vertex is 0.



We want to compute two things:

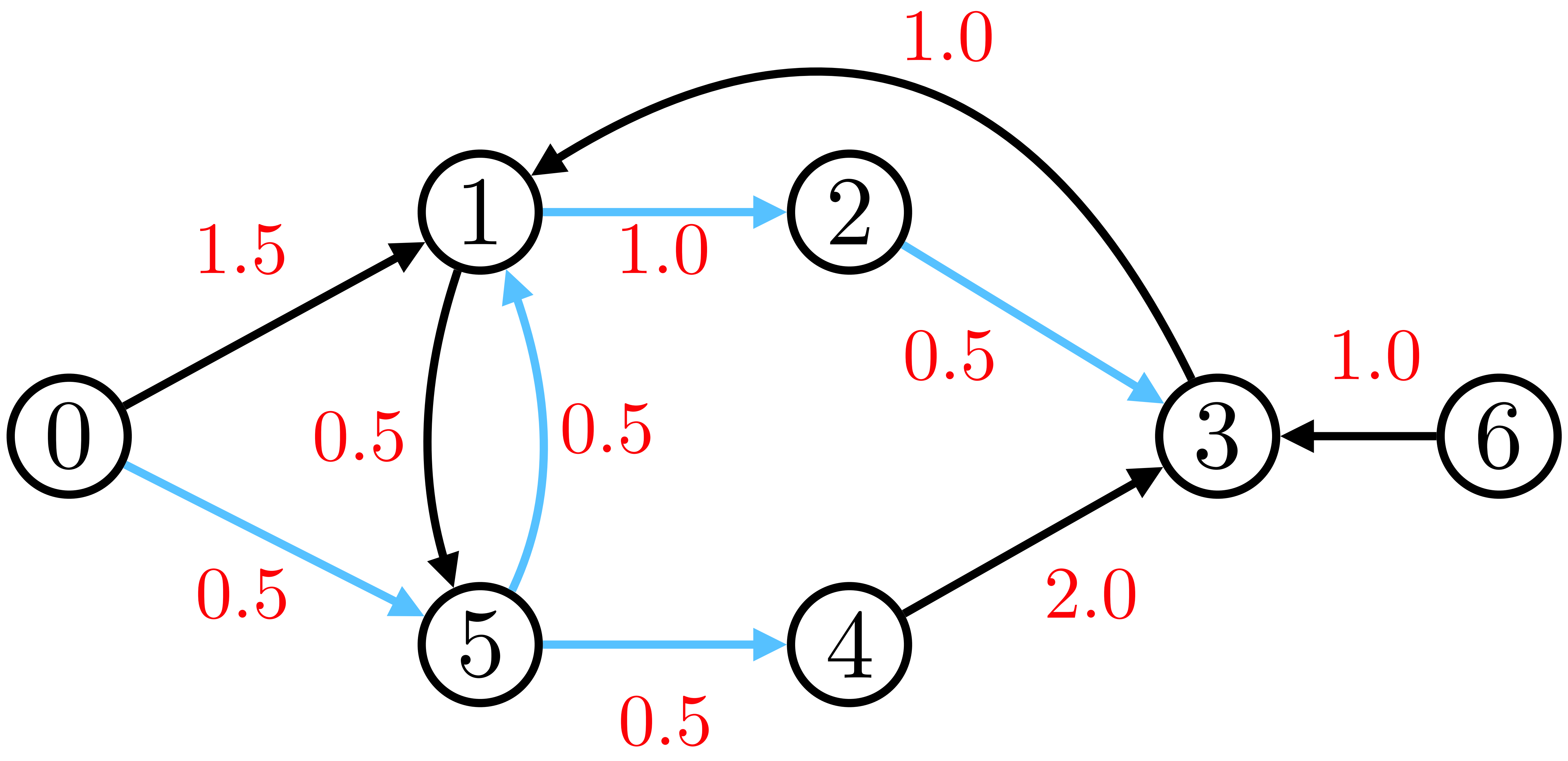
1) The distance from 0 to every other vertex in the graph:

0	1	2	2.5	1	0.5	∞
0	1	2	3	4	5	6

$\texttt{dist\_to}[i] = d(0, i)$

Setup

Let's say the source vertex is 0.



We want to compute two things:

2) An array `edge_to` encoding shortest paths from vertex 0:

-1	5	1	2	5	0	-1	`edge_to`
0	1	2	3	4	5	6	

Initialization

0	∞	∞	∞	∞	∞	∞
0	1	2	3	4	5	6

`dist_to`

We initialize $\texttt{dist\_to}[0] = 0$ and $\texttt{dist\_to}[i] = \infty$ for every other vertex i.

Invariant I: The algorithm always maintains that $d(0, i) \leq \texttt{dist\_to}[i]$ for every vertex i.

Notice that this is true initially.

Initialization

-1	-1	-1	-1	-1	-1	-1	`edge_to`
0	1	2	3	4	5	6	

We initialize `edge_to` to be everywhere -1.

Invariant 2: tracing back $i \leftarrow \texttt{edge\_to}[i] \leftarrow \cdots \leftarrow 0$ gives a path from vertex 0 to i of length $\leq \texttt{dist\_to}[i]$.

Notice that this is true initially. When $\texttt{dist\_to}[i] = \infty$ there is nothing to certify.

Relaxing an Edge

The generic algorithm just repeatedly picks an edge and relaxes it.

To relax the edge $e = (u, v)$ we check if

$$\texttt{dist\_to}[u] + e.\texttt{weight} < \texttt{dist\_to}[v]$$

If so, then we do the updates:

$$\texttt{dist\_to}[v] = \texttt{dist\_to}[u] + e.\texttt{weight}$$

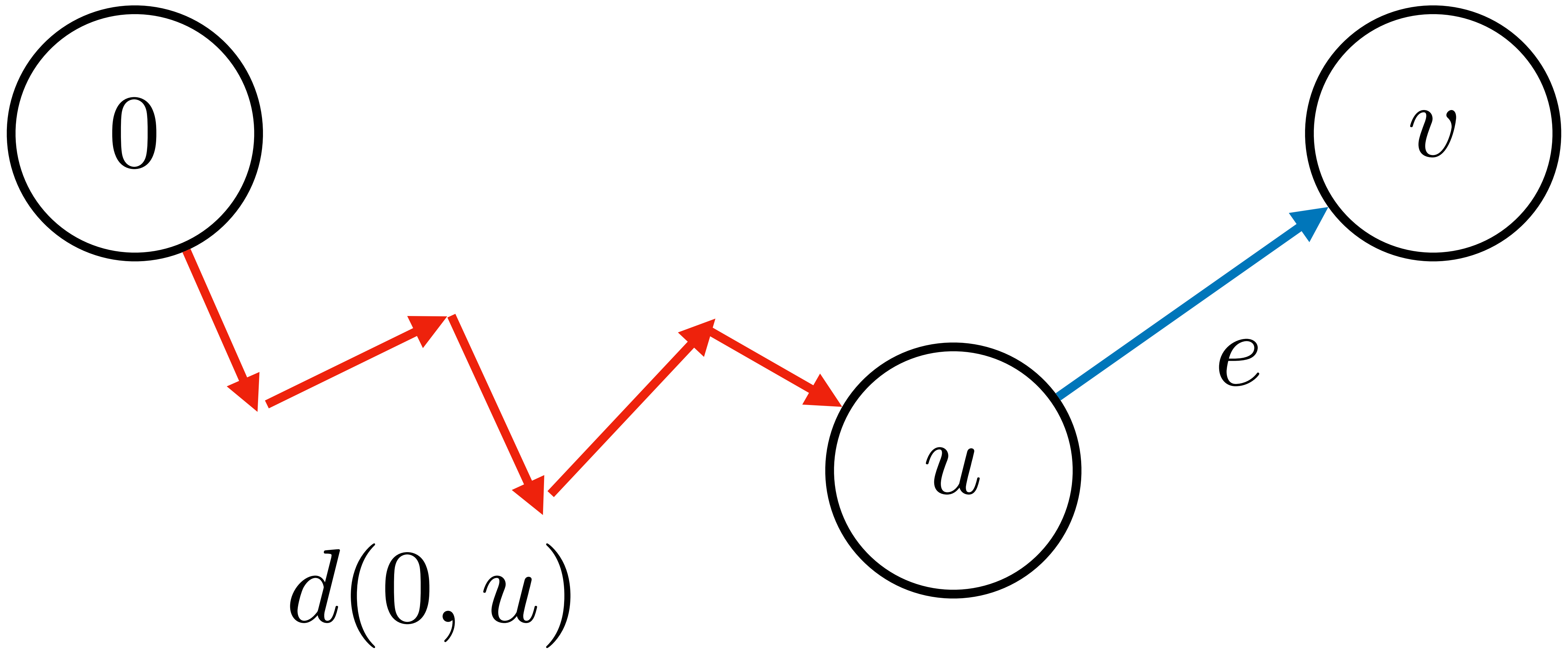
$$\texttt{edge\_to}[v] = u$$

Note that $\texttt{dist\_to}[v]$ never increases under edge relaxation.

Relaxing Preserves Invariants

Triangle Inequality: If there is an edge $e = (u, v)$ then

$$d(0, v) \leq d(0, u) + e.\texttt{weight}$$



Relaxing Preserves Invariants

Invariant 1: The algorithm always maintains that $d(0, i) \leq \texttt{dist\_to}[i]$ for every vertex i.

Suppose this invariant holds before we relax the edge $e = (u, v)$. So we know that $d(0, u) \leq \texttt{dist\_to}[u]$.

$$d(0, v) \leq d(0, u) + e.\texttt{weight}$$ triangle inequality

$$\leq \texttt{dist\_to}[u] + e.\texttt{weight}$$

So updating $\texttt{dist\_to}[v] = \texttt{dist\_to}[u] + e.\texttt{weight}$ preserves Invariant 1.

Invariant 2: Tracing back $i \leftarrow \texttt{edge\_to}[i] \leftarrow \cdots \leftarrow 0$ gives a path from vertex 0 to i of length $\leq \texttt{dist\_to}[i]$.

Suppose this invariant holds before we relax the edge $e = (u, v)$. So tracing back $u \leftarrow \texttt{edge\_to}[u] \leftarrow \cdots \leftarrow 0$ gives a path from 0 to u of length $\leq \texttt{dist\_to}[u]$.

If we do the update $\texttt{dist\_to}[v] = \texttt{dist\_to}[u] + e.\texttt{weight}$ then we also set $\texttt{edge\_to}[v] = u$.

Tracing back $v \leftarrow \texttt{edge\_to}[v] = u \leftarrow \cdots \leftarrow 0$ gives a path from 0 to v of length $\leq \texttt{dist\_to}[u] + e.\texttt{weight}$.

Relaxing A Path

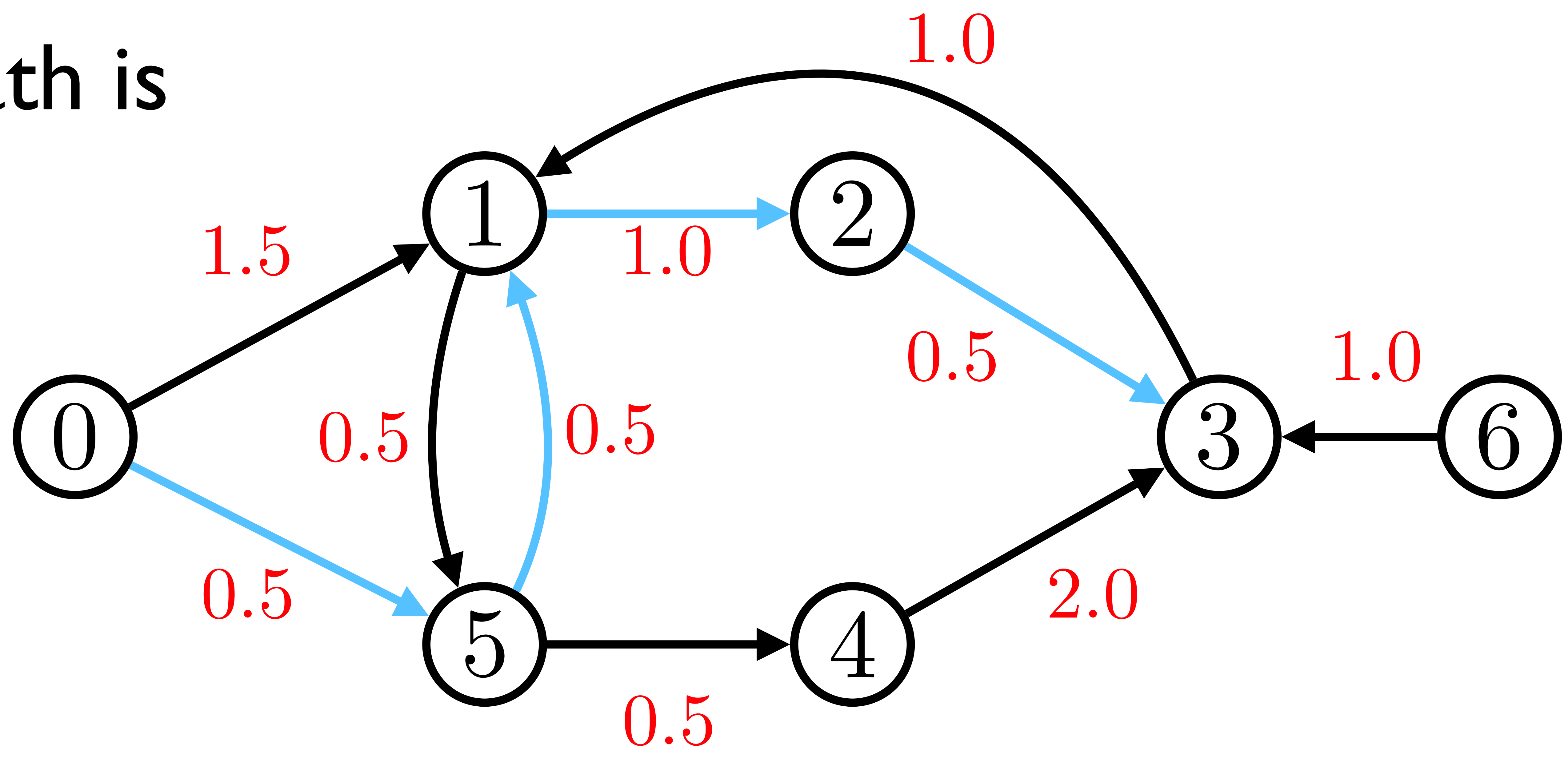
We say that a sequence of edge relaxations relaxes a path $e_1, \ldots, e_k$ if there is a subsequence that relaxes $e_1, \ldots, e_k$ in that order.

Consider the path $(0,5), (5,1), (1,2), (2,3)$.

An example sequence relaxing this path is

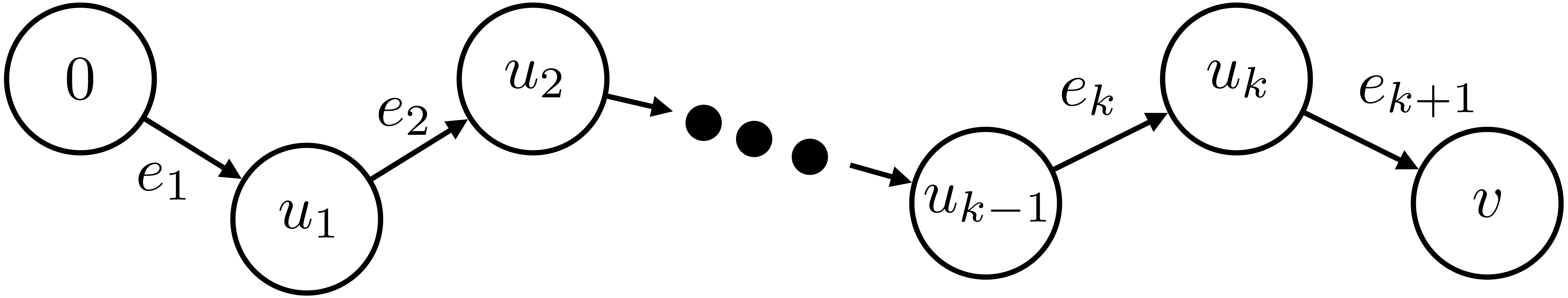
$(2,3)$ $(0,5)$ $(6,3)$ $(1,2)$ $(5,1)$

$(0,1)$ $(5,4)$ $(1,2)$ $(2,3)$



Relaxing a Shortest Path

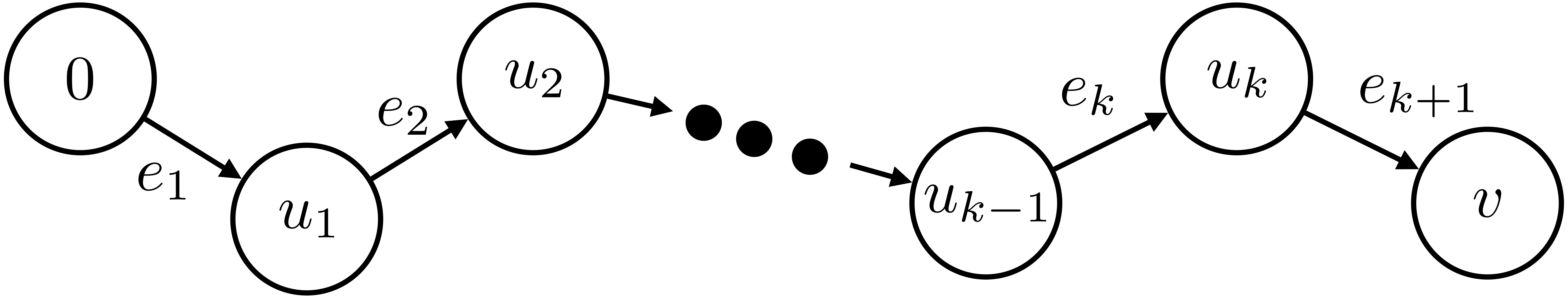
Relax a Path Property: If the algorithm relaxes a shortest path from 0 to v then $\texttt{dist\_to}[v] = d(0, v)$.



After relaxing e_1 we know

$$\texttt{dist\_to}[u_1] \leq e_1\texttt{.weight}$$

Relax a Path Property: If the algorithm relaxes a shortest path from 0 to v then $\texttt{dist\_to}[v] = d(0, v)$.



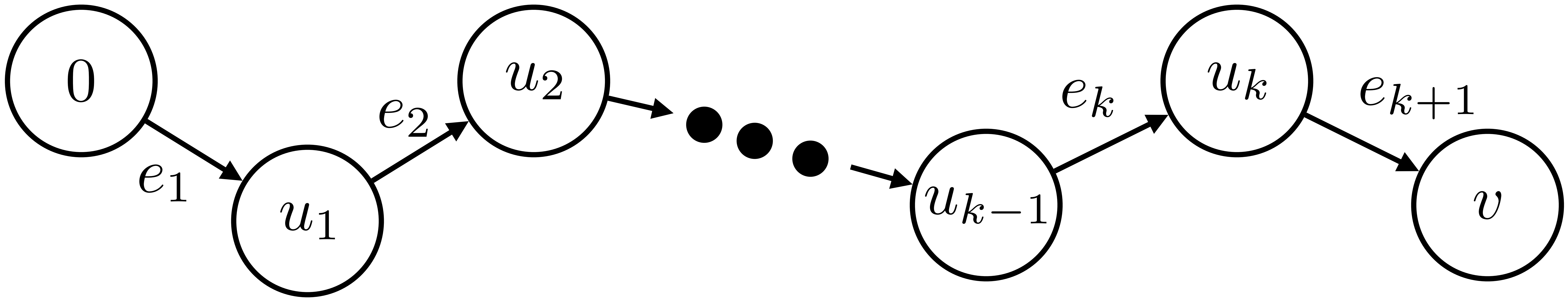
After relaxing e_1 we know

$$\texttt{dist\_to}[u_1] \leq e_1\texttt{.weight}$$

Later when we relax e_2 we will have

$$\texttt{dist\_to}[u_2] \leq \texttt{dist\_to}[u_1] + e_2\texttt{.weight}$$

Relax a Path Property: If the algorithm relaxes a shortest path from 0 to v then $\texttt{dist\_to}[v] = d(0, v)$.



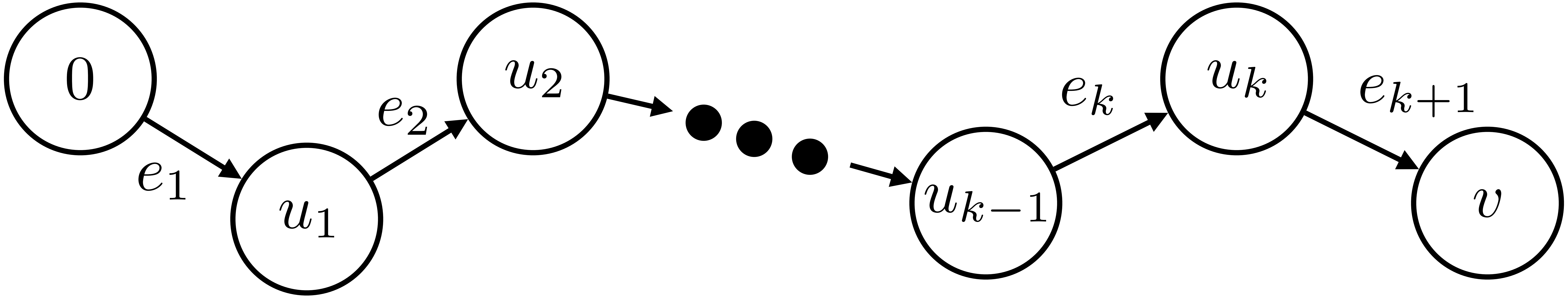
After relaxing e_1 we know

$$\texttt{dist\_to}[u_1] \leq e_1.\texttt{weight}$$

Later when we relax e_2 we will have

$$\texttt{dist\_to}[u_2] \leq e_1.\texttt{weight} + e_2.\texttt{weight}$$

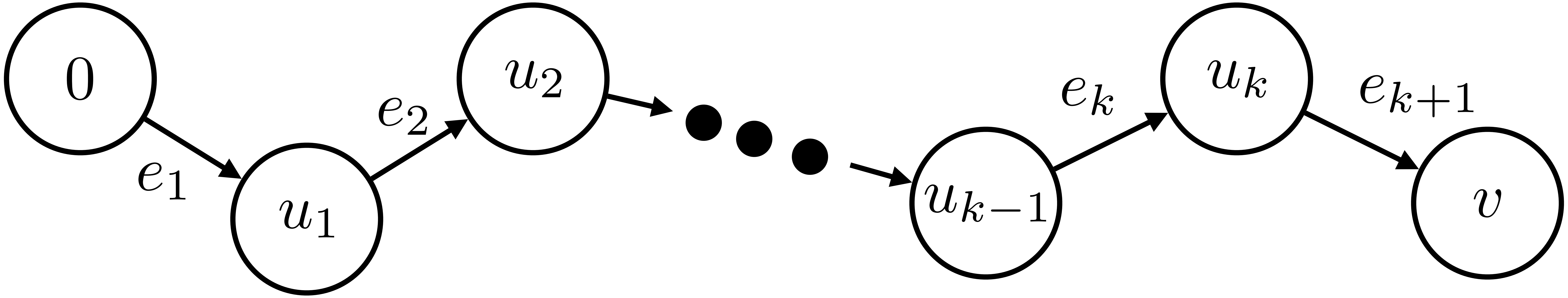
Relax a Path Property: If the algorithm relaxes a shortest path from 0 to v then $\texttt{dist\_to}[v] = d(0, v)$.



Later when we relax e_3 we will have

$$\texttt{dist\_to}[u_3] \leq e_1.\texttt{weight} + e_2.\texttt{weight} + e_3.\texttt{weight}$$

Relax a Path Property: If the algorithm relaxes a shortest path from 0 to v then $\texttt{dist\_to}[v] = d(0, v)$.



Later when we relax e_{k+1} we will have

$$\begin{aligned}\texttt{dist\_to}[v] &\leq \sum_{i=1}^{k+1} e_i.\texttt{weight} \\ &= d(0, v)\end{aligned}$$

Generic Template

Relax a sequence of edges that relaxes a shortest path from the source vertex 0 to every other vertex reachable from 0.

What is left up to the implementation is how to choose this sequence.

We will see three implementations of this template

Generic Template

Relax a sequence of edges that relaxes a shortest path from the source vertex 0 to every other vertex reachable from 0.

Bellman-Ford: Graphs without negative cycles

Do $n-1$ rounds of relaxing every edge.

Shortest paths in a DAG:

Relax the edges in topologically sorted order

Dijkstra's Algorithm: Graphs with positive edge weights

Relax edges in order of distance of the origin of the edge from the source.

Bellman-Ford Algorithm

Bellman-Ford Algorithm

Let's say we have a directed and weighted graph with n vertices.

We want to find shortest paths from the source vertex 0 to every other vertex reachable from 0.

Suppose that there is no negative-weight cycle reachable from vertex 0.

The basic code for the Bellman-Ford algorithm is beautifully simple.

Bellman-Ford Algorithm

```
for(int i=0; i < n-1; ++i) {
    for every edge e {
        relax(e);
    }
}
```

pseudocode for Bellman-Ford loop

We perform $n - 1$ rounds of relaxing every edge in the graph.

Relax Function

To relax the edge $e = (u, v)$ we check if

$$\texttt{dist\_to}[u] + e.\texttt{weight} < \texttt{dist\_to}[v]$$

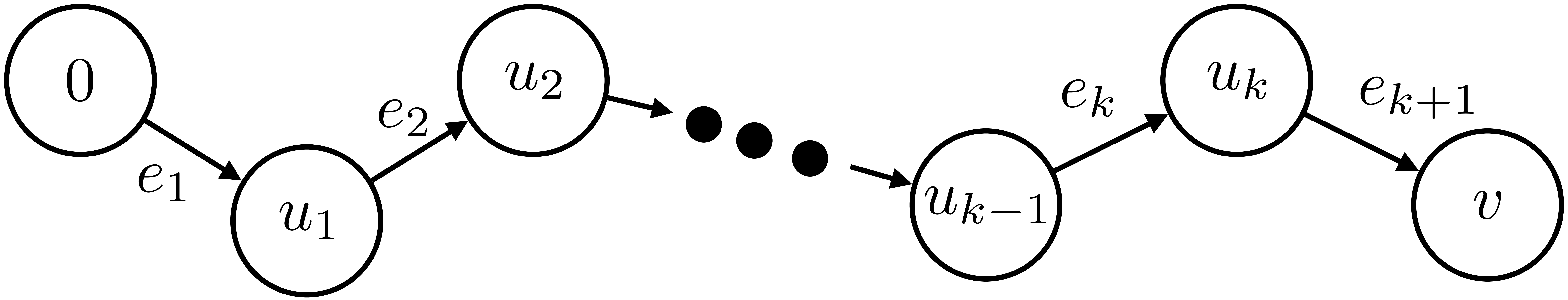
If so, then we do the updates:

$$\texttt{dist\_to}[v] = \texttt{dist\_to}[u] + e.\texttt{weight}$$

$$\texttt{edge\_to}[v] = u$$

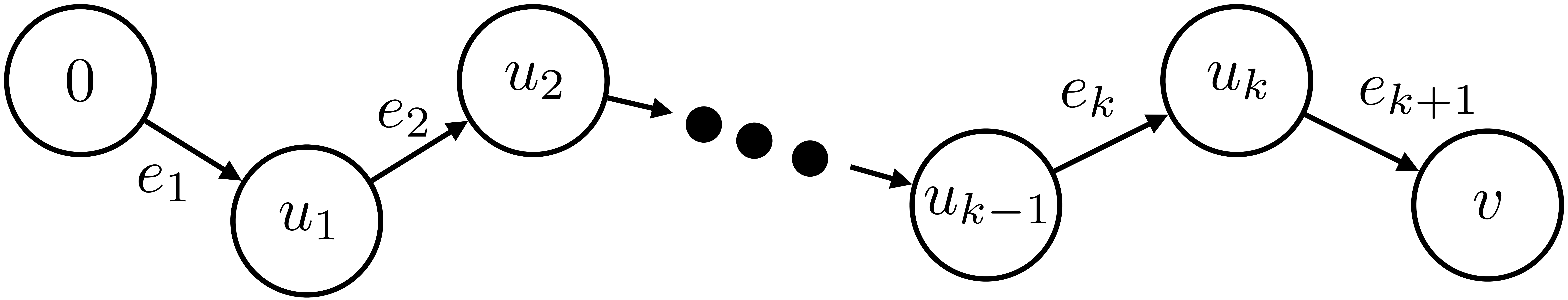
Why it Works

If there is no negative-weight cycle reachable from vertex 0, then all shortest paths from vertex 0 are simple paths with at most $n-1$ edges.



If this is a shortest path from 0 to vertex v we know that $k+1 \leq n-1$.

Why it Works

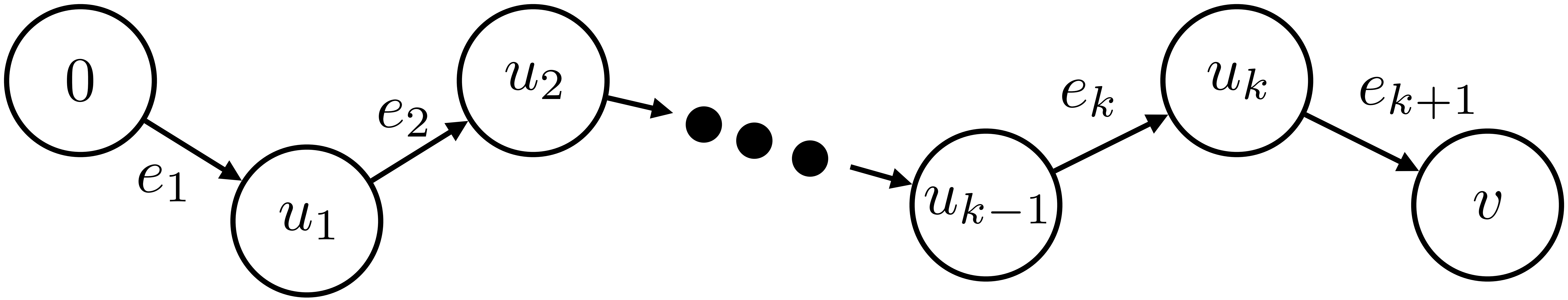


In the first round of relaxations we relax edge e_1.

In the second round of relaxations we relax edge e_2.

After $n - 1$ rounds of relaxations, we will have relaxed this shortest path.

Why it Works



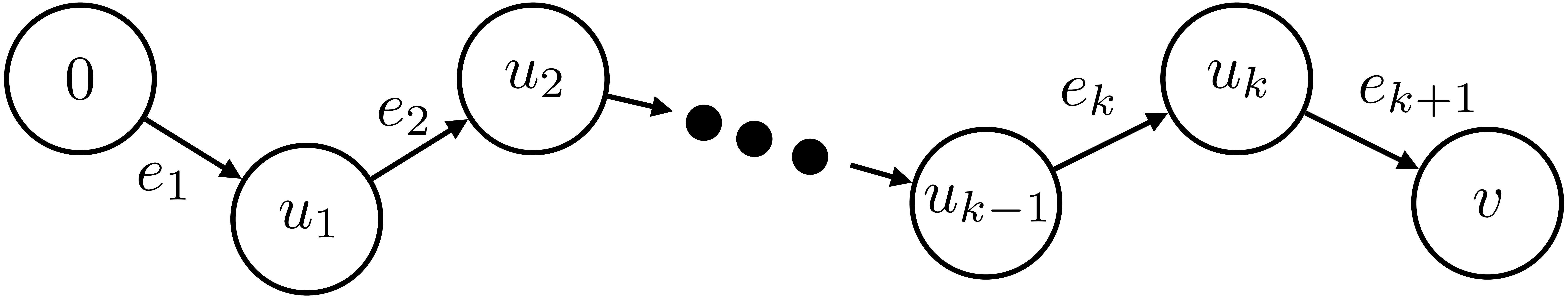
Relax a Path Property: If the algorithm relaxes a shortest path from 0 to then $\texttt{dist\_to}[v] = d(0, v)$.

At the end of the Bellman-Ford algorithm we will have

$$\texttt{dist\_to}[v] = d(0, v)$$

for any vertex v.

Why it Works



At the end of the Bellman-Ford algorithm we will have

$$\texttt{dist\_to}[v] = d(0, v)$$

for any vertex v.

Invariant 2: Tracing back $i \leftarrow \texttt{edge\_to}[i] \leftarrow \cdots \leftarrow 0$ gives a path from vertex 0 to i of length $\leq \texttt{dist\_to}[i]$.

Running Time

```
for(int i=0; i < n-1; ++i) {
    for every edge e {
        relax(e);
    }
}
```

pseudocode for Bellman-Ford loop

The running time of the Bellman-Ford algorithm is $O(|V| \cdot |E|)$ in the adjacency list model.

Relaxing an edge takes constant time.

Each iteration of the for loop takes time $O(|E|)$ in the adjacency list model.

Negative Cycles

We have already established when there are no negative weight cycles reachable from the source 0, after $|V| - 1$ iterations of the for loop

$$d(0, i) = \texttt{dist\_to}[i]$$

for every vertex i.

When there are no negative weight cycles reachable from 0, the `dist_to` values will not change on doing more iterations of the for loop by invariant I.

Fact: The input graph has a negative weight cycle reachable from 0 if and only if some `dist_to` value decreases in a $|V|^{th}$ iteration of the for loop.

Negative Cycles

Fact: The input graph has a negative weight cycle reachable from 0 if and only if some `dist_to` value decreases in a $|V|^{th}$ iteration of the for loop.

To detect a negative weight cycle we do one more iteration of the for loop and check if any `dist_to` value decreases.

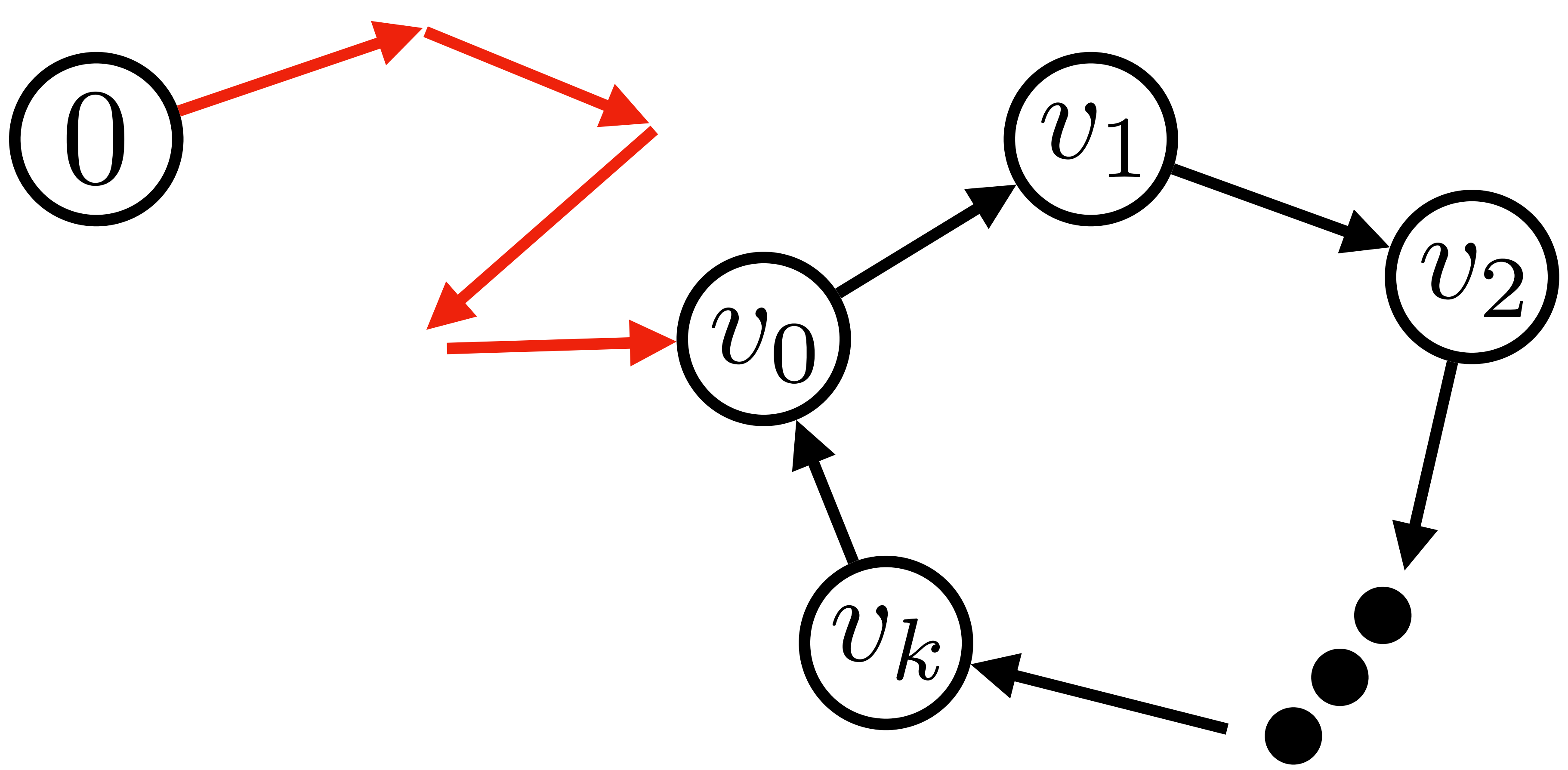
The graph defined by the `edge_to` array will contain the negative weight cycle if there is one.

We can use our directed cycle algorithm to find it.

Negative Cycles

Fact: The input graph has a negative weight cycle reachable from 0 if and only if some `dist_to` value decreases in a $|V|^{th}$ iteration of the for loop.

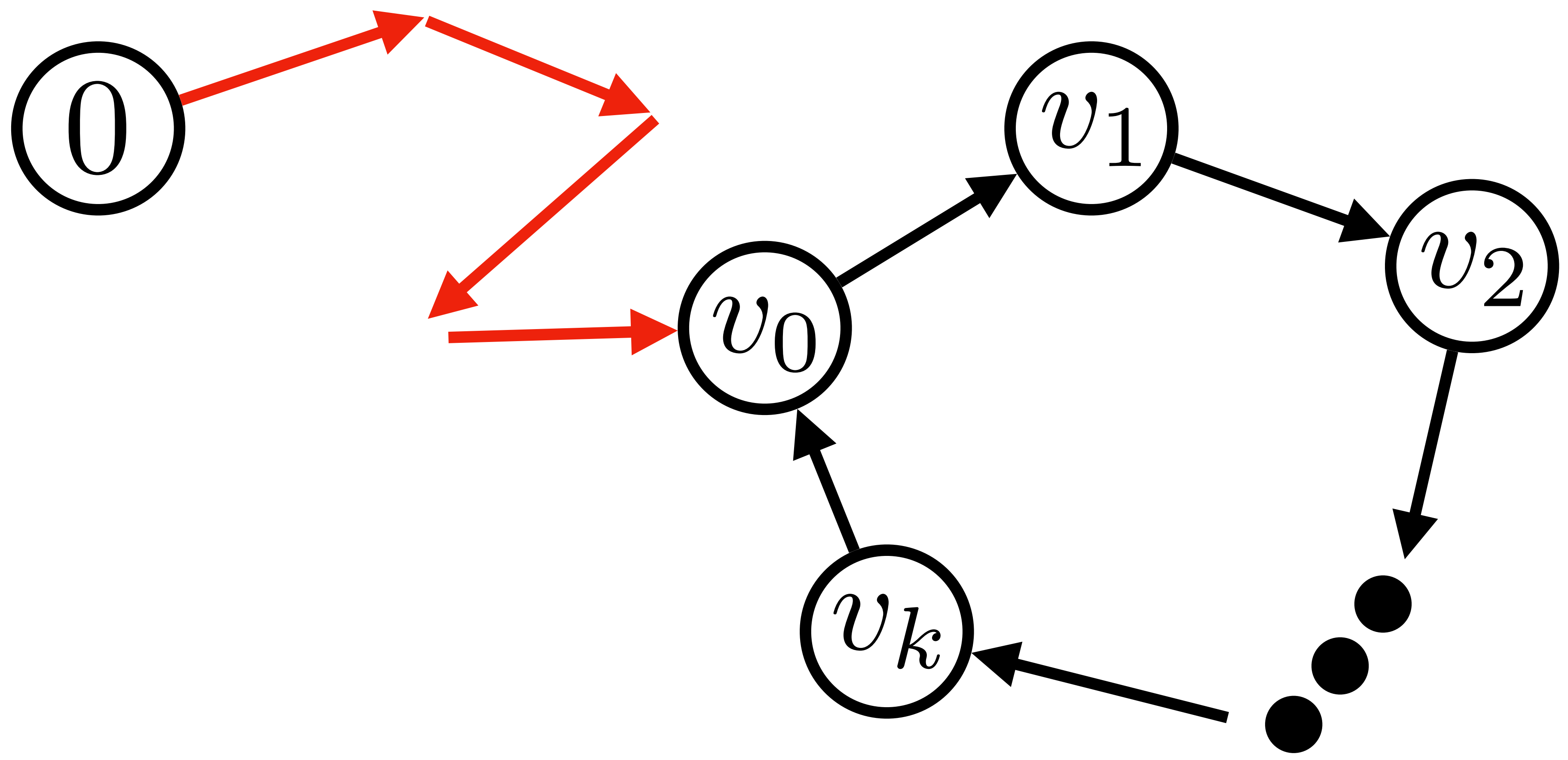
We have to see that if the graph has a negative weight cycle then some `dist_to` value decreases in the $|V|^{th}$ iteration.



Suppose this is a negative weight cycle reachable from 0.

$$\sum_{i=0}^{k} (v_{i-1}, v_i).\texttt{weight} < 0$$

We have to see that if the graph has a negative weight cycle then some $\texttt{dist\_to}$ value decreases in the $|V|^{th}$ iteration.



Let $\texttt{dist\_to}[v_i]$ be the values after $|V|-1$ iterations.

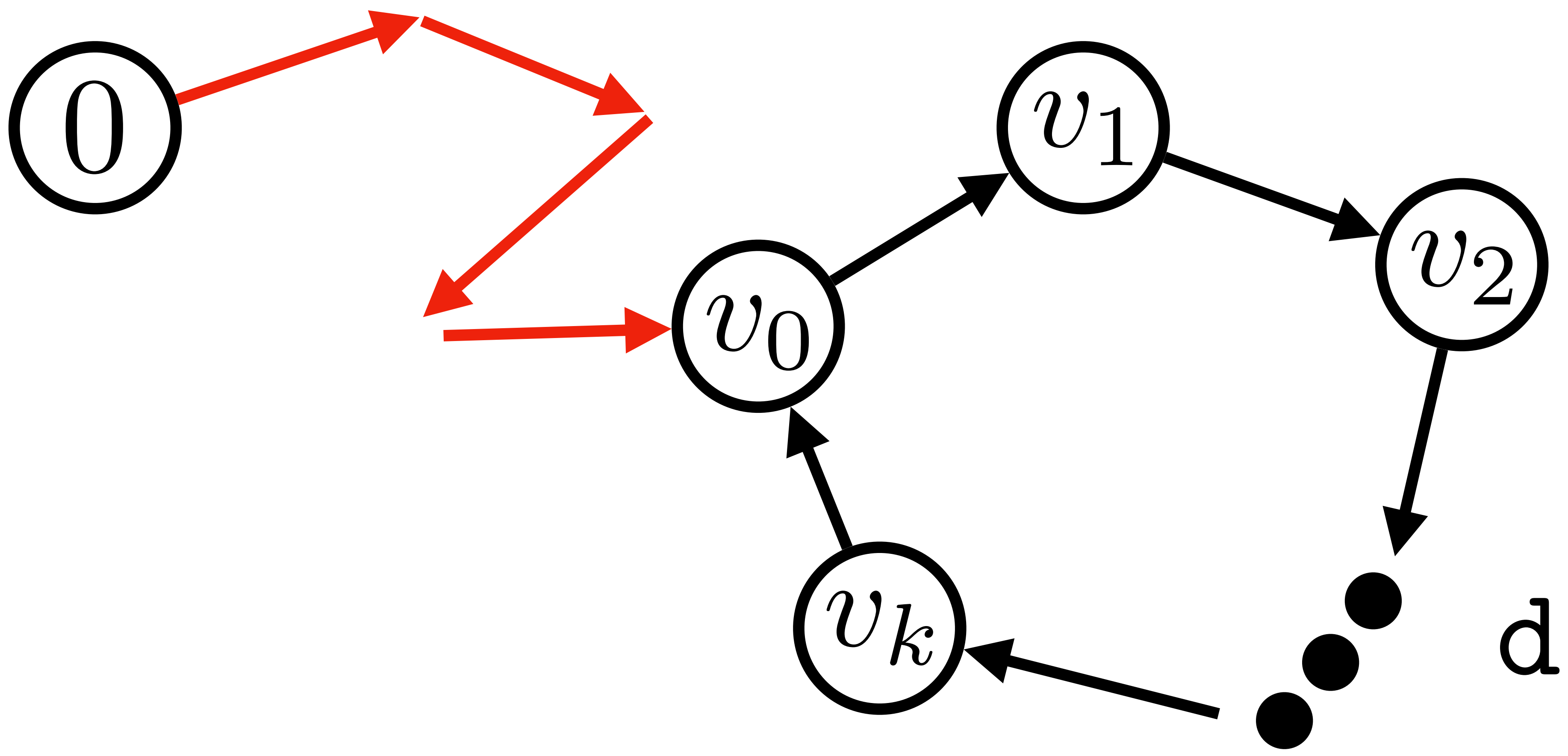
All these values are finite since the cycle is reachable.

Let $\texttt{dist\_to}'[v_i]$ be the values after $|V|$ iterations.

After relaxing all edges in the $|V|^{th}$ round:

$$\texttt{dist\_to}'[v_i] \leq \texttt{dist\_to}[v_{i-1}] + (v_{i-1}, v_i).\texttt{weight}$$

We have to see that if the graph has a negative weight cycle then some `dist_to` value decreases in the $|V|^{th}$ iteration.



$$\texttt{dist\_to}'[v_i] \leq \texttt{dist\_to}[v_{i-1}] + (v_{i-1}, v_i).\texttt{weight}$$

Summing this over the cycle gives

$$\sum_{i=0}^{k} \texttt{dist\_to}'[v_i] \leq \sum_{i=0}^{k} \texttt{dist\_to}[v_{i-1}] + (v_{i-1}, v_i).\texttt{weight}$$

We have to see that if the graph has a negative weight cycle then some `dist_to` value decreases in the $|V|^{th}$ iteration.

Summing this over the cycle gives

$$\sum_{i=0}^{k} \texttt{dist\_to}'[v_i] \leq \sum_{i=0}^{k} \texttt{dist\_to}[v_{i-1}] + (v_{i-1}, v_i).\texttt{weight}$$

If $\texttt{dist\_to}'[v_i] = \texttt{dist\_to}[v_i]$ for all i then these terms cancel. This implies

$$0 \leq \sum_{i=0}^{k} (v_{i-1}, v_i).\texttt{weight}$$

a contradiction to this being a negative weight cycle.

Dijkstra's Algorithm

Dijkstra's Algorithm

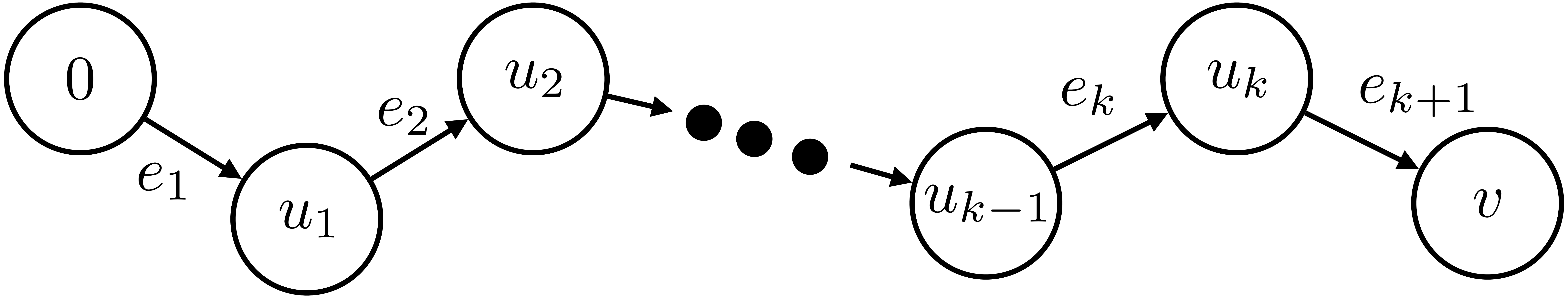
The final application of the generic shortest path algorithm we look at is Dijkstra's algorithm.

This solves the single-source shortest path problem when all edge weights are positive.

Dijkstra's algorithm follows the generic template and processes vertices in order of their distance from the source.

When we process a vertex we relax all its outgoing edges.

Why it Works

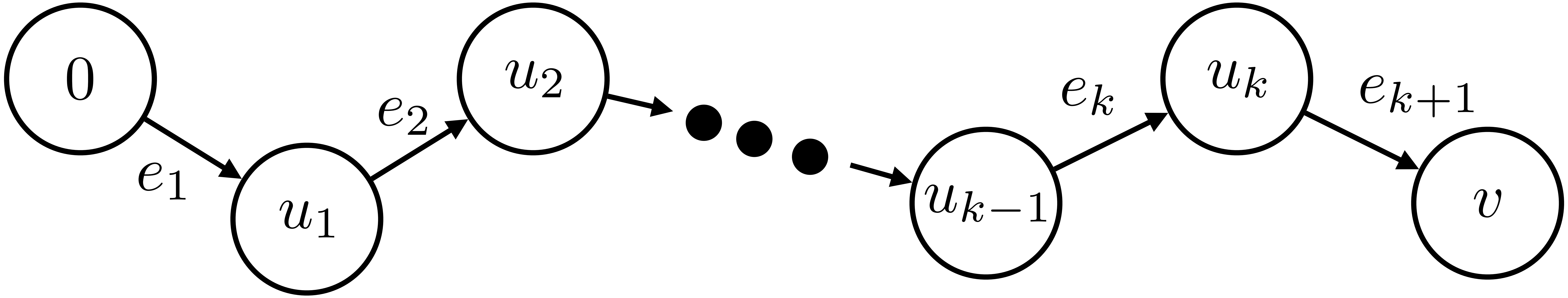


Suppose this is a shortest path from 0 to vertex v.

By the optimal substructure of shortest paths:

$$d(0, u_i) = \sum_{j=1}^{i} e_j.\texttt{weight}$$

Why it Works

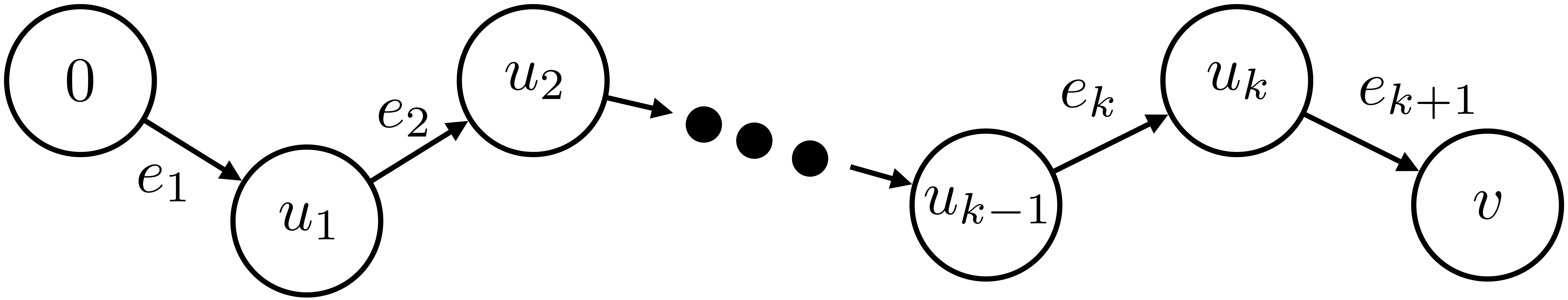


$$d(0, u_i) = \sum_{j=1}^{i} e_j.\texttt{weight}$$

All edge weights are positive, so this means

$$d(0, u_1) < d(0, u_2) < \cdots < d(0, u_k) < d(0, v)$$

Why it Works

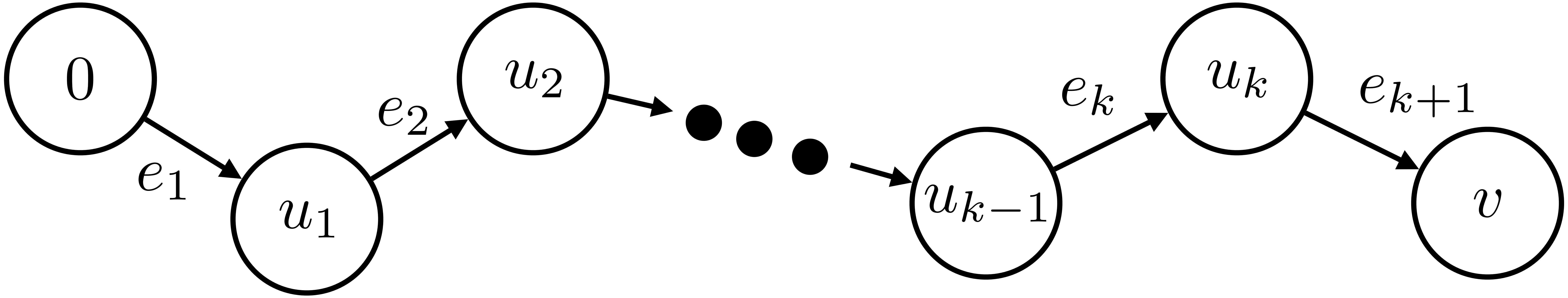


All edge weights are positive, so this means

$$d(0, u_1) < d(0, u_2) < \cdots < d(0, u_k) < d(0, v)$$

By relaxing the outgoing edges of vertices in order of the distance from 0, we will relax the edges on this path in order!

Why it Works



$$d(0, u_1) < d(0, u_2) < \cdots < d(0, u_k) < d(0, v)$$

By relaxing the outgoing edges of vertices in order of their distance from 0 we will relax this path.

Relax a Path Property: If the algorithm relaxes a shortest path from 0 to then $\texttt{dist\_to}[v] = d(0, v)$.

Dijkstra Implementation

The implementation of Dijkstra's algorithm is very similar to that of Prim's algorithm for finding a minimum spanning tree.

As in Prim we maintain a subset S of vertices.

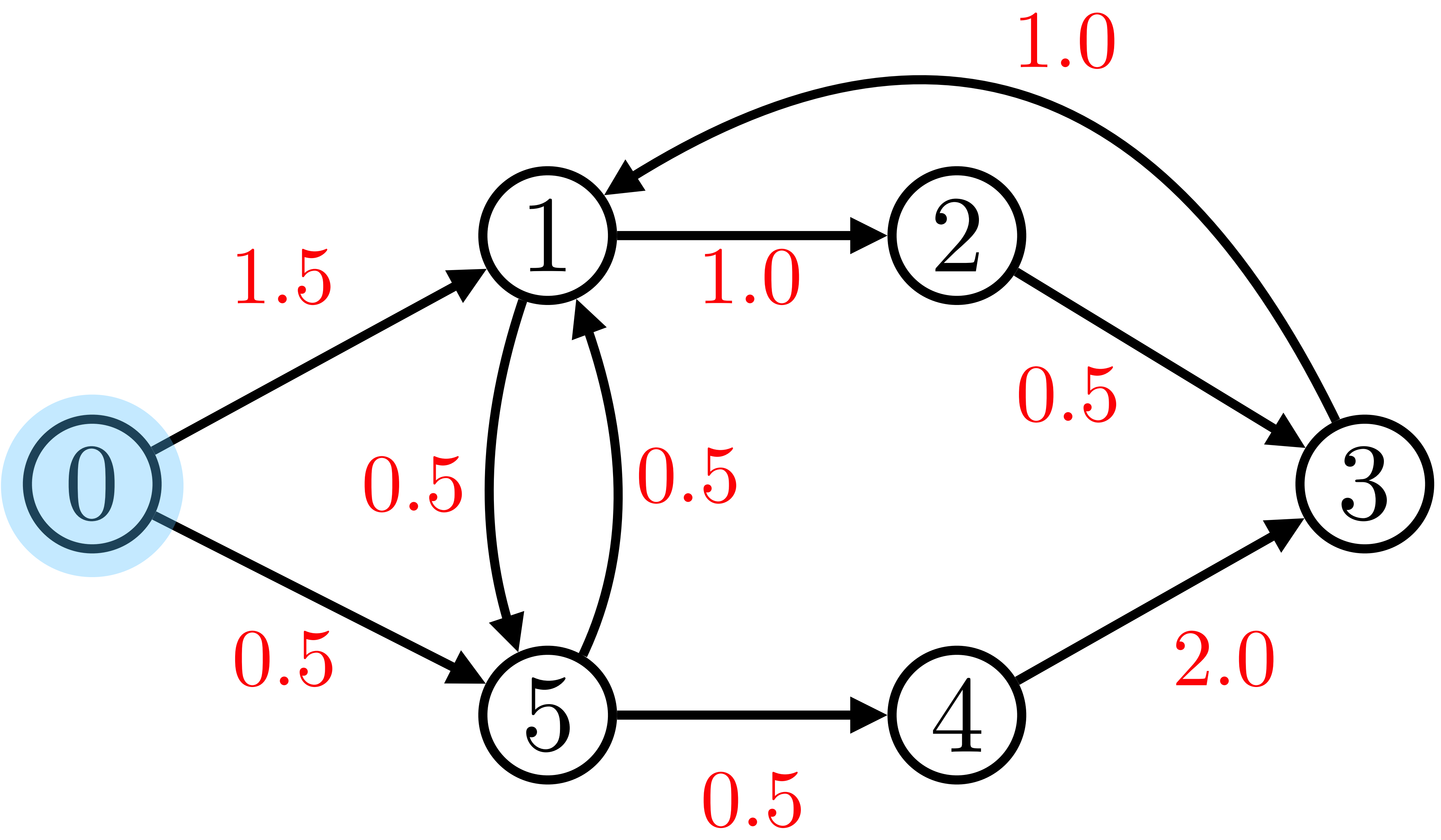
Initially $S = \{0\}$ consists just of the source vertex.

We want to maintain two invariants:

1) $d(0, i) = \texttt{dist\_to}[i]$ for all $i \in S$.

2) $d(0, i) \leq d(0, j)$ for all $i \in S, j \notin S$.

Dijkstra Example



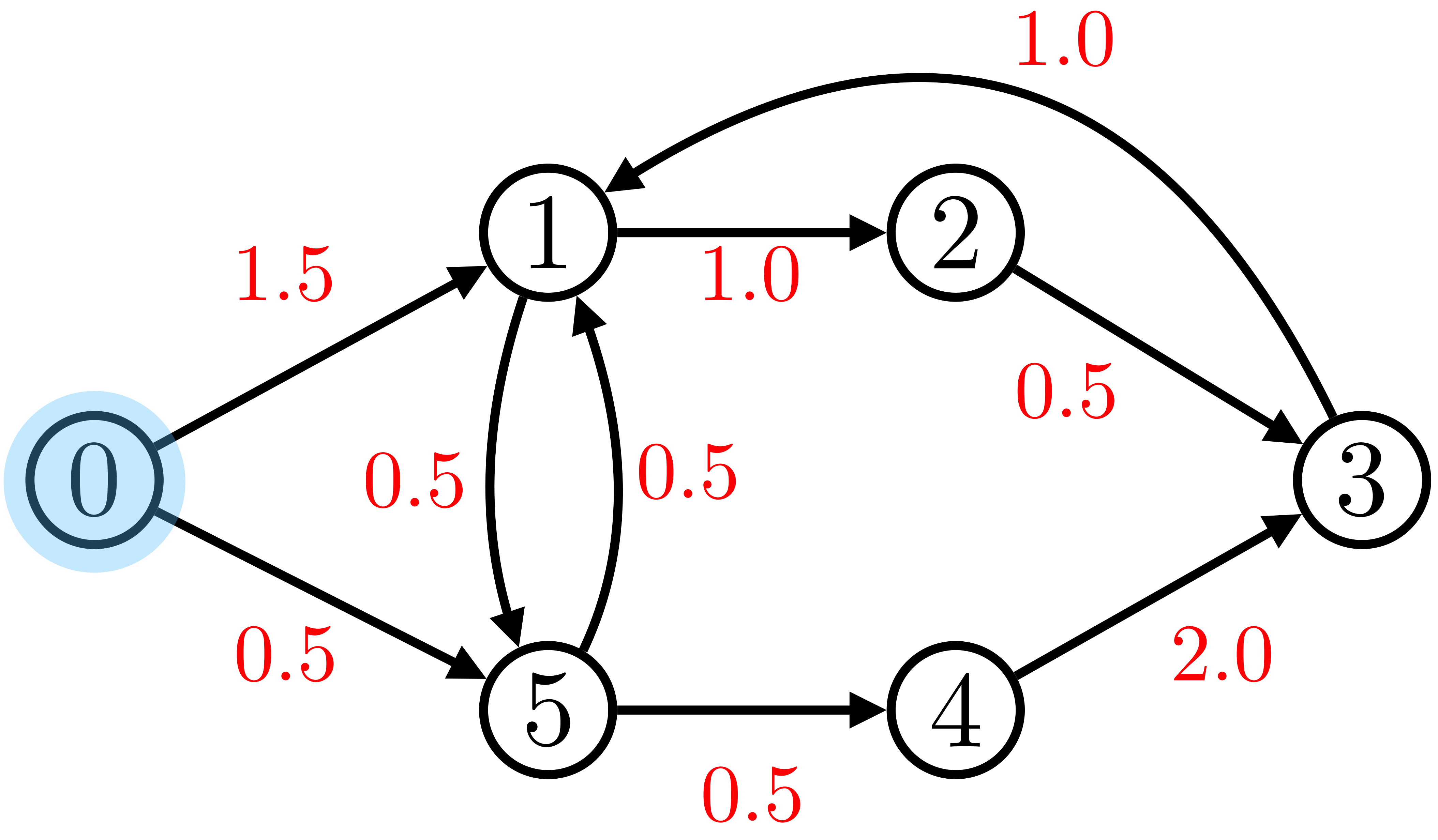
0	∞	∞	∞	∞	∞
0	1	2	3	4	5

`dist_to`

Let $S = \{0\}$ and initialize `dist_to` as usual.

We want to find the vertex closest to 0 that is not in S .

Key: Because we have positive weights, the closest vertex is an out-adjacent neighbor of 0.



0	1	2	3	4	5
0	∞	∞	∞	∞	∞

`dist_to`

$\{(0,5), 0.5\}$ $\{(0,1), 1.5\}$

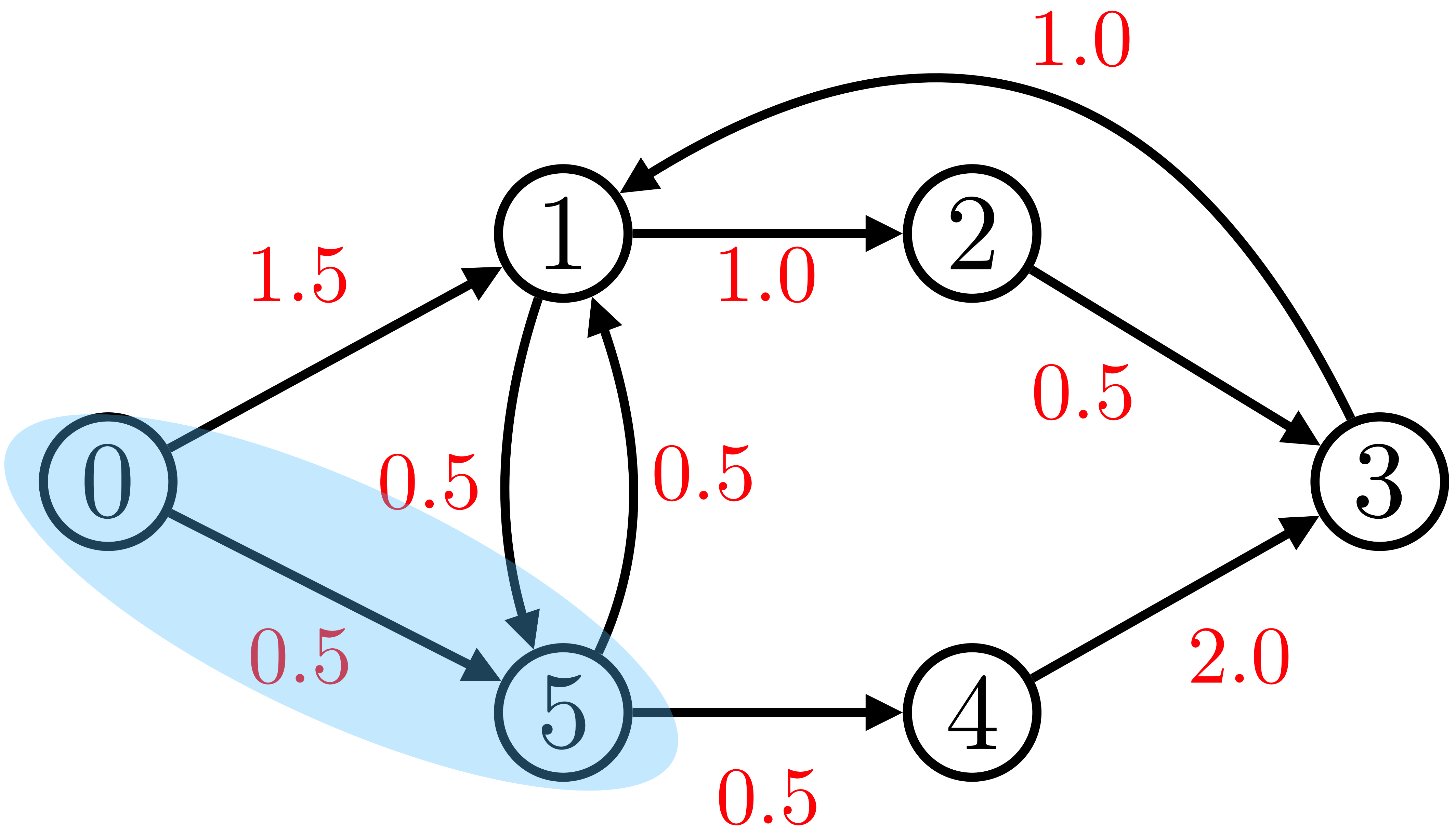
`priority_queue`

Add all outgoing edges from 0 to a minimum priority queue.

The key of edge e is

$$\texttt{dist\_to}[0] + e.\texttt{weight}$$

The top of the priority queue tells us the edge to follow to get the next vertex to add to S.



We pop the minimum element out of the queue:

$$\{(0,5), 0.5\}$$

Vertex 5 is not in S so we add it, and update $\texttt{dist\_to}[5]$.

0	1	2	3	4	5
0	∞	∞	∞	∞	0.5

`dist_to`

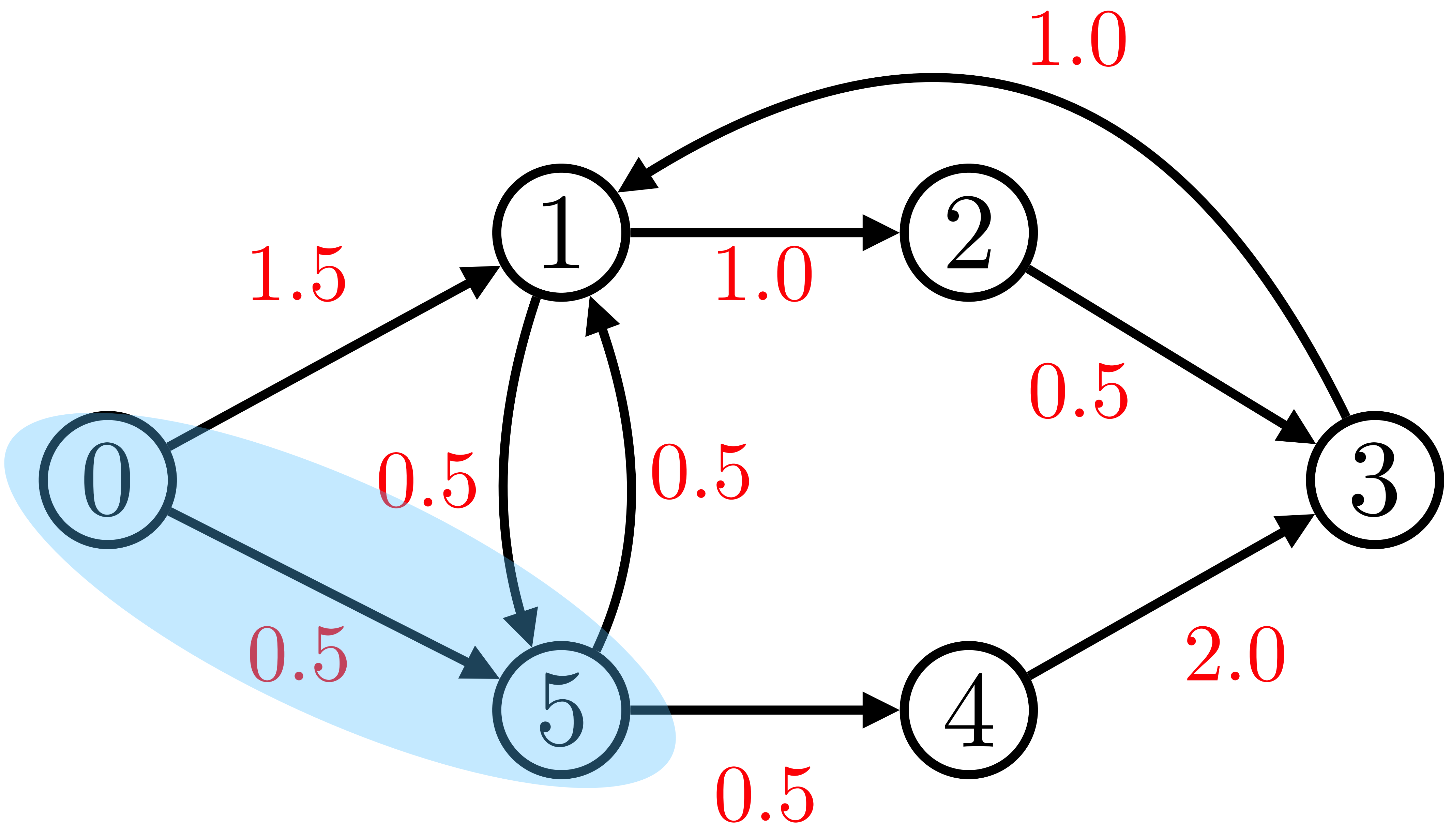
$$\{(0,1), 1.5\}$$

`priority_queue`

Our invariants still hold:

1) Vertex 5 is closer to 0 than any other vertex not in S .

2) $d(0,5) = \texttt{dist\_to}[5].$



0	∞	∞	∞	∞	0.5
0	1	2	3	4	5

`dist_to`

$$\{(0, 1), 1.5\}$$

`priority_queue`

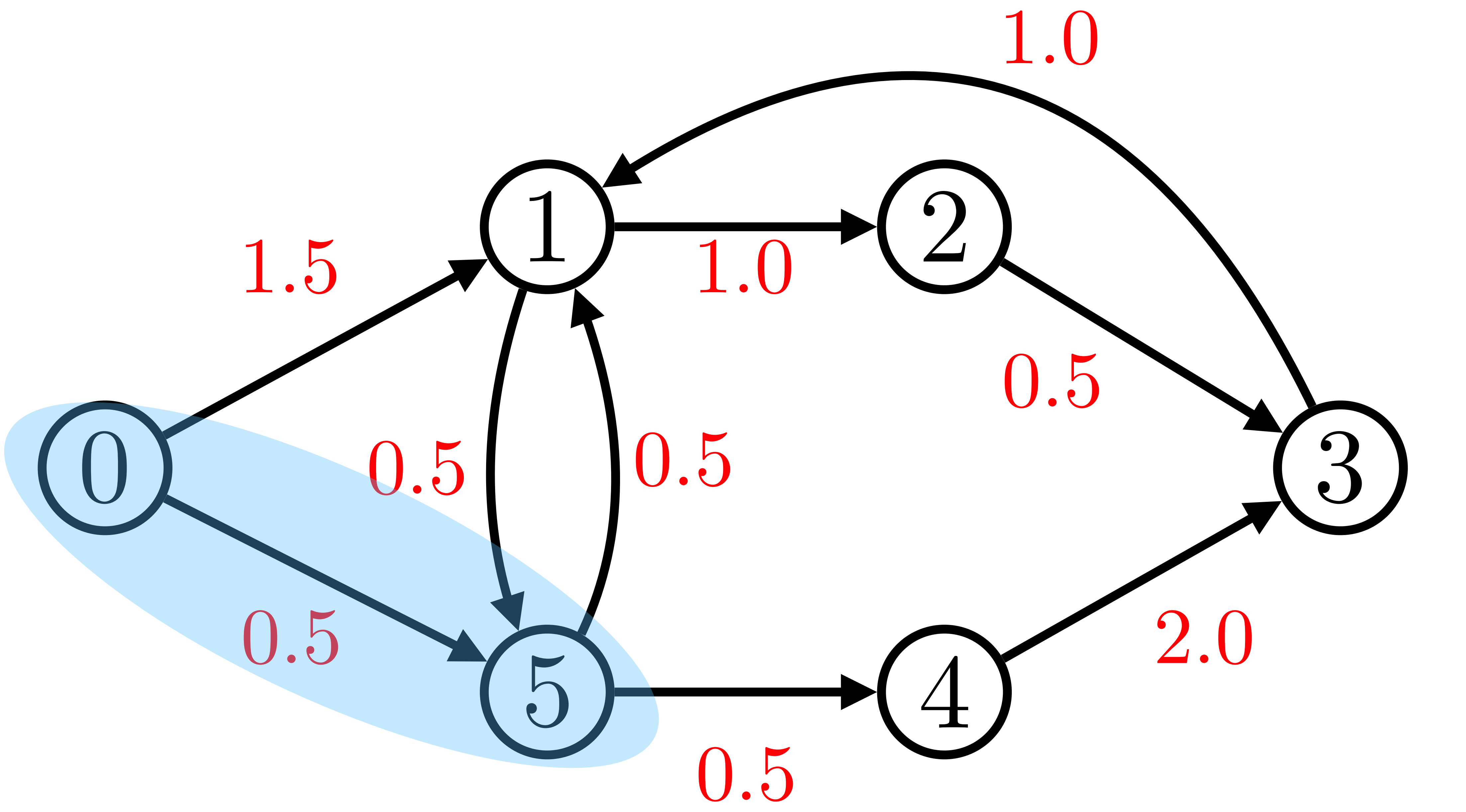
Next add all outgoing edges of vertex 5 that leave S to the priority queue.

Edges to another vertex in S are not useful.

The key for edge e is

$$\texttt{dist\_to}[5] + e.\texttt{weight}$$

We add $\{(5, 1), 1\}, \{(5, 4), 1\}$ to the queue.



0	∞	∞	∞	∞	0.5
0	1	2	3	4	5

`dist_to`

$$\{(5,1),1\}\{(5,4),1\}\ \{(0,1),1.5\}$$

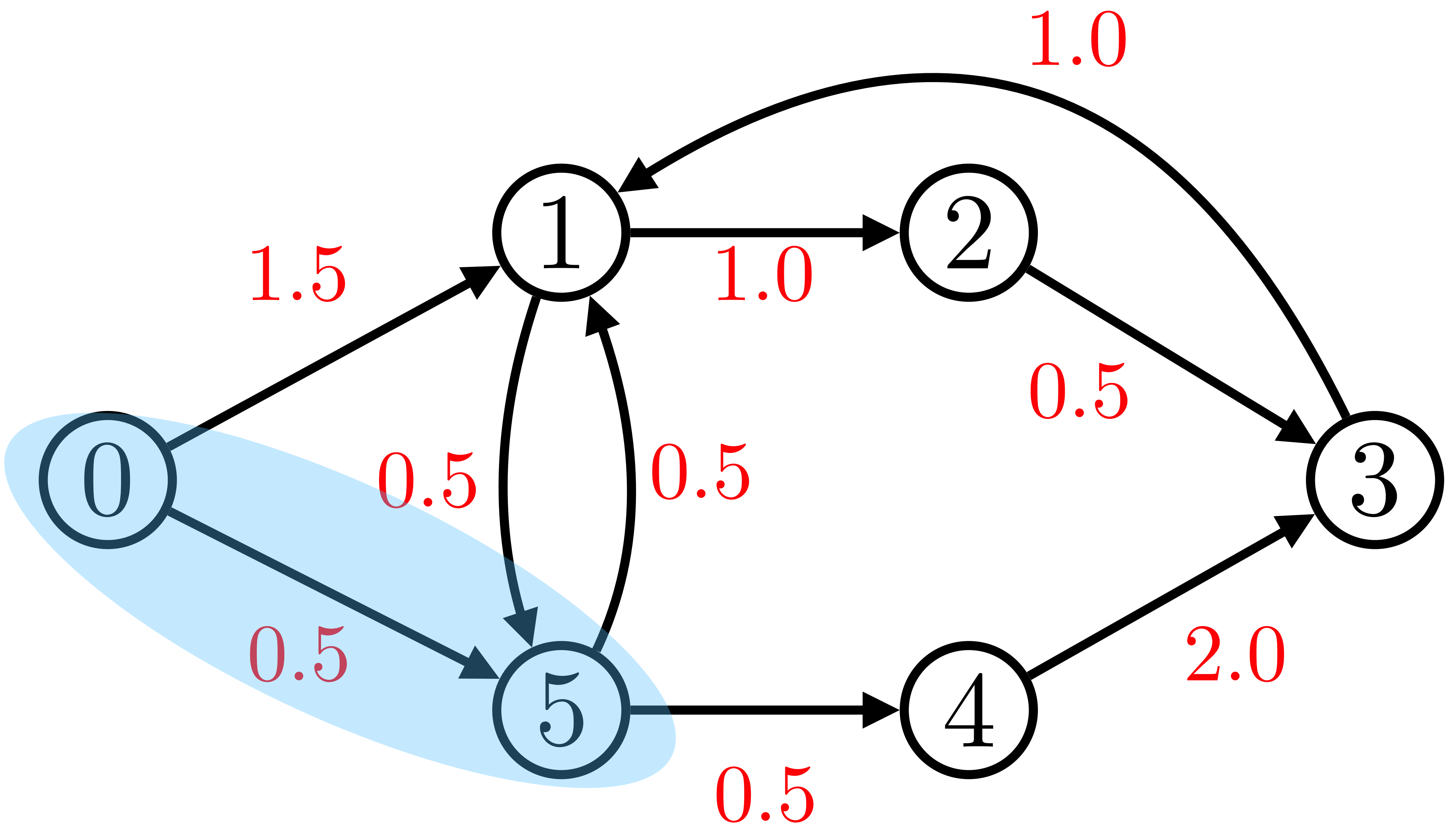
`priority_queue`

This is our current status.

The set $S = \{0, 5\}$.

The priority queue has 3 elements.

We go to the next round and
pop the top element out
of the priority queue.



0	1	2	3	4	5
0	∞	∞	∞	∞	0.5

`dist_to`

$$\{(5,4),1\} \;\; \{(0,1),1.5\}$$

`priority_queue`

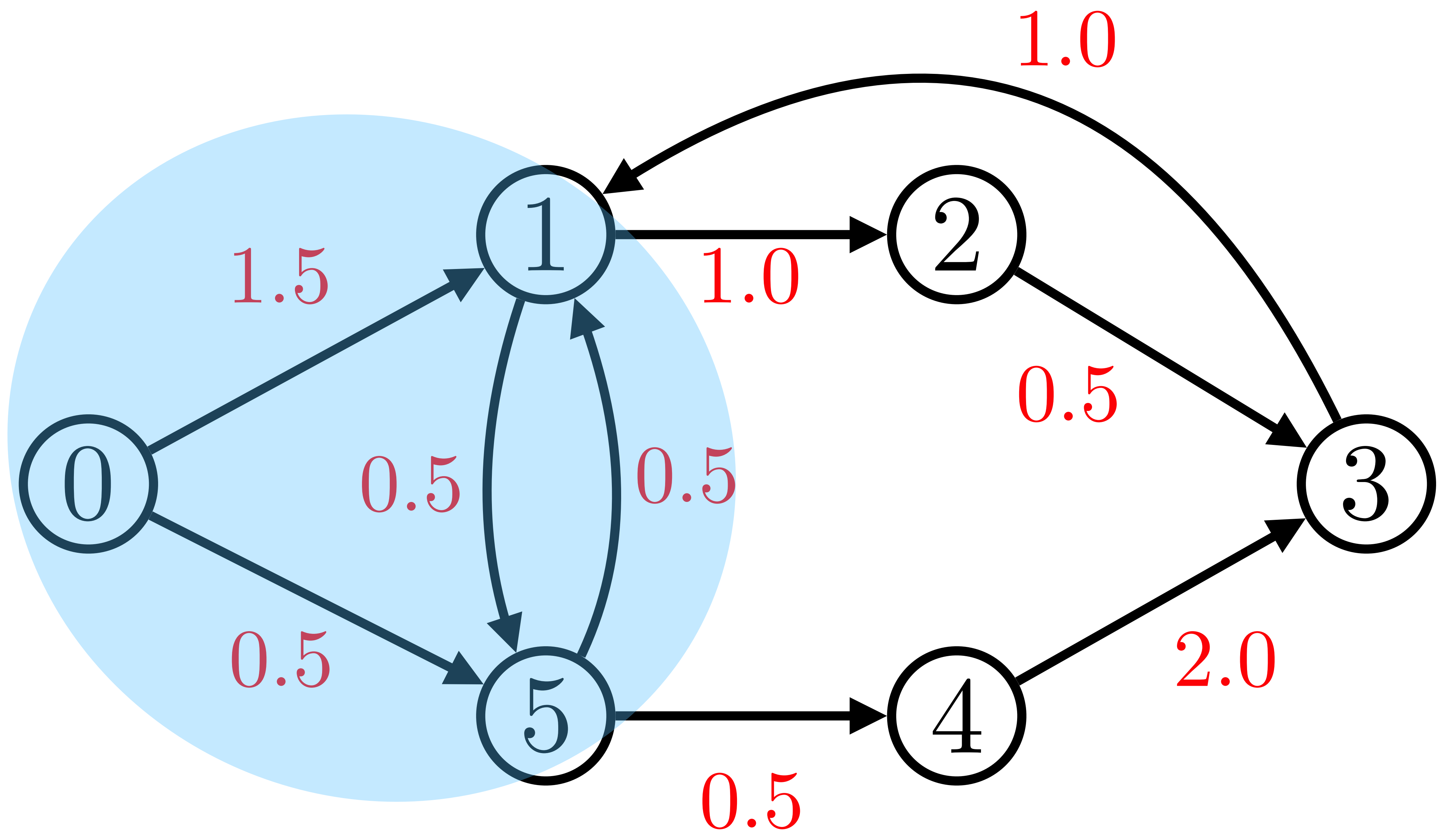
Popping gives the element

$$\{(5,1),1\}$$

The destination vertex 1 is not already in S, so we process it.

We update $\texttt{dist\_to}[1] = 1$ which is the correct distance.

We then add all outgoing edges from 1 that leave S to the priority queue.



We have updated $\texttt{dist\_to}[1]$ and added $\{(1,2),2\}$ to the priority queue.

The key value 2 is

$$\texttt{dist\_to}[1] + (1,2).\texttt{weight}$$

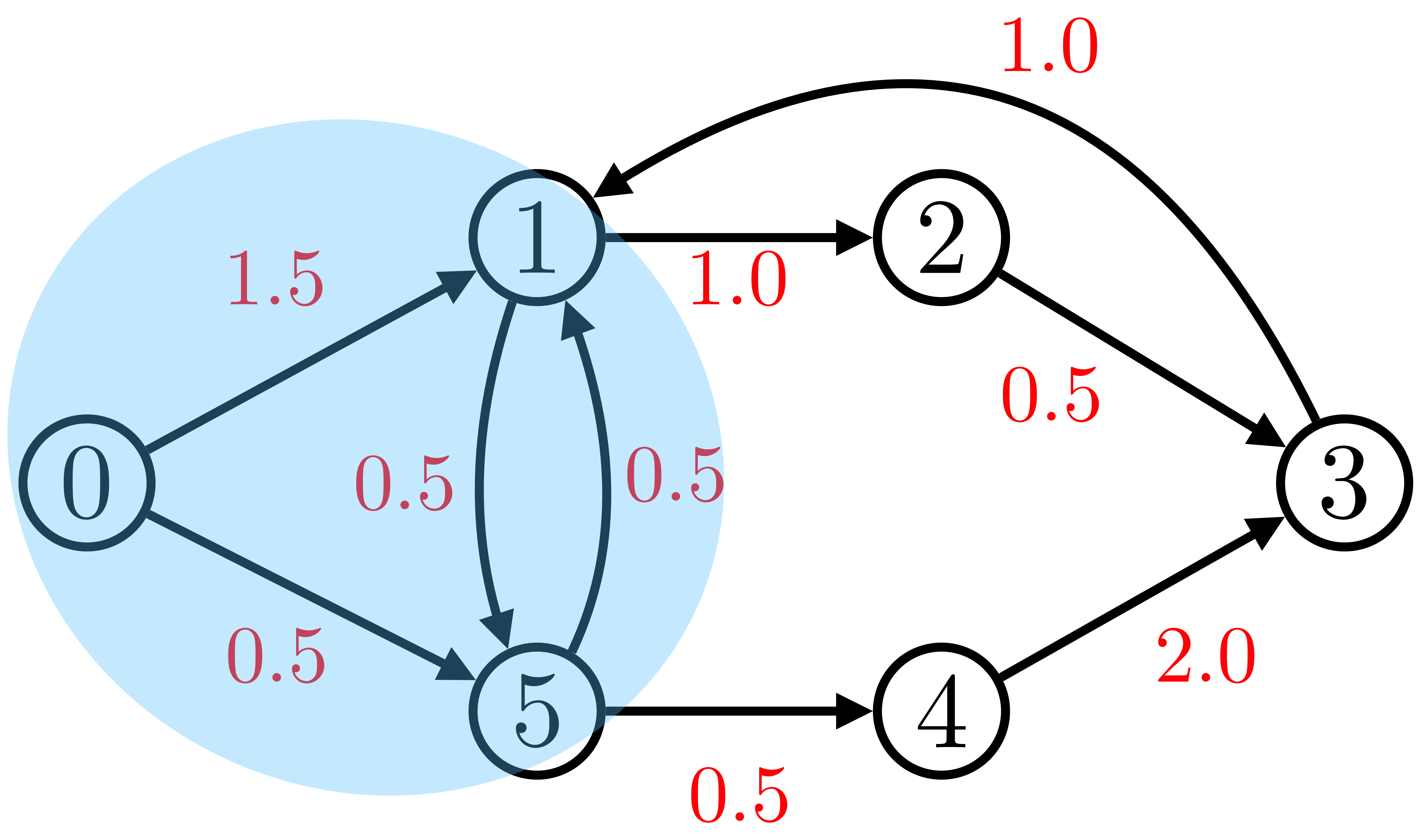
The next element to be popped out of the queue is $\{(5,4),1\}$.

0	1	2	3	4	5
0	1.0	∞	∞	∞	0.5

$\texttt{dist\_to}$

$\{(5,4),1\}$ $\{(0,1),1.5\}$ $\{(1,2),2\}$

$\texttt{priority\_queue}$



0	1.0	∞	∞	∞	0.5
0	1	2	3	4	5

dist_to

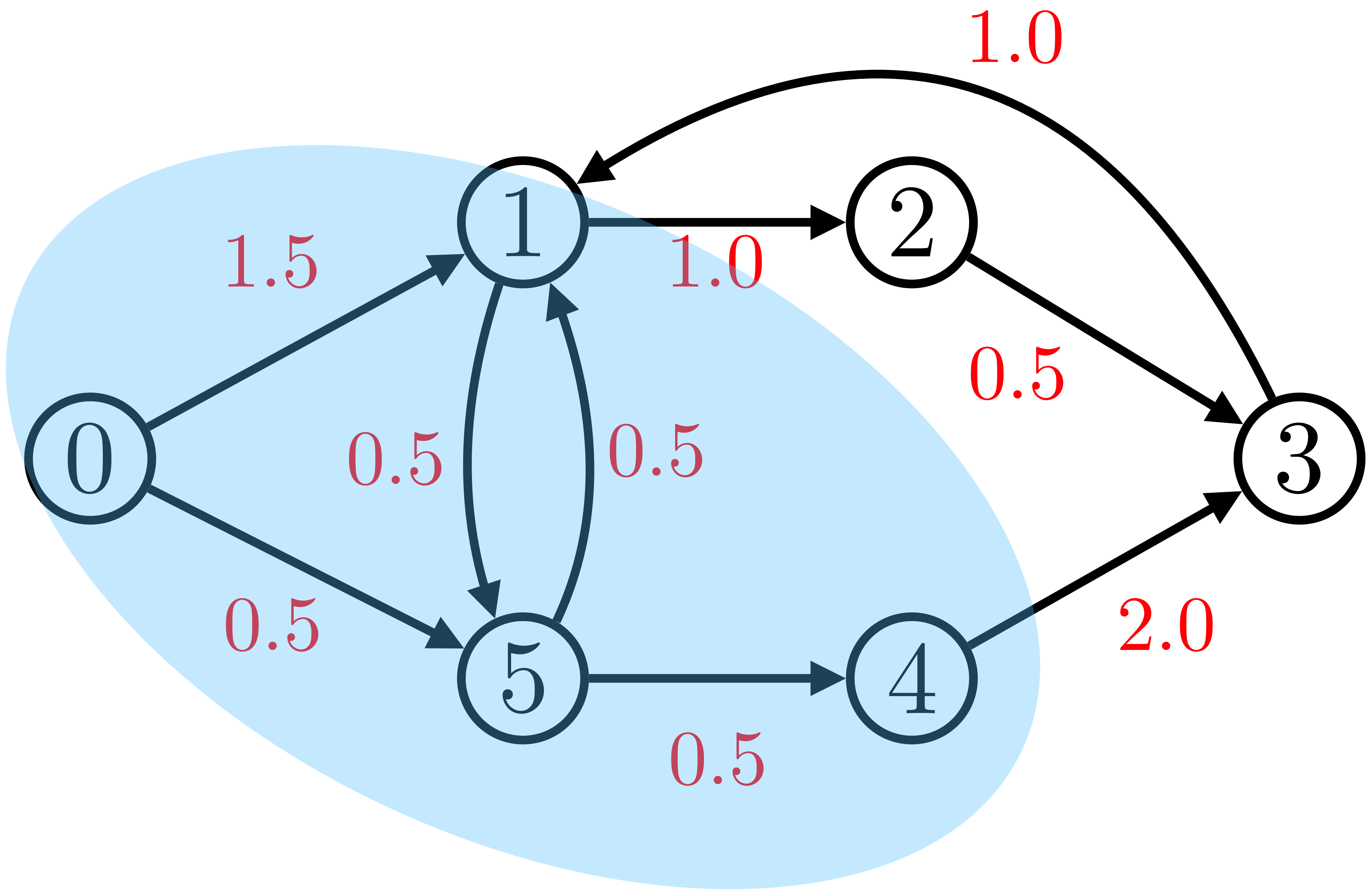
$\{(0,1),1.5\}$ $\{(1,2),2\}$

priority_queue

Pop the element $\{(5,4),1\}$ out of the queue.

The destination vertex 4 is not in S so we add it and update

$$\texttt{dist\_to}[4] = 1$$



0	1.0	∞	∞	1.0	0.5
0	1	2	3	4	5

`dist_to`

$\{(0,1),1.5\}$ $\{(1,2),2\}$ $\{(4,3),3\}$

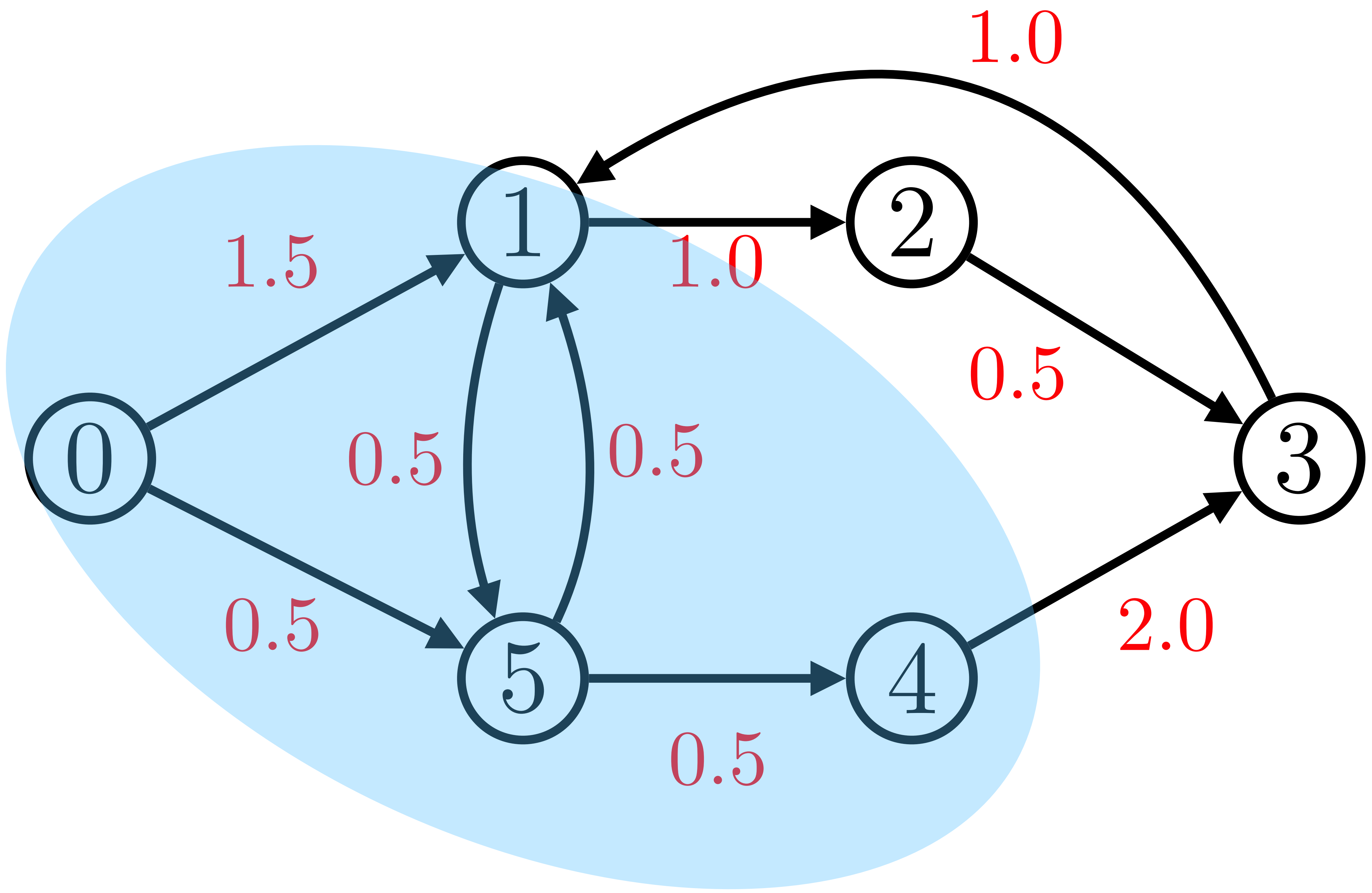
`priority_queue`

Now we process vertex 4.

We add all its outgoing edges e leaving S to the queue with key value

$$\texttt{dist\_to}[4] + e.\texttt{weight}$$

We add $\{(4,3),3\}$ to the queue.



0	1.0	∞	∞	1.0	0.5
0	1	2	3	4	5

`dist_to`

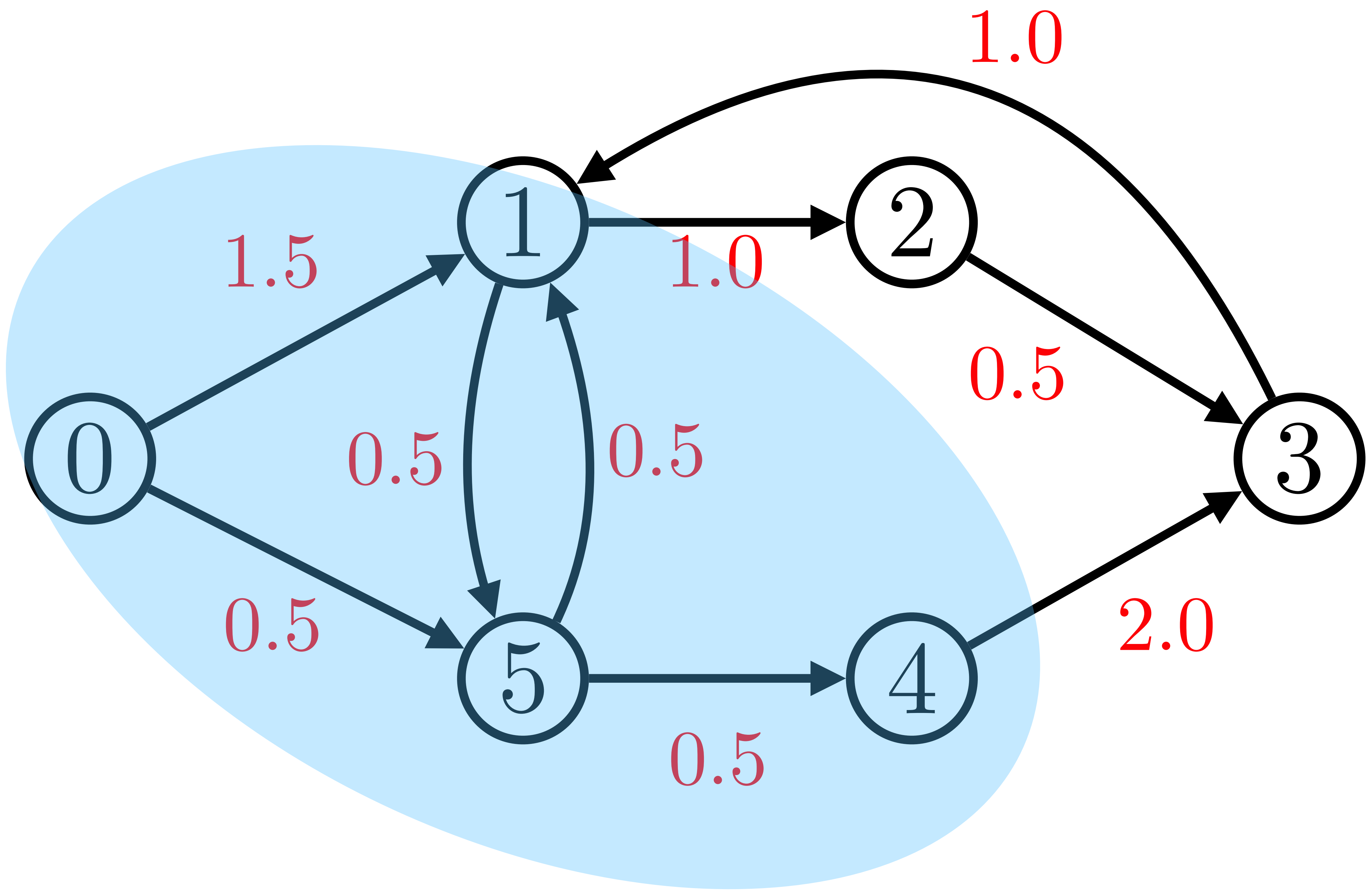
$$\{(0,1),1.5\}\ \{(1,2),2\}\ \{(4,3),3\}$$

`priority_queue`

The top element in the queue is

$$\{(0,1),1.5\}$$

We pop this element out of the queue.



0	1	2	3	4	5
0	1.0	∞	∞	1.0	0.5

`dist_to`

$\{(1,2),2\}$ $\{(4,3),3\}$

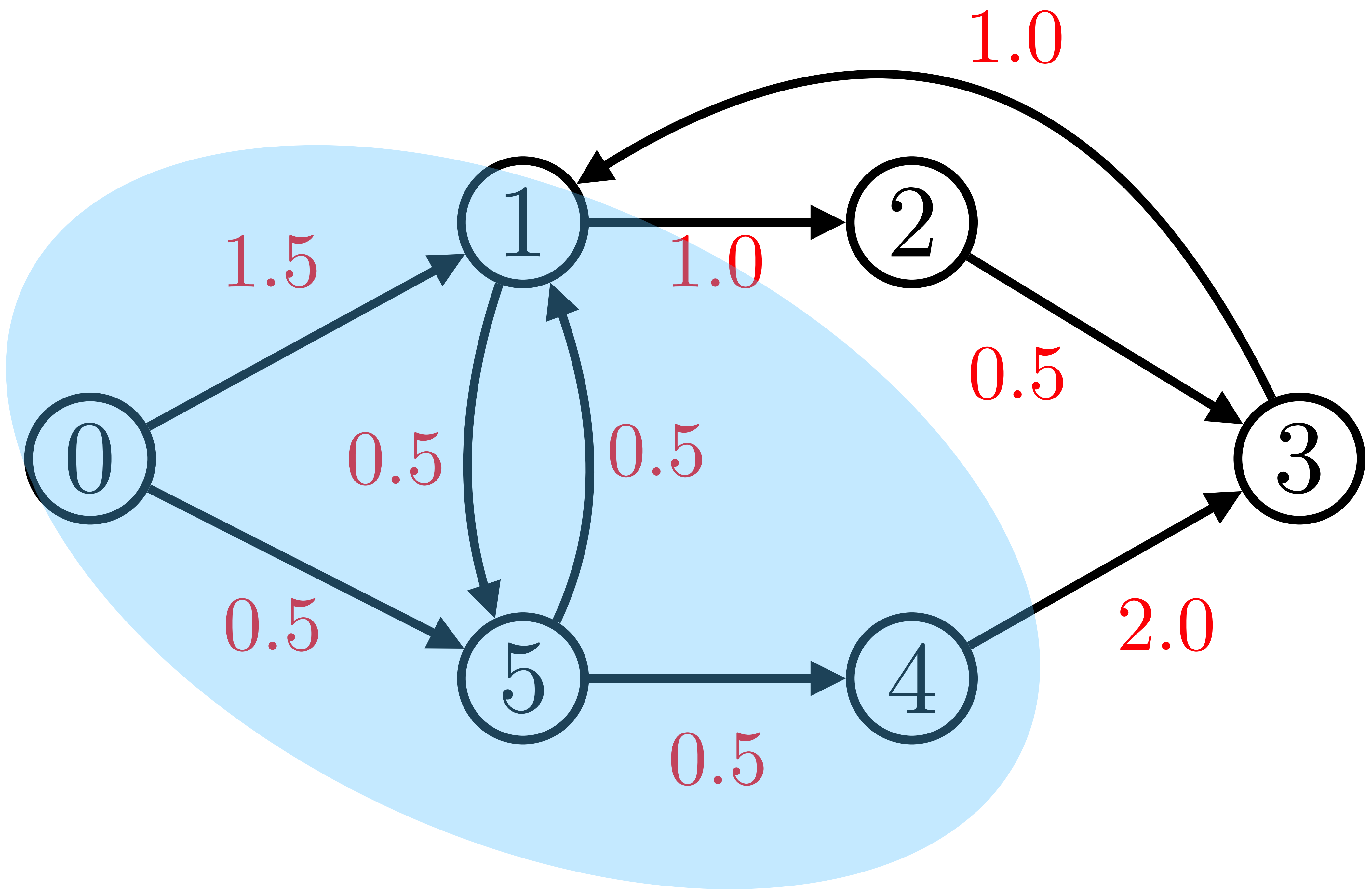
`priority_queue`

We popped $\{(0,1),1.5\}$ out of the queue.

Something different happens.

The destination vertex is 1, but 1 is already in our set S.

So we just ignore this edge.



0	1.0	∞	∞	1.0	0.5
0	1	2	3	4	5

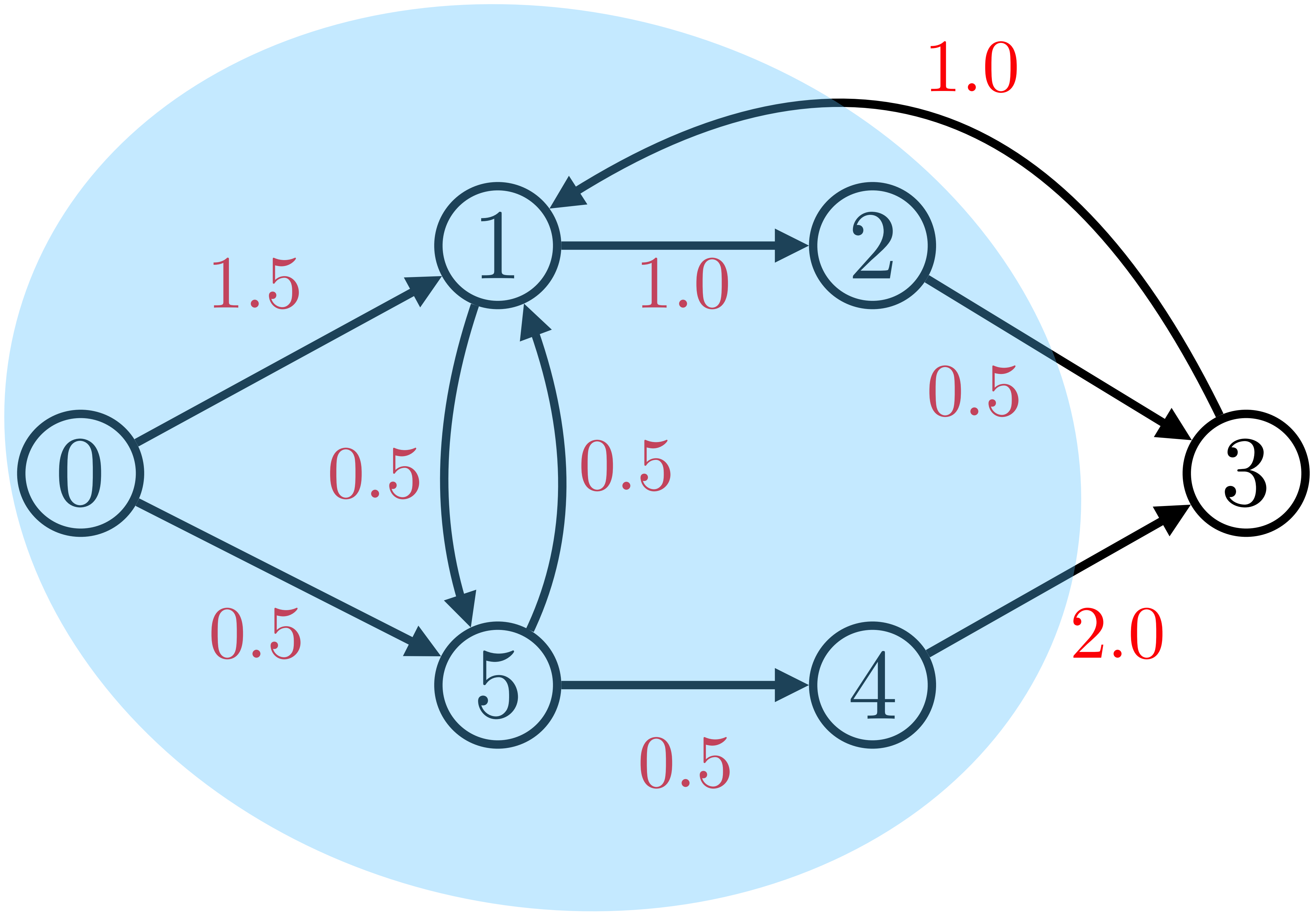
`dist_to`

$\{(1,2),2\}$ $\{(4,3),3\}$

`priority_queue`

Next we pop $\{(1,2),2\}$ out of the priority queue.

The destination vertex 2 is not in our set S so we process it.



0	1	2	3	4	5
0	1.0	2.0	∞	1.0	0.5

`dist_to`

$$\{(4,3),3\}$$

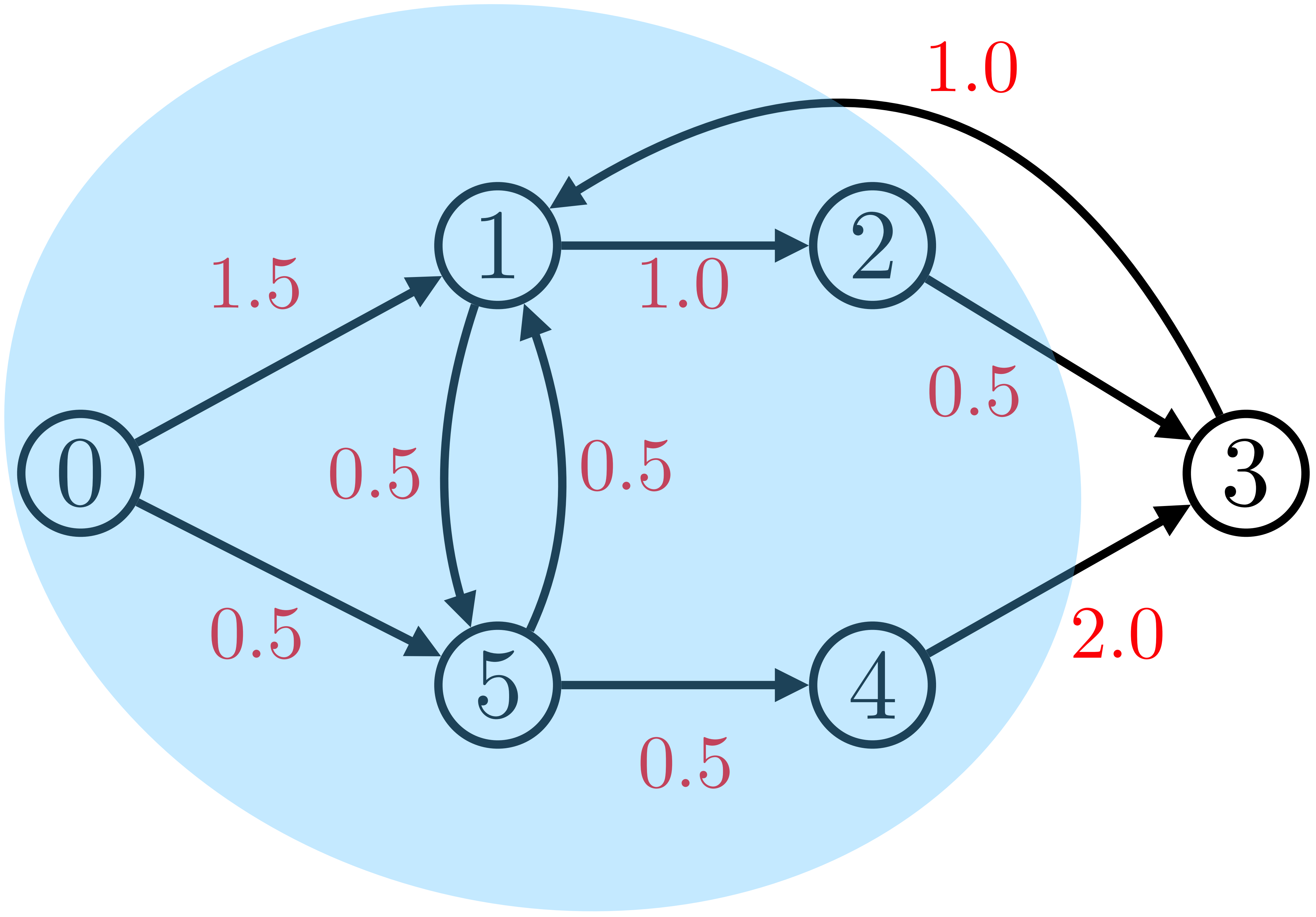
`priority_queue`

Next we pop $\{(1,2),2\}$ out of the queue.

The destination vertex 2 is not in our set S so we process it.

We set `dist_to`$[2] = 2.0$.

We add outgoing edges from 2 that leave S to the priority queue.



0	1.0	2.0	∞	1.0	0.5
0	1	2	3	4	5

`dist_to`

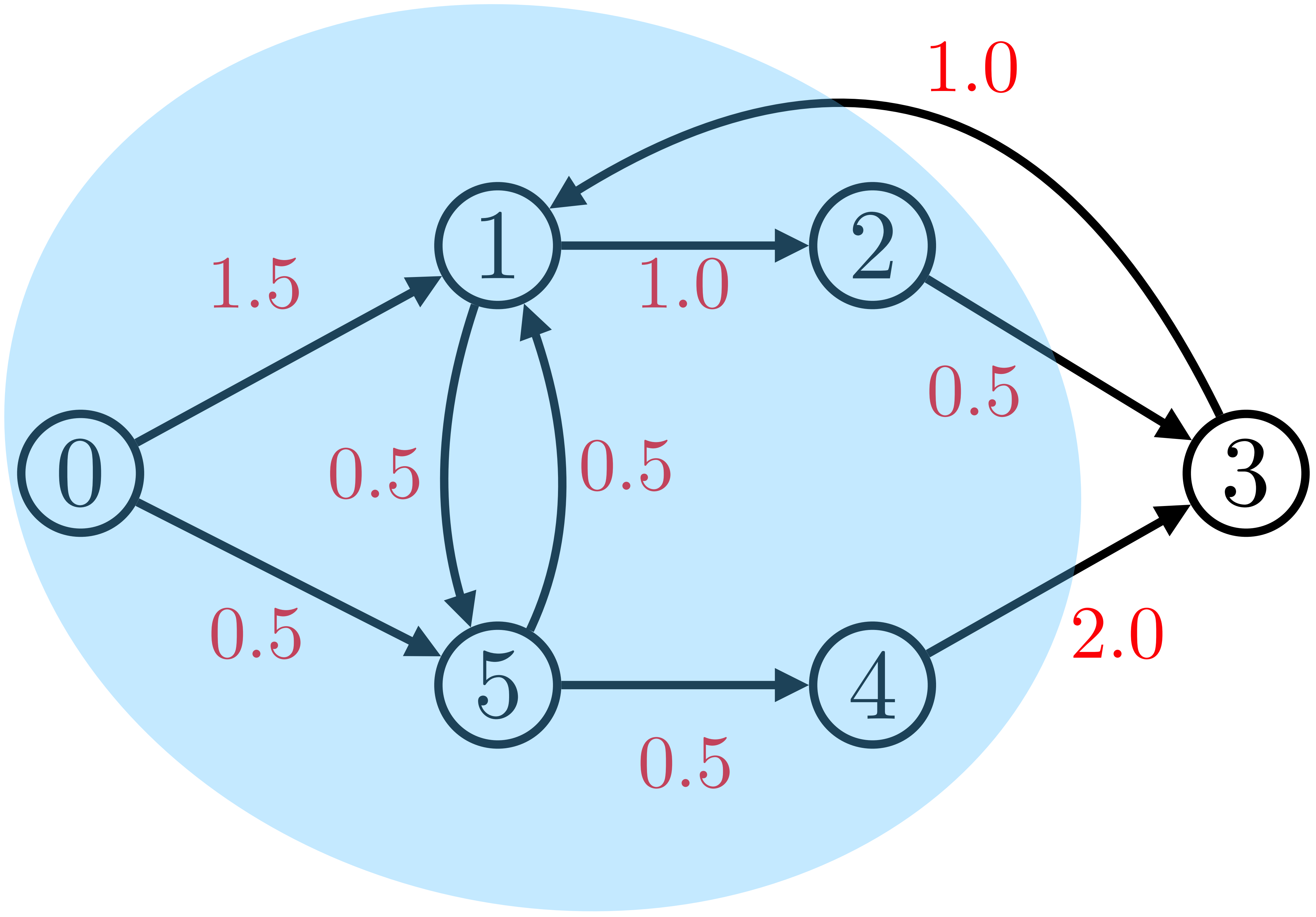
$\{(2,3),2.5\}$ $\{(4,3),3\}$

`priority_queue`

Add $\{(2,3),2.5\}$ to the queue.

The key value 2.5 is

$\texttt{dist\_to}[2] + (2,3).\texttt{weight}$



0	1	2	3	4	5
0	1.0	2.0	∞	1.0	0.5

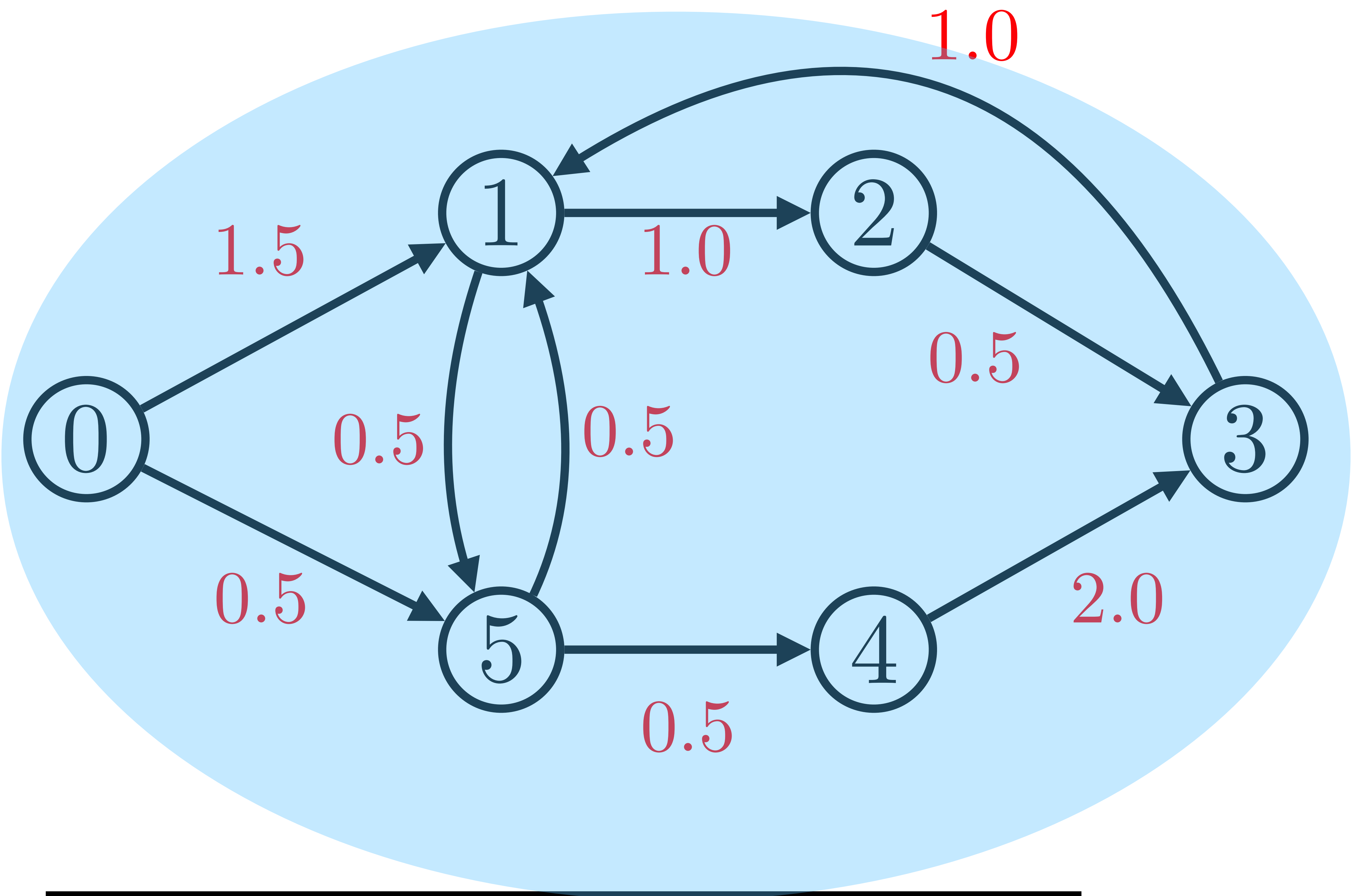
`dist_to`

$\{(2,3),2.5\}$ $\{(4,3),3\}$

`priority_queue`

We immediately pop $\{(2,3),2.5\}$ out of the queue.

The destination vertex 3 is not in S so we add it.



0	1	2	3	4	5
0	1.0	2.0	2.5	1.0	0.5

`dist_to`

$\{(4,3),3\}$

`priority_queue`

We immediately pop $\{(2,3),2.5\}$ out of the queue.

We update $\texttt{dist\_to}[3]$ with the key value

$$\texttt{dist\_to}[3] = 2.5$$

S now contains all the vertices.

We can terminate the algorithm.

Running Time

The running time of Dijkstra's algorithm is $O(|V| + |E| \log |E|)$.

The analysis is very similar to that of Prim's algorithm.

Each edge is pushed to the queue at most once.

The push and pop operations take time $O(\log |E|)$.

We spend $O(|V|)$ time for the initialization of `dist_to` and `edge_to`.

Notes

Our presentation differs from many others in two respects:

1) We assume positive edge weights rather than non-negative ones.

The same algorithm works with non-negative edge weights but then we cannot argue that every shortest path is relaxed.

2) We give a "lazy" version of Dijkstra's algorithm, just as we did with Prim's algorithm.

Typically instead Dijkstra's algorithm is described using a data structure we have not introduced, called an index priority queue.