**Graph Traversal Algorithms**

Table of Contents

[Graph Theory 4](#_Toc62999994)

[Directed and Undirected Graphs 4](#_Toc62999995)

[Adjacency Lists 6](#_Toc62999996)

[Adjacency Matrices 8](#_Toc62999997)

[Applications of Graph Theory 9](#_Toc62999998)

[Breadth First Search 10](#_Toc62999999)

[Time Complexity 14](#_Toc63000000)

[Depth First Search 15](#_Toc63000001)

[Traversal Algorithm 16](#_Toc63000002)

[Time Complexity 18](#_Toc63000003)

[Edge Classification 19](#_Toc63000004)

[Cycle Detection 20](#_Toc63000005)

[Topological Sort 21](#_Toc63000006)

[Time Complexity 23](#_Toc63000007)

[Proof 24](#_Toc63000008)

[FAQ About Graph Traversal 25](#_Toc63000009)

[Weighted Graphs 29](#_Toc63000010)

[Representation 29](#_Toc63000011)

[Notations 30](#_Toc63000012)

[Solution 32](#_Toc63000013)

[Mathematical Description 32](#_Toc63000014)

[Negative Edge Weights 37](#_Toc63000015)

[Theorems Related to Single-Source Shortest Path Algorithms 40](#_Toc63000016)

[Optimal Substructure 40](#_Toc63000017)

[Triangular Inequality 40](#_Toc63000018)

[Dijkstra’s Algorithm 41](#_Toc63000019)

[Proving Edge Relaxation is Safe 41](#_Toc63000020)

[Shortest Path in DAG 41](#_Toc63000021)

[Shortest Path in General Graphs 44](#_Toc63000022)

[Bellman-Ford Algorithm 50](#_Toc63000023)

[Algorithm Demonstration 50](#_Toc63000024)

[Time Complexity 54](#_Toc63000025)

[Theory Explanation 54](#_Toc63000026)

[Negative Weight Cycles 55](#_Toc63000027)

[Proof 57](#_Toc63000028)

## Graph Theory

Suppose we want to traverse a graph, i.e. we want to go from a source vertex to a target vertex via some path. For example, consider a Rubik’s cube, where the source is the unsolved state and the target is the solved state. The possible problems we might want to solve in this scenario include:

* The different moves we need to get to the solved state from any state
* All possible ways to get from one state to another
* All possible states and moves

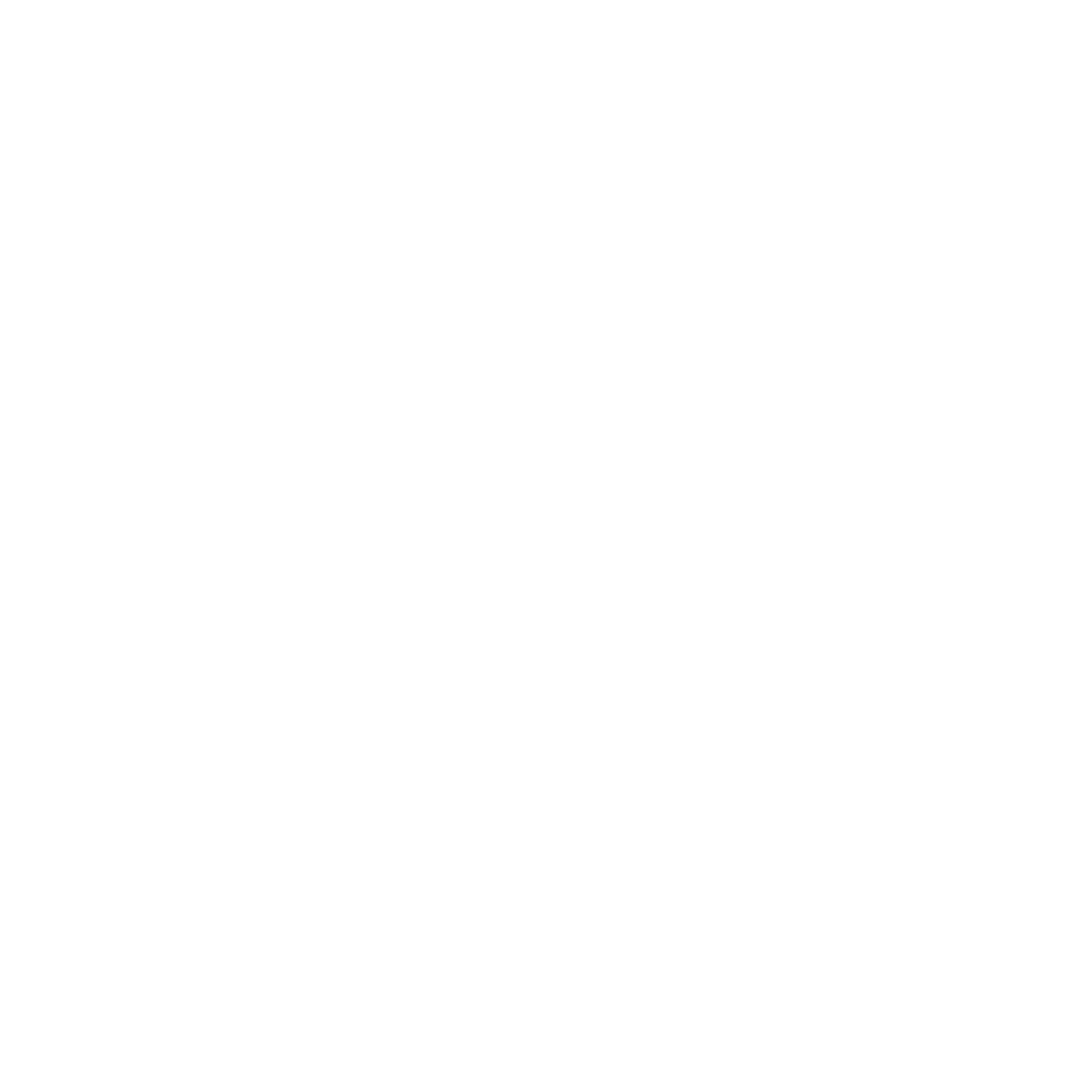
Before we can talk about how to go about doing this, we need to discuss how graphs work.

### Directed and Undirected Graphs

Any graph can be described as a set of vertices and edges, i.e. , where is the set of vertices or nodes, denoted by circles on the graph, and is the set of edges, denoted by lines or arrows on the graph.

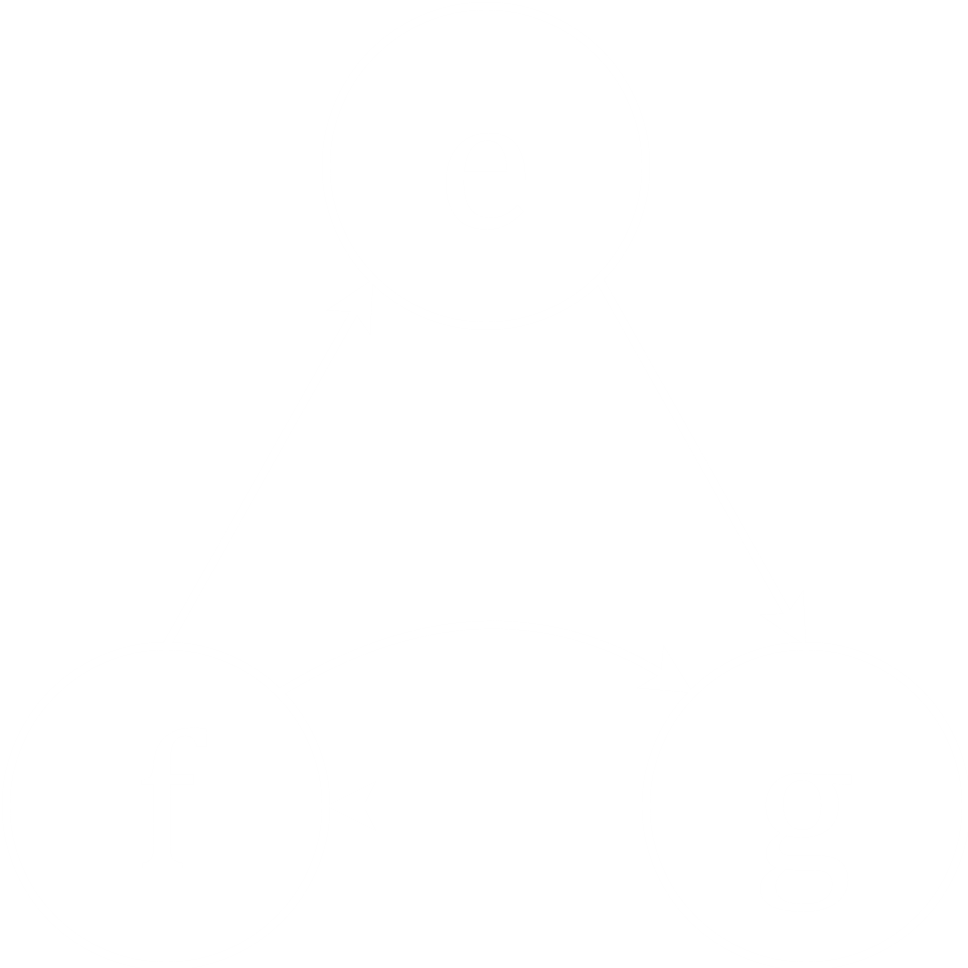
Graphs can be of two types, undirected and directed.

In an undirected graph, each edge, , has no direction. These use lines for edges, and the pair of vertices are unordered, meaning is the same as . Undirected edges are also called bi-directional edges, since we can travel along the edge in either direction.



For the undirected graph above, the set of vertices is and the set of edges is .

In a directed graph, each edge, , does have a direction. These use arrows for edges and the pair of vertices are in order, i.e. the edge goes from to .

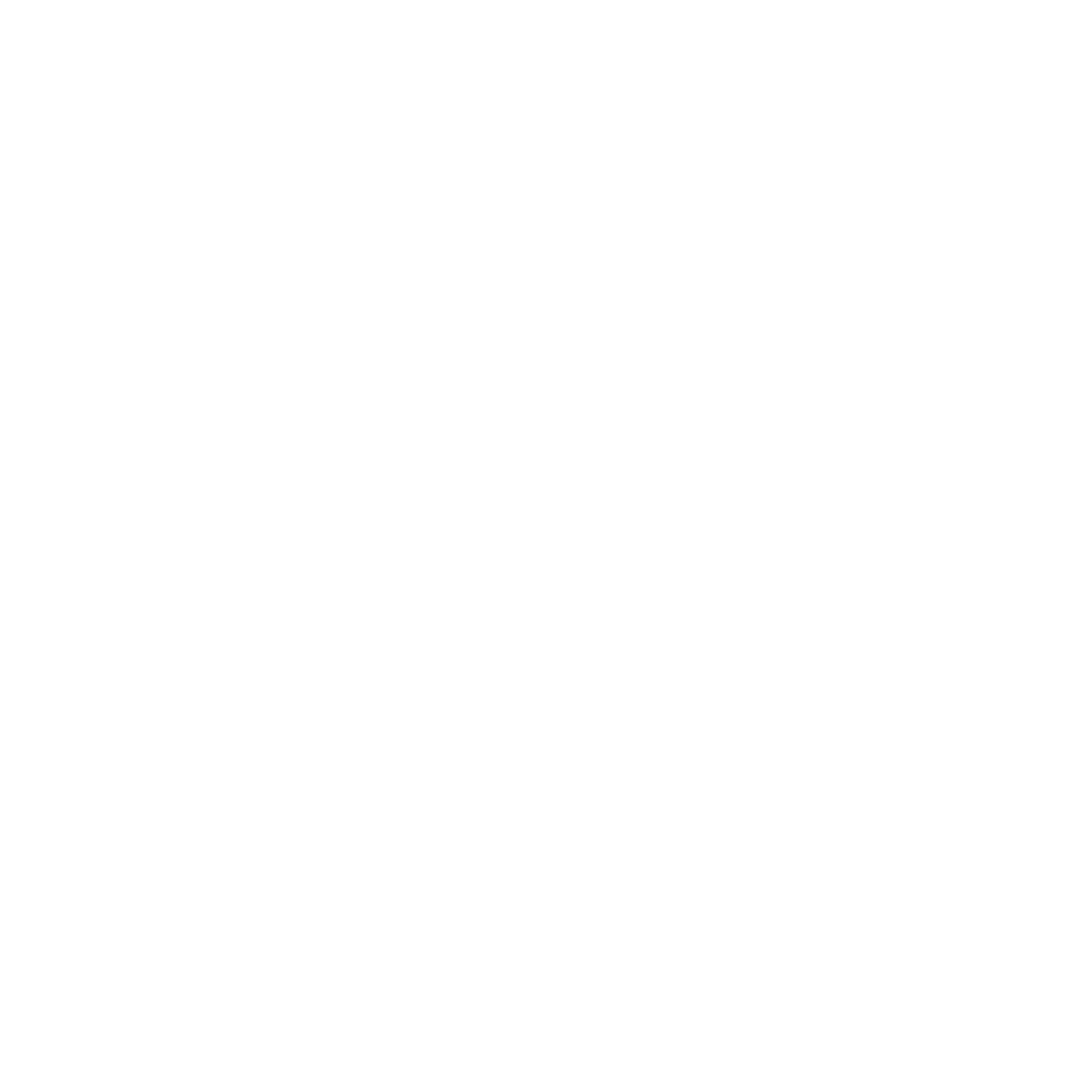
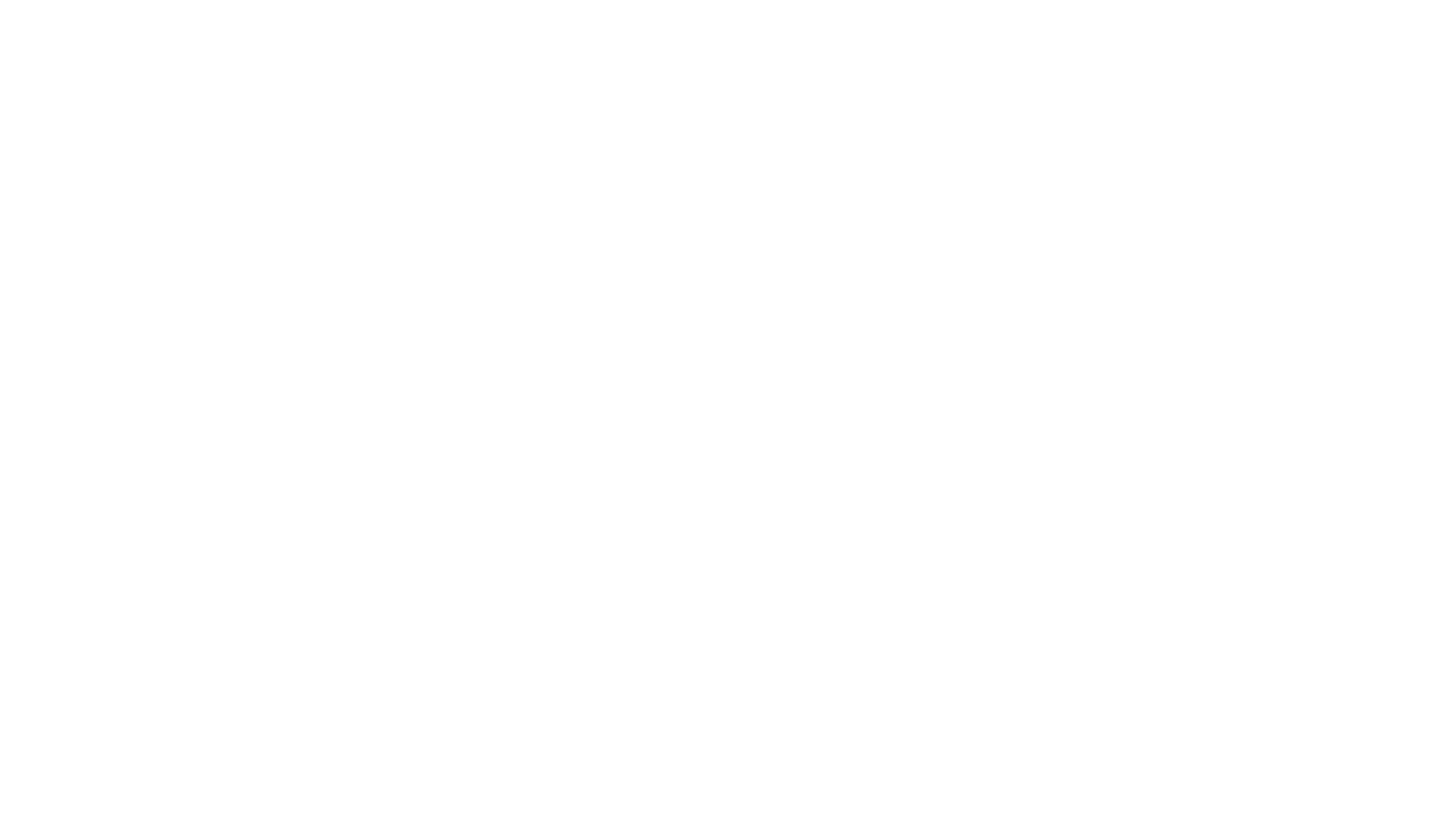


For the directed graph above, and . Notice how the order of the pairs of vertices for each edge is important here.

### Adjacency Lists

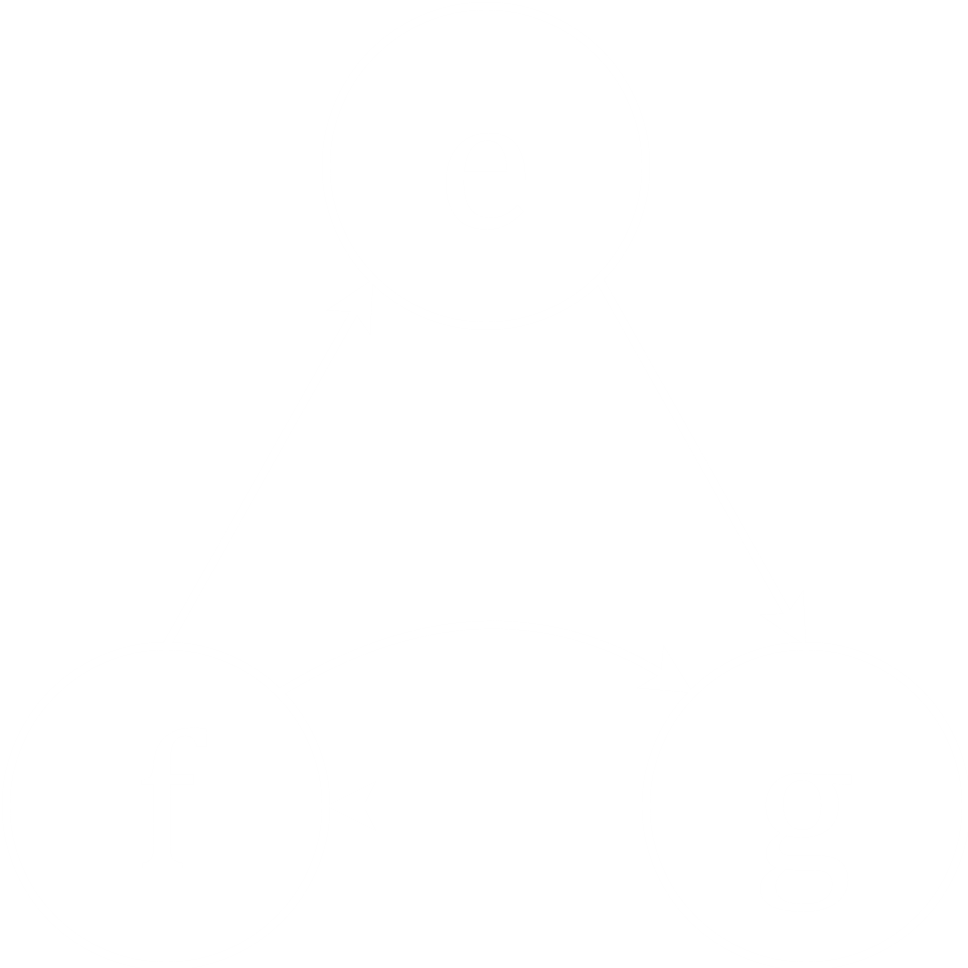
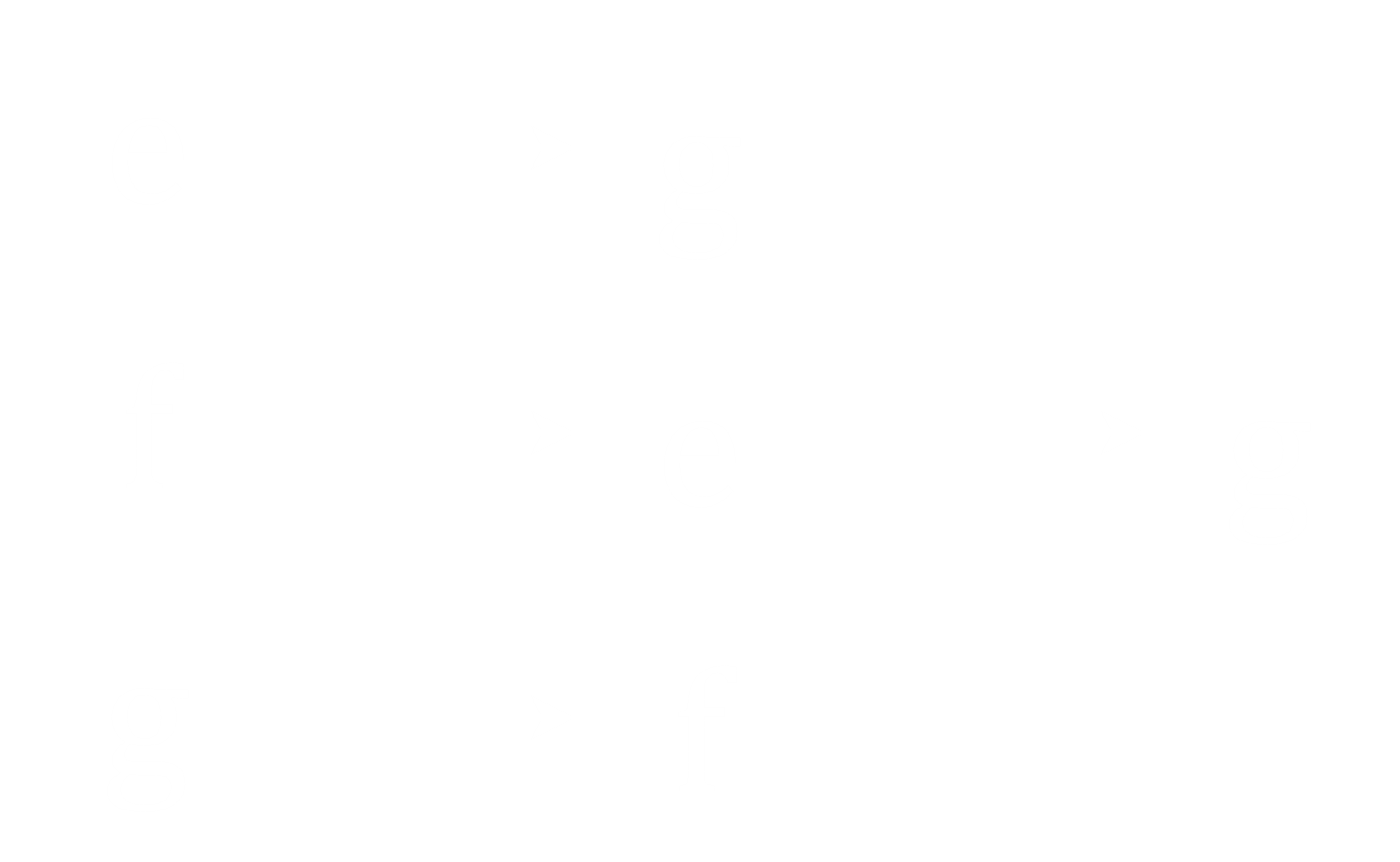
There are a few questions that we can answer using graphs. For example, in the undirected graph above, we can find out which vertices are neighbours of the vertex . It is clear from the graph that the vertices and are its neighbours, since these are the only vertices connected to it. We can find this systematically using the set of edges, searching for pairs that contain the vertex .

Finding the neighbours of a vertex in this manner is a cumbersome process. This is where adjacency lists can help us. An adjacency list is just an array, , with a size equal to the number of vertices in our graph, . Each element of the adjacency list contains a vector, which contains the neighbours of a particular vertex.

In the undirected graph above, for each vertex ,

For a directed graph, only outgoing connections are considered.

Here, for each vertex , .

Notice how, for an undirected graph, both vertices involved in an edge contain each other in the respective vectors. However, for a directed graph, only the source vertex contains the destination vertex for a given edge, not the other way around.

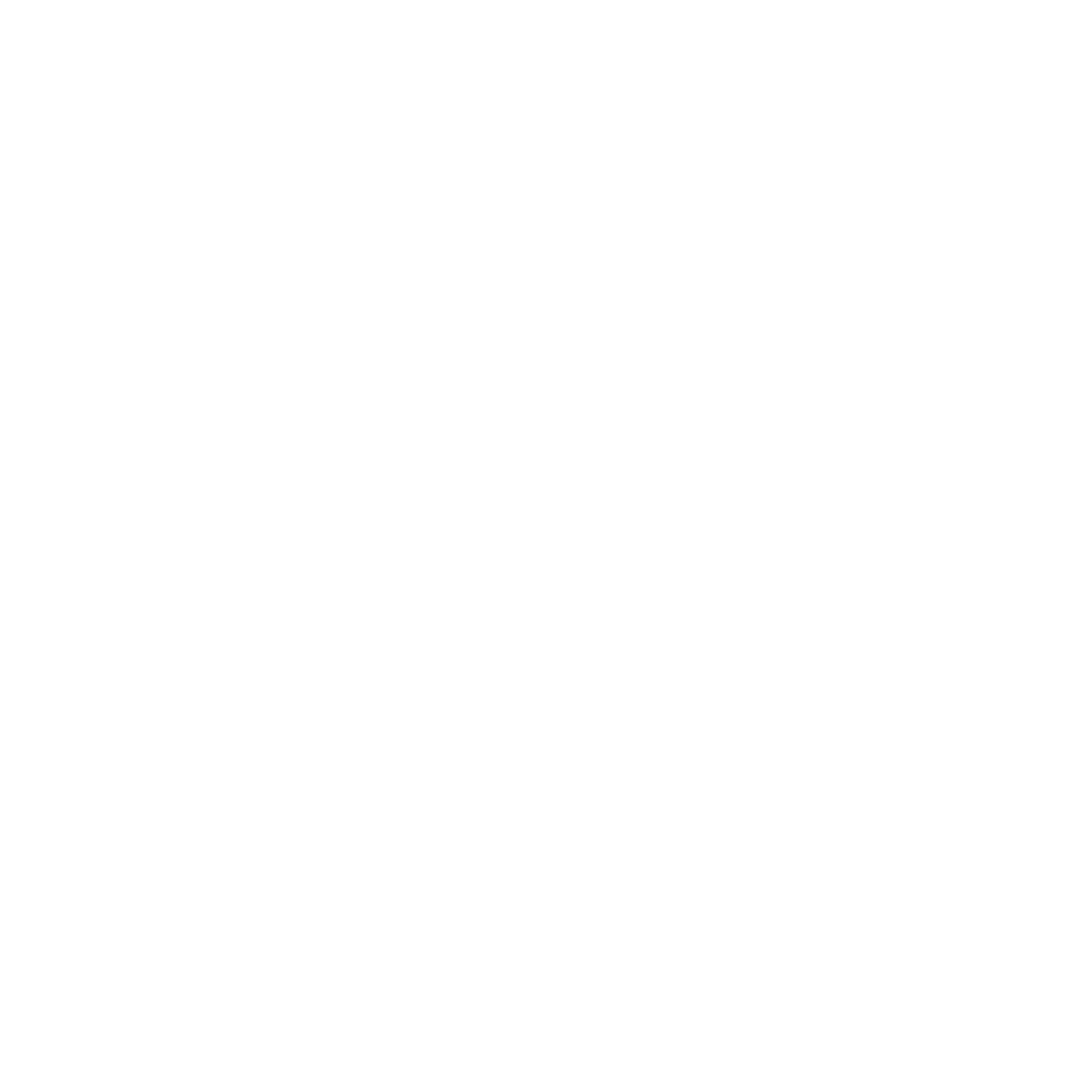
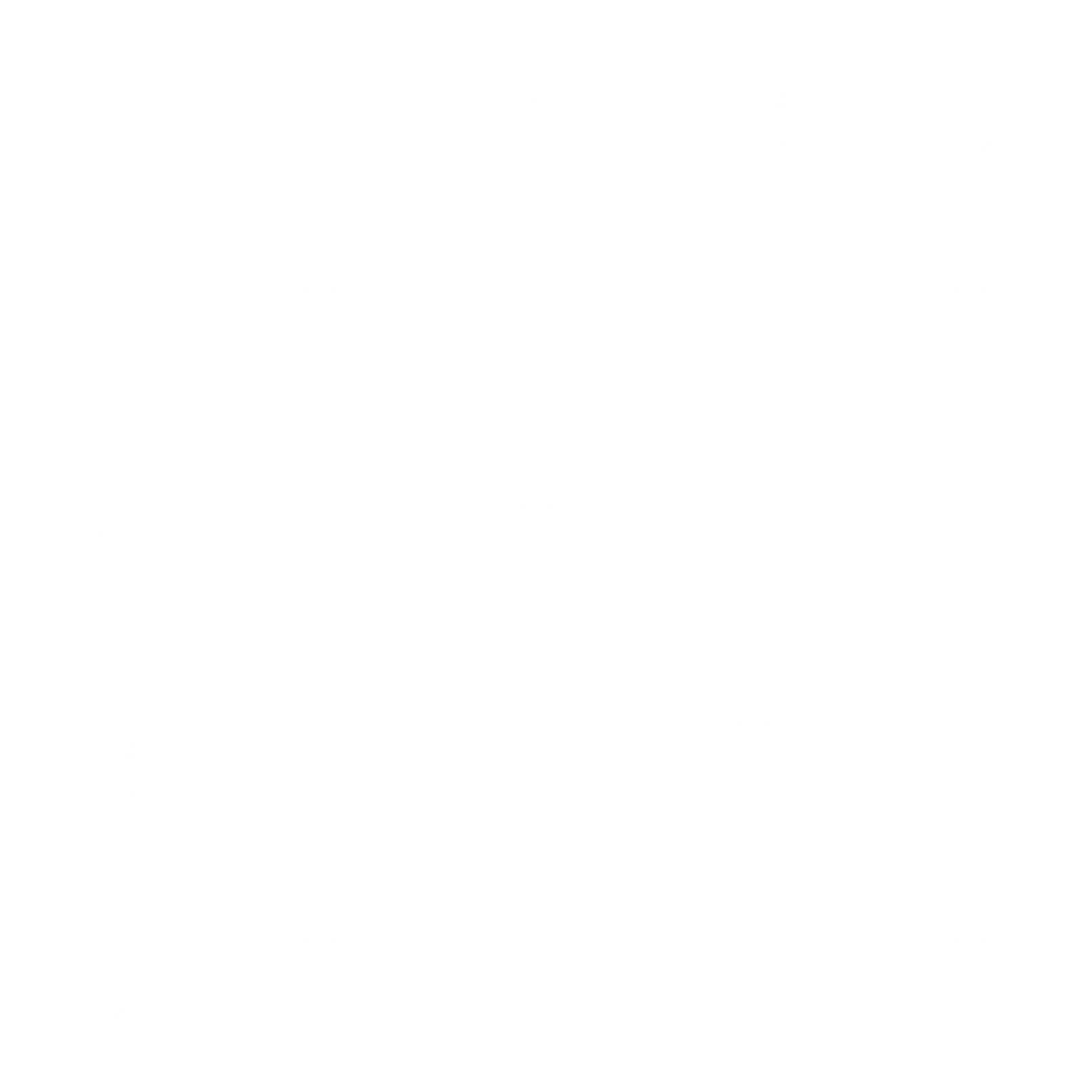
Note that in both cases, the order in which we place the vertices in the arrays or the vectors is irrelevant.

Now consider how large must be. First, we need to store all of the vertices, and we also need to store the connections for each vertex, i.e. the edges. Notice that, for the undirected graph, each edge appears twice. For the edge between the vertices and , the edge appears once from to and again from to . Thus, for an undirected graph, the size of is . For the directed graph, each edge only appears once. Thus, for a directed graph, the size of is .

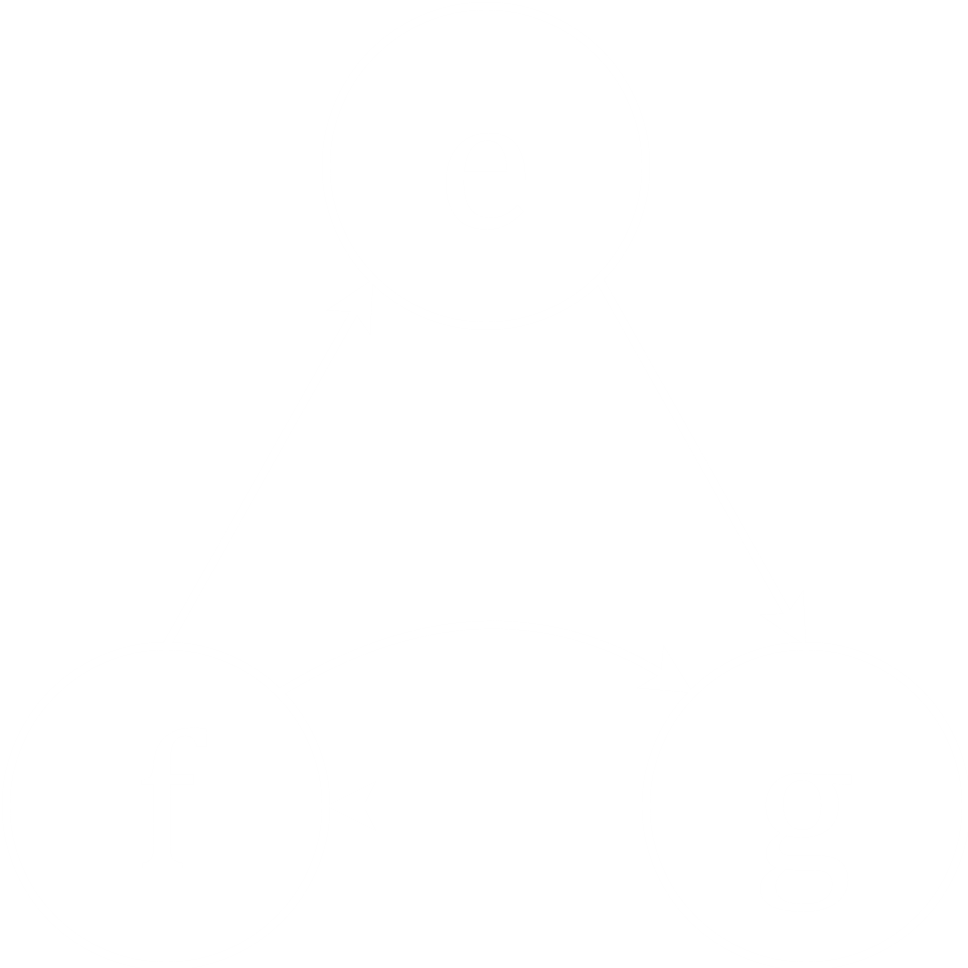
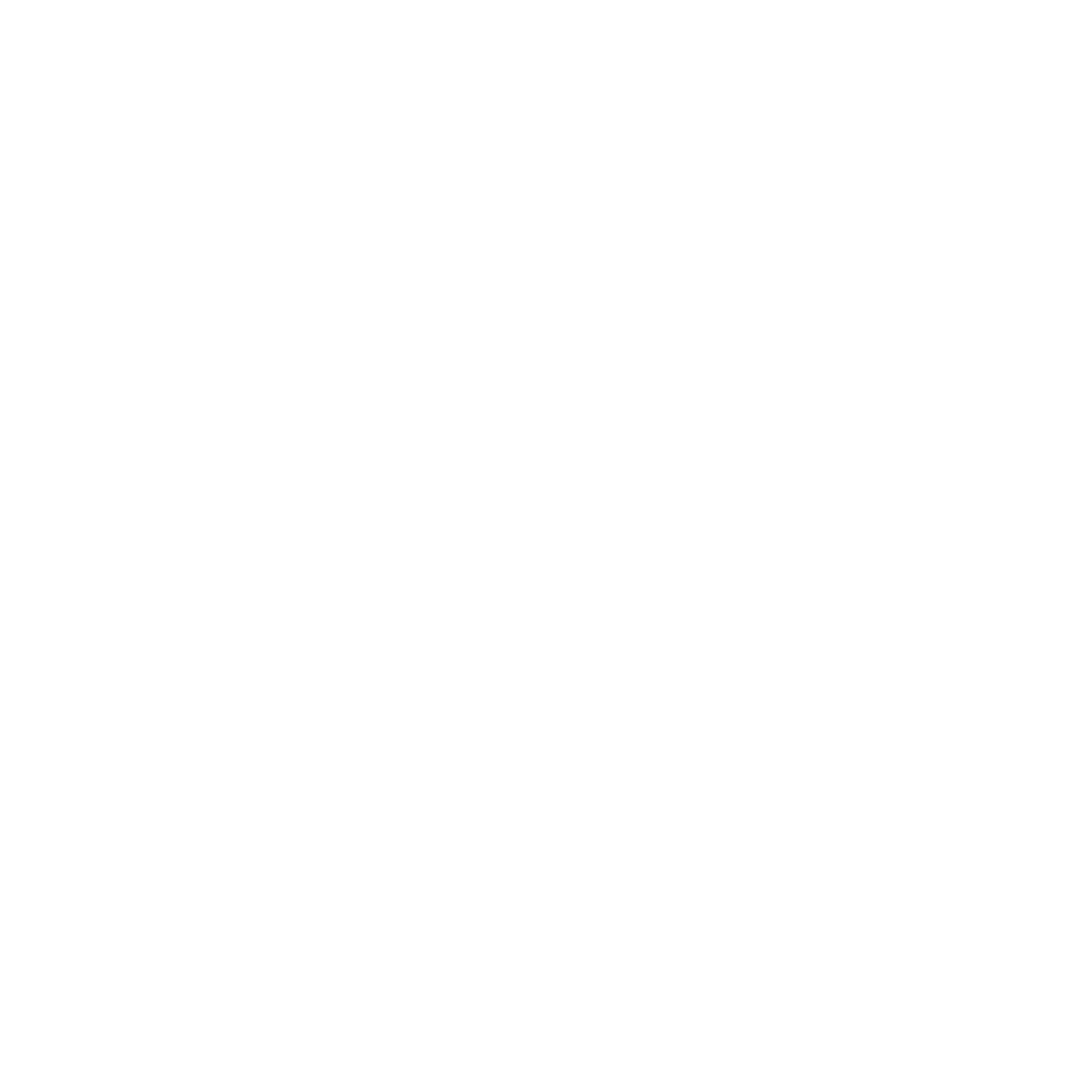
### Adjacency Matrices

An adjacency matrix represents a graph using a 2D array. Each element is either a or , with the indicating that there is no connection between the vertices and a indicating there is.

In an undirected graph, for the vertices , .

In a directed graph, for the vertices , , i.e. there is a in a cell only if there is a connection in the direction from to .

### Applications of Graph Theory

Graphs can be used for:

* Web Crawling
  + Indexing all of the pages in a website – Each page of the website is considered a vertex and if two pages are connected (e.g. via a hyperlink), it is considered an edge.
* Social Networks
  + Recommending friends to users – Each user is considered a vertex and if two users are ‘friends’, there is an edge between them. Recommendations are made between users that are connected to a common vertex.
* Network Broadcasting
  + Sending data packets – Each computer or router is considered a vertex and any devices that are connected via a network are connected along the edges.
* Garbage Collection
  + Freeing unused or unreachable memory blocks – Most modern programming languages (but not C or C++) have the ability to do this. Each memory location and variable is considered a vertex, and when a memory vertex is not reachable via a variable vertex using an edge, that memory location is considered unused and is freed up so it can be used again.
* Solving Puzzles and Games
  + Solving mazes
  + Simple state-based games – An example of this is the Rubik’s cube.

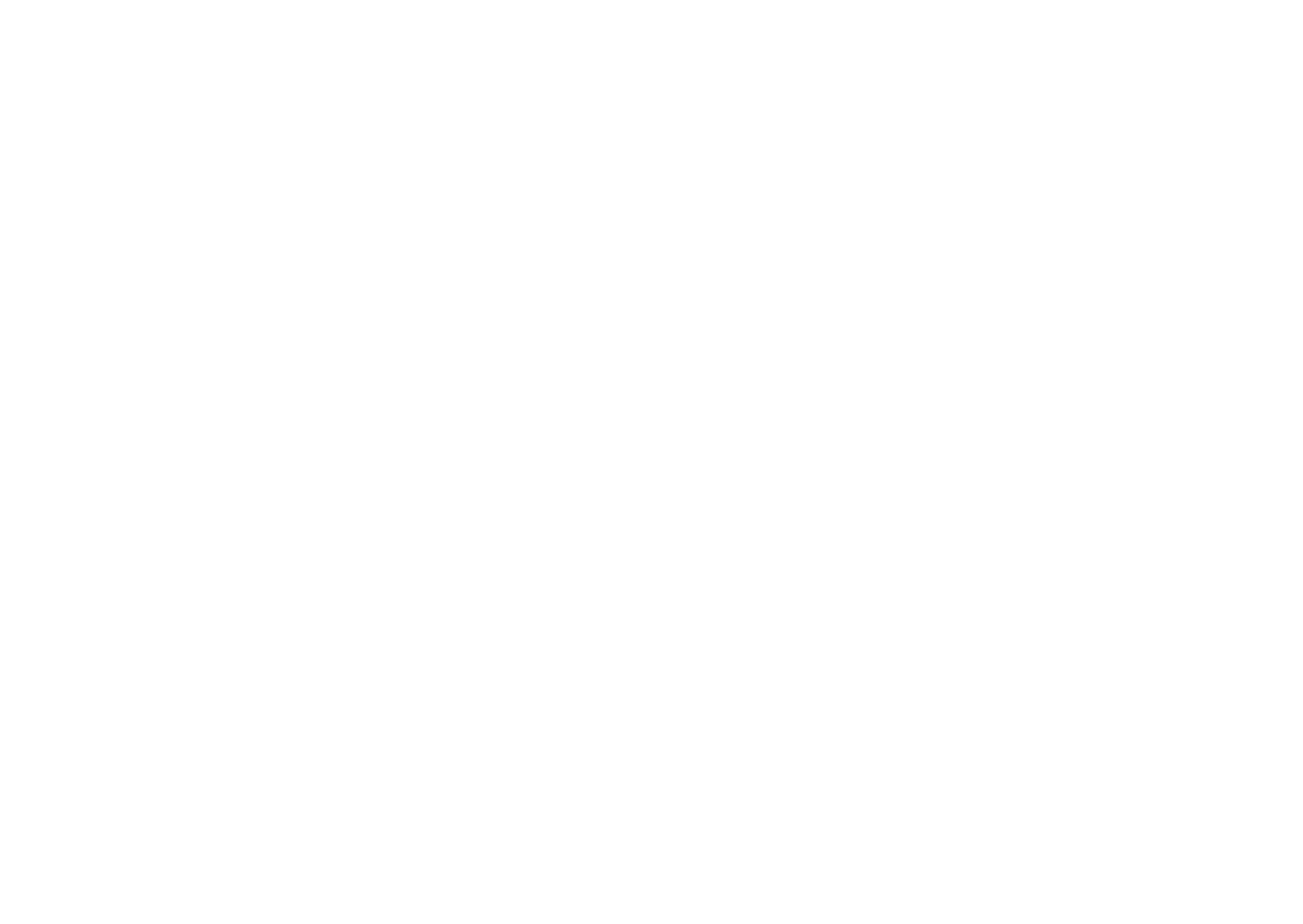
## Breadth First Search

Consider a Rubik’s cube, also called a pocket cube. If we want to solve the cube using graph traversal, then the first thing we need to do is create a configuration graph. To make the graph, we need to know the set of vertices and the set of edges.

The vertices are given by each possible state of the cube. Each smaller piece of a cube is called a cubie. Thus, for a cube, we have 8 cubies. The number of permutations for 8 cubies is Since each cubie has sides, there are possible ways of twisting the cube. Which side we are looking at is called a symmetry, and there are symmetries of the cube. These are the same. These are also faces hidden, making those twists non-valid. Thus, the total number of states of the cube is . (If you did not fully understand this, do not worry. It is not all that important to understand this part.)

The edges represent each possible move from one state to another. The edges are undirected, since we can go from one state to another and back again.

We will not be looking at every single vertex, but will consider what the graph looks like as a whole. If we work backwards, starting from the solved state, we can perform one rotation to get to each of the states that would take one rotation to solve the cube. From each of those, we can perform another rotation to get all the states that would let us solve the cube within two moves. This continues in the same manner.



Note that this graph is just a representation. We do not actually have just 3 or 5 moves in one stage, but many more.

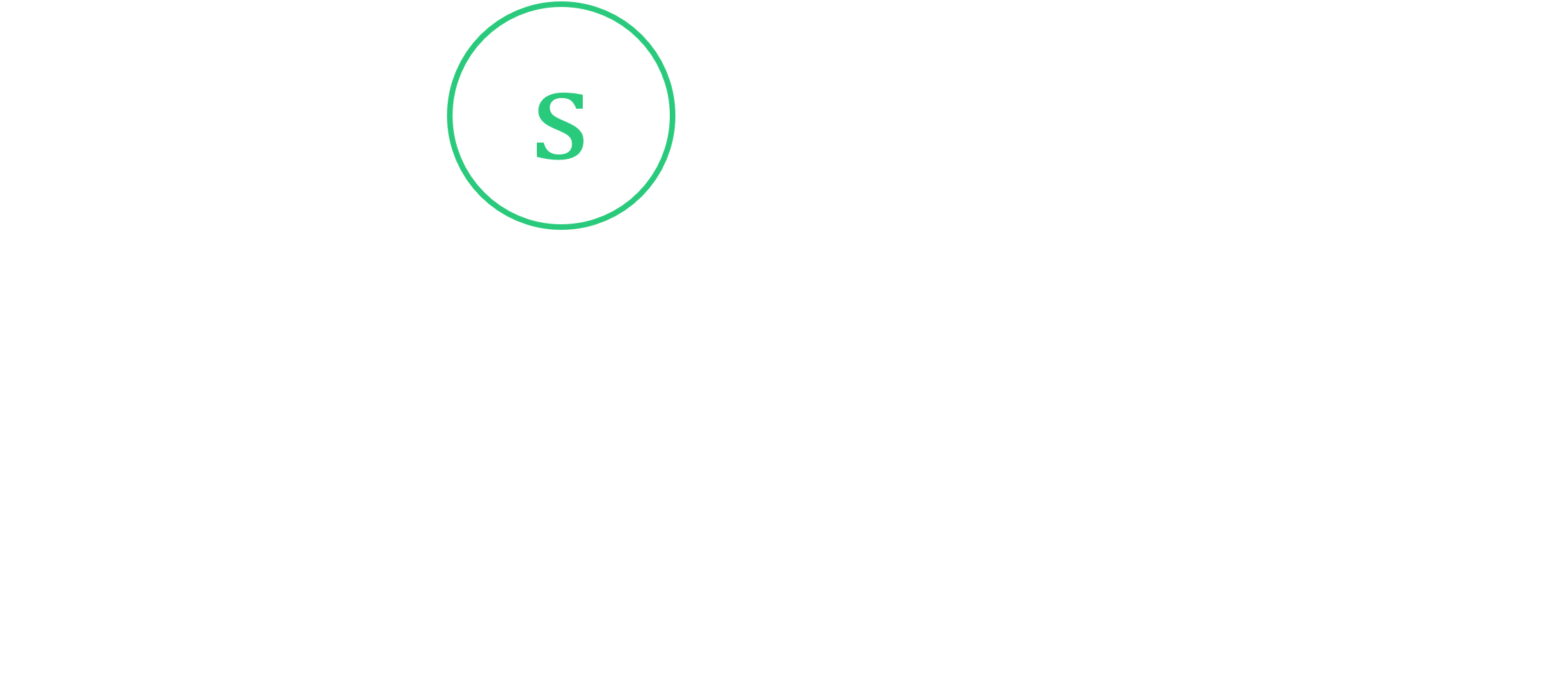
Notice how there are cases where one state can be reached from multiple states. This means that we need to ensure that we do not repeat any states, meaning we only have one state one time in the graph. If we continue in this manner, the maximum distance we can establish between the solved state and any other state, called the diameter of the graph, is , meaning any configuration of the cube will take at most 11 moves to solve. This number is called God’s number.

The steps for the solution are:

* Start from the solved state
* Traverse one layer at a time
* Keep track of previous states
* Avoid repeating states
* Keep going until all possible states have been traversed

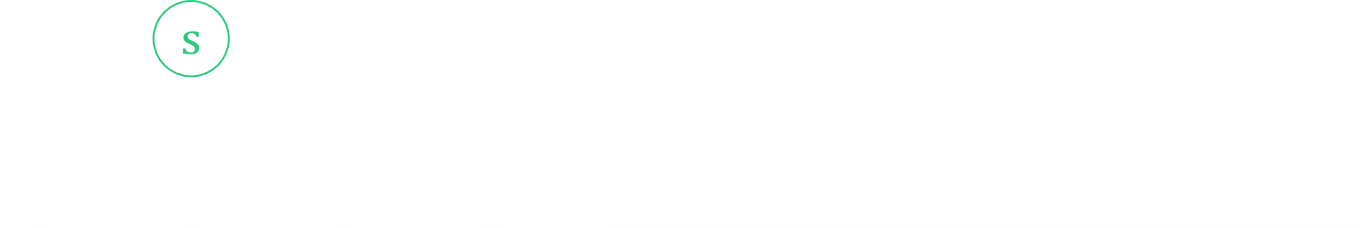
The process we just went through is essentially Breadth First Search (BFS). We started from the source state and traversed all the states one layer at a time. From the source state, we traversed all of its neighbours, then all of their neighbours and so on.

Consider the graph below.

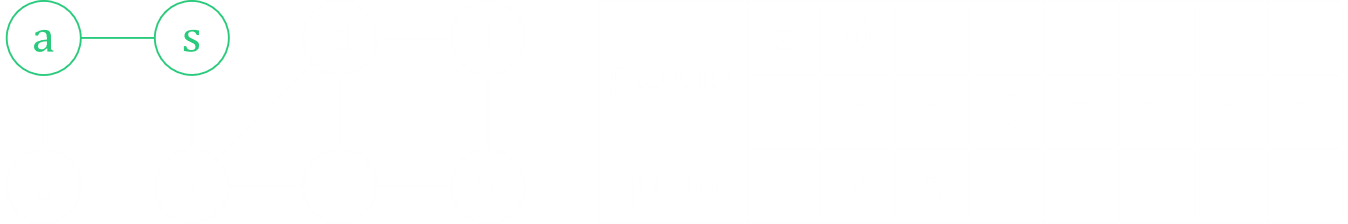


For this graph, is the source vertex and the adjacency list, is in alphabetical order, so if we have multiple neighbours, we will visit them in alphabetical order. In order to traverse the graph, we need two things. We need an array to keep track of the parents of each vertex, and we need a queue to add vertices to as we find them so they can be traversed.

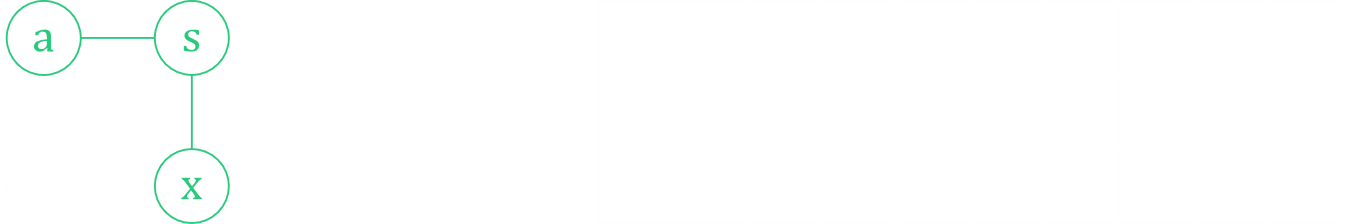
Let us begin at the source node, .



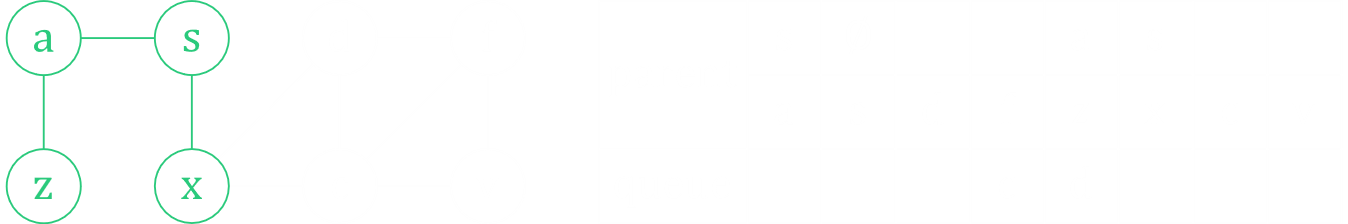
Next, we visit .

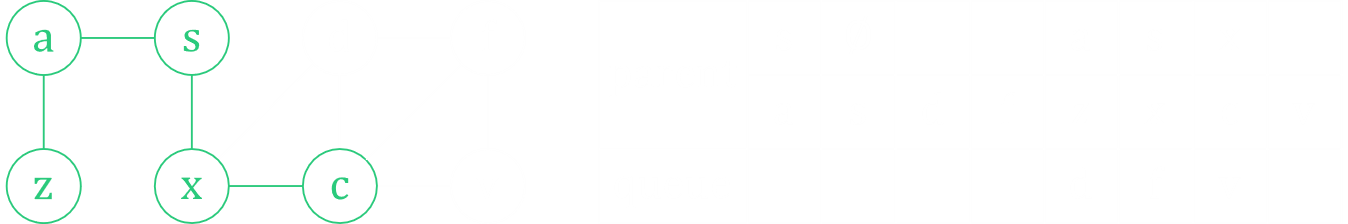


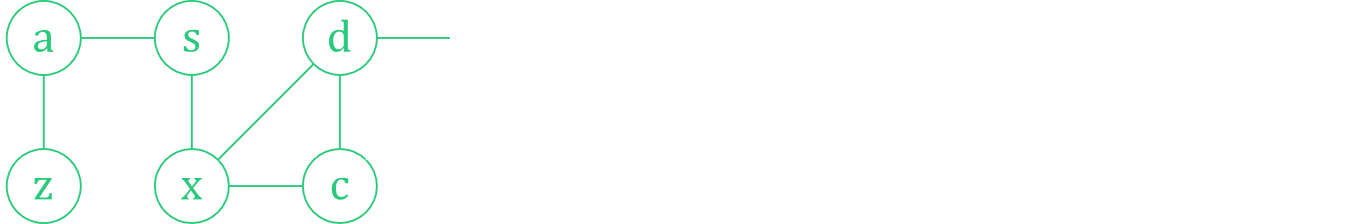
Then we visit .

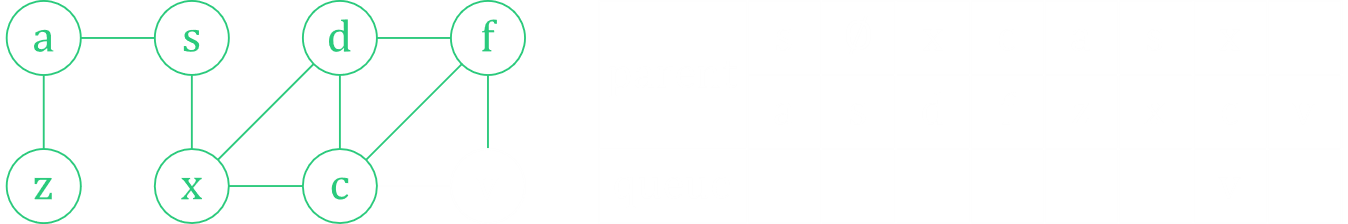


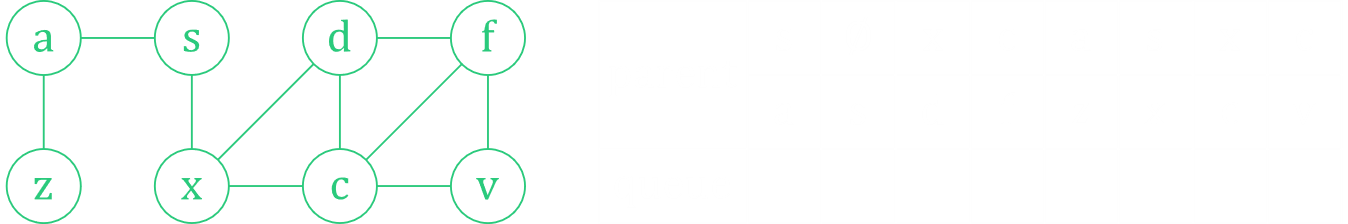
We continue in a similar manner.











Notice how we do not visit nodes that we have already visited again. For example, is a neighbour for , yet we did not visit from since had already been visited. We can use an extra array of Boolean values to keep track of which nodes have already been visited.

Also notice that the order would have changed had we not visited nodes alphabetically. For example, if we visited before , then would have been the parent for both and .

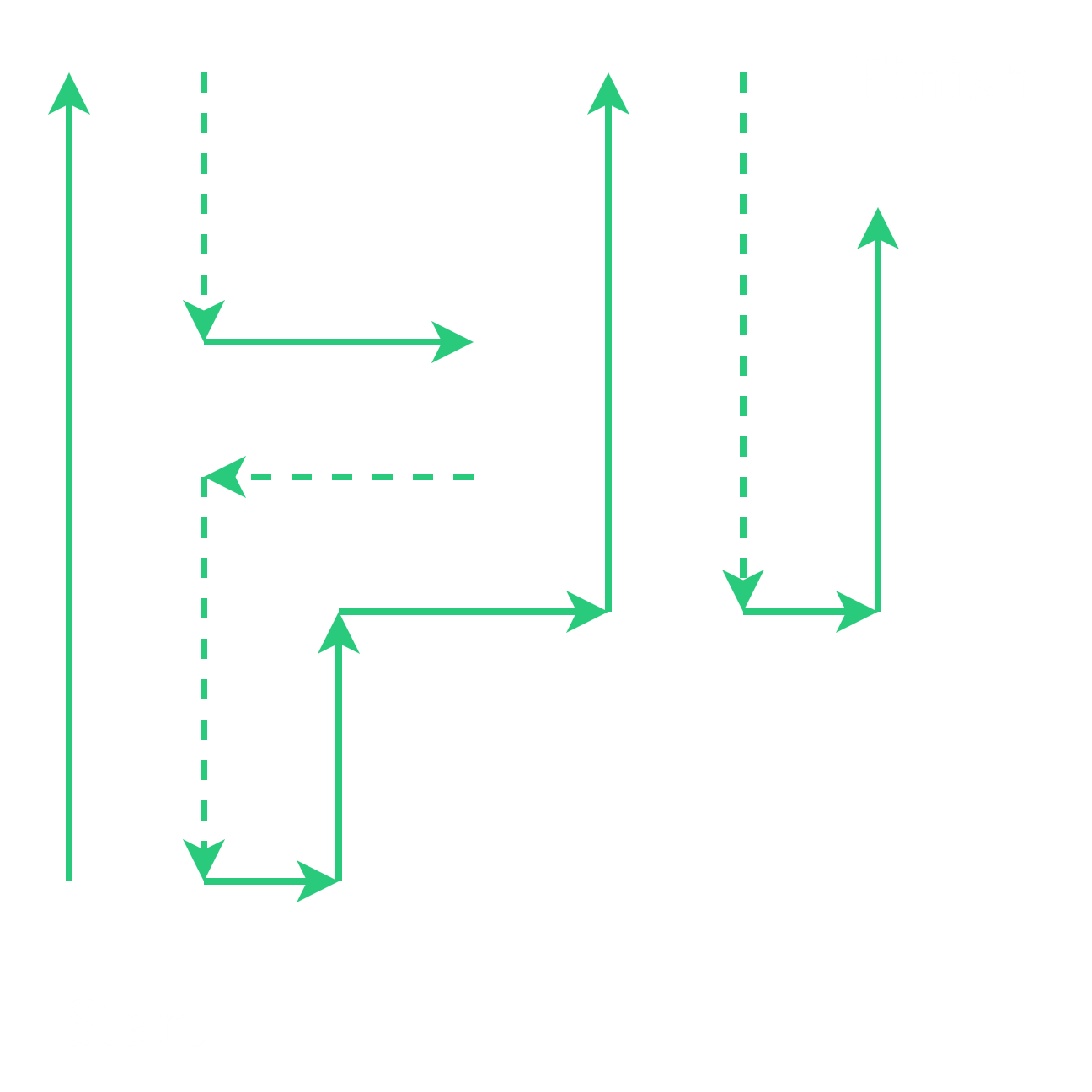
Finally, note that if we go to any node at all and then follow its parents, we will eventually reach . Thus, the path we travel from that node to is one of the shortest possible paths. For example, from to , we could have travelled , but this would not be the shortest possible path. If instead we used the parent nodes as stated, the path would be . There may, of course, be other paths that are just as short. This can be useful in the example we used for the Rubik’s cube, since it would give us the fastest possible way to solve the Rubik’s cube. We could also keep track of the length of the shortest path by keeping track of the layers we traverse while performing the original traversal.

### Time Complexity

In the BFS algorithm, we are visiting every vertex and visiting all the neighbours of every vertex and the maximum number of edges a vertex can have is . We know that for an undirected graph we check each edge twice (once from each vertex) giving edges, and for a directed one there are edges. These are the number of neighbours. Thus, the time complexity is .

## Depth First Search

Depth first search (DFS) is used to ‘explore’ graphs. Say we have a maze, like the one below.

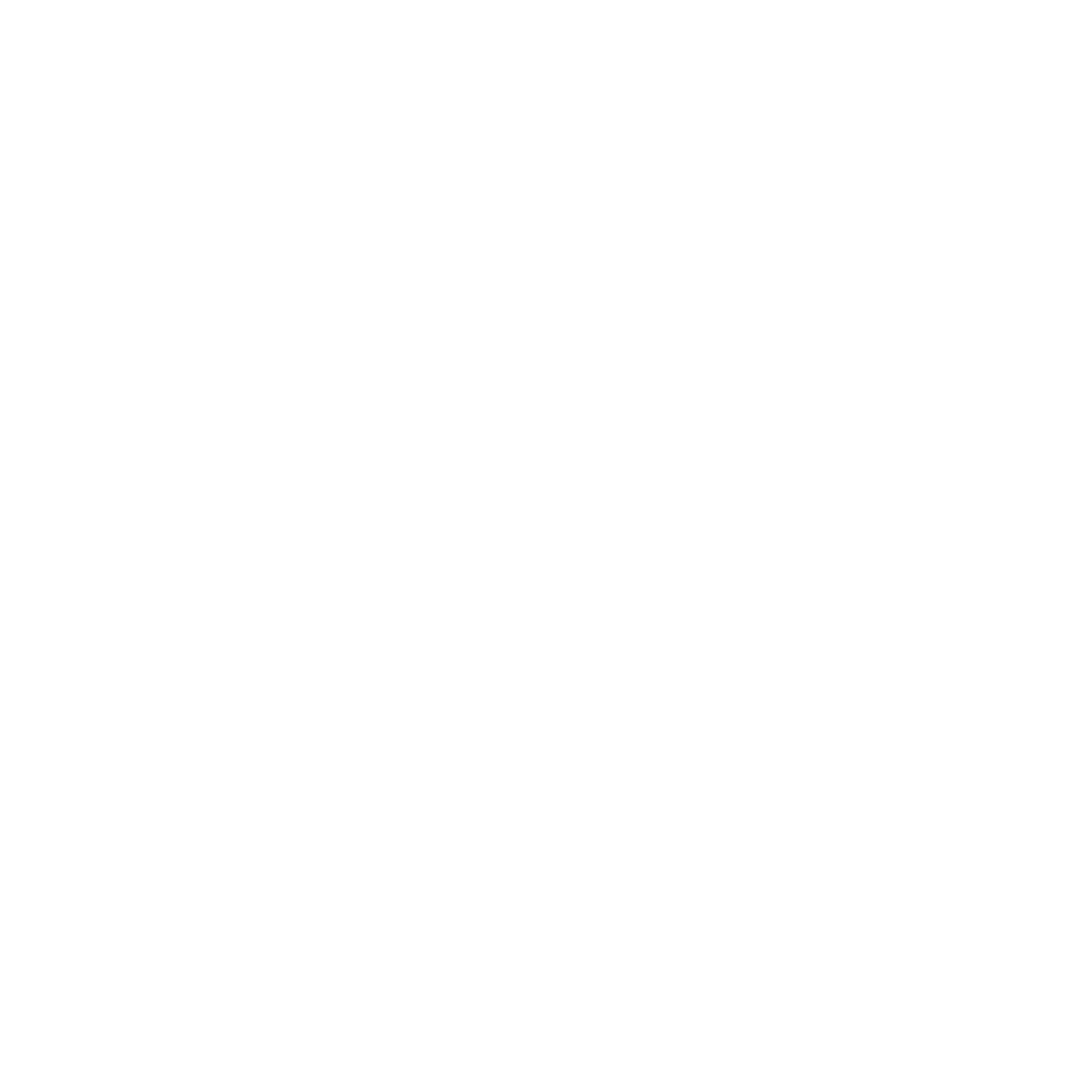


Here, our challenge might be to:

* Find a way out
* Explore all possible paths

In order to do this, we can use breadcrumbs. Essentially, breadcrumbs indicate that a path has been travelled along already. A rule we will be following while doing this is whenever we find ourselves having to make a choice between a left and a right path, we will choose the left one. The breadcrumbs are left at the junction points, so that we are able to identify that there is an unexplored option at that point. Also, we will not traverse a path we have already visited.

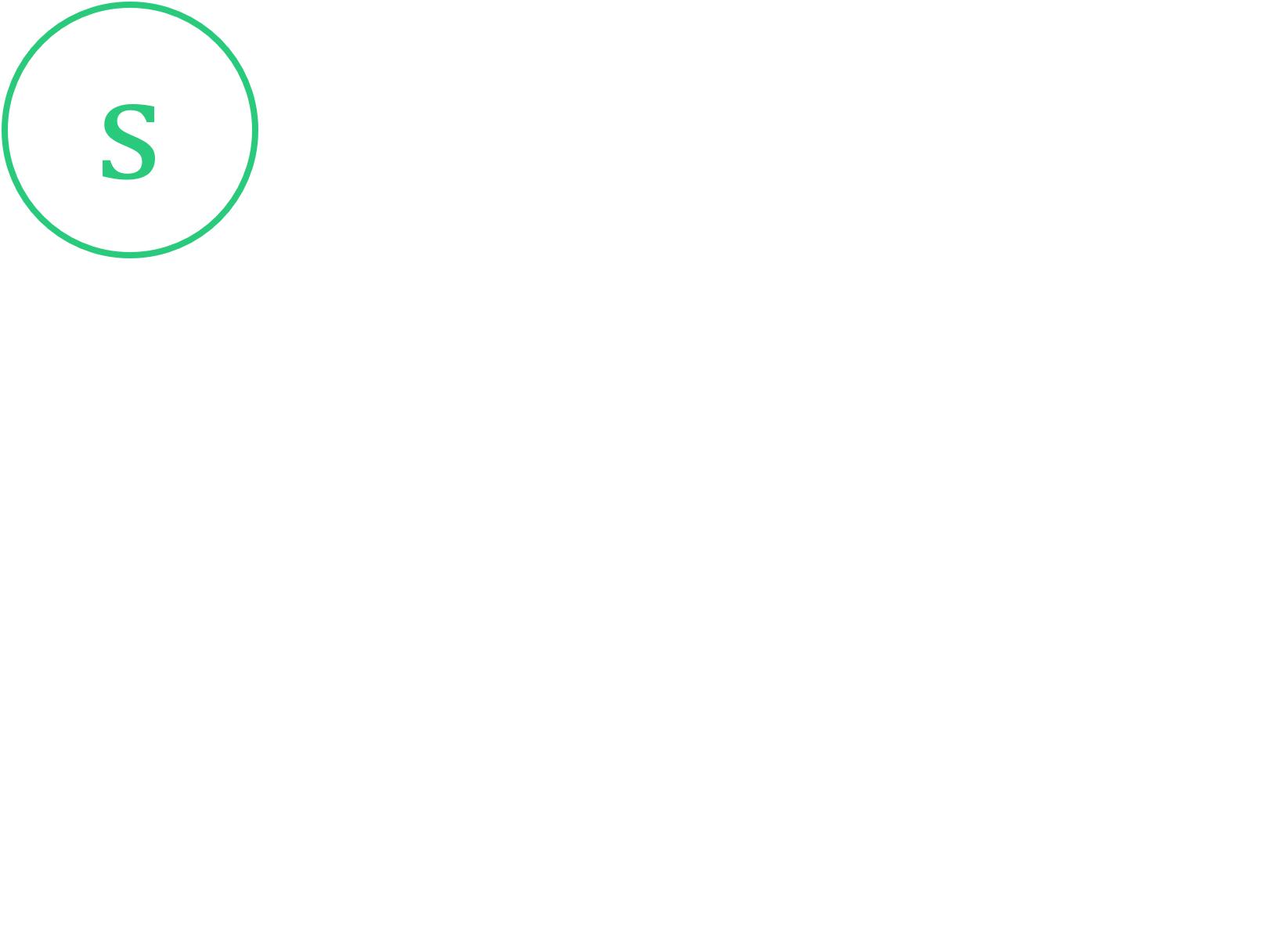
In this system, each junction point is considered a node and edges are created between junctions that are connected.



In programming, the ‘breadcrumbs’, are left using either recursion or a stack. When we reach a dead end, we will backtrack along the breadcrumbs until we find unexplored neighbours. We also need to ensure we do not repeat any of the junctions and we do this using an array. There is another functionality that is optional, which is to check and explore any junctions that are unexplored. This will be explained in a little while.

### Traversal Algorithm

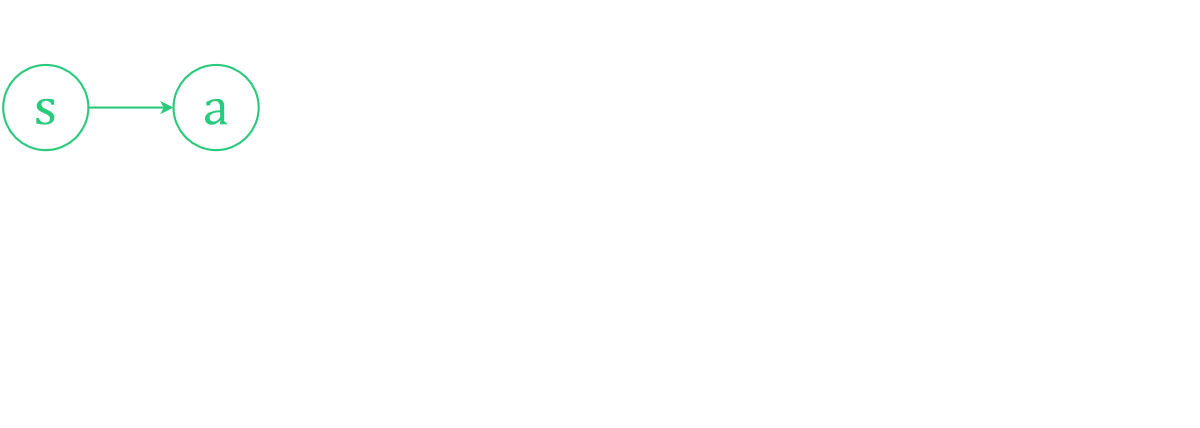
Assume that the adjacency list, , is sorted in alphabetical order.

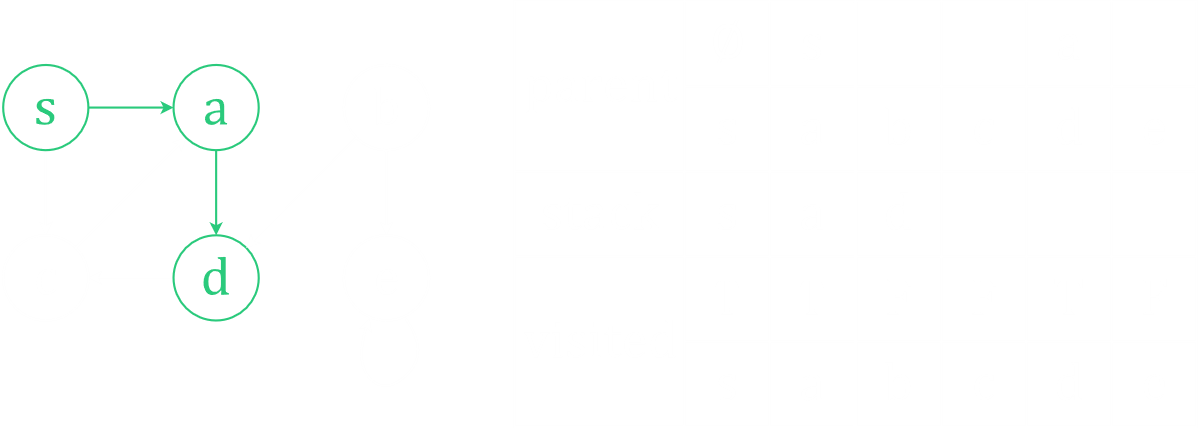


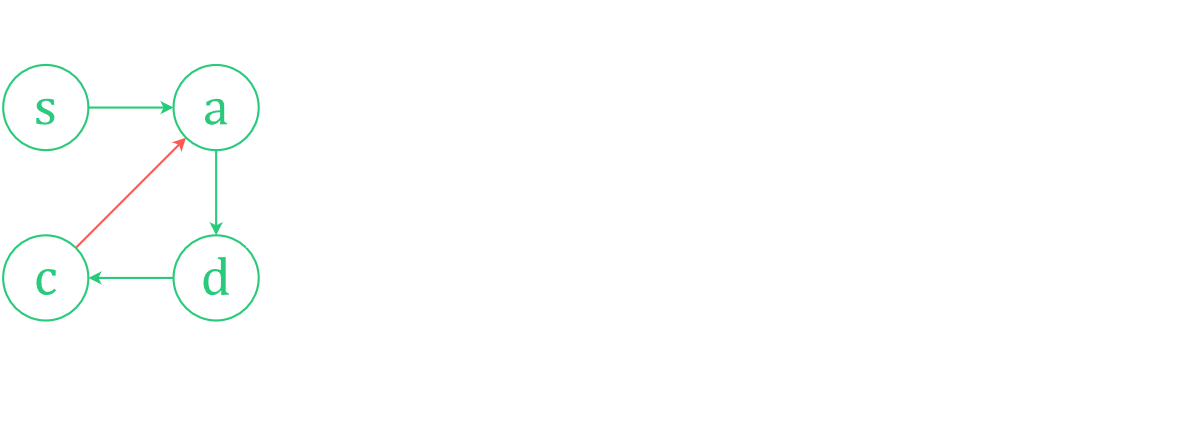
Here, we will start from the source vertex, , and our goal is to visit all of the vertices.

At every junction, the junction itself is pushed onto a stack.



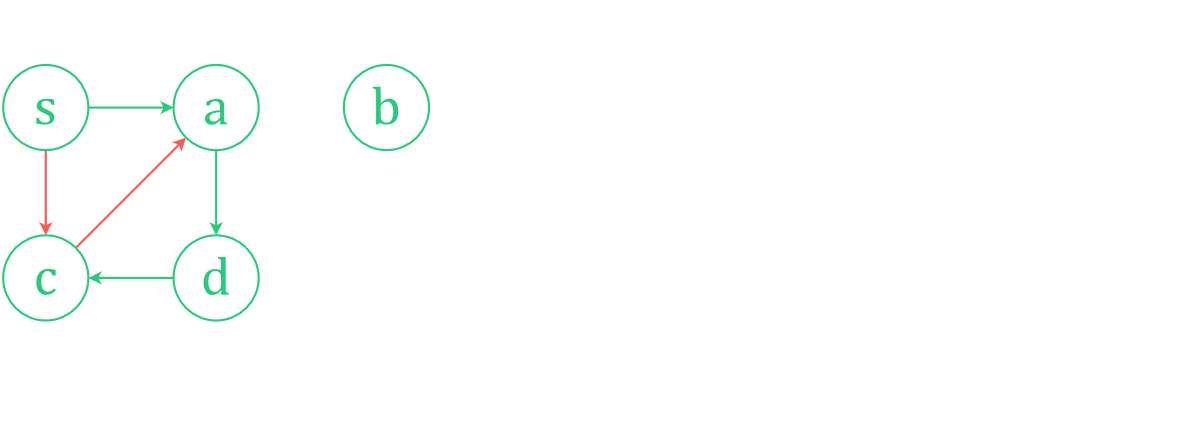






At , we can only go to , but since it is already visited, we cannot go there. Thus, we have reached a dead end and must backtrack. This is where the stack will help us. We start popping out elements one by one. At and , we find that we have no unexplored paths. At , we find an explored path, but it is to , which has already been visited. Our stack is also empty, but all the vertices have not been visited. This is because of the way the directed graph was set up.

Next, we will start looking for unexplored vertices, using the visited array. The first such vertex we find is .





The recursive function calls look something like this:

dfs(s)

dfs(a)

dfs(d)

dfs(c)

dfs(b)

dfs(e)

### Time Complexity

We are visiting vertices and each vertex can have at most edges in a directed graph and edges in an undirected one, making the time complexity .

### Edge Classification

The edges that we use to visit new vertices are called tree-edges. In the directed graph we used earlier, there were tree-edges between and , between and , between and and between and . Tree-edges are used to traverse from parent to child.

Every other edge is called a non-tree edge. These can be classified into three parts:

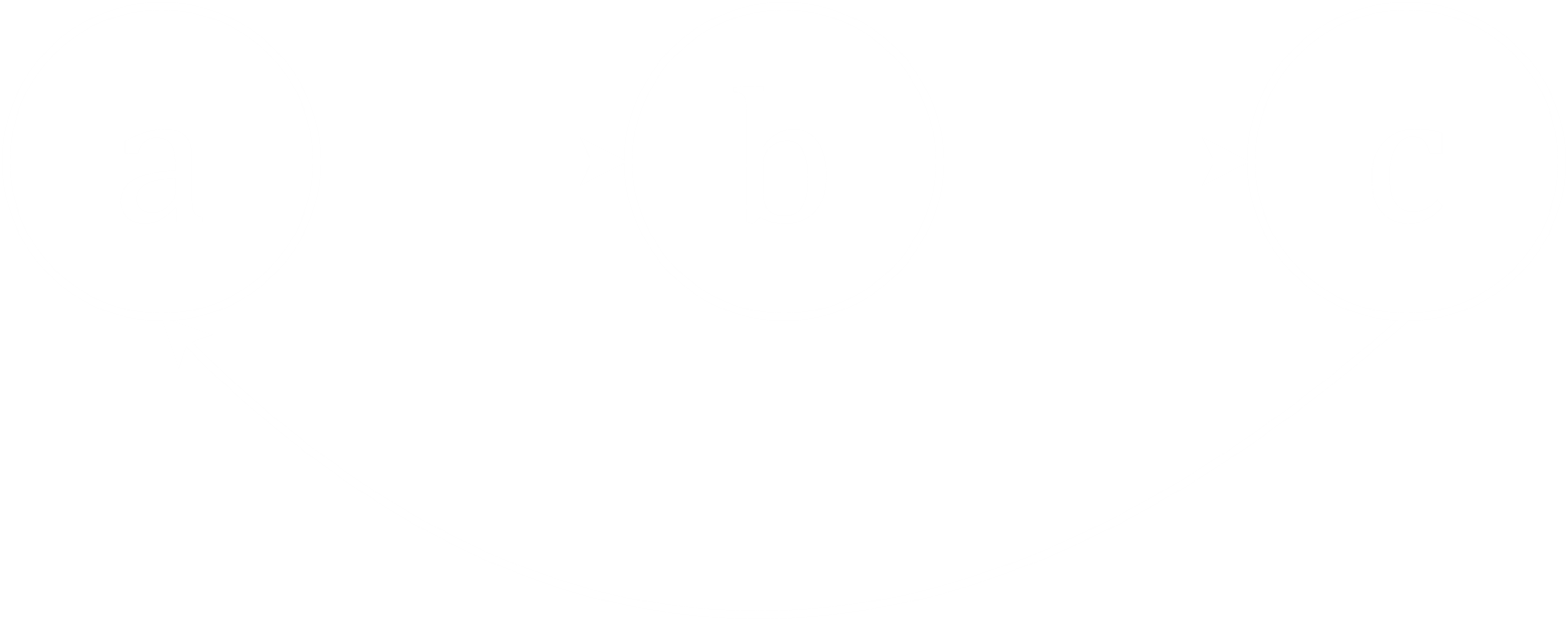
* Forward Edges - A forward edge goes from an ancestor to a descendant, such as the one from to . Note that a tree-edge cannot be a forward edge.
* Back Edges - A back edge goes from a descendant to an ancestor, such as the ones from to and from to , the self-loop.
* Cross Edges – Cross edges connect vertices that are not related by an ancestor-descendant relationship, such as the one from to . This relationship does not exist because they are not connected by a tree edge.

Note that, for an undirected graph, there can only be tree edges and back edges. There are no forward edges or cross edges.

### Cycle Detection

The classification of the edges can be useful. For example, whenever we have a back edge in our graph, we can say that there is a cycle in our graph. We shall now prove this in both directions.

First, lets try to prove that a back edge is proof of a cycle.



Starting at in the directed graph, we go to and then to . At , we find a back edge. This back edge is trying to take us to a vertex that is already in our stack, . Since is in our stack, we say that we have not finished traversing it yet.

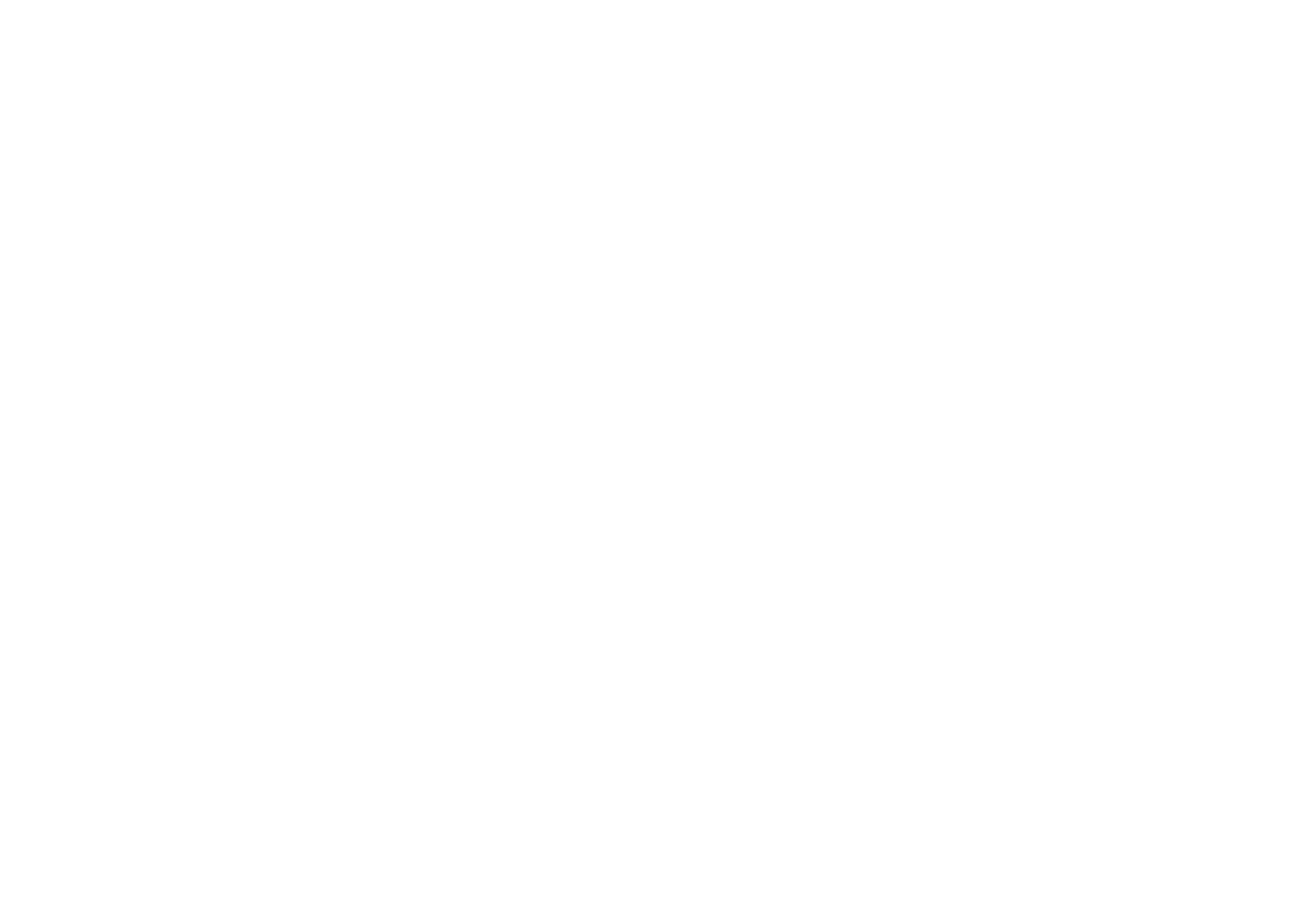
If we have a back edge from to , the definition of a back edge tells us that must be a descendant of . This means that there is a path made of tree-edges connecting to . Thus, there is definitely a cycle.

Similarly, as we follow the tree-edges from to , when we get to , we find an edge back to , which forms a cycle. However, is still in the stack and its descendants are still being traversed. This must mean that is a descendant of , which makes the edge we just found, by definition, a back edge.

## Topological Sort

A directed acyclic graph (DAG) is a directed graph that does not contain any cycles. DAGs can be used to find the schedule in which tasks that have dependencies should be executed, such that the dependencies are not violated. Here, vertices are used to denote tasks and edges are used to denote dependencies.

Consider the example below.



Algo, and DBMS are dependent on DS, DS is dependent on P&S, DM and SP-II, P&S is dependent on DM and SP-II is dependent on SP-I. Obviously, we cannot start the execution at any of the vertices that have some dependency. Since SP-I, DM and SA are the only tasks that are independent, we have to start from one of those and find the proper order of task execution.

Let’s say we start with SP-I. If we perform a depth first search from that vertex, the traversal will follow the pattern SP-I SP-II DS DBMS. DBMS has no children, so that is the only vertex we have finished traversing for now. Since we are done traversing it, we can add DBMS to a stack.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DBMS |  |  |  |  |  |  |  |

Next, we go back to DS and explore its other child, Algo. Similar to DBMS, Algo is also added to the stack.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DBMS | Algo |  |  |  |  |  |  |

Again, we return to DS and find that we have finished traversing all its children. This also means we are done traversing DS and can push it onto the stack.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DBMS | Algo | DS |  |  |  |  |  |

Going further backwards, we find that we have finished traversing SP-II and SP-I as well. We add these to the stack in the same manner, following the correct order.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DBMS | Algo | DS | SP-II | SP-I |  |  |  |

There are three more vertices left that are untraversed, because they were not reachable in this first part. Now we have to start traversing from another vertex that has no dependencies. Let’s say we start from DM this time.

DM has one child, DS, that is already traversed, so we can ignore it. Instead, we go to its other child, P&S. P&S has just the one child, DS, which has been traversed already. Thus, P&S has been traversed and can be added to the stack.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DBMS | Algo | DS | SP-II | SP-I | P&S |  |  |

After that, going backwards, all children of DM have also been traversed and DM can be added to the stack.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DBMS | Algo | DS | SP-II | SP-I | P&S | DM |  |

The only vertex left to traverse is SA, which has no dependencies or children. It is traversed as soon as we visit it. Thus, SA can be added to the stack.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DBMS | Algo | DS | SP-II | SP-I | P&S | DM | SA |

Notice that if we start popping elements from the stack now that it is complete, the elements will appear in the correct order of execution that does not violate the dependencies. This entire process is called Topological Sort.

Note that topological sort gives us just one of the possible orders. There might be more possible orders.

The order of operations we performed here are:

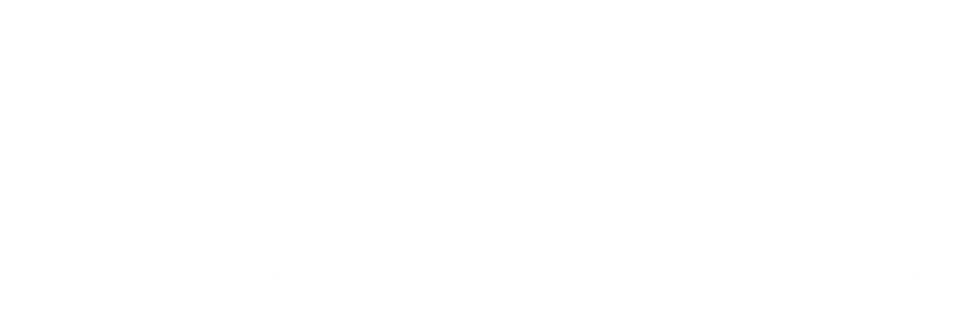
* Run DFS on any independent vertex
* Push vertices that have been completely traversed onto a stack
* Avoid repeating vertices
* Continue running DFS on other independent vertices until all the vertices have been traversed
* Pop the elements from the stack and print them (Task Ordering)

### Time Complexity

The time complexity of topological sort is . We ran a normal DFS, which is , and popping all the elements will take time as well.

### Proof

In topological sort, the claim we are making is that for any edge in a DAG, finished before in DFS.



Consider the two vertices above. We have a tree-edge from to . This means that has to be completed before can be completed. The only way for it to ever be possible for to be completed first, is if we had a back-edge from to . However, by definition, a DAG cannot have any back-edges, since they will form cycles.

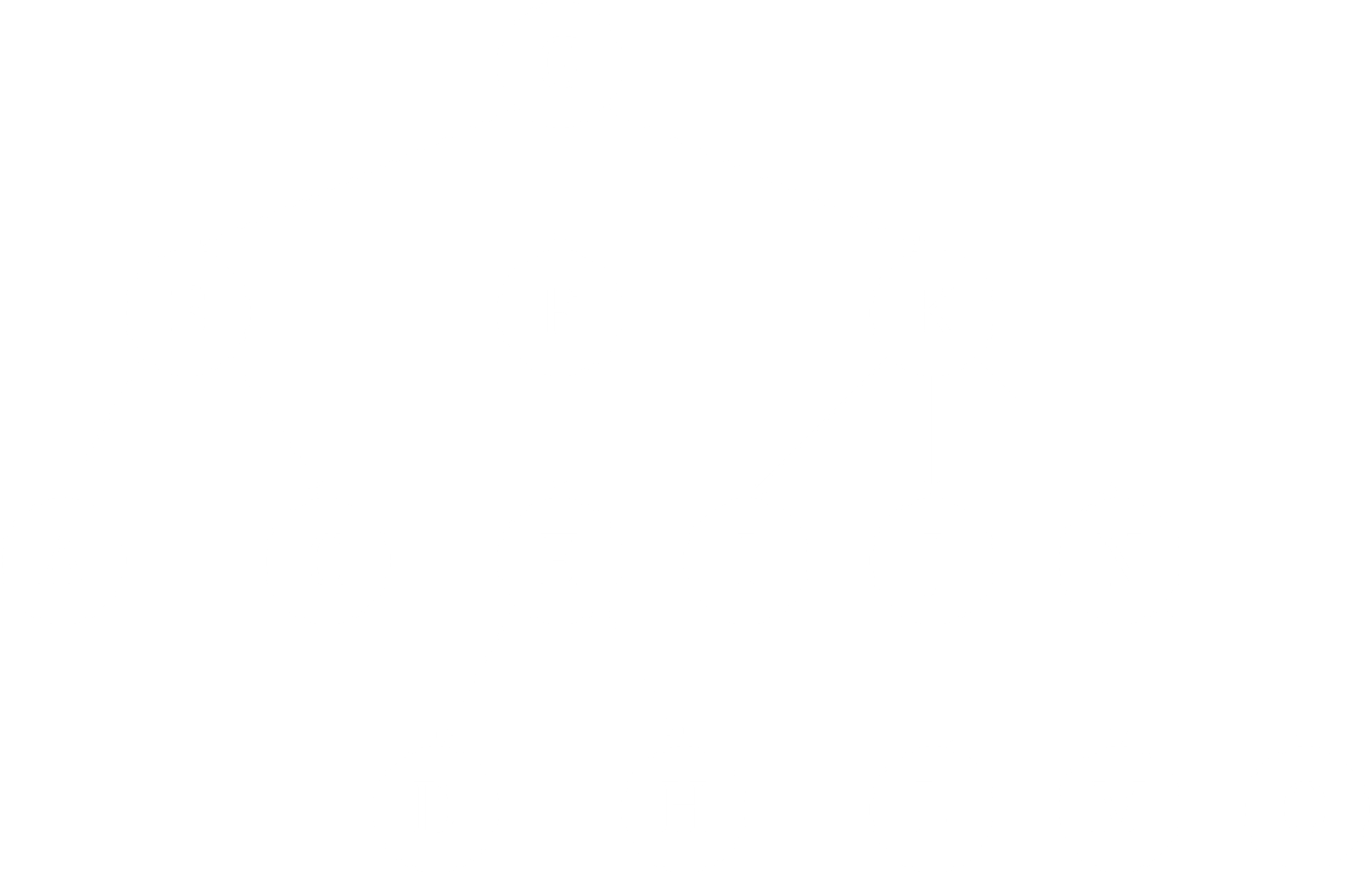
Conversely, let’s say we started working from . In that scenario, the only way to even get to is if we have an edge from to . Again, we find ourselves in the same situation, since if we have such an edge, the graph will not be a DAG.

## FAQ About Graph Traversal

1. What are the advantages and disadvantages of adjacency lists compared to adjacency matrices?

* **Memory Utilization** - If we are concerned about memory usage, we should use an adjacency list. The space complexity of an adjacency matrix is , whereas the space complexity of an adjacency list is . However, if we ignore multiple edges between a pair of nodes and self-loops, the maximum number of edges in a graph can be , since every vertex can be connected to every other vertex. In this scenario, the space complexity of an adjacency list will be as well. Despite this fact, adjacency lists are still better in terms of space complexity in most cases, since this worst-case scenario does not usually arise.
* **Finding Neighbours** – Using an adjacency list, we can find all the neighbours of a particular vertex by simply traversing the vector connected to that vertex. However, with an adjacency matrix, we would need to check every cell in the row or column for that vertex to see which values were 1, i.e. we would need to check against every vertex in the graph. This can be useful if we want to run BFS after DFS as well.
* **Checking for Connections between Two Nodes** – In contrast to the previous point, if we wanted to check if two particular nodes were connected in the graph, an adjacency matrix would be better. This is because we would only need to visit a single cell. With an adjacency list, we would need to traverse the entire vector connected to one of the vertices.

2. Consider the graph below.



Which of the following are possible results for a BFS traversal of the graph?

1. G F B K E C A I N J D H M L O
2. G K F B I J N E A C O M L H D
3. G F B K E C A I N J O M L H D
4. A B G C F K E I J N H D L M O

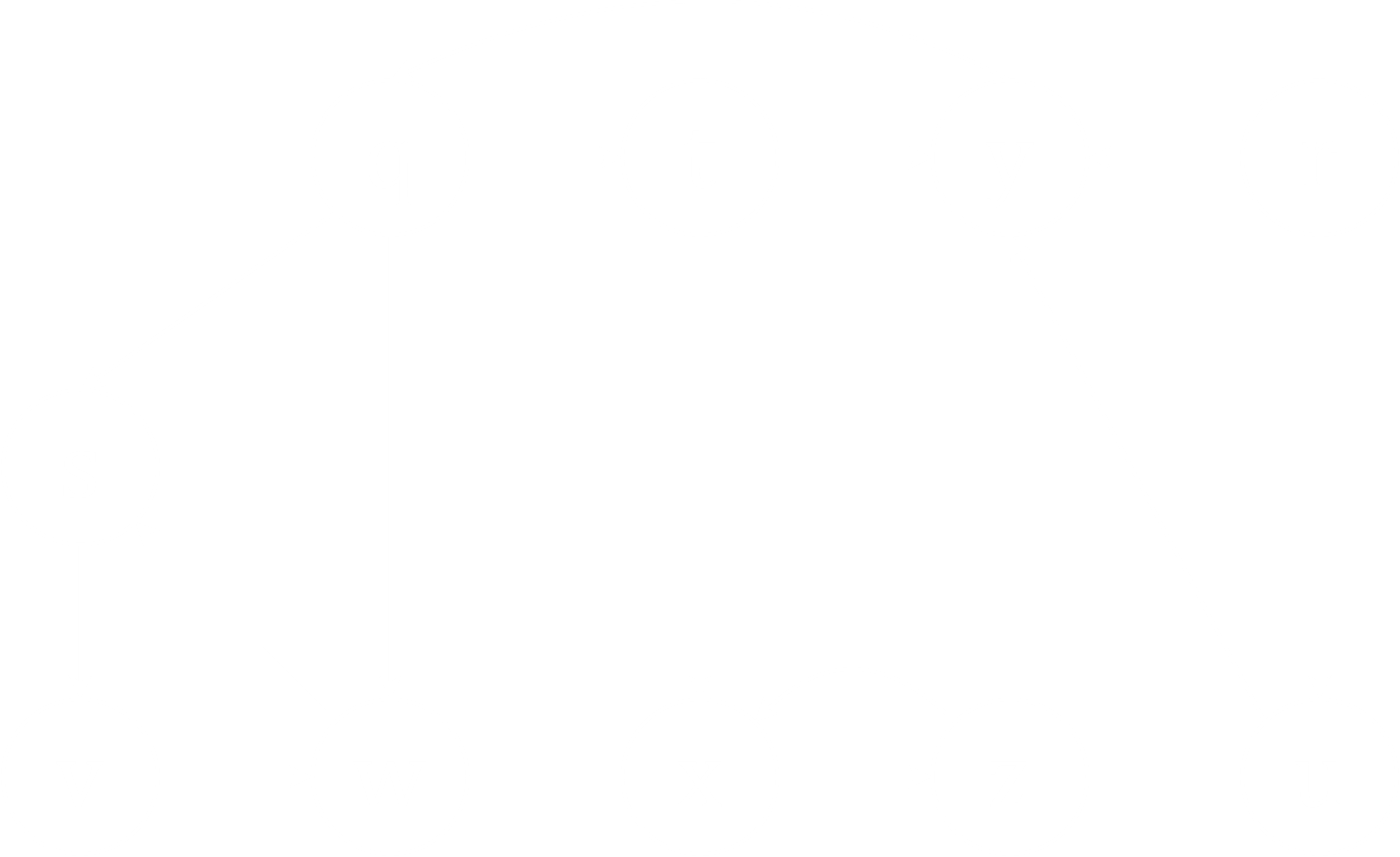
The basic principle is that we need to go level-by-level, in the correct order. Starting at , we can visit , and then . This is fine. However, we need to start traversing the next level using the same ordering we used in the first level, meaning the children of must be traversed first. This is because of the queue we are using in BFS. Thus, comes next, followed by and and then , and . Following the same pattern, the children of come first, and , and finally the children of , , and . This means the first pattern is perfectly valid.

For the second pattern, the exact same process is followed and we will find that it is valid as well. The only difference is that the order in which the first layer was traversed has been changed, with coming before and . Everything else changed accordingly.

For the third pattern, the process starts off in the same way as the first pattern and everything is alright for the first two layers. In the third layer however, we traversed the children of before traversing the children of , which is of course wrong. Thus, the third pattern is invalid.

The last pattern may seem ridiculously invalid at first, but this one is also valid. This is because we have not been told that the starting point of the traversal has to be . If we start from instead, we will find that the traversal is completely valid.

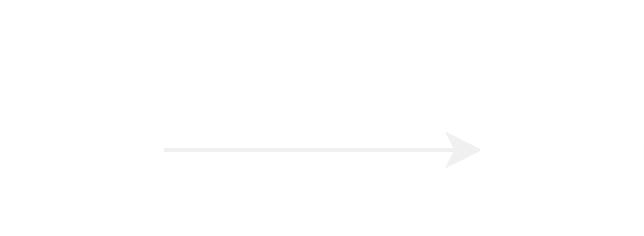
3. For the following graph, starting from q, traverse the graph using DFS. Assume the graph uses an adjacency list that is sorted in alphabetical order, where the vectors are also in alphabetical order. Also classify the edges.



|  |  |  |
| --- | --- | --- |
| Edge | Class | Discovered |
| q s | Tree-edge | q, s |
| s v | Tree-edge | q, s, v |
| v w | Tree-edge | q, s, v, w |
| w s | Back-edge | q, s, v, w |
| q t | Tree-edge | q, s, v, w, t |
| t x | Tree-edge | q, s, v, w, t, x |
| x z | Tree-edge | q, s, v, w, t, x, z |
| z x | Back-edge | q, s, v, w, t, x, z |
| t y | Tree-edge | q, s, v, w, t, x, z, y |
| y q | Back-edge | q, s, v, w, t, x, z, y |
| q w | Forward-edge | q, s, v, w, t, x, z, y |
| r u | Tree-edge | q, s, v, w, t, x, z, y, r, u |
| u y | Cross-edge | q, s, v, w, t, x, z, y, r, u |
| r y | Cross-edge | q, s, v, w, t, x, z, y, r, u |

## Weighted Graphs

We shall now consider how to get from one point to another in a weighted graph.



In the above diagram, there is an edge between the vertices s and t, and the edge has a weight w.

One example of what the weight of an edge could signify, is the amount of traffic on a certain path, such as those displayed on Google Maps. A simpler example could just be the distance between the vertices.

Thus, the problems we will be looking at could be related to finding the shortest distance between two points, or the path with the least traffic. Essentially, we will be looking into how to find the path with the least total weight.

### Representation

is a weighted graph with weight , where is the set of vertices, which could correspond to street intersections in the example we just saw, is the set of edges, which could correspond to streets, and is the set of weights associated with each of the edges, , meaning for each edge, the weight will be a real number.

We will be learning two algorithms here

* Dijkstra (pronounced di-ak-stra)

Dijkstra has a time complexity of . It works with non-negative edge weights.

* Bellman-Ford

Bellman-Ford has a time complexity of . It is able to work with negative edge weights.

Notice how the time complexity of the algorithms is independent of the weight, , of the graph.

### Notations

We donated the fact that there is a path between the vertices and like this:

This path is described in detail, showing every vertex involved in the path, like this:

where for . Here, every vertex has an edge with the two vertices on either side of it in the list.

The weight of the path is given by the sum of the weights of the individual edges.

There are two special cases for the weight.

* The weight from a vertex to itself is .

For , .

* The weight between vertices that do not have a path between them is infinite.

For no path, .

The least possible path weight between two vertices is found by finding the minimum value of the weights of all the possible paths between those vertices. This minimum value is denoted by (delta).

In the problems we will face, we will be given a graph and a source vertex, . Our job is to find the shortest path between the source and every other vertex, i.e. for all .

### Solution

### Mathematical Description

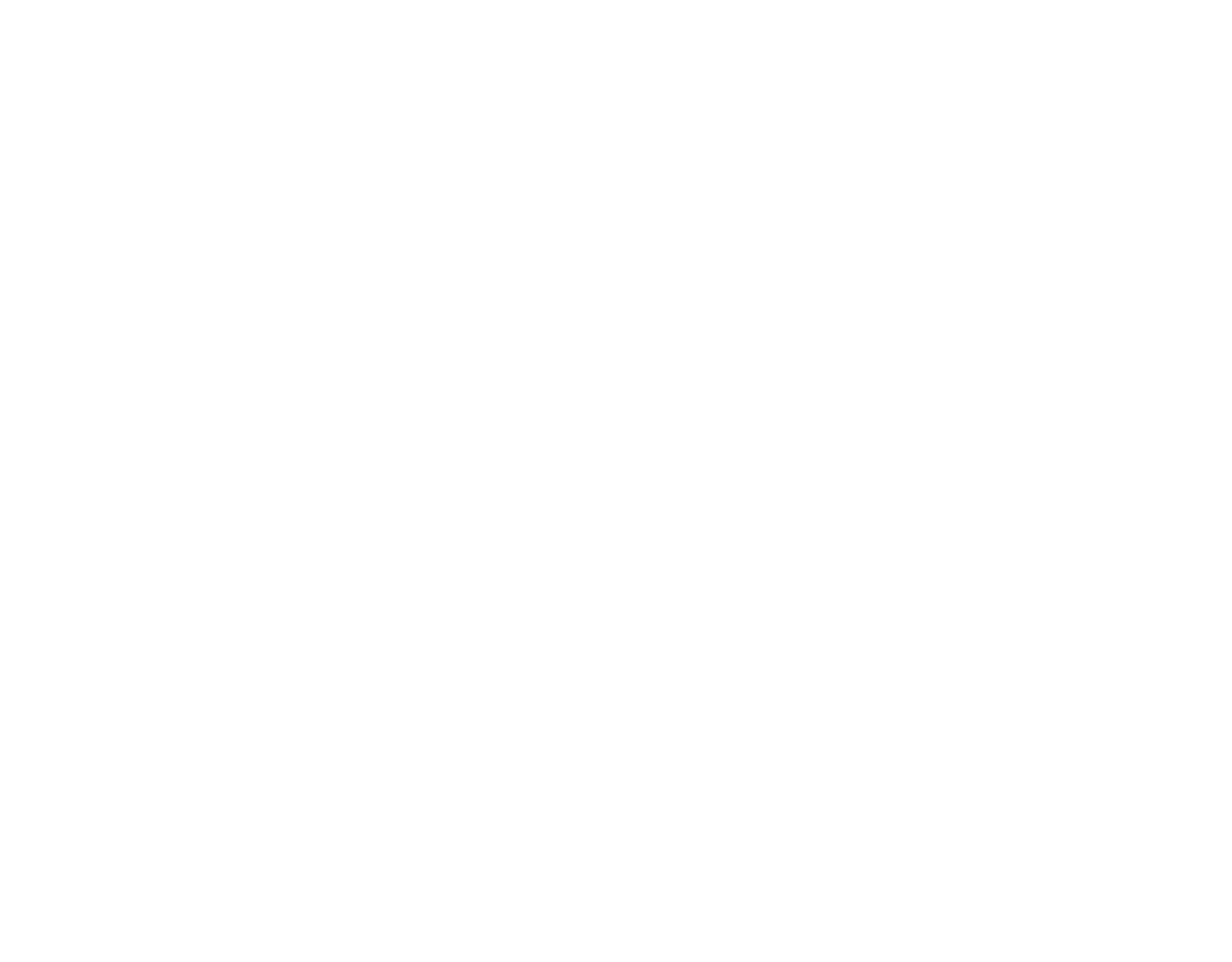
We will keep an array to keep track of the minimum distance, between and each of the vertices, . First, we will set a default value to each vertex. This value is if the vertex is , and otherwise. This is because we do not know whether paths exist between and those vertices. We will replace these values if we find smaller ones.

At the end of the algorithm, of course, we will have the least values for each of the paths.

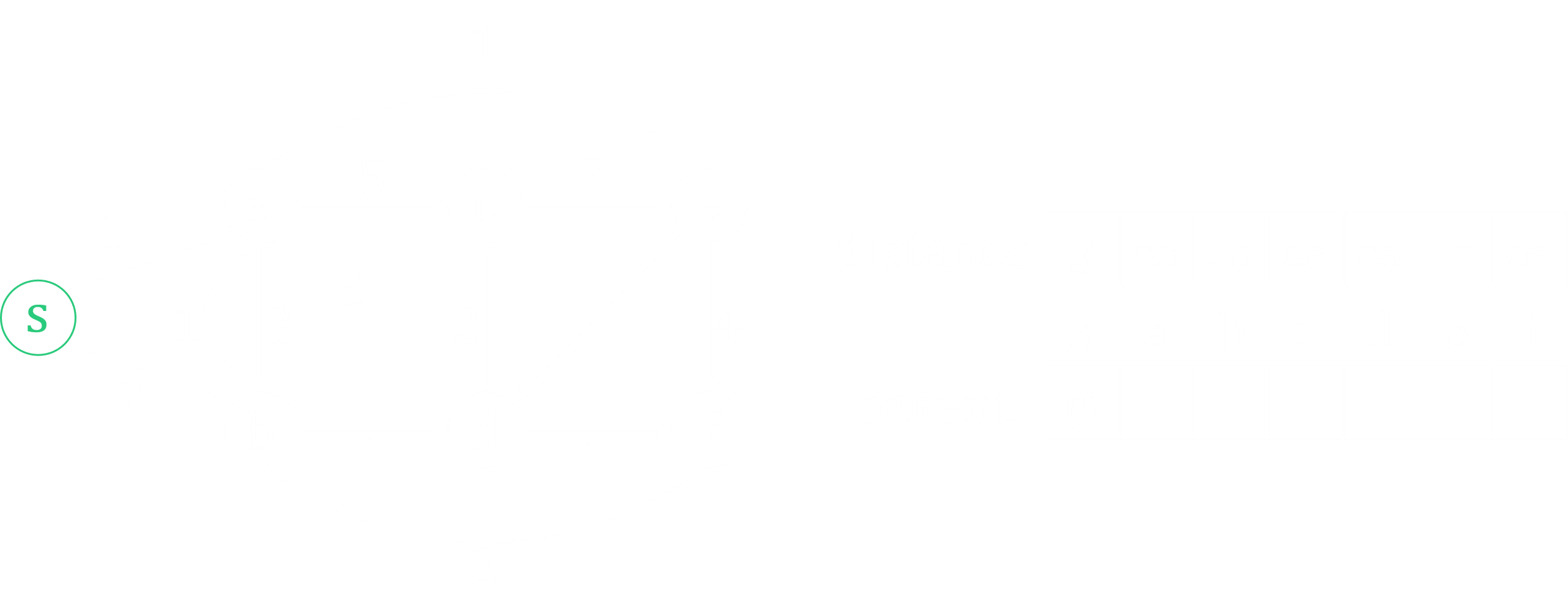
We will find different paths from to , and whenever we find a path with a total weight that is less than our current value, we will replace it. However, there is no possibility of ever becoming smaller than .

We will be iterating through all the edges in some manner, while also keeping track of the parent of on the best path, . Of course, .

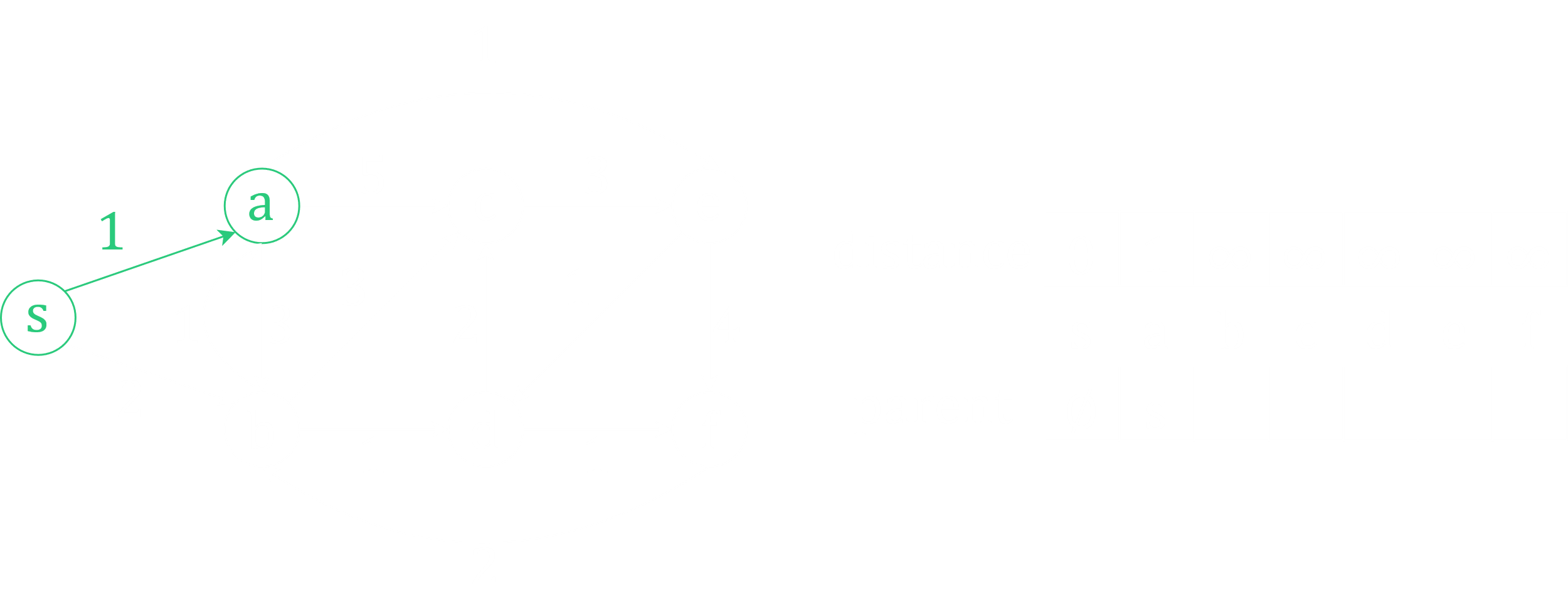
#### Algorithm



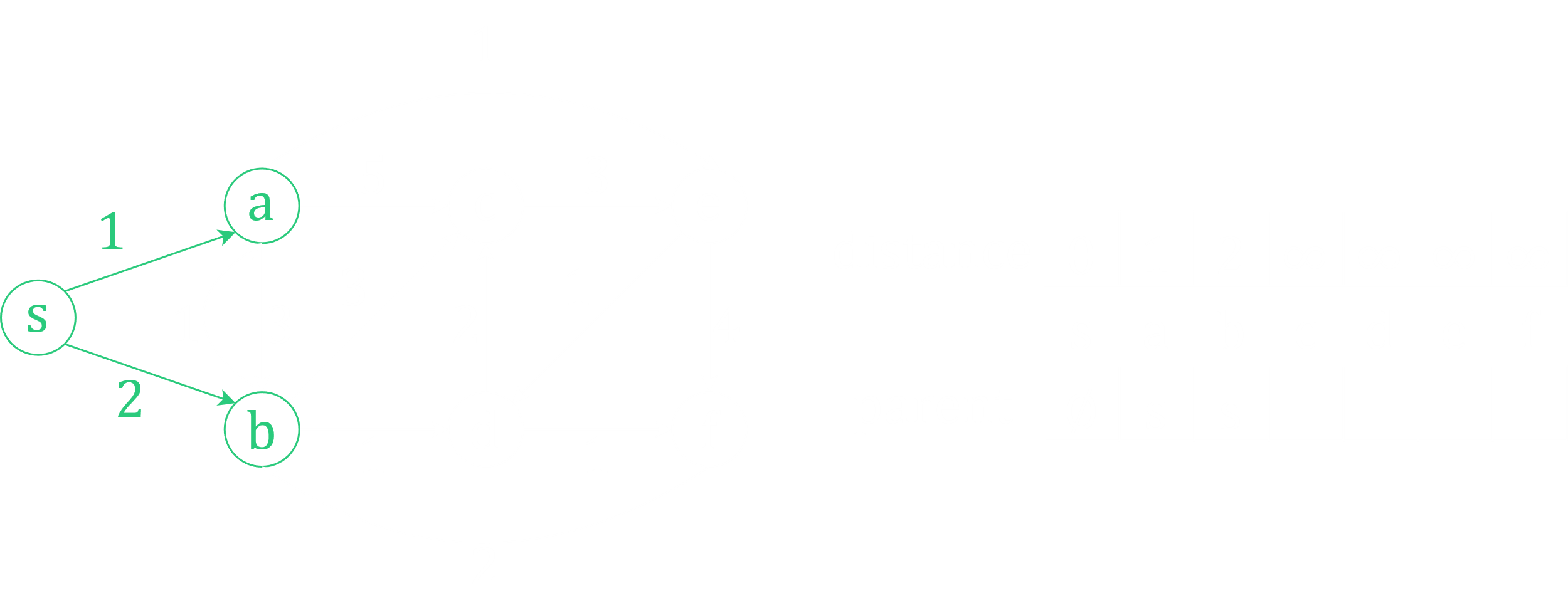
In this example, our source node is . Thus, at the beginning of the algorithm, the value for the distances of all other vertices is .



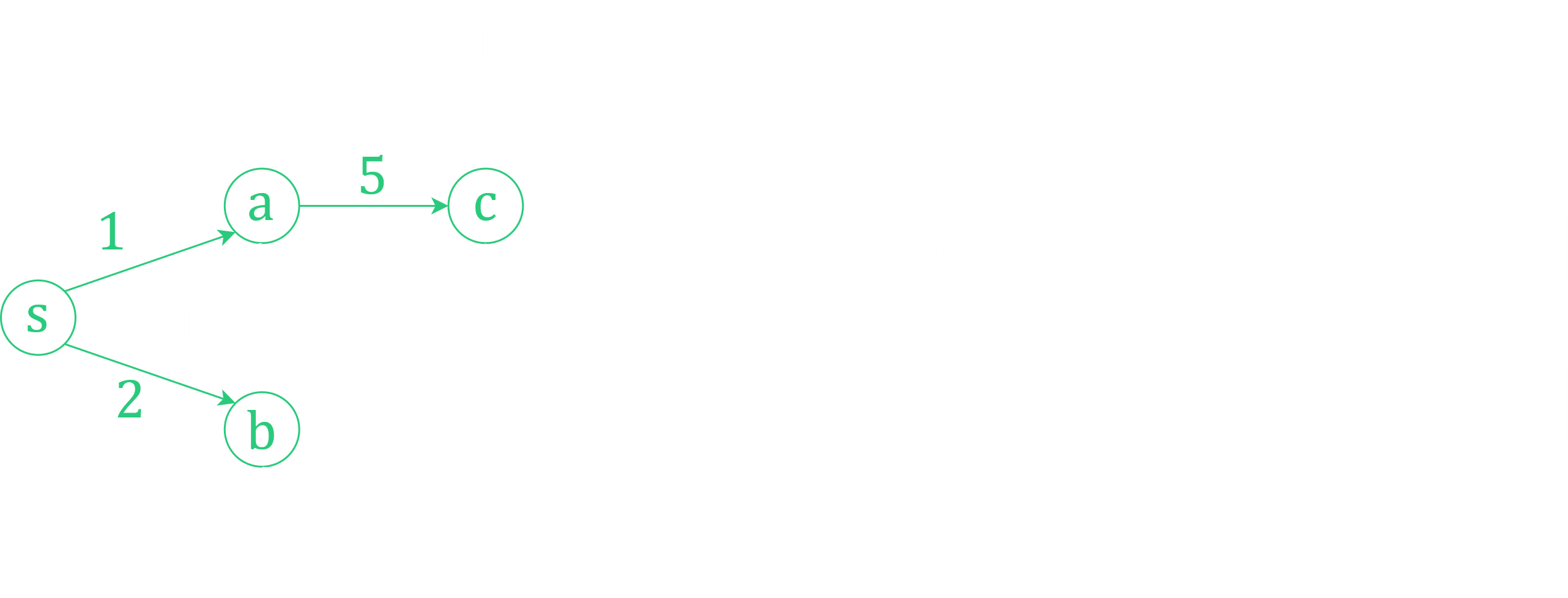
Now, we begin traversing each of the edges. First, we find that is and .



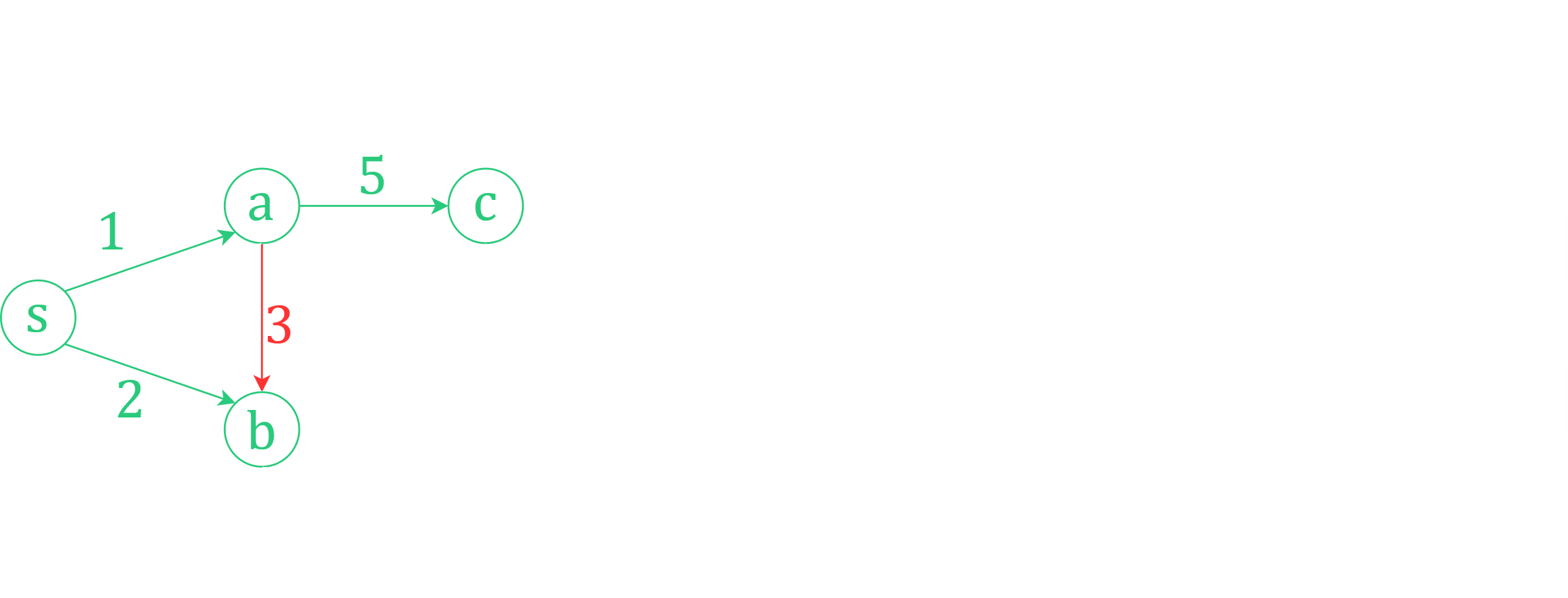
Again, from we can go to , and find that and .



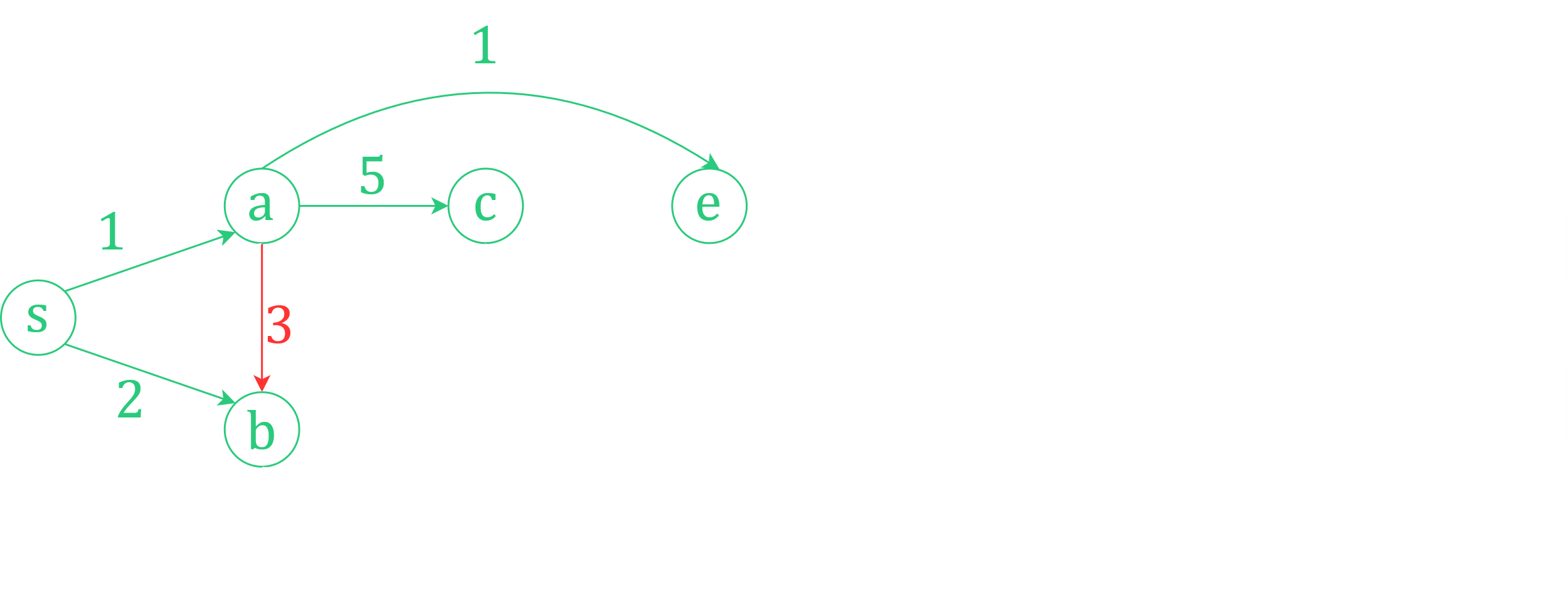
Traversing , we find that we can get from to . Here, . This is the least value we have found so far, so we will update our distance array. Also, .



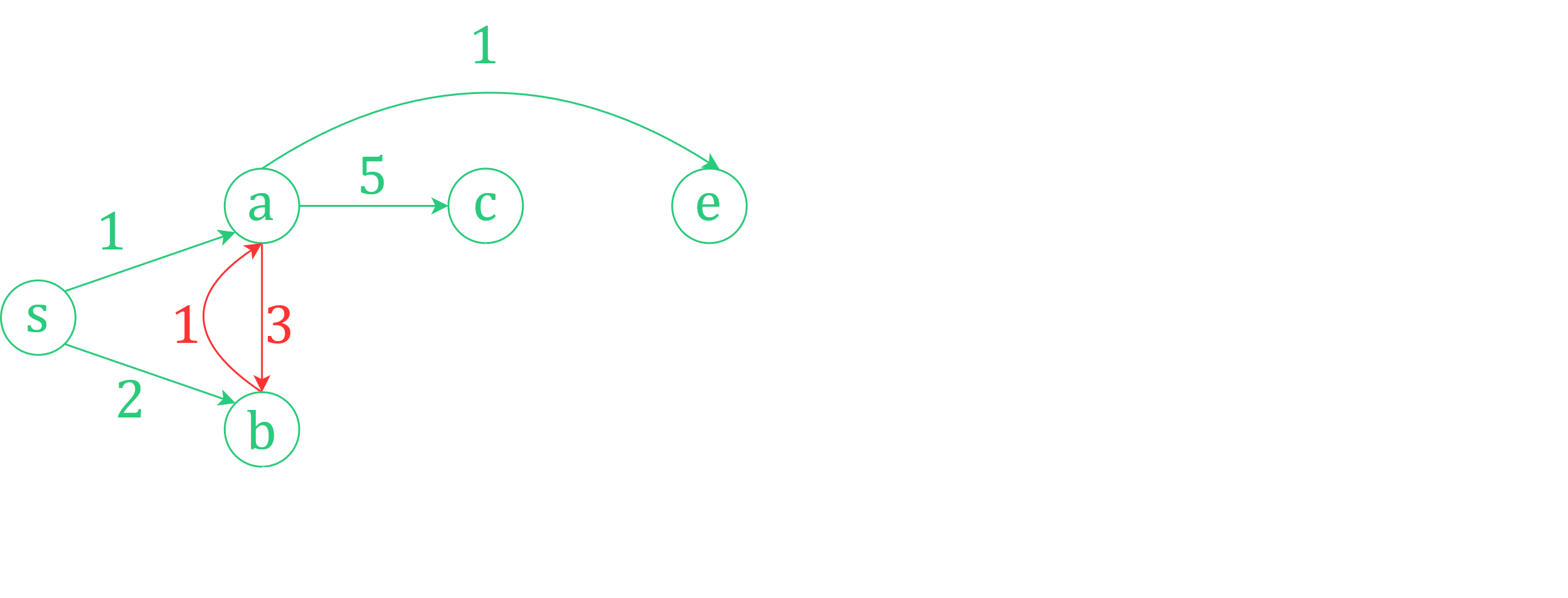
There is a path from to as well, but following this path would give us . Since this is larger than the value of we currently have, we do not use this.



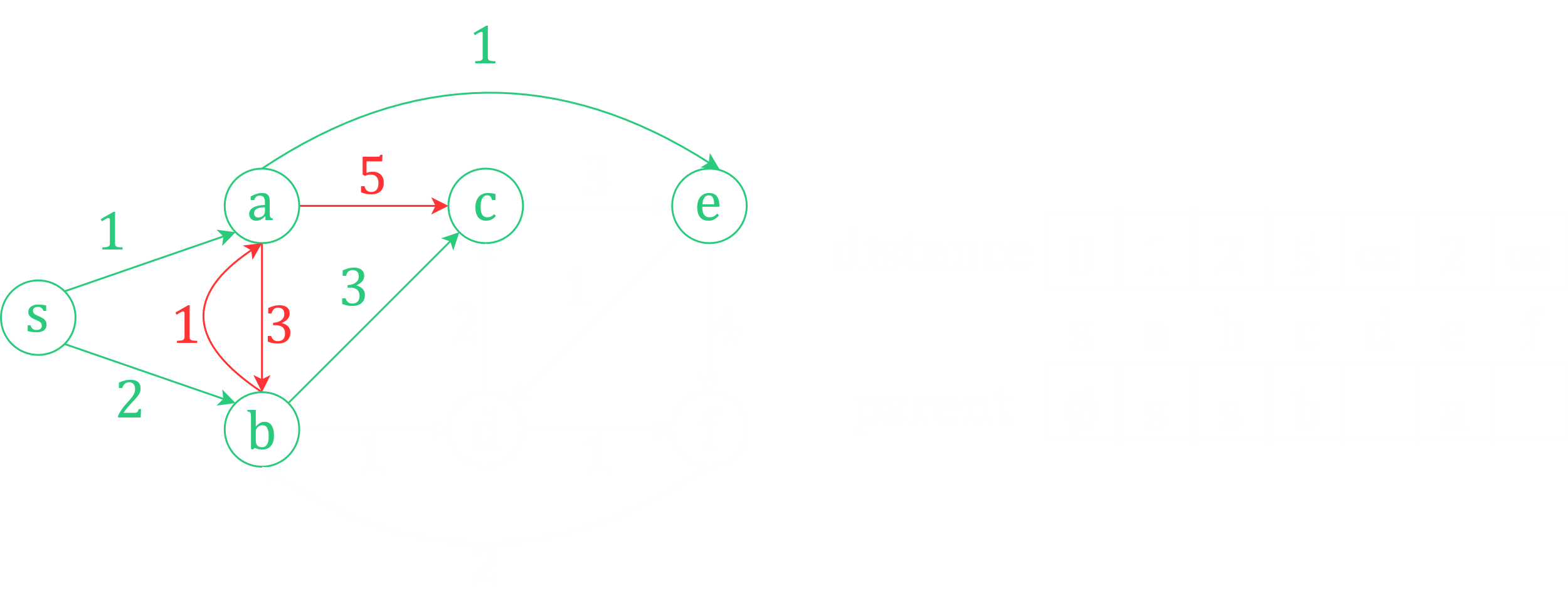
Finally, there is an edge from to . This gives us . This is the smallest value we have found so far, so we update our arrays accordingly.



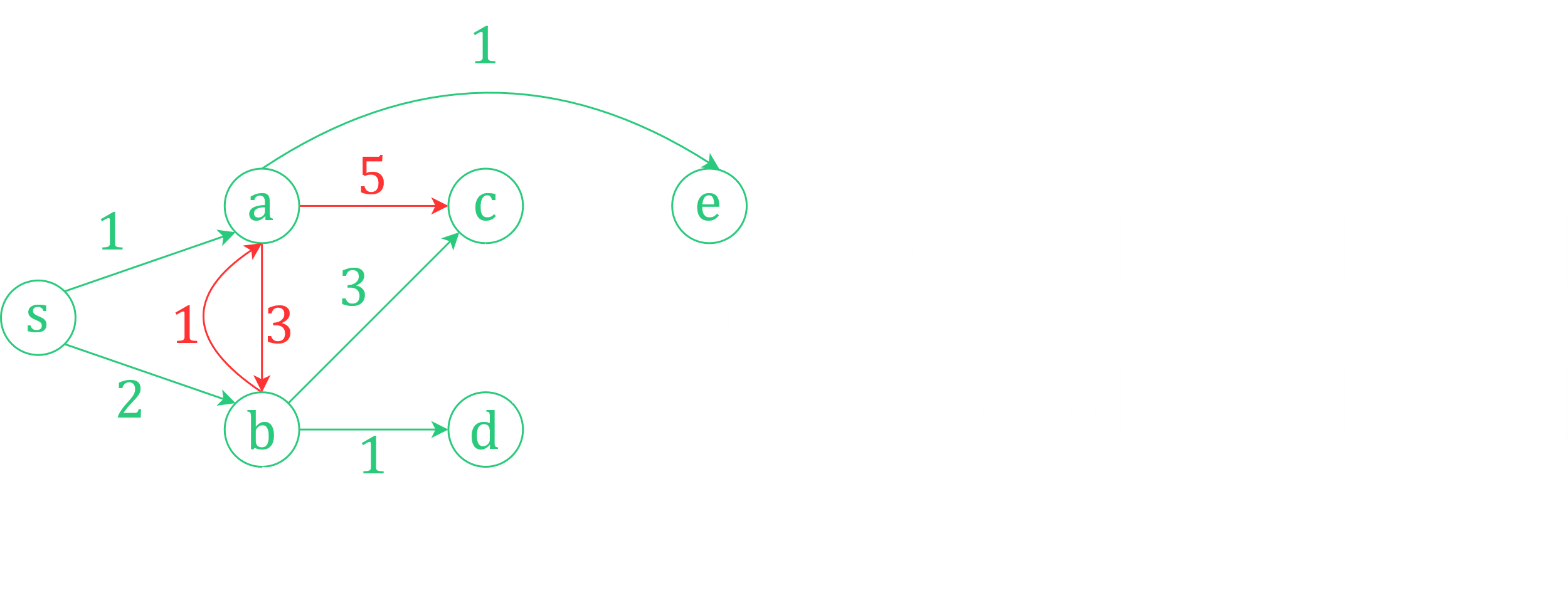
Traversing , we see that there is a path from to . This means, the path from to could also be . However, the distance from to , is , which would make . Clearly, this is larger than the value of we currently have, so we will not change the value.

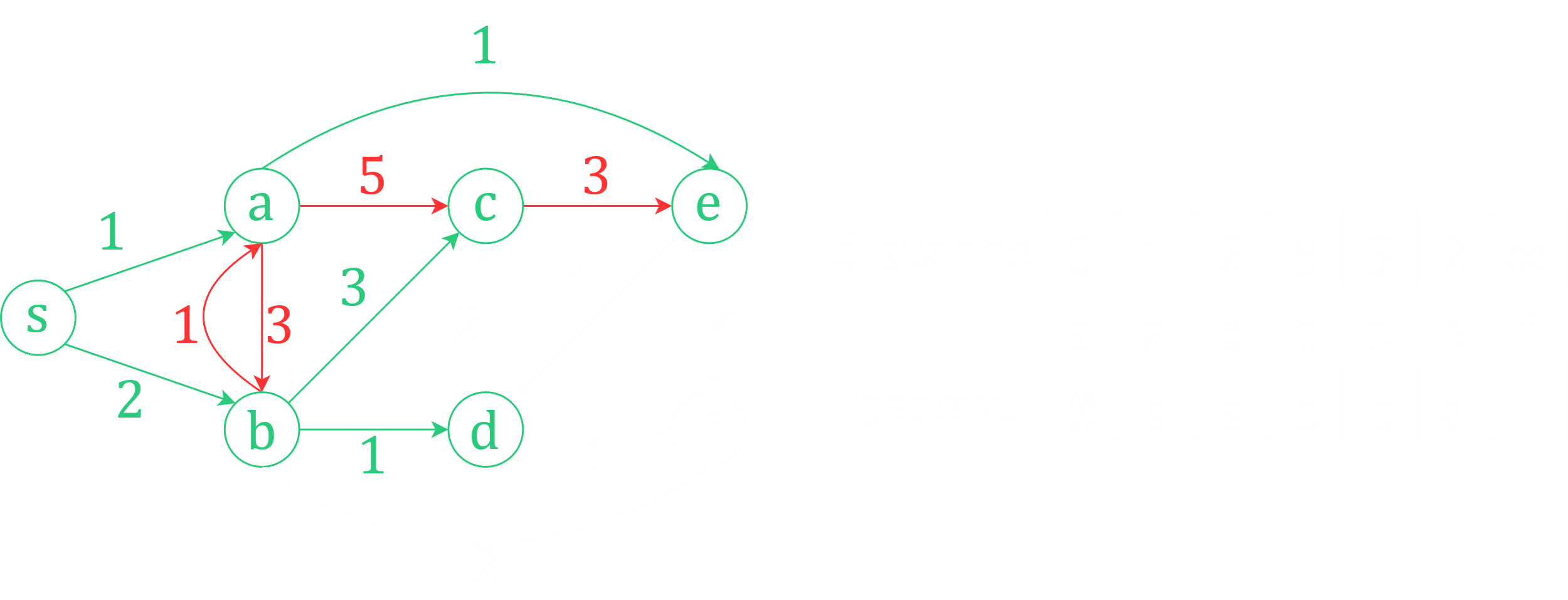


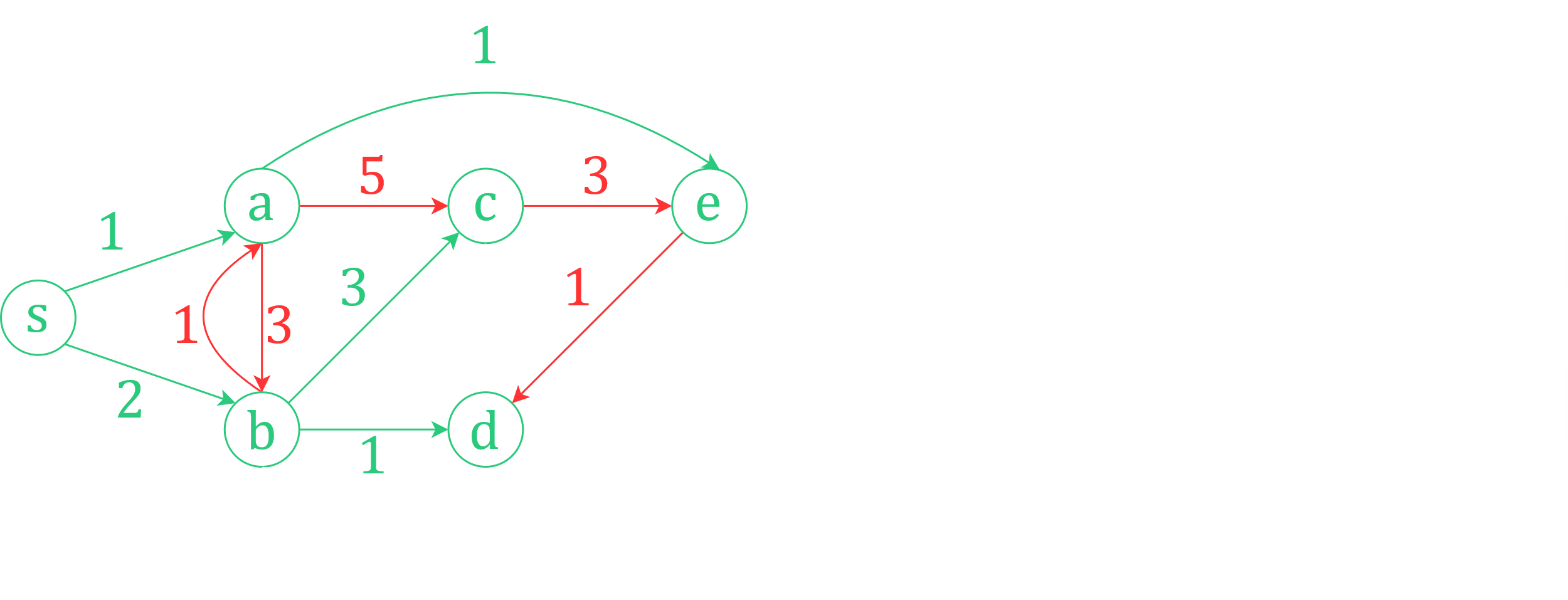
There is a path from to . If we use this path, . This is smaller than the current value of , so we will update it and also set .

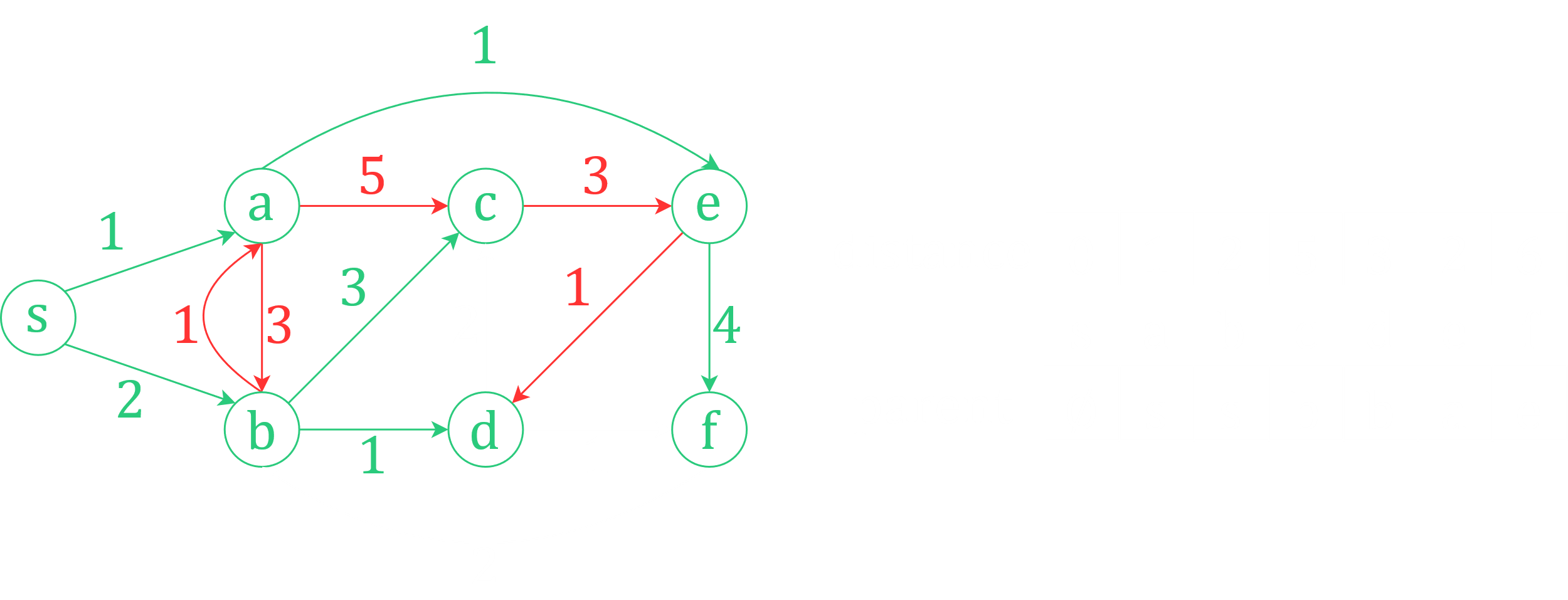


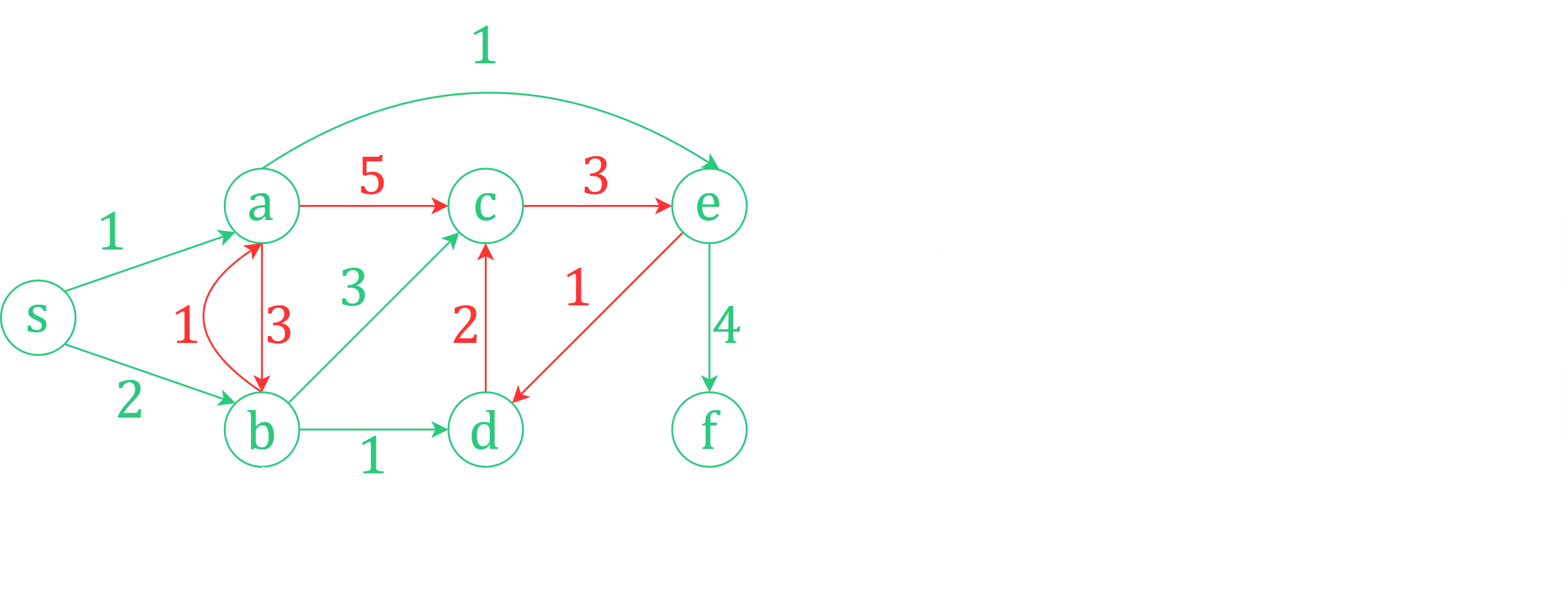
We will continue in this manner.

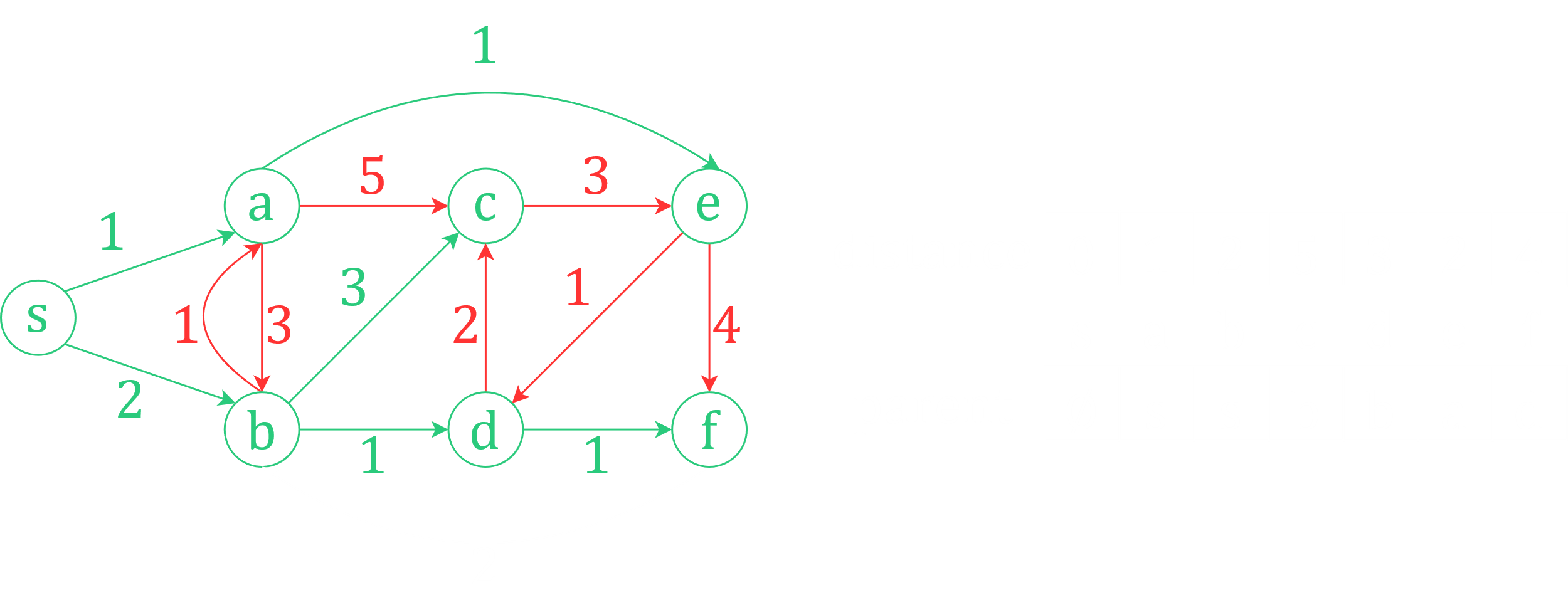


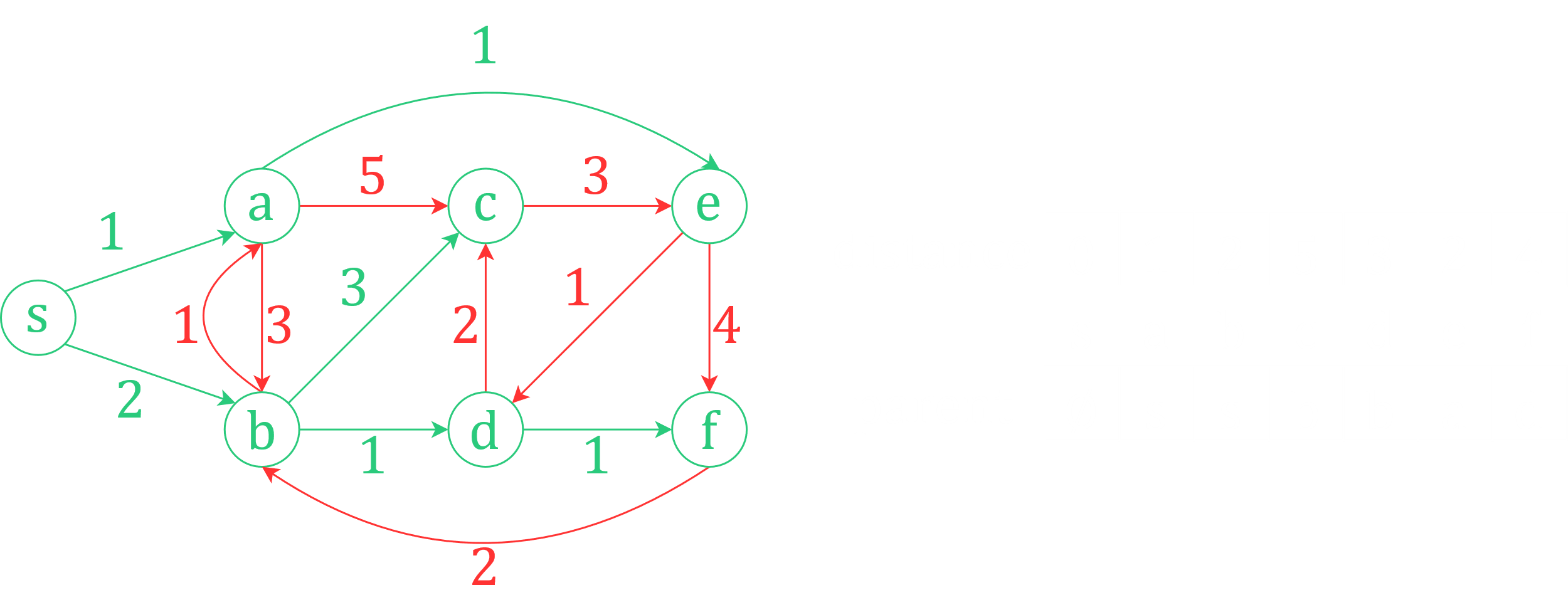












However, there is a problem here. If the graph has self-loops or cycles, we might fall into an infinite loop of repeatedly relaxing edges. Thankfully, that does not happen in this case. However, we will be looking at an example of this next. To prevent this, we need some algorithms that help us choose edges.

### Negative Edge Weights

Consider that we have a bus travelling along a particular route. Of course, we want to maximize our income. This depends on two things, the number of passengers we get on a route, and the amount of money we earn per passenger on the routes. There are also a number of expenses we need to take into consideration, such as fuel costs, tolls, staff payment, maintenance costs, etc. Taking all of this into consideration, we can assign a weight to each of the edges.

If the total income for an edge is less than the total cost, we will face a loss. This is given a positive value. If the total income is more than the total cost, we will gain a profit. This is given a negative value. The reason we do this is because the algorithms we will use will give us the least possible total value, which should be the most negative value in this case.

#### Example of the Problem

We could argue that the method we saw previously should work here as well. To understand the flaw in that thinking, consider this graph.



Till here, everything is fine. Now however, we shall begin to face problems. There is an edge from to here. If we consider this edge,

This is less than our previous value, so our algorithm will replace it. And thus, we shall begin an infinite loop.

This sort of situation is called a negative weight cycle. Because of the existence of the cycle, we are falling into an infinite loop. Clearly, the algorithm we have learnt is insufficient to solve this problem, and we need to work on something else.

## Theorems Related to Single-Source Shortest Path Algorithms

### Optimal Substructure

This theorem states that the sub-path of a shortest path is also a shortest path.

Consider that is the shortest path from to , and . The theorem tells us that in this case, each of the paths where are also the shortest paths between and .

The proof of this is that the algorithm we use to find the shortest path between any two vertices simply follows the shortest paths between each of the adjacent vertices between them. Thus, if one of the sub-paths were not the shortest path between two vertices and some other path was the shortest, the algorithm would have taken that path instead. If one of the sub-paths were not the shortest path, then the path could not be the shortest path either.

### Triangular Inequality

The triangular inequality theorem states that for all , and , . Essentially, it means that the shortest path between two vertices must be less than or equal to all other possible paths between those two vertices.

The proof of this is pretty straightforward, since if some other path had existed that was shorter, that would have been the shortest path instead.

## Dijkstra’s Algorithm

### Proving Edge Relaxation is Safe

Before we begin discussing Dijkstra’s algorithm, we need to discuss one lemma.

The lemma states that the relaxation algorithm maintains the invariant for all .

In the process of edge relaxation, we start by giving a very high value, , and gradually relax it to bring it down to . Thus, we start off with , for some vertex .

From the triangular inequality, we know that . This can be rewritten as , since we know that .

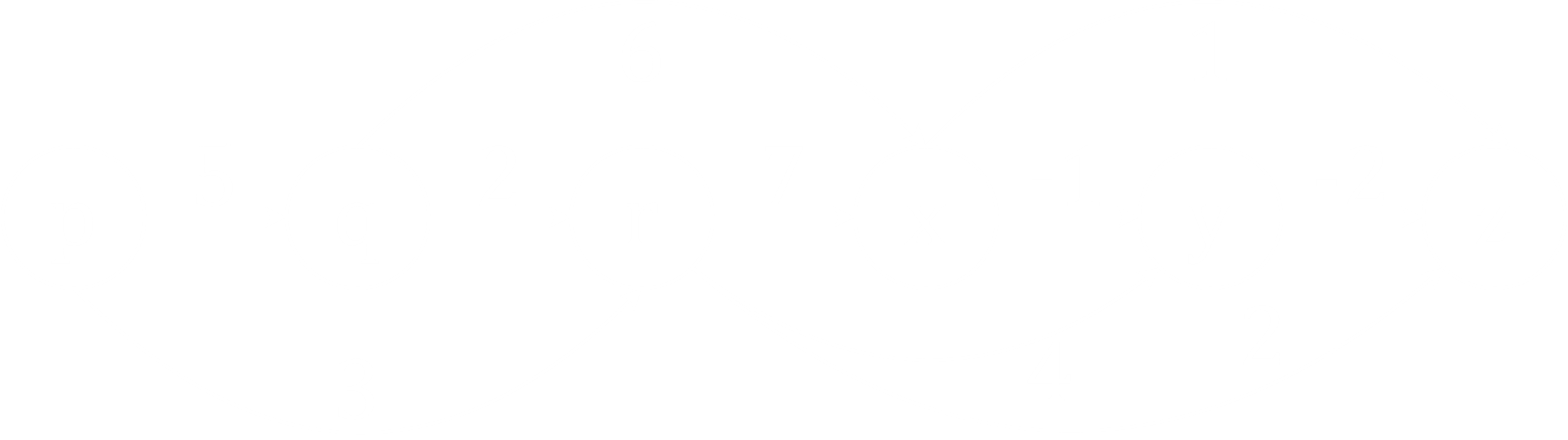
Each path between and has some cost, . Since is the shortest path from to , we can say that . Again, putting this into the previous equation we get , which can simply be written as .

### Shortest Path in DAG

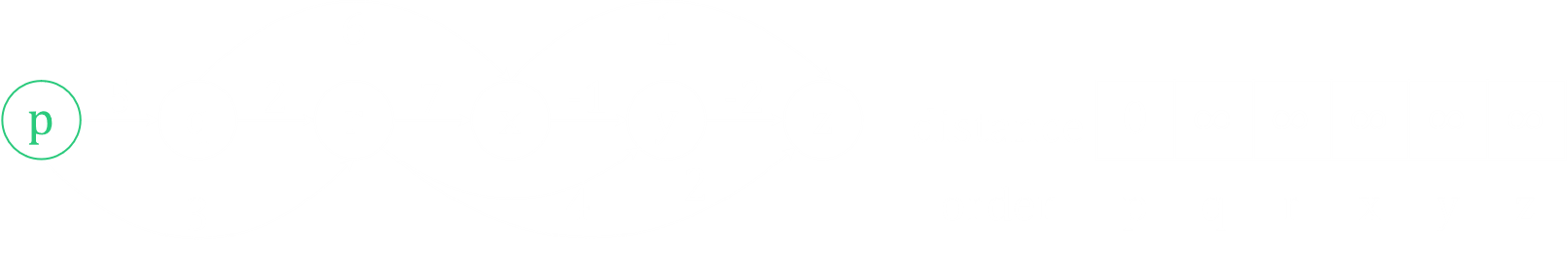
A major problem with edge relaxation is that we do not know how many times we will need to relax edges. This could be exponentially large, as we have previously seen. Our goal is to find an order in which to relax edges such that there is a limited number of relaxations.

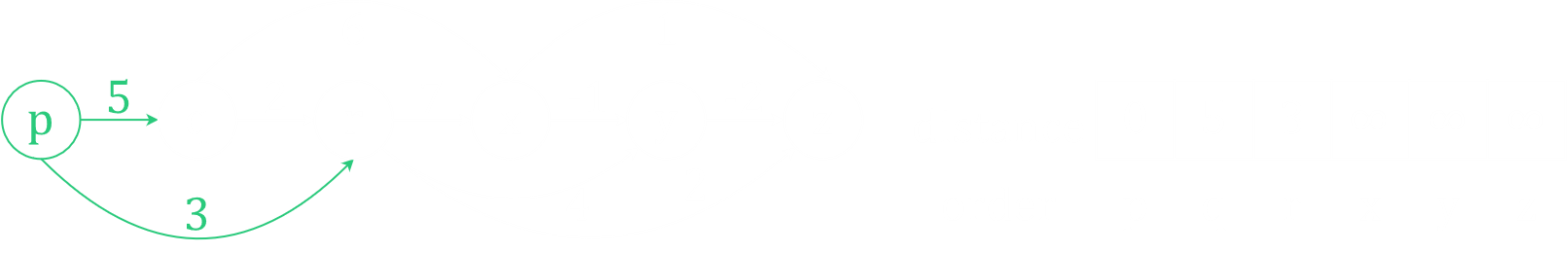
For DAGs, we have already learnt that topological sorting will provide us with an order to traverse the graph. We can use this order to relax edges in a DAG.

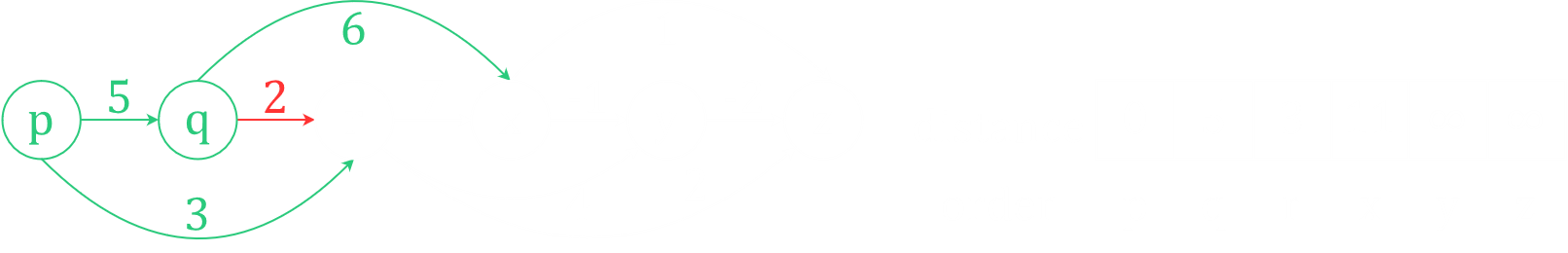
Consider the graph below:

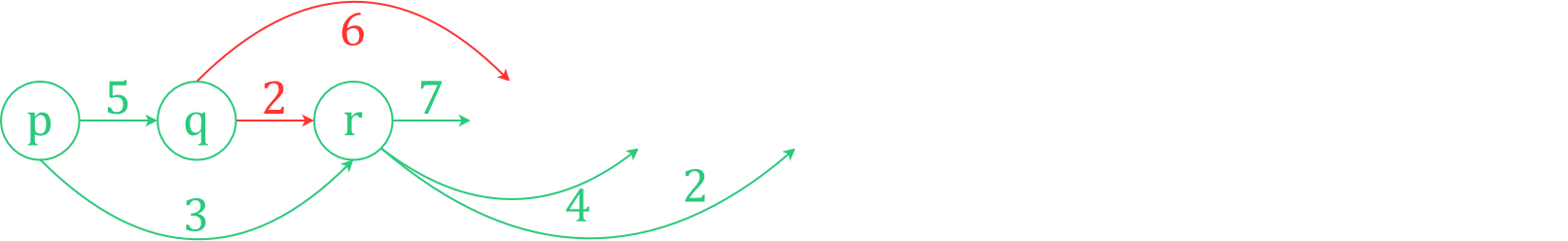


Applying topological sort, the order we will find for this graph is . Now to relax the edges. Say our source node is .

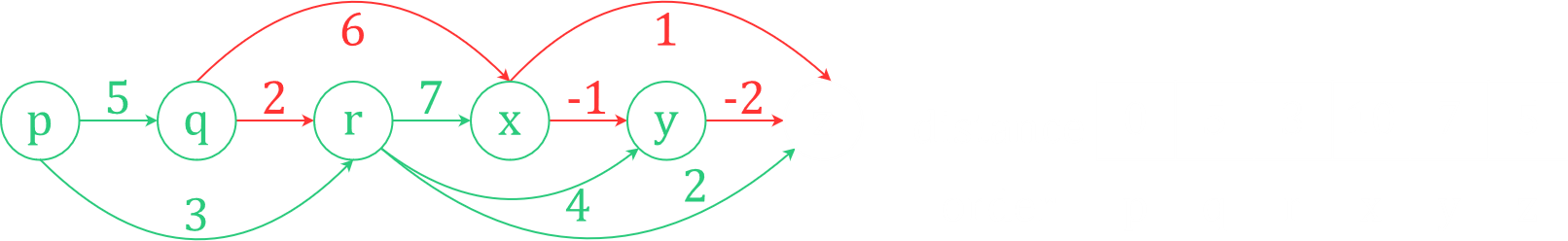


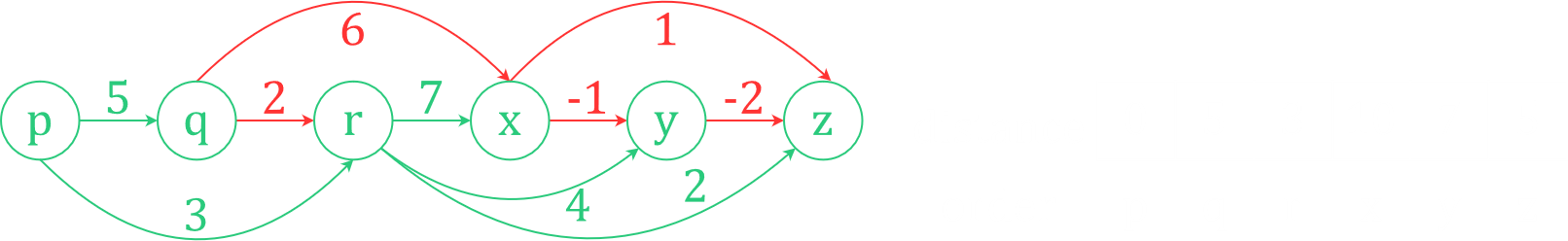












Thus, each edge is only being relaxed one time.

The time complexity of performing the topological sort is , and for relaxing the edges, we visit each vertex and each edge one time, so the time complexity is . Thus, the overall time complexity of the algorithm is .

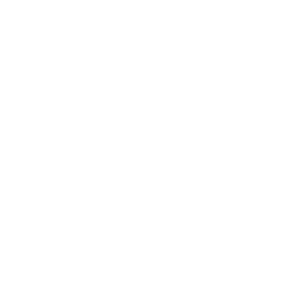
Note that the distances depend on which node we consider to be our source node when relaxing the edges. If we had chosen say to be the source, there would be no paths to and . Thus, those distances would have remained .

### Shortest Path in General Graphs

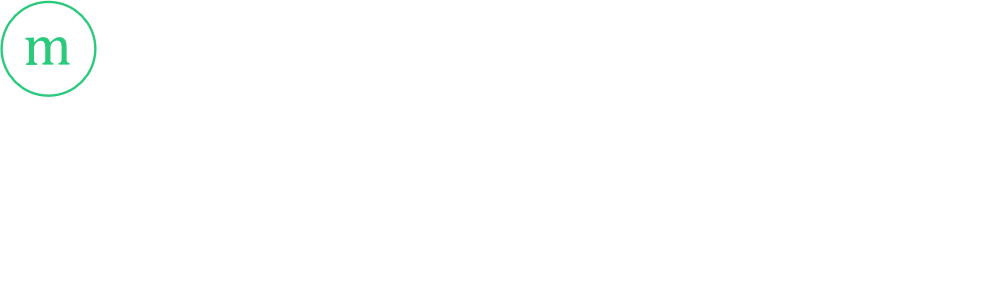
We have already studied one algorithm to find the shortest path, using BFS. However, that only works for unweighted graphs. We need to modify a few things so that it works with weighted graphs as well.

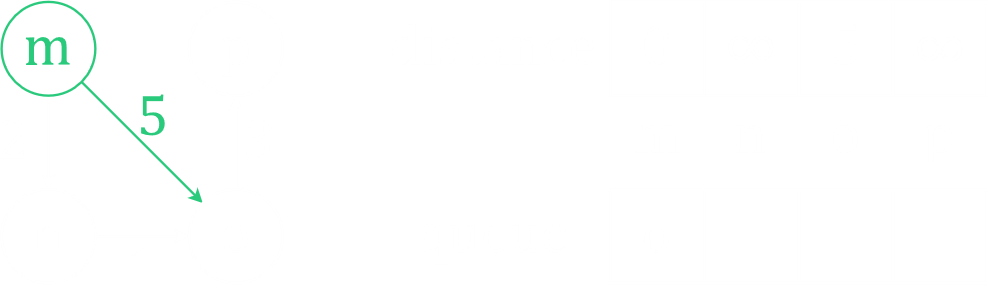
In BFS, we pushed a node into a queue if it was not visited. For weighted graphs, we will push the node into the queue if its incoming edge is relaxed. Thus, if we relax a path from to , is pushed into the queue. The reason for this is all the paths related to might now change as well due to the edge relaxation.

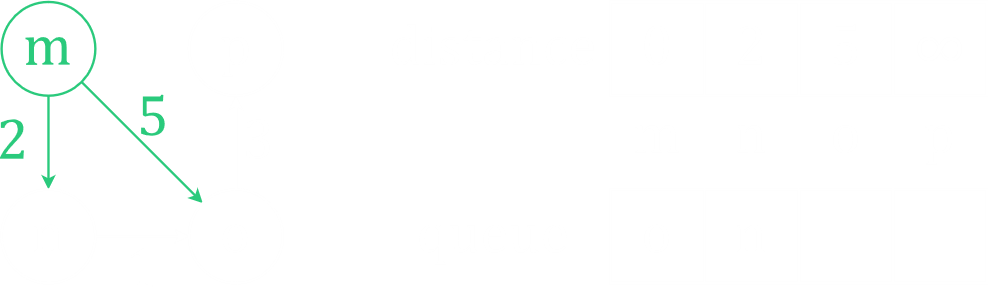
Consider the graph below:

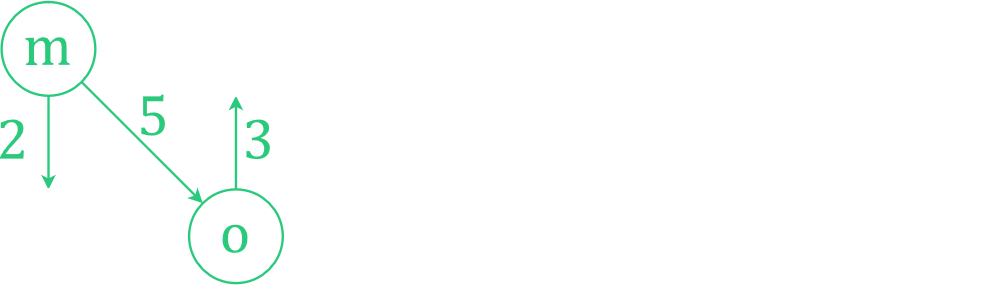


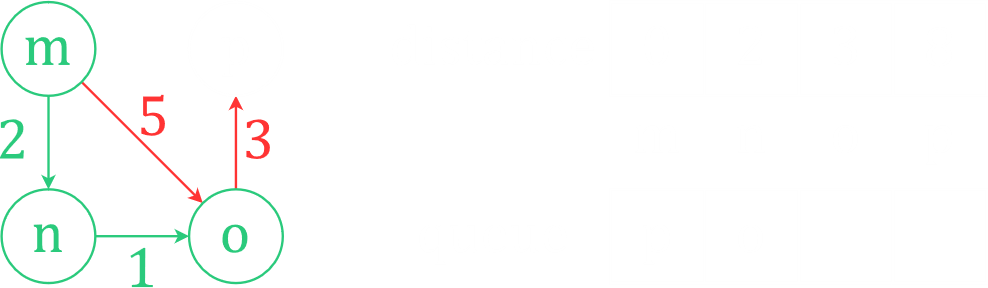
Say is the source node. In the traversal, we will make an assumption. If we have a choice between and , we will choose . This assumption will be explained in a moment, but for now, let us continue.



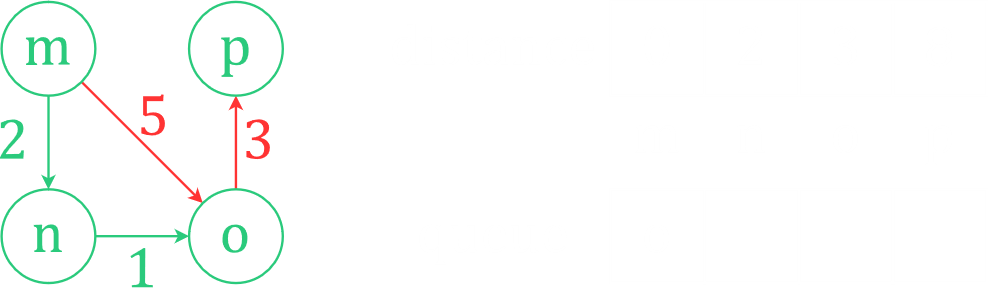


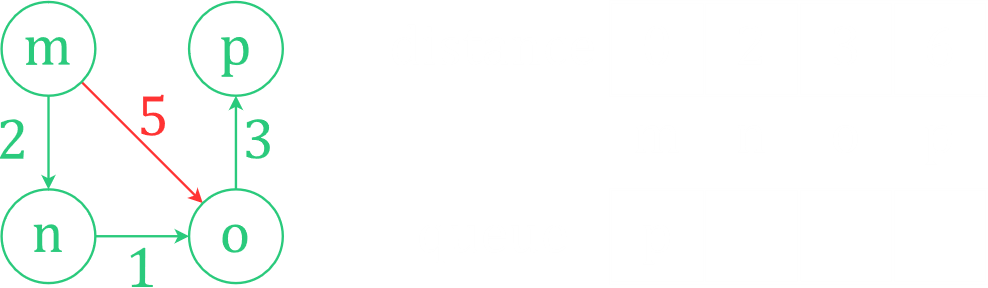


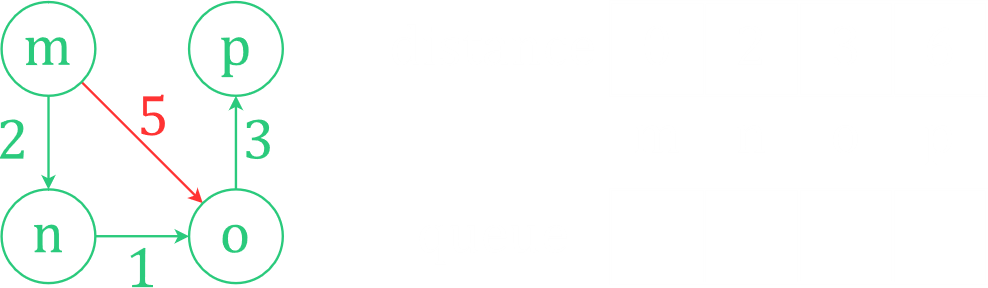




Notice that we found a smaller distance for and thus had to push it onto the queue again. will change as a result of this new finding, and putting back into the queue will allow us to make this change.





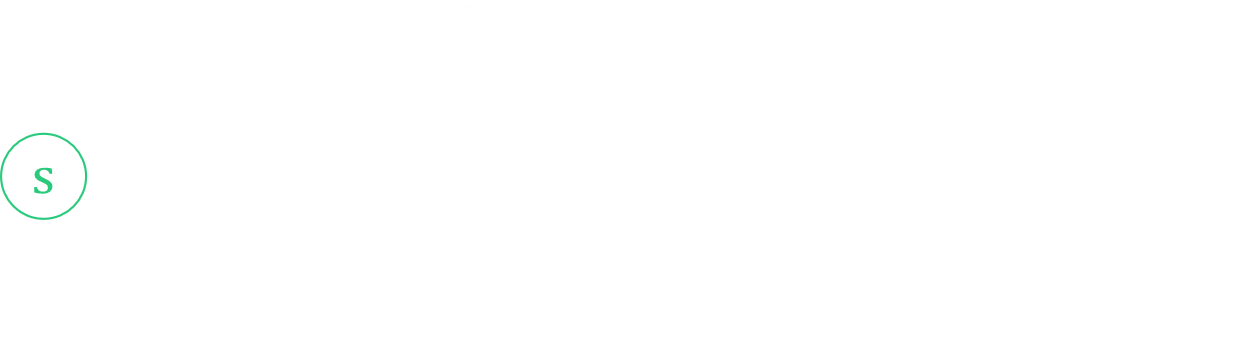


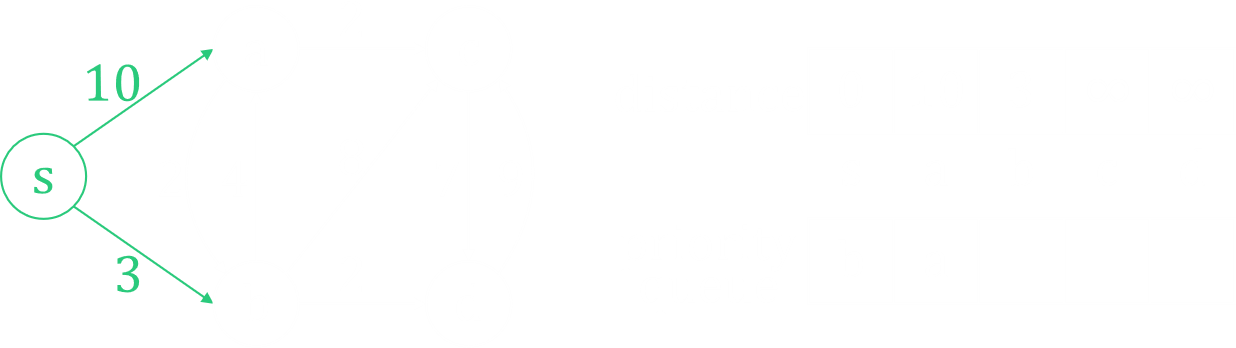
Consider what would have happened had we gone to before at the beginning. We would have gotten the exact same results. There would just be an odd stage in between where we would have pushed twice consecutively and then twice consecutively. The question is, how should we know to pick instead of first? The rule is to choose the node from the queue with the shortest distance from the source. This will ensure we have the least number of steps. We do this by using a priority queue instead of a normal queue.

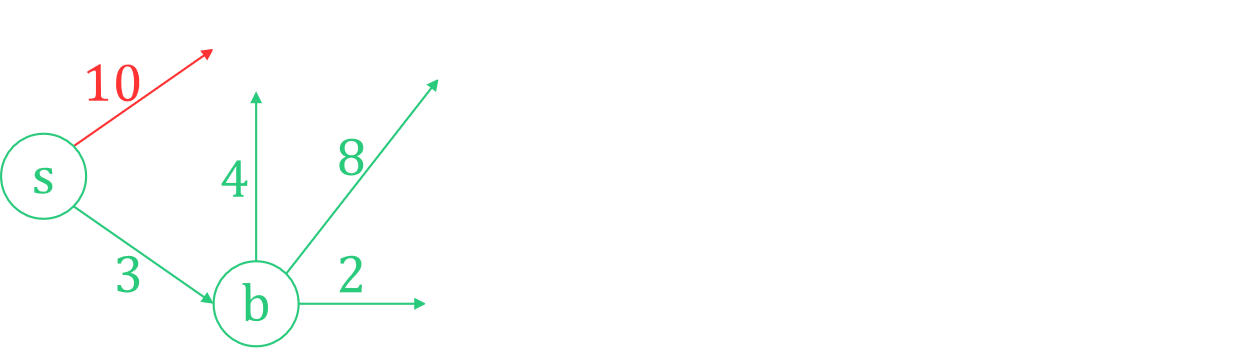
The way the STL priority queues work, the actual integer value of and would be pushed, meaning when we pop the versions of and pushed earlier, their values would not be equal to the smallest distance values of and stored, since the smallest distance values changed after they had been pushed. If we just check for this inequality when we pop them, we can avoid processing them. If we do that, we would need to process just 4 nodes (including the source node), whereas in the first case we were forced to process 6 nodes, since at no point did we have the same node in the queue twice simultaneously. This would be a boost to the time complexity.

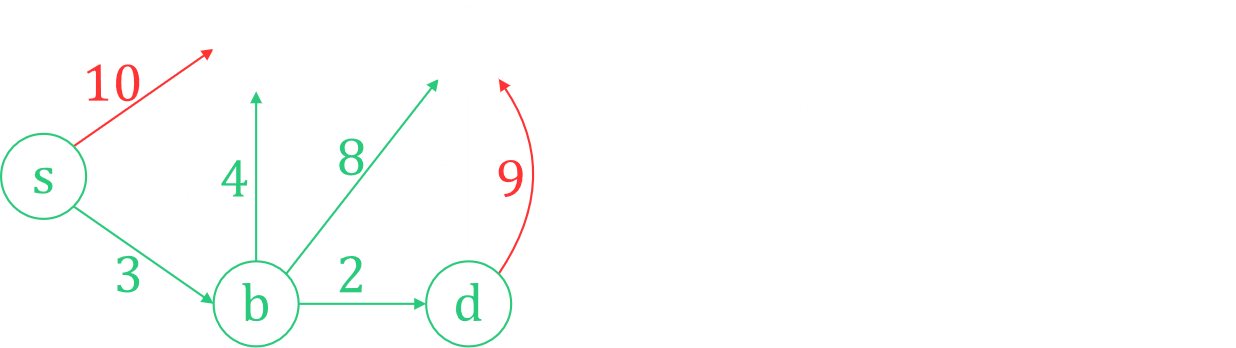
This is Dijkstra’s algorithm.

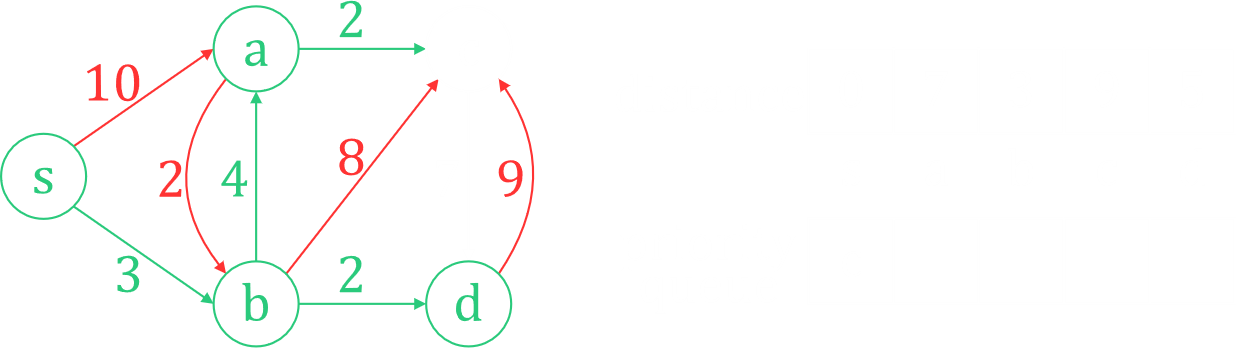
Let’s look at another example.

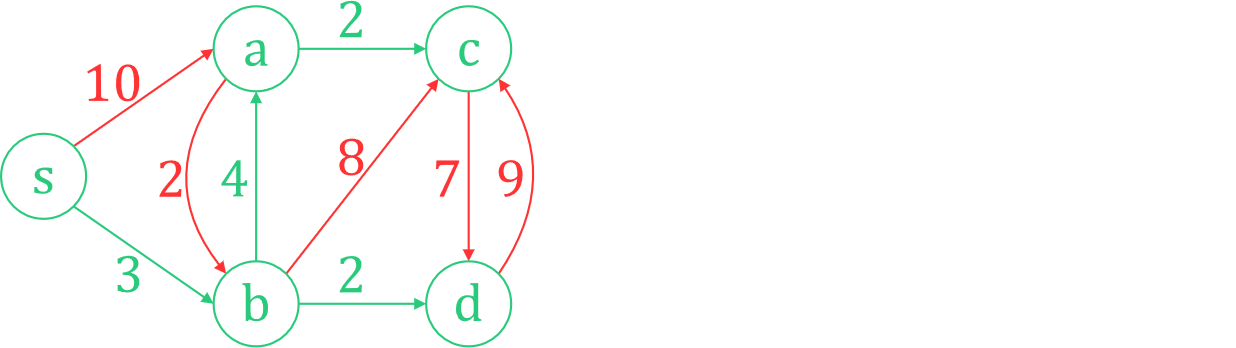












#### Time Complexity

We implement priority queues using heaps, more specifically Fibonacci heaps, but we do not need to know about that.

We are inserting nodes once each, with each insertion having a time complexity of .

We are extracting the minimum value times, once per node. Each of these has a time complexity of .

Each edge is also relaxed only one time, each having a time complexity of .

Thus, the total time complexity if .

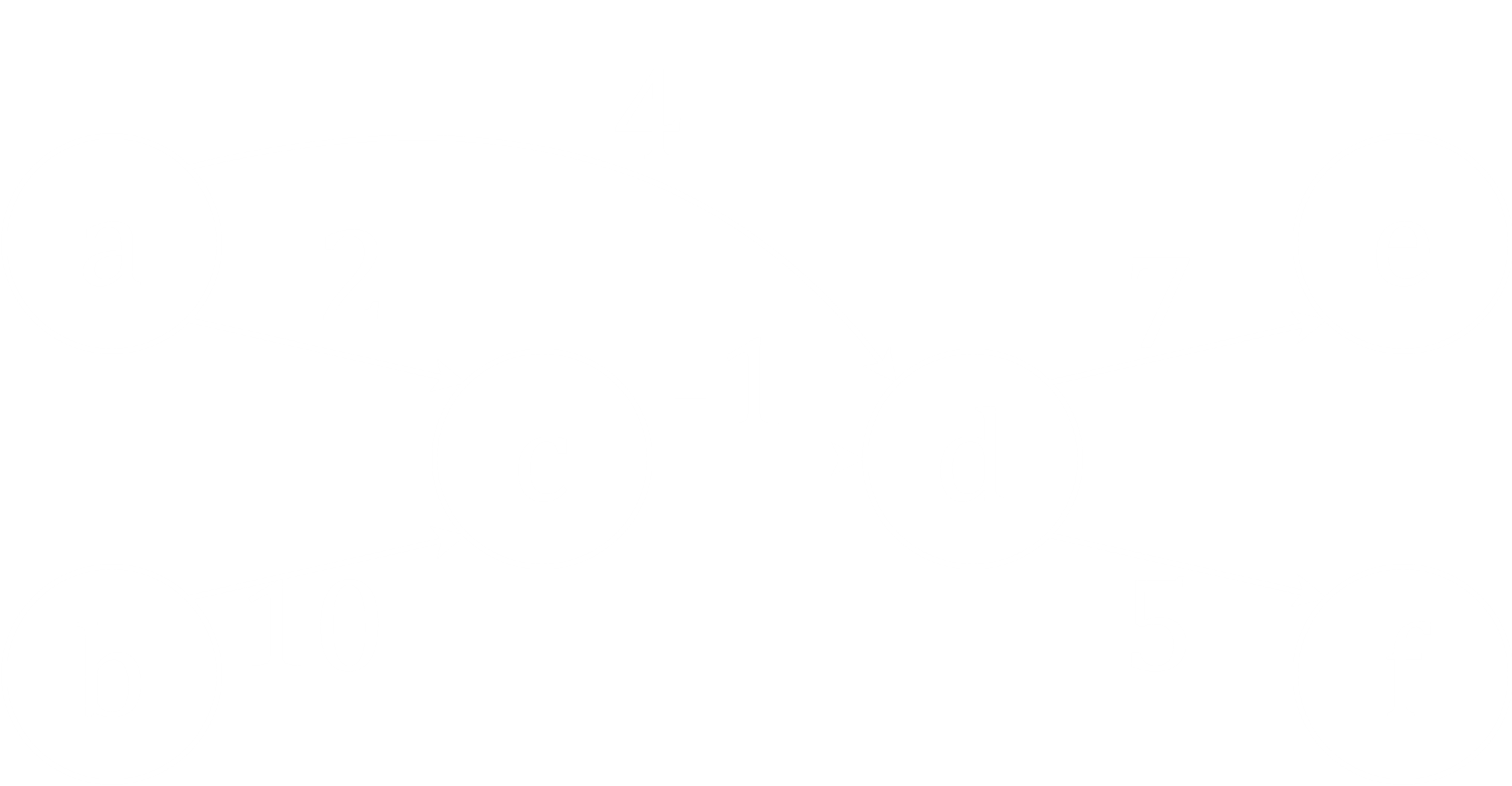
## Bellman-Ford Algorithm

A major problem with Dijkstra’s algorithm is that it does not work with negative weight cycles. Specifically, if we try to use it, the algorithm goes into an infinite loop. The Bellman-Ford algorithm can help here. Using this algorithm, we can work with graphs with negative weights, and we can detect if there is a negative weight cycle.

### Algorithm Demonstration

The basic idea behind the algorithm is to go through all the edges of the graph and relax them. We can do this in absolutely any order, and the order that is commonly followed is the order in which the input for the edges was taken from the user. This process of relaxing all the edges is repeated times. We shall discuss why this is done a little later.

Consider the graph below:



Say the order in which we got the edges is , , , , . This is the order in which we will be relaxing the edges. It is completely random.

Say our source node is .



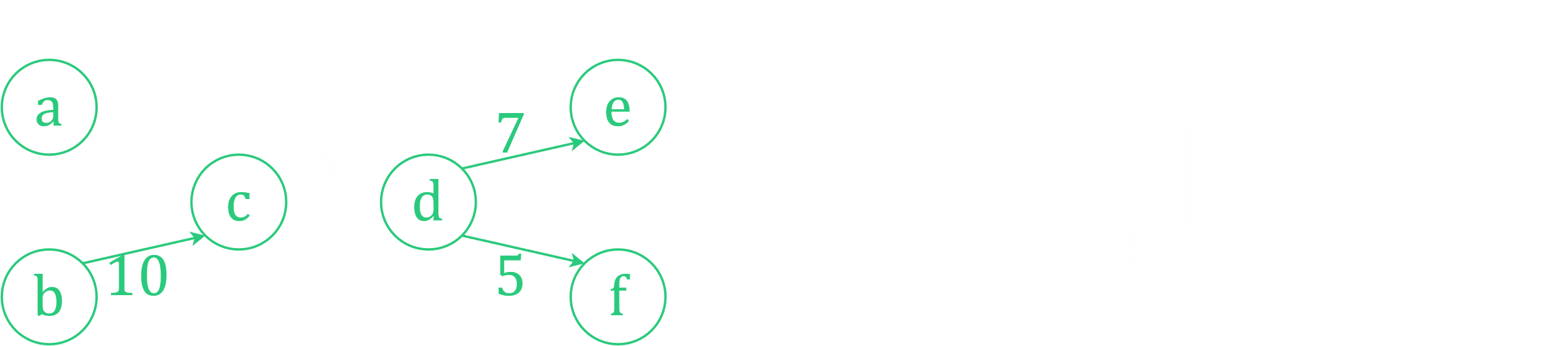
First, we try to relax . We get .

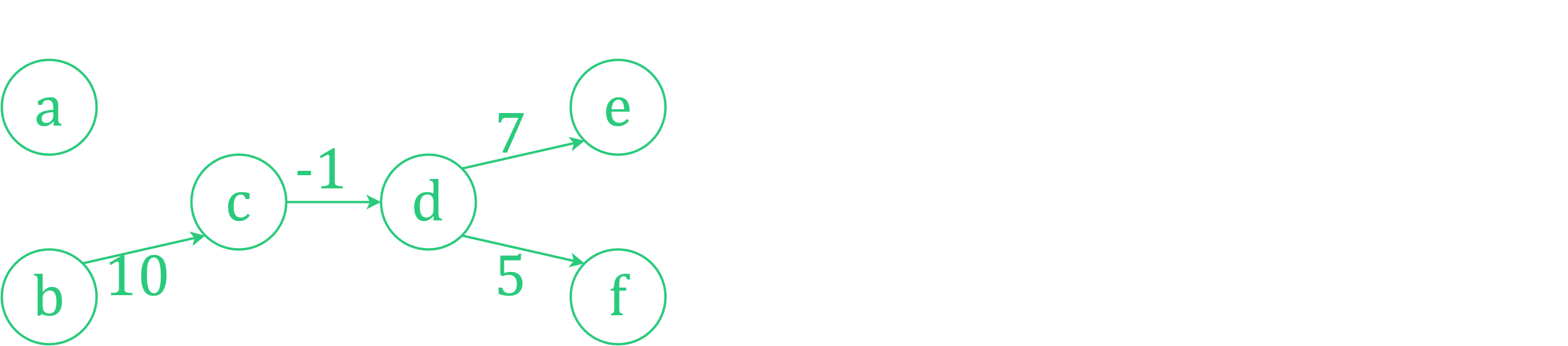


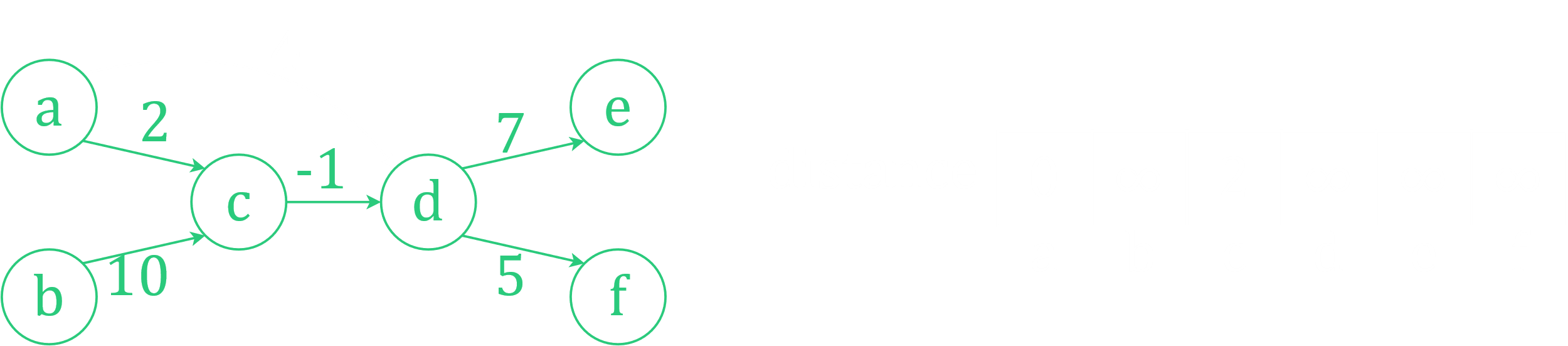
Next, we try to relax . We get .



We continue in a similar manner.





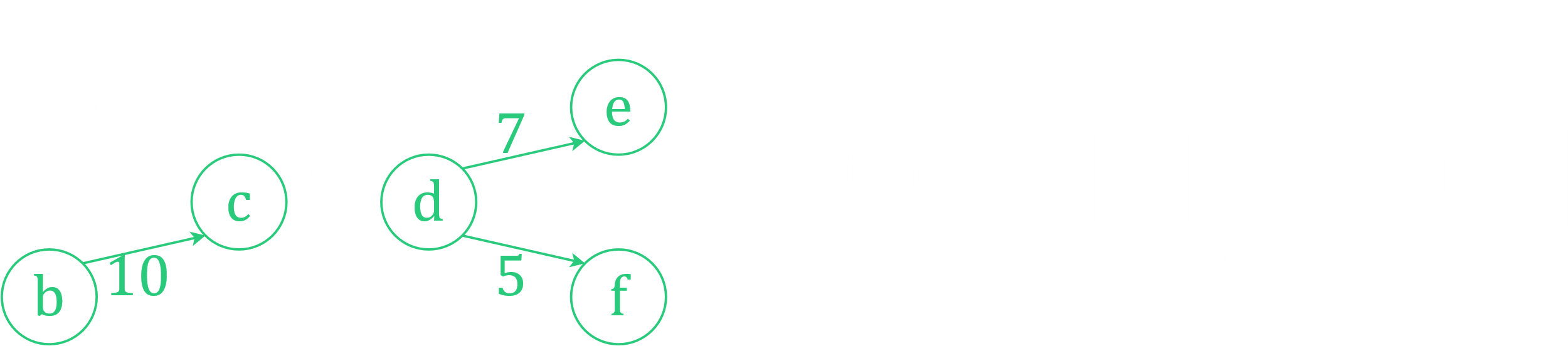




Our first pass over all the edges is complete. The only changes we saw were for the distances of and , for the edges and . All the other edges were traversed before these changes could take place, so their effects on the other edges were not felt. To see these effects, we need to pass over all the edges again.

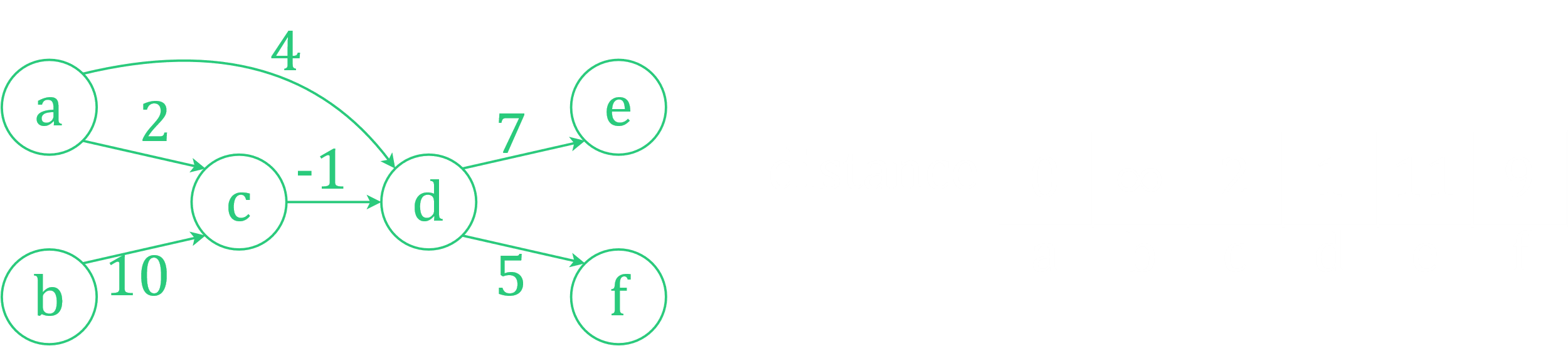








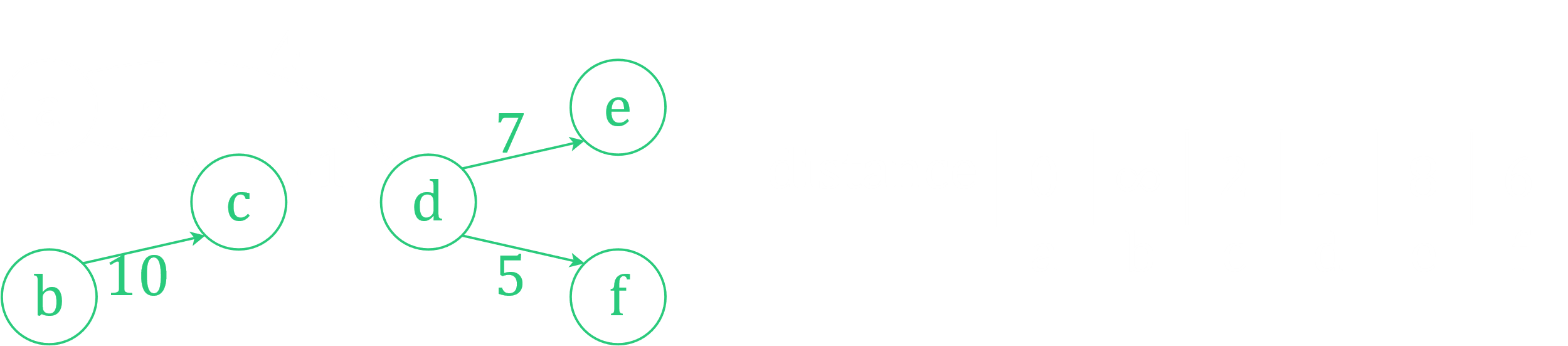


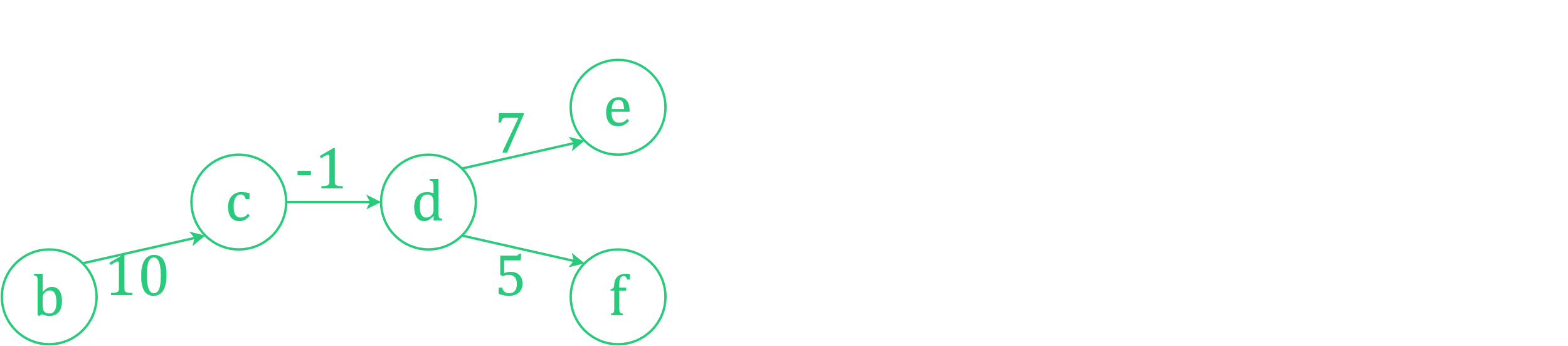


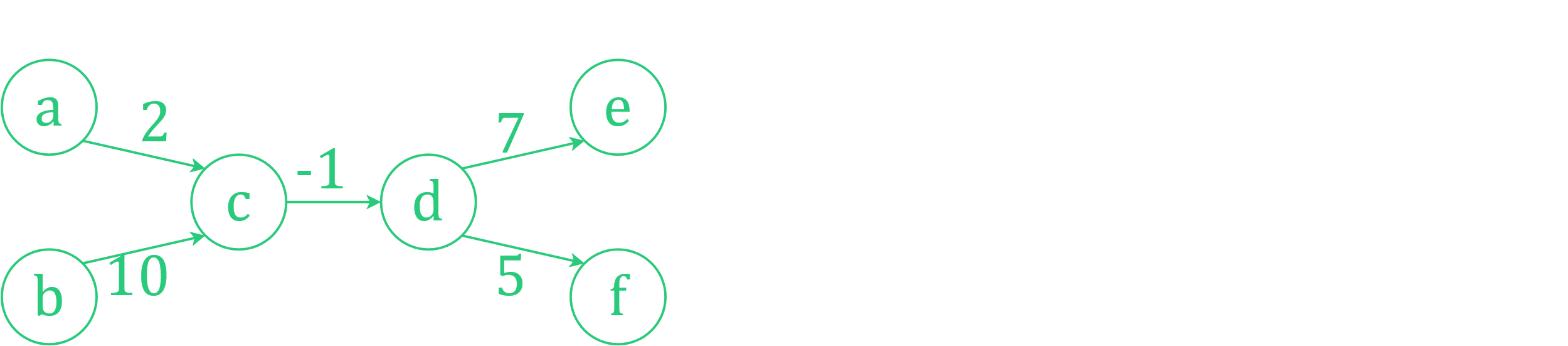
We have completed 2 passes over all the edges now. There were several more changes, but this is still not the solution. Notice that we now have the shortest paths from to and to , but not from to or from to . We need to keep going.













We have completed 3 passes, and we have all the shortest paths from to each of the other vertices. Since there are vertices in total, the actual program would still perform more passes, but there would be no further changes, so those are not being shown here.

Notice that in the first pass, we found the shortest paths to all vertices that are 1 edge away from the source node. In the second pass, we found the shortest paths to all vertices that are 2 edges away from the source node. In the third pass, we found the shortest paths to all vertices that are 3 edges away from the source node. In our example, that is the farthest it goes, which is why there are no more changes after the third pass. If there were nodes even further away, more passes would be needed. This is where the passes plays its part.

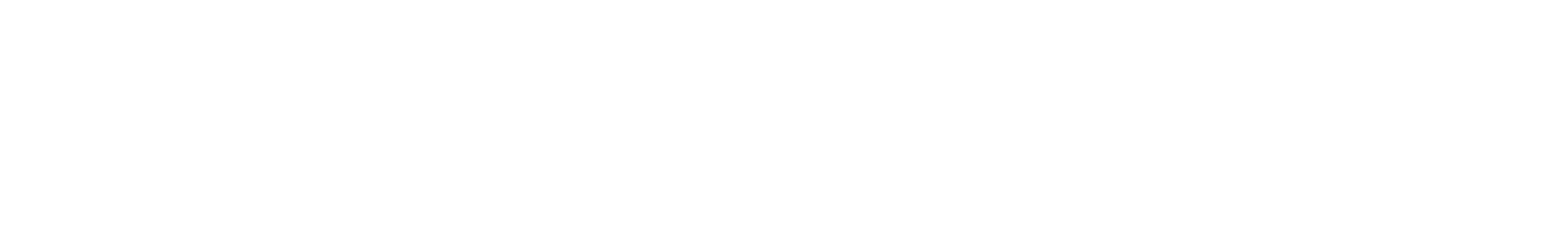
### Time Complexity

We performed passes, and in each pass, we relaxed edges. Thus, the time complexity if .

### Theory Explanation

The first question that arises is, why are we making passes? How do we know for certain that we will only need passes? We have already seen that each pass solves the problem for one extra level, so if we perform passes at most, we will get to the ()-th level. In the worst case, that is as large as the graph can get.

Consider this graph:



There are 4 vertices, and since it is just one straight line, there are levels.

The formal theorem for this is as follows. If contains no negative weight cycles, then after Bellman Ford executes, for all . Since we are trying to find the shortest path, we will use the minimum possible number of edges, avoiding paths with more edges. Due to there not being any negative weight cycles, the path will be simple. In the worst case, if we have vertices, we will need to traverse edges to get from the source to the vertex farthest from it.

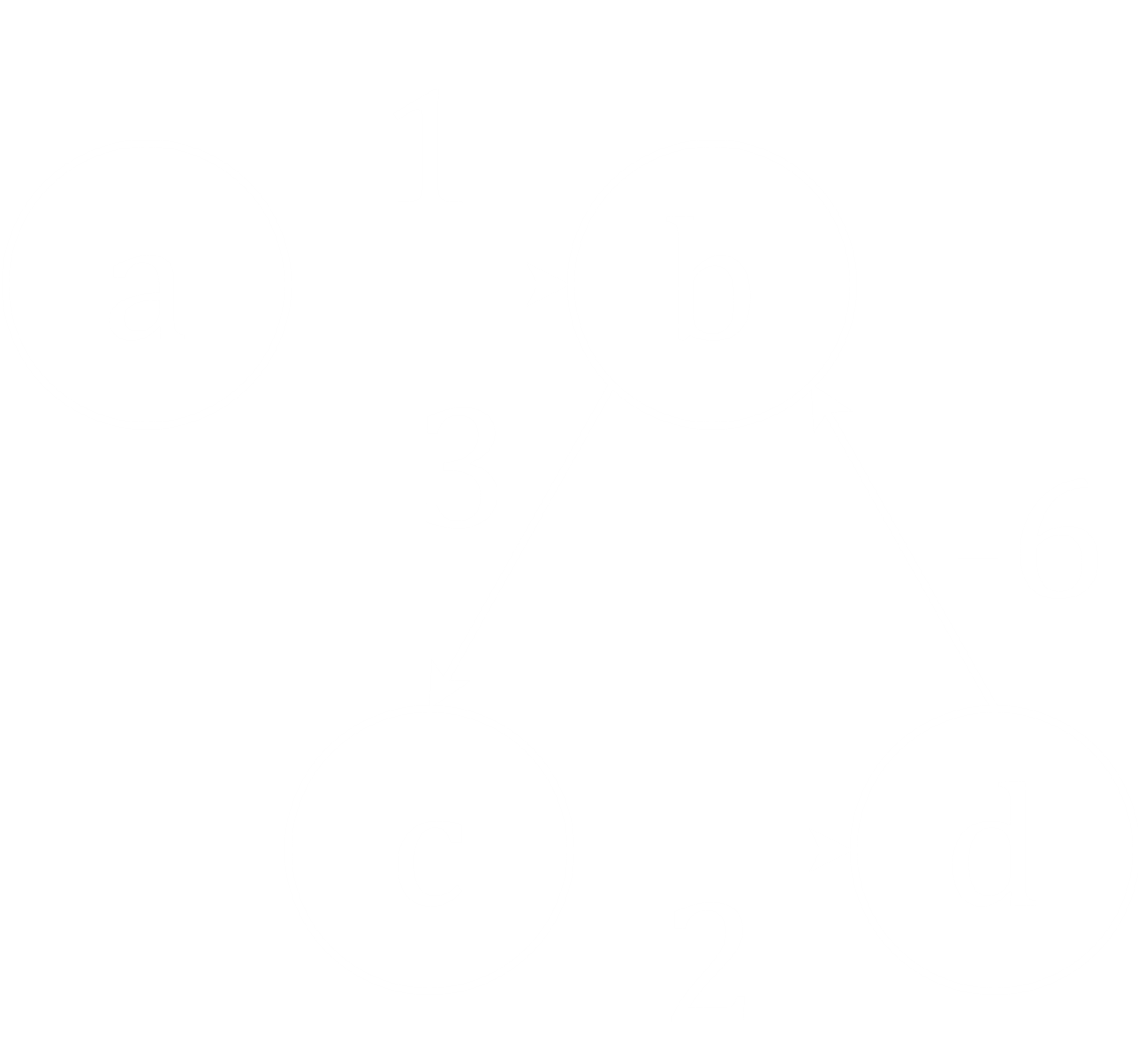
Initially, , since is the source node. Regardless of how we interact with the edges, after the first pass, we will have relaxed all edges directly connected to the source vertex. Thus, we have . After the second pass, we have and so on. Since, in the worst case, the last vertex is edges away from the source node, we will need passes to obtain .

### Negative Weight Cycles

The second ability the Bellman-Ford algorithm gives us is the ability to detect a negative weight cycle.

In the algorithm, we have seen that we will perform passes. However, what if we perform one more pass? In the example we saw above, where there was no negative weight cycle, this would have made no difference, since all possible shortest paths would have been discovered by the ()-th pass. One extra pass however, can allow us to detect a negative weight cycle.

Consider this graph:



After iterations, the distances of each of the nodes are , , and respectively. However, there will be no connection to , the source node.

Notice why this happens. Inside the loop, from to to and back to , we have a negative value, . This means the ‘distance’ of each of those nodes will start decreasing by each time. If the value of becomes less than , will recognize as its parent, thus losing the connection with . This will happen regardless of the order of the edge relaxations. If this is confusing, try to actually perform the algorithm.

We have said previously that after iterations, there is no possibility of any further edge relaxations. Nothing more will change. However, in the case where there is a negative weight cycle in the graph, if we keep performing iterations, the values will keep decreasing. This is how we can detect the negative weight cycle. After performing the iterations for the actual algorithm, we will perform one more iteration. If there are changes to the distance values, then there is a negative weight cycle.

Note that the intent of the Bellman-Ford algorithm is not to identify which edges are creating the negative weight cycle, but to simply detect that such a cycle exists. The main goal of the algorithm is still to find the shortest path from a source node to every other node in the graph. If we wish to find which edges in particular make the cycle, there are easier algorithms to do this. Performing a simple DFS on the graph would have allowed us to detect back-edges, and thus cycles.

### Proof

We previously saw a theorem related to how passes are required at most to relax all the edges. Now consider the corollary, if a value fails to converge after passes, there exists a negative-weight cycle reachable from .

If does not converge, the path from to cannot be simple, following from the original theorem. This means there have to be some repeated vertices that cause the cycle.

If the cycle were a positive weight cycle, the weight would be increasing. However, that is not what the edge relaxation process is concerned with, since it is looking for decreased weights. As such, a positive weight cycle would not have affected the algorithm. Since we have a cycle that is causing our cost to decrease, it must be a negative weight cycle.

For negative weight cycles, we say the shortest path is undefined. This is because the Bellman-Ford algorithm cannot actually find the edges of the cycle itself.

Problems that ask us to find the shortest simple path in a graph containing a negative weight cycle are called NP Hard problems. There are no polynomial-time solutions that have ever been discovered for such problems. All solutions found so far in the world are exponential.