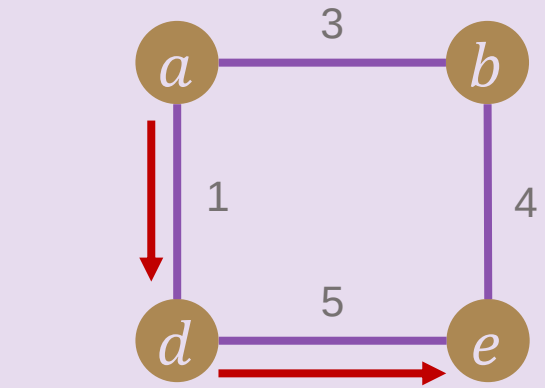
GRAPH THEORY 2

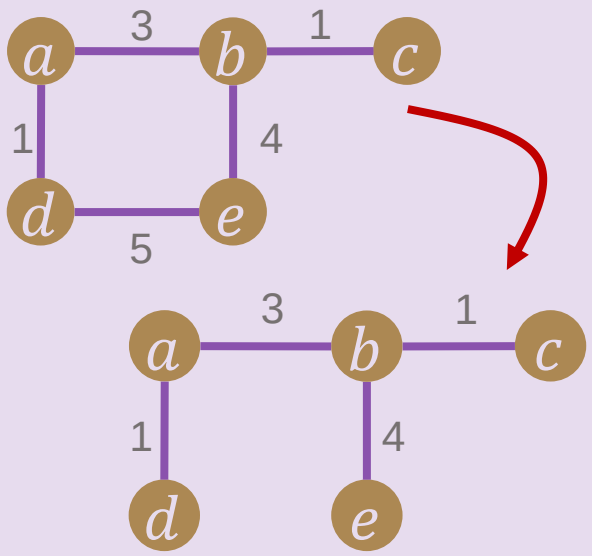
Hadrian Ang, Kyle See, April 2017

What is a shortest path?

Given a graph G, a source vertex u in G, and a destination vertex v in G, a shortest path from u to v is a path in G from u to v such that the total of the weights of all edges in the path is minimized. If G is unweighted, minimize the number of edges in the path (BFS).



$SP(a, e)$



What is a minimum cost spanning tree?

Given a graph G, a spanning tree of G is a connected subgraph of G that is a tree and contains all of its vertices. A minimum cost spanning tree of G is a spanning tree with the minimum possible total weight of all edges included in it. All spanning trees in an unweighted graph are considered minimum cost spanning trees.

Determining Shortest Paths and Minimum Cost Spanning Trees

There are standard algorithms used to determine the shortest paths and minimum cost spanning trees within graphs.

A

Single Source Shortest Path (SSSP)

- Dijkstra's Algorithm
- Bellman-Ford Algorithm

All Pairs Shortest Path (APSP)

- Floyd-Warshall Algorithm
- SSSP from each source

B

Minimum Cost Spanning Tree (MCST)

- Prim's Algorithm
- Kruskal's Algorithm
- Boruvka's Algorithm (not covered)

Notes on Shortest Path Problems

There are two typical types of shortest path problems:

- Single Source Shortest Path (SSSP) – look for the shortest path from one given vertex to any or all other vertices in the graph.
- All Pairs Shortest Path (APSP) – look for the shortest path between multiple pairs of vertices in the graph.

What algorithms to use depends on the nature of the given problem.

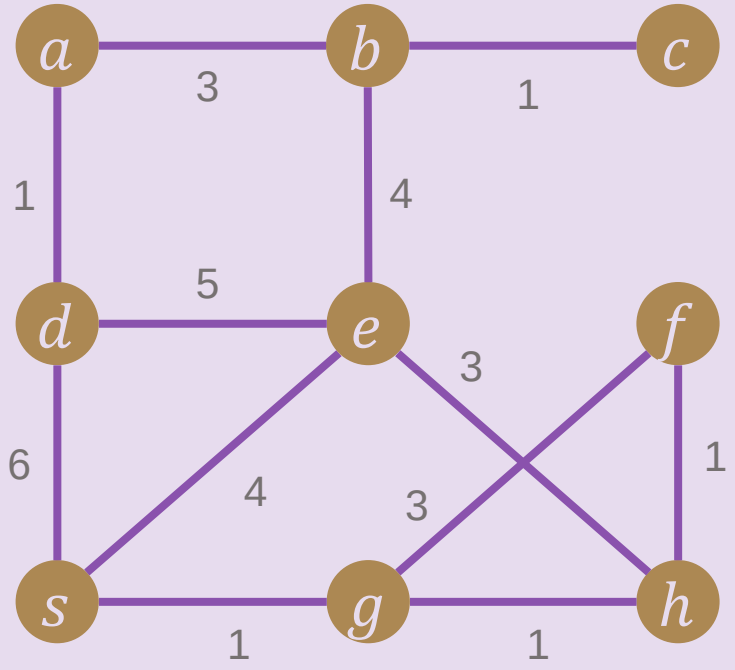
Most shortest path problems have cases where the end vertex is not reachable from the start vertex. In these cases, typically there is a default “not found” output like printing a distance of -1. All the algorithms we will discuss cover these cases by simply leaving the unreachable vertex unprocessed or never changing the initial sentinel value assigned to its distance.

Some shortest path problems require you to print the actual shortest path instead of just the distance between the vertices. All the algorithms we will discuss will have some way of “updating” current knowledge on the distances of each vertex. These are typically matched with updating a “parent” variable to allow us to trace back the path we actually took. More on this in the implementation of each individual algorithm.

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

Dijkstra's Algorithm begins with the single source vertex s having a known distance of 0 from itself and all other vertices having an unknown distance, typically labeled infinity, from s.

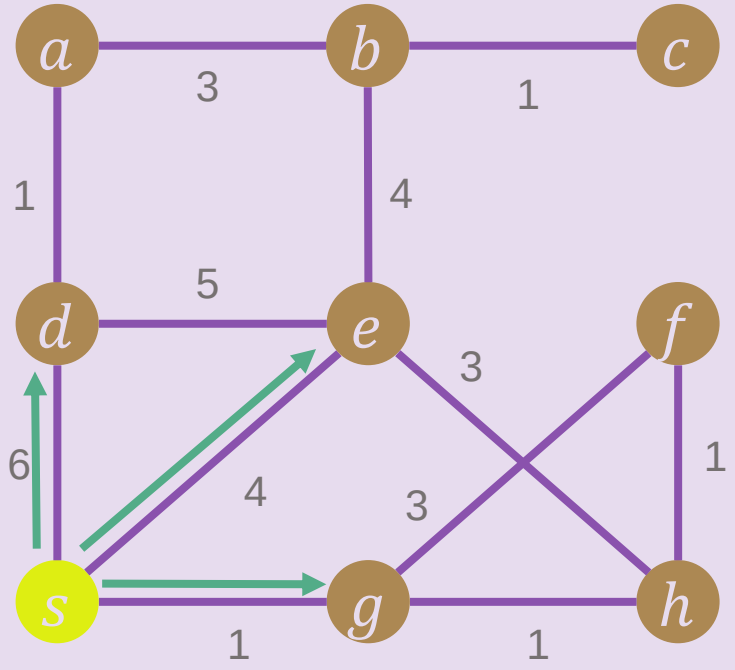


s	0	
a	∞	
b	∞	
c	∞	
d	∞	
e	∞	
f	∞	
g	∞	
h	∞	

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

The algorithm first "visits" or processes s. The algorithm determines the distance of each vertex v adjacent to s if the path consists of the edge (s, v).

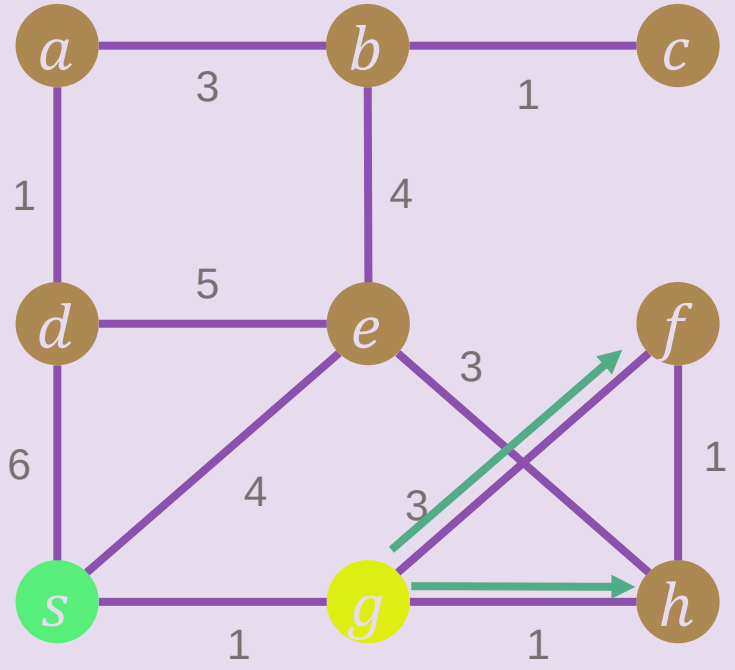


s	0	
a	∞	
b	∞	
c	∞	
d	6	
e	4	
f	∞	
g	1	
h	∞	

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

The algorithm then "visits" or processes the vertex closest to s. When visiting a vertex u, the algorithm determines the distance of each vertex v adjacent to u from s if the path includes the edge (u, v). If it is shorter than the currently known distance, replace it.

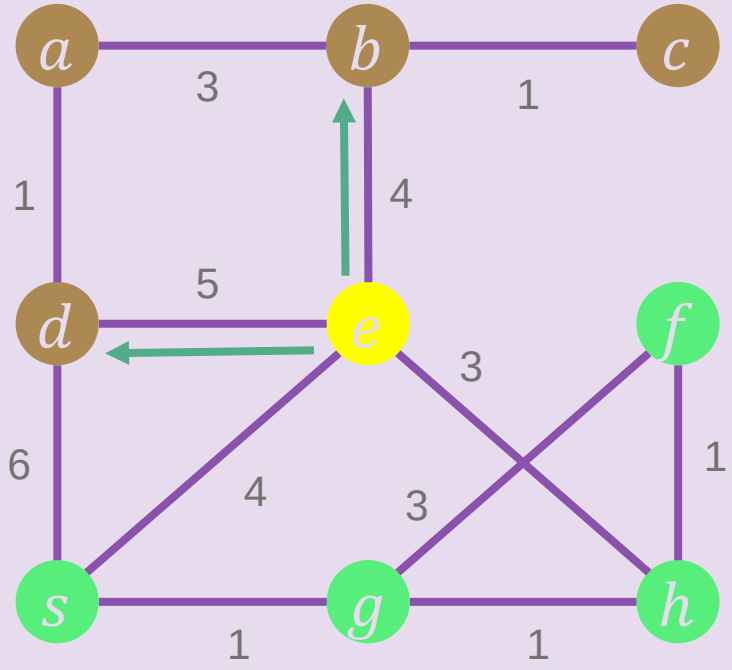


s	0	✓
a	∞	
b	∞	
c	∞	
d	6	
e	4	
f	4	
g	1	
h	2	

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

The algorithm continues visiting the vertices in order of their known distances from s, making sure not to repeat vertices already visited beforehand.

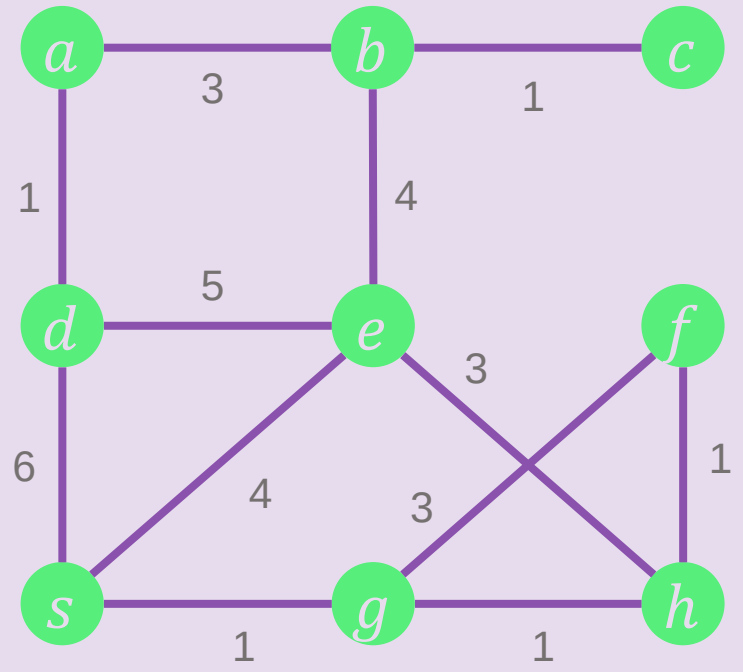


s	0	✓
a	∞	
b	8	
c	∞	
d	6	
e	4	
f	3	✓
g	1	✓
h	2	✓

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

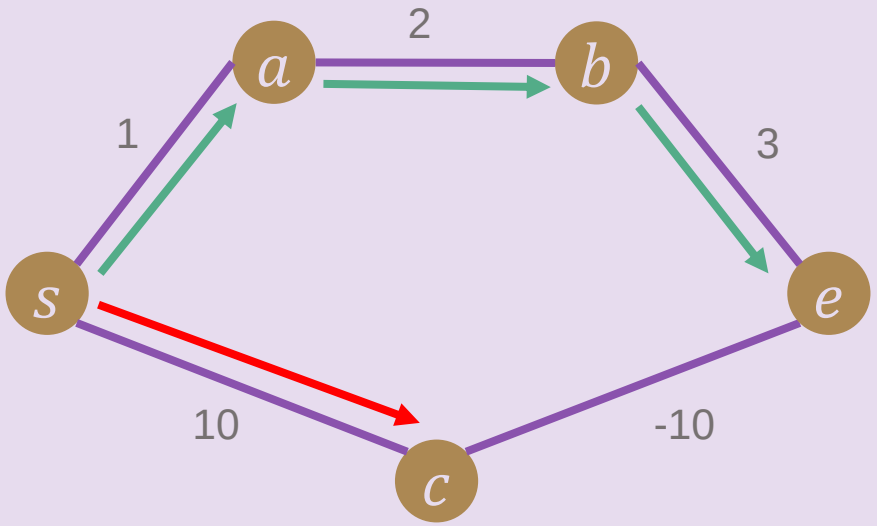
The algorithm ends when the destination vertex is reached or all vertices have been visited.



s	0	✓
a	7	✓
b	8	✓
c	9	✓
d	6	✓
e	4	✓
f	3	✓
g	1	✓
h	2	✓

Dijkstra's Algorithm

Dijkstra's Algorithm works on the idea that, since we visit vertices in order of their distance from the source vertex, it is impossible for us to find a shorter path to some vertex we have previously visited that visits the vertex we are currently visiting. Because of this, when we visit a vertex for the first time, we are guaranteed to have found a shortest path to it already. Note that because of this, Dijkstra's Algorithm does not work for graphs with negative edge weights.



Dijkstra's Algorithm Sample Implementation ($O(V^2 + E)$)

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices for that test case.
//In this sample, our source vertex is 0.
bool vis[N]; int dist[N]; vector<int> adj[N], adjw[N];

int main(){
    //read graph into adj, adjw
    //set vis[0]..vis[n-1] to false
    //set dist[1]..dist[n-1] to inf or -1 (sentinel value)
    dist[0] = 0;
    while(true){
        int next = -1;
        for(int i=0; i<n; i++){
            //add extra check if sentinel is -1
            if(!vis[i] && (next == -1 || dist[i] < dist[next]))
                next = i;
        }
        if(next == -1) break; //no more unvisited vertices
        vis[next] = true;
        for(int i=0; i<adj[next].size(); i++){
            if(vis[adj[next][i]]) continue;
            // or if dist[adj[next][i]] == -1 if sentinel is -1
            if(dist[next] + adjw[next][i] < dist[adj[next][i]]){
                dist[adj[next][i]] = dist[next] + adjw[next][i];
            }
        }
    }
    //dist[u] will contain the distance from 0 to u
}
```

```
//For constructing the path itself,
//we add a parent variable to each vertex.
//This acts like the "previous" vertex in the path
int parent[N];

//By default, the parents do not exist,
//so we set them to some sentinel value

//set parent[0]..parent[n-1] to -1

//Whenever we update the distance of a vertex,
//we know that its shortest path will contain that
//edge and the current vertex being processed is
//the previous vertex in that path.

parent[adj[next][i]] = next;

//Reconstruct the path by following each vertex's
//parent until we return to the source.

vector<int> path;
int cur = end;
while(cur != source){
    path.push_back(cur);
    cur = parent[cur];
}

//path will contain the actual path in reverse.
```

Dijkstra's Algorithm Sample Implementation ($O(V \log E + E)$)

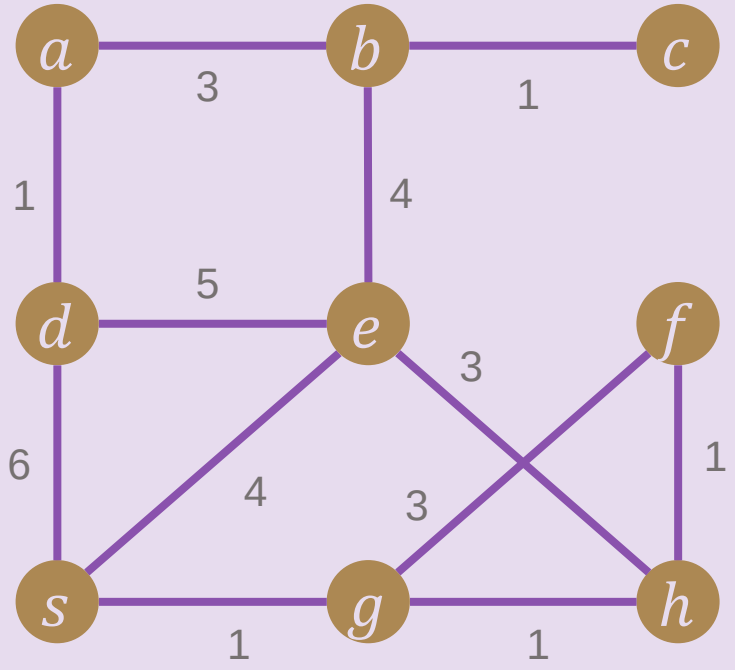
Dijkstra's Algorithm can be sped up by using a priority queue to find the next closest vertex to the source.

```
//define a new comparator for the priority queue
struct cmp{
    bool operator()(int a, int b){
        return dist[a] > dist[b]; //get the smallest distance first
    }
};

//after setting all the starting values
priority_queue<int, vector<int>, cmp> pq;
pq.push(0); //we start with the source vertex
while(pq.size() > 0){ //instead of while(true), we only need to check if the priority queue is nonempty
    int next = pq.top(); pq.pop(); //instead of searching, we can just get the next element in pq
    if(vis[next]) continue; //the same vertices will appear multiple times
    vis[next] = true;

    //process as before but push the new vertices into the priority queue
    for(int i=0; i<adj[next].size(); i++){
        if(vis[adj[next][i]]) continue;
        if(dist[next] + adjw[next][i] < dist[adj[next][i]]){
            dist[adj[next][i]] = dist[next] + adjw[next][i];
            pq.push(adj[next][i]);
        }
    }
}
```

Bellman-Ford Algorithm

The Bellman-Ford Algorithm also finds the shortest path from some given vertex to all other vertices. It is slower than Dijkstra's algorithm, making it less commonly used. However, it covers graphs with negative weight edges.

Like Dijkstra's Algorithm, the Bellman-Ford Algorithm also begins with the single source vertex s having a known distance of 0 from itself and all other vertices having an unknown distance, usually labeled infinity, from s.

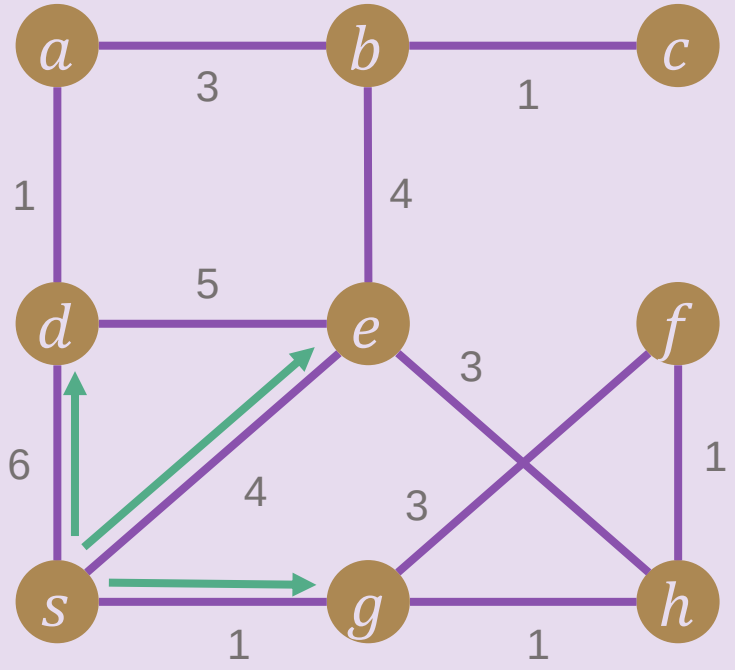


s	0
a	∞
b	∞
c	∞
d	∞
e	∞
f	∞
g	∞
h	∞

Bellman-Ford Algorithm

The Bellman-Ford Algorithm also finds the shortest path from some given vertex to all other vertices. It is slower than Dijkstra's algorithm, making it less commonly used. However, it covers graphs with negative weight edges.

It then iterates through every edge in the graph to determine if the known distance from s to the adjacent node can be "relaxed" or reduced.

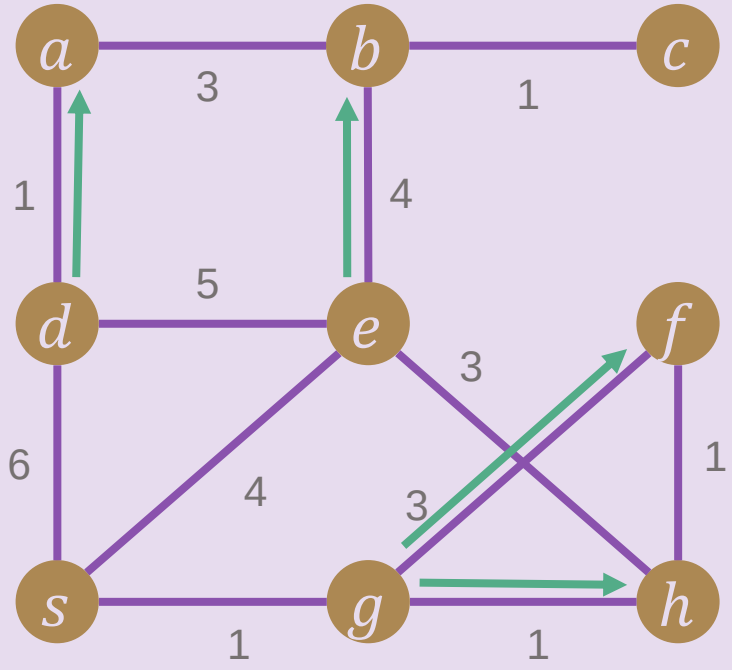


s	0	0
a	∞	∞
b	∞	∞
c	∞	∞
d	∞	6
e	∞	4
f	∞	∞
g	∞	1
h	∞	∞

Bellman-Ford Algorithm

The Bellman-Ford Algorithm also finds the shortest path from some given vertex to all other vertices. It is slower than Dijkstra's algorithm, making it less commonly used. However, it covers graphs with negative weight edges.

This is repeated multiple times. During each iteration, vertices have their distances from s reduced, and so are able to relax the vertices adjacent to them on the succeeding iterations.

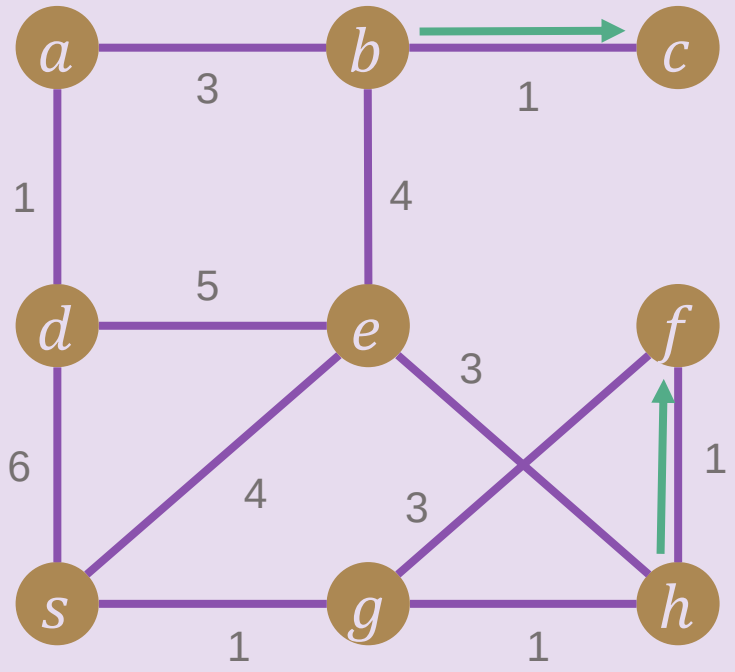


s	0	0
a	∞	7
b	∞	8
c	∞	∞
d	6	6
e	4	4
f	∞	4
g	1	1
h	∞	2

Bellman-Ford Algorithm

The Bellman-Ford Algorithm also finds the shortest path from some given vertex to all other vertices. It is slower than Dijkstra's algorithm, making it less commonly used. However, it covers graphs with negative weight edges.

If there are no negative weight cycles (which means no vertex will have a distance of $-\infty$), this will continue until an iteration where no more distances can relaxed. Here, the shortest paths for each vertex has been found.



s	0	0
a	7	7
b	8	8
c	∞	9
d	6	6
e	4	4
f	4	3
g	1	1
h	2	2

Bellman-Ford Algorithm

The Bellman-Ford Algorithm works on a similar idea as Dijkstra's Algorithm. That is, the shortest path to some vertex can be found by repeatedly finding better solutions until such a solution can no longer be found. However, Dijkstra's algorithm assumes that adding a new edge to a path can only increase its length, while the Bellman-Ford Algorithm makes no assumption and instead processes every vertex and every edge on each iteration. This allows it to handle negative edge weights.

Assuming the non-existence of negative weight cycles, it can be proven that the Bellman-Ford Algorithm will take no more than $|V| - 1$ iterations of relaxation, where $|V|$ is the number of vertices in the graph. This is because the shortest path from s to any other vertex can take no more than $|V| - 1$ jumps. The algorithm however, will continue indefinitely if a negative weight cycle exists. Because of this, if a relaxation still occurs on the $|V|$'th iteration, it is guaranteed that the graph has a negative weight cycle, and the algorithm can safely be terminated.

Bellman-Ford Algorithm Sample Implementation ($O(VE)$)

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices in that test case.
//E is the maximum possible number of edges in the input.
//e is the number of edges in that test case.
//In this sample, our source vertex is 0.
int dist[N], a[E], b[E], w[E];

int main(){
    //read graph into a, b, w
    //set dist[1]..dist[n-1] to inf or -1 (sentinel value)
    dist[0] = 0;

    //run the relaxation n times
    for(int i=0; i<n; i++){
        bool relaxed = false;
        for(int j=0; j<e; j++){
            if(dist[a[j]]+w[j] < dist[b[j]]){
                dist[b[j]] = dist[a[j]]+w[j];
                relaxed = true;
            }
            //repeat with reversed a, b for undirected graphs
        }
        if(!relaxed) break; //no more newly relaxed vertices
        else if(i == n-1){
            //negative weight cycle exists
        }
    }
    //dist[u] will contain the distance from 0 to u
}
```

```
//For constructing the path itself,
//we add a parent variable to each vertex.
//This acts like the "previous" vertex in the path
int parent[N];

//By default, the parents do not exist,
//so we set them to some sentinel value

//set parent[0]..parent[n-1] to -1

//Whenever we update the distance of a vertex,
//we know that its shortest path will contain that
//edge and the current vertex being processed is
//the previous vertex in that path.

parent[adj[next][i]] = next;

//Reconstruct the path by following each vertex's
//parent until we return to the source.

vector<int> path;
int cur = end;
while(cur != source){
    path.push_back(cur);
    cur = parent[cur];
}

//path will contain the actual path in reverse.
```

On Reconstructing Negative Weight Cycles in Directed Graphs

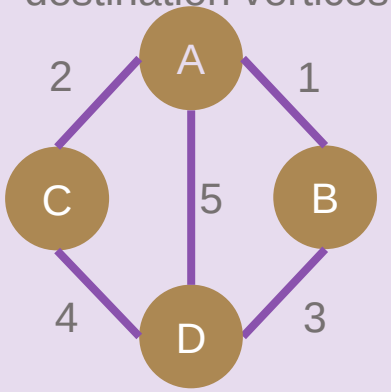
If a vertex u is relaxed during the nth iteration, it is necessarily part of some negative weight cycle. In undirected graphs, finding negative weight cycles is trivial because any negative weight edge necessarily forms a cycle with itself ($a \to b \to a$). However, this is not the case in directed graphs.

When using the method where we store the parents of each vertex, finding a cycle from u to itself may not necessarily work. The parent array will contain at least one negative weight cycle, but this is not guaranteed to be the negative weight cycle containing u. This is because parent entries may be rewritten for the same vertex multiple times by multiple different negative weight cycles in the same iteration.

When reconstructing a negative weight cycle, we have to check all the parent entries of all the vertices to find the cycle.

Floyd-Warshall Algorithm

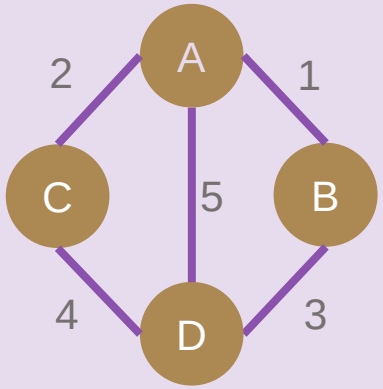
The **Floyd-Warshall Algorithm** solves the All-Pairs Shortest Path problem. In other words, it finds the shortest path between any two nodes in the graph. It does this by iterating through all vertices and checking if it can serve as an intermediate node to a shorter path between some other source and destination vertices.



	A	B	C	D
A	0	1	2	5
B	1	0	∞	3
C	2	∞	0	4
D	5	3	4	0

```
int dist[n][n];
for(int k=0; k<n; k++){
     for(int i=0; i<n; i++){
          for(int j=0; j<n; j++){
               if(dist[i][j] > dist[i][k] + dist[k][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
          }
     }
}
```

If the path passing through vertex k $(path(i,k) + path(k,j))$ is shorter than the direct path from i to j, then let us take the path through k instead of the direct one.



	A	B	C	D
A	0	1	2	4
B	1	0	3	3
C	2	3	0	4
D	4	3	4	0

Let $k = \mathrm{B}$, $i = A$, $j = D$. We know that $dist[A][B] = 1$ and $dist[B][D] = 3$. This means $dist[A][B] + dist[B][D] = 4$, which is shorter than $dist[A][D] = 5$. Thus the new value of $dist[A][D]$ is 4. We do this for all sources and destinations and then move on to the next intermediate node. When the algorithm ends, the adjacency matrix should contain the shortest path between any two nodes in the graph.

SSSP from each source

There are cases where the Floyd-Warshall Algorithm will not work for APSP. This is because either $O(V^3)$ is too slow or V^2 memory is too large.

To solve these problems, we can apply an SSSP algorithm from each starting vertex. Since there are V vertices in a given graph, Dijkstra's Algorithm will take $O(V^2 \log E + VE)$ time to complete. The actual time it takes for Dijkstra's Algorithm to complete will also be significantly reduced when the graph itself is sparse (doesn't have many edges) or disconnected, while the Floyd-Warshall Algorithm will take the same amount of time. It is also typically not necessary to find all pairs of shortest paths, but only a large number of them. If a vertex is never the starting vertex of any requested pair, then that vertex can be skipped. The same applies for the Bellman-Ford algorithm.

What's the point of using the Floyd-Warshall Algorithm then? It is much easier and faster to code than iterating Dijkstra's Algorithm V times and can save a lot of time in contests. Dijkstra's Algorithm also becomes very slow for very dense graphs (especially complete graphs) as E approaches V^2. The Floyd-Warshall Algorithm becomes faster than Dijkstra's Algorithm for this case.

Notes on Minimum Cost Spanning Tree Problems

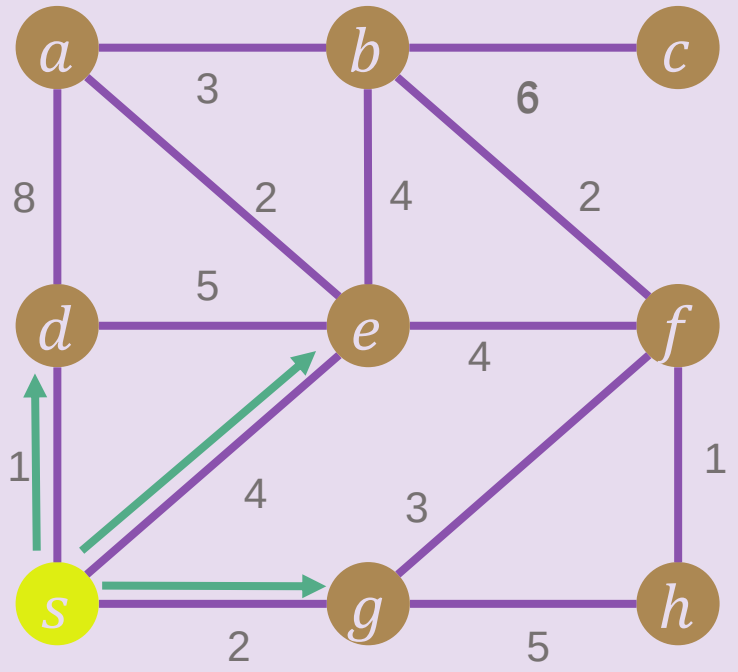
Minimum cost spanning tree problems always use undirected graphs. If used with directed graphs, the problem is called the minimum cost arborescence problem and requires a different algorithm to solve.

A variant of this problem, the minimum cost spanning forest problem, requires you to create multiple trees instead of one single tree. Both algorithms to be discussed can be used to solve these problems.

Prim's Algorithm

Prim's Algorithm finds a minimum cost spanning tree of a graph.

Prim's Algorithm begins with the single source vertex s being the only vertex in the spanning tree and determines the cost of adding each vertex u adjacent to s to the tree if we take the edge (s, u).

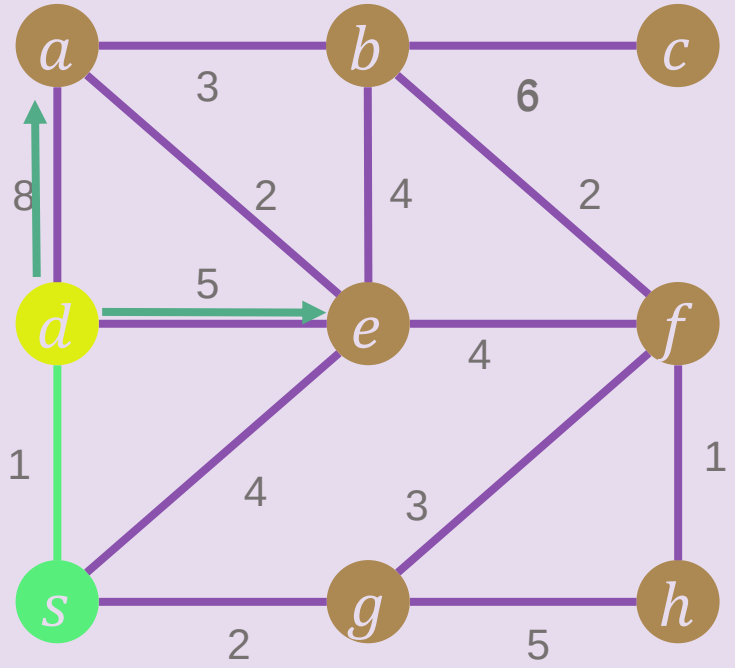


s	0	✓
a	∞	
b	∞	
c	∞	
d	1	
e	4	
f	∞	
g	2	
h	∞	

Prim's Algorithm

Prim's Algorithm finds a minimum cost spanning tree of a graph.

It takes the cheapest of these vertices and adds it to the spanning tree. It then, again, determines the cost of adding each vertex adjacent to the current vertex being added, always keeping track of the edge used in the cheapest way.

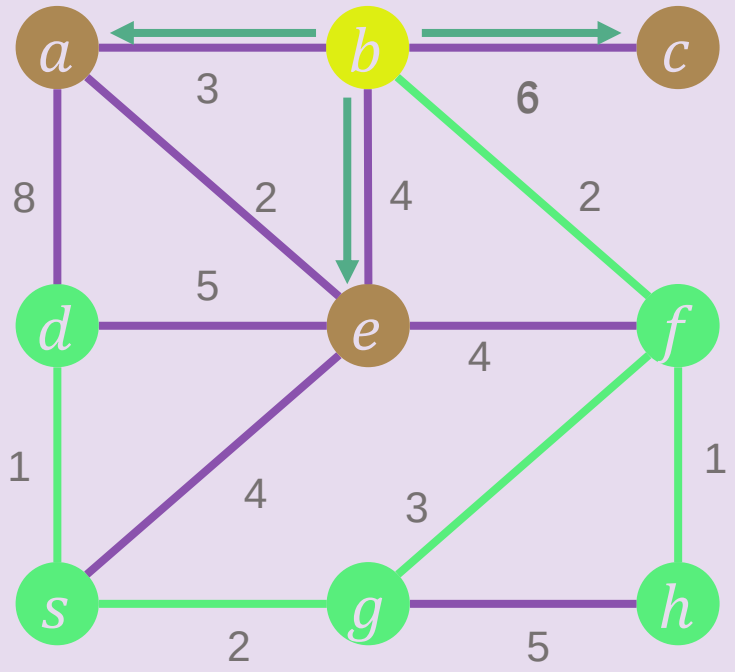


s	0	✓
a	8	
b	∞	
c	∞	
d	1	✓
e	4	
f	∞	
g	2	
h	∞	

Prim's Algorithm

Prim's Algorithm finds a minimum cost spanning tree of a graph.

This is repeated until all vertices have been added to the spanning tree.

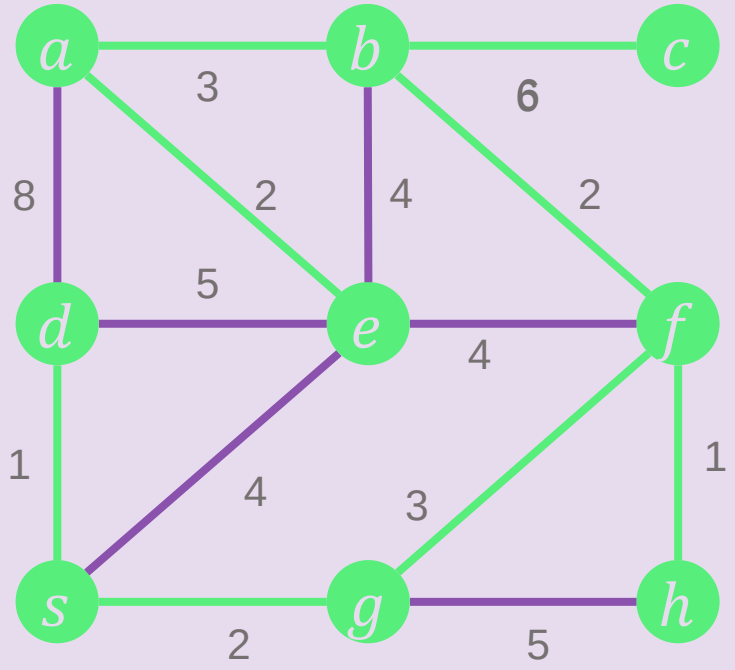


s	0	✓
a	3	
b	2	✓
c	6	
d	1	✓
e	4	
f	3	✓
g	2	✓
h	1	✓

Prim's Algorithm

Prim's Algorithm finds a minimum cost spanning tree of a graph.

Once all vertices have been added to the spanning tree. All edges in the minimum cost spanning tree of the graph, and the minimum cost spanning tree itself, have been found.



s	0	✓
a	3	✓
b	2	✓
c	6	✓
d	1	✓
e	4	✓
f	3	✓
g	2	✓
h	1	✓

Prim's Algorithm

Prim's Algorithm works very similarly to Dijkstra's Algorithm. In fact, the implementation is the same except we set the "distances" in Dijkstra's Algorithm to edge length instead of distance + edge length. This is because Prim's Algorithm works with the same idea – that selecting the next cheapest vertex and expanding from there will always give an optimal result.

Prim's Algorithm will still work on graphs with negative edge weights because, unlike Dijkstra's Algorithm, Prim's Algorithm only needs to take individual edges, instead of entire paths, into account.

Prim's Algorithm Sample Implementation ($O(V^2 + E)$)

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices for that test case.
//In this sample, our source vertex is 0.
bool vis[N]; int cost[N]; vector<int> adj[N], adjw[N];

int main(){
    //read graph into adj, adjw
    //set vis[0]..vis[n-1] to false
    //set cost[1]..cost[n-1] to inf or -1 (sentinel value)
    cost[0] = 0;
    int total = 0;
    while(true){
        int next = -1;
        for(int i=0; i<n; i++){
            //add extra check if sentinel is -1
            if(!vis[i] && (next == -1 || cost[i] < cost[next]))
                next = i;
        }
        if(next == -1) break; //no more unvisited vertices
        vis[next] = true;
        total += cost[next];
        for(int i=0; i<adj[next].size(); i++){
            if(vis[adj[next][i]]) continue;
            // or if cost[adj[next][i]] == -1 if sentinel is -1
            if(adjw[next][i] < cost[adj[next][i]]){
                cost[adj[next][i]] = adjw[next][i];
            }
        }
    } //total will contain the cost of the MCST
}
```

```
//For constructing the minimum cost spanning tree
//itself, we keep track of which edges are used to
//add vertices to the MCST.
vector<int> adjid[N]; //give edges ids
int edge[N]; //id of the edge used

//This may be implemented many ways.
//Giving edges ids is just one.

//set edge[0]..edge[n-1] to -1

//Whenever we update the cost of a vertex, we use
//the edge currently being processed.
cost[adj[next][i]] = adjid[next][i];

//Reconstruct the minimum cost spanning tree by
//finding all edges used. Most of the time, the
//problem will only ask for the ids of the edges
//used. In this case, we can simply print the ids
//stored in edge[1]..edge[n-1].

//Sometimes the problem asks for the actual edges
//(the vertices and the weight) or other
//information attached to these edges. In these
//cases, it could be easier to store the edges in
//an additional edge list or store them as objects.
```

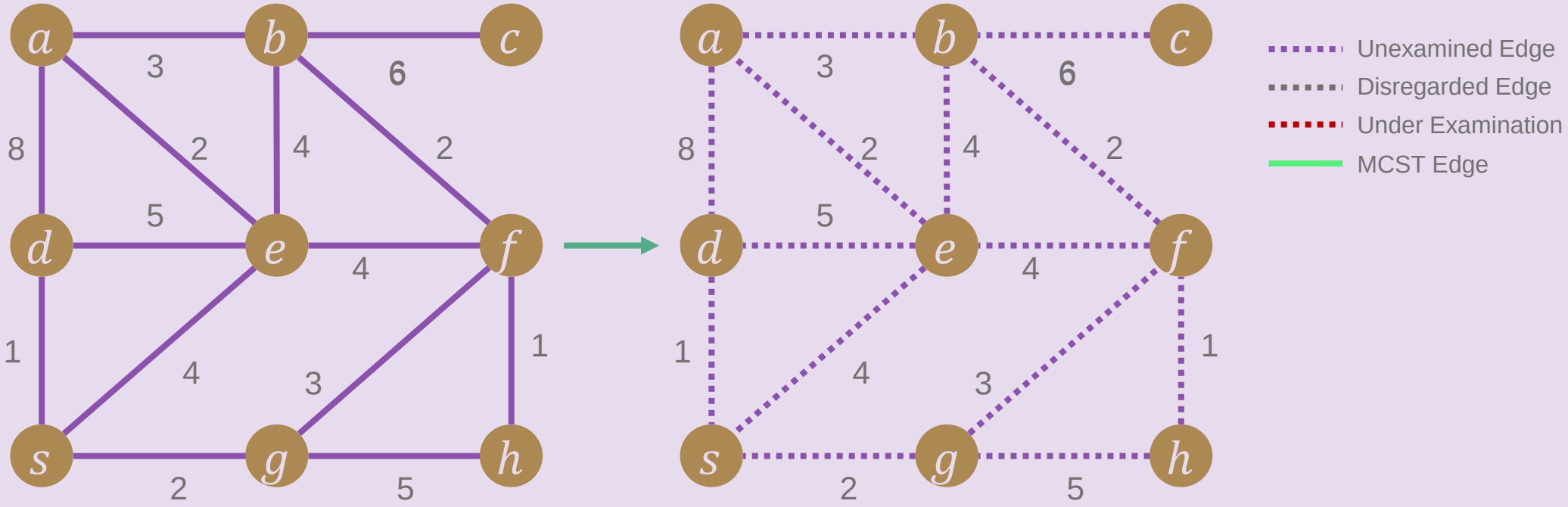
Prim's Algorithm Sample Implementation ($O(V \log E + E)$)

Like Dijkstra's Algorithm, Prim's Algorithm can be sped up by using a priority queue to find the next cheapest vertex to add.

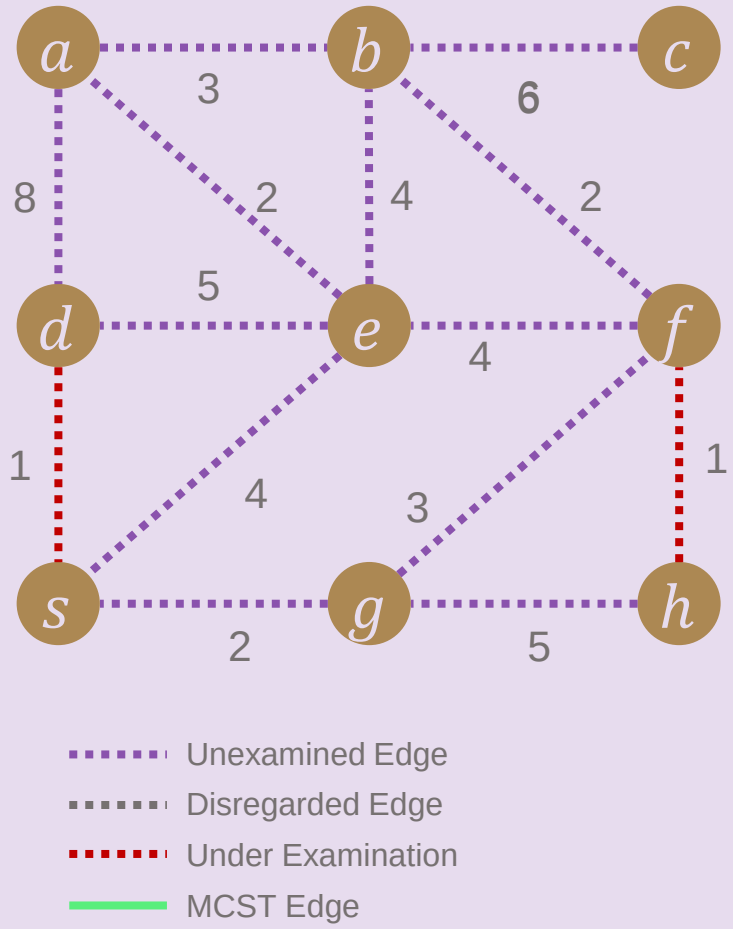
```
//define a new comparator for the priority queue
struct cmp{
    bool operator()(int a, int b){
        return cost[a] > cost[b]; //get the smallest cost first
    }
};

//after setting all the starting values
priority_queue<int, vector<int>, cmp> pq;
pq.push(0); //we start with the source vertex
while(pq.size() > 0){ //instead of while(true), we only need to check if the priority queue is nonempty
    int next = pq.top(); pq.pop(); //instead of searching, we can just get the next element in pq
    if(vis[next]) continue; //the same vertices will appear multiple times
    vis[next] = true;

    //process as before but push the new vertices into the priority queue
    for(int i=0; i<adj[next].size(); i++){
        if(vis[adj[next][i]]) continue;
        if(adjw[next][i] < cost[adj[next][i]]){
            cost[adj[next][i]] = adjw[next][i];
            pq.push(adj[next][i]);
        }
    }
}
```

Kruskal's Algorithm

Kruskal's Algorithm is another method for finding the minimum cost spanning tree given a graph. The concept is simple: we start with the the same input graph, but without any of the edges. We go through each of the edges, starting with the one with least weight. If adding the edge to the graph forms a cycle, we disregard it, otherwise we add it to our MCST. We do this until we have gone through all the edges or we have added $|v| - 1$ edges to our graph, forming the MCST.



Kruskal's Algorithm

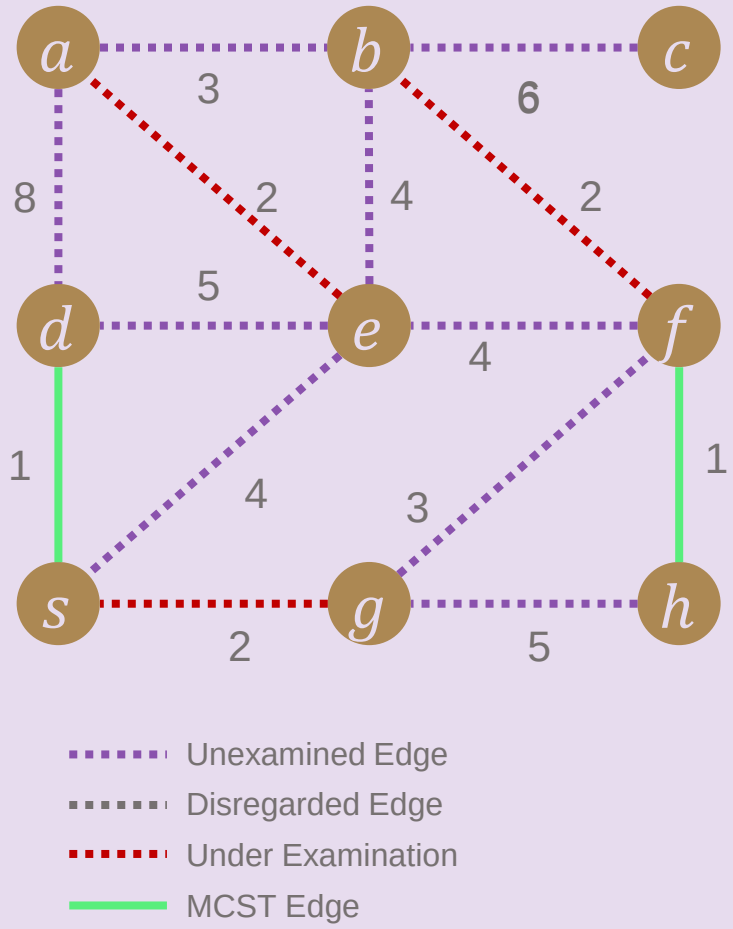


Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

First, edges are sorted in ascending order. Implementation wise, this sorting is usually done implicitly by placing all edges in the graph in a priority queue. We then poll from the queue each time we examine a new edge, guaranteeing that we go through edges with the lowest weight first.

The edges with the lowest weight in the graph are $\{f, h\}$ and $\{d, s\}$. Neither of these edges form a cycle when added to our graph.

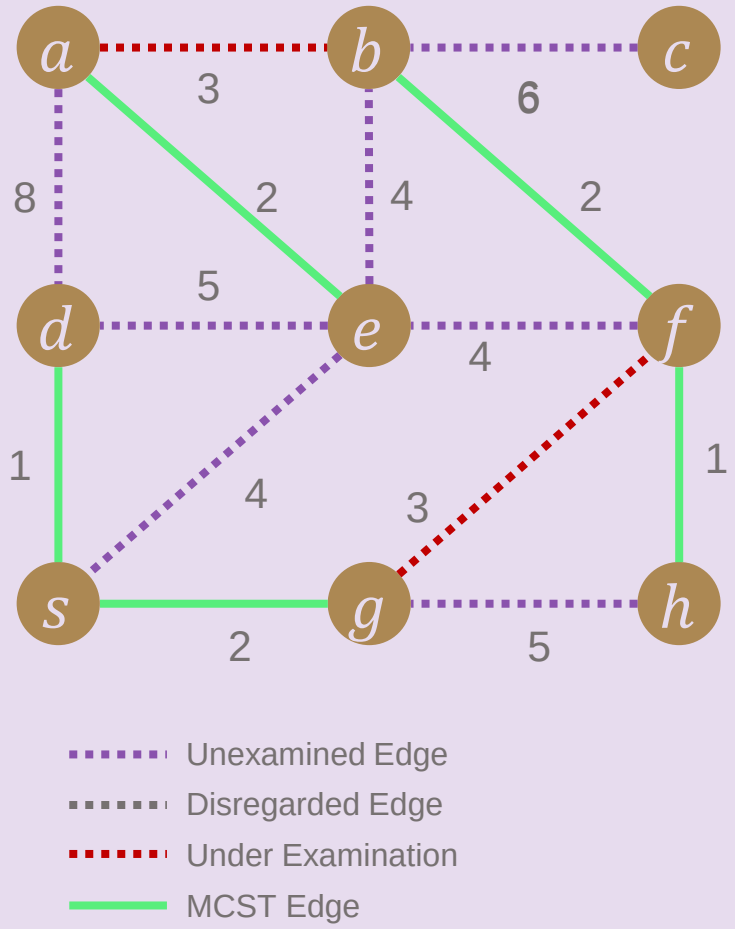
Kruskal's Algorithm



Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

We then proceed to examine more edges. $\{b, f\}$, $\{a, e\}$ and $\{s, g\}$ all have a weight of 2. None of these edges produce a cycle when added to the graph.

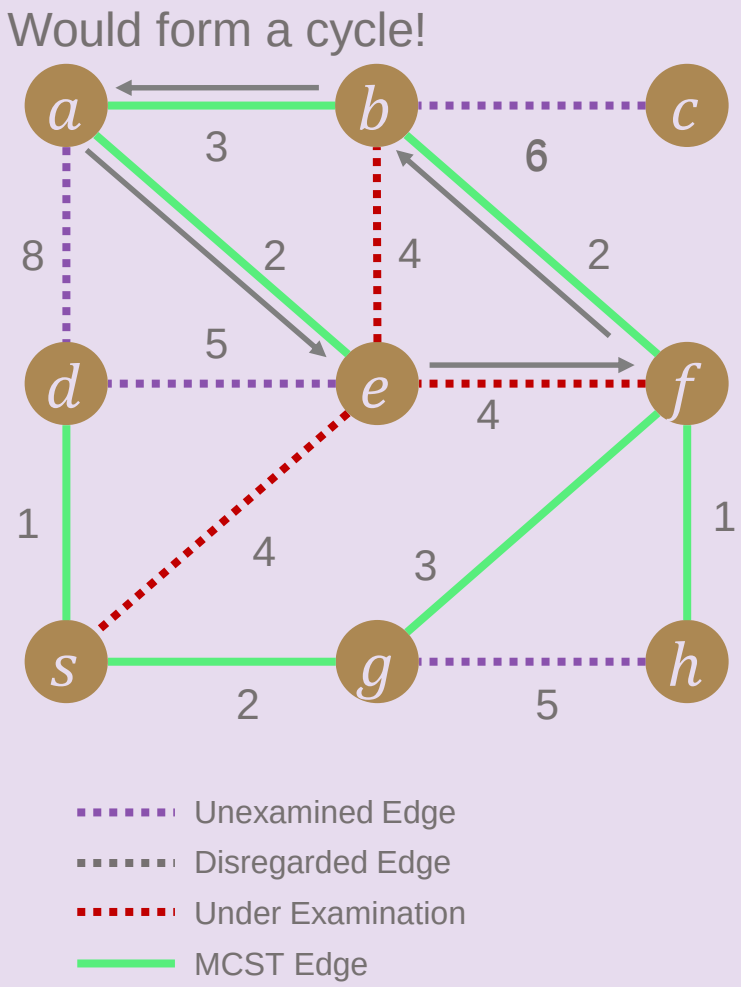
Kruskal's Algorithm



Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

$\{a, b\}$ and $\{g, f\}$ both have a weight of 3. Adding both these edges to the graph, there are still no cycles formed, so they are part of our MCST.

Kruskal's Algorithm



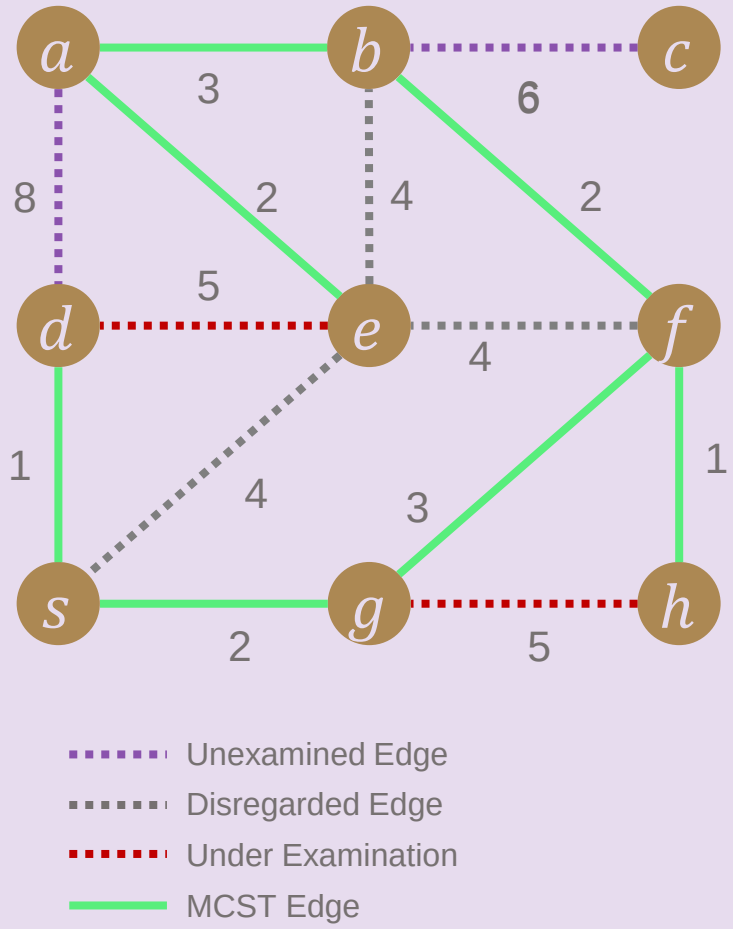
Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

$\{e, f\}$, $\{e, s\}$ and $\{b, e\}$ all have weights of 4. We first look at the edge $\{e, f\}$. Notice that if we add it to the graph, we form the cycle $a \to e \to f \to b \to a$. Thus we should disregard the edge $\{e, f\}$.

Similarly, adding the edge $\{e, s\}$ would form the cycle $a \to b \to f \to g \to s \to e \to a$, thus we should disregard it.

Adding the edge $\{b, e\}$ would also form a cycle, $a \to e \to b \to a$, thus we also disregard the edge $\{b, e\}$.

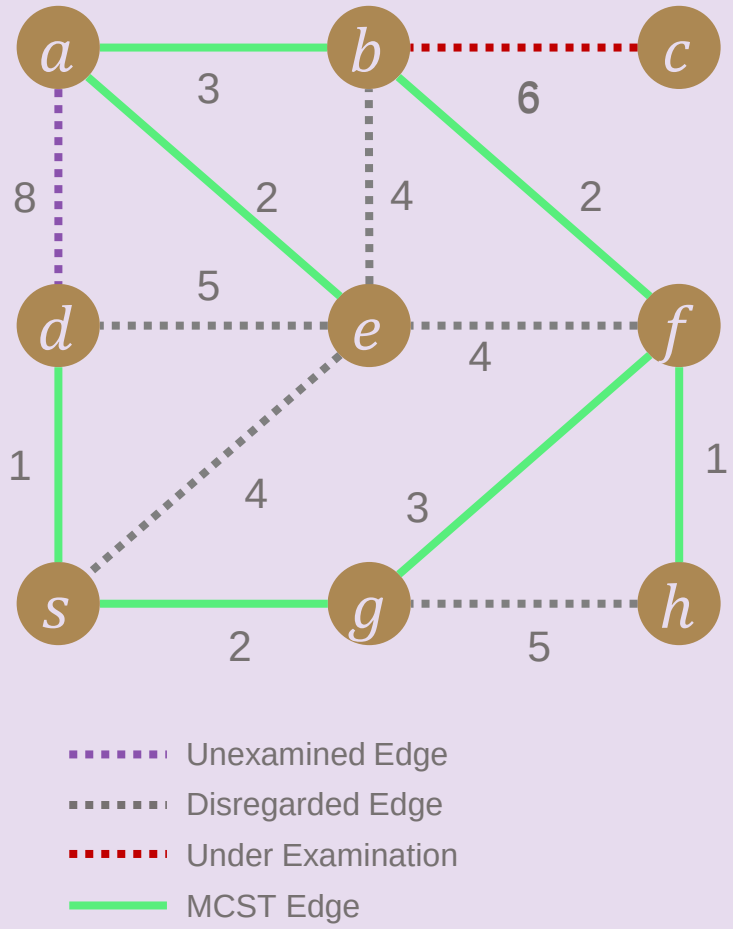
Kruskal's Algorithm



Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

The edges $\{d, e\}$ and $\{g, h\}$ both have a weight of 5. By inspection, we can see that adding either edge to the graph will result in the formation of cycles. We should thus disregard both edges.

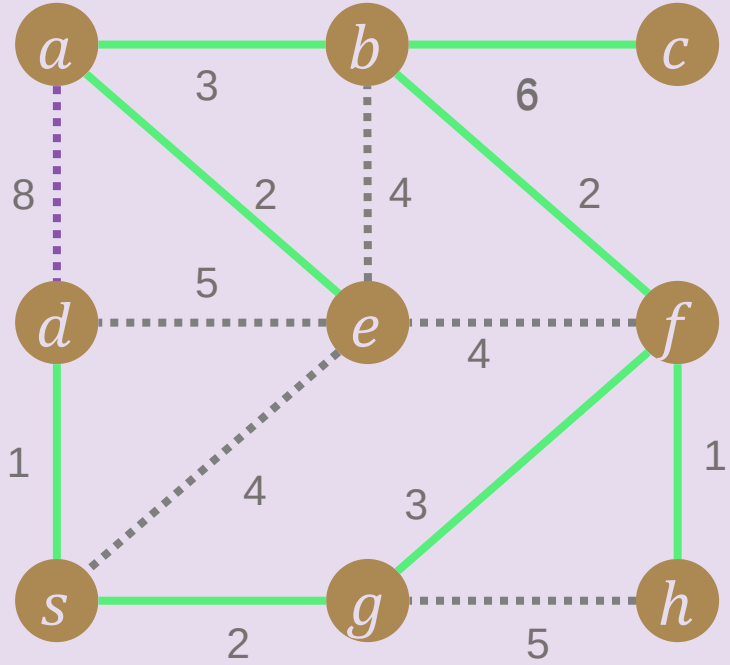
Kruskal's Algorithm



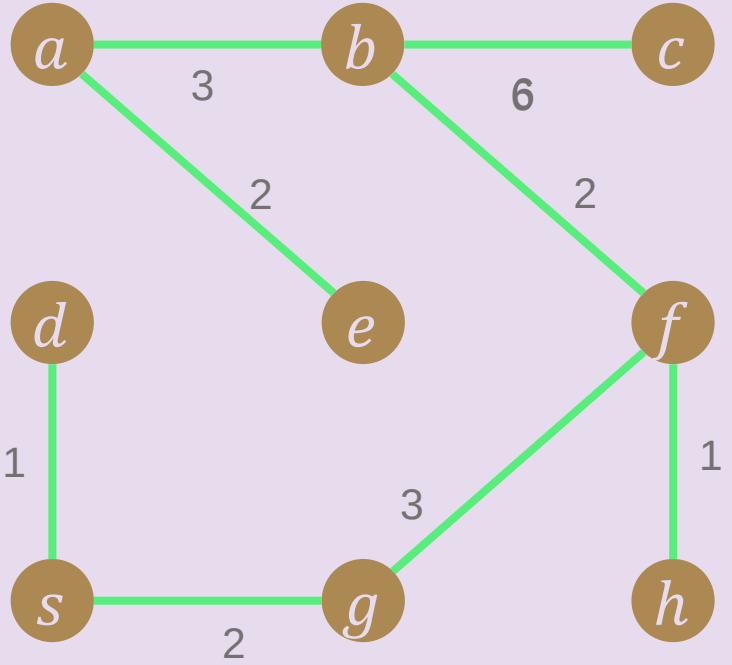
Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

The next edge is $\{b, c\}$ with a weight of 6. Adding this to our graph does not form any cycles, thus we add it to our MCST. Note that after adding $\{b, c\}$, we have added $|v| - 1$ edges to our graph. This means we may now end the algorithm.

Kruskal's Algorithm



Unexamined Edge
Disregarded Edge
Under Examination
MCST Edge



Now that the algorithm has ended, the edges that were added to the graph will form our final MCST.

Kruskal's Algorithm Sample Implementation

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices for that test case.

struct edge{
    int u, v, w; //the two vertices and the weight
    edge(int u, int v, int w){
        this->u = u; this->v = v; this->w = w;
    }
};
struct cmp{
    bool operator()(const edge &a, const edge &b){
        return a->w > b->w; //priority queue in C++ is max heap
    }
};

int main(){
    priority_queue<edge, vector<edge>, cmp> pq;
    //insert all edges into pq
    int total = 0;
    int need = n-1;
    while(need > 0){
        edge cur = pq.top(); pq.pop();
        if(!formsCycle(cur)){
            //add cur to graph
            total += cur.w;
            need--;
        }
    }
    //total will contain the cost of the MCST
}
```

Kruskal's Algorithm

In order for Kruskal's Algorithm to work, we need a way to determine if adding an edge (u, v) to the current graph creates a cycle. Since the graph is undirected, this is the same as checking whether u and v are already connected and can be easily done using a Breadth-First or Depth-First Search. That however would take $O(E)$ time for each edge, making Kruskal's Algorithm run in $O(E^2 + E \log E)$ time, which is much slower than Prim's Algorithm.

There is a much faster way for us to determine whether two vertices are connected while constructing the MCST using Kruskal's Algorithm. This is called Union-Find or Disjoint Set Union.

Union-Find / Disjoint Set Union

Union-Find or Disjoint Set Union is a method to determine which items from a number of mutually disjoint sets (each item is in exactly one set) are part of the same set. It has two operations:

- Find – Determine the set an element is part of.
- Union – Combine two sets into one.

This is done by assigning a representative element to each set. Find then returns this representative element. To check whether two elements are part of the same set, we just check whether their representative elements are the same.

This is done by creating a tree using the elements with the root as the representative element. Union then simply sets the parent of one root as the other root, effectively combining the two trees.

In Kruskal's Algorithm, we assign each connected subgraph to a set and the vertices as the elements.

Kruskal's Algorithm Sample Implementation with Union-Find

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices for that test case.

struct edge{
    int u, v, w; //the two vertices and the weight
    edge(int u, int v, int w){
        this->u = u; this->v = v; this->w = w;
    }
};
struct cmp{
    bool operator()(const edge &a, const edge &b){
        return a->w > b->w; //priority queue in C++ is max heap
    }
};

int main(){
    priority_queue<edge, vector<edge>, cmp> pq;
    //insert all edges into pq
    int total = 0;
    int need = n-1;
    while(need > 0){
        edge cur = pq.top(); pq.pop();
        if(!formsCycle(cur)){
            //add cur to graph
            total += cur.w;
            need--;
        }
    }
    //total will contain the cost of the MCST
}
```

```
//Keep track of the parent of each vertex
int par[N];

//Follow the parent of the vertex being checked
//until you reach the root.
int find(int u){
    if(par[u] == u) return u;
    return find(par[u]);
}

//Set the parent of one root to the other. Merge
//is used here because union is a reserved keyword.
void merge(int a, int b){
    par[find(a)] = find(b);
}
```

```
//Since no vertices are connected at the start, each
//is in its own tree.

//set par[0]..par[n-1] to 0..n-1 (par[i] = i)
```

```
//formsCycle(cur) is changed to checking whether
//cur.u and cur.v are in the same tree.

if(find(cur.u) != find(cur.v))
```

```
//Adding cur to the graph is just taking the union.
merge(cur.u, cur.v);
```

Union-Find Optimizations

Find takes $O(V)$ time, while Union takes $O(V)$ time if we use Find again or $O(1)$ time if we keep track of what Find returned when we first checked the parents of u and v. This makes Kruskal's Algorithm run in $O(VE + E \log E)$ time.

Find can be further optimized by replacing the parent of each vertex passed in the recursive function with the representative element. This allows the vertex to go straight to its tree's root instead of having to go through each parent first.

```
int find(int u){
    if(par[u] == u) return u;
    return par[u] = find(par[u]); //reassign before returning
}
```

Union-Find Optimizations

Union can also be optimized by making the tree with the larger depth the new root instead of using either of the two roots. This reduces the height of each tree, reducing the number of function calls Find has to go through to reach the root of a set.

```
int depth[N];

int union(int a, int b){
    int roota = find(a);
    int rootb = find(b);

    if(depth[roota] < depth[rootb]){
        par[roota] = rootb;
    }else if(depth[rootb] < depth[roota]){
        par[rootb] = roota;
    }else{
        par[rootb] = roota;
        depth[roota]++;
    }
}

int main(){
    //set depth[0]..depth[n-1] to 0 before Kruskal's Algorithm
}
```

Union-Find Optimizations

It can be proven that using both of these optimizations reduces the running time of Find to $O(\alpha^{-1}(V))$ where $\alpha^{-1}(n)$ is the inverse Ackermann function (proof outside of scope), an extremely slowly growing function that, for most practical values of n, is less than 5. This effectively reduces the complexity of Union-Find to a small constant.

Using these optimizations, the complexity of Kruskal's Algorithm reduces to $O(E \log E)$.

On Prim's and Kruskal's Algorithms

For most cases, Prim's and Kruskal's algorithms are effectively the same in terms of running time. Use which one you are more comfortable with.

Any variant of the typical MCST problem that can be covered by one of these algorithms can be covered by the other as well. However, the modifications necessary to solve the problem may be more complicated for one of them, so it is still suggested to be familiarized with both algorithms.

Basic Implementation Problems for Practice (Optional)

- CodeForces 20C – Dijkstra?
- UVa 558 – Wormholes
- UVa 821 – Page Hopping
- UVa 11631 – Dark Roads
- UVa 10147 – Highways

Problems (Required)

- UVa 1235 – Anti Brute Force Lock
- UVa 11733 – Airports
- UVa 10600 – ACM Contest and Blackout
- UVa 10557 – XYZZY
- UVa 1250 – Robot Challenge
- CodeForces 229B – Planets
- CodeForces 329B – Biridian Forest
- CodeForces 676D – Theseus and Labyrinth

Challenges (At least 3 required)

- UVa 1202 – Finding Nemo
- UVa 1253 – Infected Land
- UVa 11329 – Curious Fleas
- CodeForces 295B – Greg and Graph
- CodeForces 295C – Greg and Friends
- CodeForces 472D – Design Tutorial: Inverse the Problem

- For those who have not gotten full points in “The Cheapest Reid” from NOI 2017 eliminations, we encourage you to try it again. This does not count towards the 3 problems.