A common application of graphs is to model some sort of transportation network (airports, roads, etc.).

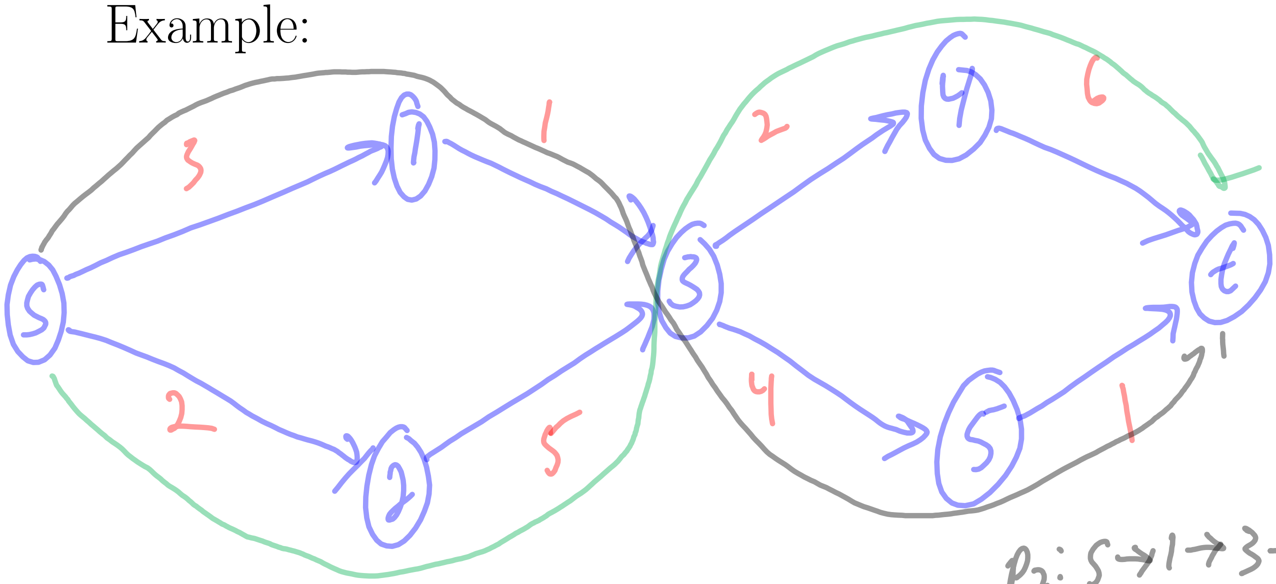
If we are currently at a location s and wish to travel to a location t, then we may want to find a path which starts at s and ends at t. There may be many such paths, and in the application, some paths may be much more expensive to follow than others.

We can assign a weight to each edge $\{u, v\}$ of the graph which represents the cost of moving from location u to location v. Now consider some path p from s to t. The weight of the path $w(p)$ is the sum of the edge weights along the path.

p_1: $s \to 2 \to 3 \to 4 \to t$

$w(p_1)$: $2+5+2+6 = 15$

Example:

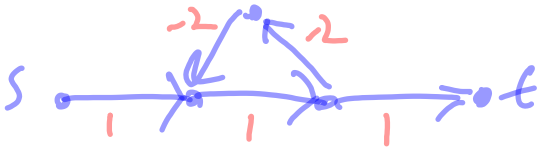


p_2: $s \to 1 \to 3 \to 5 \to t$

$w(p_2) = 3+1+4+1 = 9$

In this setting, we would be interested in computing a **shortest path** from s to t. A shortest path is a path of minimum weight from s to t. Let $\delta(u, v)$ denote the weight of a shortest path between any two vertices u and v in the graph ($\delta(u, v) = \infty$ if there are no paths from u to v).

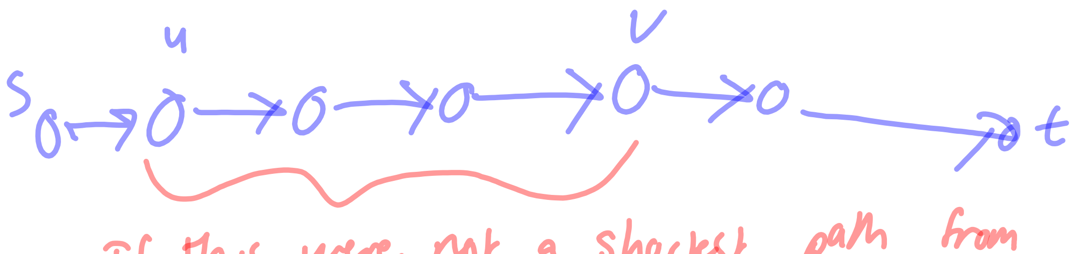
In some applications we may want to have negative weights on an edge. Note that if there is a negative-weight cycle, then some shortest paths may not exist.



The following theorem is crucial in the computation of shortest paths:

Thm A subpath of a shortest path is a shortest path.

Shortest path from s to t

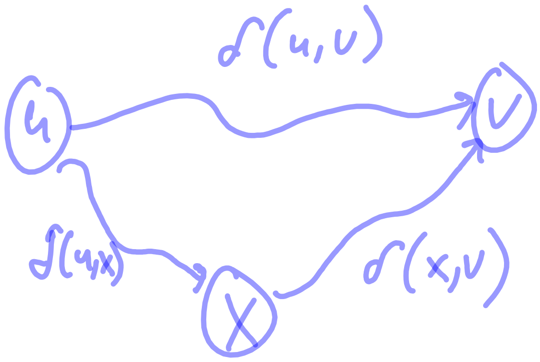


If this were not a shortest path from u to v, then we could replace this subpath with the shortest path from u to v and get a shorter path from s to t, a contradiction.

The following theorem is known as the *triangle inequality* and is also important in the computation of shortest paths:

Thm: For all $u, v, x \in V$ we have

$$\delta(u,v) \leq \delta(u,x) + \delta(x,v).$$



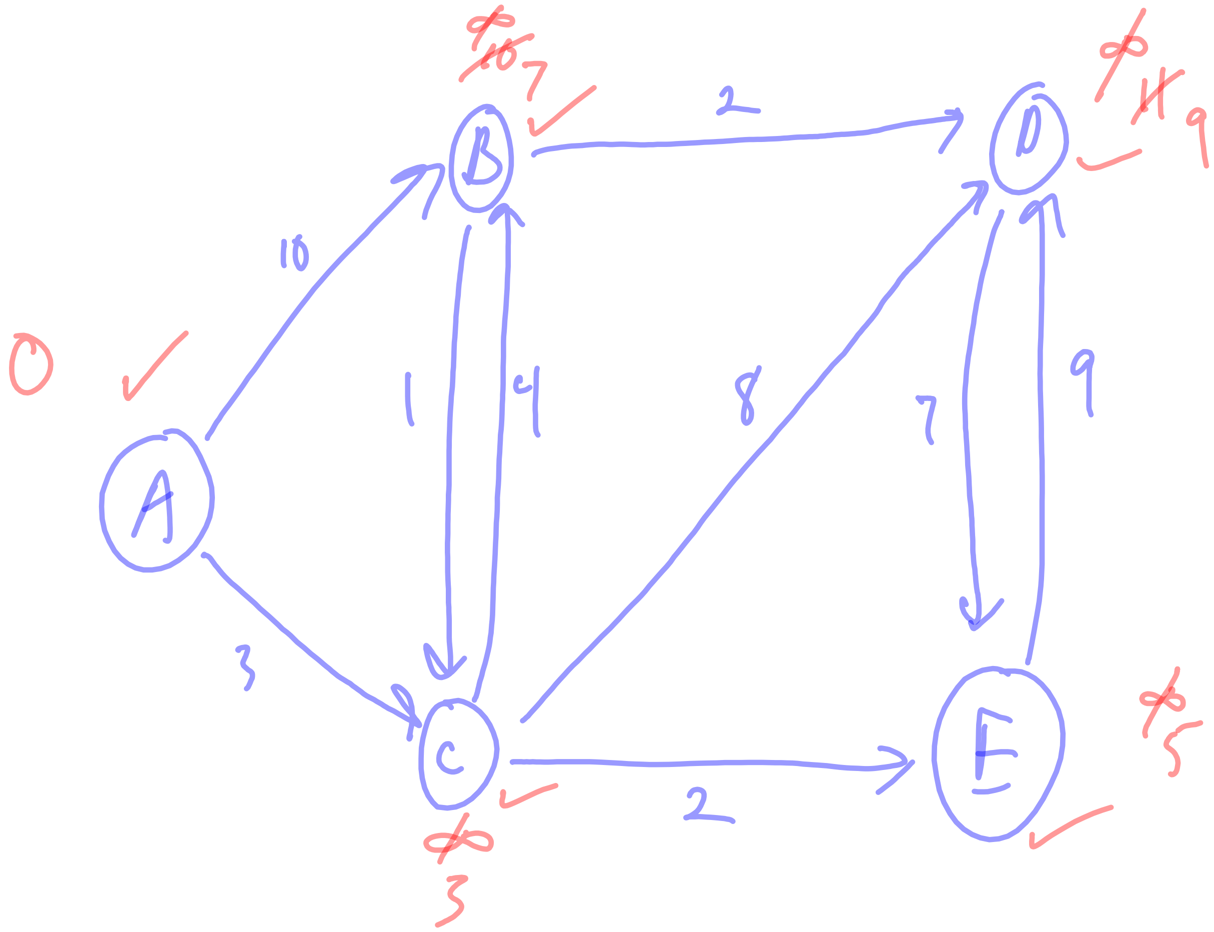
We will now consider the *single-source shortest path* problem in which we are given a graph with a designated source vertex s, and we wish to compute the shortest path weights $\delta(s, v)$ for each $v \in V$.

We will assume all edge weights are nonnegative so that we will not have any negative weight cycles.

Dijkstra's Algorithm. Idea: Greedy. For each vertex v, we maintain an upper bound $d[v]$ on $\delta(s, v)$.

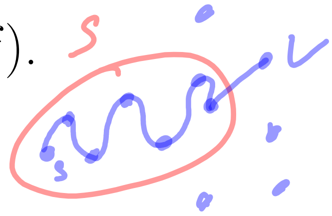
1. Maintain a set S of vertices whose shortest path weights from s are known, that is $d[v] = \delta(s, v)$.
2. At each step, add the vertex $v \in V \setminus S$ whose $d[v]$ is minimal.
3. Update $d[u]$ for any vertex u adjacent to v.

Example of Dijkstra's Algorithm:



Correctness of Dijstra's:

Theorem: (i) For all $v \in S : d[v] = \delta(s, v)$. (ii) For all $v \notin S : d[v]$ is the weight of a shortest path from s to v that uses only vertices in S (besides v itself).



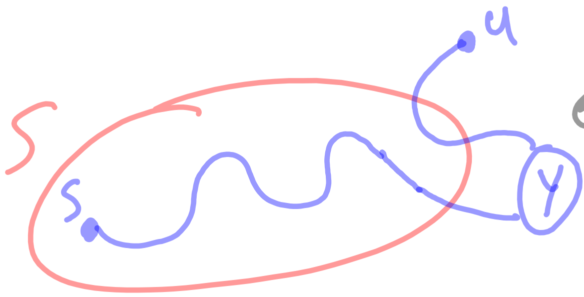
The implication of this theorem is that Dijkstra's algorithm terminates with $d[v] = \delta(s, v)$ for each $v \in V$ (because each $S = V$ at the end of the algorithm).

The proof is by induction. It is clearly true in the base case ($d[s] = 0$ and $d[v] = \infty$ for all $v \neq s$). Assume (i) and (ii) are true before an iteration, and we will show it remains true after another iteration.

Let u be the vertex added to S in this iteration. So $d[u] \le d[v]$ for all $v \in V \setminus S$.

(i) We need to show $d[u] = \delta(s,u)$.

Suppose the contrary. Then there is a path p from s to u with $w(p) < d[u]$. So there must be some vertex $y \in V \setminus S$ on the path p.



edge weights are non-negative, we have $d[y] \le d[u]$

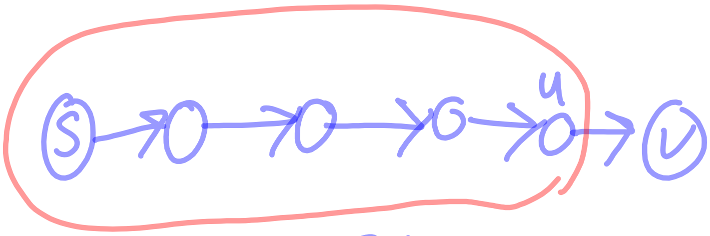
So we should have taken y instead of u, so such a path does not exist. Thus $d[u] = \delta(s,u)$.

(ii) Let $v \notin S$. Let p be a shortest path from s to v that only uses vertices in S (besides v itself).

2 cases:

1) p does not contain u. Then (ii) is true by inductive hypothesis.

2) p contains u:



p consists of vertices in $S \setminus \{u\}$, then to u, then to v.

$\Rightarrow w(p) = d[u] + w(u,v)$ which is we set $d[v]$ to after this iteration.

So (ii) is true.

The running time of Dijkstra's algorithm is $O(m \log n)$ when maintaining $V \setminus S$ as a priority queue.

Now consider the *unweighted case* in which we want to find a path with the smallest number of edges.

Certainly Dijkstra's Algorithm can still work (we can just set the weight of each edge to be 1). Can we do better?

Idea: do a modified BFS search starting from s. Recall BFS traverses the graph in "layers" where each layer will be the same distance from s.

So far we have only been computing the *length* of a shortest path. What if we want to know what the shortest path actually is?

We can build a shortest path tree similarly to how we built MST, BFS, and DFS trees (we remember the “predecessor” for each vertex during the computation).