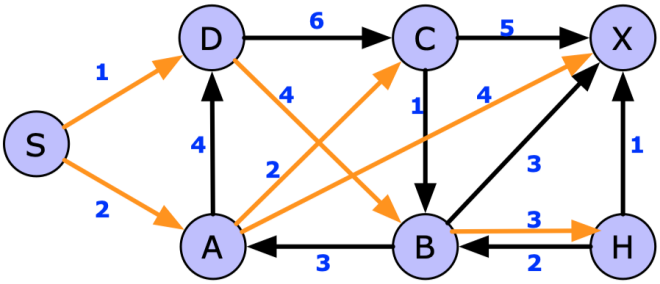
Graph Algorithms 4: Single Source Shortest Path

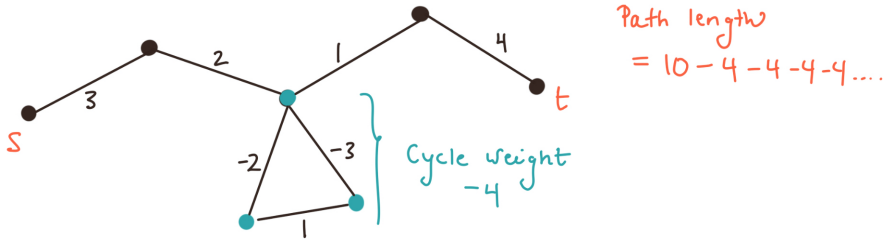
There are many everyday problems that involve finding the **shortest path** from one object to another. For example, suppose we are given a map consisting of cities and roads, and we want to find the shortest route to particular destination. This is an example of solving the shortest-path problem.

1 Single Source Shortest path

Given a directed graph G with weighted edges, and a source vertex s, the **single source shortest-path** or SSSP, is the set of all shortest paths from the source vertex, s, to any other vertex v in the graph. In the example below, the SSSP is highlighted in orange. The edges represent the shortest paths from s to any other vertex in the graph. For example, the shortest path from S to X follows along the path $S \to A \to X$ having total weight 6. Note that any other path to vertex X has a larger total weight.

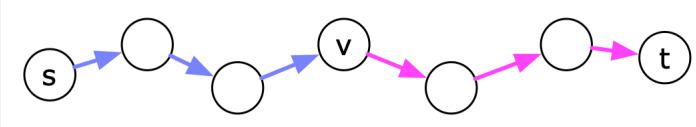


The notion of a shortest-path is well-defined, as long as there is **no negative-weight cycle** that is reachable from s. In the example below, it is clear that having a negative-weight cycle means that you could walk around the negative-weight cycle indefinitely, making the weight of the path smaller and smaller. Therefore we consider only directed graphs that have no negative-weight cycles.



The SSSP also satisfies the following important fact, which is important in developing the first algorithm used to solve this problem.

Fact 1:
Suppose G is a directed graph with no negative-weight cycle. Let p be the shortest path from vertex s to t . Suppose vertex v is an intermediary vertex on this path. The the shortest path from s to v is given by the first part of p, and the shortest path from v to t is given by the second part of p.



In this lecture we look at two algorithms which solve the SSSP problem:

- **Dijkstra's algorithm**: solves the SSSP problem for directed graphs with **non-negative** weights.
- **Bellman-ford** algorithm: solves the SSSP problem for directed graphs and allows for negative-weight edges.

2 Dijkstra's algorithm

The first algorithm we look at is Dijkstra's algorithm - a technique used to solve the SSSP problem on weighted directed graph with **no negative-weight** edges.

This algorithm is very similar to Prim's algorithm: it grows a **shortest-path tree** from the source vertex s and uses a **Priority queue** to maintain the current distances from s to any vertex v. In fact, the pseudo-code below is very similar to that of Prim's. The main difference lies is the distance attribute $v.d$:

Dijkstra: The $v.d$ attribute stores the **total** distance from s to v

Prim: The $v.d$ attribute stores the distance from v to the **closest node** in the tree.

We shall see how this difference creates a different tree as we work though the first example below.

2.1 The Algorithm:

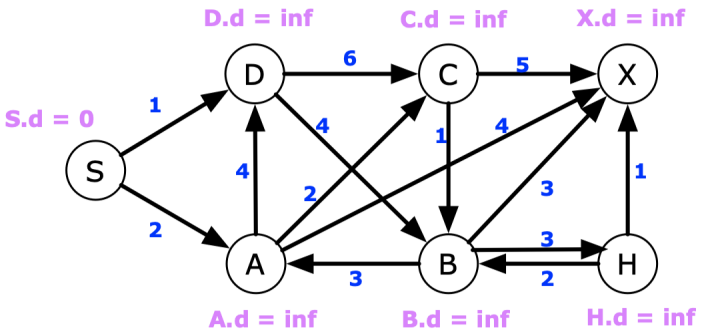
Recall from our lecture on Prim's algorithm that we defined two main operations on the priority queue, Q:

Extract-min(Q) and Decrease-key(Q,v,w)

These operations are defined in the same way there were for Prim's algorithm. In particular, *Decrease-key(Q,v,w)* assigns $v.d = w$ and updates the heap accordingly. Each of these run in time $O(\log n)$ for a heap of size n. As in our demonstration of Prims algorithm, we will not illustrate the items in the priority queue. Instead, we show the values of the $v.d$ attributes above each vertex, and the reader must keep in mind that those values are stored in a heap, where the minimum element is extracted using *Extract-min(Q)*.

Dijkstra's SSSP

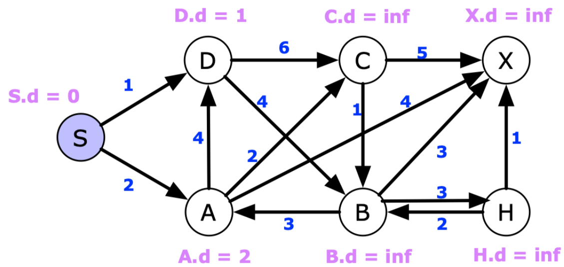
Step 1: Initialize the distances: set $v.d = \infty$ for all vertices and set $s.d = 0$
Add all vertices to the priority queue Q.
Initialize an empty tree T



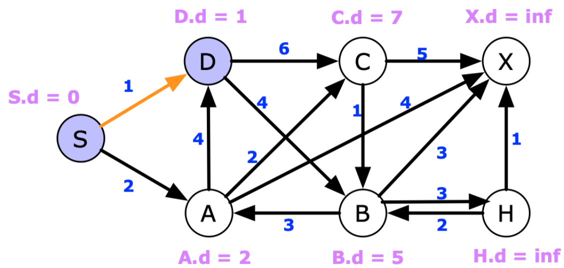
Step 2: Build the SSSP tree:

While Q not empty:
  u =Extract-min(Q)
  for each v in $Adj\,[u]$
    if $v.d > u.d + w(u,v)$ **We have found a shorter route to v*
      Decrease-key(Q,v,$u.d + w(u,v)$)
      $v.parent = u$

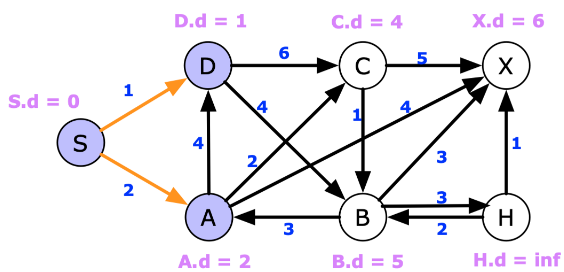
In the first iteration of the above while loop, vertex S is removed from Q and the distance attributes are updated for vertices A and D. The vertices in white are all those that are still in the queue, and those in blue have already been removed.



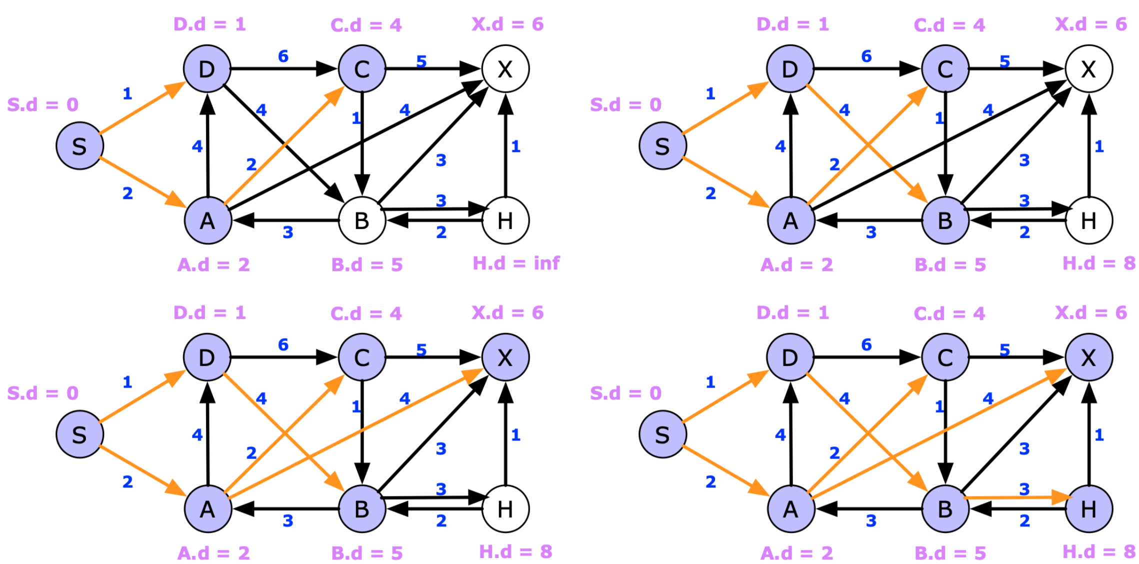
At the next iteration, vertex D is removed from Q (it is the current minimum, because S has been removed). Neighbor C is updated to $C.d = D.d + 6 = 1 + 6 = 7$. Similarly, neighbor B is updated to $4 + 1 = 5$.



Vertex A is the next to be removed (A.d = 2). Neighbor C is updated to $C.d = 2 + 2 = 4$ and neighbor X is updated to $2 + 4 = 6$.



The remaining stages of Dijkstra's are shown below:



The final tree edges are shown in orange. These edges represent the shortest path from vertex S to any other vertex in the graph. As in Prim's algorithm, the parent points in Dijkstra's algorithm are set whenever the distance is updated in the priority queue. Again, this update may occur several times. However, once a vertex v is *removed* from the priority queue, the *last* update to $v.parent$ is correct, and corresponds to the actual edge in the SSSP.

Runtime: There are exactly $O(V)$ Extract-min operations, and each one takes $O(\log V)$ time in a heap of size $O(V)$. This accounts for a runtime of $O(V \log V)$. Each directed edge accounts for a Decrease-key operation, and each Decrease-key runs in time $O(\log V)$. This accounts for a runtime of $O(E \log V)$. Building the initial heap and all other operations above run in time $O(V)$. The overall runtime is then $O(E \log V)$.

3 Bellman Ford

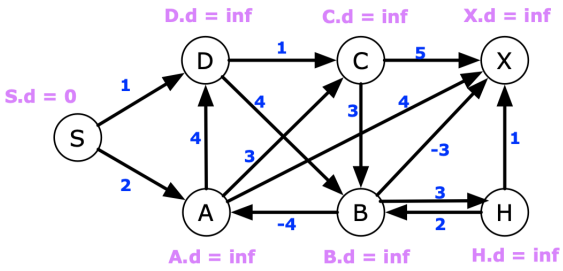
The next algorithm we look at is the Bellman-Ford algorithm, which solves the general SSSP problem given a directed graph G which **may contain negative weights**. Recall from above that although we may have negative weights, no negative-weight cycle can exist. The Bellman-ford algorithm will actually **detect a negative-weight cycle** and return **false** if one exists. Otherwise the algorithm produces all shortest paths from S.

The main difference between Dijkstra's algorithm and Bellman-Ford is that the latter reviews *all* edges in the graph at each iteration in order to determine if any of them produce shorter paths. Recall that Dijkstra's algorithm only reviews the edges adjacent to the last vertex added to the shortest-path tree. Bellman-ford does not require a Priority Queue, since we examine all edges at each iteration.

The algorithm is described below, and we use a slightly different example as we did for Dijskra's aglroithm (the graph below now has negative weights).

Bellman-Ford

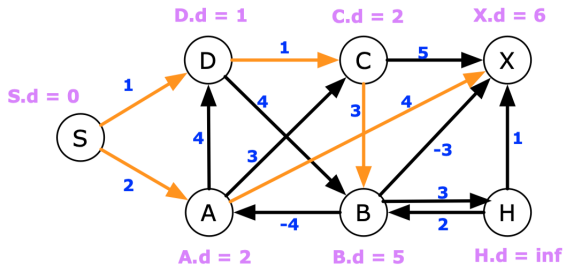
Step 1: Initialize the distances: for each v, set $v.d = \infty$. Set $s.d = 0$.
Initialize an empty tree T



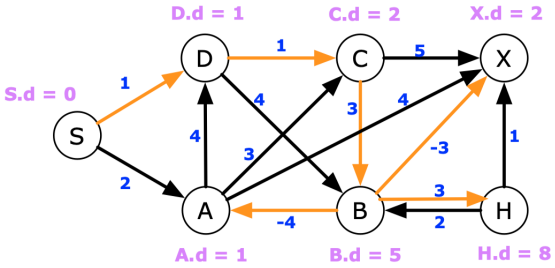
Step 2: Build the tree:

loop through exactly $V - 1$ times::
  for each edge (u, v) in E:
    if $v.d > u.d + w(u, v)$
      $v.d = u.d + w(u, v)$ **Found a shorter route*
      $v.parent = u$

The first iteration above processes every single edge in the tree to determine if it can update $v.d$. The order in which the edges are processes certainly effects the intermediary stages. In the example below, the edges were processed in the following order: edges out of S, edges out of A, edges out of B, edges out of C, edges out of D, edges out of H. The result after the first iteration of the outer for loop is shown below:



We repeat this process again, processing the edges in the same order. It may be that many edges produce no changes to the distance attributes, however some edges reduce the distance attributes. For example, the edge from B to A now sets $A.d = B.d - 4 = 5 - 4 = 1$. Similarly, the edge from B to X reduces $X.d$ to $X.d = 5 - 3 = 2$



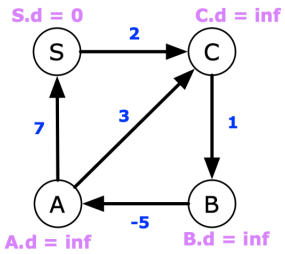
The next pass through all the edges results in no changes. The main for loop repeats $V - 1$ times, and at each pass, no more changes are made to any of the distance attributes.

After step 2, the final SSSP should be complete. Step 3 performs an additional pass through the graph, and if an update is made, then this means there is a negative-weight cycle in the graph, and we must return FALSE.

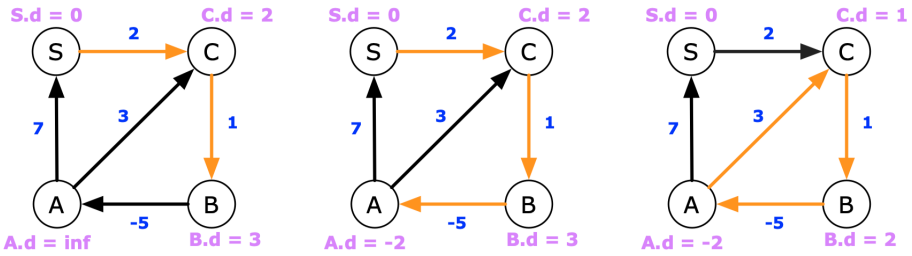
Step 3: The shortest paths should now be complete. If any edge can still reduce $v.d$, then there must be a negative-weight cycle. Check that by reviewing the effect of all edges:

for each edge (u, v) in E:
  If $v.d > u.d + w(u, v)$ **This means there must be a negative-weight cycle*
    return FALSE
return TRUE

The example below on 4 vertices contains a negative-weight cycle. Since $V = 4$, the SSSP should be complete after 3 passes. The first 3 passes are shown below:



When step 3 is carried out on the above graph on the right, the edge from B to A will reduce $A.d = 2 - 5 = -3$, resulting in a return value of FALSE. In this case, Bellman-ford has identified a negative-weight cycle.



Runtime: The nested for-loops in step 2 above account for a total runtime of $O(VE)$. Step 3 runs in time $O(E)$ and step 1 runs in time $O(V)$. Thus the overall runtime is $O(EV)$.

4 The SSSP problem in DAGs

Recall that a DAG is a directed graph with no cycles. We saw in a previous lecture, that it is possible to *topologically sort* these graphs. In this section, we look at a simple way this topological sort can be used to determine the shortest paths from s. Note that negative-weights are acceptable for this type of graph, since no cycles exist at all, negative or otherwise.

The topological sort enables us to determine in which order we should process the shortest paths. Moving from left to right in the topological sort, any path from vertex u to v in the figure below can only pass through vertices between u and v in the topological sort. Therefore we can actually process the shortest path lengths from left to right in the topological sort.

Dag-Shortest-Path

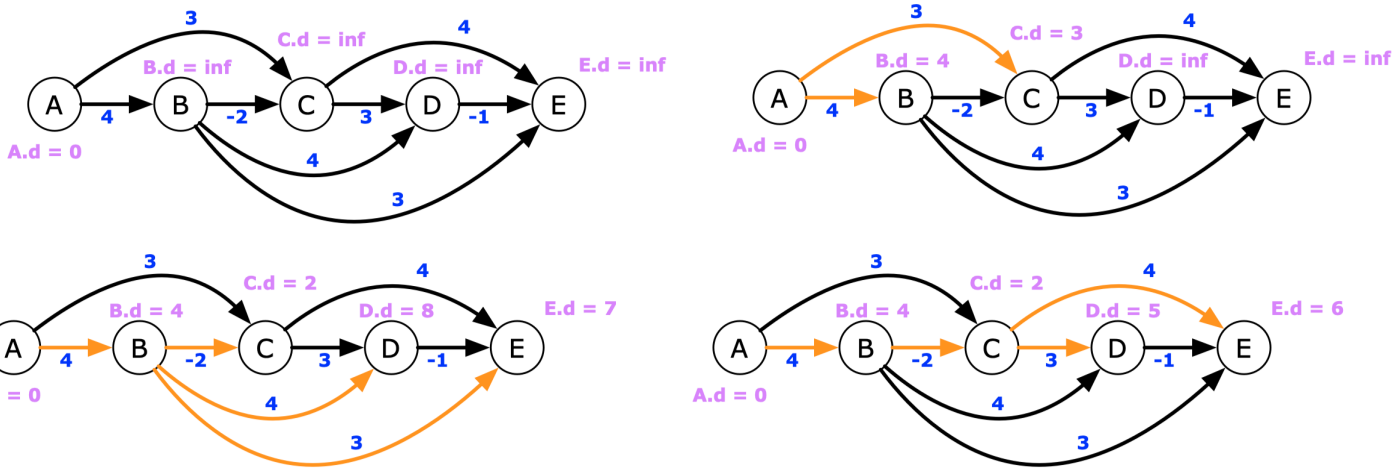
Step 1: Topologically sort the vertices of G.

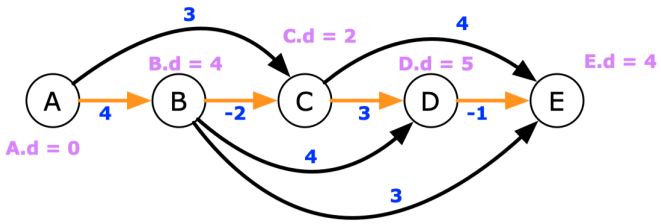
Step 2: Initialize the distances: $v.d = \infty$ for all vertices, except $s.d = 0$

Step 3: Loop through the vertices in the order they appear in the topological sort:

For each vertex u in topological order:
  for each v in $Adj\,[u]$:
    if $v.d > u.d + w(u, v)$
      $v.d = u.d + w(u, v)$
      $v.parent = u$

Example:





Runtime: Recall that Step 1 above takes time $O(V + E)$. In step 3, we carry out a constant amount of work for each edge, for a total of $O(E)$, and since the for-loop itself executes V times, the total runtime of step 3 is $O(V + E)$. Step 2 is clearly just $O(V)$. The overall runtime is $O(V + E)$, making this a much faster algorithm than Bellman-Ford or Dijkstra's.