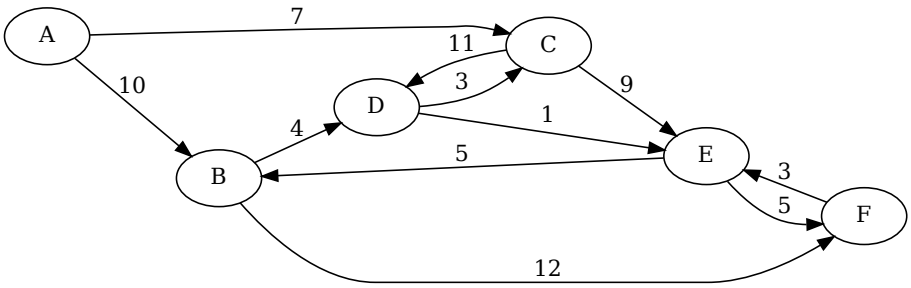
TD 5: SSSP

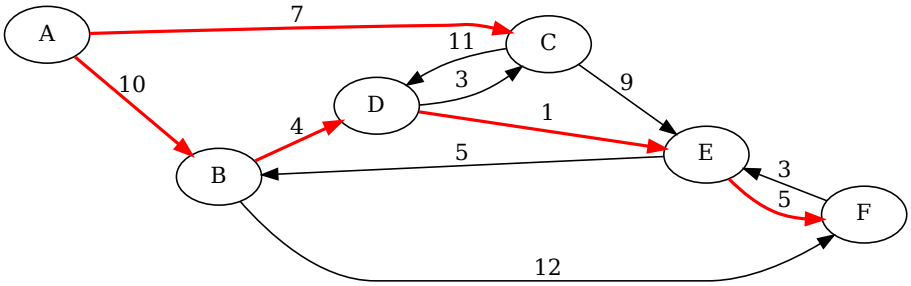
1 Execute Dijkstra's algorithm

Execute Dijkstra's algorithm on the graph below, starting from vertex A. Show the distances calculated and the shortest-path tree from A.



Solution:

Shortest path tree:



Distances:

		A	C	B	D	E	F
A	0						
B	∞	10					
C	∞	7					
D	∞		18	14			
E	∞		16		15		
F	∞			22		20	

2 Make Dijkstra Work

Recall that Dijkstra's algorithm may fail if the input graph contains edges with negative weights. We consider several special cases of graphs with negative weights and try to decide if Dijkstra's algorithm may still be correct in those cases. For the questions below, assume that the graph in question does **not** have negative cycles. We are given a digraph G and want to execute Dijkstra's algorithm starting from s.

1. Is Dijkstra's algorithm correct if we are promised that G is a DAG (with possibly negative weights)?
2. Is Dijkstra's algorithm correct if G contains only one arc of negative weight?
3. Is Dijkstra's algorithm correct if all the negative arcs of G are incident on s?

Solution:

The answer to the first two questions is negative and a counter-example is given in the course slides.

The answer to the last question is, however, positive. Let us prove that if all negative arcs of a digraph are incident on s, then Dijkstra's algorithm executed from s will produce a correct result. We always assume that G has no negative cycles.

First, we observe that arcs going into s are irrelevant. What we mean by this is two things: (i) Dijkstra's algorithm will never use an arc xs in a path (ii) no shortest path $s \to y$ ever uses an arc xs. The second claim follows because if a shortest path $s \to y$ ever uses xs, then s appears twice in the path, which means that the path contains a cycle. Since no cycle has negative weight, the path can be made shorter by removing the cycle (and the arc xs). For the first claim, Dijkstra will only select an arc ij if it creates a shortcut, that is, $\text{dist}[j] > \text{dist}[i] + w(ij)$. For $j = s$ we would need to have $\text{dist}[s] > \text{dist}[i] + w(is)$. Since $\text{dist}[s]$ is initially 0, this would mean that $\text{dist}[i] + w(is) < 0$, which implies that there is a negative weight cycle in the graph, contradiction.

We therefore focus on a digraph G where all negative arcs are coming out of s and prove that Dijkstra correctly produces shortest paths from s. The part of the proof which states that Dijkstra never underestimates the true distance ($\text{dist}(s, v) \leq \text{dist}_D[v]$) still holds, as Dijkstra only updates a distance when a path is found. So what we need to do is check that the second half of the proof also still holds.

As before, we proceed by induction on the number k of vertices extracted from the priority queue. When we extract the first vertex (s), the distance from s to this vertex is correctly calculated. Suppose that we have calculated the correct distances to the first $k-1$ extracted vertices and we just extracted the k-th vertex, u. Let P be the (true) shortest $s \to u$ path, and x_1 the last vertex of this path which we have already extracted. Let x_2 be the vertex immediately after x_1 in the path. Note that it is possible that $s = x_1$ and that $u = x_2$. We have:

- $\text{dist}_D[u] \leq \text{dist}_D[x_2]$ at the time that u was extracted, because we extract the vertex whose current distance is minimum.
- $\text{dist}_D[x_2] \leq \text{dist}(s, x_1) + w(x_1x_2)$, because when we processed x_1 we had $\text{dist}_D[x_1] = \text{dist}(s, x_1)$ (by IH), and we considered the edge x_1x_2 as a possible shortcut to x_2.
- $\text{dist}(s, x_1) + w(x_1x_2) = \text{dist}(s, x_2)$ because x_1 is in the shortest path from s to x_2.
- $\text{dist}(s, x_2) \leq \text{dist}(s, u)$. **This is the part where we use our assumption!** In particular, $x_2 \neq s$ (because x_2 has not been extracted yet), so the arcs of the path $s \to u$ following x_2 are all positive. Since P is a shortest path, its weight is equal to $\text{dist}(s, u)$.

3 Make Bellman-Ford work (less)

As we discussed in class, one possible way to try to speed up the Bellman-Ford algorithm is the following: in each iteration of the main loop, check if any of the distances are modified. If nothing changes, then there is no point in repeating the outer loop, so we can immediately stop. For this exercise we consider this version of the BF algorithm.

1. Give an example where this version of the BF algorithm computes the correct answer in time $O(m)$.
2. Nevertheless, show that this improvement does not affect the worst-case performance of BF by giving an example where even this version takes time $\Omega(mn)$.

Observe that the same graph can be used as an answer to both questions by using a different ordering of the vertices/arcs and you don't even need negative weights. Is the following statement true?

- True or False: For every edge-weighted digraph G there exists an ordering of its **arcs** such that the BF algorithm executes in time $O(m)$.

Solution:

For the first two questions we can use the following simple example: the graph is a directed path with source s and vertices $x_1, x_2, \ldots, x_n$, so the graph only has the arcs $s \to x_1 \to x_2 \to \ldots \to x_n$. All arcs have weight 1.

If we order the arcs in the natural way $sx_1, x_1x_2, x_2x_3, \ldots, x_{n-1}x_n$, the BF algorithm only needs one iteration of the main loop to find the best distances (and one more iteration to verify that the distances have stabilized). If, on the other hand we use the reverse ordering $x_{n-1}x_n, x_{n-2}x_{n-1}, \ldots, x_1x_2, sx_1$, every iteration of the main loop will only improve the distance to one vertex, so we will need n iterations.

The above give us an intuition: if the graph is already ordered in a way that is similar to the shortest-path tree from s, then BF will not need many iterations. We use this intuition to show that the last statement is True. Let T be the shortest-path tree from s in G and order the arcs xy of G in order of increasing distance of x from s **in the tree**. In other words, the ordering starts with arcs coming out of s, then continues with arcs coming out of the children of s, then with arcs coming out of their children, and so on. Order the arcs of G in this way and execute BF. We claim that only one iteration of the main loop is necessary to compute the shortest paths.

We prove this by induction. Initially, we have the correct distance from s. We observe that whenever we process a tree arc xy, such that we have calculated the correct distance to x, we will calculate the correct distance to y, which is a child of x in T. Therefore, once we process all arcs incident on s, we have the correct distances to children of s in T. Now, for the inductive step, we claim that once we process all arcs coming out of vertices of level i we have the correct distances to vertices at level $i + 1$. Hence, one iteration of the main loop suffices.

4 Shortest Simple Paths and Negative Cycles

One annoying aspect of our discussions of graphs with negative weights has been that negative cycles make it complicated to define what exactly is a shortest path (a path may repeat a negative cycle many times, arbitrarily reducing its cost). One way to work around this problem could be to simply forbid this. For this exercise we define the shortest path distance as the weight of the shortest **simple** path, that is, the shortest path that does not repeat any vertex. This is always well-defined, even if the input graph has negative cycles.

Unfortunately, defining the problem in this way makes it impossible to solve! Prove the following:

- If there is a polynomial-time algorithm that takes as input an edge-weighted digraph G and two vertices s, t and computes the simple path from s to t that has minimum weight, then there exists a polynomial time algorithm for the HAMILTONIAN PATH problem.

The HAMILTONIAN PATH problem is the following: given an undirected unweighted graph G, decide if there is a path that visits each vertex of G exactly once. HAMILTONIAN PATH is one of the most famous NP-complete problem and you can take it as a given for this exercise that it admits no polynomial-time algorithm.

Solution:

Consider the following simple reduction: we are given as input an instance $G = (V, E)$ of HAMILTONIAN PATH. Construct a weighted graph by giving each edge of E weight -1. Now, the new graph has two vertices s, t such that the simple $s \to t$ path of minimum cost has cost $\leq -(n-1)$ if and only if G has a Hamiltonian Path. This is not hard to see, as a simple path with this cost must use all vertices of the graph.

We now observe that if we had a polynomial-time algorithm to find the shortest **simple** path of minimum cost, we could use it to decide if a given unweighted graph has a Hamiltonian Path. Since the latter problem cannot be solved in polynomial time (under standard hypotheses), the supposed algorithm does not exist.

5 Constraint Systems

We are given a set of n integer variables $x_1, \ldots, x_n$ and a system of m inequalities, all of which have the form $x_i - x_j \leq b_k$, for some integer b. Our goal is to determine a feasible solution of the system of inequalities, or

correctly decide that none exists.

As an example, consider the following system:

$$\begin{aligned} x_1 - x_2 &\leq 5 \\ x_1 - x_3 &\leq 4 \\ x_2 - x_3 &\leq -3 \\ x_3 - x_1 &\leq 3 \end{aligned}$$

1. Show that the system above has an integer solution.
2. Show that the system no longer has a solution if we replace the last inequality by $x_3 - x_1 \leq -3$.
3. Show that if such a system has a feasible solution $(v_1, v_2, \ldots, v_n)$, then for all offset values d, the solution $(v_1 + d, v_2 + d, \ldots, v_n + d)$ is also valid.
4. Give an efficient algorithm that produces a solution of a given system or decides that none exists.
5. Give a modification of your algorithm which handles the more general problem where some constraints are allowed to involve only one variable, that is, have the form $x_i \leq b_k$ or $-x_i \leq b_k$.

Solution:

For the first question, one solution is $(x_1, x_2, x_3) = (0, -3, 0)$.

For the second question, we observe that if we have the inequalities $x_1 - x_2 \leq 5$, $x_2 - x_3 \leq -3$ and $x_3 - x_1 \leq -3$, summing them gives $0 \leq -1$, which is a contradiction, so no solution exists.

For the third question, $(x_i + d) - (x_j + d) = x_i - x_j$ so the new solution is clearly still valid.

For the fourth question, we construct a directed graph which has a vertex for each variable $x_1, \ldots, x_n$ and add a new vertex s with arcs of weight 0 to everyone. Then, for each inequality $x_i - x_j \leq b_k$ we add an arc $x_j x_i$ of weight b_k. Our algorithm now is to execute Bellman-Ford from s. If we detect a negative cycle we answer that no solution exists. Otherwise, we output as solution the shortest path distances from s.

The complexity of this algorithm is clearly $O(mn)$, as it is dominated by running Bellman-Ford. Let us then discuss its correctness.

First, we claim that if the algorithm reports that no solution exists, this is indeed the case. The algorithm will report this if it discovers a negative cycle $x_{i_1}, x_{i_2}, \ldots, x_{i_h}$ such that $W = w(x_{i_1}x_{i_2}) + w(x_{i_2}x_{i_3}) + \ldots + w(x_{i_h}x_1) < 0$. These arcs exist because we have the inequalities $x_{i_2} - x_{i_1} \leq w(x_{i_1}x_{i_2})$, $x_3 - x_2 \leq w(x_{i_2}x_{i_3})$, $\ldots$, $x_{i_1} - x_{i_h} \leq w(x_{i_h}x_{i_1})$. Summing up these inequalities we have $0 \leq W < 0$ contradiction, so no solution exists.

Second, we claim that if the algorithm reports a solution, then the solution is feasible. Observe that in this case the graph contains no negative cycle, so distances from s are well-defined. Take any constraint, which must have the form $x_i - x_j \leq w(x_j x_i) \Leftrightarrow x_i \leq x_j + w(x_j x_i)$. We claim that our solution is feasible, that is, $\text{dist}(s, x_i) \leq \text{dist}(s, x_j) + w(x_j x_i)$. This follows from the properties of shortest paths.

For the final question we follow the same strategy, except s plays the role of a variable x_0 which is always set to 0. So, the constraint $x_i \leq b_k$ is represented by an arc sx_i of weight b_k; and the constraint $-x_i \leq b_k$ is represented by an arc $x_i s$ of weight b_k. Note that if we have multiple parallel arcs, we keep the one of minimum weight.

We now execute BF from s as before and observe that the correctness arguments remain essentially unchanged. In particular, if a negative cycle exists we can prove that no solution exists by summing up the relevant constraints; while otherwise the distance from s to s will be calculated to 0, implying that the constraints of the new types are also satisfied.