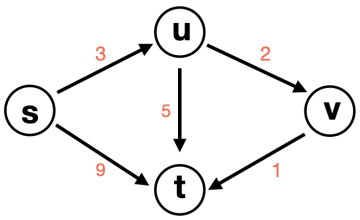
Lab 13: Shortest Paths

Collaboration level 0 (no restrictions). Open notes.

1. Step through `Dijkstra(G, s, t)` on the graph shown below. Complete the table below to show what the arrays $d[]$ and $p[]$ (parent) are at each step of the algorithm. Indicate what vertices are explored (the first, second, third and fourth).

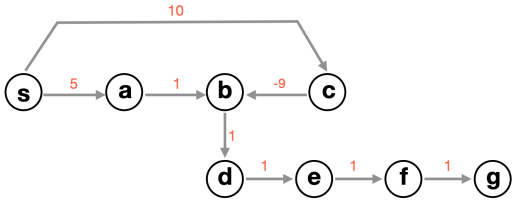


	d[s]	d[u]	d[v]	d[t]	p[s]	p[u]	p[v]	p[t]
Initially the state is	0	∞	∞	∞	null	null	null	null
immediately after the first vertex is explored	0	3	∞	9	null	s	null	s
immediately after the second vertex is explored								
immediately after the third vertex is explored								
immediately after the last vertex is explored								

2. Consider the directed graph below and assume you want to compute SSSP(s).

   Note that the graph has a cycle, and an edge with negative weight.

   Run Dijkstra's algorithm on the graph above step by step. Are there any vertices for which d[x] is correct? Are there any vertices for which d[x] is incorrect? Why?

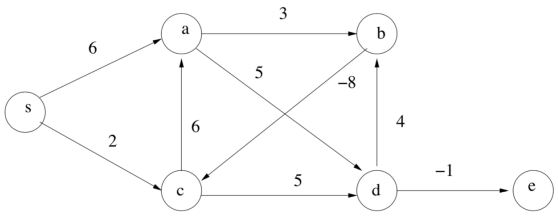


	dist of explored vertex	d[s]	d[a]	d[b]	d[c]	d[d]	d[e]	d[f]	d[g]
Initially the state is		0	∞	∞	∞	∞	∞	∞	∞
immediately after the first vertex is									
immediately after the second vertex is									
immediately after the third vertex is									

3. Prove that the following claim is false by showing a counterexample:

   Claim: Let $G = (V, E)$ be a directed graph with negative-weight edges, but no negative-weight cycles. Let $w, w < 0$, be the smallest weight in G. Then one can compute SSSP in the following way: transform G into a graph with all positive weights by adding $-w$ to all edges, run Dijkstra, and subtract from each shortest path the corresponding number of edges times $-w$. Thus, SSSP can be solved by Dijkstra's algorithm even on graph with negative weights.

4. Cosider the directed graph below and assume you want to compute SSSP(s).



Note that the graph has a cycle (so we cannot use SSSP on DAGs), and not all edge weights are positive (so we cannot use Dijkstra's algorithm).

(a) Compute SSSP(s) by hand and show the SSSP-tree. Let's keep this for reference.

(b) Consider the following algorithm:

- initialize: set $d[s] = 0$ and $d[v] = \infty$ for all other vertices v
- do one round of edge relaxation

Consider the following edge order:

$$(s,a),(s,c),(a,b),(a,d),(b,c),(c,a),(c,d),(d,b),(d,e)$$

What are the distances $d[v]$ at the end of the round? Does this compute SSSP(s)?

(c) After one round of edge relaxation as above, you run a second round (in the same order). What happens?

(d) Consider the same algorithm, but with a different edge order:

$$(d,b),(d,e),(c,d),(b,c),(c,a),(a,b),(a,d),(s,a),(s,c)$$

What are the distances $d[v]$ at the end of the round? For what vertices x is $d[x]$ correct?

(e) Perform a second round, same order. At the end of the second round, for what vertices x is $d[x]$ correct?

(f) Perform a third round, same order. At the end of the third round, for what vertices x is $d[x]$ correct?

(g) Perform a fourth round, same order. At the end of the fourth round, for what vertices x is $d[x]$ correct?

(h) Perform a fifth round, same order. At the end of the fifth round, for what vertices x is $d[x]$ correct?

(i) Perform a sixth round, same order. At the end of the sixth round, what vertices x have changed their $d[x]$? What will happen if you run further rounds?

5. Ater working throuh the previous problem, you have discovered that the following algorithm, if run for a sufficient number of rounds k, will compute SSSP(s):

   Algorithm:

   - initialize: set $d[s] = 0$ and $d[v] = \infty$ for all other vertices v
   - for $i = 1$ to k do: perform one round of edge relaxation

   Reminder that by one round of relaxation we mean that *all* edges in the graph are relaxed (in some arbitrary order).

   Fill in the sentences below so that they are true:

   (a) After one round of edge relaxation, it is guaranteed that $d[x] = \delta(s, x)$ for all vertices x whose shortest paths from s consist of ..... edges.

   (b) After i rounds of edge relaxation, it is guaranteed that $d[x] = \delta(s, x)$ for all vertices x whose shortest paths from s consist of ..... edges.

   Based on this, how many rounds k will be sufficient to find any possible shortest path?

   (Hint: What is the largest number of edges on a shortest path?)

After working through the previous problem, you have discovered Bellman-Ford's algorithm!

Algorithm Bellman-Ford (vertex s)

- Initialize: for each $v \in V$ then: $d[v] = \infty$, $pred[v] = NULL$, $d[s] = 0$

  //do $|V| - 1$ rounds of edge relaxation
- For $k = 1$ to $|V| - 1$:

  for each edge $e = (u, v) \in E$: Relax (u, v)

6. Give example of a graph G=(V,E) with an arbitrary number of vertices for which one round of relaxation in Bellman-Ford algorithm is always sufficient, no matter the order in which the edges are relaxed.

7. Give example of a graph G=(V,E) with an arbitrary number of vertices for which $|V| - 1$ rounds of relaxation in Bellman-Ford algorithm are necessary in the worst case (for a worst-case ordering of edges).

8. Bellman-Ford as written above performs $|V| - 1$ rounds, no matter what the graph is. It is written like this for clarity. Show how to update it so that it performs only as many rounds as necessary.

9. What happens if we let Bellman-Ford run for an additional round?

10. **Arbitrage:** Suppose the various economies of the world use a set of currencies $C_1, C_2, ..., C_n$ –think of these as dollars pounds, bitcoins, etc. For each ordered pair (C_i, C_j), the bank lets you trade one unit of C_i to r_{ij} units of C_j. This can be modeled as a graph with the vertices being the currencies, and the edges being the exchange rates r_{ij}. This is a complete directed graph with exactly $n \times (n - 1)$ edges.

    (a) Devise an efficient algorithm which, given all exchange rates r_{ij}, a starting currency C_s, target currency C_t, computes a sequence of exchanges that results in the greatest amount of C_t. What is the run time of your algorithm?

    Hint: Model this as a shortest-path problem, with the cost of a path and relaxing an edge appropriately adjusted. The question then is: Dijkstra or Bellman-Ford? Try to step through the correctness proof of Dijkstra's algorithm and see what the assumption of $w > 0$ would be in this case (edges weights are multiplied rather than added and we want to maximize rather than minimize).

    (b) Due to fluctuations in the markets, it is occasionally possible to find a sequence of exchanges that lets you start with currency A, change into currencies B, C, D, etc.. and then end up changing back to A with more money than you started (this is called *arbitrage*). Come up with an algorithm that, given a set of currencies and the exchange rates r_{ij} between them, determines if arbitrage is possible.
    Hint: Express this in terms of the graph having a certain type of cycle