Week 8 Tutorial Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to the tutorial problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, pseudocode may be provided where it illustrates a particular useful concept.

Implementation checklist

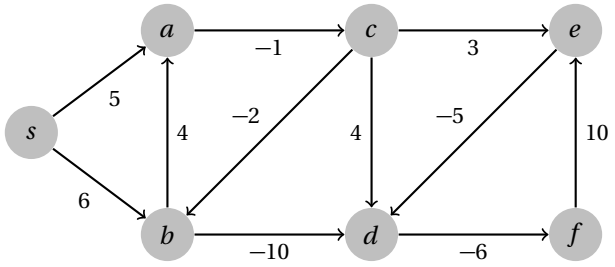
It will be most beneficial for your learning if you have completed this checklist **before** the tutorial.

By the end of week 8, write Python code for:

- Bellman-Ford single source shortest paths.
- Floyd-Warshall all-pairs shortest paths.

Tutorial Problems

Problem 1. (Preparation) Use Bellman-Ford to determine the shortest paths from vertex s to all other vertices in this graph. Afterwards, indicate to which vertices s has a well defined shortest path, and which do not by indicating the distance as $-\infty$. Draw the resulting shortest path tree containing the vertices with well defined shortest paths. For consistency, you should relax the edges in the following order: $s \to a$, $s \to b$, $a \to c$, $b \to a$, $b \to d$, $c \to b$, $c \to d$, $c \to e$, $d \to f$, $e \to d$ and $f \to e$.



Solution

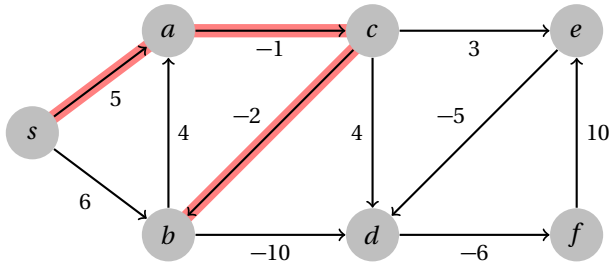
The distances at each iteration are shown below. If you followed the order specified, you should have the same distances. Relaxing the edges in a different order may lead to a different table, but the end distances should be the same (except for the vertices reachable via negative cycles).

Vertex	Iteration						
	0	1	2	3	4	5	6
s	0	0	0	0	0	0	0
a	∞	5	5	5	5	5	5
b	∞	2	2	2	2	2	2
c	∞	4	4	4	4	4	4
d	∞	-4	-8	-9	-9	-10	-10
e	∞	0	-4	-4	-5	-5	-6
f	∞	-10	-14	-14	-15	-15	-16

Since another round of relaxation would decrease the distance of vertex d, it must be reachable via a negative cycle. All of the vertices reachable from d therefore have undefined distance estimates. The final distances are therefore

s	a	b	c	d	e	f
0	5	2	4	$-\infty$	$-\infty$	$-\infty$

The shortest path tree is shown below



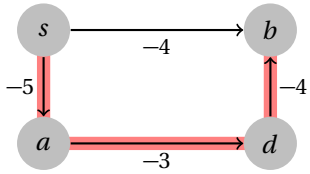
Problem 2. Consider the following algorithm for single-source shortest paths on a graph with negative weights:

- Find the minimum weight edge in the graph, say it has weight w
- Subtract w from the weight of every edge in the graph. The graph now has no negative weights
- Run Dijkstra's algorithm on the modified graph
- Add w back to the weight of the edges and compute the lengths of the resulting shortest paths

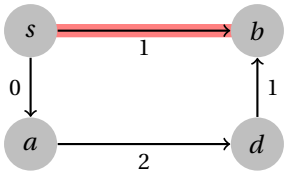
Prove by giving a counterexample that this algorithm is incorrect.

Solution

The problem with this algorithm is that it changes the length of paths with more edges a larger amount than it changes the lengths of paths with fewer edges, since it adds a constant amount to every edge weight. A good graph to break this would therefore be one where a shortest path has more edges than an alternative.



In this case, the shortest path from s to b is $s \to a \to d \to b$ with length -12. If we subtract -5 from every edge weight in the graph, then it becomes the following, where the shortest path is now $s \to b$, so we get the wrong answer.



Problem 3. Describe an algorithm that given a graph G determines whether or not G contains a negative cycle. Your algorithm should run in $O(VE)$ time.

Solution

Remember that the Bellman-Ford algorithm can detect whether there is a negative cycle that is reachable from the source vertex. The obvious tempting solution is to just run Bellman-Ford from every possible source vertex and see whether a negative cycle is detected. This would be very slow though, taking $O(V^2E)$ time. Instead, we would rather run just one Bellman-Ford, but how can we be sure that we can definitely reach the negative cycle if one exists? The easiest way to ensure this is to simply add a new vertex to the graph, and add an edge from that vertex to every other vertex in the graph. Running Bellman-Ford on this vertex will then definitely visit every vertex, and hence will definitely detect a negative cycle if one is present. Since we only need to run Bellman-Ford once, this algorithm takes $O(VE)$ time.

Problem 4. Improve the Floyd-Warshall algorithm so that you can also reconstruct the shortest paths in addition to the distances. Your improvement should not worsen the time complexity of Floyd-Warshall, and you should be able to reconstruct a path of length k in $O(k)$ time.

Solution

There are many approaches to make this work.

Solution 1: Track the intermediate vertex

The simplest way to keep track of the path information is to record, for each pair of vertices u, v, which intermediate vertex was optimal for going between them. This can be tracked during the algorithm like so.

```
1:  function FLOYD_WARSHALL(G = (V, E))
2:      Set dist[1..n][1..n] = ∞
3:      Set dist[v][v] = 0 for all vertices v
4:      Set dist[u][v] = w(u,v) for all edges e = (u,v) in E
5:      Set mid[1..n][1..n] = null    // mid[u][v] = optimal intermediate vertex
6:      for each vertex k = 1 to n do
7:          for each vertex u = 1 to n do
8:              for each vertex v = 1 to n do
9:                  if dist[u][k] + dist[k][v] < dist[u][v] then
10:                     dist[u][v] = dist[u][k] + dist[k][v]
```

```
                    mid[u][v] = k
                end if
            end for
        end for
    end for
    return dist[1..n][1..n], mid[1..n][1..n]
end function
```

For each pair of nodes u and v that have a finite distance, to reconstruct the path, we then need to do a sort of divide-and-conquer style path reconstruction.

```
function GET_PATH(u, v)
    if mid[u][v] = null then
        return [u, v]
    else
        left = GET_PATH(u, mid[u][v])
        right = GET_PATH(mid[u][v], v)
        return left.pop_back() + right    // Remove the duplicate from the middle
    end if
end function
```

Solution 2: Track the successor

Another slightly cleaner solution is to remember for each pair u, v, what is the first vertex on a shortest path from u to v. To maintain this, if we update the pair u, v and decide that vertex k is the new best intermediate vertex, then the first vertex on a shortest path from u to v is just the first vertex on a shortest path from u to k. We can implement this as follows.

```
function FLOYD_WARSHALL(G = (V, E))
    Set dist[1..n][1..n] = ∞
    Set dist[v][v] = 0 for all vertices v
    Set dist[u][v] = w(u,v) for all edges e = (u,v) in E
    Set succ[1..n][1..n] = null    // succ[u][v] = successor of u on shortest path to v
    Set succ[u][v] = v for all edges e = (u,v) in E
    for each vertex k = 1 to n do
        for each vertex u = 1 to n do
            for each vertex v = 1 to n do
                if dist[u][k] + dist[k][v] < dist[u][v] then
                    dist[u][v] = dist[u][k] + dist[k][v]
                    succ[u][v] = succ[u][k]
                end if
            end for
        end for
    end for
    return dist[1..n][1..n], succ[1..n][1..n]
end function
```

Reconstructing the paths is now easier as we do not need any fancy recursion.

```
function GET_PATH(u, v)
    Set path = [u]
    while u ≠ v do
        u = succ[u][v]
        path.append(u)
```

```
    end while
end function
```

In both cases, we add only a constant amount of overhead to the algorithm so we do not worsen the complexity. We can also reconstruct paths of length k in $O(k)$ time in both cases.

Problem 5. Given a graph that contains a negative weight cycle, give an algorithm to determine the vertices of one such cycle. Your algorithm should run in $O(VE)$ time.

Solution

Recall from Problem 3 that we can use Bellman-Ford to detect the presence of a negative cycle by adding a new source vertex that connects to all others. Let's start by doing this. Remember that when Bellman-Ford terminates, the predecessors will all form a shortest path tree, except for the vertices reachable from a negative cycle. These predecessors may not form a tree, but may actually form cycles since they will point to the vertices that caused them to relax most recently. To guarantee that the predecessors form a cycle, we will run Bellman-Ford for $|V|$ iterations (instead of $|V|-1$) since the extra relaxation will cause the infinite cycle to propagate all the way around.

To produce a negative cycle, we therefore begin at an arbitrary vertex that is reachable from a negative cycle. Such a vertex will be any one that was relaxed during iteration $|V|$ of Bellman-Ford. This vertex may not necessarily be **in** a negative cycle, as it could be on a path that simply travels through one and then leaves it. In order to get to the negative cycle that caused this, we should travel backwards along the predecessors. Since there are at most $|V|$ vertices in any path, we can just travel back $|V|$ times. This will guarantee that we enter whatever cycle caused this vertex to have an undefined shortest path.

Once we are inside the cycle, we can then simply backtrack around it until we find the first vertex again, at which point we stop and return the sequence that we found. Some pseudocode is shown below.

```
function FIND_NEGATIVE_CYCLE(G = (V, E))
    s = G.add_vertex()    // Add a new vertex to G
    for each vertex v = 1 to n do
        G.add_edge(s, v, w = 0)
    end for
    dist[1..n], pred[1..n] = BELLMAN_FORD(G, s)    // Run for |V| iterations to guarantee cycle
    Let v = any vertex that was relaxed during iteration |V|
    for i = 1 to n do
        v = pred[v]    // Backtrack into the cycle
    end for
    cycle = [v]
    u = pred[v]
    while u ≠ v do
        cycle.append(u)
        u = pred[u]
    end while
    return reverse(cycle)    // We built the cycle backwards so reverse it
end function
```

Problem 6. Add a post-processing step to Floyd-Warshall that sets $\text{dist}[u][v] = -\infty$ if there is an arbitrarily short path between vertex u and vertex v, i.e. if u can travel to v via a negative weight cycle. Your post processing should run in $O(V^3)$ time or better, i.e. it should not worsen the complexity of the overall algorithm.

Solution

Recall that the Floyd-Warshall algorithm can detect the presence of negative cycles by looking at the distances of vertices to themselves. If a vertex can reach itself with a negative distance, then it must be contained in a negative cycle. We can therefore post process the graph by checking each intermediate vertex k that is part of a negative cycle, and then setting the lengths of all paths $u \rightsquigarrow v$ such that u can reach k and k can reach v to $-\infty$. This process takes $O(V^3)$. See the pseudocode below.

```
1: function FLOYD_WARSHALL_POSTPROCESS(G = (V, E), dist[1..n])
2:     for each vertex k = 1 to n do
3:         if dist[k][k] < 0 then
4:             for each vertex u = 1 to n do
5:                 for each vertex v = 1 to n do
6:                     if dist[u][k] + dist[k][v] < ∞ then
7:                         dist[u][v] = −∞
8:                     end if
9:                 end for
10:            end for
11:        end if
12:    end for
13: end function
```

Problem 7. Arbitrage is the process of exploiting conversion rates between commodities to make a profit. Arbitrage may occur over many steps. For example, one could purchase US dollars, convert it into Great British Pounds, and then back into Australian dollars. If the prices were right, this could result in a profit. Given a list of currencies and the best conversion rate between each pair, devise an algorithm that determines whether arbitrage is possible, i.e. whether or not you could make a profit. Your algorithm should run in $O(n^3)$ where n is the number of currencies.

Solution

Let's imagine this as a shortest path problem. We wish to begin at one currency and walk through a path containing some other currencies before arriving back at our initial currency with a net profit. Making a profit means that the net conversion rate between a currency and itself should be less than 1. Note that if we convert between currencies whose exchange rates are $r_1, r_2, ..., r_k$, then the final conversion rate is $r_1 \times r_2 \times ... \times r_k$. One way to model this problem is to therefore create a graph on n vertices where each vertex represents a currency. We add a directed edge between a pair of vertices with the exchange rate as the weight. We could then modify the Floyd-Warshall algorithm to compute the product of the weights rather than the sum, and it would then find for us the best conversion rate between any pair of currencies. Arbitrage is possible if a currency can be converted into itself at a rate of less than 1, i.e. if we find an entry on the diagonal of the distance matrix that is less than 1.

An alternate solution that does not require us to modify Floyd-Warshall is to create the same graph but use the logarithms of the exchange rates as the edge weights. We can then run the ordinary Floyd-Warshall algorithm. This works because $\log(r_1) + \log(r_2) = \log(r_1 \times r_2)$. Since $\log(r) < 0$ if and only if $r < 1$, we know that arbitrage is possible if and only if the graph contained a negative cycle, i.e. if a vertex has a negative distance to itself.

Supplementary Problems

Problem 8. Implement Bellman-Ford and test your code's correctness by solving the following problem on UVA Online Judge: `https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=499`.

Problem 9. Devise an algorithm for counting the number of paths between two given vertices s and t in a directed acyclic graph which runs in $O(V+E)$. Don't worry about the magnitude of the answer (i.e. assume that all arithmetic operations take constant time, despite the fact that the answer might be exponential in the input size.)

Solution

We will approach this using a dynamic programming algorithm. Suppose we are at some vertex u. Then the number of different ways in which we could get to t involves trying all of our different edges (u, v) and then counting the number of resulting paths from all of the v. So let us define the following subproblems.

$$\texttt{DP}[u] = \{\text{The number of paths from } u \text{ to } t\}.$$

The recurrence is then given by

$$\texttt{DP}[u] = \begin{cases} 1 & \text{if } u = t, \\ \displaystyle\sum_{v \in \text{adj}[u]} \texttt{DP}[v] & \text{otherwise.} \end{cases}$$

The answer is the value of $\texttt{DP}[s]$. The subproblems are dependent in a reverse topological order, since in order to compute $\texttt{DP}[u]$ we must know the value of all of u's descendants.

```
1:  function COUNT_PATHS(G = (V, E), s, t)
2:      Set count[1..n] = 0
3:      count[t] = 1
4:      Set order = reverse(topological_sort(G))
5:      for each vertex u in order do
6:          for each edge (u, v) adjacent to u do
7:              count[u] = count[u] + count[v]
8:          end for
9:      end for
10:     return count[s]
11: end function
```

Problem 10. (Advanced) An improvement to the Bellman-Ford algorithm is the so-called "Shortest Paths Faster Algorithm", or SPFA. SPFA works like Bellman-Ford, relaxing edges until no more improvements are made, but instead of relaxing every edge $|V|-1$ times, we maintain a queue of vertices that have been relaxed. At each iteration, we take an item from the queue, relax each of its outgoing edges, and add any vertices whose distances changed to the queue if they are not in it already. The queue initially contains the source vertex s.

(a) Reason why SPFA is correct if the graph contains no negative-weight cycles.

(b) The algorithm described above will cycle forever in the presence of a negative-weight cycle. Add a simple condition to the algorithm to fix this and detect the presence of negative cycle if encountered.

(c) What is the worst-case time complexity of SPFA?

Solution

The proof of correctness for SPFA is pretty much exactly the same as for Bellman-Ford, so we will not write the whole thing formally. At the first iteration, all shortest paths consisting of at most 1 edge will be found by relaxing the edges out of s. Subsequently, all shortest paths of length at most two will be found when those vertices relax their outgoing edges, and so on. By induction, we can show that all shortest paths of length k will be relaxed after the shortest paths of length $k-1$. SPFA essentially does the same thing as Bellman-Ford, but it avoids relaxing edges that will not make a difference.

In the presence of a negative weight cycle, SPFA will continue to add the vertices of the cycle to the queue forever. To stop this, we enforce that each vertex can only be removed from the queue at most $n-1$ times. If a vertex is removed n times, this implies that the shortest path to it contains $n+1$ vertices since the queue always serves vertices in order of the number of edges on their shortest path, and in between multiple occurrences of the same vertex, the path length must increase.

Since we may remove each vertex from the queue n times, and each time we process every one of its edges, the amount of work done is at most $O(VE)$, the same as Bellman-Ford. Although this sounds bad, on average SPFA performs much better.

Problem 11. (Advanced) In this problem, we will derive a fast algorithm for the all-pairs shortest path problem on sparse graphs with negative weights[1]. We will assume for simplicity that the graph has no negative cycles, but the techniques here can easily be adapted to handle them. The key ingredient of the algorithm is based on the idea of *potentials*.

(a) Let $d[v] : v \in V$ be the distances to each vertex $v \in V$ from an arbitrary source vertex. Prove that for each edge $(u,v) \in E$

$$d[u] - d[v] + w(u,v) \geq 0.$$

We define the quantity $p[u] = d[u]$ as the *potential* of the vertex u.

(b) Give a simple algorithm to compute a set of valid potentials. Potentials should not be infinity for any vertex.

Suppose now that we modify the weight of every edge (u,v) in the graph to

$$w'(u,v) = p[u] - p[v] + w(u,v)$$

(c) Prove that a shortest path in the modified graph was also a shortest path in the original, unmodified graph.

(d) Deduce that we can solve the all-pairs shortest path problem on sparse graphs with negative weights in $O(V^2 \log(V))$ time.

Solution

(a) This is a direct result of the triangle inequality. The triangle inequality states that

$$d[u] + w(u,v) \geq d[v],$$

which must be true since if this quantity were less than $d[v]$, then the path $s \rightsquigarrow u \rightarrow v$ would be shorter than the shortest path to v, a contradiction. Subtracting $d[v]$ from both sides of the triangle inequality yields the desired result.

(b) A solution that almost works is to pick an arbitrary source vertex and run Bellman-Ford to compute the shortest distances. This will fail though if the source we pick cannot reach every vertex since then some of the distances will be infinity. To correct this, we instead add a new vertex to the graph and connect it to every vertex with an edge weight of zero. We then run Bellman-Ford from this new vertex. By design, it is connected to every other vertex so none of the distances will be infinity. The resulting distances can be used as potentials.

(c) Consider a path in the modified graph $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_k$. Its total length will be

$$p[v_1] - p[v_2] + w(v_1,v_2) + p[v_2] - p[v_3] + w(v_2,v_3) + ... + p[v_{k-1}] - p[v_k] + w(v_{k-1},v_k).$$

Observe that except for the first and last, the potentials all cancel since they appear as a positive term and then as a negative term immediately after. This leaves us with

$$p[v_1] + w(v_1,v_2) + w(v_2,v_3) + ...{}_+ w(v_{k-1},v_k) - p[v_k].$$

[1] This algorithm is known as Johnson's algorithm.

Note that the only potentials left in the equation are of the first and last vertex, in other words, they do not depend on the path taken at all. The remaining terms simply constitute the path length of the path $v_1 \to v_2 \to ... \to v_k$ in the original, unmodified graph, hence any shortest path in the modified graph is also a shortest path in the unmodified graph.

(d) With all of these ingredients, we can derive *Johnson's algorithm*. First, we run Bellman-Ford from a new source vertex that connects to every other vertex in order to obtain a set of valid potentials. We then create the new graph by modifying the edge weights as above. Now since all of the edge weights are nonnegative by (a), we can use Dijkstra's algorithm to compute shortest paths from every possible source vertex. Finally we can recover the true shortest paths by subtracting away the potentials that we added. One invocation of Bellman-Ford costs $O(VE)$ and V Dijkstra's costs $O(VE\log(V))$, and hence the total cost is $O(VE\log(V))$, which is $O(V^2\log(V))$ in sparse graphs.

Problem 12. (Advanced) Consider a directed acyclic graph G where each edge is labelled with a character from some finite alphabet A. Given a string S over the alphabet A, count the number of paths in G whose edge labels spell out the string S. Your algorithm should run in $O((V+E)n)$, where n is the length of the string S.

Solution

This problem can be solved very similarly to the ordinary path-counting problem on a DAG (Problem 9). We will use dynamic programming. Let us define the subproblems

$$\texttt{DP}[u,i] = \{\text{The number of paths from } u \text{ that spell out } S[i..n]\}$$

If we are at a particular vertex u and we want to write $S[i..n]$, then we need to find all of our outgoing edges (u,v) that are labelled $S[i]$, and then try to write $S[i+1..n]$ from vertex v. A recurrence can therefore be written as follows.

$$\texttt{DP}[u,i] = \begin{cases} 1 & \text{if } i = n+1, \\ \displaystyle\sum_{\substack{v \in \text{adj}[u] \\ l(u,v)=S[i]}} \texttt{DP}[v,i+1] & \text{otherwise} \end{cases}$$

where $l(u,v)$ denotes the label of the edge (u,v). The solution to the problem is then the sum of $\texttt{DP}[u,1]$ for all vertices $u \in V$. We have Vn subproblems, and for each edge in the graph, it gets processed at most once per value of i, so the total time complexity is $O(Vn+En) = O((V+E)n)$ as required.

Problem 13. (Advanced) Describe how an instance of the unbounded knapsack problem can be converted into a corresponding directed acyclic graph. Which graph problem correctly models the unbounded knapsack problem on this graph?

Solution

Create a graph with $C+1$ vertices, labelled 0 to C, where C is the capacity of the knapsack. For each item available with weight w and value v, create an edge from every vertex $c \geq w$ to the vertex $c-w$ with weight v. The solution to the unbounded knapsack problem is the critical path of this graph.

Problem 14. (Advanced) A problem closely related to the transitive closure problem mentioned in lectures is the *transitive reduction*. In a sense, the transitive reduction is the opposite of the transitive closure. It is a directed graph with the fewest possible edges that has the same reachability as the original graph. In other words, for all pairs of vertices u and v, there is a path between u and v in the transitive reduction if and only if there is a path between u and v in the original graph. Give an algorithm for computing the transitive reduction of a directed acyclic graph. Your algorithm should run in $O(V^2+VE)$ time.

Solution

As usual, we will assume that G is a simple graph. If it is not, we can simply remove duplicate edges and loops at the beginning since they contribute nothing to the reachability. Consider each edge (u, v) in the graph G. If (u, v) is the only way to travel from u to v, then clearly we must keep it. Otherwise, if there is some other way to get from u to v, since the graph is simple, there must be a path consisting of multiple edges that leads from u to v. If this is the case, removing (u, v) does not harm the reachability, and since the other vertices on the $u \rightsquigarrow v$ path must maintain their respective reachabilities, we should remove (u, v), rather than removing anything on that path.

Given this, our algorithm is as follows. Let's use the critical path dynamic programming algorithm to compute the longest paths between every pair of vertices in the graph. Computing the longest paths that start at a particular vertex takes $O(V + E)$, so doing this from every vertex takes $O(V^2 + VE)$ time. Then, for each edge (u, v), we decide to keep it if the longest path from u to v is length 1 (i.e. just this edge), otherwise we delete it. This algorithm takes $O(V^2 + VE)$ time in total.