# Numerics

**Spring 2011 Final, Problem 3c.** Compute $\sqrt[3]{6006}$ using two iterations of Newton's method, i.e., fill out the following table. Your entry for $x_1$ should be fully simplified. Your entry for $x_2$ can be left unsimplified.

| $i$ | $x_i$ |
|---|---|
| 0 | 1 |
| 1 | |
| 2 | |

**Solution:** The function we use is $f(x) = x^3 - 6006$. As a result, we may write our formula for $x_{i+1}$ as follows:

$$x_{i+1} = x_i - \frac{x_i^3 - 6006}{3x_i^2} = x_i - \frac{1}{3} \cdot \left( x_i - \frac{6006}{x_i^2} \right) = \frac{2}{3} \cdot x_i + \frac{1}{3} \cdot \frac{6006}{x_i^2}$$

When we plug in $x_0 = 1$, we get $x_1 = \frac{2}{3} + \frac{1}{3} \cdot 6006 = 2002 + \frac{2}{3}$. We do not need to simplify the formula for $x_2$, so we can simply substitute in the value for $x_1$ in the above equation.

**Spring 2010 Final, Problem 2i.** Suppose you want to estimate the value of $1000/7$. Show that if the starting value is $x_0 = 10^9$ in Newton's method for division, then you will never get to a value that is accurate to 6 decimal places.

**Solution:** Newton's method says that the error term $\epsilon$ is squared with every iteration. The value $x_0 = 10^9$ has an error term $\epsilon > 1000$, so the guesses produced will diverge, instead of converging to the desired result.

**Spring 2007 Final, Problem 7c.** True or False? Newton's Method for computing $\sqrt{2}$ essentially squares the number of correct digits at each iteration. Explain your answer.

**Solution:** False. Newton's method squares the error term with every iteration; the number of correct digits will double with each iteration as long as the initial error term is low.

# BFS, DFS, and Topological Sort

**Spring 2010 Final, Problem 3.** Given an undirected, connected graph $G = (V, E)$, a *bridge* in the graph is an edge whose removal would break the graph into two pieces. (In other words, a bridge is an edge $e \in E$ such that $G = (V, E)$ is connected but $G' = (V, E - \{e\})$ is disconnected.)

**(a)** Show that every bridge must appear in every BFS tree of the graph.

**(b)** Show that for any edge $e$, you can test whether $e$ is a bridge or not in $O(E)$ time.

**(c)** Combine parts **(a)** and **(b)** to design an algorithm that finds *all* bridges in $G$ in $O(V \cdot E)$ time.

**Solution:**

**(a)** In a connected graph, BFS starting from any node $s$ will reach all nodes. Therefore, for any pair of nodes $(u, v)$ in the graph, there is a path in the BFS tree from $s$ to $u$ and a path in the BFS tree from $s$ to $v$. As a consequence, the BFS tree connects any pair of nodes $(u, v)$. So if we remove all the edges except for the ones in the BFS tree, then the graph will still be connected. As a result, all of the bridges must be in the BFS tree.

**(b)** Remove $e = \{u, v\}$ from $G$ and perform BFS from $u$. This will take time $O(E)$ in total to build each of the level sets. We can then check whether the graph is connected by iterating through the vertices in $V$. As soon as we discover a single vertex that is not connected, we can halt and return a value. As a result, we will never examine more than one vertex that is not contained in the level set, so this check will not adversely affect the runtime.

**(c)** Perform BFS to generate a BFS tree. Then we know that any bridge must be contained in the BFS tree. There are $|V| - 1$ edges in the BFS tree, so we can perform $O(V)$ checks, each of which takes time $O(E)$, to figure out which of those edges are the bridges.

**Fall 2008 Final, Problem 5.** Consider two vertices, $s$ and $t$, in some directed acyclic graph $G = (V, E)$. Give an efficient algorithm to determine whether the number of paths in $G$ from $s$ to $t$ is odd or even. Analyze its running time in terms of $|V|$ and $|E|$.

**Solution:** To solve this problem, we will actually solve a more general problem: computing the total number of paths from $s$ to $t$. To do so, we will examine the vertices in topological order. If a vertex can be reached using $k$ distinct paths, then each of its successors can be reached by taking one of those $k$ paths and then following the last edge from the vertex to its successor. We can use this technique to generate path counts in the following way:

COUNT-DAG-PATHS$(G = (V, E), s, t)$

```
1  ⟨v_1, ..., v_n⟩ = TOPOLOGICAL-SORT(V)
2  initialize dictionary count
3  for i = 1 to n
4      count[v_i] = 0
5  count[s] = 1
6  for i = 1 to n
7      for each u ∈ adj[v_i]
8          count[u] = count[u] + count[v_i]
9  return count[t]
```

**Spring 2010 Quiz 2, Problem 6.** You are at an airport in a foreign city and would like to choose a hotel that has the maximum number of shortest paths from the airport (so that you reduce the risk of getting lost). Suppose you are given a city map with unit distance between each pair of directly connected locations. Design an $O(V + E)$-time algorithm that finds the number of shortest paths between the airport (the source vertex $s$) and the hotel (the target vertex $t$).

**Solution:** To make this work, we perform a modified version of BFS, somewhat inspired by the results of the previous problem. Any shortest path to a node on the $i$th level of BFS must be a length $i$ path, and so each edge in the path must cross between different levels of the BFS. Hence, if there are $k$ paths into a node $v$, then any nodes adjacent to $v$ in the *next* level of BFS will be reachable by $k$ paths passing through $v$. We may write an algorithm to compute these path counts as follows:

COUNT-SHORTEST-PATHS$(G = (V, E), s, t)$

```
1  initialize dictionary count
2  for each v ∈ V
3      count[v] = 0
4  initialize dictionary level
5  level[s] = 0
6  initialize set frontier = {s}
7  while frontier is not empty
8      initialize set next = ∅
9      for each u ∈ frontier
10         for each v ∈ adj[u]
11             if v is not in level
12                 level[v] = level[u] + 1
13                 next = next ∪ {v}
14             if level[v] is equal to level[u] + 1
15                 count[v] = count[v] + count[u]
16     frontier = next
17 return count[t]
```

**Fall 2007 Quiz 2, Problem 3.** Another way of performing topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0 (no incoming edges), output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if $G$ has cycles?

**Solution:** To make sure that we can make this algorithm run in $O(V + E)$ time, we need a way to efficiently figure out which vertices have no more incoming edges. To that end, we store a dictionary $incoming$ giving the count of incoming edges for each node and a set $zero$ containing all of the nodes with $incoming[v] = 0$. We update $incoming[v]$ every time we process a node $u$ with an edge $(u, v)$, and as soon as $incoming[v]$ becomes 0, we add it to the set $zero$.

```
TOPOLOGICAL-SORT(G = (V, E))
 1  initialize dictionary incoming
 2  for all vertices u ∈ V
 3      incoming[u] = 0
 4  for all vertices u ∈ V
 5      for all vertices v ∈ adj[u]
 6          incoming[v] = incoming[v] + 1
 7  initialize set zero = ∅
 8  for all vertices u ∈ V
 9      if incoming[u] is equal to 0
10          zero = zero ∪ {u}
11  initialize list result = []
12  while zero is not empty
13      newZero = ∅
14      for u in zero
15          LIST-APPEND(result, u)
16          for all vertices v ∈ adj[u]
17              incoming[v] = incoming[v] − 1
18              if incoming[v] is equal to 0
19                  newZero = newZero ∪ {v}
20      zero = newZero
21  return result
```

When this algorithm has cycles, at some point in the algorithm there will still be nodes with $incoming[v] > 0$, but there will be no nodes with $incoming[v] = 0$, so the algorithm will stop processing and return the topological ordering on the vertices that it has found.

**Spring 2011 Quiz 2, Problem 4.** Suppose that you want to get from vertex $s$ to vertex $t$ in an unweighed graph $G = (V, E)$, but you would like to stop by vertex $u$ if it is possible to do so without increasing the length of your path by more than a factor of $\alpha$.

Describe an efficient algorithm that would determine an optimal $s$-$t$ path given your preference for stopping at $u$ along the way if doing so is not prohibitively costly. (It should either return the shortest path from $s$ to $t$, or the shortest path from $s$ to $t$ containing $u$, depending on the situation.)

If it helps, imagine that there are burgers at $u$.

**Solution:** Use BFS to find three shortest paths in the undirected graph: the path from $s$ to $t$, the path from $s$ to $u$, and the path from $u$ to $t$. Then compare $\alpha \cdot d[s, t]$ to $d[s, u] + d[u, t]$ to figure out whether it's worth it to take a detour through $u$.

# Shortest Paths

**Spring 2011 Final, Problem 6.** Consider a connected weighted directed graph $G = (V, E, w)$. Define the ***fatness*** of a path $P$ to be the maximum weight of any edge in $P$. Give an efficient algorithm that, given such a graph and two vertices $u, v \in V$, finds the minimum possible fatness of a path from $u$ to $v$ in $G$.

**Solution:** To solve this problem, it is sufficient to update the standard RELAX method so that instead of summing edge weights, we take the maximum of the two edge weights. Then we can use Dijkstra's to compute shortest paths.

**Fall 2009 Quiz 2, Problem 5.** Consider a road network modelled as a weighted undirected graph $G$ with positive edge weights where edges represent roads connecting cities in $G$. However, some roads are known to be very rough, and while traversing from city $s$ to $t$ we never want to take a route that takes more than a single rough road. Assume a boolean attribute $r[e]$ for each edge $r$ which indicates if $e$ is rough or not. Give an efficient algorithm to compute the shortest distance between two cities $s$ and $t$ that doesn't traverse more than a single rough road. (Hint: Transform $G$ and use a standard shortest-path algorithm as a black-box.)

**Solution:** For each vertex $v \in G$, construct a pair of vertices $v_{\text{smooth}}$ and $v_{\text{rough}}$. For each smooth edge $e = (u, v)$, add a directed edge from $u_{\text{smooth}}$ to $v_{\text{smooth}}$ and a directed edge from $u_{\text{rough}}$ to $v_{\text{rough}}$. For each rough edge $e = (u, v)$, add a directed edge from $u_{\text{smooth}}$ to $v_{\text{rough}}$. Then generate all shortest paths rooted at $s_{\text{smooth}}$, and pick the shorter of $d[s_{\text{smooth}}, t_{\text{smooth}}]$ and $d[s_{\text{smooth}}, t_{\text{rough}}]$.

**Fall 2008 Final, Problem 4.** Consider a directed graph $G$ where each edge $(u, v) \in E$ has both a weight $w(u, v)$ (not necessarily positive) as well as a color $color(u, v) \in \{\text{red}, \text{blue}\}$.

The weight of a path $p = v_1 \to v_2 \to \ldots \to v_k$ is equal to the sum of the weights of the edges, plus 5 for each pair of adjacent edges that are not the same color. That is, when traversing a path, it costs an additional 5 units to switch from one edge to another of a different color.

Give an efficient algorithm to find a lowest-cost path between two vertices $s$ and $t$, and analyze its running time. (You may assume that there exists such a path.) For full credit, your algorithm should have a running time of $O(VE)$, but partial credit will be awarded for slower solutions.

**Solution 1:** For bookkeeping reasons, we first do a BFS from $s$ to figure out which nodes and edges are actually reachable. Then we remove all nodes and edges that aren't reachable. This ensures that $V = O(E)$, and any $O(V^2)$ term will be dominated be an $O(VE)$ term.

Next, we create a new graph $G'$ and add nodes and edges in the following way. For each node $u$ in $V$, add two nodes $u_{\text{red}}$ and $u_{\text{blue}}$ to $G'$, and add two directed edges of weight 5: one from $u_{\text{red}}$ to $u_{\text{blue}}$, the other from $u_{\text{blue}}$ to $u_{\text{red}}$. For each red edge $(u, v)$, add an edge from $u_{\text{red}}$ to $v_{\text{red}}$. For each blue edge $(u, v)$, add an edge from $u_{\text{blue}}$ to $v_{\text{blue}}$. Once $G'$ has been constructed, run Bellman-Ford twice to calculate the four distances $d[s_{\text{red}}, t_{\text{red}}]$, $d[s_{\text{red}}, t_{\text{blue}}]$, $d[s_{\text{blue}}, t_{\text{red}}]$, and $d[s_{\text{blue}}, t_{\text{blue}}]$.

Say that we are given a path in $G$. We can transform that path to a path in $G'$ by using each of the corresponding edges in $G$, and then using the cost 5 edges whenever we have a red edge in the path adjacent to a blue edge. As a result, when we take a path in $G$ and translate it to a path in $G'$, the resulting cost will be exactly the cost we are trying to minimize. Hence, this transformation will return the correct results.

**Solution 2:** Just as in the first solution, we construct a graph $G'$ and add nodes and edges as follows. For each node $u \in V$, add two nodes $u_{\text{red}}$ and $u_{\text{blue}}$ to $G'$. Any path going into $u_{\text{red}}$ will be a path ending in a red edge, and any path going into $u_{\text{blue}}$ will be a path ending in a blue edge. For each edge $(u, v) \in E$, we add two edges that lead to the same destination: if $(u, v)$ is red, then both edges point to $v_{\text{red}}$, and if $(u, v)$ is blue, then both edges point to $v_{\text{blue}}$. The two edges emanage from $u_{\text{red}}$ and $u_{\text{blue}}$. If an edge goes from red to red or blue to blue, then we keep the same edge weight as in the graph $G$. If instead the edge crosses from one color to the other, then we add 5 to the original edge weight. This ensures that whenever we have a path that ends in, say, a blue edge (and therefore ends at a node in the blue graph), we always incur the extra cost of 5 when crossing to the red graph.

**Fall 2010 Quiz 2, Problem 4.** When an airline is compiling flight plans to all destinations from an airport it serves, the flight plans are plotted through the air over other airports in case the plane needs to make an emergency landing. In other words, flights can be taken only along pre-defined edges between airports. Two airports are adjacent if there is an edge between them. The airline also likes to ensure that all the airports along a flight plan will be no more than three edges away from an airport that the airline regularly serves.

Given a graph with $V$ vertices representing all the airports, the subset $W$ of $V$ which are served by the airline, the distance $w(u, v)$ for each pair of adjacent airports $u$, $v$, and a base airport $s$, give an algorithm which finds the shortest distance from $s$ to all other airports, with the airports along the path never more than $3$ edges from an airport in $W$.

**Solution:** We want a shortest path, with the additional restriction that no node in the shortest path can be more than 3 edges away from $W$. To accomplish this, we first find all of the nodes that are at most 3 edges away from some airport in $W$. An easy way to do this is to perform BFS with the "start node" initialized to the set of all nodes in $W$. Once we perform this $O(V + E)$ step, we can remove all other nodes in the graph, and then just run one of the standard shortest-path algorithms.

**Spring 2009 Quiz 2, Problem 3.** Beverly owns a vacation home in Hawaii and wishes to rent the place out for $n$ days beginning on May 1st (on the $n + 1$st day, she plans to take a vacation there herself). She has obtained $m$ bids, each of which has a starting day $s_i$ and ending day $e_i$ (between $1$ and $n$), and the amount $\$_i$ that the bidder is willing to pay. Beverly can only rent the house to a single bidder on any given day. (That is, she may not accept two bids $b_i$ and $b_j$ such that the intervals $(s_i, e_i)$ and $(s_j, e_j)$ overlap.)

Beverly decides to model this problem as a directed graph with weighted edges so that she can use a standard graph algorithm (or a minor variation of a standard algorithm) to find the bids to accept which maximizes her revenue. Describe such a model, and give the asymptotic cost of finding the set of bids that maximizes the revenues. Assume that the bids are given as list of tuples, not necessarily sorted in any particular order. Try to find an algorithm that is as efficient as possible.

**Solution:** Construct a graph with one node for each of the $n$ days, with one extra node for the $n + 1$st day. A bid $\$_i$ on interval $(s_i, e_i)$ corresponds to an edge from node $s_i$ to node $e_i + 1$ of weight $\$_i$. In this way, we ensure that if Beverly chooses a particular bidder $i$ to occupy her home, then it is first available one day after the bidder $i$ has moved out. We also add edges of weight $0$ from one day to the next, to account for the days when the most profitable thing to do is wait for the next bidder's vacation to start.

Given this graph, our goal will be to find the longest (most profitable) path from Day 1 to Day $n + 1$. To do this efficiently, we can take advantage of the fact that the graph generated in this manner is directed and acyclic, and it is therefore possible to negate all edge lengths and compute shortest paths in $O(V + E)$ time. The number of vertices in the graph we construct is $n + 1$, where $n$ is the number of days. We construct one edge for each bid and one edge for each adjacent pair of days. Hence, the total number of edges is $m + n$. As a result, the algorithm will take time $O(n + 1 + m + n) = O(m + n)$ in total.

**Fall 2009 Final, Problem 7.** Given four vertices $u$, $v$, $s$, and $t$ in a directed weighted graph $G = (V, E)$ with non-negative edge weights, present an algorithm to find out if there exists a vertex $v_c \in V$ which is part of some shortest path from $u$ to $v$ and also a part of some shortest path from $s$ to $t$. The algorithm should run in $O(E + V \log V)$ time. Partial credit will be given to less efficient algorithms provided your complexity analysis is accurate.

**Solution:** If there is a shortest path from $u$ to $v$ containing $v_c$, then we know that $d[u, v_c] + d[v_c, v] = d[u, v]$. So to find such a $v_c$, we run Dijkstra's rooted at $u$ and $s$, and then we reverse all edges and run Dijkstra's rooted at $v$ and $t$. The result will be four tables: all shortest paths from $u$, all shortest paths from $s$, all shortest paths to $v$, and all shortest paths to $t$. Each of these will be in a dictionary with $O(1)$ lookup. So to find $v_c$, we loop through all vertices and perform $O(1)$ lookups and $O(1)$ mathematical operations to discover whether the current vertex is at the intersection of some pair of shortest paths between $u$ and $v$ and between $s$ and $t$.

6.006 Introduction to Algorithms
Fall 2011