Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — April 29, 2014
Professors Srini Devadas, Nancy Lynch and Vinod Vaikuntanathan — Problem Set 5

Problem Set 5

All parts are due Tuesday, April 29 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Your Name: Eric Klinkhammer

Collaborators: George Cheng, Yang Dai

Part A

Problem 5-1. Hobbits' Gold

(a) With IOUs
Because the hobbit's are searching for the maximum amount of gold they can find, they are in effect searching for the longest path. In order to reduce this problem to one of finding the shortest path, we can multiply all the edge lengths by negative one.

Now that the graph is a DAG, we can use topological sort to get the sequence of nodes that will allow Bellman-Ford to converge after only one iteration. That relaxation will, in a DAG, find the shortest path.

This approach is $O(|V| + |E|)$ because topological sort takes that amount of time. A single relaxation of Bellman-Ford is $O(|E|)$, as is the intial graph transformation because it only multiplied each edge once.

(b) Without IOUs
This time, the hobbits are not allowed to owe any money. In graph terms, the path length from the source to the target must never have a negative path length (from the source to a node in the path).

As before, we initially transform the graph by multiplying all of the edges by negative one.

Now, we need to modify the topological sort slightly. Topological sort works by first constructing a DFS tree, which requires a depth first traversal. To do that requires a stack, so that when a dead end is reached, parent nodes can be popped off and checked for other children. In this algorithm, there will be a second stack, as well as a sum (initially 0).

When examining a discovered nodes children, if the edge length of the child and the current sum would be positive (remembering that the edge lengths were all multiplied by -1), then that child is not discovered. If the edge and the sum would be negative, then both push that edge value onto the stack and add it to sum. When a node no longer has any children or all of its children are not discoverable because of the constraint, then you have reached a leaf in the DFS tree. At this point, pop a node off the stack to re-examine the parent node for other children. At the same time, pop a value off of the edge stack and decrement that value from sum. Continue this process, discovering all valid children subject to the constraint before looking at the parent until the stack is empty and the entire graph has been processed.

The relevant part in pseudocode:

```
S = stack()
vals = stack()
S.push(source)
vals.push(0)
sum = 0
while( s is not empty):
    node = s.pop()
    sum -= vals.pop()
    for w in node.adj_list:
        if d(node,w) + sum < 0 and w has not been discovered:
        # Net positive sum
            s.push(w)
            vals.push( d(node,w))
    # Use node to construct DFS tree as normal
```

As above, the limiting step in this algorithm is the topological sort, so this algorithm still runs in $O(|V| + |E|)$ time. The additional stack adds a constant factor (twice the number of pushes and pops) to the process, but that the asymptotic efficiency remains the same. Also, copied from part a: this approach is $O(|V| + |E|)$ because topological sort takes that amount of time. A single relaxation of Bellman-Ford is $O(|E|)$, as is the intial graph transformation because it only multiplied each edge once.

Problem 5-2. Dijkstra Properties and Proofs

(a) Base Case
Invariant 1 holds in the base case because, before any iterations of Dijkstra, S is an empty set. Similarly, for all $u \in V$, $d[u] = \delta_s(s, u)$ because $d[u] = \infty$ and there is obviously not a path through S to the node u because there are no nodes in S (and therefore no paths through).

(b) $\delta_s(s, w) = \delta(s, w)$ before iteration
Assume, for the purposes of contradiction, that there existed a better path not through nodes in s to w. This would imply that, for some node v, $\delta(s, v) + \delta(v, w) < \delta_s(s, w)$. However, v is either in Q or has not been discovered . In the case of the former, that means that $d[v]$ is less than $d[w]$, which is impossible becasue it would have been processed first if that were the case. If it has not been discovered, then it must be a descendent of a node currently in Q (assuming it is in connected component). However, if it were a descendent, its path length would be at least as great as the path length of its ancestor in the Q, and thus it also cannot be a part of the shortest path to w (this is why no negative edges is important).

Therefore, for any node w currently being processed, the best path from the source to that node passes only through processed nodes (in S).

(c) Part 1 of Invariant Holds After Iteration
From part b (of this problem), we know that there is a best path from s to w that only passes through nodes in S. Additionally, from part 1 of the invariant, the distances of all the nodes in S (and thus the distance to the ancestor of w) are correct. Together, we know that the best path to w must be only through nodes in S and that all nodes in S are correct. There will not be a shorter path found, so $d[w]$ records the correct distance before the iteration. After the iteration, w is in S, and the invariant remains true because the only additional node (w) satisfies the requirement.

(d) Part 2 of Invariant Holds After Iteration
This step proves that for all nodes u in V, $d[u] = \delta_s(s, u)$.

There are three possibilities for u: it can be in S, it can be w (the node moving from Q to S), or it can be a node remaining in Q.

The first part of the invariant is true after an iteration (from above), so after the iteration u must still have the correct distance recorded. The path associated with that distance must only be through S because, before the iteration, $d[u]$ was less than all the distances of objects still in the queue (otherwise it would have been processed first). Going through a node not currently in S, in other words, would make the path longer. Therefore, the invariant is true in this case.

The second case is if u is the node moving. Since the invariant was true before the iteration, we know that the best distance through S was recorded for u. Additionally, since part 1 was true before and after the iteration, the distance for w remains correct. Essentially, since it was a path through only S before and it remains the correct path afterwords, the invariant that it is the correct path through S after is true.

The third case is if u is a node still in Q. Assuming that $d[u] = \delta_s(s, u)$ before the iteration, the only case that would change its value is if node w was the parent of u. If that were the case, we know that the distance of w is correct (from part 1), and therefore the new path through w to u is also correct and through S. The invariant would hold in this case. If w were not the parent of u, then the best path would remain whatever it was before the iteration, and, if it held before the iteration, the invariant will hold after.

(e) $u \in S$ and $v \in Q$ then $\delta(u) \leq \delta(v)$
The above statement holds after any number of iterations of Dijkstra's algorithm. At all times, the distances through nodes in S to both u and v were correct (part 2 of invariant). Because u was processed before v (the reason u entered S before v), it must have a smaller distance than v (a property of the priority queue). The distance of u will remain the same (part 1 of invariant) and the distance of v will always depend on a shortest path through S (part 2). Since v is not in S, it must have a path length longer than all nodes in S (again, otherwise it would have been processed first and be in S).

Problem 5-3. Hobbits' Gold II

(a) IOUs and Infinite Cycles
Since this is no longer a DAG, and there are negative edges, we use Bellman-Ford to find the shortest path of the graph after performing the same transformation we did in the first problem (multiplying all of the edges by negative one so that the shortest path is the one that corresponds to the most money).

Bellman-Ford already takes care of finding infinite negative cycles. After relaxing all of the edges for all of the vertices, it will perform one final relaxation. If that relaxation reveals that the values did not converge, then there is a negative cycle in the graph, and the hobbit's can now find unlimited gold.

Bellman-Ford runs in $O(VE)$ time, while the graph transformation runs in $O(E)$ time, so this approach runs in $O(VE)$ time.

(b) Infinite Cycles Without IOUs
In this case, the hobbit's must always have a positive amount of gold. We begin the

same way, by multiplying all of the edge lengths by -1, and do so again in O(E) time.

We again will use Bellman-Ford, except this time, when we relax an edge there is an extra condition. If u is the edge from v to w, we can only update w if $d[v] + u < d[w] < 0$. If, at the end of running the algorithm, the destination node does not have a valid parent pointer, then the hobbits will always get stuck at a gate. As above, if the an additional relaxation reveals that the path lengths do not converge to a value, then the hobbits can get unlimited gold. Otherwise, Bellman-Ford works as expected in $O(VE)$ time, outputing the proper path, meaning the entire algorithm will be $O(VE)$.

(c) Enough Gold
In this problem, the Hobbits are in an identical situation as in part a, but, if they encounter infinite gold, they have a finite target. We begin by running Bellman-Ford as in part A after the same transformation. If there is no infinte cycle, we're done, and Bellman-Ford tells us the path. If, however, there is an infinite cycle, we have to identify it.

We have the parent pointers, and we can trace back from the destination node to the cycle. The cycle will be identifiable by a repeated node. All the nodes inbetween the repeated node, including the repeated node, are in the cycle. The repeated node is the entrance to the cycle.

At this point, we have two of the three components needed to construct the path. We have the cycle and the path from the cycle to the destination. We now only need a path to the cycle from the source. However, because there is a source of infinite gold and IOUs are acceptable, it actually doesn't matter how we reach this cycle, as long as we correctly identify which node in the cycle the path from s touches first. We will use BFS to find that path.

Before doing so, we will make two alterations to the graph. The first is that we will add a node connected to all nodes in the cycle. The second is that we will treat all edges as unweighted and reverse their direction. Now, the graph is ready to run BFS with the added node as a source and the original source node as the destination.

Now, we have a path (almost certainly not optimal) from the source node to the first node in the cycle. We also have the cycle, and the path to the destination. The final path depends on the actual values of the path lengths. It will certainly have the path to the cycle, the cycle a certain number of times, and the path to the destination. The number of cycles can be computed with a simple division. Number of Cycles = $\frac{\text{Enough}-\text{Path to cycle}-\text{Path from cycle to dest}}{\text{Cycle Length}}$.

This approach takes $O(VE)$ time. Bellman-Ford takes $O(VE)$ time, but all of the other steps are much faster. The BFS is only $O(V + E)$, and the transformations are all $O(E)$. The final path computation is a constant time division (rounding up if necessary).