Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — Wednesday, November 2, 2016
Professors Erik Demaine, Debayan Gupta, and Ronitt Rubinfeld — Problem Set 4

Problem Set 4

All parts are due Tuesday, November 15, 2016 at 11:59PM.

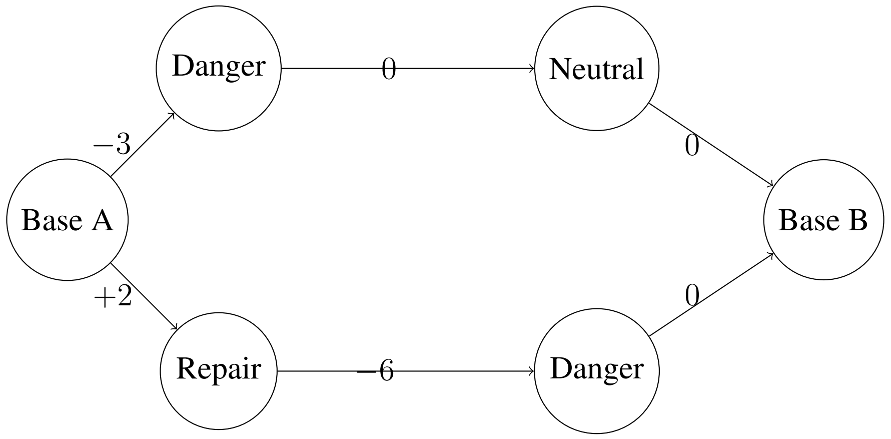
Name: Milo Knowles

Collaborators: Ayush Sharma, Magnus Johnson

Part A

Problem 4-1.

(a) Make the weight of every edge the weight of the node it leads into. Then, remove the weights from all the nodes. Now, we have an edge weighted graph, and every path will still have the same cost as in the node-weighted case. Edges that go into a node with no weight (i.e Base B) will simply have no weight. For example, if we modified the example from Problem 1:



(b) The total cost of a path represents the net change in integrity of Speedy's armor, which we can call D. In order for Speedy to successfully complete a mission,

$C + D \geq 0 \implies D \geq -C$

If this is true, then Speedy can complete the mission, because his armor integrity will be ≥ 0 when he reaches his destination.

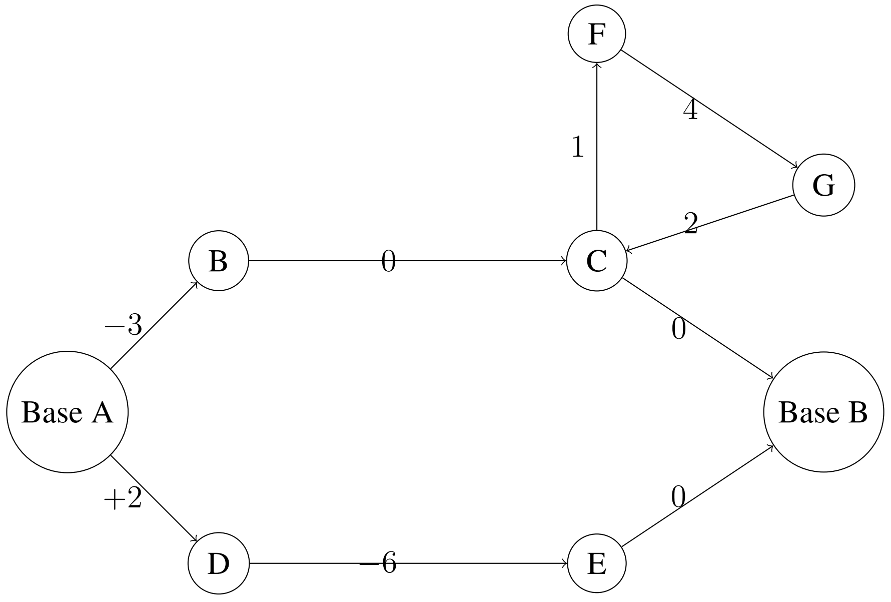
We can adapt the Bellman-Ford algorithm to find a single-source shortest path from Base A to Base B on our network of roads. However, in this case our "shortest path" is the path that maximizes the weight of the path (this is best for Speedy. To adapt

Bellman Ford to do this, we initialize all of our nodes with a weight of negative infinity, and change our relaxation function to the following for two vertices u and v.

if $v.d + w_{uv} > u.d$:
then $u.d = v.d + w_{uv}$

Assuming there are no positive-weight cycles, Bellman-Ford can find a best in $O(VE)$ time, because it requires $|V| - 1$ rounds of edge relaxation over E edges to find the shortest path. If the weight of the best path is $\geq -C$, then we know that Speedy can complete his mission.

(c) Speedy is caught in a positive-weight cycle. Because Speedy is still traversing the graph, but his armor integrity has not dropped below 0, this is the only possibility. An example graph looks like the following:



In this case, if Speedy arrives at vertex C, he will enter a never-ending loop from C to F to G to C, increasing his armor integrity with each cycle. He will never reach Base B because he will always be able to improve his armor by going around the cycle.

(d) Similar to Part B, we can use an adapted version of the Bellman Ford algorithm to find the highest-weight path to every node in the graph. We can start at any arbitrary Normally, the algorithm will find the highest-weight path to every node in $|V| - 1$ iterations. However, if on the Vth iteration we can still perform relaxations on on the graph and the highest weight of vertices continues to increase, this means that there must be positive-weight cycles in the graph.

If there is a positive weight cycle, any node that is reachable from that positive weight cycle will continue to increase it's weight as Bellman-Ford runs. On the Vth iteration of Bellman-Ford, start storing all of the nodes whose weight increases in a dictionary.

Do Bellman-Ford another $|v| - 1$ times. Every time a nodes weight increases, check if it is already in the dictionary, and if not, add it.

After $|v| - 1$ more iterations of Bellman-Ford, every node that is reachable from a positive-weight cycle will have been identified. Speedy could be at any one of these nodes.

The overall runtime of this algorithm is $O(VE)$, because we ran $2|v| - 2$ iterations through Bellman-Ford.

Problem 4-2.

(a) In order to ensure that Jerry's path ends up at a hiding spot on every 3rd move, we create a graph G_3, which is a triple copy of G. Each vertex v in G has 3 vertices in G_3: v_1, v_2, and v_3. For every edge (u, v) in the original graph, create the edges (u_1, v_2), (u_2, v_3), and (u_3, v_1). This will take $O(V + E)$ time, because we are creating $2V$ extra vertices, and $2E$ extra edges. The structure of G_3 ensures that, if Jerry starts at some vertex with a subscript 1, he will end up at another vertex with subscript 1 in three moves.

Remove every vertex with subscript 1 from G_3 that is not a hiding spot. Now, every path that exists from s_1 to f_1 is guaranteed to end up at a hiding spot on every 3rd move. This takes $O(h)$ time, where h is the number of hiding spots (guaranteed to be $O(V)$).

Jerry can only move North or East in this X by Y graph, so there cannot be any cycles. Therefore, this is a Directed-Acyclic-Graph. We can find a single-source-shortest-path on a DAG in $O(V + E)$ time by topologically sorting our graph G and then doing one iteration of Bellman-Ford, relaxing nodes in the order of the topological sort. Run Bellman-Ford to find the shortest path from s_1 to f_1 in G_3. This path will require the least effort, and guarantee that Jerry ends up at a hiding spot every 3rd minute.

The runtime of this algorithm is $O(V + E)$, because we add up the $O(V + E)$ time to create our augmented graph, $O(V)$ time to remove non-hiding spots, and $O(V + E)$ time to run Bellman-Ford on our DAG.

(b) Similar to Part A, we create an augmented version $G_3(V, E, w)$ of our graph $G(V, E, w)$. First, create self-loops of weight zero for every vertex in G. Then, for each vertex v in G, create 3 vertices in G_3: v_1, v_2, and v_3. For every edge (u, v) in the original graph, create the edges (u_1, v_2), (u_2, v_3), and (u_3, v_1) in G_3.

Remove every vertex with subscript 1 from G_3 that is not a hiding spot. Now, every path that exists from s_1 to f_1 is guaranteed to traverse at most 3 edges between hiding spots, and takes into account the fact that Jerry can wait at any vertex. Creating our augmented graph G_3 takes $O(V + E)$ time, because it adds $2V$ vertices and $3E$ edges, and removes $O(V)$ vertices.

Use Dijkstra's Algorithm on G_3 in $O(V \log V + E)$ time using a Fibonacci heap. Dijkstra's Algorithm works for weighted graphs with cycles, so we are fine.

The algorithm takes $O(V+E)$ time to build our augmented graph, and $O(V \log V+E)$ time to find the shortest path on it. Therefore, the overall runtime is $O(V \log V + E)$.

Problem 4-3.

(a) Let s be the vertex of the starting configuration, and f be the solved cube configuration. Our graph is unweighted, so we can find a shortest path in $O(V + E)$ time using a breadth-first-search. To find the actual sequence of moves that achieves the fastest solution, start at f and retrace the steps of the breadth-first-search using parent pointers until s is found, filling an array of moves from back to front. This will take $O(V)$ time, so the overall runtime is $O(V + E)$.

(b) We run a breadth-first-search from every vertex in G, which will give us the shortest path from every vertex to every other vertex. We must do V searches that each take $O(V+E)$ time. This gives a total runtime of $O(V^2+VE)$. However, because this is a very dense graph, $E >> V$, so the V^2 term can be ignored and our runtime simplifies to $O(VE)$.

To find the shortest path from some c_1 to c_2, trace back parent pointers from c_2 to c_1, which will take $O(k)$ time since we know that the path has length k.

Part B

Problem 4-4.

(a) *Submit your implementation on alg.csail.mit.edu*

(b) *Submit your implementation on alg.csail.mit.edu*

-to update b: -take min of x-¿y (2) + y-¿z (2) — x-¿y(1) + y -¿ z (2) — x-¿y(2) + y-¿z (1)