Assignment 4

Jake Levin
Collaborators: Joe Salter, Luke Petruzzi

April 2017

1 Connectivity

a) We will prove that the cut condition outlined in the problem holds if and only if there is way to convert all the edges into arcs that produces a strongly connected directed graph. We model this problem by saying $A \iff B$, where A is the event that the cut condition holds, and B is the event that there is a way to convert all the edges into arcs such that we produce a strongly connected graph. We break this problem up into the following two proofs:

$\underline{A \to B}$:
We need to show that if the cut condition holds, then this implies that we can convert all the edges into arcs and produce a strongly connected graph. To prove this we examine all of the possible cases where the cut condition holds.

1. Every cut has at least two edges crossing it. In this case if this condition holds we can take the base case of a minimum of two edges and convert each of these edges into arcs that are directed in opposition directions across the cut. This ensures that there is a path that goes both ways across this cut, and therefore maintains strong connectivity between all nodes across these cuts.

2. At least one of each of an arc and an edge. In this case if this condition holds, we can the base case of a minimum of one arc and one edge and convert the edge into an arc that is directed in the opposite direction of the arc across the cut. This ensures that there is a path that goes both ways across this cut, maintains strong connectivity between all nodes across these cuts.

3. At least two arcs and no edges, but at least one arc crosses in each direction. In this case if this conditions holds, we do not have to redirect anything. As it is, all nodes across these cuts will have an arc pointing across the cut in both directions and so the graph remains strongly connected across these cuts.

Thus if the cut condition holds for all possible cuts within the graph, we have proven that this implies that we can convert all the edges (if any) in the now

directed graph into arcs such that the entire graph will be strongly connected.

$\underline{B \rightarrow A}$:
We will prove this by examining the contrapositive of this statement $\neg A \rightarrow \neg B$. We therefore need to show that if the cut condition does not hold, that this then implies that there is no way to convert the edges in the graph into arcs such that this produces a strongly connected directed graph. If the cut condition does not hold, then this tells us that we can take a cut of the graph where there are less than 2 edges crossing it, and if there is a singular edge then there are no additional arcs crossing it, or there are no edges and any number of arcs but they all cross in the same direction. In the case where there is one singular edge this edge can be converted into an arc pointing in one of either direction across the cut, but not both. This therefore produces a graph that is directed, but a path only exists between nodes across this cut in a singular direction. By definition this implies that we have distinct sets of strong components divided by the cut, and therefore B cannot not hold. In the case where we have a number of arcs that cross our cut, if the cut condition does not hold then all these arcs point in the same direction and therefore all paths across this cut are unidirectional again producing distinct sets of strong components across the cut. Therefore the graph cannot be strongly connected across this cut.

Thus as the contrapositive of $B \rightarrow A$ holds, we have proven that if there is a way to convert all edges in the graph into arcs such that this produces a strongly connected directed graph, this then implies that the cut condition holds.

We have shown that $A \rightarrow B$ and $B \rightarrow A$ both hold, and thus have proven $A \iff B$: the cut condition holds if and only if there is way to convert all the edges into arcs that produces a strongly connected directed graph.

b) Given a mixed graph, we seek to devise an algorithm that produces either a cut that violates the cut condition or directs all the edges so that the resulting directed graph is strongly connected. To do so we will use an algorithm proposed by Tarjan, Chung, and Garey, for finding a strongly connected orientation in a mixed graph in linear time.[1]

We start by stating two lemmas that this algorithm will ultimately extend. The proofs for the following lemmas can be seen in brief in the aforementioned article and in full in Boesch and Tindell's *Robbin's theorem for mixed multigraphs*:

Lemma 1. If a mixed graph is orientable, and contains the undirected edge v, w on a cycle, then orienting the undirected edge in this graph in the direction of the cycle produces an orientable graph.

[1] Original article available at http://www.math.ucsd.edu/ fan/mypaps/fanpap/79scoomm.PDF

Lemma 2. If a mixed graph is orientable, we can take a cut of the graph and produce two subsets X and X' with one undirected edge connecting them and any number of undirected edges going from X' to X, such that orienting the undirected edge in the direction of X to X' forms an orientable graph.

We use a two pass depth-first search algorithm to first orient all of the undirected edges in our graph (using Lemmas 1 and 2), and then to check that the directed graph we produce is indeed strongly connected. We start our first DFS of G from an arbitrary vertex, allowing multi-directional traversal along undirected edges. As we encounter new vertices in the graph we assign them each a number 1 to n where n is the total number of vertices in the graph. Note that if our depth-first search terminates before it numbers all of the vertices then we know by default that the graph is not orientable nor strongly connected so we can break our search. We can divide the edges we encounter into two groups, "tree-edges" which when traversed lead to new vertices, and "non-tree" edges which when traversed lead to previously seen vertices. Note that all of the tree edges taken together form a spanning tree of the graph. Once we reach a previously visited node and thus backtrack, we orient any undirected edges we encounter as we backtrack. We do so by maintaining number $low(v)$ for each vertex, initialized to $low(v) = v$, and such that $1 \leq low(v) \leq v$. When retreating on an edge (v, w) we reorient and update $low(v)$ as follows:

Case 1. Edge is a non-tree edge or $low(w) < w$. Here we replace $low(v)$ by $\min low(v), low(w)$. If edge is undirected orient it from v to w.

Case 2. Edge is a tree edge and $low(w) = w$. If the edge is directed we can abort the search as we know the graph is not orientable. If it is undirected we orient it from w to v and set $low(w) = low(v)$.

This first pass in its entirety runs in $O(n + m)$ time by definition of DFS. Assuming it doesn't abort the following lemmas hold for this pass:

Lemma 3. We know through simple inductive reasoning on the number of retreats performed in the search that for any v, $low(v) \leq v$, and that there is a path from v to $low(v)$. Therefore after a successfully completed search we know there is a path from any vertex v to the start vertex.

Lemma 4. If not all vertices are reached during the search, or if it aborts, then our graph is not orientable. This also implies that if our graph is orientable then it is strongly connected.

We prove the above lemama by considering a retreat along edge (v, w). If $low(w) < w$ just before this retreat takes place, then this can only occur if there is an edge in the original graph (x, y), such that x but not y is a descendant of w in our spanning tree. This is because if this edge were to exist, by the nature of our DFS algorithm we know that $y < w \leq x$. It follows then that eventually

this descendant ordering will result so that $low(w) \leq y < w$. Alternatively if $low(w) < w$ then it is not a descendant of w and we get the opposite descendant characteristics for our edge (x, y). This property of the descendants of nodes proves both parts of our lemma and allows us to preserve orientability when possible for all cases, and can be seen in greater detail in Tarjan, Chung, and Garey's original article.

Additionally if all nodes are not reached in our DFS, then all vertices are not reachable from the first vertex, it is not strongly connected, and the graph is not orientable. If our search aborts then Lemma 4 implies there must have been a bad cut, which we showed in part a means the graph is not strongly connected. We can then apply the second pass to do a backwards search from the first vertex and make sure every other vertex is reachable on the graph. If this is the case then we know the graph is strongly connected by definition, and has been appropriately oriented by Lemma 3. If the graph fales this second pass however, we know by Lemma 4 that the graph must not be strongly connected and therefore not orientable. These lemmas and subsequent conclusions together prove sufficiently that a mixed multigraph can only be orientable if it is strongly connected and has no cuts that violate the cut condition.

We know that this algorithm runs in time complexity $O(|V| + |E|)$. This is because it is quite simply two modifications of the depth-first search algorithm, with a trivial constant factor to account for orienting operations.

2 Performance Measurement

a) We look to prove that the counter can be added to a given program block if and only if the block is useful. We model this problem as $A \iff B$, where A is the event that the counter can be added to a given program block such that it is guaranteed to be executed at least every k steps (where k is a constant depending on the program), and B is the event that the block the counter is added to is useful. We break this problem up into the following two proofs:

$\underline{A \to B}$:
We will prove that A implies B by taking the contrapositive of this statement, $\neg B \to \neg A$. Therefore we look to show that if the block the counter is added to is not useful, this then implies that when the counter is added to a block we cannot guarantee that it will be executed at least every k steps. We know this to be true because the definition of a useful block as outlined in the problem states that if the counter is not useful, then it is not on every cycle of the graph and thus there are cycles within the graph that do not trigger an increment within the counter. Therefore it is impossible to guarantee that the counter will be executed at least every k steps, because it is easy to imagine a scenario where there are multiple program blocks that form independent cycles, one of which is the main cycle of the larger program and contains a loop arc and thus

is self cyclical and is executed exponentially more times than a secondary sub cycle. If in this case the counter were to be placed in this secondary cycle - which is possible if the counter is not useful - then there is no guarantee that the program block that contains the counter would be executed at least every k steps.

$\underline{B \rightarrow A}$: Again, we will prove that B implies A by taking the contrapositive of the statement $\neg A \rightarrow \neg B$. We therefore look to show that if we cannot guarantee that the program block the counter is placed in is executed at least every k steps each time the program is run, then this implies that the program block is not useful. If we cannot guarantee that the program block the counter is placed in is executed at least every k steps, then this implies that within the larger program graph there is a cycle of program block nodes that take more than k steps before the counter is reached. By our definition of k, this then implies that the counter is on a node that is not a part of every cycle, or it would have been executed within this cycle of at least k steps. By our definition of a useful node, if the node the counter has been added to is not a part of every cycle, then that node is not useful. Thus we have shown that if the program block the counter is added to is not executed at least every k steps, then the program block is not useful, i.e. $\neg A \rightarrow \neg B$.

We have proven that $\neg B \rightarrow \neg A$ and $\neg A \rightarrow \neg B$ hold, thus $A \rightarrow B$ and $B \rightarrow A$ hold, thus $A \iff B$.

b) Outlined below is a polynomial-time algorithm to identify all the useful blocks in a program. A block is useful in the program if it is a part of all cycles in that program. The specifications of the problem indicate that we can ignore cases where we deal with acyclic graphs or graphs with multiple non-trivial strong components. This also then means by definition that there can be only one loop-arc in the graph. Our algorithm works by individually removing nodes in the graph and checking if the graph is acyclic. We do this because we know that by the definition of a useful node occurring on every cycle of the graph, if a useful node were to be removed the graph would become acyclic. Therefore, our algorithm will iterate through each node in the graph n. At each iteration remove node n and run a depth-first search on the remaining nodes in the graph to determine whether the graph contains a cycle. If it does not, we know that we have removed a useful node and so we mark it as useful. When the algorithm terminates after examining each node we will have found all useful nodes in the graph.

To prove that this algorithm is correct we have to show that the only case where our graph becomes acyclic is when a useful node has been removed, thus proving that any time a node is removed and results in an acyclic graph it must be useful. Assuming there are at least 2 cycles in the graph, our useful nodes are by definition the only nodes in our graph that appear in all the cycles.

Additionally we know that some number of them must exist in our graph as by definition without any of them the graph would contain disjoint cycles and would therefore be acyclic. If we were to remove a useful node, we see that by definition at least 2 of these cycles must become disjoint. This results in a graph that is no longer cyclic and our algorithm holds. If instead we were to remove a non useful node we know by definition that there is at least one other cycle in the graph that does not contain this node. Therefore we conclude that although removing this node may break an unknown number of cycles, assuming that there is another node in the graph that is useful (which we assume by the specifications of the problem) then there is another node in the graph that is contained in all cycles and our graph will still be cyclic. It is also possible that our graph might only contain a single cycle to begin with. In this case however we see trivially that by definition all the nodes are useful in this graph and no further calculation is required. Thus, we have proven that *only* removing a useful node from our graph will cause it to become acyclic.

The time complexity of our algorithm can be analyzed as follows:
We know depth-first search by definition runs in $O(|V| + |E|)$ time. Since our algorithm performs dfs as we iterate through all vertices in the graph, our ultimate run time complexity is $O(|V|(|V| + |E|))$.

However, it is worth noting that our problem can ultimately be reduced to a question of how to detect all overlapping nodes between cycles and keep track of the range of nodes that are found in all cycles. Therefore, if we were to index the nodes and run a single modified depth-first search on the graph we can simultaneously maintain two range variables for each node that correspond to the first and last node we've examined that occur in all of that nodes possible cycles. Any time a cycle is terminated and a new cycle begins, we simply have to mark this as having occurred and then set the new cycle of that node to start with the current node in our dfs stack. If we do this we could actually produce the same results in $O(|V| + |E|)$ time.

3 Pirates

In this problem, we are given an undirected graph where each edge represents a channel and each node a destination of interest. We want to find a spanning tree of the graph so that we can reach every node, however our goal is to find the spanning tree that contains channels with the largest possible depths so we can maximize the size of our ship. Therefore, our goal is to then look for the smallest weighted edge within this maximum spanning tree, i.e. the shallowest channel of the deepest channels possible that we can take so that we can reach every destination of interest. To do so we will use a modified version of an algorithm for finding minimum bottleneck spanning trees in undirected graphs, instead looking for a "maximum" bottleneck spanning tree. The algorithm for finding minimum bottleneck spanning trees upon which our algorithm is based

can be found in CLRS, Problem 23-3. Our algorithm is essentially a modified binary search algorithm, therefore outlined below is a recursive subroutine we will use to narrow down our set of edges until we are left with the maximal weighted edges that form a spanning tree of our graph.

We will define a value b, such that b is originally the median weight of all edges in our graph (thereby dividing our graph into two equal sets of edges, all in one strictly greater than or equal to all in the other). This simply requires examining each edge in the graph, and therefore runs in $O(E)$ time. Since we are concerned with finding maximal weighted edges, we will remove the smaller set of these edges so that only the larger edges are remaining. We then examine these edges to determine whether the remaining graph is connected, and proceed as follows (note that to test for connectivity we can simply run a breadth-first search on the graph and check to see if any vertices remain - this runs in linear time):

If the remaining edges are connected: In this case we know that our set of edges form a possible spanning tree of the graph. However we do not yet know if this spanning contains the maximum weights possible. Therefore, we will run our subroutine again on this graph of higher weighted edges in order to reduce the total edges in our graph while maximizing their weights, thereby searching for the absolute maximal spanning tree.

If the remaining edges are not connected. Then we know that our set of edges can not possibly form a spanning tree of the graph. Therefore we need to widen our search range and will run our subroutine but this time defining b as the median weight of the smaller set of edges that we removed.

This binary search runs until we reach a point where increasing b by 1 edge weight results in a disconnected graph. We then examine the previously connected graph (which we do in $O(E)$ time), as we know that we have now found the spanning tree that contains the absolute maximal possible edge weights. We know this to be correct because we have iteratively reduced our set of edges to the smallest set of the largest edges that can possible span the tree, and thus have ensured that all other possible spanning trees contain edge weights of lesser values. We can then examine each edge in the graph to find the edge with the smallest weight. This edge will be the channel with the shallowest depth on the graph that contains the deepest possible channels that can reach each destination of interest.

Because this edge is the smallest of the absolute maximal edges, we know by definition that we have found the correct bottleneck channel. We know this to be correct because if this edge is the smallest of the weights in our spanning tree, any other path we traverse to reach a destination will be made up of edges that have larger weights and thus represent deeper channels. Additionally as shown above the algorithm will never find a possible spanning tree for the same graph

that will contain greater edge weights. Thus, this bottleneck edge weight represents the maximum draft our new ship can be while still being able to reach all the desired ports from the pirate base. Furthermore this algorithm will always produce the same maximal spanning tree and thus the same minimum weight along this spanning tree for a given graph, and is therefore deterministic.

Each division of the sets requires finding a median edge weight which can be done in $O(E)$ time, and each check for connectivity requires a simple BFS in $O(E)$ time. Additionally the number of edges we consider each time is halved, resulting in run time $O(E + E/2 + E/4... + 1) = 2E - 1 = O(E)$. Thus our overall time complexity is $O(E)$.

4 Electric Cars

Our problem asks us to determine reachability of vertices in an edge weighted graph where reachability is defined by the ability to reach a node on the graph from a source vertex and then return to that vertex by taking paths with total arc weights at most C. In order to solve this problem we must find the shortest possible path from a designated source vertex to every other vertex on the graph, as well as the shortest possible return paths from each other vertex on the graph to the source vertex. If we sum the combined arc weights required to reach and return from a given vertex from the source along these shortest paths, we can compare this total value to C to determine if the vertex is reachable.

To do so we will use two passes of a modified version of the Bellman-Ford algorithm (outlined in lecture and the slides) for finding the shortest paths from a source vertex to all other vertices in a weighted graph. Note that Bellman-Ford can handle negative weighted arcs which are a possibility in this graph. Additionally, we do not have to worry about the algorithm returning upon finding a negative cycle because our graph by definition cannot contain one. Our first pass of Bellman-Ford is responsible for finding the shortest path from the source vertex to all of the desired destination nodes in our graph and proceeds as follows:

We run Bellman-Ford with our home being the source vertex. As we iterate through each possible outgoing arc of each vertex we update an index that maintains the shortest possible distance from the source to that vertex. The only modification here from the original Bellman-Ford algorithm is that we must ensure that none of these shortest paths are illegal, i.e. that at any point along the path our car would have run out of battery and have a negative (or 0) charge.

We start by initializing the index of the distance at the source to 0, and the distance for all vertices to infinity. For each arc (v, w) with weight x, we check to see if the current distance index of vertex v + weight x is less than the distance index at vertex w. If so, we update that value with the new shortest

path distance. Additionally if during our check $d[v] + x < 0$ (which can occur when dealing with negative edges), we set that distance to be 0 so as to avoid accidentally representing an overcharge of our actual battery capacity. We must also check to make sure that the battery charge capacity C is greater than the current shortest distance calculation. If not, then we know this is an illegal path and we move on to the next possible arc and mark this vertex as unreachable (reset its distance index to infinity). Once all vertices have been examined we terminate our search. This modification of the Bellman-Ford algorithm ensures that we have now obtained an indexed list that maintains the shortest path from the source vertex, thereby giving us the minimum battery charge required to reach each destination while accurately detecting when reaching a destination would exceed our battery limit and is therefore unreachable.

However we must now also find the shortest possible return paths from all of our destinations vertices back to the source. To do so we first reverse the direction of all the arcs in the graph. We can then run a second pass of a modified Bellman-Ford algorithm on this graph keeping our home as the source vertex, which will identify the shortest possible return paths from every other vertex to the source. Similar to our first iteration we start by initializing the distance index at our source to 0, and the distance for all other vertices to infinity. The distance here at each index signifies the minimum charge necessary to return home from the vertex. We run Bellman-Ford again, checking for each new arc (v, w) with weight x to see if $d'[v] + x$ is less than $d'[w]$. If so we update our indexed list so that $d'[w] = d'[v] + x$. Additionally, if at any point our path distance drops below 0 this indicates that we gain a charge following this path home and therefore the minimum charge required to get home from this vertex is 0. Once this pass of the Bellman-Ford has examined every vertex in our graph, we can terminate our algorithm. This gives us a second list of indexed distances that contain the shortest paths from every destination vertex to the source, and is therefore an accurate calculation of the minimum battery charge required to get home from any destination in the graph.

Finally, to determine which of our destination vertices are reachable we simply iterate through each vertex and sum the values of the costs to reach and return from that vertex as indicated by the results of our two indexed destination lists from our Bellman-Ford searches. If $d[v] + d'[v] > C$, then this vertex must be unreachable. If $d[v] + d'[v] < C$ then we have enough battery to go to and from the vertex and thus it is reachable.

We know this is correct because our values represent the absolute minimum battery cost (the shortest path) to and from that vertex. Thus summing these costs gives us the minimal charge required for the vertex to be reachable, and we can compare this directly against the car's battery life. Additionally our modified Bellman-Ford search algorithms have ensured that we don't accidentally over calculate the possible charge our battery can hold when dealing with negative arc weights. Therefore we guarantee that we won't accidentally in-

clude a negative offset in the cost of a certain path which is greater than the car can accurately traverse. Our algorithm also ensures that we don't accidentally includes destinations that we could never get to in the first place, even if the return path home has a negative charge that could potentially offset the exhaustive cost of getting to that node in the first place.

The time complexity of this algorithm can be analyzed as follows:
First, we know that Bellman-Ford runs in $O(|V||E|)$ time, therefore our 2 searches cost a total of $O(2(|V||E|))$. Our additional boundary checks to ensure correctness will add an insignificant constant factor to this time complexity. Additionally we perform a final iteration through each vertex to sum the total cost reaching and returning from each possible destination. This costs an additional $O(|V|)$ time. Thus in total our algorithm runs in time complexity $O(2(|V||E|) + |V|)$ time.