Problem 1 - Revisiting Inference in Graphs

The algorithm for this one is decently simple:

1. Attach the source node to all X (in that direction) with capacity ∞. Attach all Y to the sink node (in that direction) with capacity ∞.
2. Represent each undirected edge in the original graph as two (oppositely) directional edges, each with capacity 1. Run the polynomial-time max-flow algorithm, and retrieve the minimum-cut partition using the paths in the residual graph. The minimum cut's value is the minimum sum. Everything on the source side is 0, everything else is 1.

Proving this minimizes the differences, on the other hand, may be a hassle. Our algorithm retrieves some partition that is formed entirely of 0s on one side and 1s on the other - whether or not it's actually a solution is up for grabs.

Let's take a page from Kleinberg-Tardos and correctly out the gate state that in some directed graph (I'll rectify this later) where you have a source and a sink and each edge is capacity 1, if there are k edge-disjoint paths then the maximum flow between the source and the sink is at least the number of edge-disjoint paths between the source and the sink. This follows directly from the functionality of the max-flow algorithm - paths of this sort in a flow graph can each be exhausted through once, lending a single unit of flow to the maximum total, and the residual graph may at some point include paths involving the reverse of parts of these disjoint paths (which accounts for the flow being at least k) - the long story short is that there is guaranteed to be k paths that don't need to interact with the backwards edges in the residual graph at all, each of which can flow k units.

So that's done with. Before we move on to proving the converse of the above statement, let's establish why our problem here is equivalent in one form to the above : we have a source extending infinite capacity to all X and all Y extending infinite capacity to a Y. In terms of identifying disjoint paths, however, if the source needs to extend m edges to the m members of X, m *must* be greater than or equal to k - if it were less than k then we would know off the bat that there aren't k disjoint paths (too few edges to start). This, combined with the fact that a max-flow algorithm cannot cut an infinite capacity edge when non-infinite edges exist (automatically violating min-cut) means that a source projecting m edges to m members of X is the same - for both disjoint paths and for the max-flow algorithm - as just having the source projecting the edges that the m members of X would anyway, as a sort of conglomeration of all the X. A similar logic applies for the conglomeration of all Y as equivalent to just having the sink take all the edges that would otherwise go into the Ys.

Back to the converse of two paragraphs ago. That is - we must show that if the max flow is greater than or equal to k then there are guaranteed to be k edge-disjoint paths. So, if we've got a flow value we know is $\geq k$, and we create k edge-disjoint paths, and knowing that any edge that is actually being flowed through has flow 1, we just need to prove that a flow of v in a 0-or-1 flow graph contains v edge-disjoint paths. Through induction, we can see that if the flow v is 0, there are no paths (of course, nothing to be flowed through means no way to reach the sink), and if it's not 0, there must be some edge carrying a single unit of flow. This edge's flow must be conserved and must have been conserved: thus there was some other edge that carried flow to the edge's first vertex, and there will be some other edge that will carry flow out of the edge's second vertex. We can follow this pattern ad infinitum: eventually we reach the sink or a node we've seen before. If we reach the sink, there is a path from source to sink, and if we were to *not* flow through it we'd have total max flow $v-1$, which through the induction hypothesis forms $v-1$ paths - re-add the non-flowed one and see that it adds $1+v-1=v$ as the total flow, and there are v paths. If the path eventually reaches one of its nodes, on the other hand, it forms a cycle - flowing 0 through all the cycle members after the path finally does find the sink is equivalent to never having found them in the first place in terms of the flow, and the edges can thus be used for other paths while the number of paths still extant is retained - trading one including a cycle for one not at no cost to the current path or the flow. As such, if the flow in a directed graph is at least k, there are k edge-disjoint paths in that graph.

By extension, in all directed flow networks, the maximum number of edge-disjoint paths is equal to the minimum number of edges whose removal separates source and sink. This is because if some set of edges' removal fully separates source and sink, then each edge-disjoint path must use at least one of them, and because the max number of edge-disjoint paths (for the case where all edges have capacity 1, anyway) is equivalent to the min-cut, whose partition counts the number of edges whose cuts are required. Because

removing that number of edges fully separates source and sink, each edge-disjoint path must use at least one of them - meaning that the maximum number of edge-disjoint paths is equal to the minimum number of edges required cut.

To translate from the given undirected graph to a directed graph (to which all of the above is applicable), decompose each edge (u, v) into (v, u) and (u, v). Issues arise as edge-disjoint paths may not use the same edge but are fine with using different edges going in opposite directions connecting the same vertices: however, any issues with this can be resolved by changing one's flow to 0 and bolstering the other's. Any flow network has a maximum flow in which for all twinned edges of this sort, one direction or the other has 0 units flowing through it. If we have an extant maximum flow which has nonzero flow on both of the twinned edges, the residual graph has the ability to transfer flow and capacity from one twinned edge to the other and thus subtracting the smaller flow from both edges in the final graph creates a maximum flow with the same value where one of the edges has flow 0.

Obviously, the minimum number of edges across the partition forms the minimum number of differences (each of which is of value 1). Thus minimizing edges minimizes the sum of differences. Notably, if we were to approach this using any sort of fractions instead of just 0 and 1, we would see that any replacement of the optimal solution described above using transitional fractions (i.e. instead of X-0-1-Y, X-1/3-2/3-Y) has at best an equivalently good solution, and cannot get any better - you can distribute a straight line of 0s followed by 1s over a set of edges - but can absolutely get worse - encountering a fork or a consolidation of edges anywhere along the graph involves the addition of another vertex with a different value, as in the predecessor problem (i.e. two X's each contacting two nodes of 1/3 each contacting a single 2/3 and then a Y has sum 4/3, instead of the equivalent solution above which is still 1).

Problem 2 - Maximum In-Set

The algorithm for computing the in-set of maximum value on a directed acyclic graph is:

1. Create the source vertex s and attach an edge (s, v) with capacity c_v equivalent to the vertex's value to all vertices v with a positive value. If all the vertices are positive, the whole graph is the in-set.
2. Create the sink vertex t and attach an edge (v, t) with capacity c_v equivalent to *the absolute value* of the vertex's value to all vertices v with a negative value. If all the vertices are negative, the empty set is the in-set.
3. All previously extant edges have capacity ∞.
4. Run the polynomial time maximum-flow algorithm on s, t. Retrieve the vertices reachable from s (by a BFS or DFS) in the residual graph - this is the maximum in-set, which forms everything on the source side of the minimum cut (except the source itself).

To explain why this works, I'll just briefly delineate why it works in polynomial time - clearly setting the problem up as a max-flow problem just requires adding one edge per vertex, which is an additional $O(n)$, and following the paths in the residual graph that emerge by either a BFS or a DFS takes at most $O(n+m)$ time. These costs are both polynomial, and probably (I don't know of any max-flow algorithm that runs in less than $O(n+m)$, anyway) pale in comparison to the (also polynomial) time required to actually run the max-flow algorithm itself. This is thus a polynomial algorithm.

Now to the actual functionality of the algorithm. Observe that in setting up the graph to work for a max-flow calculation, we've speciated edges into three types: either an edge is between the source and a vertex of positive value, between a vertex of negative value and the sink, or between two vertices, each of which must be connected to either the source or the sink (and they can both be connected to just one of those). This last 'intra-vertex' type of edge is assigned an infinite capacity. This immediately rules it out for elimination in the partition of the min-cut, which means that the partition will have to cut out lines of the first two types. This means that such a partition must leave an in-set as the source side of the partition - a certain quantity of edges from the source to the vertices will be cut, and those vertices will thus be left on the sink side of the partition. These vertices may project uncut infinite-capacity edges to others on the source side, but this doesn't violate the source side as an in-set, and the source side will not project similar edges to the sink side, because the min-cut algorithm would have thus recognized that as a valid path to the sink, and depending on the value of the recipient vertex, would have made a different cut including both the projecting vertex and the recipient, or neither of them. Regardless, any paths not cut by the min-cut algorithm on the source-to-vertex side of the graph would need to be cut on the vertex-to-sink side of the graph, again excluding infinite-capacity edges. This prevents any edges from leaving the remnant in-set - the same logic as above prevents any vertices on the source side of the partition from projecting edges to the sink side of the partition, either because such infinite-capacity edges would have been taken account of by the min-cut algorithm or because (vertex-to-sink) the min-cut algorithm actually did cut them. In short, then, *the min-cut algorithm must create a partition whose source side is an in-set.* All edges leaving must be vertex-to-sink edges, which are cut in the process of the algorithm, and those cut on the source-to-vertex side delineate vertices on the sink side.

When the min-cut is calculated, it will cut across those lines that connect the source to vertices and vertices to the sink. This means that the actual value of the min-cut - the minimum capacity required to halt all flow - is formed of the capacities from the source to the vertices that *don't* make it into the source side of the partition, and are thus on the sink side, plus the capacities from the vertices that *are* in the source side of the partition to the sink. If the partition is (S, T), this means that the minimum cut is $\Sigma_{(s,v):v\in T} + \Sigma_{(v,t):v\in S}$. This term, notably, is minimized. Now, the capacity of the minimum cut should be equivalent to the absolute maximum capacity possible (which is the capacity from the source to each positive vertex) minus the value of the maximum in-set (as that value is the capacity going into the in-set minus the capacity that would have come out. The subtraction is thus just those capacities leaving the in-set + the capacities entering vertices *not* in the in-set, which is the min-cut). The absolute maximum is a constant on a particular graph, so maximizing the value of the in-set minimizes the value of the minimum cut, and vice versa. For proof, the maximum capacity possible is $\Sigma_{(s,v)}$, so the min-cut is $\Sigma_{(s,v)} - (\Sigma_{(s,v):v\in S}$ [capacities going into in-set] $+\Sigma_{(v,t):v\in S}[capacities coming out of in-set]) = \Sigma_{(s,v):v\in T} + \Sigma_{(v,t):v\in S}$, as I established above. Minimizing the min-cut thus means maximizing the in-set's value, and as the min-cut algorithm gets the minimum, this algorithm gets the maximum in-set.

Problem 3 - Highly Active Subsets

The algorithm for finding a sufficiently (strictly greater than a given x) active subset on a graph G is the following:

1. Create a set of vertices Y such that each member $y \in Y$ is the representation of each single edge connecting a pair of vertices in the original undirected graph G or the representation of each single edge or the pair of edges connecting a pair of vertices in the original directed graph G (i.e. if $(a, b), (b, c), (c, b) \in E$, $AB \in Y$ and $BC \in Y$). Attach the source to these vertices (s, x) with capacity equivalent to the weight of the matching edge's (or edges's, summed, for directed graphs) endpoints.
2. Attach each of these edge-based 'two letter' vertices in Y to its two corresponding 'one letter' vertices (the endpoints of the original edge or edges) in V, with infinite capacity. Connect each 'one letter' vertex in V to the sink, (v, t), with capacity equivalent to x, the mark for sufficient activity.
3. Run a polynomial-time max-flow-min-cut algorithm on the source and sink. All 'one letter' vertices reachable from paths starting at the sink in the residual graph constitute a sufficiently active subset (although not necessarily the one of greatest cardinality) - which is to say, those 'one letter' vertices on the source side of the partition formed by the min-cut. If the only thing on the source side of the partition is the source itself, there's no sufficiently active subset.

In so many words, represent edges as vertices and attach the source to them with their weight as the capacity, attach those to the actual vertices with infinite capacity, and attach the actual vertices to the sink with x capacity. The way to go about proving this is to essentially conceptualize it like one would the baseball elimination problem - "success" in a set of vertices being sufficiently active is a function of how weighted the edges are between the vertices (how many wins there are to go around) and how many vertices there are in the set (how many teams the wins are distributed through). Should the former divided by the latter exceed the limit x, the set is sufficiently active. It follows that a single vertex cannot be a sufficiently active set (no edges), two vertices can be one if their shared edge has a weight greater than $2x$, three vertices can be if one of the (up to 3) edges has weight greater than $3x$, or if the sum of the weights of the edges is greater than $3x$ in total. Instead of trying every subset like this, it's easier to set the flow network up as described above. In the baseball elimination problem, x is the maximum number of games a potentially eliminated team has yet to play; the weights between vertices are the number of games those two other teams have yet to play against each other.

The first bottleneck on the flow to consider is the edges from the source to the 'two letter' vertices - all those added together form the absolute maximum for the flow through the graph (naturally, if nothing else bottlenecks them then they do) and if the maximum flow is actually this value, it means that the edges from 'one letter' vertices to the sink had a high enough capacity to absorb all of their flow, which is to say that there is no sufficiently active subset (no minimum partition on that side of the graph = cut on the source side of the graph = no reachable paths to any 'one letter' nodes). Thus, for some sufficiently active subset to exist in the graph, the maximum flow in the graph must be less than the sum of all the edge weights. There has to be some cut on the source side of the graph, which indicates that there has to have been at least two cases in which the capacity from the 'one letter' vertices to the sink was completely filled, with flow left to spare on the source to 'two letter' side.

If we suppose that such a sufficiently active subset exists, then it must be the case that the maximum flow in the graph is less than the sum of all edge weights, as we established above. The minimum cut will have this lesser value, f. If this is the case, then there is a set T of 'one-letter' vertices reachable from the source in the residual graph - those that fall on the source side of the partition. If some 'two letter' vertex projects one edge into a 'one letter' vertex in T, and the other edge into a 'one letter' vertex not in T, then that latter infinite capacity edge would cross the partition - which is defined by the minimum cut. The minimum cut cannot cut infinite capacity edges - this is thus a contradiction. Any 'two letter' vertex which connects to a 'one letter' vertex not in T must be on the sink side of the partition. Similarly, if both 'one letter' vertices belonging to a 'two letter' vertex are in T, but the 'two letter' vertex is on the sink side of the partition, moving that 'two letter' vertex onto the source side of the partition reduces the value of the minimum cut by the capacity of the edge from the source to the 'two letter' vertex. This must be a nonzero value and as such any minimum cut creates a partition such that when two 'one letter' vertices are in T, their corresponding 'two letter' vertex is on the source side of the partition.

Great. If a given partition is a minimum cut, then it's guaranteed that a 'two letter' vertex falls on the source side if and only if its corresponding 'one letter' vertices fall on the source side as well. The value of

the minimum cut itself is the summed value of all the edges it removes - since it won't ever do this with infinite capacity edges, we're restricted to source-to-'two letter' edges where one of the 'two letter' vertex's constituents don't belong to T, and 'one letter'-to-sink edges, where the 'one letter' vertex does belong to T. The minimum capacity cut is thus equal to the sum of the weights of the edges *not* in the sufficiently active set plus the number of vertices in the sufficiently active set times x. The first term there - the sum of those edges who have one endpoint not in the sufficiently active set - is equivalent to the absolute maximum possible capacity (i.e. all the edge weights) minus those edge weights which *do* have both endpoints in the sufficiently active set. Since (because we assumed a sufficiently active set exists) the actual maximum flow is lesser than the absolute maximum possible, and that is equivalent to the capacity of the min cut, we can rearrange the equation algebraically to say that the number of vertices in T times x minus the sum of edges with both endpoints in T is less than 0, which is to say that the sum of the edges with both endpoints in T is strictly greater than the number of vertices in $T * x$. This confirms that to exist, a sufficiently active set must have the property that the weights of its edges summed are greater than the number of vertices times x (which is a rephrasing of the original problem, so we knew that already), but in the process, also confirms that the operation of the above algorithm does, in fact, check the truth of the assumption of the set's existence. Of course, given the above property's behavior, if it were the case that that property did not hold true, such a set would not exist - as the min-cut algorithm is guaranteed to find any that exist.