6.12 INDEPENDENT PATHS, CONNECTIVITY, AND CUT SETS

A pair of vertices in a network will typically be connected by many paths of many different lengths. These paths will usually not be independent however. That is, they will share some vertices or edges, as in Fig. 6.10 for instance (page 140). If we restrict ourselves to independent paths, then the number of

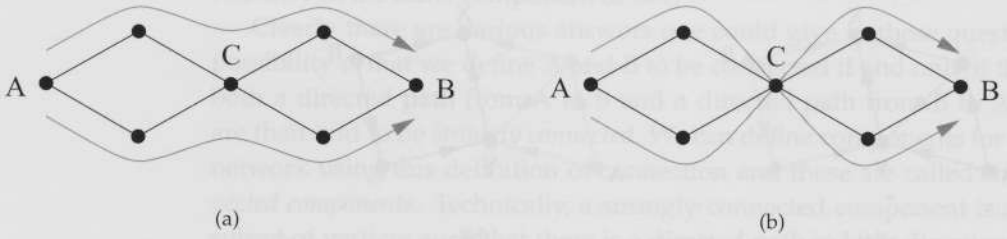


Figure 6.15: Edge independent paths. (a) There are two edge-independent paths from A to B in this figure, as denoted by the arrows, but there is only one vertex-independent path, because all paths must pass through the center vertex C. (b) The edge-independent paths are not unique; there are two different ways of choosing two independent paths from A to B in this case.

paths between a given pair of vertices is much smaller. The number of independent paths between vertices gives a simple measure of how strongly the vertices are connected to one another, and has been the topic of much study in the graph theory literature.

There are two species of independent path: edge-independent and vertex-independent. Two paths connecting a given pair of vertices are *edge-independent* if they share no edges. Two paths are *vertex-independent* (or *node-independent*) if they share no vertices other than the starting and ending vertices. If two paths are vertex-independent then they are also edge-independent, but the reverse is not true: it is possible to be edge-independent but not vertex-independent. For instance, the network shown in Fig. 6.15a has two edge-independent paths from A to B, as denoted by the arrows, but only one vertex-independent path—the two edge-independent paths are not vertex-independent because they share the intermediate vertex C.

Independent paths are also sometimes called *disjoint paths*, primarily in the mathematical literature. One also sees the terms *edge-disjoint* and *vertex-disjoint*, describing edge and vertex independence.

The edge- or vertex-independent paths between two vertices are not necessarily unique. There may be more than one way of choosing a set of independent paths. For instance Fig. 6.15b shows the same network as Fig. 6.15a, but with the two paths chosen a different way, so that they cross over as they pass through the central vertex C.

It takes only a moment's reflection to convince oneself that there can be only a finite number of independent paths between any two vertices in a finite network. Each path must contain at least one edge and no two paths can share

an edge, so the number of independent paths cannot exceed the number of edges in the network.

The number of independent paths between a pair of vertices is called the *connectivity* of the vertices.[26] If we wish to be explicit about whether we are considering edge- or vertex-independence, we refer to *edge* or *vertex connectivity*. The vertices A and B in Fig. 6.15 have edge connectivity 2 but vertex connectivity 1 (since there is only one vertex-independent path between them).

The connectivity of a pair of vertices can be thought of as a measure of how strongly connected those vertices are. A pair that have only a single independent path between them are perhaps more tenuously connected than a pair that have many paths. This idea is sometimes exploited in the analysis of networks, for instance in algorithmic methods for discovering clusters or communities of strongly linked vertices within networks [122].

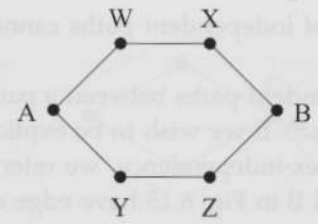
Connectivity can also be visualized in terms of "bottlenecks" between vertices. Vertices A and B in Fig. 6.15, for instance, are connected by only one vertex-independent path because vertex C forms a bottleneck through which only one path can go. This idea of bottlenecks is formalized by the notion of cut sets as follows.

Consider an undirected network. (In fact the developments here apply equally to directed ones, but for simplicity let us stick with the undirected case for now.) A *cut set*, or more properly a *vertex cut set*, is a set of vertices whose removal will disconnect a specified pair of vertices. For example, the central vertex C in Fig. 6.15 forms a cut set of size 1 for the vertices A and B. If it is removed, there will be no path from A to B. There are also other cut sets for A and B in this network, although all the others are larger than size 1.

An *edge cut set* is the equivalent construct for edges—it is a set of edges whose removal will disconnect a specified pair of vertices.

A *minimum cut set* is the smallest cut set that will disconnect a specified pair of vertices. In Fig. 6.15 the single vertex C is a minimum vertex cut set for vertices A and B. A minimum cut set need not be unique. For instance, there is a variety of minimum vertex cut sets of size two between the vertices A and B in this network:

[26] The word "connectivity" is occasionally also used in the networks literature as a synonym for "degree." Given that the word also has the older meaning discussed here, however, this seems an imprudent thing to do, and we avoid it in this book.



{W,Y}, {W,Z}, {X,Y}, and {X,Z} are all minimum cut sets for this network. (There are also many different minimum edge cut sets.) Of course all the minimum cut sets must have the same size.

An important early theorem in graph theory addresses the size of cut sets. *Menger's theorem* states:

> If there is no cut set of size less than n between a given pair of vertices, then there are at least n independent paths between the same vertices.

The theorem applies both to edges and to vertices and was first proved by Karl Menger [216] for the vertex case, although many other proofs have been given since. A simple one can be found in Ref. [324].

To understand why Menger's theorem is important, consider the following argument. If the minimum vertex cut set between two vertices has size n, Menger's theorem tells us that there must be at least n vertex-independent paths between those vertices. That is, the number of vertex-independent paths is greater than or equal to the size of the minimum cut set. Conversely, if we know there to be exactly n vertex-independent paths between two vertices, then, at the very least, we have to remove one vertex from each path in order to disconnect the two vertices, so the size of the minimum cut set must be at least n. We thus conclude that the number of vertex-independent paths must be both greater than or equal to *and* less than or equal to the size of the minimum cut set, which can only be true if the two are in fact equal. Thus Menger's theorem implies that:

> The size of the minimum vertex cut set that disconnects a given pair of vertices in a network is equal to the vertex connectivity of the same vertices.

Given that Menger's theorem also applies for edges, a similar argument can be used to show that the same result also applies for edge cut sets and edge connectivity.

The edge version of Menger's theorem has a further corollary that will be of some importance to us when we come to study computer algorithms for analyzing networks. It concerns the idea of *maximum flow*. Imagine a network of water pipes in the shape of some network of interest. The edges of the

network correspond to the pipes and the vertices to junctions between pipes. Suppose that there is a maximum rate r, in terms of volume per unit time, at which water can flow through any pipe. What then is the maximum rate at which water than can flow through the network from one vertex, A, to another, B? The answer is that this maximum flow is equal to the number of edge-independent paths times the pipe capacity r.

We can construct a proof of this result starting from Menger's theorem. First, we observe that if there are n independent paths between A and B, each of which can carry water at rate r, then the network as a whole can carry a flow of at least nr between A and B, i.e., nr is a lower bound on the maximum flow.

At the same time, by Menger's theorem, we know that there exists a cut set of n edges between A and B. If we push the maximum flow (whatever it is) through the network from A to B and then remove one of the edges in this cut set, the maximum flow will be reduced by at most r, since that is the maximum flow an edge can carry. Thus if we remove all n edges in the cut set one by one, we remove at most nr of flow. But, since the cut set disconnects the vertices A and B, this removal must stop all of the flow. Hence the total capacity is at most nr, i.e., nr is an upper bound on the maximum flow.

Thus nr is both an upper and a lower bound on the maximum flow, and hence the maximum flow must in fact be exactly equal to nr.

This in outline is a proof of the *max-flow/min-cut theorem*, in the special case in which each pipe can carry the same fixed flow. The theorem says that the maximum flow between two vertices is always equal to the size of the minimum cut set times the capacity of a single pipe. The full max-flow/min-cut theorem applies also to weighted networks in which individual pipes can have different capacities. We look at this more general case in the following section.

In combination, Menger's theorem for edges and the max-flow/min-cut theorem show that for a pair of vertices in an undirected network three quantities are all numerically equal to each other: the edge connectivity of the pair (i.e., the number of edge-independent paths connecting them), the size of the minimum edge cut set (i.e., the number of edges that must be removed to disconnect them), and the maximum flow between the vertices if each edge in the network can carry at most one unit of flow. Although we have stated these results for the undirected case, nothing in any of the proofs demands an undirected network, and these three quantities are equal for directed networks as well.

The equality of the maximum flow, the connectivity, and the cut set size has an important practical consequence. There are simple computer algorithms, such as the augmenting path algorithm of Section 10.5.1, that can calculate maximum flows quite quickly (in polynomial time) for any given network, and

the equality means that we can use these same algorithms to quickly calculate a connectivity or the size of a cut set as well. Maximum flow algorithms are now the standard numerical method for connectivities and cut sets.

6.12.1 MAXIMUM FLOWS AND CUT SETS ON WEIGHTED NETWORKS

As discussed in Section 6.3, networks can have weights on their edges that indicate that some edges are stronger or more prominent than others. In some cases these weights can represent capacities of the edges to conduct a flow of some kind. For example, they might represent maximum traffic throughput on the roads of a road network or maximum data capacity of Internet lines. We can ask questions about network flows on such networks similar to those we asked in the last section, but with the added twist that different edges can now have different capacities. For example, we can ask what the maximum possible flow is between a specified pair of vertices. We can also ask about cut sets. An edge cut set is defined as before to be a set of edges whose removal from the network would disconnect the specified pair of vertices. A *minimum* edge cut set is defined as being a cut set such that the sum of the weights on the edges of the set has the minimum possible value. Note that it is not now the number of edges that is minimized, but their weight. Nonetheless, this definition is a proper generalization of the one we had before—we can think of the unweighted case as being a special case of the weighted one in which the weights on all edges are equal, and the sum of the weights in the cut set is then simply proportional to the number of edges in the set.

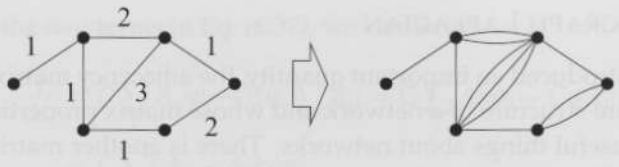
Maximum flows and minimum cut sets on weighted networks are related by the max-flow/min-cut theorem in its most general form:

> The maximum flow between a given pair of vertices in a network is equal to the sum of the weights on the edges of the minimum edge cut set that separates the same two vertices.

We can prove this theorem using the results of the previous section.[27]

Consider first the special case in which the capacities of all the edges in our network are integer multiples of some fixed capacity r. We then transform our network by replacing each edge of capacity kr (with k integer) by k parallel edges of capacity r each. For instance, if $r = 1$ we would have something like this:

[27] For a first principles proof that is not based on Menger's theorem see, for instance, Ahuja *et al.* [8].



It is clear that the maximum flow between any two vertices in the transformed network is the same as that between the corresponding vertices in the original. At the same time the transformed network now has the form of a simple unweighted network of the type considered in Section 6.12, and hence, from the results of that section, we can immediately say that the maximum flow in the network is equal to the size *in unit edges* of the minimum edge cut set.

We note also that the minimum cut set on the transformed network must include either all or none of the parallel edges between any adjacent pair of vertices; there is no point cutting one such edge unless you cut all of the others as well—you have to cut all of them to disconnect the vertices. Thus the minimum cut set on the transformed network is also a cut set on the original network. And it is a minimum cut set on the original network, because every cut set on the original network is also a cut set with the same weight on the transformed network, and if there were any smaller cut set on the original network then there would be a corresponding one on the transformed network, which, by hypothesis, there is not.

Thus the maximum flows on the two networks are the same, the minimum cuts are also the same, and the maximum flow and minimum cut are equal on the transformed network. It therefore follows that the maximum flow and minimum cut are equal on the original network.

This demonstrates the theorem for the case where all edges are constrained to have weights that are integer multiples of r. This constraint can now be removed, however, by simply allowing r to tend to zero. This makes the units in which we measure edge weights smaller and smaller, and in the limit $r \to 0$ the edges can have any weight—any weight can be represented as a (very large) integer multiple of r—and hence the max-flow/min-cut theorem in the form presented above must be generally true.

Again there exist efficient computer algorithms for calculating maximum flows on weighted networks, so the max-flow/min-cut theorem allows us to calculate minimum cut weights efficiently also, and this is now the standard way of performing such calculations.[28]

[28] Another interesting and slightly surprising computational use of the max-flow/min-cut the-

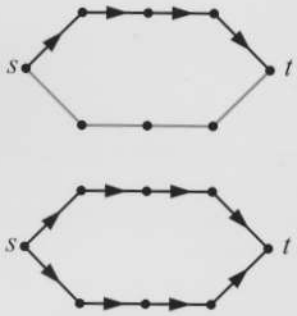
10.5 MAXIMUM FLOWS AND MINIMUM CUTS

In Section 6.12 we discussed the ideas of connectivity, independent paths, cut sets, and maximum flows in networks. In particular, we defined two paths that connect the same vertices s and t to be edge-independent if they share none of the same edges and vertex-independent if they share none of the same vertices except for s and t themselves. And the edge or vertex connectivity of the vertices is the number of edge- or vertex-independent paths between them. We also showed that the edge or vertex connectivity is equal to the size of the minimum edge or vertex cut set—the minimum number of edges or vertices that need to be removed from the network to disconnect s from t. Connectivity is thus a simple measure of the robustness of the connection between a pair of vertices. Finally, we showed that the edge-connectivity is also equal to the maximum flow that can pass from s to t if we think of the network as a network of pipes, each of which can carry one unit of flow.

In this section we look at algorithms for calculating maximum flows between vertices on networks. As we will see, there is a simple algorithm, the Ford–Fulkerson or augmenting path algorithm, that calculates the maximum flow between two vertices in average time $O((m+n)m/n)$. Once we have this maximum flow, then we also immediately know the number of edge-

independent paths and the size of the minimum edge cut set between the same vertices. With a small extension, the algorithm can also find the particular edges that constitute the minimum edge cut set. A simple modification of the augmenting path algorithm allows us also to calculate vertex-independent paths and vertex cuts sets.

All the developments of this section are described for undirected networks, but in fact the algorithms work perfectly well, without modification, for directed networks as well. Readers who want to know more about maximum flow algorithms are recommended to look at the book by Ahuja *et al.* [8], which contains several hundred pages on the topic and covers almost every conceivable detail.



A simple breadth-first search finds a path from source s to target t (top) in this network. A second search using only the edges not used in the first finds a second path (bottom).

10.5.1 THE AUGMENTING PATH ALGORITHM

In this section we describe the augmenting path algorithm of Ford and Fulkerson for calculating maximum flows between vertices in a network.[14] The case of primary interest to us is the one where each edge in the network can carry the same single unit of flow. The algorithm can be used in the more general case where the edges have varying capacities, but we will not discuss that case here.[15]

The basic idea behind the augmenting path algorithm is a simple one. We first find a path from source s to target t using the breadth-first search algorithm of Section 10.3.[16] This "uses up" some of the edges in the network, filling them to capacity so that they can carry no more flow. Then we find another path from s to t among the remaining edges and we repeat this procedure until no more paths can be found.

Unfortunately, this does not yet give us a working algorithm, because as we have described it the procedure will not always find the maximum flow. Consider Fig. 10.6a. If we apply breadth-first search between s and t we find

[14]The augmenting path algorithm is not the only algorithm for calculating maximum flows. It is, however, the simplest and its average performance is about as good as any other, so it is a good choice for everyday calculations. It's worth noting, however, that the *worst-case* performance of the algorithm is quite poor: for pathological networks, the algorithm can take a very long time to run. Another algorithm, the *preflow-push algorithm* [8], has much better worst-case performance and comparable average-case performance, but is considerably more complicated to implement.

[15]See Ahuja *et al.* [8] or Cormen *et al.* [81] for details of the general case.

[16]Technically, the augmenting path algorithm doesn't specify how paths are to be found. Here we study the particular version in which paths are found using breadth-first search, which is known to be one of the better-performing variants. Sometimes this variant is called the *shortest augmenting path algorithm* or the *Edmonds–Karp algorithm*.

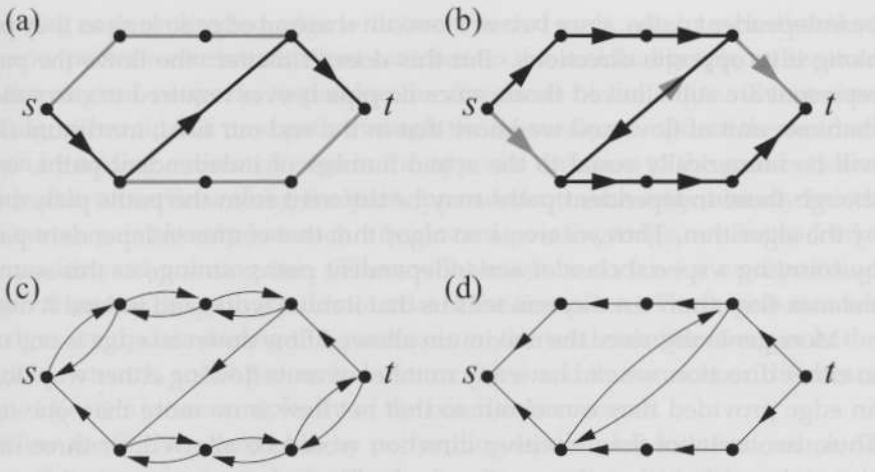


Figure 10.6: The augmenting path algorithm. (a) We find a first path from source s to target t using breadth-first search. This leaves no more independent paths from s to t among the remaining edges. (b) However, if we allow flows in both directions along an edge (such as the central edge in this network), then we can find another path. Panels (c) and (d) show the residual graphs corresponding to panels (a) and (b).

the path marked in bold. Unfortunately, once we have filled all the edges along this path to capacity there are no more paths from s to t that can be constructed with the remaining edges, so the algorithm stops after finding just one path. It is clear, however, that there are in fact two edge-independent paths from s to t—along the top and bottom of the network—and a maximum flow of two, so the algorithm has given the wrong answer.

There is however, a simple fix for this problem, which is to allow fluid to flow simultaneously *both ways* down an edge in our network. That is, we allow a state in which there is one unit of flow in each direction along any given edge. If the edges were real pipes, then this would not be possible: if a pipe is full of fluid flowing one way, then there is no room for fluid flowing the other way too. However, if fluid *were* flowing both ways down an edge, the net flow in and out of either end of that edge would be zero—the two flows would effectively cancel out giving zero net flow. And zero flow down an edge certainly is possible.

So we use a trick and allow our algorithm to place a unit of flow both ways down any edge, but declare this to mean in practice that there is no flow at all on that edge. This means that the paths we will find will no longer necessarily

be independent paths, since two of them can share an edge so long as they pass along it in opposite directions. But this doesn't matter: the flows the paths represent are still allowed flows, since no pipe is ever required to carry more than one unit of flow, and we know that in the end our final, maximum flow will be numerically equal to the actual number of independent paths, even though those independent paths may be different from the paths picked out by the algorithm. Thus we create an algorithm that counts independent paths by counting a special class of *non*-independent paths: strange as this sounds, the max-flow/min-cut theorem tells us that it must work, and indeed it does.

More generally, since the maximum allowed flow down an edge is one unit in either direction, we can have any number of units flowing either way down an edge provided they cancel out so that net flow is no more than one unit. Thus, two units of flow in either direction would be allowed, or three units one way and four the other, and so forth. Three units one way and five the other would not be allowed, however.[17]

To see how this works in practice, consider Fig. 10.6 again. We begin by performing a breadth-first search that finds the path shown in panel (a). Now, however, there is a second path to be found, as shown in panel (b), making use of the fact that we are still allowed to send one unit of flow *backwards* along the edge in the center of the network. After this, however, there are no more paths left from s to t and so the algorithm stops and tells us that the maximum possible flow is two units, which is the correct answer.

This is merely one example of the algorithm: we still have to prove that it gives the correct answer in all cases, which we do in Section 10.5.3. To understand the proof, however, we first need to understand how the algorithm is implemented.

10.5.2 IMPLEMENTATION AND RUNNING TIME

Implementation of the augmenting path algorithm makes use of a *residual graph*, which is a directed network in which the edges connect the pairs of vertices on the original network between which we still have capacity available to carry one or more units of flow in the given direction. For instance, Figs. 10.6c and 10.6d show the residual graphs corresponding to the flow states in 10.6a and 10.6b.

The residual graph is constructed by first taking the initial network and

[17] On networks with directed edges, we allow either the same flow in both directions along an edge (i.e., zero net flow) or one more unit in the forward direction than in the backward direction, but not vice versa.

replacing each undirected edge with two directed ones, one in each direction. We now perform our breadth-first searches on this residual graph, rather than on the original network, respecting the directions of the edges. Every time our algorithm finds a new path through the network, we update the residual graph by adding a directed edge in the opposite direction to the path between every pair of vertices along the path, provided no such edge already exists. If a vertex pair already has such a backward-pointing edge, we instead take away a forward-pointing one. (There will always be such a forward-pointing edge, otherwise the path would not exist in the first place.) The largest number of edges we update during this process is m, the total number of edges in the original network, so the process takes time $O(m)$ and thus makes no difference to the $O(m+n)$ time complexity of the breadth-first search.

Now we find the next path by performing another breadth-first search on the updated residual graph. By always working on the residual graph in this way, we ensure that we find only paths along edges that have not yet reached their maximum flow. Such paths are called *augmenting paths*. The process is repeated until our breadth-first search fails to find any augmenting path from s to t, at which point we have found all the paths there are and the number of paths found is equal to the number of units in the maximum flow from s to t.

Each breadth-first search, along with the corresponding updates to the residual graph, takes time $O(m+n)$ for a network stored in adjacency list format (see Sections 9.4 and 10.3). Moreover, the number of independent paths from s to t can be no greater than the smaller of the two degrees k_s and k_t of the source and target vertices (since each path must leave or enter one of those vertices along some edge, and that edge can carry at most one path). Thus the running time of the algorithm is $O\bigl(\min(k_s,k_t)(m+n)\bigr)$. If we are interested in the average running time over many pairs of vertices, then we can make use of the fact that $\langle \min(k_s,k_t) \rangle \leq \langle k \rangle$ (where the averages are over all vertices), and recalling that $\langle k \rangle = 2m/n$ (Eq. (6.23)), this implies that the average running time of the algorithm is $O\bigl((m+n)m/n\bigr)$, which is $O(n)$ on a sparse network with $m \propto n$. (On the other hand, on a dense graph where $m \propto n^2$, we would have $O(n^3)$, which is much worse.)

10.5.3 WHY THE ALGORITHM GIVES CORRECT ANSWERS

It is plausible but not immediately obvious that the augmenting path algorithm correctly finds maximum flows. We can prove that it does as follows.

Suppose at some point during the operation of the algorithm (including the very beginning) we have found some (or no) paths for flow from s to t, but any paths we have found do not yet constitute the maximum possible flow.

That is, there is still room in the network for more flow from s to t. If this is the case then, as we will now show, there must exist at least one augmenting path from s to t, which by definition carries one unit of flow. And if there exists an augmenting path, our breadth-first search will always find it, and so the algorithm will go on finding augmenting paths until there is no room in the network for more flow, i.e., we have reached the maximum flow, which is equal to the number of paths found.

Thus the proof that the algorithm is correct requires only that we prove the following theorem:

> If at some point in our algorithm the flow from s to t is less than the maximum possible flow, then there must exist at least one augmenting path on the current residual graph.

Consider such a point in the operation of the algorithm and consider the flows on the network as represented by f, the set of all individual net flows along the edges of the network. And consider also the maximum possible flow from s to t, represented by f_{max}, the corresponding set of individual net flows. By hypothesis, the total flow out of s and into t is greater in f_{max} than in f. Let us calculate the difference flow $\Delta f = f_{max} - f$, by which we mean we subtract the net flow along each edge in f from the net flow along the corresponding edge in f_{max}, respecting flow direction—see Fig. 10.7. (For instance, the difference of two unit flows in the same direction would be zero while the difference of two in opposite directions would be two in one direction or the other.)

Since the total flow is greater in f_{max} than in f, the difference flow Δf must have a net flow out of s and net flow into t. What's more, because the "fluid" composing the flow is conserved at vertices, every vertex except s and t must have zero net flow in or out of it in both f_{max} and f and hence also in Δf. But if each vertex other than s and t has zero net flow, then the flow from s to t must form at least one path across the network—it must leave every vertex it enters, except the last one, vertex t. Let us choose any one of these paths formed by the flow from s to t in Δf and let us call this path p.

Since there is a positive flow in Δf in the forward direction along each edge in p, there must have been no such flow in f along any of the same edges. If there were such a flow in f then when we performed the subtraction $\Delta f = f_{max} - f$ the flow in Δf would be either zero or negative on the edge in question (depending on the flow in f_{max}), but could not be positive. Thus we can always safely add to f a unit of flow forward along each edge in p without overloading any of the edges. But this immediately implies that p is an augmenting path for f.

Thus, for any flow that is not maximal, at least one augmenting path always

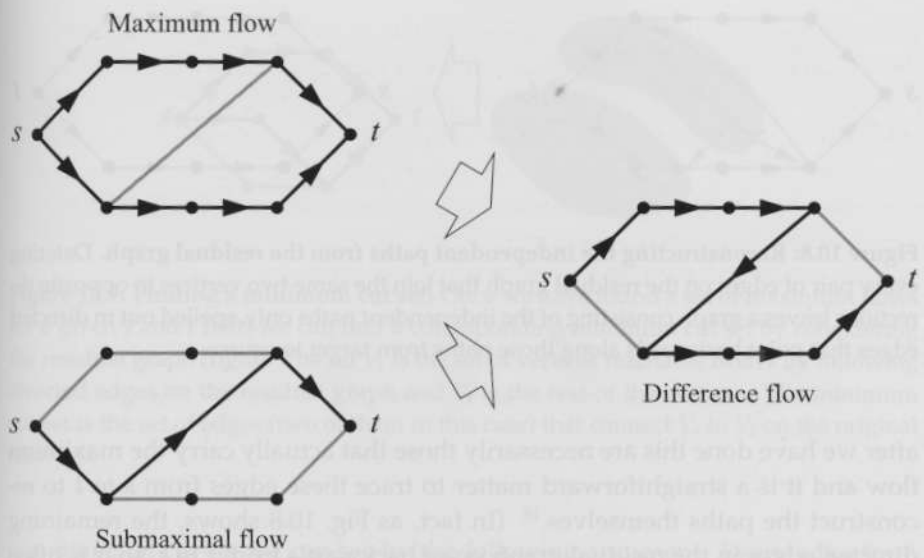


Figure 10.7: Correctness of the augmenting path algorithm. If we subtract from the maximum flow f_{max} (upper left) any submaximal flow f (lower left), the resulting difference flow (right) necessarily contains at least one path from s to t, and that path is necessarily an augmenting path for f.

exists, and hence it follows that the augmenting path algorithm as described above is correct and will always find the maximum flow.

10.5.4 FINDING INDEPENDENT PATHS AND MINIMUM CUT SETS

Once we have found the maximum possible flow between a given pair of vertices, we also automatically have the size of the minimum edge cut set and the number of edge-independent paths, which are both numerically equal to the number of units in the maximum flow (see Section 6.12).

We might also wish to know exactly where the independent paths run. The augmenting path algorithm does not give us this directly since, as we have seen, the augmenting paths it finds are not necessarily the same as the independent paths, but only a very small extension of the algorithm is necessary to find the independent paths: we take the final residual graph produced at the end of the algorithm and remove from it every pair of directed edges that joins the same two vertices in opposite directions—see Fig. 10.8. In other words we are removing all network edges that carry no net flow. The edges remaining

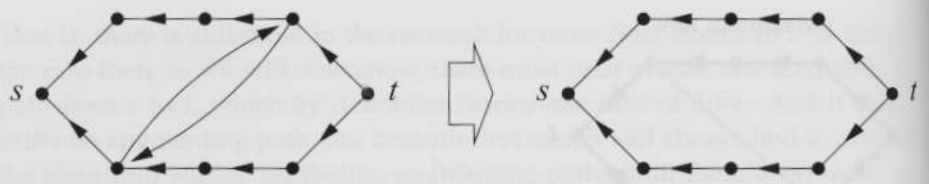


Figure 10.8: Reconstructing the independent paths from the residual graph. Deleting every pair of edges on the residual graph that join the same two vertices in opposite directions leaves a graph consisting of the independent paths only, spelled out in directed edges that point backwards along those paths from target to source.

after we have done this are necessarily those that actually carry the maximum flow and it is a straightforward matter to trace these edges from s to t to reconstruct the paths themselves.[18] (In fact, as Fig. 10.8 shows, the remaining directed edges in the residual graph point backwards from t to s, so it is often easier to reconstruct the paths backwards.)

Another thing we might want is the set of edges that constitutes the minimum cut set for the vertices s and t. In fact in most cases there is more than one cut set of the minimum size, so more generally we would like to find one of the minimum cut sets. Again we can do this by a small extension of the augmenting path algorithm. The procedure is illustrated in Fig. 10.9. We again consider the final residual graph generated at the end of the algorithm. By definition this graph has no directed path in it from s to t (since if it did the algorithm would not have stopped yet). Thus we can reach some subset of vertices by starting at vertex s, but we cannot reach all of them. (For example, we cannot reach t.) Let V_s be the subset of vertices reachable from s by some path on the residual graph and let V_t be the set of all the other vertices in the graph that are not in V_s. Then the set of edges on the original graph that connect vertices in V_s to vertices in V_t constitutes a minimum cut set for s and t.

Why does this work? Clearly if we removed all edges that connect vertices in V_s to those in V_t we disconnect s and t, since then there is no path at all between s and t. Thus the edges between V_s and V_t constitute a cut set. That it

[18] Note, however, that the independent paths are not necessarily unique: there can be more than one choice of paths and some of them may not be found by this algorithm. Furthermore, there can be points in the network where paths come together at a vertex and then part ways again. If such points exist, you will have to make a choice about which way to go at the parting point. It doesn't matter what choice you make in the sense that all choices lead to a correct set of paths, but different choices will give different sets of paths.

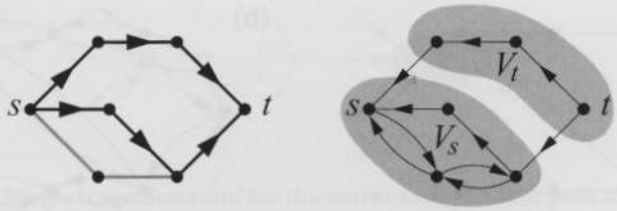


Figure 10.9: Finding a minimum cut set. Once we have found a set of maximum flows for a given s and t (left) we can find a corresponding minimum cut set by considering the residual graph (right). The set V_s is the set of vertices reachable from s by following directed edges on the residual graph and V_t is the rest of the vertices. The minimum cut set is the set of edges (two of them in this case) that connect V_s to V_t on the original network.

is a *minimum* cut set we can see by the following argument. Every edge from a vertex in V_s to a vertex in V_t must be carrying a unit of flow from V_s to V_t. If it were not, then it would have available capacity away from V_s, meaning that there would be a corresponding directed edge away from V_s in the residual graph. In that case, however, the vertex at the far end of that edge would be reachable from V_s on the residual graph and therefore would be a part of V_s. Since the vertex in question is, by hypothesis, in V_t and not in V_s, it follows that it must be carrying a unit of the maximum flow from s to t.

Now, since every edge in the cut set between V_s and V_t is carrying a unit of flow, the size of that cut set is numerically equal to the size of the flow from V_s to V_t, which is also the flow from s to t. And, by the max-flow/min-cut theorem, a cut set between s and t that is equal in size to the maximum flow between s and t is a minimum cut set, and hence our result is proved.

10.5.5 FINDING VERTEX-INDEPENDENT PATHS

Once we know how to find edge-independent paths it is straightforward to find vertex-independent paths as well. First, note that any set of vertex-independent paths between two vertices s and t is necessarily also a set of edge-independent paths: if two paths share none of the same vertices, then they also share none of the same edges. Thus, we can find vertex-independent paths using the same algorithm that we used to find edge-independent paths, but adding the restriction that no two paths may pass through the same vertex. One way to impose this restriction is the following. First, we replace our undirected network with a directed one, as shown in Fig. 10.10, with a directed

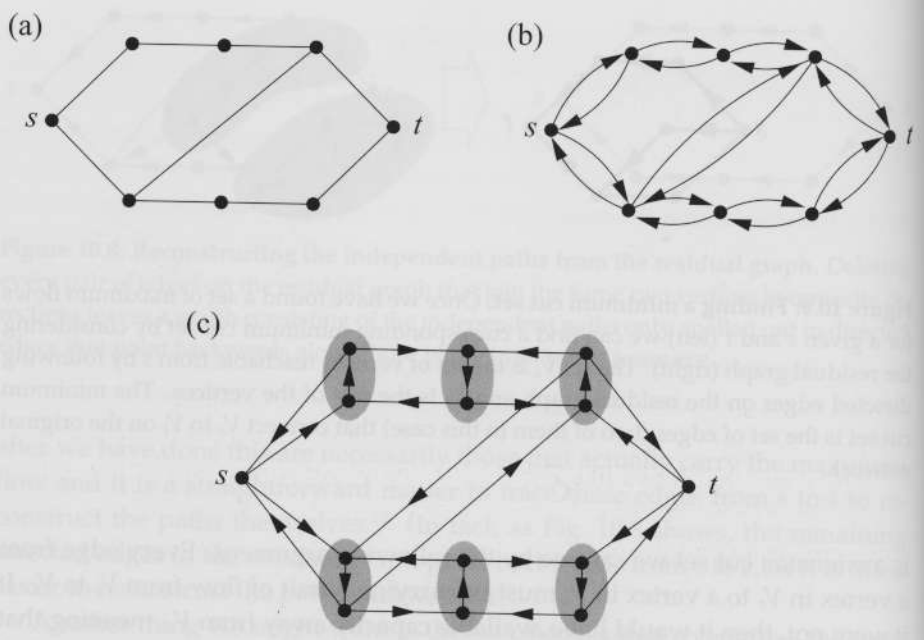


Figure 10.10: Mapping from the vertex-independent path problem to the edge-independent path problem. Starting with an undirected network (a), we (b) replace each edge by two directed edges, then (c) replace each vertex, except for s and t, with a pair of vertices with a directed edge between them (shaded) following the prescription in Fig. 10.11. Edge-independent paths on the final network then correspond to vertex-independent paths on the initial network.

edge in either direction between every connected pair of vertices. This does not change the maximum flow possible in the network and hence does not change the number of independent paths either.

Second, we replace each of the vertices in the network, except s and t, with a construct like that shown in Fig. 10.11. Each vertex is replaced with two vertices separated by a directed edge. All original incoming edges connect to the first of these two (on the left in Fig. 10.11) and all outgoing edges to the second. This new construct functions as the original vertex did, allowing flows to pass in along ingoing edges and out along outgoing ones, but with one important difference: assuming that the new edge joining the two vertices has unit capacity like all others, we are now limited to just one unit of flow through the entire construct, since every path through the construct must traverse this central edge. Thus every allowed flow on this network corresponds to a flow

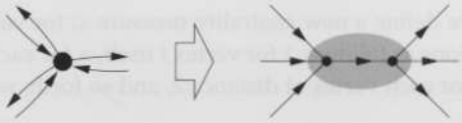


Figure 10.11: Vertex transformation for the vertex-independent path algorithm. Each vertex in the network is replaced by a pair of vertices joined by a single directed edge. All incoming edges are connected to one of the pair and all outgoing edges to the other as shown.

on the original network with at most a single unit passing though each vertex.

Transforming the entire network of Fig. 10.10a using this method gives us a network that looks like Fig. 10.10c. Now we simply apply the normal augmenting path algorithm to this directed network, and the number of *edge*-independent paths we find is equal to the number of *vertex*-independent paths on the original network.
