Chapter 14

Network Flow and Matching

In this chapter, we examine the network flow problem, a graph problem to which many problems can be reduced. In fact, some problems that don’t even appear to be graph problems can be reduced to network flow, yielding efficient algorithms. We begin by defining the problem.

Let \( \mathbb{Z} \) denote the set of integers, and let \( \mathbb{Z}^>0 \) denote the set of positive integers. A flow network is a 4-tuple \( (G, u, v, C) \), where

- \( G = (V, E) \) is a directed graph;
- \( u \in V \) is the source vertex;
- \( v \in V \) is the sink vertex; and
- \( C : E \to \mathbb{Z}^>0 \) gives a positive integer capacity for each edge.

For example, Figure 14.1 shows a flow network whose source is 0, whose sink is 5, and whose edges all have capacity 1. Intuitively, the capacities represent the maximum flow that the associated edges can support. We are interested in finding the maximum total flow from \( u \) to \( v \) that the network can support.

The above definition is more general than what is typical. The standard definition prohibits incoming edges to the source and outgoing edges from the sink. However, this more general definition is useful for the development of our algorithms.

In order to define formally a network flow, we need some additional notation. For a vertex \( x \) in a directed graph, let

- \( x^- = \{(w, x) \in E\} \), the set of incoming edges to \( x \); and
A flow for a flow network \((V, E, u, v, C)\) is function \(F : E \to \mathbb{N}\) such that

- for each \(e \in E\), \(F(e) \leq C(e)\); and
- for each vertex \(x \in V \setminus \{u, v\}\),
  \[
  \sum_{e \in x^-} F(e) = \sum_{e \in x^+} F(e).
  \]

Thus, the flow on each edge is no more than that edge’s capacity, and the total flow into a vertex other than the source or the sink is the same as the total flow out of that vertex. An example of a flow on the network shown in Figure 14.1 would have a flow of 1 on every edge except (4, 1); this edge would have a flow of 0.

We leave it as an exercise to show that for any flow \(F\) of a flow network \((G, u, v, C)\),

\[
\sum_{e \in u^-} F(e) - \sum_{e \in u^+} F(e) = \sum_{e \in v^-} F(e) - \sum_{e \in v^+} F(e). \quad (14.1)
\]

We therefore define the value of a flow to be the above difference — the net flow out of the source, or equivalently, the net flow into the sink. Thus,
the flow described above for the network in Figure 14.1 has a value of 2. Given a flow network, the network flow problem is to find a network flow with maximum value. Clearly, 2 is the maximum value of any flow for the network in Figure 14.1.

In the next two sections, we will examine algorithms for the network flow problem. In the remainder of the chapter, we will consider the bipartite matching problem, and show how to reduce it to network flow.

### 14.1 The Ford-Fulkerson Algorithm

A flow for a given network can be found by finding a simple path (i.e., one in which no vertices are repeated) from the source to the sink. We will refer to such a path as an augmenting path. If no augmenting path exists, then the maximum flow must be 0. Otherwise, suppose $m$ is the minimum capacity of any edge in some particular augmenting path. We can clearly place a flow of $m$ on each edge in that path. We can then solve the smaller network flow problem obtained by reducing the capacity of each edge on the augmenting path by $m$, and removing any edge whose capacity would become 0. If we measure the size of a problem instance as the value of a maximum flow, then the resulting problem instance is clearly smaller. We can then combine the flow obtained from the solution to the smaller problem with the first flow we obtained.

The above approach clearly finds a flow for a given network. Unfortunately, that flow is not guaranteed to be a maximum flow. To see why, consider the flow network in Figure 14.1. Suppose the first augmenting path found is $(0, 2, 4, 1, 3, 5)$. We can put a flow of 1 on each of these edges. In the smaller instance, each of these edges would be removed, so that there is no augmenting path in the resulting network. The flow found on the smaller instance is therefore empty, so that the final flow has value 1. As we have already seen, a maximum flow for this network has value 2.

The most obvious approach to repairing this algorithm is to be more careful in how we choose the augmenting path. However, it turns out that a more straightforward approach is to be more careful in how we construct the smaller problem instance. The problem with the reduction described above is that once we decide to place a flow on an edge, we cannot reverse that decision. If we are more careful, we can construct a smaller instance that allows us to reverse these decisions.

Specifically, when we decrease the capacity of an edge by $m$, we also increase by $m$ the capacity of the edge going the opposite direction. If there
is no such edge, we add it to the graph. When we combine the two flows, we allow flows in opposite directions to cancel each other; i.e., if edge \((x, y)\) has flow \(k\) and edge \((y, x)\) has flow \(k' \leq k\), we set the flow on \((x, y)\) to \(k - k'\) and the flow on \((y, x)\) to 0. Note that because any edge added to the graph by the reduction will have a capacity of \(m\), and the initial flow will be \(m\) in the opposite direction, the combination of the two flows will result in no flow on any edge that was added to the graph. We can therefore remove these edges from the resulting flow.

Let us now formalize the construction outlined above. Let \((G, u, v, C)\) be a flow network, and let \(P\) be the set of edges in some augmenting path. Let \(m\) be the minimum capacity of any edge in \(P\). We define the residual network of \((G, u, v, C)\) with respect to \(P\) to be the flow network \((G', u, v, C')\), where \(G'\) and \(C'\) are defined as follows:

- \(G'\) is constructed from \(G\) by removing any edges in \(P\) with capacity \(m\) and by adding edges \((y, x)\) such that \((x, y) \in P\) and \((y, x)\) is not an edge in \(G\).

- \(C'((x, y))\) is defined as follows for each edge \((x, y) \in G'\):
  - If \((x, y) \in P\), \(C'((x, y)) = C((x, y)) - m\).
  - If \((y, x) \in P\) and \((x, y)\) is an edge in \(G\), \(C'((x, y)) = C((x, y)) + m\).
  - If \((y, x) \in P\) and \((x, y)\) is not an edge in \(G\), \(C'((x, y)) = m\).
  - Otherwise, \(C'((x, y)) = C((x, y))\).

Thus, Figure 14.2(a) shows the residual network for the flow network in Figure 14.1 with respect to the augmenting path \(\langle 0, 2, 4, 1, 3, 5 \rangle\). This graph has an augmenting path: \(\langle 0, 1, 4, 5 \rangle\). The residual network with respect to this augmenting path is shown in Figure 14.2(b). There is no augmenting path in this graph. If we combine the flows obtained by assigning a flow of 1 to each edge in the respective paths, the flows on the edges \((4, 1)\) in the original graph and \((1, 4)\) in the graph in Figure 14.2(a) cancel each other out. The resulting flow therefore has a flow of 1 on each edge except \((4, 1)\) in the original network. This flow has a value of 2, which is maximum.

We now need to prove the correctness of this reduction. We begin by showing the following lemma.

**Lemma 14.1** Let \((G, u, v, C)\) be a flow network, and let \(P\) be the set of edges on some augmenting path. Let \(F_1\) be the flow obtained by adding a flow of \(m\) to each edge in \(P\), where \(m\) is the minimum capacity of any edge in \(P\). Let \(F_2\) be a maximum flow on the residual graph of \((G, u, v, C)\) with
Chapter 14. Network Flow and Matching

Figure 14.2 Examples of residual graphs

(a)

(b)

Proof: We must first show that the combination of the two flows does not give a flow where there is no edge in $G$. This can only happen if there is a flow in $F_2$ on an edge $(x, y)$ that is not in $G$. Then $(y, x)$ must be an edge in $P$. The capacity of $(x, y)$ in the residual graph is therefore $m$. Because the flow on $(y, x)$ in $F_1$ is $m$, the combination of $F_1$ and $F_2$ cannot give a positive flow on $(x, y)$.

We will now show that in the combination of $F_1$ with $F_2$, the flow on each edge $(x, y)$ is no more than $C((x, y))$. The only way this can happen is if there is positive flow on $(x, y)$ in $F_2$. We consider three cases.

Case 1: $(x, y) \in P$. Then $C'((x, y)) = C((x, y)) - m$. The sum of the two flows on $(x, y)$ is therefore at most $C((x, y))$.

Case 2: $(y, x) \in P$. Then $C'((x, y)) = C((x, y)) + m$. In the combination of $F_1$ with $F_2$, the flow on $(x, y)$ is its flow in $F_2$, minus $m$. This total flow can be no more than $C((x, y))$.

Case 3: $(x, y) \not\in P$ and $(y, x) \not\in P$. Then $C'((x, y)) = C((x, y))$. In the
CHAPTER 14. NETWORK FLOW AND MATCHING

combination of $F_1$ with $F_2$, the flow on $(x, y)$ is simply its flow in $F_2$, which can be no more than $C((x, y))$.

Finally, it is clear that for each vertex $w$ in $G$ other than $u$ and $v$, the total flow into $w$ must equal the total flow out of $w$, and that the total flow out of $u$ is $k + m$. □

Using the above Lemma, we can prove the theorem below. Combined with Lemma 14.1, this theorem ensures that the reduction yields a maximum flow for the given network.

**Theorem 14.2** Let $(G, u, v, C)$ be a flow network with maximum flow $k$, and let $P$ be the set of edges in some augmenting path. Let $m$ be the minimum capacity of any edge in $P$, and let $(G', u, v, C')$ be the residual network of $(G, u, v, C)$ with respect to $P$. Then the maximum flow for $(G', u, v, C')$ has value $k - m$.

**Proof:** Let $F_1$ be a maximum flow for $(G, u, v, C)$. Then $F_1$ has value $k$. To simplify our discussion, let us interpret $F_1$ as a function $F_1 : V \times V \rightarrow \mathbb{Z}$, where $V$ is the set of vertices in $G$, such that $F_1(x, y)$ gives the flow over $(x, y)$ minus the flow over $(y, x)$; if either of these edges does not exist, we use 0 for this edge’s flow.

Let us assign flows to the edges of $G'$ as follows:

1. If $(x, y) \in P$ and $F_1(x, y) \geq m$, assign a flow of $F_1(x, y) - m$ to $(x, y)$. Because $C'(x, y)) = C((x, y)) - m$, the flow on this edge does not exceed the capacity.

2. If $(x, y) \in P$ and $F_1(x, y) < m$, assign a flow of $m - F_1(x, y)$ to $(y, x)$. Because $C'(y, x)) = C((y, x)) + m$ if $(y, x) \in G$, or $C'(y, x)) = m$ if $(y, x) \notin G$, the flow on this edge does not exceed the capacity.

3. If $(x, y) \notin P$, $(y, x) \notin P$, and $F_1(x, y) > 0$, assign a flow of $F_1(x, y)$ to $(x, y)$. Because $C'(x, y)) = C((x, y))$, the flow on this edge does not exceed its capacity.

4. Assign a flow of 0 to all other edges.

Because the above construction essentially reduces the flow $F_1$ by $m$ along $P$, the sum of incoming flows equals the sum of outgoing flows for all vertices in $G'$ except $u$ and $v$. This assignment is therefore a network flow, which we will denote $F_2$. Furthermore, $P$ contains exactly one edge incident on $u$ — the first edge, which we will call $(u, w)$. The effect of Steps
1 and 2 therefore decreases the net flow from \( u \) by \( m \). The value of \( F_2 \) is therefore \( k - m \). From Lemma 14.1, the combination of \( F_2 \) with a flow of \( m \) along \( P \) gives a flow of \( k \) for \( (G, u, v, C) \). Because this is true for any flow on \( (G', u, v, C') \), and because \( k \) is the maximum flow for \( (G, u, v, C) \), we conclude that \( F_2 \) must be a maximum flow for \( (G', u, v, C') \).

Though this reduction is not a transformation, we can implement it using a loop by maintaining a graph in which the flows on augmenting paths are combined as they are found. The resulting algorithm, known as the Ford-Fulkerson algorithm, is shown in Figure 14.3. Because the Graph ADT (see Figure 9.3 on page 312) provides no operation for removing an edge, we will allow the residual graph to have edges with capacity 0. The proof of correctness is easily shown using Theorem 14.2; the details are left as an exercise.

Four auxiliary functions are specified in Figure 14.3. We leave it as exercises to show that if \( G \) is represented as a ListGraph, then

- CopyGraph can be implemented to return a ListGraph and to run in \( O(n + a) \) time, where \( n \) is the number of vertices and \( a \) is the number of edges in \( G \); and

- AddFlow can be implemented to run in \( O(n + a) \) time, where \( n \) is the number of vertices in \( F \) (or equivalently \( R \)) and \( a \) is the number of edges in \( F \) and \( R \) together, assuming \( F \) and \( R \) are implemented as ListGraphs.

Furthermore, MinVal can clearly be implemented to run in \( O(n) \) time, where \( n \) is the number of Edges in \( L \), and FindPath can be implemented to run in \( O(n + a) \) time using either depth-first search or breadth-first search (see Exercise 11.6), where \( n \) is the number of vertices and \( a \) is the number of edges in its first argument \( G \).

Note that FindPath does not specify which augmenting path will be chosen. As a result, the Ford-Fulkerson algorithm can perform very poorly. Consider, for example, the flow network shown in Figure 14.4(a), where \( k \) is some large integer. It is easily seen by inspection that the maximum flow is \( 2k \). Suppose the algorithm first selects the augmenting path \( \langle 0, 1, 2, 3 \rangle \). The minimum capacity on this path is 1, and the resulting residual graph is shown in Figure 14.4(b). Suppose the algorithm then chooses the augmenting path \( \langle 0, 2, 1, 3 \rangle \). The minimum capacity is again 1, and the resulting residual graph is shown in Figure 14.4(c). It is easily seen that this process can continue increasing the flow by 1 until the maximum flow of \( 2k \) is achieved.
**Figure 14.3** The Ford-Fulkerson algorithm for network flow

**Precondition:** $G$ is directed Graph in which each edge contains a Nat giving its capacity, and source and sink are distinct Nats less than the number of vertices in $G$.

**Postcondition:** Returns a directed graph $F$ in which the contents of the edges give a maximum flow for $G$ from source to sink.

```
NETWORKFLOW(G, source, sink)
    F ← COPYGRAPH(G, true); R ← COPYGRAPH(G, false)
    P ← FINDPATH(R, source, sink)
    // Invariant: F combined with a maximum flow for R gives a maximum
    // flow for G. P is a CONSLIST containing the EDGES of a path from
    // source to sink in R if there is such a path; otherwise, P = nil.
    while P ≠ nil
        m ← MINVAL(P); ADDFLOW(F, P, R, m)
    P ← FINDPATH(R, source, sink)
    return F
```

**Precondition:** $G$ is a Graph, and zeroEdges is a boolean.

**Postcondition:** Returns a copy of $G$. If zeroEdges is true, the contents of all edges are set to 0; otherwise, they are unchanged.

```
COPYGRAPH(G, zeroEdges)
```

**Precondition:** $G$ is a Graph whose edges contain natural numbers, and $i$ and $j$ are distinct natural numbers strictly less than the number of vertices in $G$.

**Postcondition:** Returns a CONSLIST $P$ containing the EDGES in a simple path of non-zero EDGES from $i$ to $j$ in $G$. If no such path exists, returns nil.

```
FINDPATH(G, i, j)
```

**Precondition:** $L$ is a CONSLIST of EDGES containing positive integers.

**Postcondition:** Returns the minimum integer stored on any EDGE in $L$.

```
MINVAL(L)
```

**Precondition:** $F$ and $R$ are directed GRAPHS having the same number of vertices and whose edges contain natural numbers. $P$ is a CONSLIST of EDGES forming a simple path in $R$, and $m$ is a positive integer.

**Postcondition:** Adds a flow of $m$ to each edge in $F$ that appears in $P$ and sets $R$ to the resulting residual graph. Edges are added to each GRAPH if necessary.

```
ADDFLOW(F, P, R, m)
```
Figure 14.4 A flow network on which NetworkFlow can perform poorly

(a)  
\[
\begin{array}{c}
\text{source} \\
0 \\
\text{k} \\
1 \\
2 \\
\text{k} \\
3 \\
\text{sink}
\end{array}
\]

(b)  
\[
\begin{array}{c}
\text{source} \\
0 \\
\text{k} \\
1 \\
2 \\
\text{k} \\
3 \\
\text{sink}
\end{array}
\]

(c)  
\[
\begin{array}{c}
\text{source} \\
0 \\
\text{k} \\
1 \\
2 \\
\text{k} \\
3 \\
\text{sink}
\end{array}
\]

On the other hand the algorithm could have achieved the same flow with two augmenting paths: \( (0, 1, 3) \) and \( (0, 2, 3) \).

In the next section, we will consider how to make good augmenting path choices. For now, we note that in the worst case, the loop in NetworkFlow can iterate \( M \) times, where \( M \) is the value of the maximum flow. Assuming the initialization and the body of the loop are implemented to run in \( \Theta(n + a) \) time, where \( n \) is the number of vertices and \( a \) is the number of edges in \( G \), the algorithm runs in \( \Theta(M(n + a)) \) time in the worst case. If we assume that all vertices are reachable from the source, then \( a \geq n - 1 \), and we can simplify the running time to \( \Theta(Ma) \).

Before we move on to a discussion on finding augmenting paths, we note one additional property of the Ford-Fulkerson algorithm. As long as each augmenting path found is simple — and there is no reason a path-finding algorithm would find a path containing a cycle — the path will not contain any edges to the source or any edges from the sink. The obvious consequence is that if the graph contains edges into the source or out of the sink, they will not be used. A less obvious consequence is that after such edges are introduced into the residual graph, they will not be used. Thus, once a flow is added to an edge from the source or to the sink, the flow on that edge will never be decreased.

14.2 The Edmonds-Karp Algorithm

We have seen that the way augmenting paths are chosen by the Ford-Fulkerson algorithm can significantly impact its performance. In this sec-
tion, we consider how to select augmenting paths in order to avoid very bad performance. One might suppose that we should try to select edges with high capacity; however, the approach we take does not even consider the edge capacities. Rather, we instead select a shortest augmenting path, in terms of the number of edges. The resulting network flow algorithm is known as the Edmonds-Karp algorithm.

Exercise 11.6 on page 389 outlined the breadth-first search technique for finding a shortest path in $\Theta(n + a)$ time, where $n$ is the number of vertices and $a$ is the number of edges. We will therefore focus on analyzing the running time of the Edmonds-Karp algorithm. We first show the following lemma.

**Lemma 14.3** No iteration of the Edmonds-Karp algorithm can bring any vertex nearer to the source.

**Proof:** By contradiction. Let $R$ be the residual graph at the beginning of an iteration, and let $R'$ be the residual graph at the end of the iteration. Suppose some vertex $x$ is closer to the source $u$ in $R'$ than in $R$. Specifically, let $x$ be a vertex that is closest to $u$ in $R'$ of all such vertices. There must be some edge on the shortest path from $u$ to $x$ in $R'$ that is not in $R$, for otherwise this path would also be a path from $u$ to $x$ in $R$. Specifically, the last edge $(w, x)$ on this path must have been added, for otherwise $w$, which is closer than $x$ to $u$ in $R'$, would also be closer to $u$ in $R'$ than in $R$. This means that $(x, w)$ is on a shortest augmenting path in $R$, so that $x$ is closer than $w$ to $u$ in $R$. Therefore, $w$ is closer to $u$ in $R'$ than in $R$ — a contradiction. □

Knowing that no vertex ever gets any closer to the source over the course of the Edmonds-Karp algorithm, we will now prove a lemma that shows when a vertex must get farther away from the source. If a vertex is farther than $n - 1$ edges from the source, where $n$ is the number of vertices, then it must be unreachable. By Lemma 14.3, once a vertex becomes unreachable, it will never become reachable. This next lemma will therefore enable us to bound the number of iterations of the Edmonds-Karp algorithm.

**Lemma 14.4** Suppose that when some vertex $x$ is at a distance $d$ from the source $u$, the Edmonds-Karp algorithm removes an edge $(x, y)$. Suppose that later this edge is added again. Then after this edge is added, the distance from $u$ to $x$ is at least $d + 2$. 

Proof: If \((x, y)\) is removed, then this edge must be on a shortest augmenting path. Then before this edge is removed, the distance from \(u\) to \(y\) is \(d + 1\). When \((x, y)\) is added again, \((y, x)\) must be on a shortest augmenting path. From Lemma 14.3, the distance from \(u\) to \(y\) is still at least \(d + 1\) when this path is found. The distance to \(x\) is therefore at least \(d + 2\). From Lemma 14.3, the distance from \(u\) to \(x\) must still be at least \(d + 2\) after the edge \((x, y)\) is added.

\[ \Box \]

**Theorem 14.5** In the worst case, the Edmonds-Karp algorithm iterates no more than \(na\) times, where \(n\) is the number of vertices and \(a\) is the number of edges in \(G\).

Proof: Because a flow equal to the minimum edge capacity on a shortest augmenting path is added by each iteration, each iteration removes at least one edge. By Lemmas 14.3 and 14.4, no edge can be removed more than \(n/2\) times. When edges are added, they are always added in the opposite direction of an existing edge; hence, at most \(2a\) distinct edges ever appear in the residual graph. The loop can therefore iterate at most \(na\) times. \(\Box\)

If the initialization and the body of the loop are implemented to run in \(\Theta(n + a)\) time, we can conclude that the algorithm runs in \(O(na(n + a))\) time in the worst case. Furthermore, the analysis of the last section still applies, so that the running time is in \(O(\min(M, na(n + a)))\), where \(M\) is the value of the maximum flow. If we assume that every vertex is reachable from the source, we can simplify this to \(O(\min(Ma, na^2))\).

### 14.3 Bipartite Matching

A *matching* in an undirected graph is a subset of the edges such that no two edges are incident on the same vertex. In this section, we consider the problem of finding a matching of maximum size in a given bipartite graph, as defined in Exercise 13.13 on page 444. Specifically, we will show that this problem can be reduced to the network flow problem.

As an example, consider the bipartite graph shown in Figure 14.5. We claim that the heavier edges, namely, \(\{0, 4\}, \{2, 5\}, \text{and} \{3, 7\}\), form a matching of maximum size. Clearly, these edges form a matching because no two of them share a common vertex. To see that it is of maximum size, we first note that any larger matching must contain all of the vertices in \(\{0, 1, 2, 3\}\) as endpoints. However, the only edges incident on 1 and 3 are \(\{1, 7\}\) and
{3, 7}, respectively, and they share a common vertex. Hence, any matching must exclude either 1 or 3. Therefore, there is no matching of size larger than 3.

We will now show how to reduce bipartite matching to network flow. Given a bipartite graph $G$, we construct an instance of network flow as follows. For simplicity, we will assume that the vertices of the bipartite graph have already been partitioned into the sets $V_1$ and $V_2$ (see Exercise 13.13). We first direct all of the edges from $V_1$ to $V_2$. We then add a new source vertex $u$ and edges from $u$ to each vertex in $V_1$. Next, we add a new sink vertex $v$ and edges from each vertex in $V_2$ to $v$. Finally, we assign a capacity of 1 to each edge. See Figure 14.6 for the result of applying this reduction to the graph in Figure 14.5.

Consider any matching in the given bipartite graph. We can construct a flow in the constructed network by adding a flow of 1 to each edge in the matching, as well as to each edge leading to a matched vertex in $V_1$ and to each edge leading from a matched vertex in $V_2$. Clearly, any unmatched vertex from the bipartite graph will have a flow of 0 on all of its incoming and outgoing edges. Furthermore, each matched vertex in $V_1$ will have a flow of 1 on its incoming edge and a flow of 1 on the single outgoing edge in the matching. Likewise, each matched vertex in $V_2$ will have a flow of 1 on the single incoming edge in the matching and a flow of 1 on its outgoing edge. Thus, we have constructed a flow whose value is the number of edges in the matching.

Conversely, consider any flow on the constructed network. Because any vertex in $V_1$ can have an incoming flow of at most 1, at most one of its outgoing edges will contain a positive flow. Likewise, because any vertex in $V_2$ can have an outgoing flow of at most 1, at most one of its incoming edges will contain a positive flow. The edges from $V_1$ to $V_2$ containing positive flow
 therefore correspond to a matching in the bipartite graph. Furthermore, the value of the flow is the number of edges in the matching.

We conclude that the edges from $V_1$ to $V_2$ in a maximum flow for the constructed network are the edges in a maximum-sized matching for the given bipartite graph. The resulting flow network has $n+2$ vertices and $n+a$ edges, where $n$ and $a$ are the number of vertices and edges, respectively, in the bipartite graph. The Edmonds-Karp algorithm will therefore solve the constructed network flow instance in $O(M(n + a))$ time, where $M$ is the number of edges in the maximum-sized matching. Because $M$ can be no more than $n/2$, if we assume that each vertex is incident on at least one edge, the running time is in $O(na)$. Furthermore, it is not hard to construct the flow network in $O(n + a)$ time, so the bipartite matching problem can be solved in $O(na)$ time.

Rather than presenting the code for the reduction, let us first examine the reduction more carefully to see if we can optimize the bipartite matching algorithm. For example, the addition of new vertices and edges is only needed to form a flow network. We could instead adapt one of the network
flow algorithms to operate without the source and/or the sink explicitly represented.

We also note that as flow is added, the edges containing the flow — which are the edges of a matching — have their direction reversed. Rather than explicitly reversing the direction of the edges, we could keep track of which edges have been included in the matching in some other way. For example, we could use an array \texttt{matching[0..n-1]} such that \texttt{matching[i]} gives the vertex to which \texttt{i} is matched, or is \texttt{-1} if \texttt{i} is unmatched. Because a matching has at most one edge incident on any vertex, this may end up being a more efficient way of keeping track of the vertices adjacent (in the flow network) to vertices in \(V_2\). The maximum-sized matching could also be returned via this array.

As we observed at the end of Section 14.1, once flow is added to any edge from the source or to any edge to the sink, that flow is never removed. To put this in terms of the matching algorithm, once a vertex is matched, it remains matched, although the vertex to which it is matched may change. Furthermore, we claim that if we ever attempt to add a vertex \(w \in V_1\) to the current matching \(M\) and are unable to do so (i.e., there is no path from \(w\) to an unmatched vertex in \(V_2\)), then we will never be able to add \(w\) to the matching.

To see why this is true, notice that if there were a maximum-sized matching containing all currently matched vertices and \(w\), then there is a matching \(M'\) containing no other vertices from \(V_1\). If we delete all vertices from \(V_1\) that are unmatched in \(M'\), then \(M'\) is clearly a maximum-sized matching for the resulting graph. The Ford-Fulkerson algorithm must therefore be able to find a path that yields \(M'\) from \(M\).

As a result, we only need to do a single search from each vertex in \(V_1\).

The following theorem summarizes this property.

\textbf{Theorem 14.6} Let \(G\) be a bipartite graph, and let \(S\) be the set of vertices in some matching on \(G\). Suppose some maximum-sized matching includes all of the vertices in \(S\). Let \(i\) be some vertex not in \(S\). Then there is a maximum-sized matching including \(S \cup \{i\}\) if there is a matching including \(S \cup \{i\}\).

In order to implement the above optimizations, it is helpful to define a data structure called \texttt{MatchingGraph}, which implements the \texttt{Graph} ADT. Its purpose is to represent a particular directed graph \(G'\) derived from a given bipartite graph \(G\) and a matching on \(G\). Suppose \(G\) has vertices 0, 1, \ldots, \(n-1\). \(G'\) then contains the vertices 0, 1, \ldots, \(n\). If \(\{i, j\}\) is an edge in
CHAPTER 14. NETWORK FLOW AND MATCHING

Figure 14.7 The MatchingGraph for the bipartite graph shown in Figure 14.5 with matching \{\{0, 5\}, \{3, 7\}\}

$G$ and $j$ is unmatched, then $G'$ will contain the edge $(i, n)$. Every other edge in $G'$ will represent two edges in $G$ — an edge not in the matching followed by an edge in the matching. Thus, for $0 \leq i < n$, $0 \leq j < n$, and $i \neq j$, $G'$ contains the edge $(i, j)$ iff there is a vertex $k$ in $G$ such that \{k, j\} is in the matching and \{i, k\} is an edge in $G$. For example, Figure 14.7 shows the MatchingGraph for the bipartite graph of Figure 14.5 with matching \{\{0, 5\}, \{3, 7\}\}.

Suppose the two partitions of $G$ are $V_1$ and $V_2$. Then augmenting paths in the flow network constructed by the reduction correspond to paths from unmatched vertices in $V_1$ to $n$ in $G'$. In order to find an augmenting path in the flow network, we need to find a path to $n$ in $G'$ from an unmatched vertex in $G$. For example, consider the MatchingGraph shown in Figure 14.7. We can add vertex 2 to the matching by finding a path from 2 to 9 in $G'$.

Taking the path \langle 2, 0, 9 \rangle could yield the augmenting path \langle 2, 5, 0, 4 \rangle, which produces the matching shown in Figure 14.5. This path could also yield the augmenting path \langle 2, 5, 0, 6 \rangle because 0 is adjacent to two unmatched vertices, 4 and 6. Alternatively, taking the path \langle 2, 9 \rangle would yield the augmenting path \langle 2, 8 \rangle.

Note that $G'$ actually represents two flow networks. If $(i, j)$ is an edge in $G'$ and $j \neq n$, then $i$ and $j$ must both be in the same partition. Therefore, the subgraph induced by $V_1 \cup \{n\}$ represents the flow network in which edges
in the matching lead from $V_2$ to $V_1$. Symmetrically, the subgraph induced by $V_2 \cup \{n\}$ represents the flow network in which edges in the matching lead from $V_1$ to $V_2$ — i.e., the flow network that would be constructed by swapping $V_1$ with $V_2$. Thus, 4 could be added to the matching by finding a path from 4 to 9 in $G'$. The only such path, $\langle 4, 5, 9 \rangle$, would also yield the matching shown in Figure 14.5.

To implement a MatchingGraph, we need two representation variables:

- a `Graph` bipartite representing the bipartite graph; and
- a readable array `matching[0..n-1]` representing the matching, so that `matching[i]` gives the vertex to which $i$ is matched, or $-1$ if $i$ is unmatched.

Its structural invariant will be that for $0 \leq i < n$, if `matching[i]` $\neq -1$, then `matching[matching[i]] = i`.

A partial implementation is shown in Figure 14.8 — we only include implementations of those operations we will actually be using. These operations include an additional operation for adding an edge to the matching, while removing any edges that might be incident on either endpoint. We also include a constructor that constructs a MatchingGraph from a given bipartite graph with an empty matching. We use the `data` variable of an `Edge` to store the intermediate vertex between the two edges of the bipartite graph represented by that `Edge`.

Note that this implementation is not secure, because its constructor allows an outside reference to `bipartite`, and because `matching` is readable. We could easily modify the implementation so that the constructor stores a copy of its input graph and the Matching operation returns a copy of `matching`; however, if we write our matching algorithm so that it doesn’t change these items except via operations in MatchingGraph, we can avoid this extra copying.

In order to complete the matching algorithm, we need to be able to find a path from $i$ to $j$ in a directed graph. Because the maximum flow on the constructed flow network is no more than $n/2$, where $n$ is the number of vertices in the bipartite graph, the Ford-Fulkerson algorithm will not perform badly. Therefore, we will use depth-first search to find paths. (We leave it as an exercise to implement the algorithm using breadth-first search.)

We therefore need a Searcher with the following representation variable:

- `incoming[0..n-1]`: a readable array of Edges giving the incoming edge in the depth-first spanning tree for each vertex reached.
Figure 14.8 MatchingGraph implementation of Graph (partial)

Structural Invariant: For \(0 \leq i < n\), if \(\text{matching}[i] \neq -1\), then \(\text{matching}[\text{matching}[i]] = i\).

Precondition: \(G\) refers to a bipartite Graph.
Postcondition: Constructs a MatchingGraph representing \(G\) with an empty matching.

\[
\text{MatchingGraph}(G) \\
\quad n \leftarrow G.\text{Size}(); \quad \text{bipartite} \leftarrow G; \quad \text{matching} \leftarrow \text{new Array}[0..n - 1] \\
\quad \text{for } i \leftarrow 0 \text{ to } n - 1 \\
\quad \quad \text{matching}[i] \leftarrow -1
\]

\[
\text{MatchingGraph.Size}() \\
\quad \text{return bipartite.Size}() + 1
\]

\[
\text{MatchingGraph.AllFrom}(i) \\
\quad n \leftarrow \text{bipartite.Size}(); \quad L \leftarrow \text{new ConsList}() \\
\quad \text{if } i < n \\
\quad \quad \text{foundUnmatched} \leftarrow \text{false}; \quad \text{adj} \leftarrow \text{bipartite.AllFrom}(i) \\
\quad \quad \text{while not adj.\text{isEmpty}()} \\
\quad \quad \quad e \leftarrow \text{adj.\text{head}()}; \quad \text{adj} \leftarrow \text{adj.\text{tail}()} \\
\quad \quad \quad k \leftarrow e.\text{dest}(); \quad j \leftarrow \text{matching}[k] \\
\quad \quad \quad \text{if } j = -1 \text{ and not foundUnmatched} \\
\quad \quad \quad \quad L \leftarrow \text{new ConsList(new Edge}(i, n, k), L) \\
\quad \quad \quad \quad \text{foundUnmatched} \leftarrow \text{true} \\
\quad \quad \text{else if } j \neq -1 \\
\quad \quad \quad L \leftarrow \text{new ConsList(new Edge}(i, j, k), L) \\
\quad \quad \text{return } L
\]

Precondition: \(i\) and \(j\) are distinct Nats representing vertices in the bipartite graph.
Postcondition: Adds \(\{i, j\}\) to the matching, removing from the matching any edge incident on \(i\) or \(j\).

\[
\text{MatchingGraph.Match}(i, j) \\
\quad \text{if } \text{matching}[i] \neq -1 \\
\quad \quad \text{matching}[\text{matching}[i]] \leftarrow -1 \\
\quad \text{if } \text{matching}[j] \neq -1 \\
\quad \quad \text{matching}[\text{matching}[j]] \leftarrow -1 \\
\quad \text{matching}[i] \leftarrow j; \quad \text{matching}[j] \leftarrow i
\]
The implementation of PathSearcher is shown in Figure 14.9.

By Theorem 14.6, a depth-first search of a MatchingGraph can be used to determine whether a given vertex can be safely matched. Note that this theorem doesn’t specify which partition the vertex comes from. In particular, we really don’t need to know the partition to which any vertex belongs — we can simply test them in any order, and add the ones that can be safely added. We therefore no longer need to require that the first $k$ vertices form the first partition. The algorithm is shown in Figure 14.10. Note that $in$ and $M$ maintain references to the incoming variable in $s$ and the matching variable of matchGraph, respectively (this would not be possible if PathSearcher and MatchingGraph were secure). Note also that as long as an augmenting path is not found, we do not need to select any unselected nodes because no unselected node leads to $n$.

Let $n$ be the number of vertices and $a$ be the number of edges in $G$. To simplify the analysis of the running time, suppose each vertex has at least one incident edge, so that $n \in O(a)$. Let us first focus on a single iteration of the for loop. Clearly, the running time of the call to Dfs is in $O(a)$. The number of iterations of the inner loop is at most the current size of the matching, so its running time is in $O(n) \subseteq O(a)$. The call to SelectAll also runs in $O(n) \subseteq O(a)$ time. We therefore conclude that a single iteration of the for loop runs in $O(a)$ time, so that the entire algorithm runs in $O(na)$ time.

To show that the running time of the algorithm is in $\Omega(na)$, we will first construct a graph with $4k$ vertices and $4k - 1$ edges for $k \in \mathbb{N}$. We will show that the algorithm runs in $\Omega(k^2)$ time for these graphs. We will then generalize the construction to an arbitrary number $n$ of vertices and $a$ edges such that $n - 1 \leq a < n(n + 20)/32$. We will show that the algorithm runs in $\Omega(na)$ time for these graphs.
**Precondition:** $G$ is a bipartite graph.

**Postcondition:** Returns an array $M[0..n-1]$ describing a maximum-sized matching of $G$, so that $M[i] = j$ if $j$ and $i$ are matched, and $M[i] = -1$ if $i$ is unmatched.

**Algorithm:**

```plaintext
Matching(G)
    n ← G.Size(); sel ← new Selector(n)
    s ← new PathSearcher(n + 1)
    matchGraph ← new MatchingGraph(G)
    in ← s.Incoming(); M ← matchGraph.Matching()

    // Invariant: $M$ represents a matching, and there is no matching
    // containing the matched vertices in $M$ and any unmatched vertex $j < i$.
    for i ← 0 to $n - 1$
        if $M[i] = -1$
            DFS(matchGraph, i, sel, s)
            if not sel.IsSelected(n)
                j ← n
                while j ≠ i
                    e ← in[j]; k ← e.Source()
                    matchGraph.Match(k, e.Data()); j ← k
                sel.SelectAll()
    return M
```

We begin by setting $V = \{i \mid 0 \leq i < 4k\}$ (refer to Figure 14.11 for the case in which $k = 4$). We then add the following edges:

- for $0 \leq i < k$, the edges $\{2i, 2k+i\}$ and $\{2i+1, 2k+i\}$;

- for $0 < i < k$, the edge $\{2i, 2k+i-1\}$; and

- for $0 \leq i < k$, the edge $\{2k-1, 3k+i\}$.

We arrange the edges so that when we try to add vertex $2i$ for $0 \leq i < k$, we first encounter the edge $\{2i, 2k+i\}$. Because $2k+i$ is not in the matching, it is added. It will then be impossible to add vertex $2i+1$, but each node $2k+j$, for $0 \leq j < i$, will be reached in the search for an augmenting path. (For example, consider the search when trying to add 5
to the matching \{\{0, 8\}, \{2, 9\}, \{4, 10\}\} in Figure 14.11.) Constructing this matching therefore uses \(\Omega(k^2)\) time.

We can now generalize the above construction to arbitrary \(n\) by adding or removing a few vertices adjacent to \(2k - 1\). Furthermore, we can add edges \(\{2i, 2k + j\}\) for \(0 \leq i < k\) and \(0 \leq j < i - 1\) without increasing the size of the maximum-sized matching. However, these additional edges must all be traversed when we try to add vertex \(2i + 1\) to the matching. This construction therefore forces the algorithm to use \(\Omega(na)\) time. Furthermore, the number of edges added can be as many as

\[
\sum_{i=0}^{k-1} (i - 1) = \frac{k(k - 1)}{2} - k \\
= \frac{k^2 - 3k}{2} \\
= \frac{n^2 - 12n}{32}.
\]

Including the \(n - 1\) original edges, the total number of edges \(a\) is in the range

\[
 n - 1 \leq a < \frac{n(n + 20)}{32}.
\]

The above construction is more general than we really need, but its generality shows that some simple modifications to the algorithm won’t improve its asymptotic running time. For example, the graph is connected, so processing connected components separately won’t help. Also, the two partitions are the same size, so processing the smaller (or larger) partition first won’t help either. Furthermore, using breadth-first search won’t help because it will process just as many edges when no augmenting path exists.
On the other hand, this algorithm is not the most efficient one known for this problem. In the exercises, we explore how it might be improved.

Although the optimizations we made over a direct reduction to network flow did not improve the asymptotic running time of the algorithm, the resulting algorithm may have other advantages. For example, suppose we are trying to match jobs with job applicants. Each applicant may be qualified for several jobs. We wish to fill as many jobs as possible, but still assign jobs so that priority is given to those who applied earlier. If we process the applicants in the order in which they applied, we will obey this priority.

### 14.4 Summary

The network flow problem is a general combinatorial optimization problem to which many other problems can be reduced. Although the Ford-Fulkerson algorithm can behave poorly when the maximum flow is large in comparison to the size of the graph, its flexibility makes it useful for those cases in which the maximum flow is known to be small. For cases in which the maximum flow may be large, the Edmonds-Karp algorithm, which is simply the Ford-Fulkerson algorithm using breadth-first search to find augmenting paths, performs adequately.

The bipartite matching problem is an example of a problem which occurs quite often in practice and which can be reduced to network flow to yield a reasonably efficient algorithm. Furthermore, a careful study of the reduction yields insight into the problem that leads to a more general algorithm.

### 14.5 Exercises

**Exercise 14.1** Prove Equation (14.1) on page 446. [*Hint: Show by induction that the net flow out of any set of vertices including the source but not the sink is equal to the left-hand side.]*

**Exercise 14.2** Prove that NETWORKFLOW, shown in Figure 14.3, meets its specification.

**Exercise 14.3** Implement COPYGRAPH, specified in Figure 14.3, to return a LISTGRAPH and run in $O(n + a)$ time, where $n$ is the number of vertices and $a$ is the number of edges in the given graph.

**Exercise 14.4** Implement ADDFLOW, specified in Figure 14.3, to run in $O(n + a)$ time, where $n$ is the number of vertices in $F$ and $a$ is the number
CHAPTER 14. NETWORK FLOW AND MATCHING

of edges in $F$ and $R$ together. For the purposes of your analysis, you may assume that $F$ and $R$ are implemented as ListGraphs, and that the edges in $P$ form a simple path in $R$.

* Exercise 14.5 Suppose we generalize the network flow problem to allow positive rational edge capacities. Prove that the Ford-Fulkerson algorithm always finds a maximum flow for such a network.

** Exercise 14.6 Suppose we generalize the network flow problem to allow positive real edge capacities. Give such a flow network for which the Ford-Fulkerson algorithm does not terminate and does not converge to a maximum flow.

Exercise 14.7 Implement the matching algorithm using breadth-first search (see Exercise 11.6) to find augmenting paths.

** Exercise 14.8 Let $G$ be a bipartite graph whose partitions are \( \{ i \mid 0 \leq i < k \} \) and \( \{ i \mid k \leq i < n \} \), and let $M$ be a matching on $G$ smaller than the maximum size. Suppose the minimum length of any augmenting path in $G$ is $l$. Let $S = \{ P_1, \ldots, P_m \}$ be a maximal set of vertex-disjoint augmenting paths of length $l$; i.e., any augmenting path of length $l$ shares at least one vertex with some path in $S$. We define the symmetric difference of two sets $A$ and $B$ as $A \oplus B = (A \cup B) \setminus (A \cap B)$; thus, the symmetric difference is the set of elements in exactly one of the two sets.

a. Prove that $M' = M \oplus (P_1 \cup \cdots \cup P_m)$ is a matching with $m$ more edges than $M$.

b. Prove that any augmenting path in $M'$, where $M'$ is as defined above, has more than $l$ edges. [Hint: For the case in which $P$ shares a vertex with some $P_i$, define $T = (M \oplus M') \oplus P$. Prove that $T = (P_1 \cup \cdots \cup P_m) \oplus P$ and that $T$ has at least $(m + 1)l$ edges.]

c. Prove that the size of every matching exceeds the size of $M$ by no more than $n/l$.

d. Give an algorithm to find a maximal set of minimum-length augmenting paths. Your algorithm should run in $O(a)$ time, where $a$ is the number of edges in $G$, assuming $G$ is represented as a ListGraph
CHAPTER 14. NETWORK FLOW AND MATCHING

and \( n \in O(a) \). [Hint: Use both breadth-first search and a modified depth-first search.]

e. Give an \( O(a\sqrt{n}) \) algorithm to find a maximum-sized matching in \( G \).

**Exercise 14.9** Suppose we modify the network flow problem so that the input includes an array \( \text{cap}[0..n-1] \) of integers such that for each vertex \( i \), \( \text{cap}[i] \) gives an upper bound on the flow we allow to go to and from vertex \( i \). Show how to reduce this problem to the ordinary network flow problem. Your reduction must run in \( O(n+a) \) time, where \( n \) is the number of vertices and \( a \) is the number of edges in the graph.

**Exercise 14.10** We define an \( n \times n \) grid to be an undirected graph \((V,E)\) where \( V = \{ (i, j) \mid 1 \leq i \leq n, 1 \leq j \leq n \} \), and two vertices \((i, j)\) and \((i', j')\) are adjacent iff either \( i = i' \) and \( j = j' \pm 1 \) or \( j = j' \) and \( i = i' \pm 1 \). Thus, each vertex in a grid has at most 4 neighbors. We call the vertices with fewer than 4 neighbors boundary vertices (i.e., these are vertices \((1, j)\), \((n, j)\), \((i, 1)\), or \((i, n)\)). Give an \( O(mn^2) \) algorithm which takes a value \( n \in \mathbb{N} \) and \( m \leq n^2 \) starting vertices \((i, j) \in [1..n] \times [1..n]\) and determines whether there exists a set of \( m \) vertex-disjoint paths in the \( n \times n \) grid, each connecting a starting node with a boundary node. You may assume you have an algorithm for the problem stated in Exercise 14.9.

* **Exercise 14.11** A path cover of a directed graph is a set of paths such that every vertex is included in exactly one path. The size of a path cover is the number of paths in the set. Show how to reduce the problem of finding a minimum-sized path cover in a directed acyclic graph to the problem of finding a maximum-sized matching in a bipartite graph. The total running time of the algorithm should be in \( O(na) \).

* **Exercise 14.12** We are given two arrays of integers, \( R[1..m] \) and \( C[1..n] \) such that

\[
\sum_{i=1}^{m} R[i] = \sum_{i=1}^{n} C[i] = k.
\]

Give an \( O(kmn) \) algorithm that returns an \( m \times n \) matrix of 0s and 1s such that row \( i \) contains exactly \( R[i] \) 1s and column \( j \) contains exactly \( C[j] \) 1s, for all \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \). If there is no such matrix, your algorithm should return nil.
**Exercise 14.13** Given a connected undirected graph $G$, the edge connectivity of $G$ is the minimum number of edges whose removal would disconnect the graph. Give an $O(n^2a)$ algorithm to find the edge connectivity of a given connected undirected graph with $n$ vertices and $a$ edges. For your running time analysis, you may assume the graph is represented as a ListGraph. Prove the correctness of your algorithm.

14.6 Chapter Notes

The NetworkFlow algorithm is due to Ford and Fulkerson [40]. The running-time analysis of the use of breadth-first search in the Ford-Fulkerson algorithm is due to Edmonds and Karp [35] and Dinic [30]. Asymptotically faster algorithms exist — to date, the fastest known is due to Goldberg and Rao [55]. Their algorithm has a running time in

$$O\left(\min\left(n^{2/3}, a^{1/2}\right)a \log\left(n^2/a + 2\right) \log C\right),$$

where $C$ is the maximum capacity of any edge.

The technique of finding a maximum-sized matching using augmenting paths is due to Berge [14]. He showed that in an arbitrary undirected graph, a matching is of maximum size iff no augmenting path exists. Finding augmenting paths in arbitrary undirected graphs is more challenging, however, because we must avoid returning to the same vertex from which we started. The first efficient algorithm for finding an augmenting path in an arbitrary undirected graph is due to Edmonds [33]. The algorithm suggested by Exercise 14.8 is due to Hopcroft and Karp [61], and is the asymptotically fastest known algorithm for finding a maximum-sized matching in a bipartite graph. The structure of this exercise is based on a problem in Cormen, et al. [25]. An $O(a\sqrt{n})$ algorithm for arbitrary undirected graphs was later given by Micali and Vazirani [88].

A solution to Exercise 14.6 is given by Ford and Fulkerson [41].