Advanced Graph Algorithms

12

This chapter discusses a selection of advanced graph algorithms.

Section 12.1 presents an algorithm for finding the strongly connected components of a graph. After this, we will learn how to efficiently solve the 2SAT problem using the algorithm.

Section 12.2 focuses on Eulerian and Hamiltonian paths. An Eulerian path goes through each edge of the graph exactly once, and a Hamiltonian path visits each node exactly once. While the concepts look quite similar at first glance, the computational problems related to them are very different.

Section 12.3 first shows how we can determine the maximum flow from a source to a sink in a graph. After this, we will see how to reduce several other graph problems to the maximum flow problem.

Section 12.4 discusses properties of depth-first search and problems related to biconnected graphs.

12.1 Strong Connectivity

A directed graph is called *strongly connected* if there is a path from any node to all other nodes in the graph. For example, the left graph in Fig. 12.1 is strongly connected while the right graph is not. The right graph is not strongly connected, because, for example, there is no path from node 2 to node 1.

A directed graph can always be divided into strongly connected components. Each such component contains a maximal set of nodes such that there is a path from any node to all other nodes, and the components form an acyclic *component graph* that represents the deep structure of the original graph. For example, Fig. 12.2 shows a graph, its strongly connected components and the corresponding component graph. The components are $A = \{1, 2\}$, $B = \{3, 6, 7\}$, $C = \{4\}$, and $D = \{5\}$.

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_12

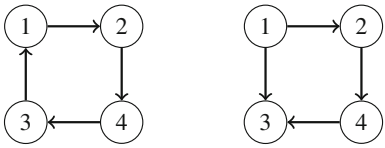


Fig. 12.1 The left graph is strongly connected, the right graph is not

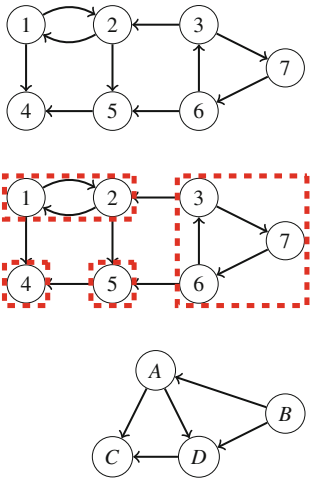


Fig. 12.2 A graph, its strongly connected components and the component graph

A component graph is a directed acyclic graph, so it is easier to process than the original graph. Since the graph does not contain cycles, we can always construct a topological sort and use dynamic programming to process it.

12.1.1 Kosaraju's Algorithm

Kosaraju's algorithm is an efficient method for finding the strongly connected components of a graph. The algorithm performs two depth-first searches: the first search constructs a list of nodes according to the structure of the graph, and the second search forms the strongly connected components.

The first phase of Kosaraju's algorithm constructs a list of nodes in the order in which depth-first search processes them. The algorithm goes through the nodes and begins a depth-first search at each unprocessed node. Each node will be added to the list after it has been processed.

For example, Fig. 12.3 shows the processing order of the nodes in our example graph. The notation x/y means that processing the node started at time x and finished at time y. The resulting list is [4, 5, 2, 1, 6, 7, 3]

The second phase of Kosaraju's algorithm forms the strongly connected components. First, the algorithm reverses every edge of the graph. This guarantees that during the second search, we will always find valid strongly connected components. Figure 12.4 shows the graph in our example after reversing the edges.

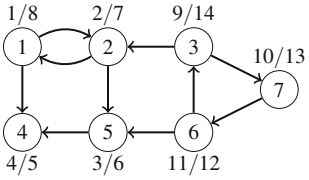
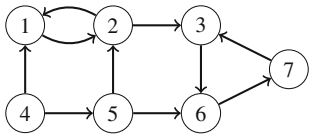


Fig. 12.3 The processing order of the nodes

Fig. 12.4 A graph with reversed edges



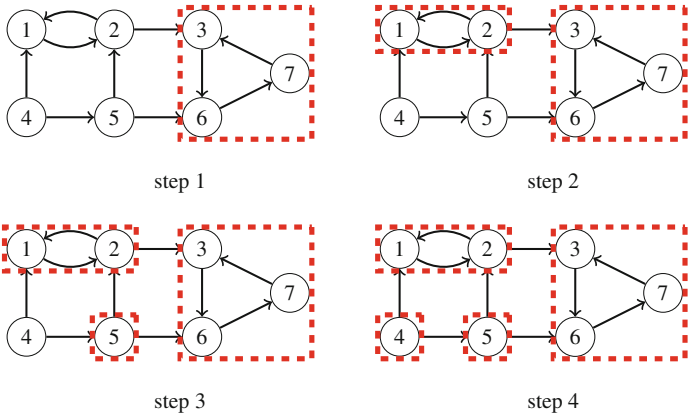


Fig. 12.5 Constructing the strongly connected components

After this, the algorithm goes through the list of nodes created by the first search, in *reverse* order. If a node does not belong to a component, the algorithm creates a new component by starting a depth-first search that adds all new nodes found during the search to the new component. Note that since all edges are reversed, the components do not "leak" to other parts of the graph.

Figure 12.5 shows how the algorithm processes our example graph. The processing order of the nodes is [3, 7, 6, 1, 2, 5, 4]. First, node 3 generates the component {3, 6, 7}. Then, nodes 7 and 6 are skipped, because they already belong to a component. After this, node 1 generates the component {1, 2}, and node 2 is skipped. Finally, nodes 5 and 4 generate the components {5} and {4}.

The time complexity of the algorithm is $O(n+m)$, because the algorithm performs two depth-first searches.

12.1.2 2SAT Problem

In the *2SAT problem*, we are given a logical formula

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

where each a_i and b_i is either a logical variable ($x_1, x_2, \ldots, x_n$) or a negation of a logical variable ($\neg x_1, \neg x_2, \ldots, \neg x_n$). The symbols "$\wedge$" and "$\vee$" denote logical operators "and" and "or." Our task is to assign each variable a value so that the formula is true, or state that this is not possible.

For example, the formula

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

is true when the variables are assigned as follows:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

However, the formula

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

is always false, regardless of how we assign the values. The reason for this is that we cannot choose a value for x_1 without creating a contradiction. If x_1 is false, both x_2 and $\neg x_2$ should be true which is impossible, and if x_1 is true, both x_3 and $\neg x_3$ should be true which is also impossible.

An instance of the 2SAT problem can be represented as an *implication graph* whose nodes correspond to variables x_i and negations $\neg x_i$, and edges determine the connections between the variables. Each pair $(a_i \vee b_i)$ generates two edges: $\neg a_i \rightarrow b_i$ and $\neg b_i \rightarrow a_i$. This means that if a_i does not hold, b_i must hold, and vice versa. For example, Fig. 12.6 shows the implication graph of L_1, and Fig. 12.7 shows the implication graph of L_2.

The structure of the implication graph tells us whether it is possible to assign the values of the variables so that the formula is true. This can be done exactly when there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly

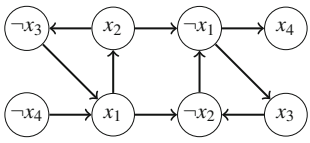


Fig. 12.6 The implication graph of L_1

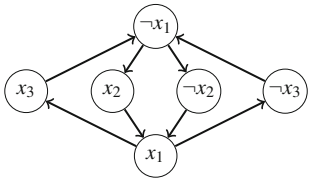


Fig. 12.7 The implication graph of L_2

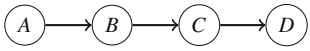


Fig. 12.8 The component graph of L_1

connected component. If there are such nodes, the graph contains a path from x_i to $\neg x_i$ and also a path from $\neg x_i$ to x_i, so both x_i and $\neg x_i$ should be true which is not possible. For example, the implication graph of L_1 does not have nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component, so there is a solution. Then, in the implication graph of L_2 all nodes belong to the same strongly connected component, so there are no solutions.

If a solution exists, the values for the variables can be found by going through the nodes of the component graph in a reverse topological sort order. At each step, we process a component that does not contain edges that lead to an unprocessed component. If the variables in the component have not been assigned values, their values will be determined according to the values in the component, and if they already have values, the values remain unchanged. The process continues until each variable has been assigned a value.

Figure 12.8 shows the component graph of L_1. The components are $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$, and $D = \{x_4\}$. When constructing the solution, we first process the component D where x_4 becomes true. After this, we process the component C where x_1 and x_2 become false and x_3 becomes true. All variables have been assigned values, so the remaining components A and B do not change the values of the variables.

Note that this method works, because the implication graph has a special structure: if there is a path from node x_i to node x_j and from node x_j to node $\neg x_j$, then node x_i never becomes true. The reason for this is that there is also a path from node $\neg x_j$ to node $\neg x_i$, and both x_i and x_j become false.

A more difficult problem is the *3SAT problem*, where each part of the formula is of the form $(a_i \vee b_i \vee c_i)$. This problem is NP-hard, so no efficient algorithm for solving the problem is known.

12.2 Complete Paths

In this section we discuss two special types of paths in graphs: an Eulerian path is a path that goes through each edge exactly once, and a Hamiltonian path is a path that visits each node exactly once. While such paths look quite similar at first glance, the computational problems related to them are very different.

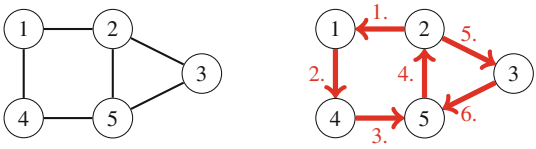


Fig. 12.9 A graph and an Eulerian path

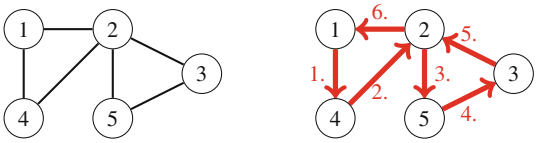


Fig. 12.10 A graph and an Eulerian circuit

12.2.1 Eulerian Paths

An *Eulerian path* is a path that goes exactly once through each edge of a graph. Furthermore, if such a path starts and ends at the same node, it is called an *Eulerian circuit*. Figure 12.9 shows an Eulerian path from node 2 to node 5, and Fig. 12.10 shows an Eulerian circuit that starts and ends at node 1.

The existence of Eulerian paths and circuits depends on the degrees of the nodes. First, an undirected graph has an Eulerian path exactly when all the edges belong to the same connected component and

- the degree of each node is even, *or*
- the degree of exactly two nodes is odd, and the degree of all other nodes is even.

In the first case, each Eulerian path is also an Eulerian circuit. In the second case, the odd-degree nodes are the endpoints of an Eulerian path, which is not an Eulerian circuit. In Fig. 12.9, nodes 1, 3, and 4 have degree 2, and nodes 2 and 5 have degree 3. Exactly two nodes have an odd degree, so there is an Eulerian path between nodes 2 and 5, but the graph does not have an Eulerian circuit. In Fig. 12.10, all nodes have an even degree, so the graph has an Eulerian circuit.

To determine whether a directed graph has Eulerian paths, we focus on indegrees and outdegrees of the nodes. A directed graph contains an Eulerian path exactly when all the edges belong to the same strongly connected component and

- in each node, the indegree equals the outdegree, *or*
- in one node, the indegree is one larger than the outdegree, in another node, the outdegree is one larger than the indegree, and in all other nodes, the indegree equals the outdegree.

In the first case, each Eulerian path is also an Eulerian circuit, and in the second case, the graph has an Eulerian path that begins at the node whose outdegree is larger and ends at the node whose indegree is larger. For example, in Fig. 12.11, nodes 1, 3, and 4 have both indegree 1 and outdegree 1, node 2 has indegree 1 and outdegree

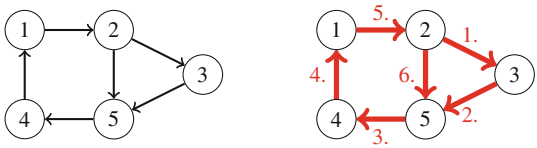


Fig. 12.11 A directed graph and an Eulerian path

2, and node 5 has indegree 2 and outdegree 1. Hence, the graph contains an Eulerian path from node 2 to node 5.

Construction *Hierholzer's algorithm* is an efficient method for constructing an Eulerian circuit for a graph. The algorithm consists of several rounds, each of which adds new edges to the circuit. Of course, we assume that the graph contains an Eulerian circuit; otherwise Hierholzer's algorithm cannot find it.

The algorithm begins with an empty circuit that contains only a single node and then extends the circuit step by step by adding subcircuits to it. The process continues until all edges have been added to the circuit. The circuit is extended by finding a node x that belongs to the circuit but has an outgoing edge that is not included in the circuit. Then, a new path from node x that only contains edges that are not yet in the circuit is constructed. Sooner or later, the path will return to node x, which creates a subcircuit.

If a graph does not have an Eulerian circuit but has an Eulerian path, we can still use Hierholzer's algorithm to find the path by adding an extra edge to the graph and removing the edge after the circuit has been constructed. For example, in an undirected graph, we add the extra edge between the two odd-degree nodes.

As an example, Fig. 12.12 shows how Hierholzer's algorithm constructs an Eulerian circuit in an undirected graph. First, the algorithm adds a subcircuit $1 \to 2 \to 3 \to 1$, then a subcircuit $2 \to 5 \to 6 \to 2$, and finally a subcircuit $6 \to 3 \to 4 \to 7 \to 6$. After this, since all edges have been added to the circuit, we have successfully constructed an Eulerian circuit.

12.2.2 Hamiltonian Paths

A *Hamiltonian path* is a path that visits each node of a graph exactly once. Furthermore, if a such a path begins and ends at the same node, it is called a *Hamiltonian circuit*. For example, Fig. 12.13 shows a graph that has both a Hamiltonian path and a Hamiltonian circuit.

Problems related to Hamiltonian paths are NP-hard: nobody knows a general way to efficiently check if a graph has a Hamiltonian path or circuit. Of course, in some special cases we can be certain that a graph contains a Hamiltonian path. For example, if the graph is complete, i.e., there is an edge between all pairs of nodes, it surely contains a Hamiltonian path.

A simple way to search for a Hamiltonian path is to use a backtracking algorithm that goes through all possible ways to construct a path. The time complexity of such

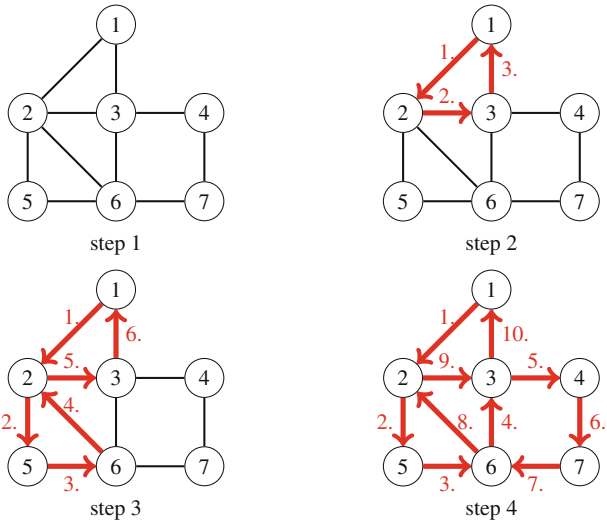


Fig. 12.12 Hierholzer's algorithm

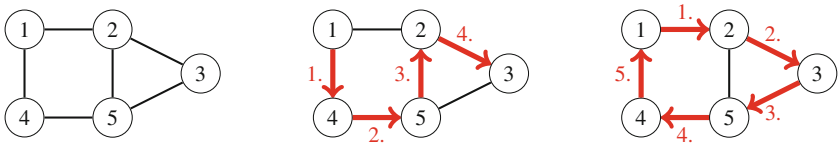


Fig. 12.13 A graph, a Hamiltonian path and a Hamiltonian circuit

an algorithm is at least $O(n!)$, because there are $n!$ different ways to choose the order of n nodes. Then, using dynamic programming, we can create a more efficient $O(2^n n^2)$ time solution, which determines for each subset of nodes S and each node $x \in S$ if there is a path that visits all nodes of S exactly once and ends at node x.

12.2.3 Applications

De Bruijn Sequences A *De Bruijn sequence* is a string that contains every string of length n exactly once as a substring, for a fixed alphabet of k characters. The length of such a string is $k^n + n - 1$ characters. For example, when $n = 3$ and $k = 2$, an example of a De Bruijn sequence is

0001011100.

The substrings of this string are all combinations of three bits: 000, 001, 010, 011, 100, 101, 110, and 111.

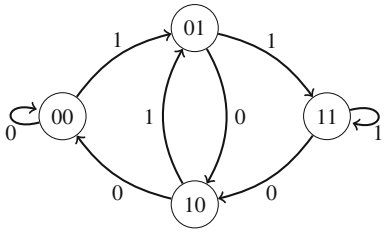


Fig. 12.14 Constructing a De Bruijn sequence from an Eulerian path

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Fig. 12.15 An open knight's tour on a 5×5 board

A De Bruijn sequence always corresponds to an Eulerian path in a graph where each node contains a string of $n - 1$ characters, and each edge adds one character to the string. For example, the graph in Fig. 12.14 corresponds to the scenario where $n = 3$ and $k = 2$. To create a De Bruijn sequence, we start at an arbitrary node and follow an Eulerian path that visits each edge exactly once. When the characters in the starting node and on the edges are added together, the resulting string has $k^n + n - 1$ characters and is a valid De Bruijn sequence.

Knight's Tours A *knight's tour* is a sequence of moves of a knight on an $n \times n$ chessboard following the rules of chess such that the knight visits each square exactly once. A knight's tour is called *closed* if the knight finally returns to the starting square and otherwise it is called *open*. For example, Fig. 12.15 shows an open knight's tour on a 5×5 board.

A knight's tour corresponds to a Hamiltonian path in a graph whose nodes represent the squares of the board, and two nodes are connected with an edge if a knight can move between the squares according to the rules of chess. A natural way to construct a knight's tour is to use backtracking. Since there is a large number of possible moves, the search can be made more efficient by using *heuristics* that attempt to guide the knight so that a complete tour will be found quickly.

Warnsdorf's rule is a simple and effective heuristic for finding a knight's tour. Using the rule, it is possible to efficiently construct a tour even on a large board. The idea is to always move the knight so that it ends up in a square where the number of possible follow-up moves is as *small* as possible. For example, in Fig. 12.16, there are five possible squares to which the knight can move (squares $a \ldots e$). In this situation, Warnsdorf's rule moves the knight to square a, because after this choice, there is only a single possible move. The other choices would move the knight to squares where there would be three moves available.

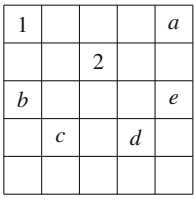


Fig. 12.16 Using Warndorf's rule to construct a knight's tour

12.3 Maximum Flows

In the *maximum flow* problem, we are given a directed weighted graph that contains two special nodes: a *source* is a node with no incoming edges, and a *sink* is a node with no outgoing edges. Our task is to send as much flow as possible from the source to the sink. Each edge has a capacity that restricts the flow that can go through the edge, and in each intermediate node, the incoming and outgoing flow has to be equal.

As an example, consider the graph in Fig. 12.17, where node 1 is the source and node 6 is the sink. The maximum flow in this graph is 7, shown in Fig. 12.18. The notation v/k means that a flow of v units is routed through an edge whose capacity is k units. The size of the flow is 7, because the source sends $3 + 4$ units of flow and the sink receives $5 + 2$ units of flow. It is easy to see that this flow is maximum, because the total capacity of the edges leading to the sink is 7.

It turns out that the maximum flow problem is connected to another graph problem, the *minimum cut* problem, where our task is to remove a set of edges from the graph such that there will be no path from the source to the sink after the removal and the total weight of the removed edges is minimum.

For example, consider again the graph in Fig. 12.17. The minimum cut size is 7, because it suffices to remove the edges $2 \to 3$ and $4 \to 5$, as shown in Fig. 12.19. After removing the edges, there will be no path from the source to the sink. The size of the cut is $6 + 1 = 7$, and the cut is minimum, because there is no valid cut whose weight would be less than 7.

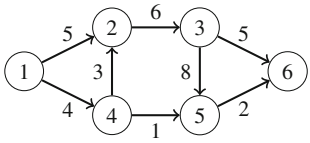


Fig. 12.17 A graph with source 1 and sink 6

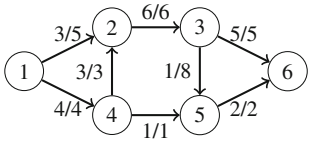


Fig. 12.18 The maximum flow of the graph is 7

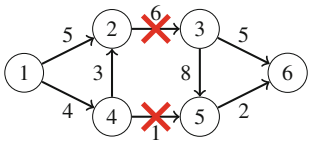


Fig. 12.19 The minimum cut of the graph is 7

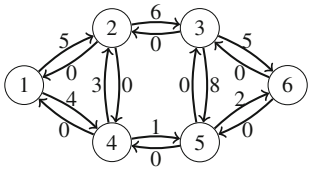


Fig. 12.20 Graph representation in the Ford–Fulkerson algorithm

It is not a coincidence that the maximum flow and minimum cut are equal in our example graph. Rather, it turns out that they are *always* equal, so the concepts are two sides of the same coin. Next we will discuss the Ford–Fulkerson algorithm that can be used to find the maximum flow and minimum cut of a graph. The algorithm also helps us to understand *why* they are equal.

12.3.1 Ford–Fulkerson Algorithm

The *Ford–Fulkerson algorithm* finds the maximum flow in a graph. The algorithm begins with an empty flow, and at each step finds a path from the source to the sink that generates more flow. Finally, when the algorithm cannot increase the flow anymore, the maximum flow has been found.

The algorithm uses a special graph representation where each original edge has a reverse edge in another direction. The weight of each edge indicates how much more flow we could route through it. At the beginning of the algorithm, the weight of each original edge equals the capacity of the edge, and the weight of each reverse edge is zero. Figure 12.20 shows the new representation for our example graph.

The Ford–Fulkerson algorithm consists of several rounds. On each round, the algorithm finds a path from the source to the sink such that each edge on the path has a positive weight. If there is more than one possible path available, any of them can be chosen. After choosing the path, the flow increases by x units, where x is the smallest edge weight on the path. In addition, the weight of each edge on the path decreases by x, and the weight of each reverse edge increases by x.

The idea is that increasing the flow decreases the amount of flow that can go through the edges in the future. On the other hand, it is possible to cancel flow later using the reverse edges if it turns out that it would be beneficial to route the flow in another way. The algorithm increases the flow as long as there is a path from the source to the sink through positive-weight edges. Then, if there are no such paths, the algorithm terminates and the maximum flow has been found.

Figure 12.21 shows how the Ford–Fulkerson algorithm finds the maximum flow for our example graph. In this case, there are four rounds. On the first round, the

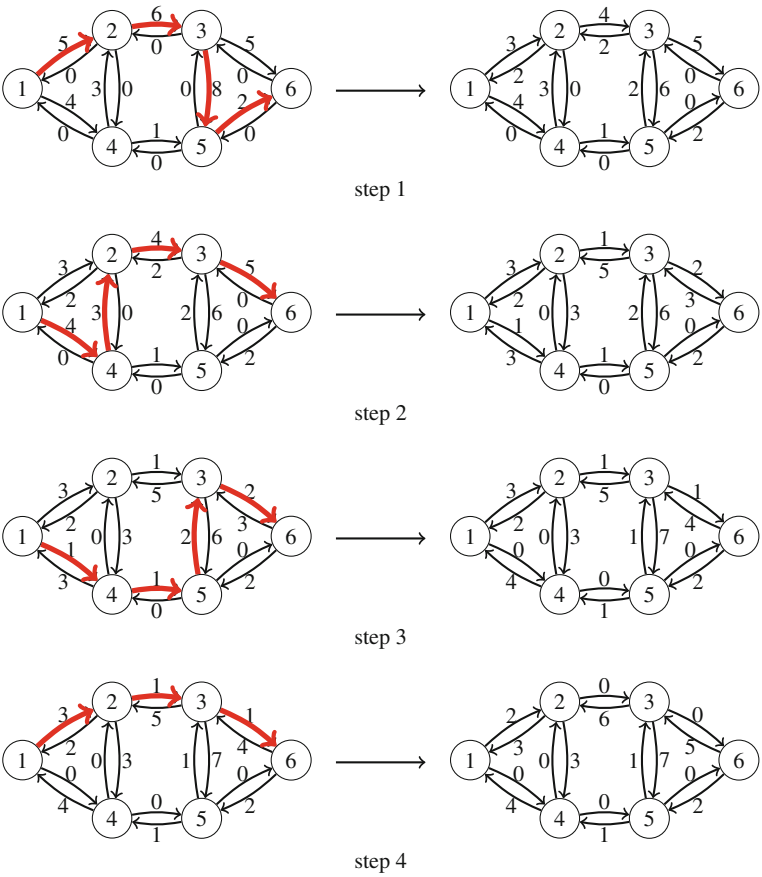


Fig. 12.21 The Ford–Fulkerson algorithm

algorithm chooses the path $1 \to 2 \to 3 \to 5 \to 6$. The minimum edge weight on this path is 2, so the flow increases by 2 units. Then, the algorithm chooses three other paths that increase the flow by 3, 1, and 1 units. After this, there is no path with positive-weight edges, so the maximum flow is $2 + 3 + 1 + 1 = 7$.

Finding Paths The Ford–Fulkerson algorithm does not specify how we should choose the paths that increase the flow. In any case, the algorithm will terminate sooner or later and correctly find the maximum flow. However, the efficiency of the algorithm depends on how the paths are chosen. A simple way to find paths is to use depth-first search. Usually this works well, but in the worst case, each path only increases the flow by one unit, and the algorithm is slow. Fortunately, we can avoid this situation by using one of the following techniques:

The *Edmonds–Karp algorithm* chooses each path so that the number of edges on the path is as small as possible. This can be done by using breadth-first search instead

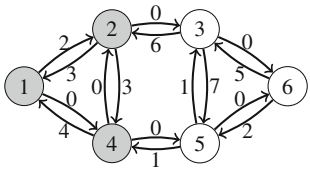


Fig. 12.22 Nodes 1, 2, and 4 belong to the set A

of depth-first search for finding paths. It can be proved that this guarantees that the flow increases quickly, and the time complexity of the algorithm is $O(m^2 n)$.

The *capacity scaling algorithm*[1] uses depth-first search to find paths where each edge weight is at least an integer threshold value. Initially, the threshold value is some large number, for example, the sum of all edge weights of the graph. Always when a path cannot be found, the threshold value is divided by 2. The algorithm terminates when the threshold value becomes 0. The time complexity of the algorithm is $O(m^2 \log c)$, where c is the initial threshold value.

In practice, the capacity scaling algorithm is easier to implement, because depth-first search can be used for finding paths. Both algorithms are efficient enough for problems that typically appear in programming contests.

Minimum Cuts It turns out that once the Ford–Fulkerson algorithm has found a maximum flow, it has also determined a minimum cut. Consider the graph produced by the algorithm, and let A be the set of nodes that can be reached from the source using positive-weight edges. Now the minimum cut consists of the edges of the original graph that start at some node in A, end at some node outside A, and whose capacity is fully used in the maximum flow. For example, in Fig. 12.22, A consists of nodes 1, 2, and 4, and the minimum cut edges are $2 \rightarrow 3$ and $4 \rightarrow 5$, whose weight is $6 + 1 = 7$.

Why is the flow produced by the algorithm maximum and why is the cut minimum? The reason is that a graph cannot contain a flow whose size is larger than the weight of any cut of the graph. Hence, always when a flow and a cut are equal, they are a maximum flow and a minimum cut.

To see why the above holds, consider any cut of the graph such that the source belongs to A, the sink belongs to B, and there are some edges between the sets (Fig. 12.23). The size of the cut is the sum of the weights of the edges that go from A to B. This is an upper bound for the flow in the graph, because the flow has to proceed from A to B. Thus, the size of a maximum flow is smaller than or equal to the size of any cut in the graph. On the other hand, the Ford–Fulkerson algorithm produces a flow whose size is *exactly* as large as the size of a cut in the graph. Thus, the flow has to be a maximum flow, and the cut has to be a minimum cut.

[1] This elegant algorithm is not very well known; a detailed description can be found in a textbook by Ahuja, Magnanti, and Orlin [1].

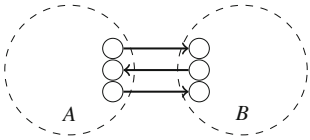


Fig. 12.23 Routing the flow from A to B

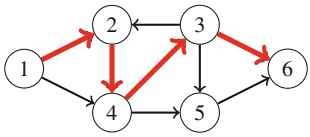
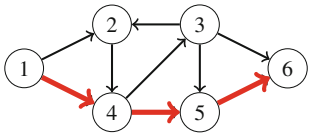


Fig. 12.24 Two edge-disjoint paths from node 1 to node 6



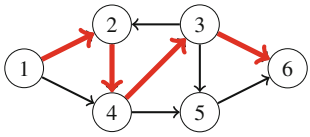


Fig. 12.25 A node-disjoint path from node 1 to node 6

12.3.2 Disjoint Paths

Many graph problems can be solved by reducing them to the maximum flow problem. Our first example of such a problem is as follows: we are given a directed graph with a source and a sink, and our task is to find the maximum number of disjoint paths from the source to the sink.

Edge-Disjoint Paths We first focus on the problem of finding the maximum number of *edge-disjoint paths* from the source to the sink. This means that each edge may appear in at most one path. For example, in Fig. 12.24, the maximum number of edge-disjoint paths is 2 ($1 \to 2 \to 4 \to 3 \to 6$ and $1 \to 4 \to 5 \to 6$).

It turns out that the maximum number of edge-disjoint paths always equals the maximum flow of the graph where the capacity of each edge is one. After the maximum flow has been constructed, the edge-disjoint paths can be found greedily by following paths from the source to the sink.

Node-Disjoint Paths Then, consider the problem of finding the maximum number of *node-disjoint paths* from the source to the sink. In this case, every node, except for the source and sink, may appear in at most one path, which may reduce the maximum number of disjoint paths. Indeed, in our example graph, the maximum number of node-disjoint paths is 1 (Fig. 12.25).

We can reduce also this problem to the maximum flow problem. Since each node can appear in at most one path, we have to limit the flow that goes through the nodes. A standard construction for this is to divide each node into two nodes such that

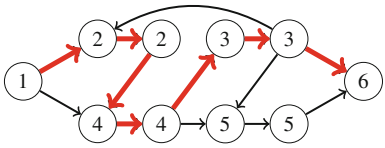


Fig. 12.26 A construction that limits the flow through the nodes

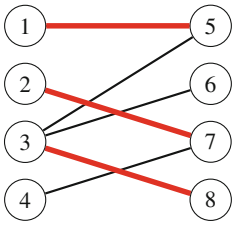


Fig. 12.27 Maximum matching

the first node has the incoming edges of the original node, the second node has the outgoing edges of the original node, and there is a new edge from the first node to the second node. Figure 12.26 shows the resulting graph and its maximum flow in our example.

12.3.3 Maximum Matchings

A *maximum matching* of a graph is a maximum-size set of node pairs where each pair is connected with an edge and each node belongs to at most one pair. While solving the maximum matching problem in a general graph requires tricky algorithms, the problem is much easier to solve if we assume that the graph is bipartite. In this case we can reduce the problem to the maximum flow problem.

The nodes of a bipartite graph can always be divided into two groups such that all edges of the graph go from the left group to the right group. For example, Fig. 12.27 shows a maximum matching of a bipartite graph whose left group is $\{1, 2, 3, 4\}$ and right group is $\{5, 6, 7, 8\}$.

We can reduce the bipartite maximum matching problem to the maximum flow problem by adding two new nodes to the graph: a source and a sink. We also add edges from the source to each left node and from each right node to the sink. After this, the size of a maximum flow in the resulting graph equals the size of a maximum matching in the original graph. For example, Fig. 12.28 shows the reduction and the maximum flow for our example graph.

Hall's Theorem *Hall's theorem* can be used to find out whether a bipartite graph has a matching that contains all left or right nodes. If the number of left and right nodes is the same, Hall's theorem tells us if it is possible to construct a *perfect matching* that contains all nodes of the graph.

Assume that we want to find a matching that contains all left nodes. Let X be any set of left nodes and let $f(X)$ be the set of their neighbors. According to Hall's

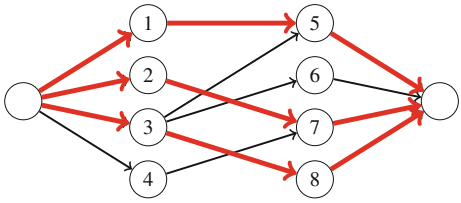


Fig. 12.28 Maximum matching as a maximum flow

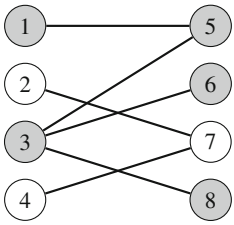


Fig. 12.29 $X = \{1, 3\}$ and $f(X) = \{5, 6, 8\}$

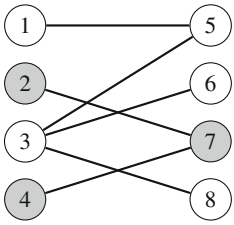


Fig. 12.30 $X = \{2, 4\}$ and $f(X) = \{7\}$

theorem, a matching that contains all left nodes exists exactly when for every possible set X, the condition $|X| \leq |f(X)|$ holds.

Let us study Hall's theorem in our example graph. First, let $X = \{1, 3\}$ which yields $f(X) = \{5, 6, 8\}$ (Fig. 12.29). The condition of Hall's theorem holds, because $|X| = 2$ and $|f(X)| = 3$. Then, let $X = \{2, 4\}$ which yields $f(X) = \{7\}$ (Fig. 12.30). In this case, $|X| = 2$ and $|f(X)| = 1$, so the condition of Hall's theorem does not hold. This means that it is not possible to form a perfect matching for the graph. This result is not surprising, because we already know that the maximum matching of the graph is 3 and not 4.

If the condition of Hall's theorem does not hold, the set X explains *why* we cannot form such a matching. Since X contains more nodes than $f(X)$, there are no pairs for all nodes in X. For example, in Fig. 12.30, both nodes 2 and 4 should be connected with node 7, which is not possible.

Kőnig's Theorem A *minimum node cover* of a graph is a minimum set of nodes such that each edge of the graph has at least one endpoint in the set. In a general graph, finding a minimum node cover is a NP-hard problem. However, if the graph is bipartite, *Kőnig's theorem* tells us that the size of a minimum node cover always equals the size of a maximum matching. Thus, we can calculate the size of a minimum node cover using a maximum flow algorithm.

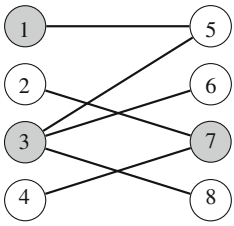


Fig. 12.31 A minimum node cover

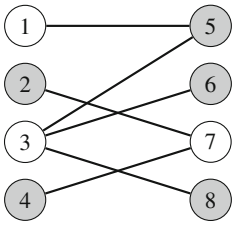


Fig. 12.32 A maximum independent set

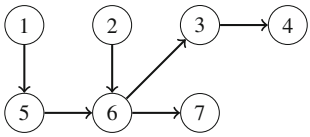


Fig. 12.33 An example graph for constructing path covers

For example, since the maximum matching of our example graph is 3, Kőnig's theorem tells us that the size of a minimum node cover is also 3. Figure 12.31 shows how such a cover can be constructed.

The nodes that do *not* belong to a minimum node cover form a *maximum independent set*. This is the largest possible set of nodes such that no two nodes in the set are connected with an edge. Again, finding a maximum independent set in a general graph is a NP-hard problem, but in a bipartite graph we can use Kőnig's theorem to solve the problem efficiently. Figure 12.32 shows a maximum independent set for our example graph.

12.3.4 Path Covers

A *path cover* is a set of paths in a graph such that each node of the graph belongs to at least one path. It turns out that in directed acyclic graphs, we can reduce the problem of finding a minimum path cover to the problem of finding a maximum flow in another graph.

Node-Disjoint Path Covers In a *node-disjoint* path cover, each node belongs to exactly one path. As an example, consider the graph in Fig. 12.33. A minimum node-disjoint path cover of this graph consists of three paths (Fig. 12.34).

We can find a minimum node-disjoint path cover by constructing a *matching graph* where each node of the original graph is represented by two nodes: a left node and a

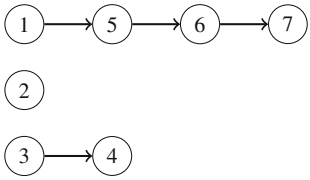


Fig. 12.34 A minimum node-disjoint path cover

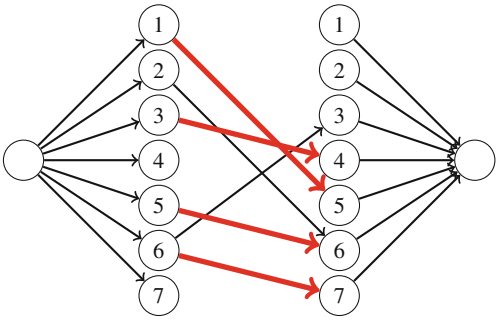


Fig. 12.35 A matching graph for finding a minimum node-disjoint path cover

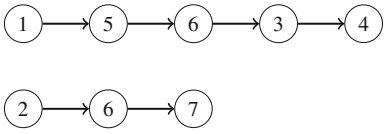


Fig. 12.36 A minimum general path cover

right node. There is an edge from a left node to a right node if there is such an edge in the original graph. In addition, the matching graph contains a source and a sink, and there are edges from the source to all left nodes and from all right nodes to the sink. Each edge in the maximum matching of the matching graph corresponds to an edge in the minimum node-disjoint path cover of the original graph. Thus, the size of the minimum node-disjoint path cover is $n - c$, where n is the number of nodes in the original graph, and c is the size of the maximum matching.

For example, Fig. 12.35 shows the matching graph for the graph in Fig. 12.33. The maximum matching is 4, so the minimum node-disjoint path cover consists of $7 - 4 = 3$ paths.

General Path Covers A *general path cover* is a path cover where a node can belong to more than one path. A minimum general path cover may be smaller than a minimum node-disjoint path cover, because a node can be used multiple times in paths. Consider again the graph in Fig. 12.33. The minimum general path cover of this graph consists of two paths (Fig. 12.36).

A minimum general path cover can be found almost like a minimum node-disjoint path cover. It suffices to add some new edges to the matching graph so that there is an edge $a \rightarrow b$ always when there is a path from a to b in the original graph (possibly through several nodes). Figure 12.37 shows the resulting matching graph for our example graph.

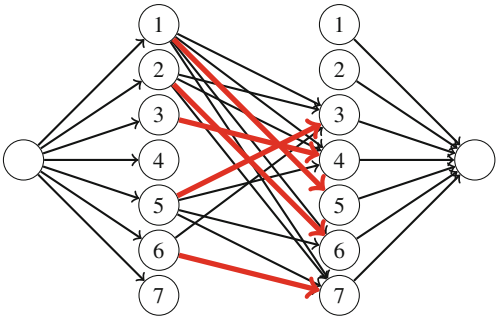


Fig. 12.37 A matching graph for finding a minimum general path cover

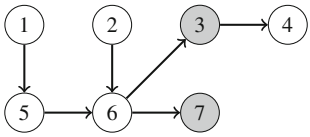


Fig. 12.38 Nodes 3 and 7 form a maximum antichain

Dilworth's Theorem An *antichain* is a set of nodes in a graph such that there is no path from any node to another node using the edges of the graph. *Dilworth's theorem* states that in a directed acyclic graph, the size of a minimum general path cover equals the size of a maximum antichain. For example, in Fig. 12.38, nodes 3 and 7 form an antichain of two nodes. This is a maximum antichain, because a minimum general path cover of this graph has two paths (Fig. 12.36).

12.4 Depth-First Search Trees

When depth-first search processes a connected graph, it also creates a rooted directed *spanning tree* that can be called a *depth-first search tree*. Then, the edges of the graph can be classified according to their roles during the search. In an undirected graph, there will be two types of edges: *tree edges* that belong to the depth-first search tree and *back edges* that point to already visited nodes. Note that a back edge always points to an ancestor of a node.

For example, Fig. 12.39 shows a graph and its depth-first search tree. The solid edges are tree edges, and the dashed edges are back edges.

In this section, we will discuss some applications for depth-first search trees in graph processing.

12.4.1 Biconnectivity

A connected graph is called *biconnected* if it remains connected after removing any single node (and its edges) from the graph. For example, in Fig. 12.40, the left

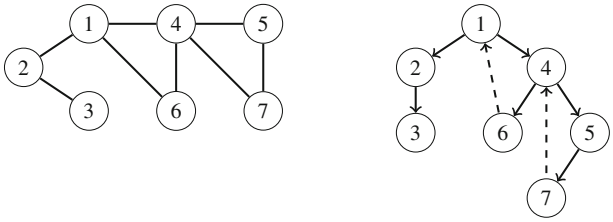


Fig. 12.39 A graph and its depth-first search tree

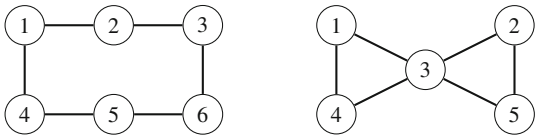


Fig. 12.40 The left graph is biconnected, the right graph is not

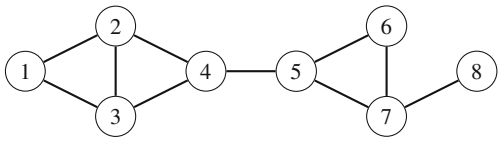


Fig. 12.41 A graph with three articulation points and two bridges

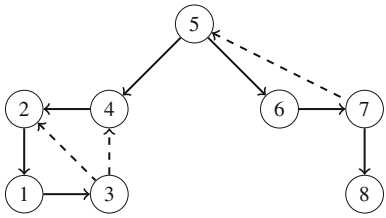


Fig. 12.42 Finding bridges and articulation points using depth-first search

graph is biconnected, but the right graph is not. The right graph is not biconnected, because removing node 3 from the graph disconnects the graph by dividing it into two components $\{1, 4\}$ and $\{2, 5\}$.

A node is called an *articulation point* if removing the node from the graph disconnects the graph. Thus, a biconnected graph does not have articulation points. In a similar way, an edge is called a *bridge* if removing the edge from the graph disconnects the graph. For example, in Fig. 12.41, nodes 4, 5, and 7 are articulation points, and edges 4–5 and 7–8 are bridges.

We can use depth-first search to efficiently find all articulation points and bridges in a graph. First, to find bridges, we begin a depth-first search at an arbitrary node, which builds a depth-first search tree. For example, Fig. 12.42 shows a depth-first search tree for our example graph.

An edge $a \rightarrow b$ corresponds to a bridge exactly when it is a tree edge, and there is no back edge from the subtree of b to a or any ancestor of a. For example, in Fig. 12.42, edge $5 \rightarrow 4$ is a bridge, because there is no back edge from nodes

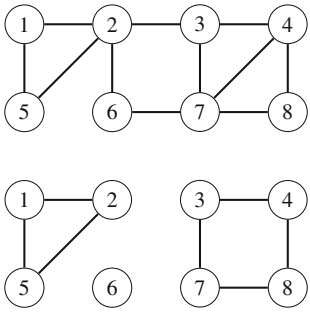


Fig. 12.43 A graph and an Eulerian subgraph

$\{1, 2, 3, 4\}$ to node 5. However, edge $6 \to 7$ is not a bridge, because there is a back edge $7 \to 5$, and node 5 is an ancestor of node 6.

Finding articulation points is a bit more difficult, but we can again use the depth-first search tree. First, if a node x is the root of the tree, it is an articulation point exactly when it has two or more children. Then, if x is not the root, it is an articulation point exactly when it has a child whose subtree does not contain a back edge to an ancestor of x.

For example, in Fig. 12.42, node 5 is an articulation point, because it is the root and has two children, and node 7 is an articulation point, because the subtree of its child 8 does not contain a back edge to an ancestor of 7. However, node 2 is not an articulation point, because there is a back edge $3 \to 4$, and node 8 is not an articulation point, because it does not have any children.

12.4.2 Eulerian Subgraphs

An *Eulerian subgraph* of a graph contains the nodes of the graph and a subset of the edges such that the degree of each node is *even*. For example, Fig. 12.43 shows a graph and its Eulerian subgraph.

Consider the problem of calculating the total number of Eulerian subgraphs for a connected graph. It turns out that there is a simple formula for this: there are always 2^k Eulerian subgraphs were k is the number of back edges in the depth-first search tree of the graph. Note that $k = m - (n - 1)$ where n is the number of nodes and m is the number of edges.

The depth-first search tree helps to understand why this formula holds. Consider any fixed subset of back edges in the depth-first search tree. To create an Eulerian subgraph that contains these edges, we need to choose a subset of the tree edges so that each node has an even degree. To do this, we process the tree from bottom to top and always include a tree edge in the subgraph exactly when it points to a node whose degree is even with the edge. Then, since the sum of degrees is even, also the degree of the root node will be even.