Chapter 2. Matchings and flows

Section 2.1. Matching problems

The dance-class problem.

A group of boys and girls, with just as many boys as girls, want to dance together; hence, they have to be matched in couples. Each boy prefers to dance with some of the girls, and each girl prefers to dance with some of the boys. A boy and a girl will dance together if and only if both like each other. The question is whether the boys and girls can be combined to dancing pairs such that nobody has to stand aside.

In the corresponding graph problem two sets of vertices V_1 and V_2 are defined that denote the boys and the girls, respectively. An edge between a vertex $v_1 \in V_1$ and $v_2 \in V_2$ is defined if boy v_1 and girl v_2 like to dance with each other. We are asked to find a subset M of the edges such that each vertex is incident to exactly one edge.

A subset M of the edges of a graph $G = (V, E)$ is called a *matching* if each vertex is incident to at most one edge of M. The matching M is called *perfect* or *complete* if each vertex is incident to exactly one edge of M.

The graph by which the dance-class problem is modeled is a so-called *bipartite graph*. A graph $G = (V, E)$ is bipartite if the vertices can be partitioned into two sets V_1 and V_2 ($V_1 \cup V_2 = V, V_1 \cap V_2 = \emptyset$) such that each edge consists of one vertex from V_1 and one vertex from V_2. In general, the number of vertices in V_1 and V_2 is not equal.

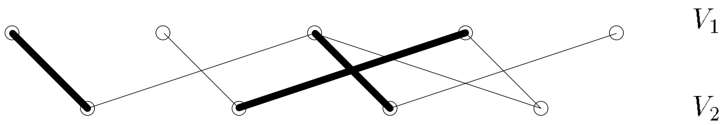


Figure 2.1: A matching in a bipartite graph.

In the following, edges in a matching will be drawn as fat edges. The next theorem provides a characterization of a bipartite graph.

Theorem 2.1
A graph $G = (V, E)$ is bipartite if and only if it does not contain a circuit of odd length.

The problem of finding a maximum cardinality matching or a perfect matching arises frequently in different mathematical and practical situations. Consider the following examples.

- Systems of Distinct Representatives (SDR's). Let S be a set of elements and let $S_1, \dots, S_K$ be subsets of S. The question is whether there exist K different elements $s_1, \dots, s_K$ in S such that $s_k \in S_k$ $(k = 1, \dots, K)$.

Example $S = \{s_1, s_2, s_3, s_4, s_5\}$.

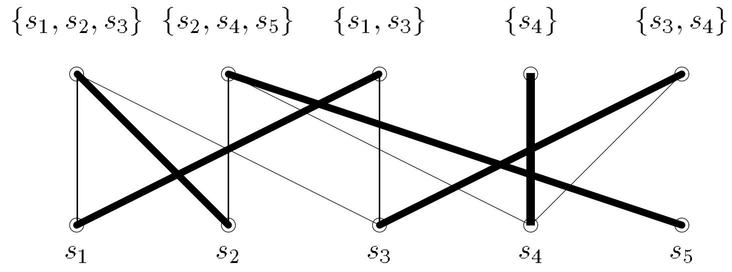


Figure 2.2: A systems of distinct representatives.

- Let $A = (a_{ij})$ be an $n \times n$ matrix with entries that are either zero or one. Let the rows be denoted by $r_1, \ldots, r_n$ and the columns by $c_1, \ldots, c_n$. The question is whether it is possible to combine the rows and columns to pairs such that each row and each column appears in exactly one pair, where each of the chosen pairs (r_i, c_j) must correspond to an entry $a_{ij} = 1$.

  A more practically oriented application is the following:
- A set of jobs in a company must be fulfilled by applicants. Each candidate has the skills to perform a certain subset of the jobs. Assign suitable candidates to as many jobs as possible.

Algorithms for finding maximum matchings try to extend a given matching M to a matching M' with a cardinality that is one higher, i.e., $|M'| = |M| + 1$. These improvement algorithms are based on the concept of alternating and augmenting paths. An *alternating* path in a graph $G = (V, E)$ with respect to a matching M is a path $(v_1, v_2, \ldots, v_K)$ where either $\{v_1, v_2\}, \{v_3, v_4\}, \ldots, \{v_{2k-1}, v_{2k}\}, \ldots$ are in M or $\{v_2, v_3\}, \{v_4, v_5\}, \ldots, \{v_{2k}, v_{2k+1}\}, \ldots$ are in M. Hence, edges from M are alternated with edges from $E \backslash M$. An *augmenting* path is an alternating path in which both endvertices are free, i.e., not incident to an edge in M. Such paths always have odd length, i.e., an odd number of edges and an even number of vertices. A *matched* or *exposed* vertex is a vertex incident to an edge in M.

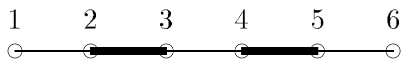


Figure 2.3: An augmenting path P with respect to M.

If there is an augmenting path P in G with respect to a matching M, then the number of edges in the matching can be increased by deleting the edges in $P \cap M$ from M and adding the edges from $P \backslash M$ to M.

Figure 2.4: The alternating path P with respect to $M' = (M \backslash (P \cap M)) \cup P \backslash M$.

Hence, we have that if there exists an augmenting path, then the matching can be increased. The converse is proved in the following theorem.

Theorem 2.2.
Let M be a matching in $G = (V, E)$. M can be extended if and only if there is an augmenting path in G with respect to M.

Proof.
We have shown above that if there exists an augmenting path P, then $M' = (M \cup P \backslash M) \backslash (P \cap M)$ is a matching with cardinality one higher than M.

Now suppose that there exists a matching M' with cardinality one higher than the cardinality of M; we have to indicate an augmenting path with respect to M. Consider the edges in the symmetric difference of M and M', which is defined as $M' \div M = (M' \backslash M) \cup (M \backslash M')$; this set contains the edges that are in M or M', but not in both. Let G' denote the graph $G' = (V, M' \div M)$. Since each vertex is incident to at most one edge in M and M', we have that all vertices in G' have degree 0,1, or 2. Since a vertex with degree equal to 2 is incident to one edge from M and one edge from M', we have that all connected components in G' are either alternating paths or alternating circuits. As the number of edges in M' is one higher than the number of edges in M, we have that there must be a path P containing more edges from M' than from M. This path P is alternating and both endpoints of P are not incident to an edge of M, which implies that P is augmenting with respect to M. □

The algorithm for finding a maximum matching can now be stated as follows.

Algorithm MM

Input: A graph $G = (V, E)$.

Output: A matching $M \subseteq E$.

STEP 1. $M = \emptyset$.

STEP 2. Find an augmenting path P. If no augmenting path is found, then stop.

STEP 3. Increase M to $(M \backslash M \cap P) \cup (P \backslash M)$; go to STEP 2.

A systematic search for augmenting paths is performed by a labeling algorithm. In general graphs it is fairly complicated; therefore, we restrict ourselves to bipartite graphs. The labeling algorithm proceeds as follows: We check for each vertex if it can be connected to an alternating path starting in a free vertex; if so, this vertex receives label Y (yes), otherwise, it receives label N (no). We proceed our labeling until either an alternating path P is discovered or until no more vertices receive label Y that were previously labeled N. The algorithm makes use of two additional sets S and T, which contain the vertices in V and W whose label changed from N to Y.

Algorithm APM

Input: A bipartite graph $G = (V \cup W, E)$ and a matching $M \subseteq E$.

Output: An augmenting path in G with respect to M, if one exists.

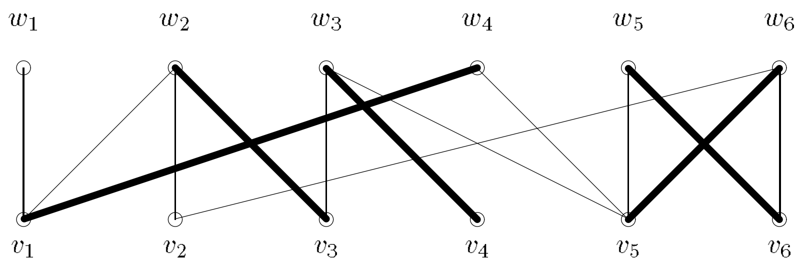
Step 1. The vertices in V and W are labeled: if a vertex $v \in V$ is unmatched, then it gets label Y; otherwise, it gets label N. All vertices in W get label N. The sets S and T contain all vertices $v \in V$ and $w \in W$ with label Y, respectively; that is, S contains all the vertices in V that are unmatched, whereas $T \leftarrow \emptyset$.

Step 2. Consider the vertices $v \in S$. All vertices $w \in W$ with label N that are connected to a vertex $v \in S$ through an edge that is not in M obtain label Y and are added to T. Set $S \leftarrow \emptyset$.

Step 3. Consider the vertices $w \in T$. If any of these vertices w is unmatched, then an augmenting path has been found. Otherwise, all vertices $v \in V$ with label N that are connected to a vertex $w \in T$ through an edge in M obtain label Y and are added to S. Set $T \leftarrow \emptyset$.

Step 4. If an augmenting path has been found or if $S = \emptyset$, stop. Otherwise, go to Step 2.

Example.

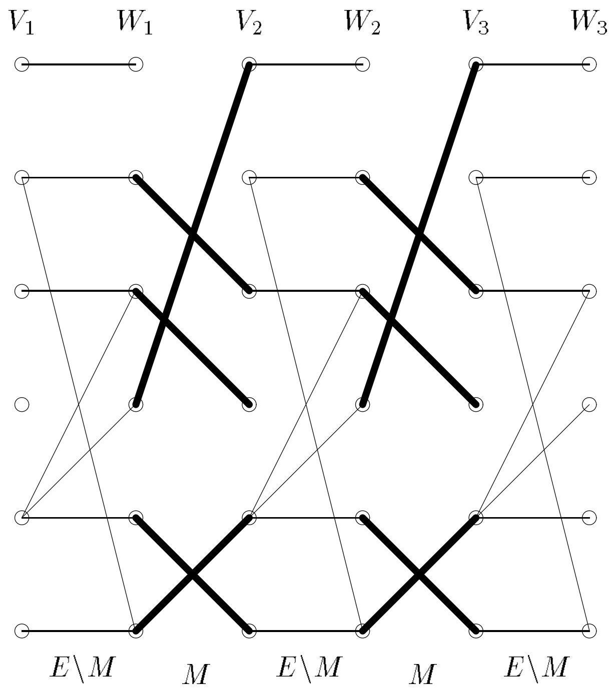


$$\begin{aligned} E\backslash M &= \{\{v_1, w_1\}, \{v_1, w_2\}, \{v_2, w_2\}, \{v_2, w_6\}, \{v_3, w_3\}, \{v_5, w_3\}, \{v_5, w_4\}, \{v_5, w_5\}, \{v_6, w_6\}\}. \\ M &= \{\{v_1, w_4\}, \{v_3, w_2\}, \{v_4, w_3\}, \{v_5, w_6\}, \{v_6, w_5\}\}. \end{aligned}$$

To illustrate the working of the labeling algorithm, the vertex set will be copied K times to $V_1, \ldots, V_K, W_1, \ldots, W_K$. The edges between copies V_k and W_k ($k = 1, \ldots, K$) are those corresponding to $E\backslash M$. The edges between copies W_k, V_{k+1} ($k = 1, \ldots, K-1$) are those corresponding to M.

In the initialization in Step 1, vertex v_2 gets label Y, whereas all other vertices get the label N: $S \leftarrow \{v_2\}$ and $T \leftarrow \emptyset$. In the first run through Step 2, the vertices w_2 and w_6 receive label Y, $T \leftarrow \{w_2, w_6\}$, and $S \leftarrow \emptyset$. In the first run through Step 3, the vertices v_3 and v_5 receive label Y, $S \leftarrow \{v_3, v_5\}$, and $T \leftarrow \emptyset$. In the second run through Step 2, the vertices w_3, w_4, and w_5 are labeled Y, $T \leftarrow \{w_3, w_4, w_5\}$, and $S \leftarrow \emptyset$. In the second run through Step 3, the vertices v_1, v_4, and v_6 are labeled Y: $S \leftarrow \{v_1, v_4, v_6\}$ and $T \leftarrow \emptyset$. In the third run through Step 2, the vertices w_1, w_2, and w_6 are labeled Y; only for w_1, the label changes. Therefore, $T \leftarrow \{w_1\}$ and $S \leftarrow \emptyset$. In the third run through Step 3, the existence of the augmenting path $\{v_2, w_6\}, \{w_6, v_5\}, \{v_5, w_4\}, \{w_4, v_1\}, \{v_1, w_1\}$ is signaled.

In the above description of algorithm APM, no attention has been paid to storing the augmenting path. This can be taken care of easily, however, by keeping a list for each vertex in the current sets S and T that denotes how this vertex is reached.



The labeling algorithm proceeds by labeling vertices in V_1, then in W_1 and V_1 etcetera. The newly labeled vertices and the corresponding edges are drawn in the following figure.

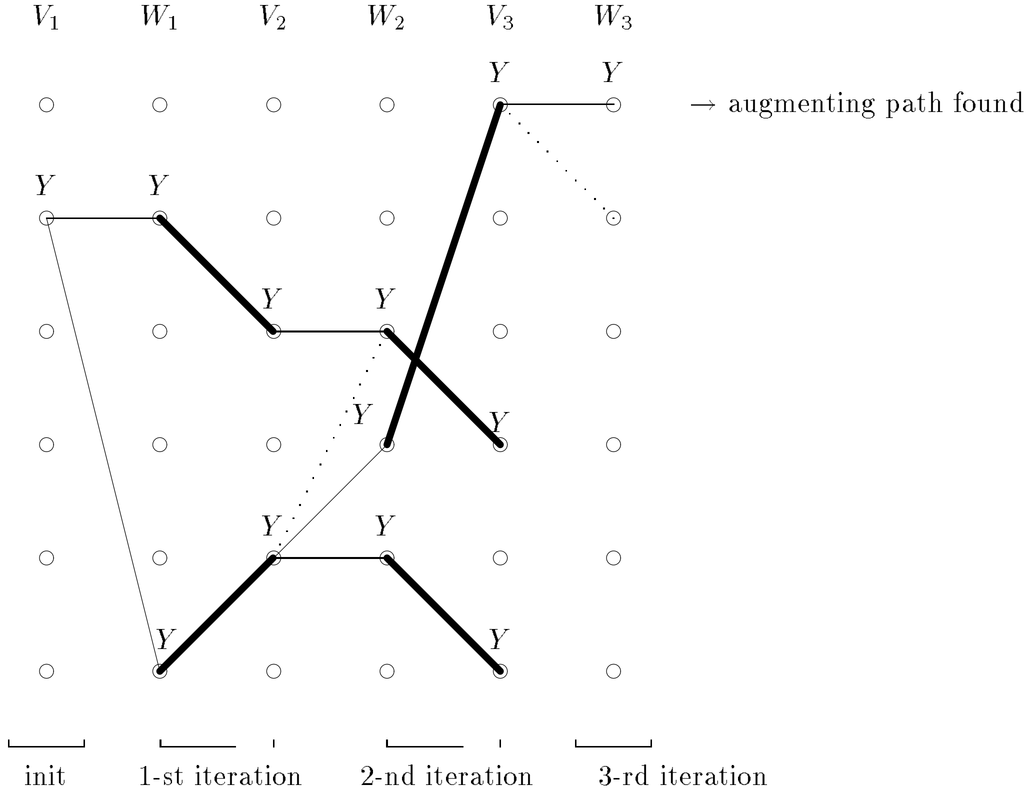
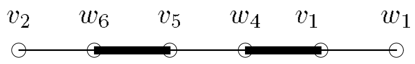


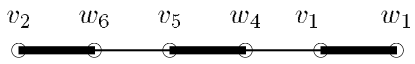
Figure 2.5.

The dotted lines concern edges that connect a vertex that has just been labeled Y to an vertex that already had been labeled Y. Clearly, the tree in Figure 2.5 consists of alternating paths only.

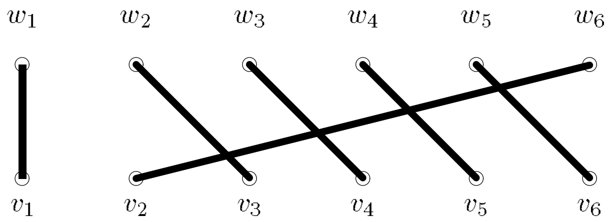
The augmenting path is the following one:



Hence, we can augment the corresponding part of the matching to:



This results in the final (perfect) matching:



Theorem 2.4
Algorithm APM finds an augmenting path, if one exists.

Proof.
Suppose that an augmenting path exists. Let $P = \{v_1, w_1, v_2, w_2, \ldots, w_K)$ be the shortest one. Initially, we have that $v_1 \in S$, and hence, it gets label Y. Now suppose that at the beginning of an iteration $v_1, w_1, \ldots, v_k$ have label Y and $w_k, v_{k+1}, \ldots, w_K$ have label N. Moreover, $v_k \in S$. In the first part of the iteration, w_k gets label Y and it enters T. Note that since P is alternating, we have that $\{v_k, w_k\} \in E \backslash M$. Furthermore, there is no vertex $w_l \in \{w_{k+1}, \ldots, w_K\}$ that has received label Y in the meantime, since this would imply that there exists a shorter augmenting path $(v_1, w_1, v_2, \ldots, v_k, w_l, \ldots, w_K)$. Since P is alternating, we have that $\{w_l, v_{l+1}\} \in M$, and hence, v_{k+1} is labeled Y in the second part of the iteration. Moreover, it enters S. Finally, the unmatched vertex w_K is labeled Y, and the path is detected. $\square$

Algorithm MM consists of repeatedly applying algorithm APM. If the vertex sets V and W both contain n vertices, then APM can be applied at most n times to augment a given matching. In each call of algorithm APM, an edge $e = \{v, w\}$ is examined at most once: namely when w is labeled Y if $e \in M$, and when v is labeled Y if $e \notin M$. Hence, we have that the running time $r(n)$ of algorithm MM on a graph $G = (V \cup W, E)$ with $|V| = |W| = n$ amounts to $O(|V| \cdot |E|)$.

As a corollary of the augmenting path algorithm, we get the following existence theorem of a perfect matching in a bipartite graph.

Theorem 2.5 (Hall)
The graph $G = (V \cup W, E)$ with $|V| = |W|$ contains a perfect matching if and only if for each subset $V' \subseteq V$ we have that the subset $W' \subseteq W$ that contains all vertices adjacent to some vertex in V' is at least as large as V', i.e., $|W'| \geq |V'|$.

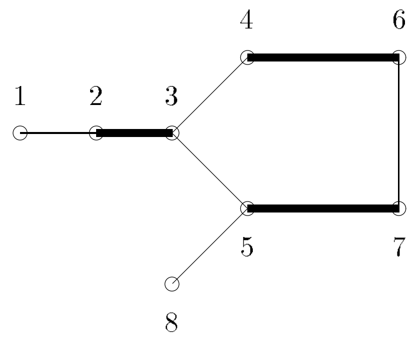
Proof.
If a perfect matching M exists, then we obviously have that for any subset $V' \subseteq V$ the set W' contains the vertices matched to vertices of V' in M, which implies that $|W'| \geq |V'|$.

Now suppose that there is no perfect matching; we will indicate a subset V' for which the condition on W' fails. Let M be a maximum cardinality matching. Run algorithm APM on M, and let V' contain the vertices in V that are labeled Y and W' the vertices in W labeled Y; since the algorithm has stopped, we have that the set W' contains exactly those vertices that are adjacent to at least one vertex in V'. Since M has maximum cardinality, no augmenting path has been found, which implies that none of the vertices $w \in W'$ is free. Since each vertex $w \in W'$ can be matched to a vertex $v \in V'$ only, we have that V' contains $|W'|$ vertices that are connected to a vertex in W' in M and at least one free vertex, as the set S initially is nonempty. Hence, $|V'| > |W'|$. $\square$

Hall presented Theorem 2.5 in the context of SDR's: There exists a system of SDR's if and only if each union of k subsets contains at least k elements.

Theorem 2.5 provides a good characterization of the existence of perfect matchings. If a graph contains a perfect matching, then you can convince somebody by providing a perfect matching. If a graph does not contain a perfect matching, then you can convince somebody by providing a set V' and the corresponding set W' that contains all vertices that are adjacent to at least one of the vertices of V' with $|V'| > |W'|$.

The labeling algorithm APM can not be applied to find matchings in general graphs. It follows immediately from Theorem 2.1 that this must be due to the presence of odd circuits. An example in which algorithm APM fails to find an augmenting path, though there is one, is provided by the following graph.



Since there is no clarity as to which vertices belong to V and which to W, we suppose that $1 \in V$ and $8 \in W$. APM labels the vertices as follows. In the initialization in Step 1, vertex 1 gets label Y, whereas all other vertices get label N: $S \leftarrow \{1\}$ and $T \leftarrow \emptyset$. In the first run through Step 2, vertex 2 receives label Y, $T \leftarrow \{2\}$, and $S \leftarrow \emptyset$. In the first run through Step

3, vertex 3 receives label Y, $S \leftarrow \{3\}$, and $T \leftarrow \emptyset$. In the second run through Step 2, the vertices 4 and 5 are labeled Y, $T \leftarrow \{4,5\}$, and $S \leftarrow \emptyset$. In the second run through Step 3, the vertices 6 and 7 are labeled Y, $S \leftarrow \{6,7\}$, and $T \leftarrow \emptyset$. In the third run through Steps 2 and 3, no vertices are relabeled, which implies that $S \leftarrow \emptyset$, and the algorithm stops. Hence, the augmenting path (1,2,3,4,6,7,5,8) remains undiscovered.

Note that this problem would not have occurred if in the third run through Steps 2 and 3 we would have allowed the vertices 6 and 7 to be included in T; the only reason that they were not included was that they had already been labeled Y, but at the time of their relabeling, they were included in S then. This problem can be solved by specifying the labels of the node according to the distance of the unmatched nodes. A node v gets label O (odd), if an alternating path of odd length from a unmatched node to v is discovered. A node gets label E (even), if an alternating of even length from an unmatched node to v is discovered. Hence, nodes can get two labels: O and E. We leave it as an exercise to the reader to apply this revision of APM to the example.

The maximum cardinality matching problem can be generalized to a weighted version by attaching weights to the edges. Given a graph $G = (V, E)$ and a function $c : E \rightarrow \mathbb{Z}^+$, the problem then becomes: find a matching M of maximum weight; the weight $c(M)$ of the matching M is equal to $\sum_{e \in M} c(e)$.

If $G = (V \cup W, E)$ is bipartite, with $|V| = |W|$, then this problem is known as the assignment problem. Kuhn devised an algorithm to solve the assignment problem in 1955; it is known as the Hungarian method. Edmonds developed an algorithm for the problem on general graphs in 1965, the so-called blossom algorithm; this has been implemented to run in $O(|V|^3)$ time.

Note that the matching of maximum weight does not need to have maximum cardinality. If you want to determine the maximum weight matching from among the matchings with maximum cardinality, then you can still use the same machinery as for finding the matching with maximum weight, however: just add a huge constant to the weight of each edge.

A problem that uses the blossom algorithm as a subprogram is the Chinese postman problem, which is discussed in the next section.

Section 2.2. The Chinese postman problem

Consider the problem that the postman faces each rainy day: he has to deliver mail, and prefers to be home as soon as possible, that is, he wants to minimize the distance that he has to walk. This problem is modeled as a minimization problem in graphs as follows. Consider the streets in which mail has to be delivered as edges, and the intersections of these streets as vertices. The cost of each edge is equal to the length of the corresponding street. The postman wants to find a cycle of minimum length that traverses each edge at least once. Formally, this problem is stated as:

Chinese postman problem

Given a graph $G = (V, E)$ and a cost function $c : E \rightarrow \mathbb{Z}^+$, we are asked to find a minimum-weight set E' of edges to add to G so that the resulting graph $G' = (V, E \cup E')$ contains an *Eulerian cycle*, i.e., a cycle containing each edge of G' exactly once.

As already mentioned in Theorem 1.3, Euler has shown that a necessary and sufficient condition for a multigraph to possess an Eulerian cycle is that the multigraph is connected and that each vertex has *even* degree. Assuming the graph G to be connected, we only have to take care of the odd-degree vertices.

By Proposition 1.1, the set V' of odd-degree vertices has even cardinality. Since the degree of all vertices in G' has to be even, we know that all vertices in V' have odd degree in the graph (V, E'), whereas all other vertices must have even degree in (V, E'). Moreover, (V, E') should not contain any circuits, since circuits have non-negative cost and can be deleted without changing the status (even/odd) of any vertex. Therefore, we must have that (V, E') consists of a set of paths connecting the odd-degree vertices. Of course, the paths between any pair of odd-degree vertices have minimum length among all paths connecting this pair of vertices.

The following five-phase algorithm solves the Chinese postman problem on a graph $G = (V, E)$ with cost function $c : E \to \mathbb{Z}^+$.

Phase 1: Find the odd-degree vertices V'.
Phase 2: Find the shortest paths between all pairs of vertices in V'.
Phase 3: Construct a complete graph H on V', where the weight of an edge $\{v, w\}$ is equal to the length of the shortest path between the vertices v and w.
Phase 4: Find a perfect matching in H with minimum weight.
Phase 5: For each edge in the matching of H, add the edges of the corresponding path to G.

Example

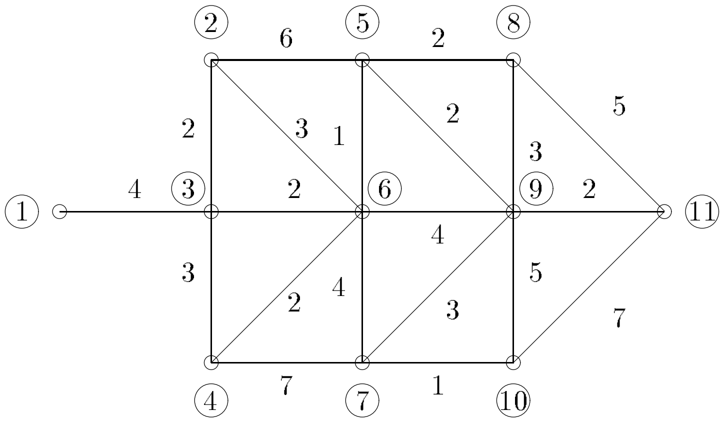
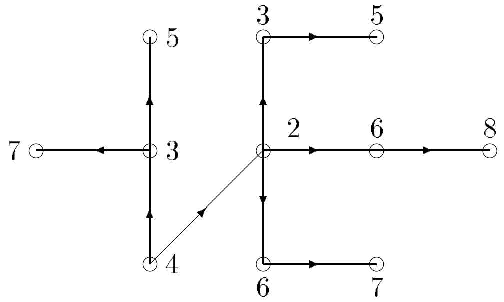


Figure 2.6.

Phase 1: The odd degree vertices are 1,2,4,8,10 and 11.
Phase 2: The shortest paths between all pairs of vertices in $\{1, 2, 4, 8, 10, 11\}$ are derived by applying algorithm SP six times, with root vertex s equal to 1,2,4,8,10 and 11, respectively.
For $s = 4$ we get the following shortest paths; the lengths are denoted by the given labels.

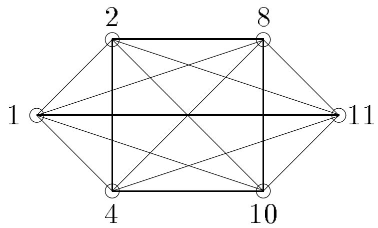


The complete set of shortest path lengths between the odd-degree vertices is given in the following table.

	1	2	4	8	10	11
1						
2	6					
4	7	5				
8	9	6	5			
10	11	8	7	7		
11	11	9	8	5	6	

Figure 2.7.

Phase 3: Find a minimum weight perfect matching in the following graph, the complete graph on the vertices $\{1,2,4,8,10,11\}$. The weights of the edges are given in the table in Figure 2.7.



Phase 4:

Matching	Weight	
$\{1,2\},\{4,8\},\{10,11\}$	$6+5+6=17$	$\rightarrow$ minimum
$\{1,2\},\{4,10\},\{8,11\}$	$6+7+5=18$	
$\{1,2\},\{4,11\},\{8,10\}$	$6+8+7=21$	

In any other matching, the weight of the edge containing vertex 1 is at least 7, whereas the other two edges have weight at least 5. Hence, the weight of this matching will amount to at least 17. Therefore, the matching $\{\{1,2\},\{4,8\},\{10,11\}\}$ is optimal. The corresponding shortest paths are:

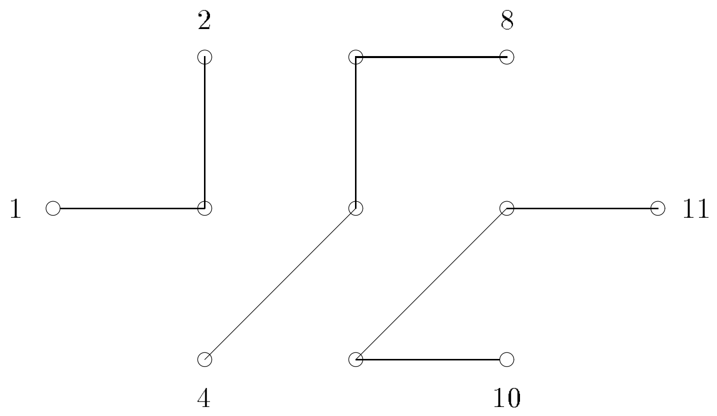
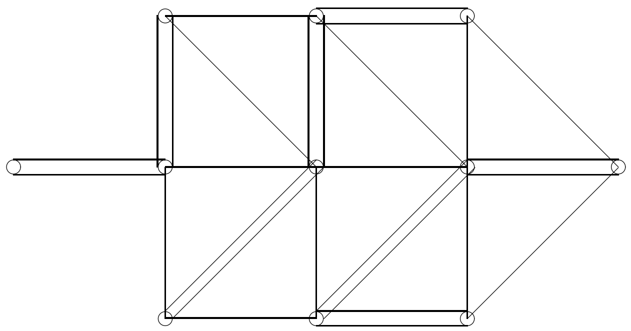


Figure 2.8.

Phase 5: The Eulerian graph contains the edges of the shortest paths of Figure 2.8 twice. Its weight is increased by 17.



The running time of our five-phase algorithm is determined as the maximum of the running times in the phases. Phase 1 takes $O(|V|)$ time, since all vertices are checked in constant time per vertex. Phase 2 takes $O(|V'|^3)$ time, since determining the shortest path from one vertex to all other vertices takes $O(|V'|^2)$ time. Phase 3 takes $O(|V'|^2)$ time, since the number of edges amounts to $|V'|(|V'|-1)/2$. From the previous subsection, it follows that Phase 4 takes $O(|V'|^3)$ time as well, while Phase 5 takes $O(|V'|^2)$ time at most. We do not know, however, what $|V'|$ is going to be; all we know is that $|V'| \leq |V|$. Hence, the running time amounts to $O(|V|^3)$.

Section 2.3. Maximum flows

Some catastrophe has happened in some far away place, and help is needed badly. Fortunately, all that is needed is available, but it is stored in some depot that is quite remote from the disaster area. We are asked to find a way to send as much help to the area as possible. Unfortunately, the time that an army of planes transported all goods to this place lies behind us, but we are allowed to make use of spare capacity in regular flights, trade services, etc. Obviously, in each transshipment point we can dump a lot of goods, but nobody is helped if it is not transported further to the disaster area. Hence, we must add the restriction that everything that is brought into a place that is not the disaster area should be transported out of this place. This problem can be modeled as a minimization problem in graphs. We need to introduce some notation first.

The pair (G, c) in which G is a digraph (V, A) and $c : A \to \mathbb{Z}^+$ is a capacity function on the arcs, is called a *network*. In the network two special vertices, a *source* s and a *sink* t are identified. A *flow* from s to t in (G, c) is a function $x : A \to \mathbb{Z}^+$ on the arcs that satisfies the following two properties.

Flow conservations constraints: For each node $v \in V \backslash \{s, t\}$ the amount of flow leaving v equals the amount of flow entering v:

$$\forall_{v \in V \backslash \{s,t\}} : \sum_{w:(v,w) \in A} x_{(v,w)} = \sum_{w:(w,v) \in A} x_{(w,v)} \ .$$

Capacity constraints: $\forall_{a \in A} : 0 \leq x_a \leq c_a$.

The *flow value* f is the net outflow from s, i.e., the amount of flow leaving s minus the amount of flow entering s. We only consider networks in which no arcs enter s. Hence, the flow value f then becomes:

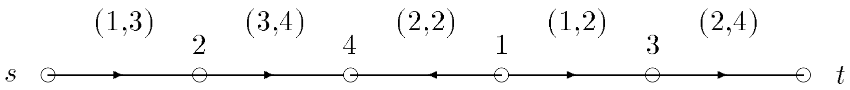
$$f = \sum_{w:(s,w) \in A} x_{(s,w)} - \sum_{w:(w,s) \in A} x_{(w,s)} = \sum_{w:(s,w) \in A} x_{(s,w)} \ .$$

Example

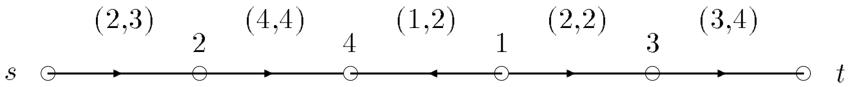
A network (G, c). A feasible flow in the network (G, c)

Figure 2.9.

A method for improving a given flow proceeds by finding paths from s to t on which the flow can be increased. Such paths are called *(flow) augmenting* paths. They differ from ordinary paths in a digraph in the sense that arcs do not need to have the right direction. Take for example the path $(s, 2, 4, 1, 3, t)$ in the graph of Figure 2.9.



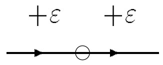
The pair (x_a, c_a) denotes the flow and the capacity of each arc a, respectively. We can increase the flow from s to t by increasing the flow over the *forward* edges (the edges in the right direction from s to t) and decreasing the flow over the *backward* edges (the edges in the opposite direction). Thus, the flow over the arcs $(s, 2), (2, 4), (1, 3)$ and $(3, t)$ is increased by one unit, and the flow over the arc $(1, 4)$ is decreased by one unit.



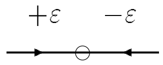
The flow on all other arcs remains equal. The outflow of s is increased by one unit, whereas the net outflow (outflow minus inflow) of the intermediate nodes is not changed and remains zero. Note that the inflow in t increases by one unit as well.

In general, an augmenting path from s to t contains forward edges, which carry flow below capacity, and backward edges, which carry a positive flow. The flow on forward arcs is increased and the flow on backward arcs is decreased with an amount ε.

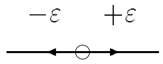
The four situations for an intermediate vertex on an augmenting path are:



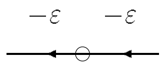
inflow and outflow increase by ε units.



inflow and outflow remain equal.



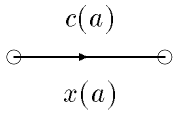
inflow and outflow remain equal.



inflow and outflow decrease by ε units.

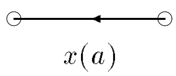
In any of the four cases, the flow conservation constraints remain valid. The value ε is determined by the capacity constraints. For each edge on the path an upper bound on ε can be derived as follows:

On a forward arc the flow can be increased at most $c(a) - x(a)$ units



$\varepsilon \le c(a) - x(a)$

On a backward are the flow can be decreased at most $x(a)$ units



$\varepsilon \le x(a)$

The value for ε is determined as the smallest upper bound from among the arcs on the augmenting path.

Augmenting paths are found systematically by the augmenting path algorithm APF; this is a labeling algorithm in which a vertex v is labeled Y whenever there exists an augmenting path from s to v. Algorithm APF makes use of two additional sets S and T. The set S contains all vertices with a label that has been changed from N to Y in the *current* iteration; that is, S contains all vertices v for which an augmenting path from s to v has been discovered in the last run. The set T is introduced only for clarity of exposition; it contains all vertices with a label that has been changed from N to Y in the *previous* iteration.

Algorithm APF

Input: A feasible flow x with value f on a network (G, c).

Output: A path from s to t on which the flow can be augmented.

STEP 1. The vertices are labeled: s gets label Y, whereas all other vertices get label N. Set $S \leftarrow \{s\}$.

STEP 2. Set $T \leftarrow S$ and set $S \leftarrow \emptyset$. Consider the vertices $v \in T$. All vertices $w \in V$ with label N that are connected to a vertex $v \in T$ by an arc (v, w) with $x_{(v,w)} < c_{(v,w)}$ or by an arc (w, v) with $x_{(w,v)} > 0$ get label Y and are added to S.

STEP 3. If t has received label Y, then an augmenting path P has been found; stop. If $S = \emptyset$, then there exists no augmenting path; stop. If none of these two cases applies, then go to STEP 2.

In the above description of algorithm APF, no attention has been paid to storing the augmenting path. This can be taken care of easily, however, by keeping a list for each vertex in the current set S that denotes how this vertex is reached.

Suppose that an augmenting path P has been discovered. Let ε_f denote the minimum value over all forward arcs $a \in P$ of $c(a) - x(a)$, and let ε_b denote the minimum value over all backward arcs $a \in P$ of $x(a)$. We augment the flow by $\varepsilon = \min\{\varepsilon_f, \varepsilon_b\}$ units.

The overall algorithm to determine a maximum flow proceeds by using the augmenting path algorithm until no augmenting path is found anymore.

Algorithm MF

Input: A network (G, c) and a source s and sink t.

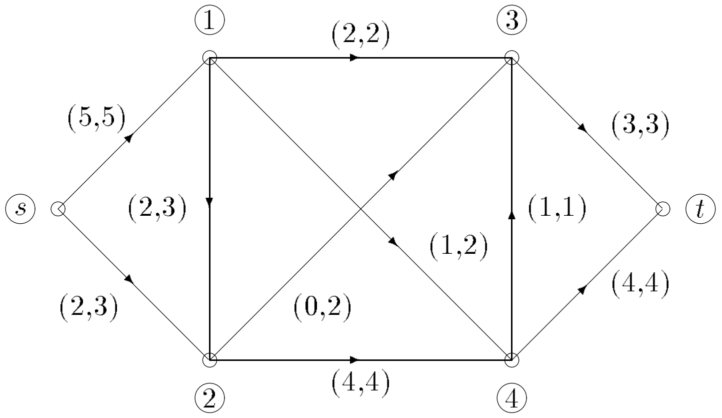
Output: A maximum flow from s to t in (G, c).

STEP 1. Start with any feasible flow, possibly the zero flow.

STEP 2. Find an augmenting path P with algorithm APF. If no augmenting path is found, then stop.

STEP 3. Augment the flow over P by as much as possible; go to STEP 2.

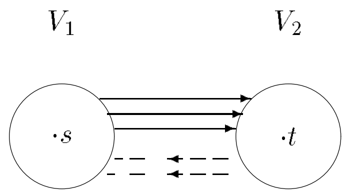
In the example of Figure 2.9, the flow that results after increasing the flow over the path $(s, 2, 4, 1, 3, t)$ by one unit is given in the following network.



In the network above, no augmenting path can be found. Algorithm APF hands labels Y to the vertices s, 2, 1, and 4 and labels N to the vertices 3 and t.

Bounds for the maximum flow value.
Clearly, a given feasible flow provides a lower bound on the maximum flow. To derive an upper bound on the flow, the so-called (s,t)-*cuts* have been introduced. An (s,t)-cut is a partition of the vertex set into two sets V_1 and V_2, where $s \in V_1$ and $t \in V_2$. The *capacity* $c(V_1, V_2)$ of the (s,t)-cut (V_1, V_2) is the total capacity of all arcs leaving V_1 and entering V_2.

$$c(V_1, V_2) = \sum_{\substack{(v,w):\ v \in V_1 \\ w \in V_2}} c_{(v,w)}$$



The capacity of the cut (V_1, V_2) is an upper bound on the value f of any flow x, as is shown below.

$$\begin{aligned} f &= \sum_{(s,v) \in A} x_{(s,v)} \\ &\overset{(0)}{=} \sum_{(s,v) \in A} x_{(s,v)} + \sum_{v \in V_1 \setminus \{s\}} \left\{ \sum_{w:(v,w) \in A} x_{(v,w)} - \sum_{w:(w,v) \in A} x_{(w,v)} \right\} \end{aligned}$$

$$\overset{(1)}{=} \sum_{v\in V_1} \sum_{w:(v,w)\in A} x_{(v,w)} - \sum_{v\in V_1} \sum_{w:(w,v)\in A} x_{(w,v)}$$

$$\overset{(2)}{=} \sum_{\substack{(v,w)\in A: \\ v\in V_1 \\ w\in V_2}} x_{(v,w)} - \sum_{\substack{(v,w)\in A: \\ v\in V_2 \\ w\in V_1}} x_{(v,w)}$$

$$\overset{(3)}{\leq} \sum_{\substack{(v,w)\in A: \\ v\in V_1 \\ w\in V_2}} x_{(v,w)}$$

$$\overset{(4)}{\leq} \sum_{\substack{(v,w)\in A: \\ v\in V_1 \\ w\in V_2}} c_{(v,w)} = c(V_1, V_2).$$

(0): The flow conservation constraints.

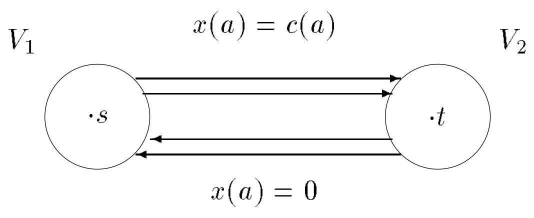
(1): Follows from rearranging the terms. Note that we assumed that no arcs enter s.

(2): Follows from removing the flow through the arcs (v, w) with $v, w \in V_1$; a flow through such an arc contributes $x_{(v,w)}$ to both terms.

(3): Flows are non-negative.

(4): The flow through any arc does not exceed its capacity.

Note that $f = c(V_1, V_2)$ if equality holds for (3) and (4). Inequality (3) becomes an equality if the arcs from V_2 to V_1 do not contain flow. Inequality (4) becomes an equality if all arcs from V_1 to V_2 contain flow at full capacity. This situation is depicted below.



Theorem 2.10. (Ford & Fulkerson) Maxflow = mincut.
If no augmenting path can be found, then there exists an (s, t)-cut with capacity equal to the value of the current flow.

Proof.
Consider the labeling of the vertices after Algorithm APF has stopped. Since no augmenting

path was determined, we have that t is labeled N. Define V_1 as the set of vertices v with label Y. Note that $s \in V_1$ and $t \in V_2$, which implies that this partition forms an (s,t)-cut. Consider any forward arc in the (s,t)-cut, that is, an arc (v,w) with $v \in V_1$ and $w \in V_2$. Since w is labeled N, we must have that this arc carries flow up to its capacity. Similarly, we know that every backward arc in the (s,t)-cut carries no flow at all. These are exactly the conditions that make (3) and (4) equalities, which was to be proved. $\square$

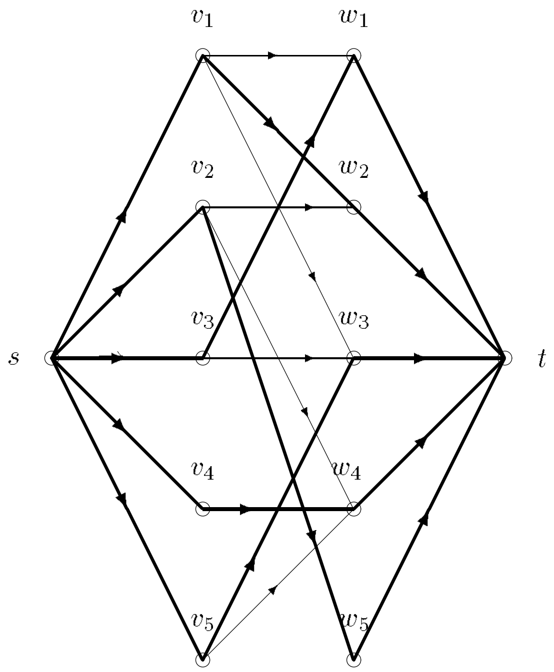
Theorem 2.10 gives a good characterization of maximum flows, i.e., for a given flow one can prove optimality *and* non-optimality by a simple argument: An optimal flow is shown to be optimal by indicating a cut with the same value, whereas a non-optimal flow is shown to be non-optimal by increasing the value of the flow, i.e., by indicating an augmenting path.

Notes

In the literature the 'incremental graph' is often used to find an augmenting path. For a given flow x from s to t in the network (G,c), each arc $a=(v,w)$ with flow $x(a)$ is replaced by two arcs as follows. If $c(a)-x(a)>0$, then we introduce the arc (v,w) with capacity $c(a)-x(a)$; if $x(a)>0$, then we introduce the arc (w,v) with capacity $x(a)$. This is depicted below.

In the incremental graph, we then just have to find a path from s to t. The advantage of using the incremental graph in this respect is that the algorithm gets well-organized; the disadvantage is that in the determination of the incremental graph every arc is evaluated, while it needs not to be considered in the determination of the augmenting path.

Maximum matching versus maximum flow.
We can solve the problem of finding a matching with maximum cardinality in case of a bipartite graph as a special case of the maximum flow algorithm, as is illustrated by the following example. The edges of $G(V_1 \cup V_2, E)$ are directed from V_1 to V_2. We add a source s with arcs (s, v) $(v \in V_1)$ and a sink t with arcs (w, t) $(w \in V_2)$. All arcs get capacity 1.



The flow problem corresponding to the system
of distinct representatives of Figure 2.2.

The fat arcs contain a flow of value one. The fat arcs connecting vertices from V_1 to V_2 correspond to edges in G that belong to the matching. Both labeling algorithms APF and APM label the vertices $\{v_3, v_4, v_5, w_2, w_5\}$. The flow can be augmented if the matching can be augmented, and vice versa.

Complexity
If the capacities of the arcs are finite, then so is the algorithm. Each (s,t)-cut has finite capacity, and in each iteration, that is, each time APF is called on, the flow is augmented by an integral number of units. Hence, the number of calls on APF is bounded from above by the capacity of the minimum cut.

The following example shows that it may take a while to find the optimal flow if the augmenting paths are not chosen carefully.

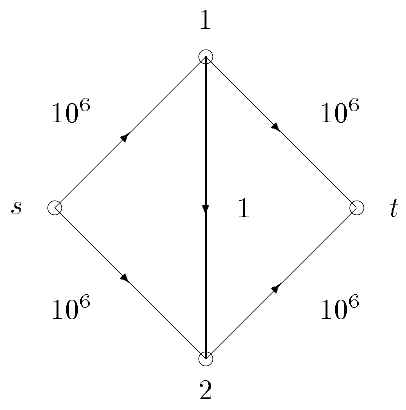


Figure 2.10.

The numbers denote capacities. We can take the paths $(s, 1, 2, t)$ and $(s, 2, 1, t)$ alternately 10^6 times. Each time, the flow is augmented by one unit. We could have taken the paths $(s, 1, t)$ and $(s, 2, t)$ one time each, instead. It seems better to take short paths or paths in which the flow can be augmented by as much as possible.

If capacities are allowed to be arbitrary reals, one can even provide examples where the flow does not even converge to the optimum, if unlucky choices for the augmenting paths are made.

From the example, it is clear that algorithm MF may have to make many calls on subalgorithm APF, if unlucky choices are made. In fact the number of calls can be equal to the capacity of a minimum cut, like in the example. An upper bound on the capacity of a minimum cut is the capacity of the cut $(\{s\}, V\backslash\{s\})$, which amounts to at most $n \cdot c$, where c is the maximum capacity of the arcs. Since the number of elementary steps of algorithm APF is $O(m)$, the total running time is $O(n \cdot m \cdot c)$. This does not seem to be really efficient as the example shows. We will now formalize what we mean with efficient.

First, observe that we cannot just look at the time needed by a computer to run the algorithm, since this is instance-dependent. We will only call an algorithm efficient if it is efficient for *each possible instance*; hence, we have to consider the *running time* of the algorithm.

An algorithm is defined to be *efficient* or *polynomial* if the number of elementary steps $r(n)$, that is, the running time, is bounded by a polynomial in the size of the input $f(n)$. More precisely, if $r(n) = O(p(f(n)))$ where p is a polynomial. This definition depends on the 'size of the input', which is measured in the number of bits needed to encode the instance.

If we return to the maximum flow problem, we see that any instance of this problem consists of a digraph $G = (V, A)$ and a capacity function $c : A \to \mathbb{Z}^+$. We need $O(n)$ bits to encode the vertices V and $O(m)$ bits to encode the arcs A. To encode the capacity c of an arc, we need $O(\log c)$ bits, and hence, we need $O(m \log c)$ bits to encode the entire capacity function. Since $n \leq m \leq m \log c$, the number of bits necessary to encode an instance of the maximum flow problem amounts to $O(m \log c)$. We see that the running time is not polynomial in the input size: c grows exponentially when $\log c$ grows linearly.

Although the maximum flow algorithm is not efficient in the way it is stated above, it can be made to run in polynomial time, if the proper choices for augmenting paths are made.