Chapter 5
Incremental Method and Maximum Network Flow

Change is incremental. Change is small.

—Theodore Melfi

In this chapter, we study the incremental method which is very different from those methods in the previous chapters. This method does not use the self-reducibility. It starts from a feasible solution, and in each iteration, computation moves from a feasible solution to another feasible solution by improving the objective function value. The incremental method has been used in the study of many problems, especially in the study of network flow.

5.1 Maximum Flow

Consider a *flow network* $G = (V, E)$, i.e., a directed graph with a nonnegative capacity $c(u, v)$ on each arc (u, v), and two given nodes, *source* s and *sink* t. An example of the flow network is shown in Fig. 5.1. For simplicity of description for flow, we may extend capacity $c(u, v)$ to every pair of nodes u and v by defining $c(u, v) = 0$ if $(u, v) \notin E$.

A *flow* in flow network G is a real function f on $V \times V$ satisfying the following three conditions:

1. (Capacity constraint) $f(u, v) \le c(u, v)$ for every $u, v \in V$.
2. (Skew symmetry) $f(u, v) = -f(v, u)$ for all $u, v \in V$.
3. (Flow conservation) $\sum_{v \in V \setminus \{u\}} f(u, v) = 0$ for every $u \in V \setminus \{s, t\}$.

The flow has the following properties.

Lemma 5.1.1 *Let f be a flow of network $G = (V, E)$. Then the following holds:*

(a) If $(u, v) \notin E$ and $(v, u) \notin E$, then $f(u, v) = 0$.
(b) For any $x \in V \setminus \{s, t\}$, $\sum_{f(u,x)>0} f(u, x) = \sum_{f(x,v)>0} f(x, v)$.

D.-Z. Du et al., *Introduction to Combinatorial Optimization*, Springer Optimization and Its Applications 196, https://doi.org/10.1007/978-3-031-10596-8_5

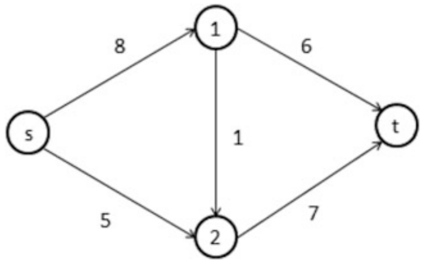

Fig. 5.1 A flow network

(c) $\sum_{f(s,v)>0} f(s,v) - \sum_{f(u,s)>0} f(u,s) = \sum_{f(u,t)>0} f(u,t) - \sum_{f(t,v)>0} f(t,v)$.

Proof

(a) By capacity constraint, $f(u,v) \leq c(u,v) = 0$ and $f(v,u) \leq c(v,u) = 0$. By skew symmetric, $f(u,v) = -f(v,u) \geq 0$. Hence, $f(u,v) = 0$.

(b) By flow conservation, for any $x \in V \setminus \{s,t\}$,

$$\sum_{f(x,u)<0} f(x,u) + \sum_{f(x,v)>0} f(x,v) = \sum_{v \in V} f(x,v) = 0.$$

By skew symmetry,

$$\sum_{f(u,x)>0} f(u,x) = -\sum_{f(x,u)<0} f(x,u) = \sum_{f(x,v)>0} f(x,v).$$

(c) By (b), we have

$$\sum_{x \in V \setminus \{s,t\}} \sum_{f(u,x)>0} f(u,x) = \sum_{x \in V \setminus \{s,t\}} \sum_{f(x,v)>0} f(x,v).$$

For $(y,z) \in E$ with $y, z \in V \setminus \{s,t\}$, if $f(y,z) > 0$, then $f(y,z)$ appears in both the left-hand and the right-hand sides, and hence it will be cancelled. After cancellation, we obtain

$$\sum_{f(s,v)>0} f(s,v) + \sum_{f(t,v)>0} = \sum_{f(u,s)>0} f(u,s) + \sum_{f(u,t)>0} f(u,t).$$

□

Now, the *flow value* of f is defined to be

$$|f| = \sum_{f(s,v)>0} f(s,v) - \sum_{f(u,s)>0} f(u,s) = \sum_{f(u,t)>0} f(u,t) - \sum_{f(t,v)>0} f(t,v).$$

In case that the source s does not have arc coming in, we have

$$|f| = \sum_{f(s,v)>0} f(s,v).$$

In general, we can also represent $|f|$ as

$$|f| = \sum_{v \in V \setminus \{s\}} f(s,v) = \sum_{u \in V \setminus \{t\}} f(u,t).$$

In Fig. 5.2, arc labels with underline give a flow. This flow has value 11.

The maximum flow problem is as follows.

Problem 5.1.2 (Maximum Flow) Given a flow network $G = (V, E)$ with arc capacity $c : V \times V \rightarrow R_+$, a source s, and a sink t, find a flow f with maximum flow value. Usually, assume that s does not have incoming arc and t does not have outgoing arc.

An important tool for study of the maximum flow problem is the residual network. The *residual network* for a flow f in a network $G = (V, E)$ with capacity c is the flow network with $G_f(V, E')$ with capacity $c'(u, v) = c(u, v) - f(u, v)$ for any $u, v \in V$ where $E' = \{(u, v) \in V \times V \mid c'(u, v) > 0\}$. For example, the flow in Fig. 5.2 has its residual network as shown in Fig. 5.3. Two important properties of the residual network are included in the following lemmas.

Lemma 5.1.3 *Suppose f' is a flow in the residual network G_f. Then $f + f'$ is a flow in network G and $|f + f'| = |f| + |f'|$.*

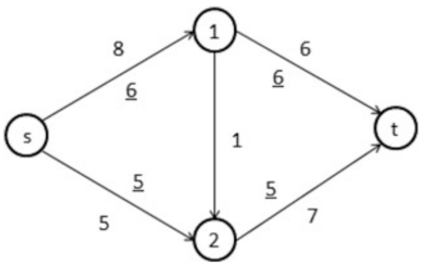

Fig. 5.2 A flow in network

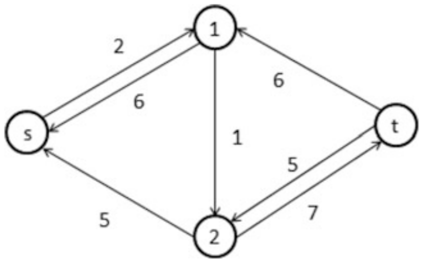

Fig. 5.3 The residual network G_f of the flow f in Fig. 5.2

Proof For any $u, v \in V$, since $f'(u, v) \leq c'(u, v) = c(u, v) - f(u, v)$, we have $f(u, v) + f'(u, v) \leq c(u, v)$, that is, $f + f'$ satisfies the capacity constraint. Moreover, $f(u, v) + f'(u, v) = -f(v, u) - f'(v, u) = -(f(v, u) + f'(v, u))$ and for every $u \in V \setminus \{s, t\}$,

$$\sum_{v \in V \setminus \{u\}} (f + f')(u, v) = \sum_{v \in V \setminus \{u\}} f(u, v) + \sum_{v \in V \setminus \{u\}} f'(u, v) = 0.$$

This means that $f + f'$ satisfies the skew symmetry and the flow conservation conditions. Therefore, $f + f'$ is a flow. Finally,

$$|f + f'| = \sum_{v \in V \setminus \{s\}} (f + f')(s, v) = \sum_{v \in V \setminus \{s\}} f(s, v) + \sum_{v \in V \setminus \{s\}} f'(s, v) = |f| + |f'|.$$

□

Lemma 5.1.4 *Suppose f' is a flow in the residual network G_f. Then $(G_f)_{f'} = G_{f+f'}$, i.e., the residual network of f' in network G_f is the residual network of $f + f'$ in network G.*

Proof The arc capacity of $(G_f)_{f'}$ is

$$c'(u, v) - f'(u, v) = c(u, v) - f(u, v) - f'(u, v) = c(u, v) - (f + f')(u, v)$$

which is the same as that in $G_{f+f'}$. □

In order to get a flow with larger value, Lemmas 5.1.3 and. 5.1.4 suggest us to find a flow f' in G_f with $|f'| > 0$. A simple way is to find a path P from s to t and define f' by

$$f'(u, v) = \begin{cases} \min_{(x,y) \in P} c'(x, y) & \text{if } (u, v) \in P, \\ 0 & \text{otherwise.} \end{cases}$$

The following algorithm is motivated from this idea.

Using this algorithm, an example is shown in Fig. 5.4. The s-t path of the residual network is called an *augmenting path*, and hence Ford-Fulkerson algorithm is an augmenting path algorithm (Algorithm 15).

Now, we may have two questions: Can Ford-Fulkerson algorithm stop within finitely many steps? When Ford-Fulkerson algorithm stops, does output reach the maximum?

The answer for the first question is negative, that is, Ford-Fulkerson algorithm may run infinitely many steps. A counterexample can be obtained from the one as shown in Fig. 5.5 by setting $m = \infty$. However, with certain condition, Ford-Fulkerson algorithm will run within finitely many steps.

Algorithm 15 Ford-Fulkerson algorithm for maximum flow

Input: A flow network $G = (V, E)$ with capacity function c, a source s, and a sink t.
Output: A flow f.

```
1: $G \leftarrow G$;
2: $f \leftarrow 0$; (i.e., $\forall u, v \in V, f(u, v) = 0$)
3: while there exists a path $P$ from $s$ to $t$ in $G$ do
4:     $\delta \leftarrow \min\{c(u, v) \mid (u, v) \in P\}$ and
5:     send a flow $f'$ with value $\delta$ from $s$ to $t$ along path $P$;
6:     $G \leftarrow G_{f'}$;
7:     $f \leftarrow f + f'$;
8: end while
9: return $f$.
```

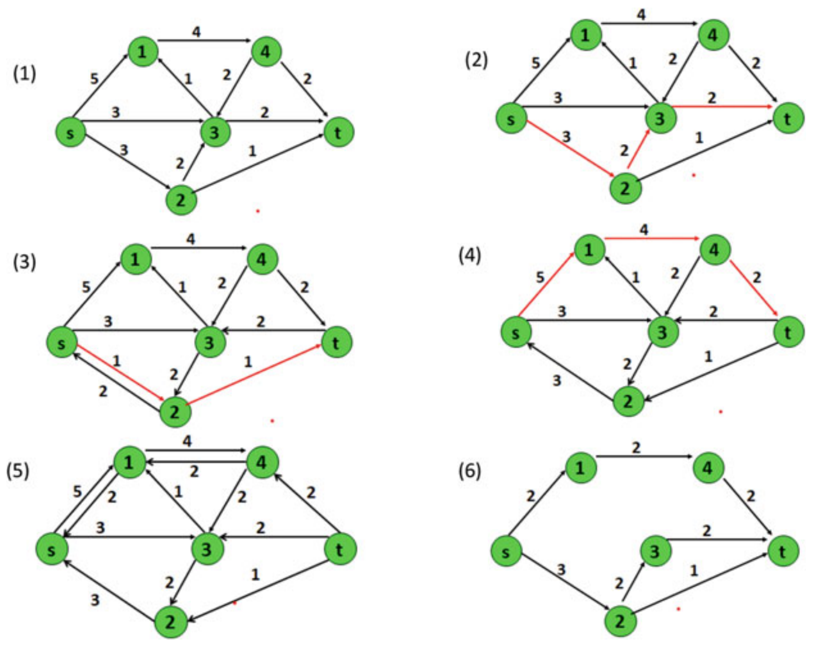

Fig. 5.4 An example for using the Ford-Fulkerson Algorithm

Theorem 5.1.5 *If every arc capacity is a finite integer, then Ford-Fulkerson algorithm runs within finitely many steps.*

Proof The flow value has upper bound $\sum_{(s,v)\in E} c(s, v)$. Since every arc capacity is an integer, in each step, the flow value will be increased by at least one. Therefore, the algorithm will run within at most $\sum_{(s,v)\in E} c(s, v)$ steps. □

Remark Ford-Fulkerson algorithm may run with infinitely many augmentations even if all arc capacities are finite, but there is an irrational arc capacity. Such an example can be found in exercises.

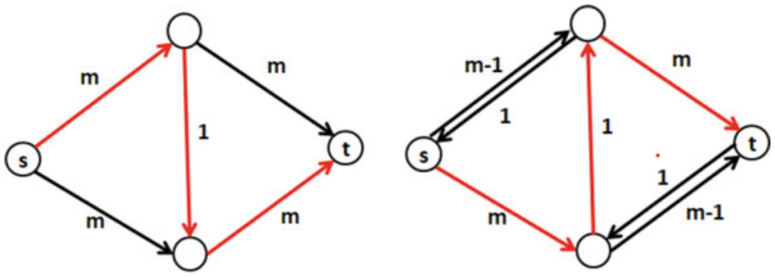

Fig. 5.5 Ford-Fulkerson algorithm runs not in polynomial time

The answer for the second question is positive. Actually, we have the following.

Theorem 5.1.6 *A flow f is maximum if and only if its residual network G_f does not contain a path from source s to sink t.*

To prove this theorem, let us first show a lemma.

A partition (S, T) of V is called an *s-t cut* if $s \in S$ and $t \in T$. The capacity of an s-t cut is defined by

$$\mathrm{CAP}(S, T) = \sum_{u \in S, v \in T} c(u, v).$$

Lemma 5.1.7 *Let (S, T) be an s-t cut. Then for any flow f,*

$$|f| = \sum_{f(u,v)>0, u \in S, v \in T} f(u, v) - \sum_{f(v,u)>0, u \in S, v \in T} f(v, u) \leq CAP(S, T).$$

Proof By Lemma 5.1.1(b),

$$\sum_{x \in S \setminus \{s\}} \sum_{f(u,x)>0} f(u, x) = \sum_{x \in S \setminus \{s\}} \sum_{f(x,v)>0} f(x, v).$$

Simplifying this equation, we will obtain

$$\sum_{f(s,x)>0, x \in S \setminus \{s\}} f(s, x) + \sum_{u \in T, x \in S \setminus \{s\}, f(u,x)>0} f(u, x)$$
$$= \sum_{f(x,s)>0, x \in S \setminus \{s\}} f(x, s) + \sum_{v \in T, x \in S \setminus \{s\}, f(x,v)>0} f(x, v).$$

Thus,

$$\begin{aligned}&\sum_{f(s,x)>0} f(s,x) + \sum_{u\in T, x\in S, f(u,x)>0} f(u,x)\\ &= \sum_{f(x,s)>0} f(x,s) + \sum_{v\in T, x\in S, f(x,v)>0} f(x,v),\end{aligned}$$

that is,

$$\begin{aligned}|f| &= \sum_{f(s,x)>0} f(s,x) - \sum_{f(x,s)>0} f(x,s)\\ &= \sum_{v\in T, x\in S, f(x,v)>0} f(x,v) - \sum_{u\in T, x\in S, f(u,x)>0} f(u,x)\\ &\le \sum_{v\in T, x\in S, f(x,v)>0} f(x,v)\\ &\le \sum_{x\in S, v\in T} c(x,v).\end{aligned}$$

□

Now, we prove Theorem 5.1.6.

Proof of Theorem 5.1.6 If residual network G_f contains a path from source s to sink t, then a positive flow can be added to f and hence f is not maximum. Next, we assume that G_f does not contain a path from s to t.

Let S be the set of all nodes each of which can be reached by a path from s. Set $T = V \setminus S$. Then (S, T) is a partition of V such that $s \in S$ and $t \in T$. Moreover, G_f has no arc from S to T. This fact implies two important facts:

(a) For any arc (u, v) with $u \in S$ and $v \in T$, $f(u, v) = c(u, v)$.
(b) For any arc (v, u) with $u \in S$ and $v \in T$, $f(v, u) = 0$.

Based on these two facts, by Lemma 5.1.7, we obtain that

$$|f| = \sum_{u\in S, v\in T} c(u,v).$$

Hence, f is a maximum flow. □

Corollary 5.1.8 *The maximum flow is equal to minimum s-t cut capacity.*

Finally, we remark that Ford-Fulkerson algorithm is not a polynomial-time. A counterexample is given in Fig. 5.5. On this counterexample, the algorithm runs in $2m$ steps. However, the input size is $O(\log m)$. Clearly, $2m$ is not a polynomial with respect to $O(\log m)$.

5.2 Edmonds-Karp Algorithm

To improve the running time of Ford-Fulkerson algorithm, a simple modification is found which works very well, that is, at each iteration, find a shortest augmenting path instead of an arbitrary augmenting. By the shortest, we mean the path contains the minimum number of arcs. This algorithm is called Admonds-Karp algorithm (Algorithm 16).

An example for using Edmonds-Karp algorithm is shown in Fig. 5.6. Compared with Fig. 5.4, we may find that input flow network is the same, but obtained maximum flows are different. Thus, for this input flow network, there are two different maximum flows. Actually, in this case, there are infinitely many maximum flows. The reader may prove it as an exercise.

To estimate the running time, let us study some properties of Edmonds-Karp algorithm.

Algorithm 16 Edmonds-Karp algorithm for maximum flow

Input: A flow network $G = (V, E)$ with capacity function c, a source s and a sink t.
Output: A flow f.

```
1: $G \leftarrow G$;
2: $f \leftarrow 0$; (i.e., $\forall u, v \in V, f(u, v) = 0$)
3: while there exists a path from $s$ to $t$ in $G$ do
4:     find a shortest path $P$ from $s$ to $t$;
5:     set $\delta \leftarrow \min\{c(u, v) \mid (u, v) \in P\}$ and
6:     send a flow $f'$ with value $\delta$ from $s$ to $t$ along path $P$;
7:     $G \leftarrow G_{f'}$;
8:     $f \leftarrow f + f'$;
9: end while
10: return $f$.
```

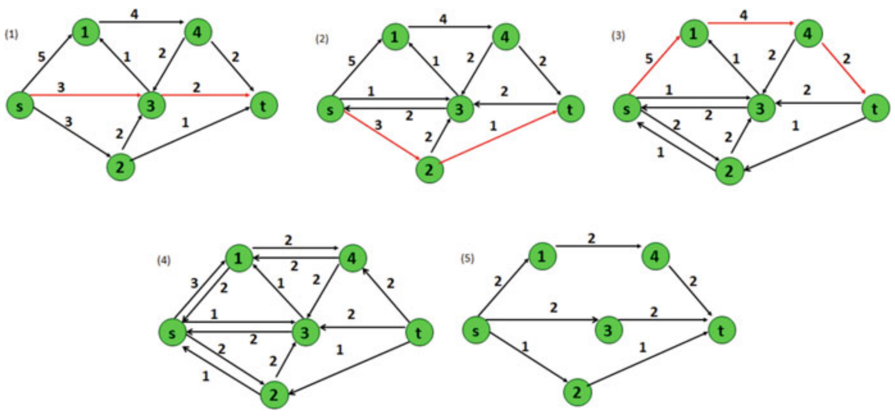

Fig. 5.6 An example for using the Edmonds-Karp algorithm

Let $\delta_f(x)$ denote the shortest path distance from source s to node x in the residual network G_f of flow f where each arc is considered to have unit distance.

Lemma 5.2.1 *When Edmonds-Karp algorithm runs, $\delta_f(x)$ increases monotonically with each flow augmentation.*

Proof For contradiction, suppose flow f' is obtained from flow f through an augmentation with path P and $\delta_{f'}(v) < \delta_f(v)$ for some node v. Without loss of generality, assume $\delta_{f'}(v)$ reaches the smallest value among such v, i.e.,

$$\delta_{f'}(u) < \delta_{f'}(v) \Rightarrow \delta_{f'}(u) \geq \delta_f(u).$$

Suppose arc (u, v) is on the shortest path from s to v in $G_{f'}$. Then $\delta_{f'}(u) = \delta_{f'}(v) - 1$ and hence $\delta_{f'}(u) \geq \delta_f(u)$. Next, let us consider two cases.

Case 1. $(u, v) \in G_f$. In this case, we have

$$\delta_f(v) \leq \delta_f(u) + 1 \leq \delta_{f'}(u) + 1 = \delta_{f'}(v),$$

a contradiction.

Case 2. $(u, v) \notin G_f$. Then arc (v, u) must lie on the augmenting path P in G_f (Fig. 5.7). Therefore,

$$\delta_f(v) = \delta_f(u) - 1 \leq \delta_{f'}(u) - 1 = \delta_{f'}(v) - 2 < \delta_{f'}(v),$$

a contradiction. □

An arc (u, v) is *critical* in residual network G_f if (u, v) has the smallest capacity in the shortest augmenting path in G_f.

Lemma 5.2.2 *Each arc (u, v) can be critical at most $(|V| + 1)/2$ times.*

Proof Suppose arc (u, v) is critical in G_f. Then (u, v) will disappear in the next residual network. Before (u, v) appears again, (v, u) has to appear in augmenting path of a residual network $G_{f'}$. Thus, we have

$$\delta_{f'}(u) = \delta_{f'}(v) + 1.$$

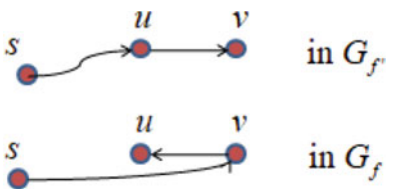

Fig. 5.7 Proof of Lemma 5.2.1

Since $\delta_f(v) \leq \delta_{f'}(v)$, we have

$$\delta_{f'}(u) = \delta_{f'}(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2.$$

By Lemma 5.2.1, the shortest path distance from s to u will increase by $2(k-1)$ when arc (u, v) can be critical k times. Since this distance is at most $|V| - 1$, we have $2(k-1) \leq |V| - 1$, and hence $k \leq (|V| + 1)/2$. □

Now, we establish a theorem on running time.

Theorem 5.2.3 *Edmonds-Karp algorithm runs in time* $O(|V| \cdot |E|^2)$.

Proof In each augmentation, there exists a critical arc. Since each arc can be critical $(|V| + 1)/2$ times, there are at most $O(|V| \cdot |E|)$ augmentations. In each augmentation, finding the shortest path takes $O(|E|)$ time, and operations on the augmenting path take also $O(|E|)$ time. Putting all together, Edmonds-Karp algorithm runs in time $O(|V| \cdot |E|^2)$. □

Note that the above theorem does not require that all arc capacities are integers. Therefore, the modification of Edmonds and Karp is twofold: (1) Make the algorithm halt within finitely many iterations, and (2) the number of iterations is bounded by a polynomial.

5.3 Applications

The maximum flow has many applications. Let us show a few examples in this section.

Example 5.3.1 Given an undirected graph $G = (V, E)$ and two distinct vertices $s, t \in V$, please give an algorithm to determine the connectivity between s and t, i.e., the maximum number of s-to-t paths that are vertex-disjoint paths (other than at s and t).

For each vertex $v \in V$, create two vertices v^+ and v^- together with an arc (v^+, v^-). For each edge $(u, v) \in E$, create two arcs (u^-, v^+) and (v^-, u^+). Then, we obtain a directed graph G' from G (Fig. 5.8). Every path from s to t in G induces a path from s^- to t^+ in G', and a family of vertex-disjoint paths from s to t in G will induce a family of arc-disjoint paths from s^- to t^+, vice versa. Therefore, assign every arc with unit capacity in G'. Then the connectivity between s and t in G is equal to the maximum flow value from s^- to t^+ in G'.

Example 5.3.2 Consider a set of wireless sensors lying in a rectangle which is a piece of boundary area of the region of interest. The region is below the rectangle and outside is above the rectangle. The monitoring area of each sensor is a unit disk, i.e., a disk with radius of unit length. A point is said to be covered by a sensor if it lies in the monitoring disk of the sensor. The set of sensors is called

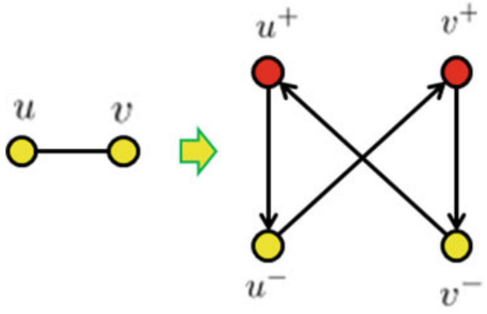

Fig. 5.8 Construct G' from G

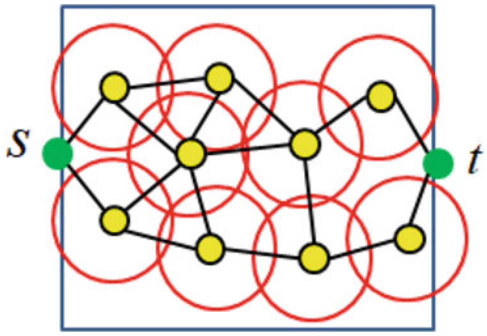

Fig. 5.9 Sensor barrier covers

a *barrier cover* if they can cover a line (not necessarily straight) connecting two vertical edges (Fig. 5.9) of the rectangle. The barrier cover is used for protecting any intruder coming from outside. Sensors are powered with batteries and hence lifetime is limited. Assume that all sensors have unit lifetime. The problem is to find the maximum number of disjoint barrier covers so that they can be used in turn to maximize the lifetime of the system.

Use two points s and t to represent two vertical edges of the rectangle; we call them *vertical lines* s and t, respectively. Construct a graph G by setting the vertex set consisting of all sensors together with s and t (Fig. 5.9). The edge is constructed based on the following rules:

- If the monitoring disk of sensor u and the monitoring disk of sensor v have nonempty intersection, then add an edge (u, v).
- If vertical line s and the monitoring disk of sensor v have nonempty intersection, then add an edge (s, v).
- If the monitoring disk of sensor u and vertical line t have nonempty intersection, then add an edge (u, t).

In graph G, every path between s and t induces a barrier cover, and every set of vertex-disjoint paths between s and t will induce a set of disjoint barrier covers, vice versa. Therefore, we can further construct G' from G as above (Fig. 5.8), so the maximization of disjoint barrier is transformed to the maximum flow problem in G'.

Definition 5.3.3 (Matching) Consider a graph $G = (V, E)$. A subset of edges is called a *matching* if edges in the subset are not adjacent to each other. In other words, a matching is an independent edge subset. A *bipartite* matching is a matching in a bipartite graph.

Example 5.3.4 (Maximum Bipartite Matching) Given a bipartite graph (U, V, E), find a matching with maximum cardinality.

This problem can be transformed into a maximum flow problem as follows. Add a source node s and a sink node t. Connect s to every node u in U by adding an arc (s, u). Connect every node v in V to t by adding an arc (v, t). Add to every edge in E the direction from U to V. Finally, assign every arc with unit capacity. An example is shown in Fig. 5.10.

Motivated from observation on the example in Fig. 5.10, we may have questions:

(1) Can we do augmentation directly in bipartite graph without putting it in a flow network?
(2) Can we perform the first three augmentations in the same time?

For both questions, the answer is yes. Let us explain the answer in the next section.

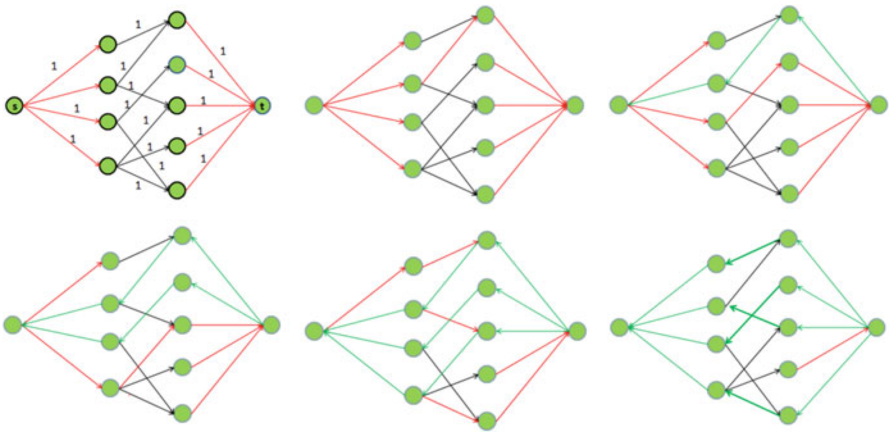

Fig. 5.10 Maximum bipartite matching is transformed to maximum flow

5.4 Matching

In this section, we study matching in a directed way. First, we define the augmenting path as follows.

Consider a matching M in a bipartite graph $G = (U, V, E)$. Let us call every edge in M as *matched* edge and every edge not in M as *unmatched* edge. A node v is called a *free node* if v is not an ending point of a matched edge.

Definition 5.4.1 (Augmenting Path) The *augmenting path* is now defined to be a path satisfying the following:

- It is an *alternating path*, that is, edges on the path are alternatively unmatched and matched.
- The path is between two free nodes.

There are totally odd number of edges in an augmenting path. The number of unmatched edges is one more than the number of matched edges. Therefore, on an augmenting path, turn all matched edges to unmatched and turn all unmatched edges to matched. Then considered matching will become a matching with one more edge. Therefore, if a matching M has an augmenting path, then M cannot be maximum. The following theorem indicates that the inverse holds.

Theorem 5.4.2 *A matching M is maximum if and only if M does not have an augmenting path.*

Proof Let M be a matching without augmenting path. For contradiction, suppose M is not maximum. Let M^* be a maximum matching. Then $|M| < |M^*|$. Consider $M \oplus M^* = (M \setminus M^*) \cup (M^* \setminus M)$, in which every node has degree at most two (Fig. 5.11).

Hence, it is disjoint union of paths and cycles. Since each node with degree two must be incident to two edges belonging to M and M', respectively. Those paths and cycles must be alternative. They can be classified into four types as shown in Fig. 5.12.

Note that in each of the first three types of connected components, the number of edges in M is not less than the number of edges in M^*. Since $|M| < |M^*|$, we have $|M \setminus M^*| < |M^* \setminus M|$. Therefore, the connected component of the fourth type must exist, that is, M has an augmenting path, a contradiction. □

We now return to the question on augmentation of several paths at the same time. The following algorithm is the result of a positive answer.

We next analyze Hopcroft-Karp algorithm (Algorithm 17).

Lemma 5.4.3 *In each iteration, the length of the shortest augmenting path is increased by at least two.*

Proof Suppose matching M' is obtained from matching M through augmentation on a maximal set of shortest augmenting paths, $\{P_1, P_2, \ldots, P_k\}$, for M. Let P be a shortest augmenting path for M'. If P is disjoint from $\{P_1, P_2, \ldots, P_k\}$, then P is

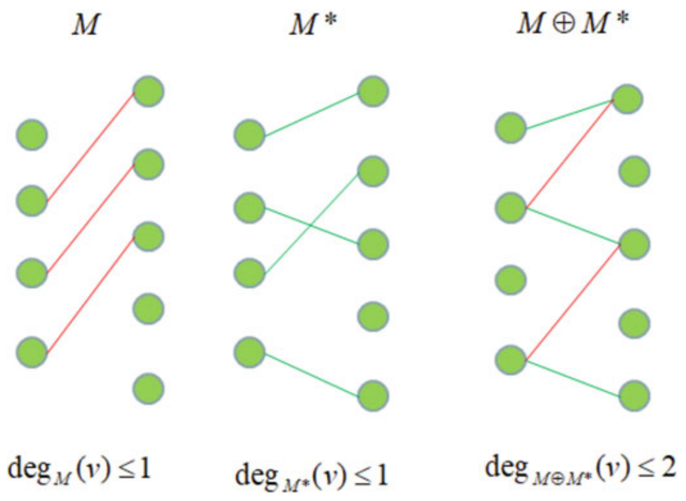

Fig. 5.11 $M \oplus M^*$

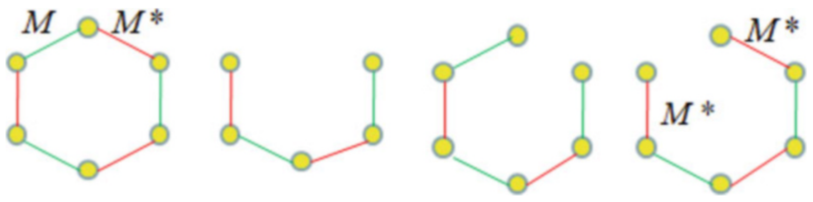

Fig. 5.12 Connected components of $M \oplus M^*$

Algorithm 17 Hopcroft-Karp algorithm for maximum bipartite matching

Input: A bipartite graph $G = (U, V, E)$.
Output: A maximum matching M.

```
1: $M \leftarrow$ any edge;
2: **while** there exists an augmenting path **do**
3:     find a maximal set of disjoint augmenting paths $\{P_1, P_2, \ldots, P_k\}$;
4:     $M \leftarrow M \oplus (P_1 \cup P_2 \cup \cdots P_k)$;
5: **end while**
6: **return** $M$.
```

also an augmenting path for M. Hence, the length of P is longer than the length of P_1. Note that the augmenting path must have odd length. Therefore, the length of P at-least-two longer than the length of P_1.

Next, assume that P has an edge lying in P_i for some i. Note that every augmenting path has two endpoints in U and V, respectively. Let u and v be two endpoints of P and u_i and v_i two endpoints of P_i where $u, u_i \in U$ and $v, v_i \in V$. Without loss of generality, assume that (x, y) is the edge lying on P and also on some P_i such that no such edge exists from y to v. Clearly,

$$\text{dist}_P(y, v) \geq \text{dist}_{P_i}(y, v_i), \tag{5.1}$$

where $\text{dist}_P(y, v)$ denotes the distance between y and v on path P. In fact, if $\text{dist}_P(y, v) < \text{dist}_{P_i}(y, v_i)$, then replacing the piece of P_i between y and v_i by the piece of P between y and v, we obtain an augmenting path for M, shorter than P_i, contradicting to shortest property of P_i. Now, we claim that the following holds.

$$\text{dist}_{P_i}(u_i, y) + 1 = \text{dist}_{P_i}(u_i, x) \leq \text{dist}_P(u, x) = \text{dist}_P(u, y) - 1. \tag{5.2}$$

To prove this claim, we may put the bipartite graph into a flow network as shown in Fig. 5.10. Then every augmenting path receives a direction from U to V, and the claim can be proved as follows.

Firstly, note that on path P, we assumed that the piece from y to v is disjoint from all $P_1, P_2, \ldots, P_k$. This assumption implies that edge (x, y) is in direction from x to y on P, so that $\text{dist}_P(u, x) = \text{dist}_P(u, y) - 1$.

Secondly, note that edge (x, y) also appears on P_i, and after augmentation, every edge in P_i must change its direction. Thus, edge (x, y) is in direction from y to x on P_i. Hence, $\text{dist}_{P_i}(u_i, y) + 1 = \text{dist}_{P_i}(u_i, x)$.

Thirdly, by Lemma 5.2.1, we have $\text{dist}_{P_i}(u_i, x) \leq \text{dist}_P(u, x)$.

Finally, putting (5.1) and (5.2) together, we obtain

$$\text{dist}_{P_i}(u_i, v_i) + 2 \leq \text{dist}_P(u, v).$$

□

Theorem 5.4.4 *Hopcroft-Karp algorithm computes a maximum bipartite matching in time* $O(|E|\sqrt{|V|})$.

Proof In each iteration, it takes $O(|E|)$ time to find a maximal set of shortest augmenting paths and to perform augmentation on these paths. (We will give more explanation after the proof of this theorem.) Let M be the matching obtained through $\sqrt{|V|}$ iterations. Let M^* be the maximum matching. Then $M \oplus M^*$ contains $|M^*| \setminus |M|$ augmenting path, each of length at least $1 + 2\sqrt{|V|}$ by Lemma 5.4.3. Therefore, each takes at least $2 + 2\sqrt{|V|}$ nodes. This implies that the number of augmenting paths in $M \oplus M^*$ is upper bounded by

$$|V|/(2 + 2\sqrt{|V|}) < \sqrt{|V|}/2.$$

Thus, M^* can be obtained from M through at most $\sqrt{|V|}/2$ iterations. Therefore, M^* can be obtained within at most $\frac{3}{2} \cdot \sqrt{|V|}$ iterations. This completes the proof. □

There are two steps in finding a maximal set of disjoint augmenting paths for a matching M in bipartite graph $G = (U, V, E)$.

In the first step, employ the breadth-first search to put nodes into different levels as follows. Initially, select all free nodes in U and put them in the first level. Next,

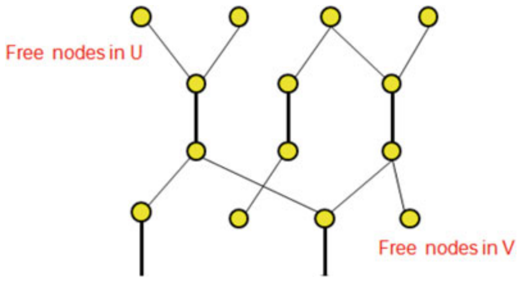

Fig. 5.13 The breadth-first search

put in the second level all nodes each with an unmatched edge connecting to a node in the first level. Then, put in the third level all nodes each with a matched edge connecting to a node in the second level. Continue in this alternating ways, until a free node in V is discovered, say in the kth level (Fig. 5.13). Let F be all free nodes in the kth level and H the obtained subgraph. If the breadth-first search comes to an end and still cannot find a free node in V, then this means that there is no augmenting path, and a maximum matching has already obtained by Hopcroft-Karp algorithm.

In the second step, employ the depth-first search to find path from each node in F to a node in the first level. Such paths will be searched one by one in H, and once a path is obtained, all nodes on this depth-first-search path will be deleted from H, until no more such path can be found.

Since both steps can work in $O(|E|)$ time, the total time for finishing this task is $O(|E|)$.

The alternating path method can also be used for the maximum matching in general graph.

Problem 5.4.5 (Maximum Graph Matching) Given a graph $G = (V, E)$, find a matching with maximum cardinality.

The *augmenting path* is also defined to be a path satisfying the following:

- It is an *alternating path*, that is, edges on the path are alternatively unmatched and matched.
- The path is between two free nodes.

Now, the proof of Theorem. 5.4.2 can be applied to the graph matching without any change, to show the following.

Theorem 5.4.6 *A matching M is maximum if and only if M does not have an augmenting path.*

Therefore, we obtained Algorithm 18 for the maximum graph matching problem.

How to find an augmenting path for matching in a general graph $G = (V, E)$? Let us introduce the Blossom algorithm of Edmonds. A *blossom* is an almost alternating odd cycle as shown in Fig. 5.14.

Algorithm 18 Algorithm for maximum graph matching

Input: A graph $G = (V, E)$.
Output: A maximum matching M.

```
1: $M \leftarrow \{$an arbitrary edge$\}$;
2: while there exists an augmenting path $P$ do
3:     $M \leftarrow M \oplus P$;
4: end while
5: return $M$.
```

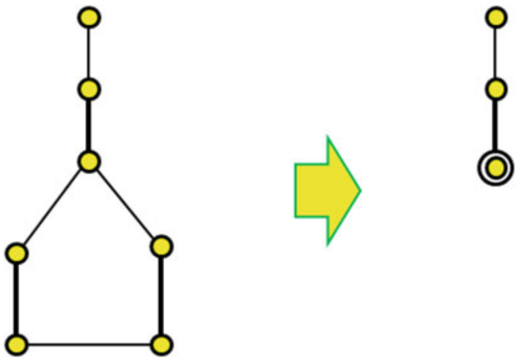
Fig. 5.14 A blossom shrinks into a node

The blossom algorithm is similar to the first step of augmenting-path finding in Hopcroft-Karp algorithm, i.e., employ the breadth-first search by using unmatched edge and matched edge alternatively. However, start from one free node x at a time.

Definition 5.4.7 (Alternating Tree) The alternating tree has a root at a free node x. Its first level consists of unmatched edges, its second level consists of matched edges, and alternatively continue.

Definition 5.4.8 (Even and Odd Nodes) In an alternating tree, a node is called an *odd* node if its distance to the root has odd length. A node is called an *even* node if its distance to the root has even length, e.g., the root is an even node.

Now, let us describe the blossom algorithm.

- For each free node x, construct an alternating tree with root x in the breadth-first-search ordering.
- At an odd node y, if no matched edge is incident to y, then y is a free node, and an augmenting path from x to y is found. If there exists a matched edge incident to y, then such a matched edge is unique, and y can be extended uniquely to an even node.
- At an even node z, if no unmatched edge is incident to z, then z cannot be extended. If there exists an unmatched edge (u, z) incident to z, then consider another ending node u of this edge. If u is a known even node, then a blossom is found; shrink the blossom into an even node. If u is not a known even node, then u can be counted as an odd node to continue our construction.

- At a level consisting of even nodes, if none of them can be extended, then there is no augmenting path starting from free node x.
- Therefore, above construction of alternating tree, we can either find an augmenting path starting from free node x or determine not existing of such a path. As soon as an augmenting path is found, we can carry out an augmentation, matching is updated, and we restart to search for an augmenting path.
- If for all free nodes, no augmenting path can be found from construction of alternating trees, then current matching is maximum.

To show the correctness of the above algorithm, it is sufficient to explain why we can shrink a blossom into a node. An explanation is given in the following.

Lemma 5.4.9 *Let B be a blossom in graph G. Let G/B denote the graph obtained from G by shrinking B into a node. Then G contains an augmenting path if and only if G/B contains an augmenting path.*

Proof Note that the alternating path can be extended passing through a blossom out-reach to its any connection (Fig. 5.15). Therefore, if an augmenting path passes through a blossom, then after shrink the blossom into a node, the augmenting path is still an augmenting path. Conversely, if an augmenting path contains a node which is obtained from a blossom, then after de-shrink the blossom, we can still obtain an augmenting path. □

Clearly, this algorithm runs in $O(|V| \cdot |E|)$ time. Thus, we have the following.

Theorem 5.4.10 *With blossom algorithm, the maximum cardinality matching in graph $G = (V, E)$ can be computed in $O(|V|^2 \cdot |E|)$ time.*

Proof To obtain a maximum matching, we can carry out at most $|V|$ augmentations. To find an augmenting path, we may spend $O(|V| \cdot |E|)$ time to construct alternating trees. Therefore, the total running time is $O(|V|^2|E|)$. □

For weighted bipartite matching and weighted graph matching, can we use the alternating path to deal with them? The answer is yes. However, it is more complicated. We can find a better way, which will be introduced in Sect. 6.8.

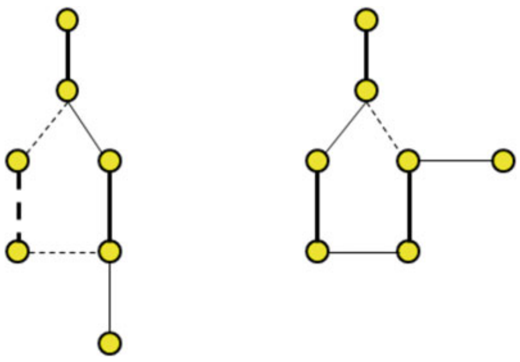
Fig. 5.15 An alternating path passes a blossom

5.5 Dinitz Algorithm

In this and the next sections, we present more algorithms for the maximum flow problem. They have running time better than Edmonds-Karp algorithm.

First, we note that the idea in Hopcroft-Karp algorithm can be extended from matching to flow. This extension gives a variation of Edmonds-Karp algorithm, called Dinitz algorithm.

Consider a flow network $G = (V, E)$. The algorithm starts with a zero flow $f(u, v) = 0$ for every arc (u, v). In each substantial iteration, consider residual network G_f for flow f. Start from source node s to do the breadth-first search until node t is reached. If t cannot be researched, then algorithm stops, and the maximum flow is already obtained. If t is reached with distance ℓ from node s, then the breadth-first-search tree contains ℓ level, and its nodes are divided into ℓ classes $V_0, V_1, \ldots, V_\ell$ where V_i is the set of all nodes each with distance i from s and $\ell \leq |V|$. Collect all arcs from V_i to V_{i+1} for $i = 0, 1, \ldots, \ell - 1$. Let $L(s)$ be the obtained levelable subnetwork. Above computation can be done in $O(|E|)$ time.

Next, the algorithm finds augmenting paths to do augmentations in the following way.

Step 1. Iteratively, for $v \neq t$ and $u \neq s$, remove, from $L(s)$, every arc (u, v) with no coming arc at u or no outgoing arc at v. Denote by $\hat{L}(s)$ the obtained levelable network.

Step 2. If $\hat{L}(s)$ is empty, then this iteration is completed, and go to the next iteration. If $\hat{L}(s)$ is not empty, then it contains a path of length ℓ, from s to t. Find such a path P by using the depth-first search. Do augmentation along the path P. Update $L(s)$ by using $\hat{L}(s)$ and deleting all critical arcs on P. Go to Step 1.

This algorithm has the following property.

Lemma 5.5.1 *Let $\delta_f(s, t)$ denote the distance from s to t in residual graph G_f of flow f. Suppose flow f' is obtained from flow f through an iteration of Dinitz algorithm. Then $\delta_{f'}(s, t) \geq \delta_f(s, t) + 2$.*

Proof The proof is similar to the proof of Lemma 5.2.1. □

The correctness of Dinitz algorithm is stated in the following theorem.

Theorem 5.5.2 *Dinitz algorithm produces a maximum flow in $O(|V|^2|E|)$ time.*

Proof By Lemma 5.5.1, Dinitz algorithm runs within $O(|V|)$ iterations. Let us estimate the running time in each iteration.

- The construction of $L(s)$ spends $O(|E|)$ time.
- It needs $O(|V|)$ time to find each augmenting path and to do augmentation. Since each augmentation will remove at least one critical arc, there are at most $O(|E|)$ augmentations. Thus, the total time for augmentations is $O(|V| \cdot |E|)$.
- Amortizing all time for removing arcs, it is at most $O(|E|)$.

Therefore, each iteration runs in $O(|V| \cdot |E|)$ time. Hence, Dinitz algorithm runs in $O(|V|^2|E|)$ time. At the end of the algorithm, G_f does not contain a path from s to t. Thus, f is a maximum flow. □

5.6 Goldberg-Tarjan Algorithm

In this section, we study a different type of incremental method for maximum network flow. In this method, a valid label will play an important role. This valid label will be on each arc to guide the incremental direction.

Consider a flow network $G = (V, E)$ with capacity $c(u, v)$ for each arc $(u, v) \in E$; s and t are source and sink, respectively. As usual, for simplicity of description, we extend capacity $c(u, v)$ to every pair of nodes u and v by defining $c(u, v) = 0$ if $(u, v) \notin E$.

A function $f : V \times V \to R$ is called a *preflow* if

1. (Capacity constraint) $f(u, v) \le c(u, v)$ for every $u, v \in V$.
2. (Skew symmetry) $f(u, v) = -f(v, u)$ for all $u, v \in V$.
3. For every $v \in V \setminus \{s, t\}$, $\sum_{v \in V \setminus \{u\}} f(u, v) \ge 0$, i.e., $\sum_{(u,v)\in E} f(u, v) \ge \sum_{(v,w)\in E} f(v, w)$.

Compared with those three conditions in the definition of flow, the first two are the same, and the third one is different. The flow conservation condition is relaxed to allow more flow coming than going out at any node other than s and t. This difference is called the *excess* at node v and denotes

$$e(v) = \sum_{(u,v)\in E} f(u, v) \ge \sum_{(v,w)\in E} f(v, w).$$

A node v is said to be *active* if $e(v) > 0$, $v \ne s$, and $v \ne t$. In preflow-relabel algorithm, the excess will be pushed from an active node toward the sink, relying on the valid distance label $d(v)$ for $v \in V$, satisfying the following conditions.

- $d(t) = 0$.
- $d(u) \le d(v) + 1$ for $(u, v) \in E$.

An arc (u, v) is said to be *admissible* if $d(u) = d(v) + 1$ and $c(u, v) > 0$. Note that if we consider a residual graph, then $c(u, v)$ should be considered as updated capacity.

Lemma 5.6.1 *Let $dist(u, v)$ denote the minimum number of arcs on the path from u to v. Then $d(u) \le dist(u, t)$.*

Proof It can be proved by induction on $dist(u, t)$. For $dist(u, t) = 0$, u must be t, and hence $d(t) \le dist(t, t)$. For $dist(u, t) = k > 0$, suppose $(u, u_1, \dots, u_k = t)$ is the shortest path from u to t. Then $dist(u_1, t) = k - 1$. By induction hypothesis,

$d(u_1) \le dist(u_1, t)$. Hence,

$$d(u, t) \le 1 + d(u_1) \le 1 + dist(u_1, t) = dist(u, t).$$

□

Next, we explain two operations, push and relabel. Consider an active node v. Suppose that there exists an admissible arc (v, w). Then a flow $\min(e(v), c(v, w))$ will be pushed along arc (v, w). If $e(v) \le c(v, w)$, then it is called a *saturated push*. Otherwise, the push is called a *non-saturated* one.

Suppose that there does not exist an admissible arc (v, w). Then relabel $d(v)$ by setting

$$d(v) = 1 + \min\{d(w) \mid c(v, w) > 0\}.$$

An important observation is stated in the following lemma.

Lemma 5.6.2 *After push and relabel, the (residual) network and label $d(\cdot)$ are updated. However, $d(\cdot)$ is still a valid label for updated network. Hence, Lemma 5.6.1 holds. Moreover, each relabel for node v will increase its label at least one.*

Proof First, consider a push along arc (v, w). This push may add a new arc (w, v) to the residual network. Therefore, we need to make sure $d(w) \le d(v) + 1$. This is true because $d(v) = d(w) + 1$.

Next, consider relabel node v. By the rule, new label for v is upper bounded by $d(w) + 1$ for any arc (v, w) with $c(v, w) > 0$. Moreover, suppose $(v, w') = \text{argmin}\{d(w) \mid c(v, w) > 0\}$. Since (v, w') is not admissible, we have $d(v) \le d(w')$. Hence, the new label for v is $d(w') + 1 \ge d(v) + 1$. □

Now, we are ready to describe the push-relabel algorithm of Goldberg and Tarjan (see Algorithm 19). An example is shown as in Fig. 5.16.

We next analyze this algorithm.

Lemma 5.6.3 *Let f be a preflow appearing in computation process of Goldberg-Tarjan algorithm. Then, in residual network G_f, every active node v has a path connecting to s.*

Proof In flow decomposition of f, there is a path flow from s to active node v. This path flow will result in a path from v to s in residual network G_f. □

Lemma 5.6.4 *For any node v, $d(v) \le 2n$ during computation, and there are at most $2n$ relabels at each node v, where n is the number of nodes. Moreover, all relabels need at most $O(mn)$ time of computation.*

Proof Note that the relabel occurs only at active nodes. If a node has never been active, then its label is at most $n - 1$. If a node v has been active, then at last time that v is active, there is a path from v to s. After push, this path still exists, and

Algorithm 19 Goldberg-Tarjan algorithm for maximum flow

Input: A flow network $G = (V, E)$ with source s, sink t, and capacity $c(u, v)$ for every $(u, v) \in E$.
Output: A flow $f : V \times V \to R$.

```
 1: $f(u, v) \leftarrow 0$ for all $(u, v) \in V \times V$;
 2: $d(v) \leftarrow dist(v, t)$;
 3: $f(s, v) \leftarrow c(s, v)$ for $(s, v) \in E$;
 4: $d(s) \leftarrow n$; $(n = |V|)$
 5: $G \leftarrow G_f$;
 6: while there is an active node $v$ do
 7:     if there is an admissible arc $(v, w)$ then
 8:         $f(v, w) \leftarrow \min(e(v), c(v, w))$
 9:     else
10:         $d(v) \leftarrow \min\{d(w) \mid c(v, w) > 0\}$
11:     end if
12:     $G \leftarrow G_f$;
13: end while
14: return $f$
```

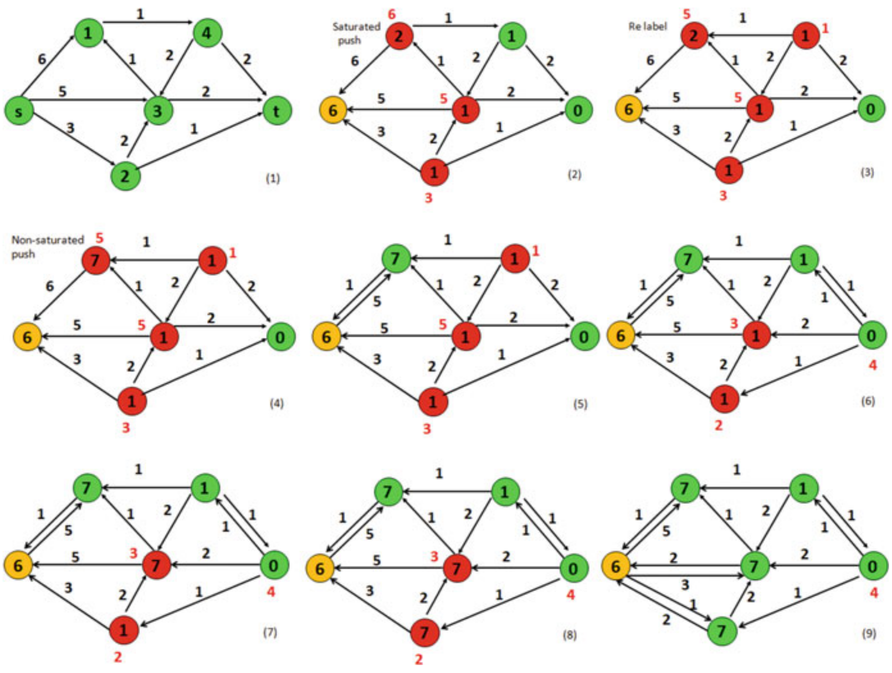

Fig. 5.16 An example for Goldberg-Tarjan algorithm (it contains three iterations from (5) to (6) and two iterations from (8) to (9))

hence, $d(v) \leq n - 1 + d(s) = 2n - 1$. Since each relabel makes a node's label increased at least once, a node can be relabeled at most $2n - 1$ times.

Let $deg(v)$ be the number of arcs at node v. Then each relabel spends time at most $deg(v)$. Therefore, all relabels need computational time at most

$$\sum_{v \in V} deg(v)(2n-1) \leq (2n-1) \cdot 2m = O(mn)$$

where $m = |E|$. □

Lemma 5.6.5 *There are at most $O(mn)$ saturated pushes in computation of Goldberg-Tarjan algorithm, where $n = |V|$ and $m = |E|$.*

Proof Note that a push must be on an arc (u, v) in G_f, and hence $(u, v) \in E$ or $(v, u) \in E$ where E is the set of arcs in input flow network G. Between two consecutive saturated pushes on an arc (u, v), there must exist a relabel for v. The total number of relabels for v is at most $2n$. Therefore, the total number of saturated pushes is at most $O(mn)$. □

Lemma 5.6.6 *There are at most $O(mn^2)$ non-saturated pushes in computation of Goldberg-Tarjan algorithm, where $n = |V|$ and $m = |E|$.*

Proof Consider a potential function $\Phi = \sum_{v:\text{active}} d(v)$. For simplicity of speaking, let us call some operation making a "deposit" if it decreases the value of Φ and "withdraw" if it increases the values of Φ.

First, note that each node has its label at most $2n - 1$. Therefore, the relabel can make total deposit at a node at most $2n - 1$. Hence, totally, the relabel can make deposit at most $2n^2 - n$.

Now, consider the saturated push. Each saturated push may increase a new active node, which results in a deposit at most $2n - 1$. Since there are totally $O(mn)$ saturated pushes, the saturated push can make totally $O(mn^2)$ deposit.

Finally, consider the non-saturated push. Each non-saturated push will remove an active node while increasing an active node. Suppose the push is on arc (u, v). Then $d(u) = d(v) + 1$. When u is removed and v may be added, the non-saturated push will make a withdraw at least one. Therefore, the number of non-saturated pushes is at most

$$n^2 + O(n^2) + O(mn^2) = O(m^2)$$

where note that initially, Φ has value at most n^2. □

Theorem 5.6.7 *Goldberg-Tarjan algorithm must terminate at a maximum flow within time $O(mn^2)$.*

Proof The algorithm must terminate after all pushes and relabels are done. Therefore, by Lemmas 5.6.4, 5.6.5, and 5.6.6, the algorithm will terminate within time

$$O(mn) + O(mn) + O(mn^2) = O(mn^2).$$

Moreover, when the algorithm terminates, there is no active node, and hence the preflow becomes a normal flow. Moreover, the label of s is still $d(s) = n$, which means that there is no path from s to t in the residual graph. Therefore, the flow is maximum. □

Note that in Goldberg-Tarjan algorithm, the selection of an active node is arbitrary. This gives an opportunity for improvement. There are two interesting rules for the selection of active node, which can improve the running time.

The First Rule (Excess Scaling) The algorithm is divided into phases, Δ-scaling phase for $\Delta = 2^{\lceil \log_2 C \rceil}, 2^{\lceil \log_2 C \rceil - 1}, \ldots, 1$ where $C = \max\{c(u, v) \mid (u, v) \in E\}$. At beginning of the Δ-scaling phase, $e(v) \le \Delta$ for every active node v. At the end of the Δ-scaling phase, $e(v) \le \Delta/2$ for every active node v. (When $\Delta = 1$, $\Delta/2$ is replaced by 0.) During the Δ-scaling phase, active node v is selected to be

$$v = \operatorname{argmin}\{d(u) \mid e(u) > \Delta/2\}.$$

In order to keep all active nodes with excess no more than Δ, a modification has to be made on flow amount in a push. Along an admissible arc (u, v), the flow of amount $\min(e(u), c(v, w), \Delta - e(w))$ is pushed.

Note that with the modification, there is no change on the relabel and the saturated push. Therefore, Lemmas 5.6.4 and 5.6.5 still hold. However, the non-saturated push occurs when pushed amount is either $e(v)$ or $\Delta - e(w)$. In either case, this amount is at least $\Delta/2$. In Fig. 5.17, an example is presented for computation in a Δ-scaling phase.

We next analyze the excess scaling algorithm, i.e., Goldberg-Tarjan algorithm with excess scaling rule.

Lemma 5.6.8 *In each Δ-scaling phase, the number of non-saturated pushes is at most $O(n^2)$.*

Proof Consider a potential function $\Phi = \sum_{v \in V} d(v) \cdot e(v)/\Delta$. For simplicity of speaking, let us call some operation making a "deposit" if it decreases the value of Φ and "withdraw" if it increases the values of Φ.

Note that $e(v)$ is nondecreasing during computation and $e(v) \le 2n$. Therefore, the relabel can deposit at most $2n$ at each node v. Hence, the relabel deposits totally at most $2n^2$ for Φ.

Every push will withdraw from Φ since it moves a certain amount value from node v to w with $d(v) = d(w) + 1$. Especially, every non-saturated push will move at least $\Delta/2$ from $e(v)$ to $e(w)$, that is, it withdraws at least $1/2$ from Φ. Thus, the total number of non-saturated pushes is at most $4n^2$ during each Δ-scaling phase. □

Theorem 5.6.9 *The excess scaling algorithm must terminate at a maximum flow within time $O(mn + n^2 \log C)$.*

Proof By Lemmas 5.6.4 and 5.6.5, the relabel and the saturated push use totally $O(mn)$ time. By Lemma 5.6.8, the non-saturated push spends totally $n^2 \log C$ time. At the end of algorithm, there is no active node, i.e., the preflow becomes a flow.

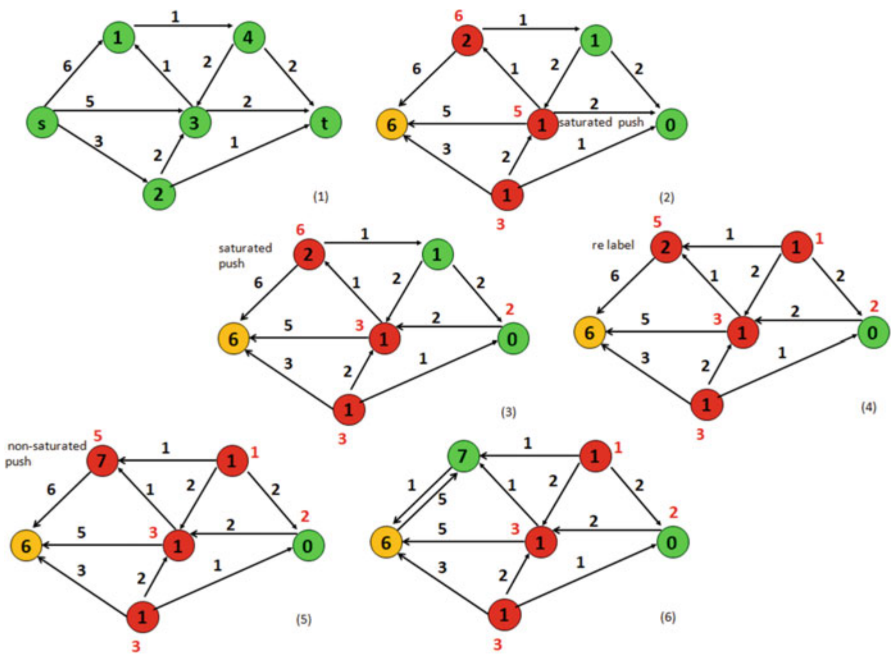

Fig. 5.17 An example for Δ-scaling phase ($\Delta = 8$)

Moreover, in its residual graph, there is no path from source s to sink t since $d(s) = n$. Thus, the flow is maximum. □

The Second Rule (Highest-Level Pushing) A *level* is subset of nodes with the same label. In this rule, the active node v is selected from the highest level, i.e.,

$$v = \text{argmax}\{d(u) \mid u \text{ is active}\}.$$

In Fig. 5.18, an example is presented for computation in Goldberg-Tarjan algorithm with this rule for selection of active node.

We next analyze Goldberg-Tarjan algorithm with highest-level pushing.

Let a *phase* be a consecutive sequence of pushes all at the same level.

Lemma 5.6.10 *There are totally at most $O(n^2)$ phases.*

Proof Let v be selected active node. A phase ends only if one of the following two cases occurs:

(a) All active nodes at level $d(v)$ have become inactive.
(b) A relabel for v occurs before v becomes inactive.

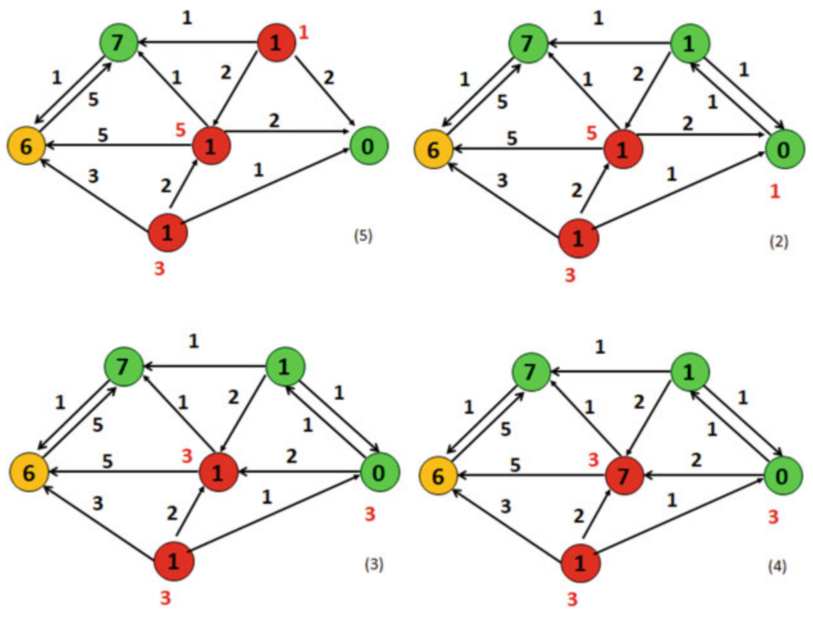

Fig. 5.18 An example for a phase of highest-level pushing

By Lemma 5.6.4, there are at most $O(n^2)$ relabels, and hence (b) occurs at most $O(n^2)$ times. Let $\Phi = \max\{d(v) \mid v \text{ is active}\}$. Then initially, $\Phi \leq n$. If (b) occurs, then Φ is increased. However, the total amount of increasing at each node is at most $2n$. Hence, Φ can be increased at most $2n^2$ by relabels. If (a) occurs, then Φ will be decreased by one. Therefore, (b) can occur at most $n + 2n^2 = O(n^2)$ times. Putting together, the number of phases is at most $O(n^2)$. □

Lemma 5.6.11 *There are at most $O(n^2m^{1/2})$ non-saturated pushes.*

Proof Let $k = \lceil m^{1/2} \rceil$. Call a phase as a *cheap one* if it contains at most k non-saturated pushes. Otherwise, call it as an *expensive phase*. Since there are at most $O(n^2)$ phases. The number of non-saturated pushes in all cheap phases is at most $O(n^2k) = O(n^2m^{1/2})$. In the following, we estimate the number of non-saturated pushes in all expensive phases.

To do so, define a potential function:

$$\Phi = \sum_{\text{active } v} z(v)$$

where $z(v)$ is the number of nodes each with label at most $d(v)$, i.e.,

$$z(v) = |\{u \in V \mid d(u) \leq d(v)\}|.$$

Again, for simplicity of speaking, let us call some operation making a "deposit" if it decreases the value of Φ and "withdraw" if it increases the values of Φ.

Each relabel makes an active node v have label increased, which will make $z(v)$ increased. However, the increase cannot exceed n, the total number of nodes. Therefore, each relabel makes a deposit with at most value n. $O(n^2)$ relabels will deposit with at most value $O(n^3)$.

Each saturated push may activate a node, which will deposit with value at most $2n$. Since there are totally $O(mn)$ saturated pushes, they can deposit with value at most $O(n^2)$.

Every non-saturated push on admissible arc (v, w) will make u inactive. Since $d(v) = d(w) + 1$, $z(v) - z(w)$ is equal to the number of nodes at the highest level in the phase containing the non-saturated push. Note that during a phase, an active node v at the level becomes inactive if and only if a non-saturated push occurs at active node v. Therefore, at beginning of an expensive phase, there must exist at least k active nodes at the highest level. This means that every non-saturated push will withdraw at least value k from Φ.

Summarized from above argument, we can conclude that the total number of non-saturated pushes in expensive phases is at most $O((n^3 + n^2m)/k) = O(n^2m^{1/2})$. Therefore, the total number of non-saturated pushes during whole computation is at most $O(n^2m^{1/2})$. □

Theorem 5.6.12 *The Goldberg-Tarjan algorithm with highest-level push must terminate at a maximum flow within time $O(n^2m^{1/2})$.*

Proof It follows immediately from Lemmas 5.6.4, 5.6.5, and 5.6.11. □

Exercises

1. A conference organizer wants to set up a review plan. There are m submitted papers and n reviewers. Each reviewer has made p papers as "prefer to review." Each paper should have at least q review reports. Find a method to determine whether such a review plan exists or not.
2. A conference organizer wants to set up a review plan. There are m submitted papers and n reviewers. Each reviewer is allowed to make at least p_1 papers as "prefer to review" and at least p_2 papers as "likely to review." Each paper should have at least q_1 review reports and at most q_2 review reports. Please give a procedure to make the review plan.
3. Let f be a flow of flow network G and f' a flow of residual network G_f. Show that $f + f'$ is a flow of G.
4. Let G be a flow network in which every arc capacity is a positive even integer. Show that its maximum flow value is an even integer.

5. Let G be a flow network in which every arc capacity is a positive odd integer. Can we conclude that its maximum flow value is an odd integer? If not, please give a counterexample.
6. Let G be a flow network. An arc (u, v) is said to be *critical* for a maximum flow f if $f(u, v) = c(u, v)$ where $c(u, v)$ is the capacity of (u, v). Show that an arc (u, v) is critical for every maximum flow if and only if decreasing it capacity by one will result in maximum flow value getting decreased by one.
7. Let A be an $m \times n$ matrix with non-negative real numbers such that for every row and every column, the sum of entries is an integer. Prove that there exists an $m \times n$ matrix B with non-negative integers and the same sums as in A, for every row and every column.
8. Suppose there exist two distinct maximum flows f_1 and f_2. Show that there exist infinitely many maximum flows.
9. Consider a directed graph G with a source s, a sink t and nonnegative arc capacities. Find a polynomial-time algorithm to determine whether G contains a unique s-t cut.
10. (This is an example on which Ford-Fulkerson algorithm runs with infinitely many augmentations.) Consider a flow network as shown in Fig. 5.19 where $x = \frac{\sqrt{5}-1}{2}$. Show by induction on k that the residual capacity $c(u, v) - f(u, v)$ on three vertical arcs can be $x^k, 0, x^{k+1}$ for every $k = 0, 1, 2, \ldots$. (Hint: The case of $k = 0$ is shown in Fig. 5.20. The induction step is as shown in Fig. 5.21.)
11. Consider a flow network $G = (V, E)$ with a source s, a sink t, and nonnegative capacities. Suppose a maximum flow f is given. If an arc is broken, find a fast algorithm to compute a new maximum flow based on f. A favorite algorithm will run in $O(|E| \log |V|)$ time.
12. Consider a flow network $G = (V, E)$ with a source s, a sink t, and nonnegative integer capacities. Suppose a maximum flow f is given. If the capacity of an

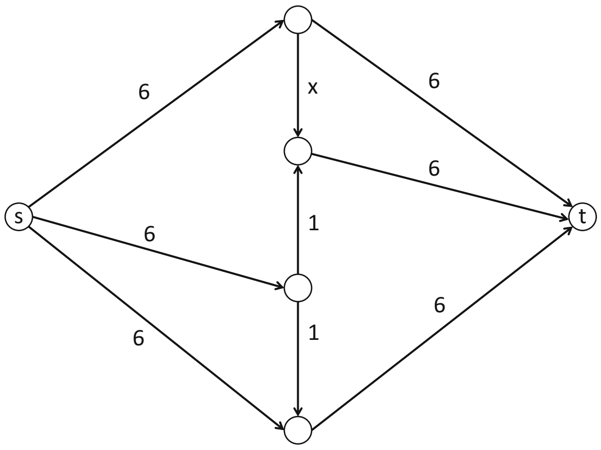

Fig. 5.19 An example for Ford-Fulkerson algorithm

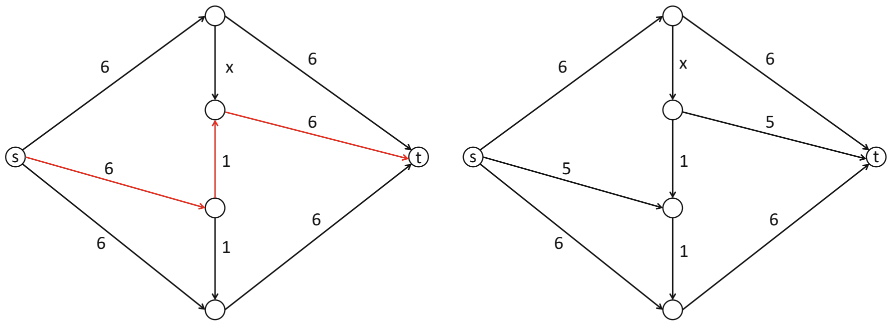

Fig. 5.20 Base step

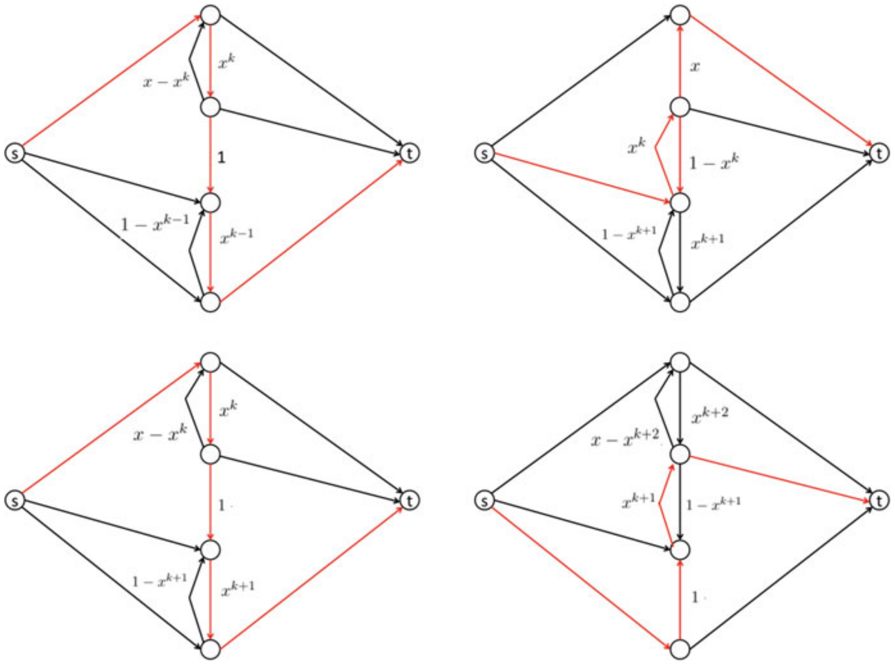

Fig. 5.21 Induction step

arc is increased by one, find a fast algorithm to update the maximum flow. A favorite algorithm runs in $O(|E| + |V|)$ time.

13. Consider a directed graph $G = (V, E)$ with a source s and a sink t. Instead of arc capacity, assume that there is the nonnegative integer node capacity $c(v)$ on each node $v \in V$, that is, the total flow passing node v cannot exceed $c(v)$. Show that the maximum flow can be computed in polynomial-time.
14. Show that the maximum flow of a flow network $G = (V, E)$ can be decomposed into at most $|E|$ path-flows.

15. Let $G = (V, E)$ be a undirected connected graph with two distinct nodes s and t. Find two disjoint node subsets S and T such that $s \in S$ and $t \in T$, to minimize $\delta(S) + \delta(T)$ where $\delta(X)$ denotes the number of edges between X and $V \setminus X$.
16. Suppose a flow network $G = (V, E)$ is symmetric, i.e., $(u, v) \in E$ if and only if $(v, u) \in E$ and $c(u, v) = c(v, u)$. Show that Edmonds-Karp algorithm terminates within at most $|V| \cdot |E|/4$ iterations.
17. Consider a directed graph G. A node-disjoint set of cycles is called a *cycle-cover* if it covers all nodes. Find a polynomial-time algorithm to determine whether a given graph G has a cycle-cover or not.
18. Consider a graph G. Given two nodes s and t, and a positive integer k, find a polynomial-time algorithm to determine whether there exist or not k edge-disjoint paths between s and t.
19. Consider a graph G. Given two nodes s and t and a positive integer k, find a polynomial-time algorithm to determine whether there exist or not k node-disjoint paths between s and t.
20. Consider a graph G. Given three nodes x, y, and z, find a polynomial-time algorithm to determine whether there exists a simple path from x to z passing through y.
21. Prove or disprove (by counterexample) the following statements.

    (a) If a flow network has unique maximum flow, then it has a unique minimum s-t cut.
    (b) If a flow network has unique minimum s-t cut, then it has a unique maximum flow.
    (c) A maximum flow must associate with a minimum s-t cut such that the flow passes through the minimum s-t cut.
    (d) A minimum s-t cut must associate with a maximum flow such that the flow passes through the minimum s-t cut.

22. Let M be a maximal matching of a graph G. Show that for any matching M' of G, $|M'| \leq 2 \cdot |M|$.
23. We say that a bipartite graph $G = (L, R, E)$ is d-regular if every vertex $v \in L \cup R$ has degree exactly d. Prove that every d-regular bipartite graph has a matching of size $|L|$.
24. There are n students who studied at a late-night study for final exam. The time has come to order pizzas. Each student has his own list of required toppings (e.g., mushroom, pepperoni, onions, garlic, sausage, etc.). Everyone wants to eat at least half a pizza, and the topping of that pizza must be in his required list. A pizza may have only one topping. How to compute the minimum number of pizzas to order to make everyone happy?
25. Consider bipartite graph $G = (U, V, E)$. Let $\mathcal{H}$ be the collection of all subgraphs H that for every $u \in U$, H has at most one edge incident to u. Let $E(H)$ denote the edge set of H and $\mathcal{I} = \{E(H) \mid H \in \mathcal{H}\}$. Show that (a) $(E, \mathcal{I})$ is a matroid and (b) all matchings in G form an intersection of two matroids.

26. Consider a graph $G = (V, E)$ with nonnegative integer function $c : V \to N$. Find an augmenting path method to compute a subgraph $H = (V, F)$ $(F \subseteq E)$ with maximum number of edges such that for every $v \in V$, $deg(v) \leq c(v)$.
27. A conference with a program committee of 30 members received 100 papers. Before making an assignment, the PC-chair first asked all PC-members each to choose 15 preferred papers. Based on what PC-members choose, the PC-chair wants to find an assignment such that each PC-member reviews 10 papers among 15 chosen ones and each paper gets 3 PC-members to review. How do we figure out whether such an assignment exists? Please design a maximum flow formulation to answer this question.
28. Let $U = \{u_1, u_2, \ldots, u_n\}$ and $V = \{v_1, v_2, \ldots, v_n\}$. A bipartite graph $G = (U, V, E)$ is *convex* if $(u_i, v_k), (u_j, v_k) \in E$ with $i < j$ imply $(u_h, v_k) \in E$ for all $h = i, i + 1, \ldots, j$. Find a greedy algorithm to compute the maximum matching in a convex bipartite graph.
29. Consider a bipartite graph $G = (U, V, E)$ and two node subsets $A \subseteq U$ and $B \subseteq V$. Show that if there exist a matching M_A covering A and a matching M_B covering B, then there exists a matching $M_{A\cup B}$ covering $A \cup B$.
30. For a graph G, let $odd(G)$ denote the number of connected components of odd size in G. Prove the following.

    (a) In any graph $G = (V, E)$, the minimum number of free nodes in any matching is

    $$\max_{U\subseteq V}(odd(G \setminus U) - |U|).$$

    (b) In any graph G, the maximum size of a matching is

    $$\min_{U\subseteq V} \frac{1}{2} \cdot (|V| + |U| - odd(G \setminus U)).$$

    (c) A graph $G = (V, E)$ has a perfect matching if and only if for any $U \subseteq V$, $odd(G \setminus U) \leq |U|$.

Historical Notes

Maximum flow problem was proposed by T. E. Harris and F. S. Ross in 1955 [204, 360] and was first solved by L.R. Ford and D.R. Fulkerson in 1956 [145]. However, Ford-Fulkerson algorithm is a pseudo polynomial-time algorithm when all arc capacities are integers. If arc capacities may not be integers, the termination of the algorithm may meet a trouble. The first strong polynomial-time algorithm was designed by Edmonds and Karp [123]. Later, various designs appeared in the literature, including Dinitz algorithm [88, 89], Goldberg-Tarjan push-relabel algorithm [178], Goldberg-Rao algorithm [175], Sherman algorithm [365], and the algorithm of Kelner, Lee, Orecchia, and Sidford [240]. Currently, the best running time is $O(|V||E|)$. This record is kept by Orlin algorithm [331] for approximation solution, running time can be further improved [239].

Matching is a classical subject in graph theory. Both maximum (cardinality) matching and minimum cost perfect matching problems in bipartite graphs can be easily transformed to maximum flow problems. However, they can also be solved with alternating path methods. So far, Hopcroft-Karp algorithm [215] is the fastest algorithm for the maximum bipartite matching. In general graph, they have to be solved with alternating path method since currently, no reduction has been found to transform matching problem to flow problem. Those algorithms were designed by Edmonds [118]. An extension of Hopcroft-Karp algorithm was made by Micali and Vazirani [313], which runs in $O(\sqrt{|E|}|V|)$ time.

For maximum weight matching, nobody has found any method to transform it to a flow problem. Therefore, we have to employ the alternating path and cycle method [118], too.

Chinese postman problem was proposed by Kwan [269], and the first polynomial-time algorithm was given by Edmonds and Johnson [122] with minimum cost perfect matching in complete graph with even number of nodes.