AI2615 算法设计与分析 2020-2021 春季学期

Lecture 12 – Network Flow

2021 年 5 月 14 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

In Fig. 1, suppose we would like to build a data transmission channel from s to t, by using the intermediate routers a, b, c, d, e. Each edge in the graph has a bandwidth, which gives a capacity constraint indicating the maximum rate of data that can be transferred. What is the maximum rate of data that can be transferred from s to t? This is the *maximum flow problem*.

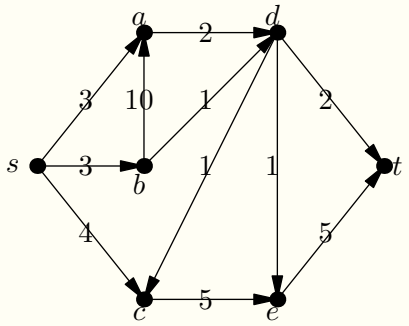

Figure 1: A flow network.

1 Maximum Flow Problem

Definition 1. Given a directed graph $G = (V, E)$ with a source s and a sink t, and a capacity $c_e > 0$ assigned to each edge e, a *flow* f is a map $f : E \to \mathbb{R}_{\geq 0}$ satisfying the followings (for $e = (u, v)$, we denote $f(e)$ by f_e or f_{uv}):

- **Capacity Constraint**: for each $e \in E$, $f_e \leq c_e$, and
- **Flow Conservation**: for each $u \in V \setminus \{s, t\}$, $\sum_{v:(v,u)\in E} f_{vu} = \sum_{w:(u,w)\in E} f_{uw}$.

The *value* of a flow f, denoted by $v(f)$, is defined as $v(f) = \sum_{v:(s,v)\in E} f_{sv}$.

Problem 2 (Maximum Flow). Given a directed graph $G = (V, E)$ with a source s, a sink t, and capacity constraints $c : E \to \mathbb{R}^+$, find a flow f with the maximum value.

Let us consider a simpler example in Fig. 2. The source s has two outgoing edges: (s, u) and (s, v). Since $c_{su} = 20$ and $c_{sv} = 10$, the total amount of flow we can push from s to t is at most 30. Therefore, we have $v(f) \leq 30$ for any flow f. In fact, there exists a flow f with value exactly 30: $f_{su} = 20, f_{sv} = 10, f_{uv} = 10, f_{ut} = 10$ and $f_{vt} = 20$. We have seen that f is a maximum flow.

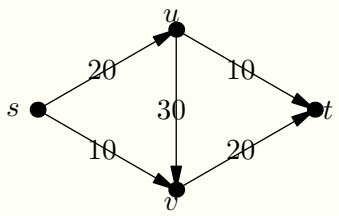

Figure 2: A flow network.

2 Ford-Fulkerson Algorithm

The maximum flow problem can be formulated by a linear program.

$$\begin{aligned} \max \quad & \sum_{u:(s,u)\in E} f_{su} \\ \text{subject to} \quad & \forall (u,v) \in E : 0 \le f_{uv} \le c_{uv} \\ & \forall u \in V \setminus \{s,t\} : \sum_{v:(v,u)\in E} f_{vu} = \sum_{w:(u,w)\in E} f_{uw} \end{aligned} \tag{1}$$

Due to the nature of the maximum flow problem, the simplex method for this linear program has a combinatorial interpretation. Suppose we start from the origin vertex $(f_{su}, f_{sv}, f_{uv}, f_{ut}, f_{vt}) = (0,0,0,0,0)$ of the simplex describing the feasible region. If we find a s-t path and assign a flow on this path that meets the capacity constraints (i.e., for each edge e on this path p, we assign $f_e = c$ where $c = \min_{e\in p} c_e$) and update f, we move from one vertex of the feasible region to another such that the objective, which is the value of the flow in this case, increases. For example, if we use the s-t path $s \to v \to t$, the maximum amount of flow we can push on this path is 10, and we update f to $(f_{su}, f_{sv}, f_{uv}, f_{ut}, f_{vt}) = (0,10,0,0,10)$. We have left the two hyper-planes defined by $f_{sv} = 0$ and $f_{vt} = 0$ respectively, and we have reached the hyper-plane $f_{sv} = 10$. In the simplex describing the feasible region, we move from one vertex to another such that the objective $\sum_{u:(s,u)\in E} f_{su}$ increases. It is clear that the objective increases. We will not formally prove the fact that we have reached another vertex.

We can perform this operation iteratively, until there is no more s-t path on which we can push a positive amount of flow, equivalently, until every s-t path contains an edge e such that $f_e = c_e$. In our example, we can push a flow of amount 10 on the path $s \to u \to t$ in the next iteration, and then push a flow of amount 10 on the path $s \to u \to v \to t$ in the following iteration. We reach $(f_{su}, f_{sv}, f_{uv}, f_{ut}, f_{vt}) = (20,10,10,10,20)$, which is maximum. See Fig. 3 for an illustration of these.

However, this operation along cannot always guarantee a maximum flow. Using the example in Fig. 2 again, if the first path we consider is $s \to u \to v \to t$, the maximum amount of flow on this path is 20. After we have reached $(f_{su}, f_{sv}, f_{uv}, f_{ut}, f_{vt}) = (20,0,20,0,20)$, we can no longer find another s-t path such that a positive amount of flow can be pushed. The value of f for now is 20, which is sub-optimal. We need another operation to continue improving f.

After we have reached $(f_{su}, f_{sv}, f_{uv}, f_{ut}, f_{vt}) = (20,0,20,0,20)$, we can push 10 units of flow on the path $s \to$

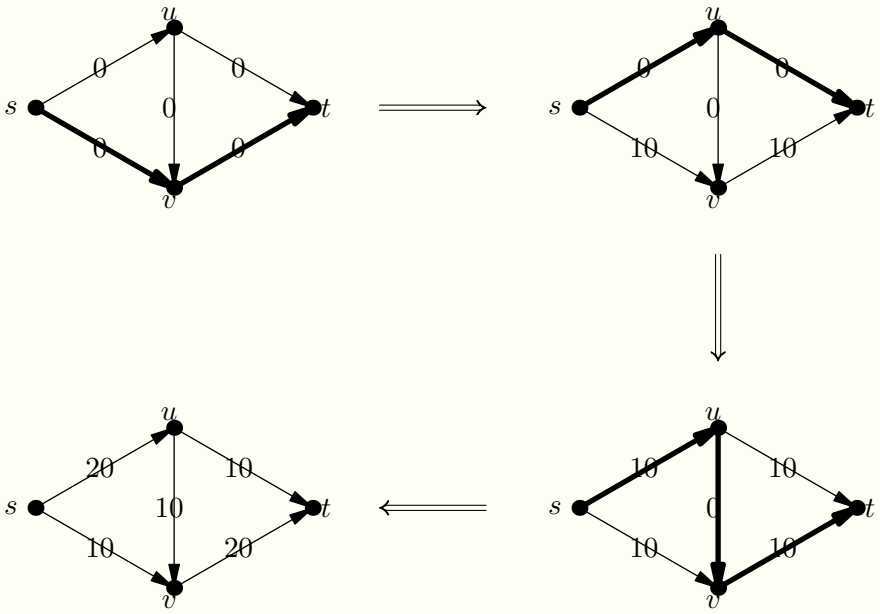

Figure 3: Iteratively find a s-t path and push a maximum amount of flow on the path.

$v \to u \to t$. Although $(v, u) \notin E$, we can achieve this by decrease the amount of flow on (u, v) from 20 to 10. In particular, the flow conservation constraint on vertex v still holds: we have increased the amount of flow on (s, v) by 10 units and decreased the amount of flow on (u, v) by 10 units. See Fig. 4 for an illustration. After this operation, we reach the maximum flow f with $(f_{su}, f_{sv}, f_{uv}, f_{ut}, f_{vt}) = (20, 10, 10, 10, 20)$.

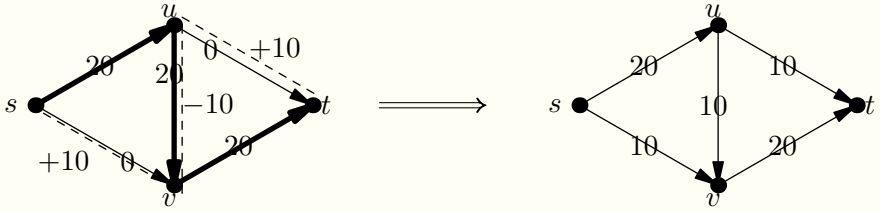

Figure 4: Push 10 units of flow on the path $s \to v \to u \to t$.

To formally discuss this operation, we define the *residual network* with respect to a flow f.

Definition 3. Given a directed graph $G = (V, E)$ with a source s, a sink t, capacity constraints $c : E \to \mathbb{R}^+$, and a flow f, the *residual network* with respect to f, denoted by $G^f = (V, E^f)$, shares the same vertex set V, and the edge set E^f is defined such that $(u, v) \in E^f$ if one of the followings holds:

1. $(u, v) \in E$ and $f_{uv} < c_{uv}$. In this case, the capacity of (u, v) is defined as $c_{uv} - f_{uv}$.
2. $(v, u) \in E$ and $f_{vu} > 0$. In this case, the capacity of (u, v) is defined as f_{vu}.

For Case 1 with $f_{uv} < c_{uv}$, we are still allowed to push extra $c_{uv} - f_{uv}$ units of flow on the edge (u, v). For Case 2 with $f_{vu} > 0$, we are allowed to push f_{vu} units of flow in the reverse direction (u, v); equivalently,

we can reduce the amount of flow on (v, u) by f_{vu} units.
An example of the residual network for the network in Fig. 2 is shown in Fig. 5.

Figure 5: A flow f with $(f_{su}, f_{sv}, f_{uv}, f_{ut}, f_{vt}) = (20, 0, 20, 0, 20)$ (left-hand side) and the residual network with respect to f (right-hand side).

For each of the two operations we discussed earlier, it corresponds to a path on the residual network G^f. We will call a path in the residual network an *augmenting path*. It turns out that the algorithm that iteratively finds an augmenting path and pushes a maximum amount of flow on the path always finds a maximum flow. This is precisely *Ford-Fulkerson Algorithm* (see Algorithm 1).

Algorithm 1 Ford-Fulkerson Algorithm

FORD-FULKERSON$(G = (V, E), s, t, c)$

1: initialize f such that $f_e = 0$ for each $e \in E$
2: **while** there exists an s-t path p (an augmenting path) in G^f:
3: AUGMENT(f, p) // The subroutine AUGMENT is defined in Algorithm 2
4: **endwhile**
5: **return** f

Algorithm 2 Subroutine AUGMENT

AUGMENT(f, p)

1: find an edge on the path p with minimum capacity b (the capacity is the one defined in G^f)
2: **for** each $e = (u, v) \in p$:
3: **if** $(u, v) \in E$: update $f(e) \leftarrow f(e) + b$
4: **if** $(v, u) \in E$: update $f(e) \leftarrow f(e) - b$
5: **endfor**

To analyze the time complexity for Ford-Fulkerson Algorithm, it is easy to see that each while-loop execution requires $O(m)$ time: it requires $O(m)$ time to find an augmenting path, to perform AUGMENT(f, p), and to update the residual network G^f. Suppose all the capacities are integers. We can push 1 unit of flow on each augmenting path at minimum. Let $C = \sum_{u:(s,u)\in E} c_{su}$. Then C is an upper bound on the value of the maximum flow. We needs to execute the while-loop for at most C times. Therefore, the time complexity for Ford-Fulkerson Algorithm is $O(Cm)$. In fact, it can be shown by providing tight examples that

this time complexity bound is tight. This implies that Ford-Fulkerson Algorithm does not always run in a polynomial time, as C may not be bounded by a polynomial of the input size. In Sect. 4, we will learn a polynomial time algorithm, *Edmonds-Karp Algorithm*, which adapts Ford-Fulkerson Algorithm by careful selections of augmenting paths.
In the next section, we will prove the correctness of Ford-Fulkerson Algorithm, based on an important and useful theorem—*the max-flow min-cut theorem.*

3 Max-Flow Min-Cut Theorem

Similar to what we have done in the strong duality theorem, we aim to find a tight upper bound for the value of the maximum flow. Clearly, $C = \sum_{u:(s,u)\in E} c_{su}$ is an upper bound, but it is not tight.

Definition 4. Given a directed weighted graph $G = (V, E, w)$, a source s and a sink t, a *cut* is a partition of V to two vertex sets L and R such that $s \in L$ and $t \in R$. The *value* of the cut is defined as

$$c(L, R) = \sum_{(u,v)\in E, u\in L, v\in R} w(u, v).$$

If viewing the capacity of an edge as the weight of it, the value of a minimum cut is a tight upper bound to the value of a maximum flow. This is the *Max-Flow Min-Cut Theorem*

Theorem 5 (Max-Flow Min-Cut). *Given a directed graph $G = (V, E)$ with a source s, a sink t and capacity constraints $c : E \to \mathbb{R}^+$, letting f be a maximum flow and $\{L, R\}$ be a cut with minimum value, we have $v(f) = c(L, R)$.*

We first show that the value of any cut is an upper bound to the value of any flow.

Lemma 6. *Given a directed graph $G = (V, E)$ with a source s, a sink t and capacity constraints $c : E \to \mathbb{R}^+$, we have $v(f) \le c(L, R)$ for any flow f and any cut $\{L, R\}$.*

Let

$$f(L, R) = \sum_{(u,v)\in E, u\in L, v\in R} f_{uv}$$

be the total amount of flow going from L to R, and let

$$f(R, L) = \sum_{(u,v)\in E, u\in R, v\in L} f_{uv}$$

be the total amount of flow going from R to L. Notice that L contains s and R contains t. The following proposition is intuitive: it says that the total amount of flow going from s to t is the total amount of flow leaving L minus the total amount of flow entering L.

Proposition 7. $v(f) = f(L, R) - f(R, L)$.

Proof. For a vertex u, let $f^{out}(u) = \sum_{w:(u,w)\in E} f_{uw}$ be the total amount of flow leaving u, and let $f^{in}(u) = \sum_{v:(v,u)\in E} f_{vu}$ be the total amount of flow entering u. By the flow conservation constraint, we have $f^{in}(u) = f^{out}(u)$ for each $u \in V \setminus \{s,t\}$, $f^{in}(s) = 0$ and $f^{out}(s) = v(f)$. Summing up these for all vertices in L, we have

$$\sum_{u\in L}\left(f^{out}(u) - f^{in}(u)\right) = f^{out}(s) + \sum_{u\in L\setminus\{s\}} 0 = v(f). \tag{2}$$

On the other hand, consider each edge (u,v). If $u,v \in L$, f_{uv} is counted once in each of $f^{out}(u)$ and $f^{in}(v)$, which is canceled in the summation $\sum_{u\in L}\left(f^{out}(u) - f^{in}(u)\right)$. If $u,v \in R$, it will not be counted in this summation. If $u \in L$ and $v \in R$, f_{uv} is counted once in $f^{out}(u)$. If $u \in R$ and $v \in L$, f_{uv} is counted once in $f^{in}(v)$. Putting these together, we have

$$\sum_{u\in L}\left(f^{out}(u) - f^{in}(u)\right) = \sum_{(u,v)\in E, u\in L, v\in R} f_{uv} - \sum_{(u,v)\in E, u\in R, v\in L} f_{uv} = f(L,R) - f(R,L). \tag{3}$$

Equations (2) and (3) imply this proposition. □

With this proposition, Lemma 6 follows immediately.

Proof of Lemma 6. By Proposition 7, we have

$$v(f) \le f(L,R) = \sum_{(u,v)\in E, u\in L, v\in R} f_{uv} \le \sum_{(u,v)\in E, u\in L, v\in R} c_{uv} = c(L,R).$$

□

Finally, to prove Theorem 5, we show that there exists a cut $\{L,R\}$ such that the flow f output by Ford-Fulkerson Algorithm satisfies $v(f) = c(L,R)$. This proves not only Theorem 5, but also the correctness of Ford-Fulkerson Algorithm.

Proposition 8. *There exists a cut $\{L,R\}$ such that the flow f output by Ford-Fulkerson Algorithm satisfies $v(f) = c(L,R)$.*

Proof. Let f be the flow output by Ford-Fulkerson Algorithm. We will construct L and R such that $v(f) = c(L,R)$. Let G^f be the residual network with respect to f. Let L be the set of vertices reachable from s in G^f, and let $R = V \setminus L$. To show $v(f) = c(L,R)$, we will show that $f(L,R) = c(L,R)$ and $f(R,L) = 0$, then Proposition 7 implies this proposition.

To show that $f(L,R) = c(L,R)$, it suffices to show that, for any edge (u,v) such that $u \in L$ and $v \in R$, we have $f_{uv} = c_{uv}$. Indeed, if $f_{uv} < c_{uv}$, then (u,v) should still be in G^f by our definition of residual network. This means v is also reachable from s (since u is reachable from s), which contradicts to $v \in R$.

To show that $f(R,L) = 0$, it suffices to show that, for any edge (u,v) such that $u \in R$ and $v \in L$, we have $f_{uv} = 0$. Indeed, if $f_{uv} > 0$, then (v,u) should still be in G^f by our definition of residual network. This means u is also reachable from s (since v is reachable from s), which contradicts to $u \in R$. □

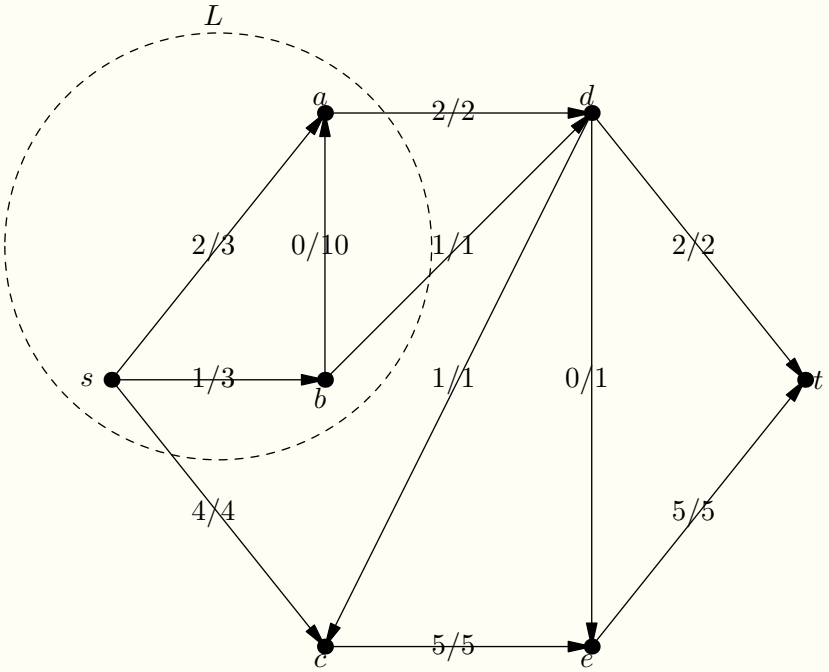

Figure 6: An illustration of the max-flow min-cut theorem. On each edge (u, v), the first number represents f_{uv} and the second number represents c_{uv}. The vertices in L is circled, and the remaining vertices are in R. The value of this flow is $f_{sa} + f_{sb} + f_{sc} = 7$ and the value of the cut is $c_{ad} + c_{bd} + c_{sc} = 7$. This flow has the maximum value, and this cut has the minimum value.

Using the graph in Fig. 1 as an example, Fig. 6 illustrates the max-flow min-cut theorem.
The max-flow min-cut theorem also suggests that the *minimum cut problem* can be solved in polynomial time.

Problem 9. Given a directed positively-weighted graph $G = (V, E, w)$, a source s and a sink t, find a cut $\{L, R\}$ (such that $s \in L$ and $t \in R$) with the minimum value.

To solve this problem, all we need to do is to find a maximum flow (assuming $w(u, v)$ is the capacity c_{uv} of (u, v)), construct the residual network G^f, and then identify all vertices that are reachable from s in G^f.

4 Edmonds-Karp Algorithm

We have mentioned in Sect. 2 that Ford-Fulkerson Algorithm's running time depends on the numerical values of the capacities, which makes it fails to run in polynomial time. Edmonds-Karp Algorithm modifies Ford-Fulkerson Algorithm as follows: at Line 2 of Algorithm 1, we use the s-t path p that contains a minimum number of edges in the residual network G^f. Notice that such a path can be found by a breadth-first search starting from s.

In this section, we will show that Edmonds-Karp Algorithm runs in time $O(m^2 n)$.

Definition 10. Given a residual network G^f and an augmenting path p, an edge $(u, v) \in p$ is *critical* if f_{uv} equals to the capacity of (u, v) in G^f after the operation $\textsc{Augment}(f, p)$.

By the natural of the algorithm, at least one edge becomes critical.

Proposition 11. *Given a residual network G^f and an augmenting path p, let f' be the flow after applying $\textsc{Augment}(f, p)$. If an edge (u, v) in G^f become critical, then (u, v) is not in $G^{f'}$, and (v, u) must be in $G^{f'}$.*

Proof. Firstly, either (u, v) or (v, u) must be in the original graph. If (u, v) is in the original graph, (u, v) becoming critical implies $f_{uv} = c_{uv}$; in this case we have $(u, v) \notin G^{f'}$ and $(v, u) \in G^{f'}$. If (v, u) is in the original graph, (u, v) becoming critical implies $f_{vu} = 0$; in this case we have $(u, v) \notin G^{f'}$ and $(v, u) \in G^{f'}$ as well. □

Let $\delta_f(v)$ be the distance from s to v in the residual network G^f, where the distance is defined as the minimum number of edges connecting from s to v (not the distance in the edge-weighted sense). The following proposition says that the distance of each vertex is non-decreasing throughout the Edmonds-Karp Algorithm.

Proposition 12. *Given a residual network G^f and an augmenting path p, let f' be the flow after applying $\textsc{Augment}(f, p)$. We have $\delta_f(v) \leq \delta_{f'}(v)$ for each vertex v.*

Proof. To show that the distance from s to any v cannot decrease, we show that, for any edge (v, u) that appears in $G^{f'}$ but not in G^f, we have $\delta_f(v) > \delta_f(u)$. This implies the proposition: if the addition of (v, u) connects a vertex v at a larger distance to a vertex u at a smaller distance, this addition does not help to reduce the distance of u.
Notice that if an edge is not critical, this edge remains in the residual graph, and it will not cause any additions of edges. The only case where (v, u) is added is when $(u, v) \in p$ becomes critical, implied by Proposition 11. However, Edmonds-Karp Algorithm finds p by a breadth-first search, so $\delta_f(v) = \delta_f(u) + 1$. Therefore, for any edge (v, u) added, we have $\delta_f(v) > \delta_f(u)$. □

Now, we are ready to evaluate the time complexity for Edmonds-Karp Algorithm.
Suppose (u, v) in G^f becomes critical after $\textsc{Augment}(f, p)$, and let f' be the flow after the augmentation. By the breadth-first search natural, we have $\delta_f(v) = \delta_f(u) + 1$. We know that (u, v) is no longer in $G^{f'}$ and (v, u) is in $G^{f'}$ by Proposition 11. The connection between u and v will remain unchanged until (v, u) becomes critical at a future iteration. Suppose at a future iteration (v, u) becomes critical in $G^{f''}$. We have,

by the breadth-first search natural, $\delta_{f''}(u) = \delta_{f''}(v) + 1$. Moreover,

$$\begin{aligned}
\delta_{f''}(u) &= \delta_{f''}(v) + 1 \\
&\geq \delta_f(v) + 1 \quad \text{(Proposition 12 implies the distance of } v \text{ never decrease throughout the algorithm)} \\
&= \delta_f(u) + 2. \qquad \text{(since } \delta_f(v) = \delta_f(u) + 1\text{)}
\end{aligned}$$

Therefore, for any edge (u, v) in the original graph, if (u, v) or (v, u) becomes critical and this is not the first time, the distance for one of u or v increases by 2. Since the distance of any vertex can increase by 2 for at most $n/2$ times before becoming ∞, each edge or its reverse can only become critical for at most $n + 1$ times. Since each while-loop iteration makes at least one edge critical, the total number of the while-loop iterations is at most mn. Since each while-loop iteration requires $O(m)$ time, the overall time complexity for Edmonds-Karp Algorithm is $O(m^2 n)$.

5 Maximum Matching and Minimum Vertex Cover in Bipartite Graphs

5.1 Maximum Matching

Definition 13. Given an undirected graph $G = (V, E)$, a *matching* is a set of edges $M \subseteq E$ that do not share any vertices in common: for any $e_1, e_2 \in M$, we have $e_1 \cap e_2 = \emptyset$.

Problem 14 (Maximum Matching in Bipartite Graphs). Given a bipartite graph $G = (A, B, E)$, find a maximum matching (a matching with maximum number of edges).

Figure 7 gives an example of a maximum matching in a bipartite graph.

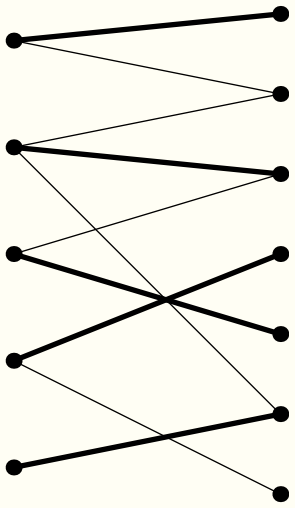

Figure 7: A maximum matching (bold edges) in a bipartite graph.

The maximum matching problem in bipartite graphs can be solved by a reduction to the maximum flow problem. Given a bipartite graph $G = (A, B, E)$, we construct a maximum flow instance $(G' = (V', E'), s, t, c)$

as follows. Create a vertex s and a vertex t. Then V' contains all the vertices in G, which is $A \cup B$, plus the two vertices s and t. Connect s to all the vertices in A, and assign capacity 1 to each of those edges. Connect all the vertices in B to t, and assign capacity 1 to each of those edges. For each edge $\{u, v\} \in E$ in G with $u \in A$ and $v \in B$, create a (directed) edge $(u, v) \in E'$ with capacity ∞ (or a very large number, say, $(|A| + |B|)^{10}$). Figure 8 shows the maximum flow instance constructed for the graph in Fig. 7.

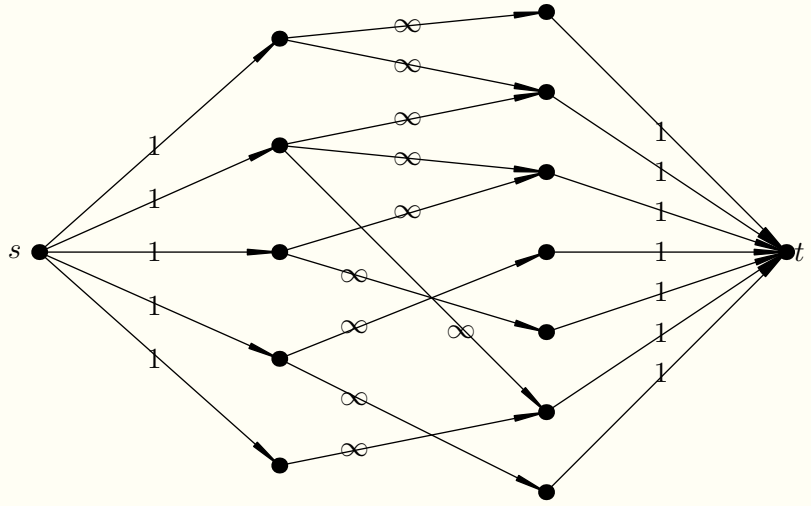

Figure 8: The maximum flow instance constructed for the graph in Fig. 7.

Now, we run Ford-Fulkerson Algorithm, or Edmonds-Karp Algorithm, on the maximum flow instance we have just constructed. Here, we exploit a property of the algorithm.

Theorem 15. *If the capacities are all integers, Ford-Fulkerson Algorithm or Edmonds-Karp Algorithm always returns an integral flow, i.e., a flow f such that $f_{uv} \in \mathbb{Z}_{\geq 0}$ for each edge (u, v).*

Exercise 16. Prove Theorem 15.

Let f be the integral flow returned by the algorithm. It is then easy to see that, for each edge e, f_e is either 0 or 1 in the graph we have constructed. For each $\{u, v\}$ in the original graph G (with $u \in A$ and $v \in B$), we select it in the matching if and only if $f_{uv} = 1$ in G'. It is clear that we obtain a matching by doing this. For each $u \in A$, since the amount of flow that goes into u is at most 1 (the only incoming edge for u is (s, u)), the amount of flow that goes out from u is at most 1 by the conservation of flow. This means that, in the original graph G, at most one edge that are incident to u is selected. Similarly, for each $v \in B$, since the amount of flow that goes out from v is at most 1 (the only outgoing edge for v is (v, t)), the amount of flow that goes into v is at most 1. This means that, in the original graph G, at most one edge that are incident to v is selected. This proves that the edges we have selected form a matching.
On the other hand, for each valid matching M, we can construct a flow f with value $v(f) = |M|$. Indeed, f is constructed as follows.

- For each $u \in A$, $f_{su} = 1$ if there exists $v \in B$ such that $\{u, v\} \in M$, and $f_{su} = 0$ if otherwise.
- For each $\{u, v\} \in E$ with $u \in A$ and $v \in B$, $f_{uv} = 1$ if $\{u, v\} \in M$, and $f_{uv} = 0$ if otherwise.
- For each $v \in B$, $f_{vt} = 1$ if there exists $u \in A$ such that $\{u, v\} \in M$, and $f_{vt} = 0$ if otherwise.

Figure 9 shows the flow corresponding to the matching in Fig. 7.

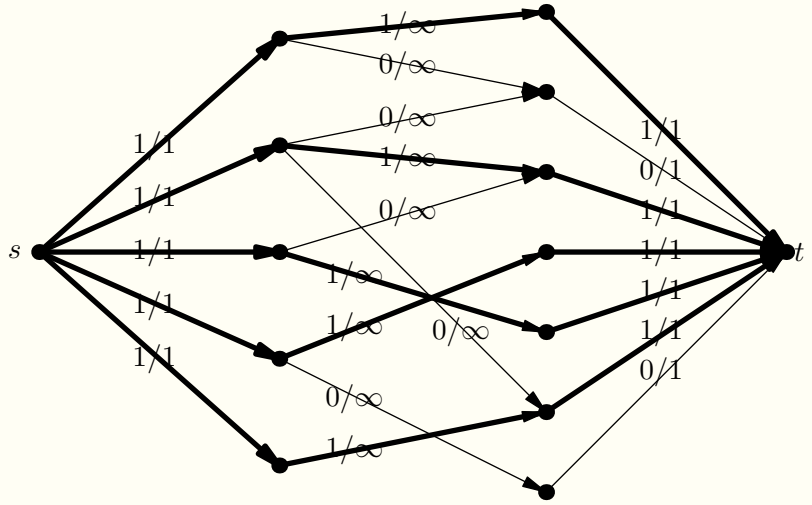

Figure 9: The flow corresponding to the matching in Fig. 7.

We have shown that the flow f found by the algorithm is integral and corresponds to a matching M such that $v(f) = |M|$. We have also proved that any matching M corresponds to a flow f with $v(f) = |M|$. Since the algorithm finds a flow with the maximum value, the matching found is also maximum.

5.2 Minimum Vertex Cover and Maximum Independent Set

Given a bipartite graph $G = (A, B, E)$, suppose we have found an integral maximum flow f in the graph G' (constructed in the previous sub-section). By viewing the capacities as the weights of the edges, the minimum cut $\{L, R\}$ also has value $v(f)$, by the max-flow min-cut theorem. Let $V_1 = L \cap A$, $V_2 = L \cap B$, $V_3 = R \cap A$ and $V_4 = R \cap B$. We have $v(f) = c(L, R) = |V_2| + |V_3|$, since the cut separates V_2 from t and separates V_3 from s. In addition, there does not exist any edge (u, v) such that $u \in V_1$ and $v \in V_4$, for otherwise the cut has value ∞ (while we know the cut has value $v(f) < \infty$). See Fig. 10 for some intuitions of these observations.

This implies that $V_1 \cup V_4$ is an independent set in G. As a result, $V_2 \cup V_3$ is a vertex cover in G (see Exercise 4 in Lecture 11). Since every cut $\{L, R\}$ defines the vertex cover $V_2 \cup V_3$ such that $c(L, R) = |V_2| + |V_3|$, and the cut $\{L, R\}$ is minimum, the vertex cover $V_2 \cup V_3$ also has the minimum size. As a result, $V_1 \cup V_4$ is also a maximum independent set. Therefore, we can use Ford-Fulkerson Algorithm, or Edmonds-Karp Algorithm, to find a minimum vertex cover, or a maximum independent set, in bipartite graphs, owing to the max-flow min-cut theorem.

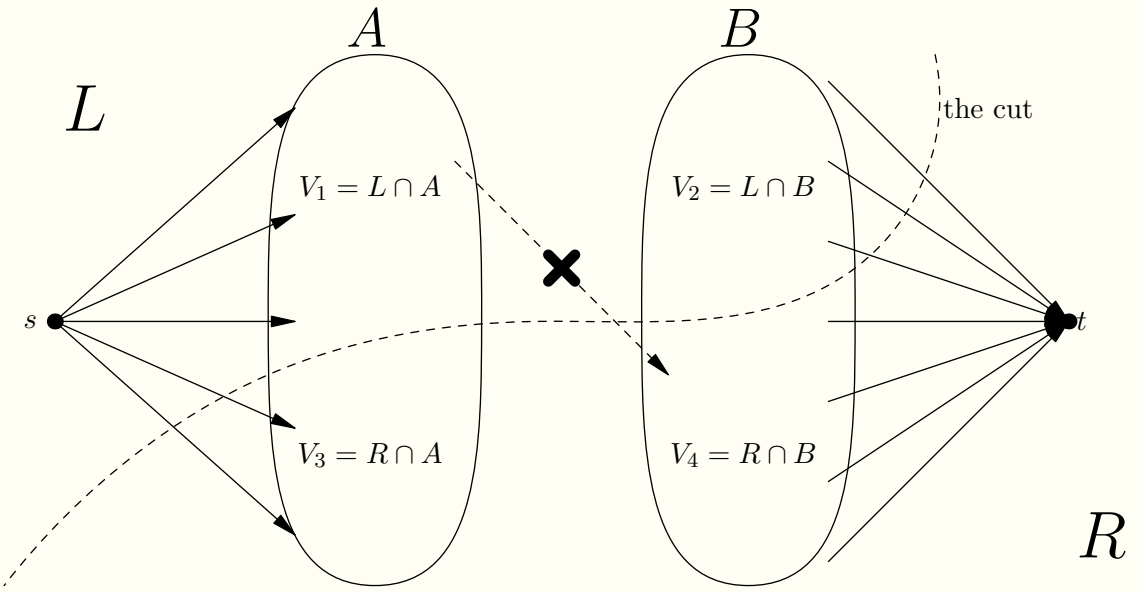

Figure 10: The cut separates V_2 from t and separates V_3 from s, so $v(f) = c(L, R) = |V_2| + |V_3|$. There does not exist any edge (u, v) such that $u \in V_1$ and $v \in V_4$

6 Strong Duality Theorem and Max-Flow Min-Cut

In this section, we will show that the max-flow min-cut theorem is essentially a special case of the strong duality theorem we learned in the previous lecture.
We compute the dual program to the linear program (1) describing the maximum flow problem. Firstly, we convert (1) to its standard form.

$$\begin{aligned} \max \quad & \textstyle\sum_{u:(s,u)\in E} f_{su} \\ \text{subject to} \quad & \forall u \in V \setminus \{s,t\} : \textstyle\sum_{u:(v,u)\in E} f_{vu} - \sum_{w:(u,w)\in E} f_{uw} \le 0 \\ & \forall u \in V \setminus \{s,t\} : -\textstyle\sum_{u:(v,u)\in E} f_{vu} + \sum_{w:(u,w)\in E} f_{uw} \le 0 \\ & \forall (u,v) \in E : f_{uv} \le c_{uv} \\ & \forall (u,v) \in E : f_{uv} \ge 0 \end{aligned} \tag{4}$$

Then, we compute the dual program, with z_u corresponding to the first set of constraints, z'_u corresponding to the second set of constraints, and y_{uv} corresponding to the third set of constraints.

$$\begin{aligned} \min \quad & \textstyle\sum_{(u,v)\in E} c_{uv} y_{uv} \\ \text{subject to} \quad & \forall w \in N^{out}(s) : z_w - z'_w + y_{sw} \ge 1 \\ & \forall v \in N^{in}(t) : z'_v - z_v + y_{vt} \ge 0 \\ & \forall (u,v) \in E, u \ne s, v \ne t : z'_u - z_u + z_v - z'_v + y_{uv} \ge 0 \\ & \forall u \in V \setminus \{s,t\} : z_u \ge 0 \\ & \forall (u,v) \in E : y_{uv} \ge 0 \end{aligned} \tag{5}$$

By letting $\hat{z}_w = z_w - z'_w$, this becomes

$$\begin{aligned} \min \quad & \textstyle\sum_{(u,v)\in E} c_{uv} y_{uv} \\ \text{subject to} \quad & \forall w \in N^{out}(s) : \hat{z}_w + y_{sw} \geq 1 \\ & \forall v \in N^{in}(t) : -\hat{z}_v + y_{vt} \geq 0 \\ & \forall (u,v) \in E, u \neq s, v \neq t : \hat{z}_v - \hat{z}_u + y_{uv} \geq 0 \\ & \forall (u,v) \in E : y_{uv} \geq 0 \end{aligned} \tag{6}$$

Now, consider an arbitrary s-t path $s \to u_1 \to \cdots \to u_k \to t$. We have

$$\begin{aligned} \hat{z}_{u_1} + y_{su_1} &\geq 1 \\ \hat{z}_{u_2} - \hat{z}_{u_1} + y_{u_1u_2} &\geq 0 \\ \hat{z}_{u_3} - \hat{z}_{u_2} + y_{u_2u_3} &\geq 0 \\ &\vdots \\ \hat{z}_{u_k} - \hat{z}_{u_{k-1}} + y_{u_{k-1}u_k} &\geq 0 \\ -\hat{z}_{u_k} + y_{u_kt} &\geq 0 \end{aligned}$$

Adding these together, we have

$$y_{su_1} + y_{u_kt} + \sum_{i=1}^{k-1} y_{u_iu_{i+1}} \geq 1.$$

On the other hand, the constraint matrix in the linear program (6) is *totally unimodular*. This implies that there exists an optimal solution to this linear program such that $y_{uv} \in \{0,1\}$ for each $(u,v) \in E$. (Search on the Internet for more information about these.) The inequality above implies that, on any s-t path p, there exists at least one edge (u,v) with $y_{uv} = 1$. Therefore, those edges (u,v) such that $y_{uv} = 1$ define a cut. To be precise, let L be all the vertices reachable from s if we remove those (u,v) with $y_{uv} = 1$, and let $R = V \setminus L$. Since every s-t path has at least one edge removed, we have $t \notin L$, so $\{L,R\}$ is a valid cut.

Moreover, we have $c(L,R) \leq OPT$, where $OPT = \sum_{(u,v)\in E} c_{uv} y_{uv}$ is the optimal objective value of the linear program (6). To see this, let $E(L,R)$ be the set of edges (u,v) with $u \in L$ and $v \in R$, and let F be the set of edges (u,v) with $y_{uv} = 1$. It suffices to show that $E(L,R) \subseteq F$ (since $c(L,R) = \sum_{e\in E(L,R)} c_e \leq \sum_{e\in F} c_e = OPT$). To see this, if we have $(u,v) \in E(L,R)$ with $(u,v) \notin F$, then v is still reachable from s if we remove those edges in F. This implies $v \in L$, which is a contradiction.

On the other hand, we have $c(L',R') \geq OPT$ for any cut $\{L',R'\}$. To see this, let $\hat{z}'_u = 1$ if $u \in L'$, and $\hat{z}'_u = 0$ if otherwise; let $y'_{uv} = 1$ if $(u,v) \in E(L',R')$, and $y'_{uv} = 0$ if otherwise. It is easy to verify that those $\hat{z}'_u$ and y'_{uv} give a valid solution to the linear program (6). In addition, the value of the cut $c(L',R')$ is precisely the objective $\sum_{(u,v)\in E} c_{uv} y'_{uv}$, which is at least OPT. Therefore, we have $c(L',R') \geq OPT$ for any cut $\{L',R'\}$.

Putting those together, $c(L,R) = OPT$. The optimal solution to the linear program (6) precisely defines a minimum cut.