NOI.PH Training: Flow 1

Network Flow Algorithms

Vernon Gutierrez

Contents

1 Introduction

Many interesting real-life situations can be modelled as a flow network problem. There are also lots of interesting theoretical problems that can be reduced to problems on flow networks. For these reasons, it is one of the most important and widespread topics in computer science. In fact, due to the sheer number of applications, it can be considered a “paradigm” on its own, along with dynamic programming, greedy, etc.

Although this topic is explicitly excluded from the IOI syllabus, most IOI competitors are familiar with it. Many flow algorithms use some basic graph algorithm (such as DFS, BFS, Dijkstra, or Bellman-Ford) as a subroutine. Hence, studying these algorithms forces one to practice these and generally improves one’s skills in solving graph problems. In addition, bipartite matching is included in the IOI syllabus, and although there are ways to solve bipartite matching without using flow networks, studying the flow network reduction deepens your understanding of bipartite matching. It will thus be helpful to study this topic for the IOI.

You may first watch these lectures from MIT or these lectures from Stanford to get an intuition for the concepts before reading the text below for details and rigor. The MIT lectures are based on the CLRS formulation of network flows, while the Stanford lectures are roughly based on the Kleinberg and Tardos formulation. The MIT/CLRS material go for rigorous proofs while the Stanford material discuss more algorithms, applications, and generalizations. I personally prefer the Stanford material, but the MIT material is also valuable for exposure to a slightly different approach. The text below tries to offer what the above two sources don’t: intuitive (hopefully) explanations of why the algorithms work and practical aspects of implementation and solving programming contest problems. It relies on and is meant to be read along with these slides from Princeton. Just follow the text, and it will prompt you to open a link to look at the slides with the specified pages for illustrations. The empirical comparisons and conclusions below are based on this article from TopCoder. There are several factual errors in that article, but I trusted the validity of the experiments. The conclusions below can be wrong if the experiments turn out to be flawed.

2 Maximum Flow and Minimum Cut

2.1 Basic Definitions

Let's say we have a directed graph $G = (V, E)$ with a special structure. There is a special node $s \in V$ called the **source**, whose in-degree is 0, and another special node $t \in V$ called the **sink**, whose out-degree is 0. We will further assume that our graph is simple (there are no self-loops and parallel edges between the same nodes), and that there is at least one edge incident to every node. (A practice problem later will force you to think about how to get around some of these assumptions.) Additionally, for each edge $e \in E$, we assign a non-negative capacity $c(e)$. A graph that satisfies all these properties is called a **flow network**.

Take a look at I.3 for a visualization of a flow network.

Intuitively, a flow network represents "stuff" (water, electricity, etc.) that can flow from one location to another via pipes. Stuff is produced at a special location called the source, and is consumed at another special location called the sink. Each pipe imposes a certain limit on the volume of stuff that can pass through it at any given point in time. A natural question to ask is the following: what is the maximum rate of flow from the source to the sink?

More formally, let us define an **st-flow** (or just a **flow**) as a function $f : E \mapsto N$ that assigns an integer to each edge, respecting the following two constraints:

- Capacity constraint: for each $e \in E$, $0 \leq f(e) \leq c(e)$
- Flow conservation constraint: for each $v \in V - \{s, t\}$, $\sum\limits_{e \text{ in to } v} f(e) = \sum\limits_{e \text{ out of } v} f(e)$

If $f(e) = c(e)$, we say that edge e is **saturated** with respect to flow f.

Take a look at I.8 for a visualization of a valid flow defined on the network you just saw. You can check that the capacities are respected and flow is conserved for the edges and the node v highlighted in blue.

We define the **value** of the flow as the sum of the flow values assigned to all edges going out of the source, $val(f) = \sum\limits_{e \text{ out of } s} f(e)$. The **maximum flow problem** is to find a flow of maximum value.

Exercise 2.1. What is the value of the flow we just defined? See I.9 for the answer.

Exercise 2.2. Is this the maximum possible value for this network? If no, what is? See I.10 for the answer.

Let's momentarily turn our attention to another problem. Interpret the capacity of each edge as the cost of removing the edge from the graph. Another interesting question would be: what is the least total cost to hijack the network, to disconnect the source from the sink and completely prevent the stuff from flowing from the source to the sink?

More formally, let us define an **st-cut** (or just a **cut**) as a partition (A, B) of the vertices with $s \in A$ and $t \in B$. The **capacity** of the cut $cap(A, B)$ is the sum of the capacities of the edges from A to B:

Take a look at I.4-I.5 to see examples of cuts.

The **minimum cut problem** is to find a cut of minimum capacity.

Exercise 2.3. What is the minimum capacity cut of our graph? See I.6 for the answer.

On first glance, these two problems appear to be unrelated. But in fact, we shall see later that they are essentially the same problem, both solvable using the same algorithms. It is no coincidence that the value of the maximum flow and the capacity of the minimum cut are the same for our given graph.

We will solve the max-flow problem first, and then see how to apply the same solution to the min-cut problem.

Exercise 2.4. Before looking at the algorithm below, think about how you might approach this problem.

2.2 Incremental Improvement: Augmenting Paths (Ford-Fulkerson)

A very natural approach to getting the maximum flow would be the following:

1. Start with an **empty flow**: let $f(e) = 0$ for all $e \in E$.
2. Find an **s-t path** of unsaturated edges: a path P that starts at s and ends at t, where for each edge e along P, $f(e) < c(e)$.
3. Augment the flow along P: for each edge e along P, increase $f(e)$ by the **bottleneck capacity** $\min_{e \in P}(c(e) - f(e))$. In other words, saturate the minimum-remaining-capacity edges (**bottleneck edges**) in P.
4. Repeat until there are no more such paths.

The idea is that we start with an obviously non-optimal answer but which surely satisfies the constraints, and then converge towards the optimum by *incremental improvements* that still respect the constraints. Take a look at I.12-I.17 for a sample run of this algorithm.

Exercise 2.5. Does this correctly find the maximum flow? Find a counterexample. See I.18 for an answer.

Our approach fails because it is too greedy. Ideally, what we want is to have some way to "undo" certain flow increases. Look at I.17 again. If we can somehow increase the flow by one unit across the edge going out of s labeled $6/10$, undo pushing one unit of flow across the edge labeled $2/2$, and finally redirect that one unit of flow towards the edge labeled $0/4$ and then towards the edge going into t labeled $6/10$, we still satisfy the capacity and flow conservation constraints, but strictly increase the flow value by one. To reach the optimal answer, we can do this once more, and then another time involving the edge labeled $8/8$ instead. In other words, what we really want is to be able to push flow "forward" normally, but also to be able to push flow "backward" along the reverse of edges that already have flow going forward. We still successively augment along s-t paths, but we now allow the usage of backward edges as part of the paths. The intuitive reason why this works is as follows. Performing a "backward" push along some edge $e = (u, v)$ amounts to splitting a previously constructed path $P = s, \dots, t$ into two parts: $P_1 = s, \dots, u$ and $P_2 = v, \dots, t$. It similarly splits our new path $P' = s, \dots, t$ into $P_1' = s, \dots, v$ and $P_2' = u, \dots, t$. In order for the augmentation along P_1' to be valid, respecting the conservation constraint on v, the flow from u to v is undone and redirected elsewhere, in particular to P_2'. This allows us to "reconstruct" an augmenting path $P'_1 \frown P_2$. But since we take away P_2 from P, we have to ensure that P_1 can still connect to the sink to be a valid augmenting path, and that the conservation constraint on u is respected. Redirecting the flow to P_2' achieves this for us, allowing us to "reconstruct" another augmenting path $P_1 \frown P_2'$.

The concept of a *residual graph* gives us a clean way to keep track of these forward and backward pushes. Given a flow network $G = (V, E)$ with edge capacities c and a flow f, we define the **residual capacity** $c_f(e)$ of some $e = (u, v)$ with respect to f as follows:

$$c_f(e) = \begin{cases} c(e) - f(e) & e = (u,v) \in E \\ f(e) & e^R = (v,u) \in E \end{cases}$$

An edge e is a **residual edge** with respect to f iff $c_f(e)$ is defined. Finally, we define the **residual graph** $G_f = (V, E_f)$ of G with respect to f as the graph with the same node set, and whose edge set is the set of residual edges $e \in E_f$ iff $c_f(e)$ is defined. Intuitively, the residual graph consists of edges which we can still use for augmentation: forward edges e with "leftover" capacity $c(e) - f(e)$, and backward edges e through which we can "undo" $f(e)$ units of flow that have previously been pushed forward in the opposite direction e^R. See I.20 for an example.

We now have a simple algorithm, first invented by Ford and Fulkerson in 1955, for finding the maximum flow.

1. Start with an **empty flow**: let $f(e) = 0$ for all $e \in E$.
2. Find an s-t path P *in the residual graph* G_f, where for each edge e along P, $c_f(e) > 0$. We call this an **augmenting path**.
3. Let $b = \min\limits_{e \in P} c_f(e)$ be the bottleneck capacity of P.
4. Augment the flow along P: for each edge e along P, if e is a forward edge ($e \in E$), increase $f(e)$ by b, otherwise decrease $f(e^R)$ by b (as e is a backward edge).
5. Repeat until there are no more such paths.

Try running this algorithm on the earlier graph. Verify that you indeed obtain the correct maximum flow value.

Is this algorithm correct? To prove it, we need to prove three things:

1. Augmentation never violates the capacity and conservation constraints.
2. The algorithm always terminates.
3. At termination, the value of the flow is maximum.

Statements 1 and 2 are quite easy to prove. Statement 3 is more subtle, and leads us into proving the equivalence of max-flow and min-cut, and that we also now have an algorithm for solving the min-cut problem. But first, try proving statements 1 and 2 and try implementing the algorithm.

Problem 2.1. Given a flow network G, prove that if f is a flow, then the function $f' = \mathit{Augment}(f, P)$ obtained by augmenting the flow f along an augmenting path P in the residual graph G_f is also a flow. In particular, verify that the capacity and conservation constraints are respected. (Hint: Since the f is changed only for the edges along P, you only need to verify that the constraints are respected for these edges. Consider forward and backward edges separately.)

Problem 2.2. Prove that the flow value strictly increases at every augmentation.

Problem 2.3. Let $C = \sum_{e \text{ out of } s} c(e)$. Prove that the Ford-Fulkerson algorithm can be implemented to run in $\mathcal{O}(EC)$ time. (Hint: Recall that we assumed that there is at least one edge incident to every node.)

Exercise 2.6. Try implementing your own version of Ford-Fulkerson first before comparing it with the implementation below. Test it on this problem: UVa 820 - Internet Bandwidth. Don't forget to print a blank line after each test case.

Here is a simple implementation of the Ford-Fulkerson algorithm. Either DFS or BFS can be used to find augmenting paths. This implementation uses DFS, chosen arbitrarily.

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 100;
const int INF = 1'000'000; // a VERY useful C++ feature
int n, m, s, t;
unordered_set<int> adj[MAX_N+1];
int c[MAX_N+1][MAX_N+1];
int f[MAX_N+1][MAX_N+1];
int p[MAX_N+1]; // parent in the DFS tree, needed for retrieving the augmenting path
bool dfs(int u) {
    if(u == t) return true;
    for(int v : adj[u]) {
        if(p[v] == -1 and c[u][v] - f[u][v] > 0) { // c[u][v] - f[u][v] is the
                                                   // residual capacity
            p[v] = u;
            if(dfs(v)) return true;
        }
    }
    return false;
}
bool find_aug_path() {
    memset(p, -1, sizeof p); // p[v] == -1 implies not discovered
    p[s] = 0; // dummy to mark the source as discovered
    return dfs(s);
}
int main() {
    memset(c, 0, sizeof c);
    memset(f, 0, sizeof f);
    // assume input is of the following format:
    // n (number of vertices) s (source) t (sink) m (number of edges)
    // u_1 v_1 capacity_1
    // ...
    // u_m v_m capacity_m
    cin >> n >> s >> t >> m;
    for(int i = 0; i < m; i++) {
        int u, v, capacity;
        cin >> u >> v >> capacity;
        adj[u].insert(v);
        adj[v].insert(u); // so that backward edges are included in the DFS
        // parallel edges are handled by just combining them into a single edge,
        // whose capacity equals the total of the capacities of the parallel edges
        c[u][v] += capacity;
        // c[v][u] += capacity; // for undirected graphs
    }
    int max_flow_value = 0;
    while(find_aug_path()) {
        int b = INF;
        for(int v = t, u = p[v]; v != s; v = u, u = p[v]) b = min(b, c[u][v] - f[u][v]);
        for(int v = t, u = p[v]; v != s; v = u, u = p[v]) f[u][v] += b, f[v][u] -= b;
        max_flow_value += b;
    }
    cout << max_flow_value << '\n';
    return 0;
}
```

Here it is again, with the comments removed, just to highlight how short and simple the algorithm really is.

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 100;
const int INF = 1'000'000;
int n, m, s, t;
set<int> adj[MAX_N+1];
int c[MAX_N+1][MAX_N+1];
int f[MAX_N+1][MAX_N+1];
int p[MAX_N+1];
bool dfs(int u) {
    if(u == t) return true;
    for(int v : adj[u]) {
        if(p[v] == -1 and c[u][v] - f[u][v] > 0) {
            p[v] = u;
            if(dfs(v)) return true;
        }
    }
    return false;
}
bool find_aug_path() {
    memset(p, -1, sizeof p);
    p[s] = 0;
    return dfs(s);
}
int main() {
    memset(c, 0, sizeof c);
    memset(f, 0, sizeof f);
    cin >> n >> s >> t >> m;
    for(int i = 0; i < m; i++) {
        int u, v, capacity;
        cin >> u >> v >> capacity;
        adj[u].insert(v);
        adj[v].insert(u);
        c[u][v] += capacity;
    }
    int max_flow_value = 0;
    while(find_aug_path()) {
        int b = INF;
        for(int v = t, u = p[v]; v != s; v = u, u = p[v]) b = min(b, c[u][v] - f[u][v]);
        for(int v = t, u = p[v]; v != s; v = u, u = p[v]) f[u][v] += b, f[v][u] -= b;
        max_flow_value += b;
    }
    cout << max_flow_value << '\n';
    return 0;
}
```

Let's notice a few things about this implementation. First, notice that we do not have to explicitly construct G_f, as just by keeping track of c and f, we can easily infer c_f for path finding. Second, notice that the way we define the residual capacity is slightly different here. We simply use $c(e) - f(e)$ and do not discriminate the forward and backward directions. We recover the original definition by defining $c(e)$ to be 0 and allowing $f(e)$ to be negative on backward edges. The augmentation procedure is also slightly modified to always increment the forward direction and to always decrement the backward direction. We do this to simplify the implementation and also to be able to handle anti-parallel edges (edges between the same nodes but in opposite directions). Take a moment to convince yourself that this works, and that the previous definition does not work for anti-parallel edges but this one does. There are other ways to deal with parallel and anti-parallel edges but this is the one I find the simplest, though slightly non-intuitive. Also notice that we use an *adjacency set* instead of an adjacency list here, to avoid storing duplicate neighbors due to parallel edges. Finally, this implementation is not the most memory-efficient possible one, but as we will see later, the running time of all practical maximum flow algorithms are all $\Omega(n^3)$, where n is the number of nodes in the graph. This means that max-flow approaches to a problem are practical in terms of time iff using $\mathcal{O}(n^2)$ memory is practical. Hence, it does not matter that we use $\mathcal{O}(n^2)$ memory here. It makes the implementation simpler and require less time to run (which is more important for max-flow-related problems). In cases where the memory limit is really tight (e.g., max-flow is only part of the problem, and the other parts need the memory, or a small bound on the maximum flow value can be proven, so that only a few augmentations are needed), it is fairly trivial to change this implementation to use a linear amount of memory.

2.3 The Max-Flow Min-Cut Theorem

We now prove that the Ford-Fulkerson algorithm yields the maximum flow. As a side effect, we also prove the equivalence of max-flow and min-cut.

The idea is to find a tight upper bound on what the value of the max-flow can be, and to show that the Ford-Fulkerson algorithm indeed reaches this bound. One such bound is an obvious one: the value of a flow is always less than or equal to the sum of the capacities of the edges going out of the source, $val(f) \le \sum_{e \text{ out of } s} c(e)$. It is not tight enough to be helpful for proving anything, but the intuition behind this bound is helpful and can be generalized. Rather than considering just the sum of the capacities going out of the source, let's generalize to any "moat" around the source, and consider the sum of the capacities going out of this "moat" (more formally, a *cut*). It makes intuitive sense that the value of a flow must be smaller than this sum, and we state this as Lemma 2.9. To prove it formally, we need the following lemma first.

Lemma 2.7 (Flow Conservation Across Cuts)**.** Let f be any flow and let (A, B) be any cut. Then the net flow across (A, B) equals the value of f.

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = val(f)$$

See I.25-I.27 for examples of what this lemma is saying.

Exercise 2.8. Prove Lemma 2.7. (Hint: Use the flow conservation constraint.) See I.29 for the answer.

Problem 2.4. As a simple application of Lemma 2.7, prove that the value of the flow is equivalent to (and thus may also be alternatively defined as) the sum of the flows of the edges going into the sink: $val(f) = \sum_{e \text{ out of } s} f(e) = \sum_{e \text{ in to } t} f(e)$.

Using Lemma 2.7, we can now prove a stronger bound on the value of the flow.

Lemma 2.9 (Weak Duality Between Flows and Cuts)**.** Let f be any flow and let (A, B) be any cut. Then the value of the flow is less than or equal to the capacity of the cut.

$$val(f) \leq cap(A, B)$$

Let's think carefully about what Lemma 2.9 is saying. It is actually saying something quite strong: the value of *any* flow is always less than or equal to the capacity of *any* cut. In particular, this means that the max-flow value is less than the min-cut capacity. If we can somehow produce a flow f and a cut (A, B) where $val(f) = cap(A, B)$, then we know that f is a max-flow, and (A, B) is a min-cut. It turns out that the Ford-Fulkerson algorithm indeed produces a flow with this property. If we prove this, we both prove that the Ford-Fulkerson algorithm correctly finds the maximum flow and that max-flow is equivalent to min-cut (and by extension, that the Ford-Fulkerson algorithm also allows us to find the minimum cut).

Lemma 2.10 (No Augmenting Paths Implies Existence of Cut Equivalent to Flow)**.** Let f be a flow such that there are no s-t paths in the residual graph G_f. Then there is a cut (A, B) where $val(f) = cap(A, B)$.

Problem 2.5. Let A be the set of nodes reachable from the source using residual edges in G_f above, and let $B = V - A$. Prove that (A, B) is a cut (i.e. that they are disjoint, that $s \in A$, and that $t \in B$).

Problem 2.6. Consider an edge $e = (u, v)$ where $u \in A$ and $v \in B$. Prove that $f(e) = c(e)$.

Problem 2.7. Consider an edge $e = (u, v)$ where $v \in A$ and $u \in B$. Prove that $f(e) = 0$.

The above two statements imply that all edges out of A are completely saturated with flow, while all edges in to A are completely empty.

Problem 2.8. Use the above facts, together with Lemma 2.7, to prove Lemma 2.10.

Lemma 2.9 and Lemma 2.10 easily imply the following corollary.

Corollary 2.1 (No Augmenting Paths Implies Maximum Flow, Minimum Cut)**.** Let f be a flow such that there are no s-t paths in the residual graph G_f. The value of f is maximum over all possible flows in G. The capacity of the cut (A, B) whose existence is guaranteed by Lemma 2.10 and whose capacity is equal to the value of f is minimum over all possible cuts in G.

Since the Ford-Fulkerson algorithm only terminates when there are no more s-t paths in the residual graph, its correctness easily follows from Corollary 2.1.

Theorem 2.11 (Correctness of Ford-Fulkerson)**.** The flow produced by the Ford-Fulkerson algorithm is a maximum flow.

The Ford-Fulkerson algorithm guarantees that in every flow network, there is a flow f and a cut (A, B) where $val(f) = cap(A, B)$, which immediately implies the following famous theorem.

Theorem 2.12 (Max-Flow Min-Cut Theorem)**.** In every flow network, the maximum value of a flow is equal to the minimum capacity of a cut.

This beautiful relationship between flows and cuts is an example of the more general mathematical principle of duality.

See I.33-I.35 for a slightly different proof.

Exercise 2.13. After applying the Ford-Fulkerson algorithm to find the maximum flow, how would you produce the actual minimum-capacity cut (partition) (A, B)?

2.4 Finding Augmenting Paths Smartly

2.4.1 Shortest Augmenting Paths (Edmonds-Karp)

We have seen a very simple algorithm for solving the max-flow min-cut problem. Unfortunately, we have only proven that it runs in $\mathcal{O}(EC)$.

Exercise 2.14. Prove another bound on the running time of Ford-Fulkerson: $\mathcal{O}(EF)$, where F is the output maximum flow value.

Both of these bounds can be bad for certain instances of the problem where the edge capacities are large. In fact, we technically do not yet have a polynomial-time algorithm, as time complexity is measured in the number of bits of input, and C (likewise F) is exponential in $\lg C$ (likewise $\lg F$), which is the number of bits required to represent the edge capacities. (Don't fret if you don't quite understand why time complexity is measured this way. This is a fine technical point.)

Exercise 2.15. Come up with an instance of the max-flow problem that causes the Ford-Fulkerson algorithm to actually require $\Omega(EC)$ amount of time. See I.38 for the answer.

An interesting but completely useless side note: the Ford-Fulkerson algorithm is not even guaranteed to terminate if the capacities are irrational numbers!

To improve the Ford-Fulkerson algorithm, we need to have a good way of finding augmenting paths. Intuitively, it makes sense that we need both an efficient path-finding algorithm, and one which leads to the fewest possible iterations of augmentation. The first condition leads us to consider finding augmenting paths with the fewest number of edges, the *shortest augmenting paths*. Fortunately, this is very simple to achieve. Just use BFS to find augmenting paths. The idea is, unlike with DFS, where the path to the sink can be long, with BFS, we can stop early when we discover the sink, and maybe that leads to a faster algorithm. Surprisingly, it is the second of the two conditions above (fewest possible iterations of augmentation) which we actually fulfill. The difference between DFS and BFS turns out to not matter too much for any single particular iteration (and hence there is no real need to stop the BFS early), but makes a difference *globally*, when we consider the running time of all the iterations together. This improvement was first invented by Edmonds and Karp in 1972. The improvement itself is not hard to invent. It is the proof that it actually works and makes the running time strictly polynomial that is difficult and which these two guys got credit for.

If you randomly picked DFS instead of BFS on your first attempt to implement the Ford-Fulkerson algorithm, now is your chance to redo and solve UVa 820 - Internet Bandwidth, before comparing with the implementation below.

Here is an implementation of the Edmonds-Karp algorithm. Notice that the only thing that changes from above is the usage of BFS instead of DFS. Everything else (residual capacities, finding the bottleneck, updating the flow) stays the same.

```
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 100;
const int INF = 1'000'000;
int n, m, s, t;
set<int> adj[MAX_N+1];
int c[MAX_N+1][MAX_N+1];
int f[MAX_N+1][MAX_N+1];
int p[MAX_N+1];
bool bfs() {
    queue<int> q;
    q.push(s);
    while(not q.empty()) {
        int u = q.front(); q.pop();
        for(int v : adj[u]) {
            if(p[v] == -1 and c[u][v] - f[u][v] > 0) {
                p[v] = u;
                q.push(v);
            }
        }
    }
    return p[t] != -1;
}
bool find_aug_path() {
    memset(p, -1, sizeof p);
    p[s] = 0;
    return bfs();
}
int main() {
    memset(c, 0, sizeof c);
    memset(f, 0, sizeof f);
    cin >> n >> s >> t >> m;
    for(int i = 0; i < m; i++) {
        int u, v, capacity;
        cin >> u >> v >> capacity;
        adj[u].insert(v);
        adj[v].insert(u);
        c[u][v] += capacity;
    }
    int max_flow_value = 0;
    while(find_aug_path()) {
        int b = INF;
        for(int v = t, u = p[v]; v != s; v = u, u = p[v]) b = min(b, c[u][v] - f[u][v]);
        for(int v = t, u = p[v]; v != s; v = u, u = p[v]) f[u][v] += b, f[v][u] -= b;
        max_flow_value += b;
    }
    cout << max_flow_value << '\n';
    return 0;
}
```

Why does this simple change make a big difference? Let's now try to analyze the running time of the shortest augmenting paths algorithm. First, we need to prove some lemmas.

Lemma 2.16 (Monotonically Non-Decreasing Distances)**.** Throughout the shortest augmenting paths algorithm, the distance from the source to any node in the residual graph never decreases from one iteration to the next.

You can convince yourself of this by running the algorithm on a few graphs and printing out the paths found by the algorithm in each iteration, but it's nice to see a (75%) formal proof.

Proof. Consider the residual graphs G_f and $G_{f'}$ associated with flows f and f' before and after applying an augmentation through augmenting path P. The bottleneck edges in P are present in G_f but absent from $G_{f'}$ (as for each bottleneck edge $e \in P$, e is saturated if it is a forward residual edge, or e^R is emptied if e is a backward residual edge). In addition, new edges which are anti-parallel to the bottleneck edges in G_f are created in $G_{f'}$.

Let's assume there is only one bottleneck edge $(u, v) \in P$ and compare the distances of u and v in G_f to their distances in $G_{f'}$. Denote the distance of a node u in a particular residual graph G_f as $d_f(u)$. Since (u, v) is an edge in P, $d_f(v) = d_f(u) + 1$. What can we say about the distances of u and v in $G_{f'}$? Since (u, v) is a bottleneck edge, it is absent from $G_{f'}$. Note that the distance to v can never decrease from one iteration to the next by removing edges pointing into v, and hence $d_{f'}(v) \geq d_f(v)$. What about the distance to u? Since (u, v) is a bottleneck edge, it is replaced by an anti-parallel edge (v, u) in $G_{f'}$. Is it possible that the distance to u decreases because of a new edge pointing into it? The answer is no. To see why, suppose that the distance to u does decrease from one iteration to the next; that is

$$d_f(u) > d_{f'}(u) \tag{1}$$

If the distance to u decreases, it can only possibly decrease by using the edge (v, u). Thus $d_{f'}(u) = d_{f'}(v) + 1$, and

$$d_{f'}(u) > d_{f'}(v) \tag{2}$$

Taking these two inequalities together, we have

$$d_f(u) > d_{f'}(v) \tag{3}$$

We have just argued that the distance to v can never decrease. Hence

$$d_{f'}(v) \geq d_f(v) \tag{4}$$

Again, taking the two previous inequalities together, we have

$$d_f(u) \geq d_f(v) \tag{5}$$

But this contradicts the fact that $d_f(v) = d_f(u) + 1$. Therefore, the distance to u cannot decrease from one iteration to the next.

We can repeatedly apply the same argument for the case when there are many bottleneck edges. Just consider edges in P in increasing order of their nodes' distance from the source and proceed by induction.

Exercise 2.17. Prove this part more rigorously.

From this, we can conclude that $d_{f'}(v) \geq d_f(v)$ for all nodes v and for all augmentation steps $f' = \mathit{Augment}(f, P)$. $\square$

In particular, the distance from the source to the sink never decreases from one iteration to the next.

Corollary 2.2 (Monotonically Non-Decreasing Augmenting Path Lengths)**.** Throughout the shortest augmenting paths algorithm, the length of an augmenting path in the residual graph never decreases from one iteration to the next. That is, for all augmentation steps $f' = \mathit{Augment}(f, P)$,

$$d_{f'}(t) \geq d_f(t)$$

Armed with Lemma 2.16, we can now prove the following.

Lemma 2.18 (Using Reverse Edges Increases Augmenting Path Length)**.** Suppose that at some iteration i of the algorithm, (u, v) is a bottleneck edge in the augmenting path P, so the residual edge (v, u) is in the next few residual graphs. If at some later iteration $i' > i$, the augmenting path P' includes (v, u), then the length of P' is strictly greater than the length of P.

Again, you can convince yourself by observing the augmenting paths found by the algorithm on several different graphs, but let's see a (75%) formal proof.

Proof. Let f and f' denote the flows (before applying the augmentation) at the earlier and the later iteration respectively. Since P' is a shortest path in $G_{f'}$ which goes through (v, u), we have

$$d_{f'}(u) = d_{f'}(v) + 1 > d_{f'}(v) \tag{6}$$

We know from Lemma 2.16 that

$$d_{f'}(v) \geq d_f(v) \tag{7}$$

Since (u, v) is an edge in P (which is a shortest path),

$$d_f(v) = d_f(u) + 1 > d_f(u) \tag{8}$$

Putting these three inequalities together, we conclude that $d_{f'}(u) > d_f(u)$. Note that this is a stronger statement than Lemma 2.16 implies, since here we have a strict inequality. From here, it is not hard to conclude that the distances to all nodes w in the path from u to t are strictly larger in $G_{f'}$ than in G_f. More briefly, $d_{f'}(w) > d_f(w)$. In particular, $d_{f'}(t) > d_f(t)$. All of this assumes we included the reverse edge (v, u) in P'. $\square$

How many times can we avoid using the reverse of a bottleneck edge before we are forced to use one to find an augmenting path? We have $\mathcal{O}(E)$ possible bottleneck edges to exhaust before we are forced to use the reverse of any one of them. Therefore, the shortest augmenting path must strictly increase after $\mathcal{O}(E)$ iterations.

Corollary 2.3 (Bound on How Long the Augmenting Path Length Remains Constant)**.** The length of the shortest augmenting path increases after at most $\mathcal{O}(E)$ iterations.

Using the above facts, it is not that hard to prove the following theorem.

Theorem 2.19 (Efficiency of Shortest Augmenting Paths)**.** The shortest augmenting paths algorithm solves the maximum flow problem in $\mathcal{O}(E^2V)$ time.

Exercise 2.20. Prove Theorem 2.19. (Hint: How many times can the length of the shortest augmenting path increase?) See I.53 for the answer.

See I.48-I.53 for another, slightly different proof.

Ford and Fulkerson did not really specify what method must be used to find augmenting paths. We can think of the Ford-Fulkerson algorithm as not really an algorithm, but more of a template, where the actual method for finding the paths can be plugged in to create a full-fledged algorithm. Ford and Fulkerson's contribution was simply to establish the paradigm of "successively find augmenting paths," and left it to future generations of computer scientists to extend and refine this paradigm. The Edmonds-Karp improvement plugs in the "Shortest Augmenting Paths" method into this template. Although Edmonds-Karp is significantly better than vanilla Ford-Fulkerson, it is still quite bad, requiring $\Omega(V^5)$ in dense graphs. We need faster improvements. This and the rest of the algorithms in this section, except for the last, are merely different variations on how to find the augmenting paths, with different time complexities, but the basic idea is the same. (Hence, correctness of each of these easily follows from the correctness of Ford-Fulkerson.) In practice, however, the most efficient max-flow algorithms today use a completely different approach (cue dramatic pondering on the nature of scientific progress): the *pre-flow push-relabel* approach, which we will see in the last part of this section. But first, let's develop our intuition using simple algorithms before diving into the more complicated approach.

2.4.2 Fattest Augmenting Paths (Edmonds-Karp)

In the same paper where Edmonds and Karp introduce their algorithm above, they also describe another intuitive way to improve the Ford-Fulkerson algorithm: take augmenting paths with the largest bottleneck capacity, the *fattest augmenting paths*. This makes sense because increasing the flow by as much as possible per iteration leads to lessening the number of iterations of augmentation required. We can do this using some simple modification of Prim's/Dijkstra's algorithm. It can be shown that this method requires $\mathcal{O}(E\lg(EC))$ augmentations in total, and therefore $\mathcal{O}(E^2\lg V\lg(EC))$ total time, though we do not prove it here. This looks like an improvement over the shortest augmenting paths method. However, on dense graphs, the logarithmic terms and the constant factor overhead of using a priority queue for Prim's/Dijkstra's become significant. Doing Prim's/Dijkstra's without a priority queue turns out to not help either even with dense graphs. Compared with the shortest augmenting paths method, for most problems, the slight improvement in the sparse graph case only is not worth the extra implementation effort.

2.4.3 Capacity-Scaling (Gabow)

A slightly different idea for improving the Ford-Fulkerson algorithm was proposed by Gabow in 1985: maintain a scaling parameter Δ and consider only residual edges whose capacities are at least Δ. This Δ is initialized to be the largest power of two smaller than the maximum capacity of any edge going out of the source. A phase of several augmentations using this fixed Δ is performed, until no more augmentations can be made, and then Δ is divided by 2. This process is repeated until Δ reaches 0. See I.43 for pseudocode. This algorithm runs in $\mathcal{O}(E^2\lg C)$ time. We will skip the proof. You can look at I.45-I.46 for it. We will also skip the implementation. It is not too hard to try it on your own. In practice, this algorithm is significantly better than the shortest augmenting paths algorithm for sparse graphs, but only marginally better for dense graphs. Be forewarned though, that an implementation of capacity-scaling using DFS performs significantly more poorly than one that uses BFS.

2.4.4 Level Graph Blocking Flows (Dinitz)

The previous two improvements we have seen are theoretically interesting, but they are not significantly better than the shortest augmenting paths algorithm to be worth using. This one is though. Dinitz invented it in 1970, and proved independently of Edmonds and Karp that the max-flow problem can be solved in polynomial time. Interestingly, this algorithm is more commonly known today as "Dinic's" algorithm, because the guy who gave the initial lectures about this algorithm kept mispronouncing Dinitz' name.

Note: If you're learning flows for the first time, then you may choose to skip this part for now, since you already have enough prerequisites to tackle and appreciate maximum matching.

The idea behind the algorithm is not that difficult. Like Edmonds and Karp's algorithm, Dinic's algorithm will find the shortest augmenting paths, but it will find all augmenting paths of a fixed length in one go. We previously discussed two natural strategies for improving the running time of augmenting path algorithms: find paths efficiently, and reducing the number of iterations of augmentation. They are not mutually exclusive. Dinic's algorithm does both. By simultaneously augmenting along all shortest paths with the same length, Dinic's algorithm will require only $\mathcal{O}(V)$ *phases* of augmentation. (Why?) Using the idea of a *level graphs* and *blocking flows*, Dinic's algorithm can find and augment along all paths with the same length in $\mathcal{O}(VE)$. This makes the total running time $\mathcal{O}(V^2E)$, which is a significant improvement over Edmonds and Karp's $\mathcal{O}(VE^2)$ for dense graphs, and which in practice happens to also work significantly better than Edmonds-Karp in general for graphs of different densities. Let's describe the algorithm in precise detail.

First, we need the concept of a *level graph*. The **level graph** of a given graph $G = (V, E)$ is the subgraph containing only edges that can possibly be part of a shortest path from the source s to the sink t. Specifically, denote $d(v)$ as the distance of a node v from s, that is, the number of edges in the shortest path from s to v. The level graph $L(G) = (V, E_L)$ contains only those edges $(u, v) \in E$ where $d(v) = d(u) + 1$. See I.50 for an example. Note that for some residual graph G_f, a shortest augmenting path only contains edges in $L(G_f)$.

The level graph is closely related to the *BFS tree*.

Next, let's introduce the idea of a *blocking flow*. A flow f is a **blocking flow** for flow network G if there are no s-t paths in the subgraph obtained by removing saturated edges from G.

Stated another way, a blocking flow is just a flow which prevents augmentation using only forward residual edges. Notice that our first (wrong) algorithm was simply: find and augment along s-t paths until the current flow is a blocking flow.

Exercise 2.21. Prove or disprove: every maximum flow is a blocking flow.

Exercise 2.22. Prove or disprove: every blocking flow is a maximum flow.

Exercise 2.23. Prove or disprove: If f is a blocking flow for G, then there are no augmenting paths in G_f.

Each blocking flow can represent the flow produced by a bunch of augmenting paths. Dinic's algorithm will repeatedly find blocking flows and update the global flow using these blocking flows instead of individual augmenting paths. Let's make this notion of "augmenting a flow with another flow" more formal. Let f and b be two flows on G. Define the augmentation of f by b (or more simply, just the sum of f and b) as flow produced by combining the two flows for each edge: $f' = f + b$ iff $f'(e) = f(e) + b(e)$ for all $e \in G$.

At a very high level, Dinic's algorithm can be described as follows. Denote the flow at iteration i as f_i. Let f_0 initially be an empty flow. Perform $\mathcal{O}(V)$ phases of augmentation. In each phase, do the following:

1. Construct the level graph $L(G_{f_i})$ of the residual graph G_{f_i}.
2. Compute a blocking flow b_i of $L(G_{f_i})$.
3. Update the flow by augmenting it with the blocking flow: let $f_{i+1} = f_i + b_i$.

This description does not look intuitive at all. Weren't we trying to find all augmenting paths with the same length all in one phase? Why these notions of level graph and blocking flow? The reason why "find all augmenting paths with the same length" is the same as "find a blocking flow in the level graph" is made clear by the following lemma.

Lemma 2.24 (Augmentation by Level Graph Blocking Flow Increases Augmenting Path Length)**.** Let f be a flow and b be a blocking flow in $L(G_f)$. The distance from the source to the sink is strictly greater in G_{f+b} than in G_f:

$$d_{f+b}(t) > d_f(t)$$

Problem 2.9. Prove Lemma 2.24. (Hint: Consider Lemma 2.18.)

If you understand the proofs for the efficiency of Edmonds-Karp algorithm and the concepts above, it is actually not impossible to complete with Dinic's algorithm on your own. All you need is to find a way to perform each phase of augmentation in $\mathcal{O}(VE)$ time.

Exercise 2.25. Attempt to complete Dinic's algorithm on your own. (Hints: For a single phase, how much time is needed to construct the level graph? At most how many individual augmenting paths can make up a blocking flow of the level graph? Using the level graph, can we find one such augmenting path in $\mathcal{O}(V)$? What if we allow the level graph to be modified every time we augment along a path? In particular, what if we can delete nodes and edges from the level graph?)

Did you figure it out? It is easy to perform steps 1 and 3 of each phase both in $\mathcal{O}(E)$ time. Step 2 is trickier. Simply performing a DFS/BFS to find individual augmenting paths in the level graph to compute the blocking flow still requires $\mathcal{O}(E)$ per path. By Corollary 2.3, this makes each phase require $\mathcal{O}(E^2)$ time. This is really just Edmonds-Karp stated in an unnecessarily fancier way. However, if we can somehow find individual augmenting paths in the level graph in $\mathcal{O}(V)$ time, then we are done. DFS happens to help us in this case. Let's assume that we get lucky, and picking the first outgoing edge in every DFS call leads us to the sink. Then, we can find one augmenting path (plus update the blocking flow and delete bottleneck edges from the level graph) in $\mathcal{O}(V)$. The problem is, we can be unlucky in the DFS, and reach a dead end, say v, that has no path to the sink, causing us to backtrack and to require $\mathcal{O}(E)$ time to find one augmenting path. In this case, however, we are sure that no augmenting paths will ever pass through v until the next phase, so we can delete v (and all edges incident to it) from the level graph before backtracking. Since there are only V nodes in the graph, this unlucky case will only happen $\mathcal{O}(V)$ times per phase. We're done.

See I.58-I.71 for illustrations, pseudocode, and a more detailed proof.

Exercise 2.26. Before looking at the implementation of Dinic's algorithm below, re-solve UVa 820 - Internet Bandwidth, this time using your own implementation of Dinic's algorithm. Compare the actual running times of Edmonds-Karp's and Dinic's algorithms for this problem. Some care is needed to ensure that deleting a node or edge from the level graph can actually be done in constant time, to ensure the overall running time of the algorithm is $\mathcal{O}(V^2E)$.

Here is a clean implementation of Dinic's algorithm.

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 100;
const int INF = 1'000'000;
int n, m, s, t;
set<int> adj[MAX_N+1];
set<int> L_adj[MAX_N+1]; // level graph
set<int> L_adj_rev[MAX_N+1]; // to avoid extra linear factor for node deletion
int c[MAX_N+1][MAX_N+1];
int f[MAX_N+1][MAX_N+1];
int d[MAX_N+1]; // distance for level graph
int p[MAX_N+1]; // parent for blocking flow

bool make_level_graph() {
    memset(d, -1, sizeof d);
    for(auto L : L_adj) L.clear();
    for(auto L : L_adj_rev) L.clear();
    d[s] = 0;
    queue<int> q;
    q.push(s);
    while(not q.empty()) {
        int u = q.front(); q.pop();
        for(int v : adj[u]) {
            if(c[u][v] - f[u][v] > 0) {
                if(d[v] == -1) {
                    d[v] = d[u] + 1;
                    q.push(v);
                }
                if(d[v] == d[u] + 1) {
                    L_adj[u].insert(v);
                    L_adj_rev[v].insert(u);
                }
            }
        }
    }
    return d[t] != -1;
}

bool dfs(int u) {
    if(u == t) return true;
    for(int v : L_adj[u]) {
        if(dfs(v)) {
            p[v] = u;
            return true;
        }
    }

    // at this point, node u has no path to the sink; delete it from level graph
    for(int w : L_adj_rev[u]) {
        L_adj[w].erase(u);
    }
    return false;
}

bool find_aug_path() {
    memset(p, -1, sizeof p);
    p[s] = 0;
    return dfs(s);
}

int main() {
    memset(c, 0, sizeof c);
    memset(f, 0, sizeof f);
    cin >> n >> s >> t >> m;
    for(int i = 0; i < m; i++) {
        int u, v, capacity;
        cin >> u >> v >> capacity;
        adj[u].insert(v);
        adj[v].insert(u);
        c[u][v] += capacity;
    }
    int max_flow_value = 0;
    while(make_level_graph()) {
        while(find_aug_path()) {
            int b = INF;
            for(int v = t, u = p[v]; v != s; v = u, u = p[v]) b = min(b, c[u][v] - f[u][v]);
            for(int v = t, u = p[v]; v != s; v = u, u = p[v]) {
                if(c[u][v] - f[u][v] == b) { // delete bottleneck edges from the level graph
                    L_adj[u].erase(v);
                    L_adj_rev[v].erase(u);
                }
                f[u][v] += b, f[v][u] -= b;
            }
            max_flow_value += b;
        }
    }
    cout << max_flow_value << '\n';
    return 0;
}
```

Unfortunately, if you try using this implementation to solve UVa 820 - Internet Bandwidth, you will get TLE. Explicitly maintaining a level graph significantly degrades the running time. Instead, let's try to retrieve the level graph implicitly using only the distance information for each node. Implementing this properly is not trivial, and a buggy implementation can very easily make the running time degenerate to $\mathcal{O}(VE^2)$.

The idea for the implementation is, when we run DFS, we use the original adjacency list to get the neighbors of a node. However, to ensure that a neighbor v is actually a neighbor of u in the current level graph, we need to check three things.

1. The distances are correct, namely: $d_v = d_u + 1$.
2. We haven't yet deleted the edge $e = (u, v)$ from the level graph in a previous augmentation step. In other words, $c(e) - f(e) > 0$.
3. We haven't yet deleted node v from the level graph. We can delete a node by marking its distance to be some dummy value like -1, so that the first condition subsumes this one.

We ignore v if it fails to satisfy any of these three properties.

In addition to checking these conditions, we have to ensure that we don't repeatedly visit some edge or node that no longer exists in the level graph. Otherwise, all iterations of DFS will degenerate to $\mathcal{O}(E)$, causing the entire algorithm to degenerate to $\mathcal{O}(VE^2)$. Observe that once we discover a neighbor v that we should ignore, we should continue to ignore it for the rest of the augmentation phase. We can therefore maintain for each node u a pointer to its first neighbor that still satisfies the three conditions above. Initially, this pointer points to its first neighbor in the original adjacency list. Starting from node u, if we are in a lucky case, its neighbor v satisfies the three conditions above, and the DFS from its neighbor v succeeds. Otherwise, we are in an unlucky case. In this case, the neighbor v will never be a valid neighbor of u in the level graph until the end of the current augmentation phase, so we move the pointer of u to its next neighbor. This effectively deletes (u, v) from the level graph and helps us avoid repeated visits. Carefully read the code below for the details, and convince yourself that this implementation is $\mathcal{O}(V^2E)$.

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 100;
const int INF = 1'000'000;
int n, m, s, t;
vector<int> adj[MAX_N+1]; // revert to adj list to make it easy to keep track of ignored
↪ neighbors
int adj_ptr[MAX_N+1]; // the index of the first neighbor of a node which is not yet ignored
int c[MAX_N+1][MAX_N+1];
int f[MAX_N+1][MAX_N+1];
int d[MAX_N+1]; // distance for level graph
int p[MAX_N+1]; // parent for blocking flow
bool make_level_graph() {
    memset(d, -1, sizeof d);
    d[s] = 0;
    queue<int> q;
    q.push(s);
    while(not q.empty()) {
        int u = q.front(); q.pop();
        for(int v : adj[u]) {
            if(c[u][v] - f[u][v] > 0 and d[v] == -1) {
                d[v] = d[u] + 1;
                q.push(v);
            }
        }
    }
    return d[t] != -1;
}
bool dfs(int u) {
    if(u == t) return true;
    for(int &i = adj_ptr[u]; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if(d[v] == d[u] + 1 and c[u][v] - f[u][v] > 0 and dfs(v)) {
            // lucky case: we immediately return, and adj_ptr[u] remains untouched
            p[v] = u;
            return true;
        }
        // Unlucky case: Either edge (u,v) or node v doesn't exist in the level graph.
        // Moving to the next neighbor increments adj_ptr[u] along with i,
        // because we assigned i by reference.
        // This effectively removes v from the neighbor list of u until the end of the
        ↪ current phase.
    }
    // node u has no path to the sink, "delete" it from level graph
    d[u] = -1;
    return false;
}
bool find_aug_path() {
    memset(p, -1, sizeof p);
    p[s] = 0;
    return dfs(s);
}
int main() {
    memset(c, 0, sizeof c);
    memset(f, 0, sizeof f);
    cin >> n >> s >> t >> m;
    for(int i = 0; i < m; i++) {
        int u, v, capacity;
        cin >> u >> v >> capacity;
        // adj list will have duplicates, but this doesn't hurt running time too much
        // BFS for making the level graph will still be O(E)
        adj[u].push_back(v);
        adj[v].push_back(u);
        c[u][v] += capacity;
    }
    int max_flow_value = 0;
    while(make_level_graph()) {
        // with each new phase, the first non-ignored neighbor is
        // its first neighbor in the original adjacency list
        memset(adj_ptr, 0, sizeof adj_ptr);
        while(find_aug_path()) {
            int b = INF;
            for(int v = t, u = p[v]; v != s; v = u, u = p[v]) b = min(b, c[u][v] - f[u][v]);
            for(int v = t, u = p[v]; v != s; v = u, u = p[v]) f[u][v] += b, f[v][u] -= b;
            max_flow_value += b;
        }
    }
    cout << max_flow_value << '\n';
    return 0;
}
```

Interestingly, using a data structure called a *link-cut tree* (invented by Sleator and Tarjan in 1982), the time required to find a blocking flow in the level graph can be reduced from $\mathcal{O}(VE)$ to $\mathcal{O}(E \lg V)$, making the total time $\mathcal{O}(VE \lg V)$. With some additional techniques and data structures, each blocking flow can be found in $\mathcal{O}(E \lg(V^2/E))$, making the total time $\mathcal{O}(VE \lg(V^2/E))$. However, because the data structures are quite complicated and the constant factors in the running time are quite large, this is not used in practice. The best bounds are $E^{1+o(1)}$, due to an algorithm made by Chen, et. al. in 2022.[1] (Note how recent it is! Research on network flows is still very active.) But that algorithm is extremely complicated and impractical today. Let's now turn our attention to an algorithm of moderate difficulty to understand, and which works very well in practice.

2.5 Push-Relabel Approach to the Maximum Flow Problem

A radically different approach to the max flow problem was introduced by Goldberg and Tarjan in 1988, called the *push-relabel* approach. Similar to the Ford-Fulkerson algorithm, the basic skeleton of the approach is not an algorithm itself. There is a part which needs to be

[1]Be careful with this expression: the $o(1)$ hides a lot of nuance! For example, check that $\log^{10} E$ and $E^{1/\log\log\log E}$ are both $E^{o(1)}$.

specified in full, and the overall algorithm can easily be improved by simple tweaks to this part. Since then, a number of different approaches with strictly better asymptotic complexities have been invented, but they have not been proven to be more efficient in practice than push-relabel algorithms. Hence, push-relabel algorithms are still the gold standard for maximum flow. Similar to our discussion of the augmenting paths method, we will first describe and implement the simplest version, and add the improvements later.

Note: If you're learning flows for the first time, then you may choose to skip this part for now, since you already have enough prerequisites to tackle and appreciate maximum matching.

This section is under construction. Stay tuned for updates. In the meantime, if you use Edmonds-Karp or Dinic's algorithm, you should be fine.

Some points for now:

- We'll invent and maintain "potential values" for the nodes. These will dynamically change in the course of the algorithm, through the "relabel" operations.
- If we think of flow as some fluid and potential values as "heights", then we want our fluid flow to go from a higher to a lower location.
- The "push" operation pushes some fluid to a neighboring node.
- If some fluid gets stuck at a "valley" with no way out, we will increase the height of some nodes so that some fluid may "flow back".
- Because of the nature of height increases, we will only ever increase a node $\mathcal{O}(V)$ times, so there will only be $\mathcal{O}(V^2)$ height increases.
- With more arguments, one can prove that there will be $\mathcal{O}(V^2E)$ push operations.
- Each push and relabel operation can be done in $\mathcal{O}(1)$ time, so the running time is $\mathcal{O}(V^2E)$.
- This is asymptotically the same as Dinic's algorithm, and is also quite fast in practice since the operations are relatively simple.
- One downside is that you don't have valid flows at every step. You only get a valid flow once the algorithm terminates. So in problems where you need to update the graph (say, add some edges) and then compute the updated flow, using push-relabel may be tricky.

2.6 Practice Problems

Before moving on to the next section, I recommend practicing what you just learned with the following easy problems first: NOI.PH Training 2018 Advanced 1 Set 1 on ProgVar.Fun

Try to use all three algorithms (shortest augmenting paths, blocking flow, and pre-flow push-relabel) to solve each problem.

3 Bipartite Matching

Given an undirected graph $G = (V, E)$, a subset of edges $M \subseteq E$ is a **matching** if every node in V is incident to at most one edge in M.

Intuitively, let's say the nodes in the graph represent people, and the edges represent mutual romantic interest. A matching captures the notion of "monogamous relationships" within the group of people.

A natural problem is the following: given a graph, find a maximum-cardinality matching. That is, pair people up so that the maximum number of people end up happy.

This problem is solvable in $\mathcal{O}(V^2E)$ time using an algorithm due to Edmonds (the same guy who co-invented the shortest augmenting paths algorithm), published in 1965 as the "paths, trees, and flowers" method and known today as Edmonds' Blossom algorithm. This algorithm is somewhat complex and is out of our current scope.

Instead, we will focus our attention on bipartite graphs and study very simple algorithms to solve the maximum matching problem for this special case.

Recall that a **bipartite graph** is a graph where the set of nodes can be partitioned into two disjoint sets L and R such that there are no edges between any two nodes in L, and similarly for R. The **maximum bipartite matching problem** is to find a maximum-cardinality matching given a bipartite graph.

3.1 Alternating Paths (Kuhn) and Berge's Lemma

Let's try to come up with an intuitive algorithm for solving maximum bipartite matching. Start by randomly matching unmatched nodes in L to any of their unmatched neighbors in R. Keep going until no such matches can be made. The code would look something like this.

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_L = 500;
const int MAX_R = 500;
// since we are matching from L, we only need adjacency lists of the nodes in L
vector<int> adj[MAX_L+1];
// for each node in R, keep track of the node that was matched to it, or -1 if node is
↪ unmatched
int partner[MAX_R+1];
int main() {
    // read the input graph here
    memset(partner, -1, sizeof partner);
    for(int u = 1; u <= L_SIZE; u++) {
        for(int v : adj[u]) {
            if(partner[v] != -1) {
                partner[v] = u;
                break;
            }
        }
    }
    return 0;
}
```

Exercise 3.1. This simple algorithm does not always work. Find a counterexample.

Our simple algorithm got stuck because all of the neighbors of every unmatched node in L are already matched. To obtain a better matching, we need to match one of the unmatched nodes – call it u – in L with one of its neighbors – call it v – in R. The problem is, all of the neighbors in R are already matched. However, if we can "undo" the matching of v with its partner p_v, then we can add a new pair to our matching. It doesn't make sense for u to just "steal" v from its partner and leave p_v unmatched. The overall size of the matching would just be the same. It would only make sense to match u and v if we can somehow find a new partner for p_v. If there exists an unmatched neighbor of p_v, then we are done. Otherwise, we will need to try to "undo" the matching of one of the neighbors of p_v. Notice that we are exactly in the same situation as when we were trying to match u. We can now try to express this algorithm recursively:

- For every node in L, try to match it with one of its neighbors
- To match a node u in L with one of its neighbors...
  - If u has an unmatched neighbor v, then add $\{u, v\}$ to the matching.
  - Otherwise, all neighbors of u are matched. For every such neighbor v, recursively try to match its current partner p_v, *taking care not to undo any of the pending matches made in the current iteration.* If this succeeds for some neighbor v, then we can add $\{u, v\}$ to the matching. If none of the partners of any of the neighbors of u can be matched, then it is not possible to add u to the matching.

An implementation in C++ would look like the following:

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_L = 500;
const int MAX_R = 500;
vector<int> adj[MAX_L+1];
int partner[MAX_R+1];
bool matched_in_curr_iteration[MAX_L+1];
bool try_match(int u) {
    matched_in_curr_iteration[u] = true;
    for(int v : adj[u]) {
        int p_v = partner[v];
        if(p_v == -1 or (not matched_in_curr_iteration[p_v] and try_match(p_v))) {
            partner[v] = u;
            return true;
        }
    }
    return false;
}
int main() {
    // read the input graph here
    memset(partner, -1, sizeof partner);
    int matching_size = 0;
    for(int u = 1; u <= L_SIZE; u++) {
        memset(matched_in_curr_iteration, false, sizeof matched_in_curr_iteration);
        if(try_match(u)) {
            matching_size++;
        }
    }
    return 0;
}
```

Here's another way to look at our current algorithm. Notice that we are essentially trying to find a path from an unmatched node in L to another unmatched node in R, at every step alternating between edges not in the matching and edges in the matching to form the path. More formally, let $G = (V = L \cup R, E)$ be a bipartite graph and M be a matching in G. An **alternating path** P in G with respect to M is a path $u, \dots, v$ where $u \in L$, $v \in R$, $\{u, v'\} \notin M$ for all $v' \in G$, $\{u', v\} \notin M$ for all $u' \in G$, $\{P_k, P_{k+1}\} \notin M$ for all odd k, and $\{P_k, P_{k+1}\} \in M$ for all even k. The algorithm can now be stated in another way:

- Let M be initially empty.
- Find an alternating path P in G with respect to M.
- Update M using P. For every edge in P, if it is in M, remove it from M, otherwise add it to M. This operation is somewhat reminiscent of bitwise-xor, and is, in fact, sometimes denoted $M \oplus P$.
- Repeat until there are no more such paths.

To make this point even clearer, let's take the code above and perform some renaming.

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_L = 500;
const int MAX_R= 500;
vector<int> adj[MAX_L+1];
int partner[MAX_R+1]; // plays the role of determining edges that are or are not in the
↪ matching
bool visited[MAX_L+1];
bool dfs(int u) {
    visited[u] = true;
    for(int v : adj[u]) {
        int p_v = partner[v];
        if(p_v == -1 or (not visited[p_v] and dfs(p_v))) {
            partner[v] = u; // this is the "bitwise-xor" part
            return true;
        }
    }
    return false;
}
int main() {
    // read the input graph here
    memset(partner, -1, sizeof partner);
    int matching_size = 0;
    for(int u = 1; u <= L_SIZE; u++) {
        memset(visited, false, sizeof visited);
        if(dfs(u)) { // DFS to find an alternating path
            matching_size++;
        }
    }
    return 0;
}
```

This algorithm is often attributed to Kuhn, although I could not find any reliable historical sources.

Why does this algorithm work? There is a lemma, first proven by Berge in 1957, that easily implies the correctness of Kuhn's algorithm.

Lemma 3.2 (Berge's Lemma)**.** A matching M in a graph G is maximum if and only if there are no alternating paths in G with respect to M.

Berge's original proof was somewhat involved, as he was trying to prove it for general graphs. Since we are currently interested only in bipartite graphs, we will shortly see a much simpler proof. In fact, you might already know the proof without knowing it.

Exercise 3.3. What is the worst case running time of Kuhn's algorithm?

3.2 Reduction to Maximum Flow

At first glance, bipartite matching and maximum flow appear to be completely different problems. However, there is a very simple reduction from the former to the latter:

- Take the original undirected bipartite graph.
- Direct all the edges from L to R.
- Add a source node with directed edges to every node in L, one edge per node in L.
- Add a sink node with directed edges from every node in R, one edge per node in R.
- Assign each edge a capacity of 1.

We now have a flow network, and we can solve for the maximum flow of this network. The maximum flow value of this network is exactly equal to the size of the maximum matching. A saturated edge (u, v) where $u \in L$ and $v \in R$ in a maximum flow is equivalent to the presence of $\{u, v\}$ in the corresponding maximum matching. See II.8-10 for examples.

It should be quite obvious why this reduction works. You may even have seen it coming in from the discussion of Kuhn's algorithm and Berge's Lemma above. In fact, if you study Kuhn's algorithm carefully, you will see that it is basically equivalent to the Ford-Fulkerson algorithm applied to the special kind of flow network we just described above. See II.9-10 for a more formal proof.

Now that we understand the reduction to maximum flow, it is easy to see that Berge's Lemma is just a special case of the Max-Flow Min-Cut Theorem (Theorem 2.12).

3.3 Analysis of Dinic's Algorithm on Simple Unit-Capacity Networks and the Hopcroft-Karp Algorithm

Since we have shown that maximum bipartite matching is reducible to maximum flow, any algorithm for solving maximum flow can also be used to solve maximum bipartite matching.

Exercise 3.4. Does using the Edmonds-Karp shortest augmenting path algorithm for bipartite matching yield any significant improvement over Kuhn's alternating path algorithm? Why or why not?

It turns out that if we use Dinic's blocking flow algorithm, the running time reduces significantly to $\mathcal{O}(E\sqrt{V})$. To see why, note the following:

- Each phase of augmentation takes $\mathcal{O}(E)$ time.
  - It takes $\mathcal{O}(E)$ time to construct the level graph.
  - At every augmentation step, all the edges in the augmenting path are saturated and removed from the level graph.
  - Since there are only $\mathcal{O}(E)$ edges in the level graph, the total time of all the augmentation steps within the phase must be $\mathcal{O}(E)$.
- After $\sqrt{V}$ phases, the current value of the flow is at most $\sqrt{V}$ away from optimal.
  - After $\sqrt{V}$ phases, the length of the shortest augmenting path is $> \sqrt{V}$.
  - This implies that there are $> \sqrt{V}$ levels in the level graph.
  - Let h be the level with the least number of nodes. The number of nodes V_h in this level must be $\leq \sqrt{V}$.
  - Intuitively, if we place a "moat" around all nodes with level $\leq h$, the nodes in level h form the "bottleneck" of all flow going through this moat, and only at most $\sqrt{V}$ flow can further go out of this moat in succeeding augmentations. More formally, consider the cut formed by placing all nodes with level $< h$ and all nodes with level $= h$ and having at most one residual edge going out of it on one side, the rest of the nodes on the other. See I.94 for an illustration. Since, every node has either indegree 1 or outdegree 1 (or both), the nodes whose outgoing edges contribute to the capacity of the cut (i.e. the nodes on the "boundary" of this cut) all have outdegree 1, and the capacity of this cut must be $\leq V_h \leq \sqrt{V}$. Therefore, the flow value can only increase by at most $\sqrt{V}$ before all the edges crossing this cut are fully saturated, implying that the current flow value is at most $\sqrt{V}$ away from optimal.
- Since the flow value increases by 1 at every augmentation step, after $\mathcal{O}(\sqrt{V})$ steps after the $\sqrt{V}$ phases, the algorithm must terminate.

The same analysis holds for any *simple unit-capacity network*. A **simple unit-capacity network** is a flow network where each edge has capacity 1 and every node has either indegree 1 or outdegree 1 or both. Dinic's algorithm achieves the same performance in more general settings than just bipartite matching.

There is "another" $\mathcal{O}(E\sqrt{V})$ algorithm for maximum bipartite matching due to Hopcroft and Karp. However, if you study it carefully, it is basically the same as performing the flow network reduction and then applying Dinic's algorithm. So if you understand how Dinic's algorithm works and why it runs in $\mathcal{O}(E\sqrt{V})$ on simple unit-capacity networks, then you also already understand how the Hopcroft-Karp algorithm works.

3.4 Practice Problems

Before moving on to the next section, I recommend practicing what you just learned with the following easy problems first: NOI.PH Training 2018 Advanced 1 Set 2 on ProgVar.Fun

4 Disjoint Paths and Menger's Theorem

As another nice application of network flows, let's consider another problem that is again seemingly unrelated to network flows.

Let's call two paths **edge-disjoint** if they have no edges in common. The **edge-disjoint paths problem** is to find the maximum number of edge-disjoint paths between two specified nodes s and t of a given directed graph.

There is an easy reduction to maximum flow. Construct a flow network by assigning a capacity of 1 to every edge. Remove all edges going into s and all edges going out of t. Let s be the source and let t be the sink. The maximum number of edge-disjoint paths in the original graph is exactly equal to the maximum flow value in the corresponding flow network.

It is somewhat obvious why this reduction works, but let's try to prove it somewhat formally.

Proof. First, let's show that if we have k disjoint paths in the original graph, then we can also produce a flow with value k in the corresponding flow network. Note that if we have non-simple paths, we can always remove the cycles to get simple paths, without changing the number of disjoint paths we have. Also note that if we have a disjoint path that uses an edge e going into s, then we can remove all the edges from the beginning of the path up to e without changing the number of disjoint paths we have. We can do something similar for paths that use an edge going out of t. Now, assign a flow value of 1 to all edges that belong to some disjoint path, and assign a flow value of 0 to all edges that don't. Since the paths are edge-disjoint, there must be k edges out of s that are saturated in the flow network, and hence we have a flow with value k.

Then, the reverse direction: if we have a flow with value k in the corresponding flow network, then we can produce k disjoint paths in the original graph. Since we have a flow value of k, there are k edges out of s that are saturated in the flow network. Starting with each one, construct paths from s to t by following edges with flow value equal to 1 and which have not yet been used before. Such edges and such paths always exist by the flow conservation constraint. By construction the paths are all edge-disjoint, and there are k of them. □

Let's now turn our attention to yet another seemingly unrelated problem. Given a directed graph and two nodes s and t, find the minimum number of edges that need to be removed so that there are no paths from s to t.

And now, for the *big* surprise, first proven by Menger in 1927.

Theorem 4.1 (Menger's Theorem in Directed Graphs)**.** In a directed graph, the maximum number of edge-disjoint paths from some node s to some node t is equal to the minimum number of edges that need to be removed to disconnect t from s.

Like in the case of Berge's Lemma above, Menger's original argument was somewhat involved, but this theorem is now very easy to prove as a special case of the Max-Flow Min-Cut Theorem. (Theorem 2.12)

There are some natural variants of the problems above, with corresponding maximum flow reductions and corresponding versions of Menger's Theorem.

If we were interested in finding edge-disjoint paths in an undirected graph instead, we can still reduce it to maximum flow by representing edge undirected edge in the original graph with two anti-parallel unit-capacity edges in the corresponding flow network (ignoring the edges coming into s and the edges going out of t). This doesn't seem to work at first glance, because two paths may be edge-disjoint in the directed graph version of the original graph, yet turn out to share an edge in the original graph, if the two paths each use one member each of a pair of anti-parallel edges. It turns out that if we do end up with a maximum flow that saturates both members of a pair of anti-parallel edge, we can make them "cancel" each other out, that is, never use either member when constructing the disjoint paths from the flow, without reducing the number of disjoint paths we can form.

Exercise 4.2. Prove that this indeed can be done. Hint: Assume that in our construction, there are two paths P and P' where edge (u, v) is in P while edge (v, u) is in P'. Recalling how "backward" pushes work in residual graphs (subsection 2.2), split each path into two sub-paths and merge each sub-path with a different sub-path than the one it was originally attached to. Complete the proof by induction.

From our reduction, it should be obvious why the following is true:

Theorem 4.3 (Menger's Theorem in Undirected Graphs)**.** In an undirected graph, the maximum number of edge-disjoint paths from some node s to some node t is equal to the minimum number of edges that need to be removed to disconnect t from s.

Problem 4.1. Two paths are **node-disjoint** if they have no nodes in common. The **node-disjoint paths problem** is to find the maximum number of node-disjoint paths between two specified nodes s and t of a given graph (ignoring s and t when considering node-disjointness). State and prove corresponding versions of Menger's Theorems for node-disjoint paths and node-connectivity in both directed and undirected graphs. You will need to find a way to reduce the node-disjoint paths problem to maximum flow first.

Problem 4.2. Give tight bounds on the running time of Dinic's algorithm on the edge-disjoint paths problem.

5 Kőnig's Theorem and Special Cases of NP-Complete Problems Reduced to Bipartite Matching

5.1 Vertex Cover

Now we turn to another problem seemingly unrelated to network flows.

Given a graph $G = (V, E)$, a subset of vertices $C \subseteq V$ is a **vertex cover** if every edge in E is incident to at least one vertex in C.

Intuitively, consider the following scenario. There are roads and intersections. Security cameras can be put only at intersections. Placing a camera at an intersection allows all the segments of roads meeting at that intersection to be guarded. The idea is to place cameras so that all roads are fully guarded.

A natural problem is the following: given a graph, find a minimum-cardinality vertex cover. That is, place the fewest number of cameras so that all roads are fully guarded.

In general, this problem is hard. That is, it has no known algorithm that runs in polynomial time. In technical terms, it is *NP-complete*.

However, for bipartite graphs, it turns out to be solvable by reducing the problem to bipartite matching. This fact was first proven by Kőnig in 1931.

5.2 Kőnig's Theorem

Theorem 5.1 (Kőnig's Theorem)**.** In a bipartite graph, the size of the maximum-cardinality matching equals the size of the minimum-cardinality vertex cover.

Exercise 5.2. Prove that $|M| \leq |C|$ for any matching M and any vertex cover C in a general graph. (There is a weak duality between matchings and vertex covers.)

Similarly to what we did for the Max-Flow Min-Cut Theorem, we can finish the proof of Kőnig's Theorem by showing an algorithm that, given a bipartite graph, explicitly constructs a vertex cover whose size is equal to size of some matching.

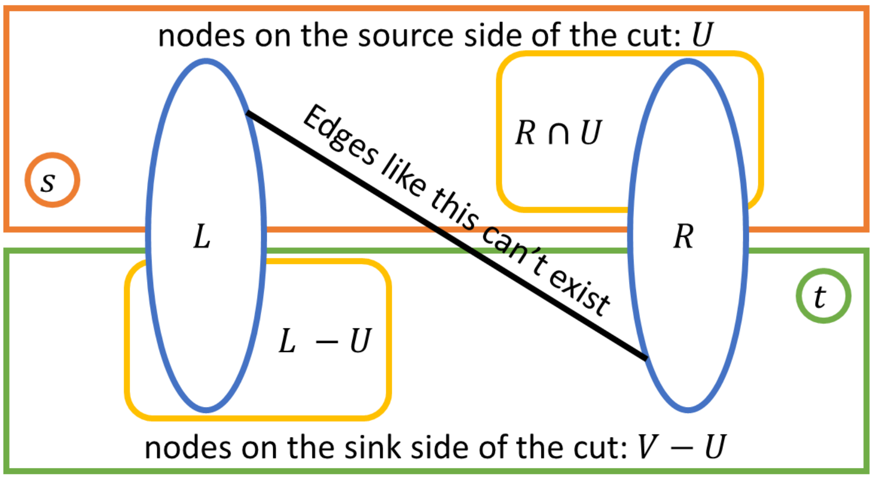
Lemma 5.3 (Kőnig's Algorithm)**.** Let $G = (V = L \cup R, E)$ be a bipartite graph and M be a maximum matching of G. Let U be the set of unmatched nodes in L, plus all the nodes reachable from these nodes via alternating paths in G with respect to M. Then the set C of unreachable nodes in L together with the reachable nodes in R, that is, $C = (L - U) \cup (R \cap U)$, is a vertex cover. Furthermore, $|C| = |M|$.

This construction is quite nasty and confusing when you first see it, so let's break it down. The way I personally find easiest to understand it is via the maximum flow (minimum cut) reduction for bipartite matching. From this point of view, the above algorithm works as follows:

- Find the minimum cut (A, B) of the flow network equivalent of the given bipartite graph. Notice that U is just $A - \{s\}$, where s is the source node of the flow network. In other words, U is the set of nodes still reachable from the source in the residual graph after the maximum flow algorithm terminates.
- From the left side of the bipartite graph, take all those nodes which are also on the sink

side of the minimum cut: $L \cap B = L - U$.

- From the right side of the bipartite graph, take all those nodes which are also on the source side of the minimum cut: $R \cap A = R \cap U$.
- The two combined form the vertex cover: $C = (L - U) \cup (R \cap U)$. In summary: *unreachable vertices in L and reachable vertices in R.*



nodes on the source side of the cut: U
s
L
Edges like this can't exist
$R \cap U$
R
t
$L - U$
nodes on the sink side of the cut: $V - U$

Figure 1: Sketch of Kőnig's Algorithm.

Before proving that this algorithm works, let's first make the following observation:

Lemma 5.4 (Matched Nodes on Same Side of the Cut)**.** For every edge $e = \{u, v\} \in M$, $u \in U$ if and only if $v \in U$.

To see why this is true, first notice that if $v \in U$, then since $\{u, v\} \in M$, the alternating path ending at v can be extended to reach u as well. Hence $u \in U$.

Then, notice that if $u \in U$, then since it is not an unmatched node, it must be reachable by some alternating path from an unmatched node in L. Since $u \in L$, the only way it can be reached is through the edge $\{v, u\}$. This implies that if $u \in U$, then it must be the case that $v \in U$ also.

Another way to see what this lemma is saying is to look at it in the context of network flows: every edge in the matching must be on the same side of the minimum cut of the network flow reduction of the bipartite graph.

Exercise 5.5. Prove the following corollary.

Corollary 5.1 (At Most One Member of a Matched Pair Is in the Cover)**.** For every edge $e = \{u, v\} \in M$, if $u \in C$ then $v \notin C$ and if $v \in C$ then $u \notin C$.

We are now ready to prove the correctness of Kőnig's Algorithm. First, let's show that C is indeed a vertex cover. We can consider four types of edges:

- Edges between a node in $L - U$ and a node in $R \cap U$. Clearly, these are covered by nodes in C.

- Edges between a node in $L - U$ and a node in $R - U$. These are covered by nodes in $L - U$.
- Edges between a node in $L \cap U$ and a node in $R \cap U$. These are covered by nodes in $R \cap U$.
- Edges between a node in $L \cap U$ and a node in $R - U$. Notice that these are the only edges which cannot possibly be covered by C. However, such edges cannot actually exist.
  - If such an edge $e = \{u, v\}$ existed, then it cannot be in M by Lemma 5.4.
  - If such an edge $e = \{u, v\}$ existed and it is not in M, then the alternating path ending at u can be extended through e to include v, and v must be in U. This contradicts the definition of e.

To see why $|C| = |M|$, observe the following:

- Because C is a vertex cover, for every edge e in the matching, at least one of the nodes incident to e is in C.
- By Corollary 5.1, at most one of the nodes incident to it is in C.
- No unmatched nodes are included in C.
  - By definition, unmatched nodes in L are in U and therefore excluded from C.
  - If an unmatched node in R is in U, then there is an alternating path to an unmatched node. This means that the size of the matching can be increased, contradicting the maximality of M. Therefore, all unmatched nodes in R are not in U and therefore excluded from C.

In summary, we have that for every edge in M exactly one of its incident nodes in C, and that only nodes incident to some edge in M are in C. Therefore $|C| = |M|$.

A final note: Kőnig's Theorem relies on an explicit construction which can only be done on bipartite graphs. It does not hold for all graphs in general. This is the reason why vertex cover is solvable in polynomial time for bipartite graphs, but it is still unknown whether the same can be achieved for all graphs in general. Most computer scientists believe (but cannot prove) that the answer to this question is negative.

Exercise 5.6. Construct a graph whose maximum matching is not of the same size as its minimum vertex cover.

5.3 Independent Set

Yet another seemingly unrelated problem.

Given a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is an **independent set** if no two nodes in S are adjacent to each other. Naturally, we want to find the maximum-cardinality independent set.

Now we have another *surprising* result.

Theorem 5.7 (Vertex Cover and Independent Set Are Complements)**.** A set of nodes S is an independent set in a graph $G = (V, E)$ if and only if its complement $V - S$ is a vertex cover.

Proof. S is an independent set.

$\iff$ For all edges $e = \{u, v\}$, at most one of u and v is in S.

$\iff$ For all edges $e = \{u, v\}$, at least one of u and v is in $V - S$.

$\iff$ $V - S$ is a vertex cover. □

This result applies to all graphs, not just to bipartite graphs. It easily implies that the complement of a minimum vertex cover is a maximum independent set.

In general, since the minimum vertex cover problem is NP-complete, the theorem above shows that the maximum independent set problem is also NP-complete.

However, for bipartite graphs, since we already have an efficient algorithm for finding minimum vertex covers, we can also easily find the maximum independent set.

5.4 Minimum Path Cover

Yet another seemingly unrelated problem.

Given a graph $G = (V, E)$, a **path cover** is a set P of node-disjoint[2] paths such that every node in V belongs to at least one path in P. Naturally, we want to find a minimum-cardinality path cover.

In general, the minimum path cover problem is NP-complete.

However, for *directed acyclic graphs*, there is a simple reduction to bipartite matching:

- Let $V' = L \cup R$, where L is a set of copies of vertices in V with positive outdegree and R is a set of copies of vertices in V with positive indegree. Note the word "copies": nodes that have both positive outdegree and positive indegree must appear twice in V', on opposite sides of the bipartition.
- Let E' be the set of edges $\{u', v'\}$, where u' is the copy of u in L and v' is the copy of v in R, for every edge (u, v) in E.
- Find a maximum matching in the bipartite graph $G' = (V', E')$.
- If the size of the maximum matching is $|M|$, then the size of the minimum path cover is $|V| - |M|$.
  - A matching of size 0 means that every node in V needs its own path.
  - Every edge $\{u', v'\}$ in the matching implies that node v can be covered by extending the path ending at u.

[2]Some versions of the path cover problem don't have this node-disjointedness condition.

6 Practical Tips for Programming Contests

- It is obviously not important to memorize who invented or proved what and when. I just put all the historical remarks to help you appreciate the rich history of the subject of network flows, one that is still an actively ongoing subject of research even about one century later after the first few theorems were proven.
- You shouldn't need to memorize Dinic's algorithm or the pre-flow push-relabel algorithm, but you should know how to code Edmonds-Karp for general graphs and Ford-Fulkerson AKA Kuhn's algorithm for bipartite graphs. A sufficiently determined (and cruel) problem setter might create cases where Edmonds-Karp or Kuhn's algorithms fail, but this is more common in Codeforces rounds and in the ICPC. It never happens in Google Code Jam and it shouldn't happen in the IOI.
- Network flows can be quite overwhelming for a beginner because of the number of available algorithms and the sheer number of possible applications of network flows. If you feel a bit overwhelmed, just pick one sufficiently fast algorithm (Edmonds-Karp should be ok) to master, and then focus on how to recognize when flows/bipartite matching can be applied. It's also quite important to be familiar with all the common problems that reduce to bipartite matching.
- It can be quite hard to recognize when network flows can be used to tackle a problem. This is all the more true when the reduction is to minimum cut rather than to maximum flow. (Personally, I feel those kinds of problems are the most beautiful kinds of applications of flows.) Some hints would be the following:
  - The input size constraints are small enough for cubic time algorithms to work. Since the most common network flow algorithms are roughly cubic time, having input sizes this small is a signal that network flows might be a viable approach. Be careful with relying on this hint too much, as some problem setters can deceptively make the input size small enough for a cubic time algorithm to work, even when a completely different approach (e.g. greedy, DP, binary search) is called for. This heuristic also fails for problems where the input size is much larger and Dinic's/Hopcroft-Karp algorithm is called for (in Codeforces rounds and in the ICPC).
  - There is some kind of partition into two sets involved in the problem. Sometimes this is a hint towards what the two sides of a cut should represent, sometimes towards what the two parts of a bipartite graph should represent.

  Often, it just boils down to intuition, and this intuition can only be developed by seeing several relevant problems, like the ones below.

7 Problems

Easier ones:

S1.1 SPOJ FASTFLOW - Fast Maximum Flow (purely for testing your Dinic's/pre-flow push-relabel implementations)

S1.2 SPOJ MATCHING - Fast Maximum Matching (purely for testing your Dinic's/Hopcroft-Karp implementations)

S1.3 NOI.PH Training 2018 Advanced 1 Set 3 on ProgVar.Fun

Harder ones:

S2.1 NOI.PH Training 2018 Advanced 1 Set 4 on ProgVar.Fun

S2.2 Google Code Jam 2014 Round 2, Problem C - Don't Break the Nile

S2.3 Google Code Jam 2015 Round 2, Problem C - Bilingual

S2.4 Google Code Jam 2017 Round 2, Problem D - Shoot the Turrets

S2.5 Google Code Jam 2017 Round 3, Problem B - Good News and Bad News

S2.6 Google Code Jam 2017 World Finals, Problem A - Dice Straight

S2.7 Google Code Jam 2018 Round 2, Problem C - Costume Change