LECTURE 1 and 2: Flows

1 Max Flow

1.1 Ford-Fulkerson, max-flow min-cut

Chapter 6, Sections 1,2,3 and Chapter 9, Sections 1,2

Max Flow.
Instance: Directed graph $N = (V, A)$, two nodes $s, t \in V$, and capacities on the arcs $c : A \to \mathbb{R}_+$. A flow is a set of numbers on the arcs such that for each node apart from s and t, the sum of the numbers on incoming arcs is equal to the sum of the numbers on outgoing arcs, and such that each number on an arc is non-negative and not higher than the capacity of the arc. The size of an (s, t)-flow is the sum of the numbers on the arcs leaving s (which is equal to the sum of the numbers on the arcs entering t).
Question: Find a maximum (s, t)-flow in D.

I assume that all of you have seen the Ford-Fulkerson algorithm, at each iteration finding a flow-augmenting path. Let me quickly repeat it. Given a flow $f : A \to \mathbb{R}_+$ with f_{ij} the flow on the arc (i, j) we construct a *residual digraph* R with arc set A^R and with capacities c^R. We assume that if $(i, j) \in A$ and $(j, i) \in A$ then $f_{ij} = 0$ or $f_{ji} = 0$ or both.

- If $f_{ij} < c_{ij}$, then the *forward arc* $(i, j) \in A^R$ and $c^R_{ij} = c_{ij} - f_{ij}$;
- If $f_{ij} > 0$, then the *backward arc* $(j, i) \in A^R$ and $c^R_{ji} = f_{ij}$.

(During the lecture I give an example.) An (s, t)-path P in R is a flow-augmenting path in N. The flow is augmented by adding $\min_{(u,v)\in P} c^R_{uv}$ to the flow to f_{ij} if (i, j) is a forward arc on P, and by distracting $\min_{(u,v)\in P} c^R_{uv}$ from flow f_{ij} if (j, i) is a backward arc on P. It implies that either the flow on one of the arcs in A reaches its capacity and will disappear from R in the next iteration, or the flow on some arc (i, j) is pushed back to 0, by which the reverse arc (j, i) disappears from R in the next iteration. Adapt R and the capacities of the arcs in A^R and reiterate. This is continued until the auxiliary graph does not have a path from s to t. In this case a maximum flow has been found.

There is a famous and beautiful duality result related to the max-flow problem, which proves correct termination of Ford-Fulkerson's Algorithm if no augmenting path is found: the so-called *max flow min cut* theorem.

An *s-t-cut* in a network is a subset S of the nodes V, such that $s \in S$ and $t \notin S$. The capacity of cut S is

$$C(S) = \sum_{\{(i,j)\in A | i\in S, j\notin S\}} c_{ij}.$$

A *minimum cut* is a cut of minimum total capacity.

Theorem 1 (Max Flow Min Cut Theorem). *The value of a maximum flow is equal to the value of the minimum cut capacity.*

Proof. I assume there is a finite maximum flow. Let f be any flow with value $F = \sum_{(s,v)\in A} f_{sv}$. That $F \leq C(S)$ for every cut S, is evident since any flow has to cross an arc from S to $V \setminus S$.

Given an optimal flow f there does not exist any augmenting path. Thus, in the residual graph of this flow t is not reachable from s. Consider the set S of reachable nodes. Then in the residual graph there does not exist an arc from any $i \in S$ to any $j \notin S$, otherwise j would have belonged to S. This means that $f_{ij} = c_{ij}$ for all $\{(i,j) \in A \mid i \in S, j \notin S\}$ (the forward arcs from nodes in S to nodes in $V \setminus S$ do not exist in the residual graph) and $f_{ij} = 0$ for all $\{(i,j) \in A \mid i \notin S, j \in S\}$ (the backward arcs from nodes not in S to nodes in S do not exist in the residual graph). Thus, all the flow crossing from S to $V \setminus S$ does not return to S, hence must exit through t. Thus $F = C(S)$.
Given that we just proved that for any flow, thus also for the maximum flow, and any cut, $F \leq C(S)$ it means that the cut S found in the residual graph of a maximum flow is a minimum cut. □

In the proof of the Max Flow Min Cut Theorem we have shown that if Ford-Fulkerson's Algorithm terminates then it terminates with a maximum flow.

Theorem 2. *A flow $f : A \to \mathbb{R}_+$ is a maximum flow if and only if in the residual graph there does not exist an (s,t)-path.*

Let us first become more precise about the *running time* the Ford Fulkerson algorithm takes.

For that we first analyse the running time of one iteration of Ford-Fulkerson's algorithm and we express running time as a number of elementary computer operations required. Elementary computer operations are e.g. addition, multiplication and comparison of two numbers, deletion of an item from a list, adding an item to an unordered list.

In an iteration of Ford Fulkerson's algorithm we start for a search in the residual network if there is a path from s to t. This is an augmenting path in the original network. We use Section 9.1 in [PS] to show that this search can be done in $O(|A|)$ operations. $|A|$ is used to denote the cardinality of the set A: the number of elements of A. The notation $O(g(n))$ is used for a function of n to indicate an upper bound on the rate with which the function grows with n.

- We write $f(n) = O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $c \in \mathbb{R}$;
- We write $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Thus, $O(|A|)$ says that the search for a path from s to t takes a number of operations that is asymptotically bounded from above by a linear function of the number of edges (arcs) in the graph.

If such a path is not found, the vertices reached from s form a min cut in the graph, as just proved, and we are done. If a path is found we need to update the residual graph. Notice that an s-t path does not contain more than $|V|$ edges. This means that we have to adapt the capacities on at most $2|V|$ edges of the residual graph, given the residual graph at the beginning of the next iteration.

After having constructed the residual graph we enter the next iteration. Therefore, overall, one iteration of the algorithm takes $O(|A| + |V|) = O(|E|)$ elementary computer operations (time).

Thus, Ford-Fulkerson's algorithm takes a total time $O(|A|+|V|) = O(|A|)$ times the number of iterations required to terminate. Unfortunately, the algorithm may not stop, but instead, only converge, and in this case it may even converge to a non-optimal flow. For those of you who are interested, read for yourself the example in [PS] Section 6.3. This is a problem that we do not encounter if all capacities are integer or rational numbers. The following is a simple example with integer capacities on which the algorithm takes an excessive number of iterations.

Consider the network with 4 nodes, s,u,v and t. We have the following arcs with their capacities: $c(s,u) = c(s,v) = c(u,t) = c(v,t) = M$ and $c(u,v) = 1$ (see for a picture Figure 9-8 in Section 9.2 in [PS]). In the first iteration the residual network is just the given network and it may be that the path found is (s,u,v,t) on which we can augment the flow by 1: $f(s,u)=f(u,v)=f(v,t)=1$. This gives the residual graph with forward arcs and capacities $c(s,u)=c(v,t)=M-1$, and backward arcs with capacities $c(u,s)=c(t,v)=c(v,u)=1$. In the next iteration in R the (s,t)-path (s,v,u,t) may be found, again allowing to augment the flow by 1. This continues for $2M$ iterations until we reach the optimum solution $f(s,u)=f(u,t)=f(s,v)=f(v,t)=M$ and $f(u,v)=0$. Clearly, we could have found this solution in only 2 augmentation steps, if we would have been more lucky.

Let me argue why we think that the running time of the algorithm on this example is considered excessive.

1.2 Polynomial running time

Chapter 8, Sections 1,2,3,4,5.

We propose a mathematically rigorous definition of *efficient algorithm*, which we will use later in developing Complexity theory:

Definition. An efficient algorithm for problem Π is an algorithm that solves *every* instance of Π in a number of elementary computer operations that is bounded by a *polynomial function* of the *size of the instance.*

Some phrases in this definition are emphasized. The algorithm is to solve *every* instance, even the hardest instance. Thus, efficiency is a *worst-case* notion.

I hope everyone knows the definition of polynomial function: f is polynomial if there exist a finite k and constants a_i, b_i, $i=1,\ldots,k$ such that

$$f(x)=\sum_{i=1}^{k} a_i x^{b_i}.$$

Non-polynomial functions are exponential functions, like

$$\begin{aligned} f(x) &= 2^x \\ f(x) &= (x\log x)^x \\ f(x) &= 10^{-28}3^{\sqrt{x}} \\ f(x) &= x! \end{aligned}$$

We call $f(x)=x^{\log x}$ pseudo-polynomial.

The *size of an instance* is defined as the number of bits that are needed in a digital computer to store all relevant data of the instance. Most of the time we can rely on an intuitive definition of the size, like in the TSP the number of points, or in Max Flow the number of vertices and the number of arcs of the graph. But let me once show you what the size of a Max Flow instance is according to the official definition.

The *size of an instance* is defined as the number of bits that are needed in a digital computer to store all relevant data of the instance. Most of the time we can rely on an intuitive definition of the size, like in the TSP the number of points, or in Max Flow the number of vertices and the number of arcs of the graph. But let me once show you what the size of a Max Flow instance is according to the official definition.

For each of n vertices we need $\log n$ bits. For each of the m arcs (i, j) we need $2 \log n + 3$ bits (the 3 extra to separate the pair and show they belong together). Finally we have to specify the capacities on the edges' Each capacities u_{ij} is specified in $\log u_{ij}$ bits (where I assume integer capacities). Thus the total length of the input is:

$$N = n \log n + m(2 \log n + 3) + \sum_{(i,j)\in A} \log u_{ij}.$$

Clearly if we have an algorithm with running time bounded by a polynomial function of n then the running time will also be bounded by a polynomial function of N, and the other way round. Thus, in fact for determining efficiency we could as well take n as size of the input. Sometimes the input size due to representation of numerical values like weights plays a crucial role. In that case we should take here $N = O(n^2(\log n + \log U))$, where $U = \max_{(i,j)\in A} u_{ij}$.

Why do we draw the line of efficiency between polynomial time and exponential (non-polynomial) time. We may illustrate this with a table that shows what technological improvement in computer speed would give us in terms of problem size that we can handle in one second.

	Now	$100\times$ faster	$1000\times$ faster
n^5	N_1	$2.5N_1$	$4N_1$
2^n	N_2	$N_2 + 7$	$N_2 + 10$

Of course, some polynomial functions like $f(n) = n^{19}$ is for moderate values of n greater than $g(n) = 2^n$. But already for $n = 200$ $g(n) > f(n)$. For most problems that are solvable by efficient algorithms the running time is usually not a high power of n; mostly 4 or less and exceptionally 6. Similarly, exponential running times of $O(3^n)$ hardly occur. Mostly it is $O(2^n)$.

We will most often speak of efficient algorithms as *polynomial time algorithms*. Based on the existence or non-existence of such algorithms we make a distinction between *easy* or *well-solved* problems and *hard* problems.

So, why are we not satisfied with Ford-Fulkerson's algorithm on instances with integer capacities? Therefore we should realise that to store an integer number a in a digital computer we need $\lceil \log a \rceil$ bits. The input of the problem is given by 4 numbers representing the vertices. This requires 2 bits per number in a digital computer. The capacity 1 requires 1 bit only. The capacity M requires $\log M$ bits. Thus, the digital size of this problem is bounded by $N = 2\log M$ for M large enough. But this means that $2M$ iterations is $O(2^N)$ iterations, and therefore the algorithm runs in $O(2^N|A|)$ time.

1.3 An efficient algorithm for max flow

Chapter 9, Section 4.

We propose now to be more careful in selecting the (s,t)-paths in the residual graphs. Just simply always select the shortest (s,t)-path, where with "shortest" we mean here the path with the smallest number of arcs on it. This would clearly have solved our bad example in just 2 iterations. Let us show that it is not only better for this example.

Given flow f and corresponding residual graph R. Calling s a level-0 node, we call all nodes reachable in one step from s level-1 nodes, etc. In this way in $O(|A_R|) = O(|A|)$ steps we find out that node t is a level-k node, meaning that the shortest path from s to t contains k arcs in R. We keep all vertices and arcs that are on any length-k path from s to t in R and call the resulting restricted graph R_r. Each arc in R_r goes from a level-i node to a level-$i+1$ node, for some i.

Take one such an (s,t)-path in R_r. Then after augmentation one of the arcs will have disappeared from R_r, either by saturation or by emptying. It could happen that the reverse arc comes in its place, but this goes from a higher-level node to a lower-level node, and hence is not part of the new

R_r. Thus after at most $|A_{R_r}|$ augmentations R_r will be empty, implying that given the flow found at that point there are no (s,t)-paths in the residual network of length at most k.

This implies that in the most straightforward implementation, in which we rebuild R_r only once, but do a path search every iteration, we require $O(|A_{R_r}||V_{R_r}|) = O(|A||V|)$ operations for augmenting along all (s,t)-paths of length k.

Clearly, we will need to consider at most $|V|$ different lengths of augmenting paths. Thus, overall the algorithm takes $O(|A||V|^2)$ time. This can be slightly improved to $O(|V|^3)$ time as in [PS] Section 9.4. They augment over several paths in one round, in any such round saturating all incoming arcs or all outgoing arcs of some node, thereby cutting out the node from R_r (read the details yourself).

I advise you to make Exercise 9 of Chapter 9 as an alternative proof of the $O(|A||V|^2)$ running time.

1.4 Variations

Suppose that i.o. 1 source and 1 sink, we have 2 sources s_1 and s_2 and 2 sinks t_1 and t_2.

Exercise 2.1. Propose a way to solve this variation of the max-flow problem.

Suppose now that the 2 sources and sinks are pairwise related in the sense that the flow going out from source s_1 is to be directed only to sink t_1, and the flow going out of source s_2 is to be directed only to sink t_2.

Exercise 2.2. How would you solve this variation of the max-flow problem?

1.5 Material and Exercises

[PS] Chapter 9: Sections 9.1-9.4 for flows;

Exercises:
From Chapter 9: Exercises 4, 9, 11