CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 23

November 26, 2019

Instructor: Sepehr Assadi

1 Network Flow: Recap

Recall the network flow problem defined in the previous lecture:

Problem 1 (**Network Flow/Maximum Flow**)**.** Given a network $G(V, E)$ with source s and sink t and capacity c_e on every edge $e \in E$, find a flow function f with maximum value, namely a *maximum flow*. A flow function is any function $f : V \times V \to \mathbb{R}^+$ that satisfy the capacity constraints ($f(u, v) \leq c_e$ for $e = (u, v)$) and preservation of flow (flow into a vertex other than s, t is equal to flow out of that vertex).

In the previous lecture, we also mentioned the following algorithm (which we will use as a black-box) for the network flow problem:

Ford-Fulkerson Algorithm: There is an algorithm for the network flow problem that runs in $O(m \cdot F)$ time where F is the value of maximum flow in the network.

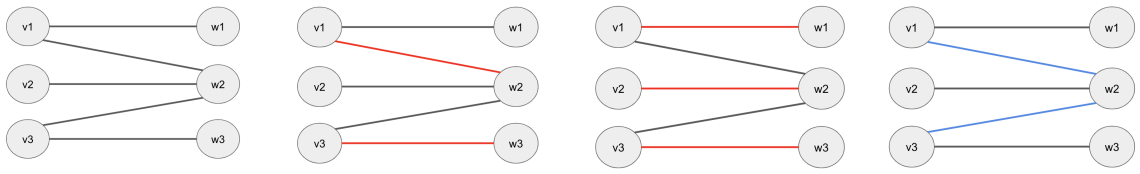
2 Some Applications of Network Flow

2.1 Bipartite Matching Problem

Suppose we have a set of n buyers and a set of n items. Each buyer i is interested in buying exactly one item from a subset S_i of all n items and each item can be sold to at most one buyer. Can we find a way of selling all the n items to the n buyers? We can model this problem as the following graph problem:

Problem 2 (**Bipartite Matching**)**.** We are given a graph $G(V, E)$ where the vertices of V can be partitioned into two sets L and R such that every edge of G is between a vertex in L and a vertex in R (i.e., there are no edges with both endpoints in L or in R). Such a graph is called *bipartite*. We define a *matching* M to be any subset of edges that *share no vertices*. A *maximum matching* is then any matching M with the largest number of edges. See Figure 1 for an illustration.

In the bipartite matching problem, we are given a bipartite graph $G(V, E)$ and the goal is to find a maximum matching M of G.



(a) A bipartite graph (b) A matching (c) A maximum matching (d) Not a matching

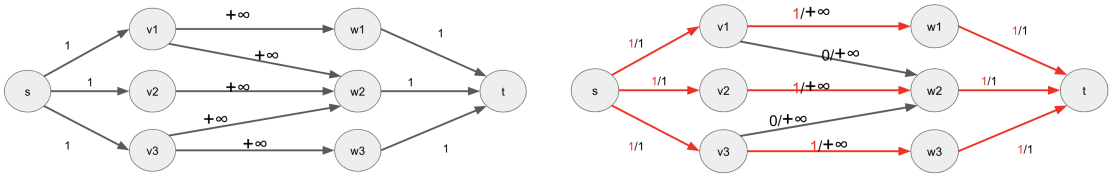
Figure 1: An illustration of bipartite matching problem

The bipartite matching problem above captures the buyer-item scenario as follows: We create a bipartite graph $G(V, E)$ with vertices in L corresponding to buyers and vertices in R corresponding to the items; we then connect every buyer-vertex i in L to every item-vertex in S_i in R. It is immediate to see that any matching in G corresponds to an assignment of (some of) items to (some of) buyers so that no item is given to more than one buyer and no buyer is given more than one item. Hence, by solving the bipartite matching problem over graph G, we can decide whether or not we can sell all the items to the buyers.

Solving bipartite matching using network flow. We now use a graph reduction to solve the bipartite matching problem using *any* algorithm for the network flow problem. In order to do this, we should show that given any bipartite graph $G(V, E)$, we can turn it into a flow network $G'(V', E')$ with a source s and sink t, such that finding the maximum flow in G' allows us to find a maximum matching in G.

Reduction: Given the bipartite graph $G(V, E)$ in the bipartite matching problem (we assume bipartition of G is $L \cup R$, i.e., $V = L \cup R$ and edges in E are between L and R only), construct the following flow network $G'(V', E')$ with source s and sink t and capacity c_e on each edge $e \in E'$ (see Figure 2 for an illustration of this reduction in the context of example given in Figure 1):

- We set $V' = V \cup \{s, t\}$ for two new vertices s and t. We add a *directed* edge (u, v) to E' whenever a vertex $u \in L$ of G has an edge to $v \in R$ of G (note that the edges in G are undirected while edges in G' should be directed since G' is a network). Moreover, for any vertex $u \in L$ we add an edge (s, u) to E' and for any vertex $v \in R$, we add an edge (v, t).
- We set the capacity of any edge $e = (s, u)$ for $u \in L$ to be one, i.e., $c_e = 1$. Similarly, we set the capacity of any edge $e' = (v, t)$ for $v \in R$ to one also, i.e., $c_{e'} = 1$. We set the capacity of all remaining edges to be $+\infty$ (we could have also set them equal to one instead).



(a) The network created for the graph in Figure 1

(b) A maximum flow in this network

Figure 2: An illustration of the reduction for the bipartite matching problem

We then find a maximum flow f from s to t in the network G'. We create a matching M in the original graph G by picking any edge $\{u, v\}$ for $u \in L$ and $v \in R$ to be added to M if and only if $f(u, v) = 1$ in this flow (note that we are only doing this for edges between L and R and are ignore the outgoing edges of s and incoming edges of t in G' as these edges do not even belong to the graph G).

Proof of Correctness: We first have to prove that the set of edges M found by the algorithm is indeed a matching and then prove its size is maximum.

- M is a matching: For M not to be a matching, a vertex v should be shared by more than one edge of M. However, this cannot happen because if $v \in L$, then only one unit of flow can ever each v (since capacity of (s, v) edge is one and the graph is acyclic) and so at most one outgoing edge of v can have flow equal to one and hence be added to M; similarly, if $v \in R$, then only one unit of flow can ever go out of v (since capacity of (v, t) edge is one) and so at most one incoming edge of v can have flow equal to one and be added to M.
- M is a *maximum* matching: We prove that any flow f in G' corresponds to a matching M in G with size of M equal to the value of flow f and vice versa. This then implies that maximum flow corresponds

to a maximum matching and thus M is a maximum matching. By the above part, we know that for any flow f in G' there is a matching M of the same size in G. We now prove the other direction.

Fix any matching M in G and consider the flow f where for any edge $\{u, v\}$ in M, we add a *flow path* $f(s, u) = 1$, $f(u, v) = 1$, and $f(v, t) = 1$ to our flow f. This definition of f clearly satisfies the preservation flow. Moreover, since by definition of a matching M, the vertices u, v only appear once in the matching, we have that f also satisfies the capacity constraints and hence is indeed a flow. The value of this flow also is equal to $\sum_{v \in L} f(s, v)$ which is equal to the size of M by definition.

This concludes the proof of correctness of the algorithm.

Runtime analysis: We can create the network above in $O(n + m)$ time by traversing the edges of graph G directly. The runtime of the algorithm/reduction is then dominated by the runtime of the maximum flow algorithm we use. As stated earlier, for this course, we only need to know that we can run Ford-Fulkerson algorithm and find this maximum flow in $O(m \cdot F)$ time where F is the value of maximum flow. As the value of maximum flow in our network is at most n, this gives an algorithm with runtime $O(mn)$ for the bipartite matching problem.

Important remark: We should always state the runtime of our algorithms in terms of parameters of the *original* input. So in this problem, even though we did a reduction to the maximum flow problem and ended up getting an algorithm with $O(m \cdot F)$ time, F is not a well-defined term in the context of the bipartite matching problem (it is a parameter of the solution not the input). So we should then switch F to a parameter governed by the input which in our case was $O(n)$. Hence, we state the runtime as $O(mn)$.

2.2 Exam Scheduling Problem

We examine another simple application of network flows. Consider the following problem: There are multiple classes that we need to schedule a final exam for, in one of many available rooms, during one of many available time-slots, with one of available proctors that can only attend a certain number of exams and in certain time-slots (too many constraints!); can we find an exam scheduling that works for everyone? Formally,

Problem 3 (**Exam Scheduling**)**.** Suppose we are given:

- A set of n courses where course i has $E[i]$ enrolled students;
- A set of r rooms where room j has $S[j]$ seats;
- A set of t available time-slots for taking an exam denoted simply by $\{1, \ldots, t\}$;
- A set of p proctors where for each proctor k we are given a list $T[k] \subseteq \{1, \ldots, t\}$ of available time-slots.

Moreover, we have the following constraints:

(i) We can only assign a course i to a room j if there are enough seats for the students, i.e., $E[i] \leq S[j]$. We are *not* allowed to assign a course to multiple rooms.

(ii) In any given time-slot, any room can only be assigned to a single exam, i.e., we cannot assign multiple exams at the same time in the same room.

(iii) A proctor k can only attend an exam at time-slot t if $t \in T[k]$.

(iv) No proctor is allowed to attend more than 3 exams in total and each exam requires exactly one proctor.

Find a schedule of exams, namely, a collection of n four-tuples (course, room, time, proctor) that satisfy all above constraints or output that no such schedule is possible.

Before we get to the solution, we shall note that these types of "tuple selection" problems form a general array of problems that are solvable by network flow algorithms and thus it is worth familiarizing yourself with solving them in more details.

Solving exam scheduling using network flow. We create the following flow network $G(V, E)$ (see Figure 3 for an illustration):

- There are n vertices in a set C corresponding to the courses, r vertices in R corresponding to the rooms, t vertices in T corresponding to time-slots, and p vertices in P corresponding to the proctors. We also add a source vertex s and sink vertex t.
- We connect:
  - source s to all vertices (courses) in C with edges of capacity 1;
  - any course-vertex i in C to any room-vertex j in R if $E[i] \leq S[j]$ with an edge of capacity 1;
  - any room-vertex j in R to any time-vertex t in T with an edge of capacity 1;
  - any time-vertex t in T to any proctor-vertex k in P if $t \in T[k]$ with an edge of capacity 1;
  - any proctor-vertex k in P to sink t with an edge of capacity 3.

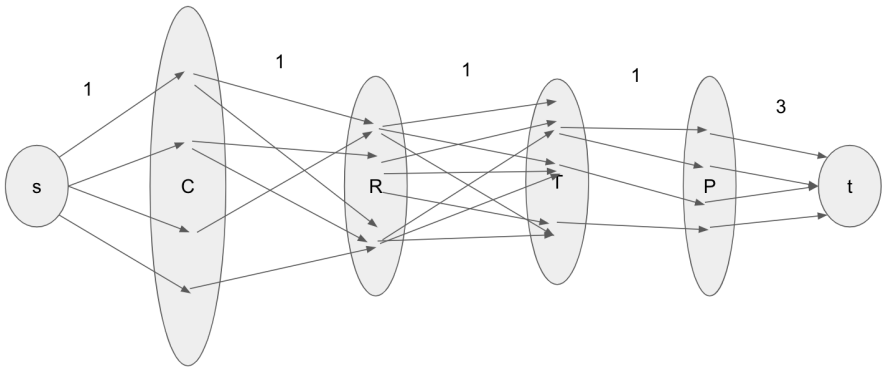


Figure 3: An illustration of the reduction for the exam scheduling problem

We then find a maximum flow f from s to t in the network G with the given capacities. For every one of the flow paths

$$(s, \text{course-vertex}_i, \text{room-vertex}_j, \text{time-vertex}_\ell, \text{proctor-vertex}_k, t),$$

we return the tuple $(\text{course}_i, \text{room}_j, \text{time}_\ell, \text{proctor}_k)$ as the a tuple of (course, room, time, proctor) (since the edges going out of s have capacity 1, we will have at most n different flow paths in f each carrying one unit of flow; moreover, except for edges between P and t, all the other edges of these paths are disjoint since those edges have capacity one in the network). Finally, if the number of found tuples is less than n we output that no schedule is possible and otherwise output the found tuples as the schedule of all courses. This concludes the reduction.

Proof of Correctness: We prove that the value of flow f is equal to the maximum number of (course, room, time, proctor) tuples that we can satisfy *simultaneously*. This implies that when value of f is less than n, it is not possible to schedule all the courses. We also show that when value of f is equal to n, the set of returned tuples form a valid schedule of all courses.

The proof is similar to the case of bipartite matching problem: we first prove that the set of 4-tuples returned by the algorithm is indeed a valid schedule and then prove its size is maximum.

- The returned tuples form a valid schedule: Firstly, the tuple

$$(\text{course-vertex}_i, \text{room-vertex}_j, \text{time-vertex}_\ell, \text{proctor-vertex}_k)$$

gives a correct schedule of a (course, room, time, proctor) since each course can only be scheduled in a room with larger number of seats and each proctor can only be scheduled in the given available time-slots (by definition of edges of the network). Moreover, since capacity of every edge other than P to t edges in G is one, we know that each pair $(s, \text{course-vertex}_i)$, $(\text{room-vertex}_j, \text{time-vertex}_\ell)$, and $(\text{time-vertex}_\ell, \text{proctor-vertex}_k)$ appear at most once in the schedule: this means a course is scheduled at most once, each room is assigned only once to a particular time-slot, and each proctor is assigned only once to a time-slot. Moreover, since capacity of each P to t edge is 3, each proctor is assigned to at most 3 courses in total. This implies that the schedule found satisfies all the constraints.

- The returned tuples has maximum size: We prove that any flow f in G corresponds to a schedule with the same number of tuples as size of f and vice versa. This then implies that maximum flow corresponds to a maximum size schedule, proving the result. By the above part, we know that for any flow f in G there is a schedule of the same size. We now prove the other direction.

  For any schedule of $(course, room, time, proctor)$, we can define a flow f where:

  - $f(s, \text{course-vertex}_i) = 1$, $f(\text{course-vertex}_i, \text{room-vertex}_j) = 1$, $f(\text{room-vertex}_j, \text{time-vertex}_\ell) = 1$, and $f(\text{time-vertex}_\ell, \text{proctor-vertex}_k) = 1$ if $(\text{course}_i, \text{room}_j, \text{time}_\ell, \text{proctor}_k)$ appears in schedule.
  - $f(\text{proctor-vertex}_k, t)$ is equal to the number of times proctor_k is assigned to a course in schedule.

  By a similar argument as in the first part, we can see that this is a valid flow (satisfying both capacity constraints and preservation of flow) and its value is equal to the number of tuples in the schedule.

This concludes the proof of correctness of the algorithm.

Runtime analysis: The network G has $N = n + r + t + p + 2$ vertices and $O(n \cdot r + r \cdot t + t \cdot p) = O(N^2)$ edges. Moreover, we can construct the network in $O(N^2)$ time. Finally, by running Ford-Fulkerson algorithm, we can find the max flow in $O(N^2 \cdot F)$ time where F is the value of maximum flow in the network which is at most $O(N)$ (since outgoing edges of s have capacity $n = O(N)$). Hence, the runtime of the algorithm is $O(N^3)$ which is $O((n + r + t + p)^3)$ in the original parameters of the problem (note that we could have been slightly more careful and found a slightly better upper bound on the runtime but since this is not the main focus of this question, we can simply go with the simplest bound on the runtime).

3 Final Remarks on Network Flows

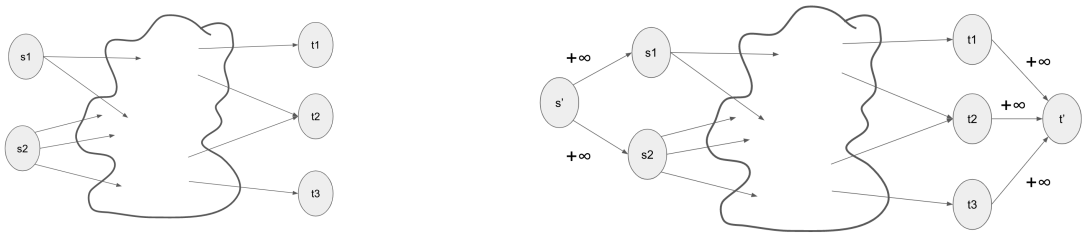
We conclude our study of network flows in this lecture by making the following general (and hopefully useful) remarks about flows.

Multiple Source and/or Multiple Sinks

What if we want to have a scenario with multiple source vertices that can generate flow and/or multiple sinks that can consume it?

This scenario can be captured as follows: we have source vertices $s_1, \ldots, s_k$ (with no incoming edge) and sink vertices $t_1, \ldots, t_\ell$ (with no outgoing edge) and a flow f is preserved on vertices $V - \{s_1, \ldots, s_k, t_1, \ldots, t_\ell\}$ (and satisfy the capacity constraints). The value of f is then considered $\sum_{i=1}^{k} \sum_v f(s_i, v)$ (which is also equal to $\sum_{j=1}^{\ell} \sum_v f(v, t_j)$).

We can solve this variant of multi-source multi-sink maximum flow problem by doing a reduction to the original maximum flow. Simply create a new network $G'(V', E')$ where $V' = V \cup \{s', t'\}$ for two new vertices s' and t' and $E' = E \cup \{(s', s_1), \ldots, (s', s_k)\} \cup \{(t_1, t'), \ldots, (t_\ell, t')\}$ where the capacity of edges in E is the same as before and capacity of new edges is $+\infty$ (the proof of correctness is straightforward). See Figure 4 for an illustration.



(a) A 2-source 3-sink maximum flow instance

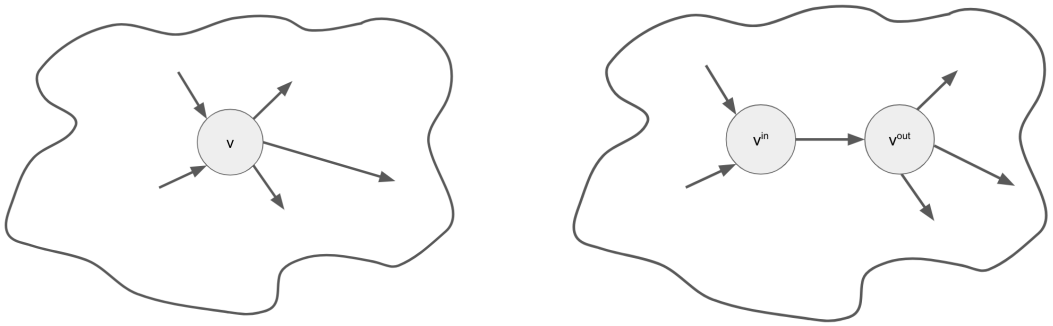
(b) The (original) maximum flow reduction

Figure 4: An illustration of the reduction for the multi-source multi-sink maximum flow problem.

Finding Edge-Disjoint or Vertex-Disjoint Paths

Perhaps the simplest application of maximum flow is to find the *maximum number of edge-disjoint paths* from a vertex s to a vertex t in a given graph $G(V, E)$. Simply turn G into a network by assigning capacity one to every edge (and if G is originally undirected, add both direction of edges to make it directed). Then find the maximum flow from s to t in this network. It is very easy to prove that this flow corresponds to the maximum number of edge disjoint paths from s to t in G (prove it on your own as an exercise). Moreover, the value of such flow is at most $n - 1$ and hence Ford-Fulkerson solves this problem in $O(mn)$ time.

What if we are interested in finding vertex-disjoint paths from s to t (obviously, all these paths should still share s and t but beside s, t no other vertex should be used in more than one of them)? A simple way is to introduce something like "vertex capacities" by doing the following: turn every vertex v in the original graph G into two vertices v^{in} and v^{out}, and turn any edge (u, v) in the original graph to an edge (u^{out}, v^{in}) (with capacity one); then also add the edge (v^{in}, v^{out}) with capacity one to the graph (see Figure 5).



(a) A single vertex in the original graph

(b) The two copies of a vertex in the new network

Figure 5: An illustration of the reduction for finding vertex-disjoint paths.

By finding maximum flow in this network, we can find maximum number of vertex disjoint-paths from s to t (the proof is again straightforward and is left as an exercise). This gives an $O(mn)$ time algorithm for this problem.

Note: This idea of introducing "vertex capacities" is quite general and can be used in other settings of network flows as well (we can add arbitrary capacity c_v to a vertex v by making the edge (v^{in}, v^{out}) have this capacity c_v).

Integrality of Max Flows

Recall that we said a flow f is a function $V \times V \to \mathbb{R}^+$ which satisfies capacity constraints and preservation of flow. Note that importantly, this means the value of flow can be fractional on some edges. However,

we so far always treated the maximum flow found from s to t in our graph to be *integral*, namely, the flow passed through every edge is an integer. But is this a correct assumption or can it be the case that in some networks the maximum flow should necessarily use some fractional values?

It turns out that as long as capacity of every edge is an integer, there always exists a maximum flow with integral values over every edge (clearly, if capacities of some edges are rational, we may need our maximum flow to also be rational). This statement requires a proof and follows from the proof of correctness of Ford-Fulkerson's algorithm. However, for the purpose of this course, you may always assume that as long as capacities are integer, the maximum flow found by the algorithms is also integral.

Cycle-Freeness of Max Flows

Another property of maximum flows we can always assume in this course is that the set of edges (u, v) with $f(u, v) > 0$ do *not* contain a (directed) cycle (even though the network may indeed have cycles in it) – this again requires a formal proof but we will skip that for the purpose of this course.

This concludes our study of maximum flow problem and graph algorithms in general.