CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 22

April 14, 2022

Instructor: Sepehr Assadi

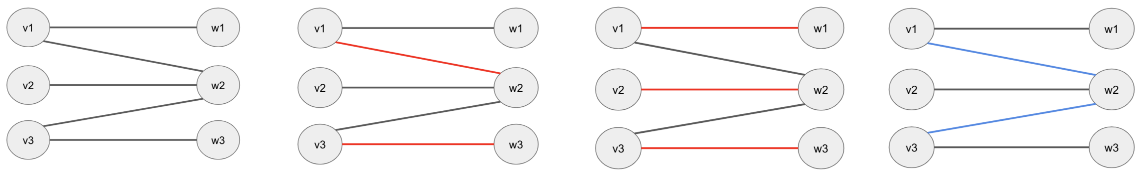
1 Application II: Bipartite Matching Problem

We continue with our study of applications of network flow. Recall that we defined the bipartite matching problem in the previous lecture as follows:

Problem 1 (**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

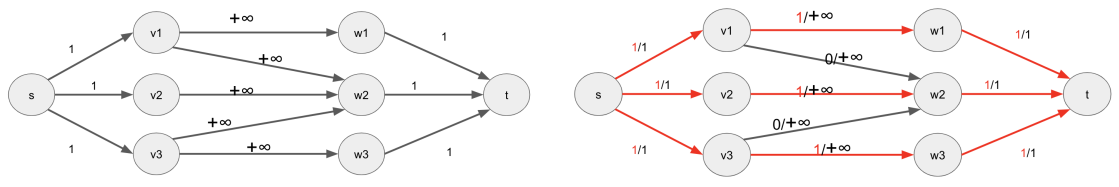
The bipartite matching problem above captures the buyer-item scenario as follows: We create a bipartite graph $G = (V, E)$ with vertices in L corresponding to buyers and vertices in R corresponding to the items; we then connect every buyer-vertex i in L to every item-vertex in S_i in R. It is immediate 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.

Bipartite matching is an extremely useful problem in practice, for instance in online advertising (which advertisement to show to which viewer), auctions (which items to sell to which bidder), etc.

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.

Again, an important reminder that we should always state the runtime of our algorithms in terms of parameters of the *original* input, so the runtime here is $O(mn)$.

2 Application III: Exam Scheduling Problem

We examine another 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 2 (Exam Scheduling). Suppose we are given:

- A set of c 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 c 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 c 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.

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

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

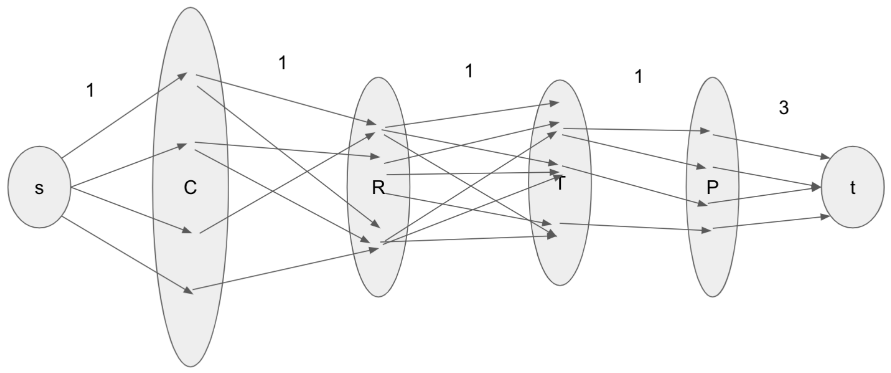


Figure 3: An illustration of the reduction for the exam scheduling problem

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 c 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 c 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 c, it is not possible to schedule all the courses. We also show that when value of f is equal to c, 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 $n = c + r + t + p + 2$ vertices and $O(c \cdot r + r \cdot t + t \cdot p) = O(n^2)$ 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 $c = O(n)$). Hence, the runtime of the algorithm is $O(n^3)$ which is $O((c + r + t + p)^3)$ 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).

This concludes the study of our network flows in this course.