CMPUT 675: Topics in Algorithms and Combinatorial Optimization (Fall 2009)

Lecture 1: Basics, Bipartite Matching

Lecturer: Mohammad R. Salavatipour **Scriber:** Mohammad R. Salavatipour

Date: Sept 3, 2009

This lecture starts with some basic notions. Then we study the problem of finding a maximum (cardinality) bipartite matching.

1 P, NP, and NP-complete

Here we give a very quick overview of the notions of P, NP, and NP-complete. For those who have not seen these notions before, we refer to any standard text book on complexity theory (such as "Introduction to Theory of Computation" by Sipser).

First let's define what is a decision problem. An abstract problems whose solution is always either **yes** or **no**. For example, "given an integer P, is P a prime number?" or "given a graph $G = (V, E)$ and two vertices $s, t \in V$, is it true that there is a path from s to t in G?". Note that many optimization problems can be re-formulated as a decision one. For example, in the decision version of the Maximum flow problem we are given a network $G = (V, E)$ with edge capacities c_e, a source $s \in V$ and a sink $t \in V$, and an integer ℓ. Question is: is there a flow from s to t in G such that its value is at least ℓ?

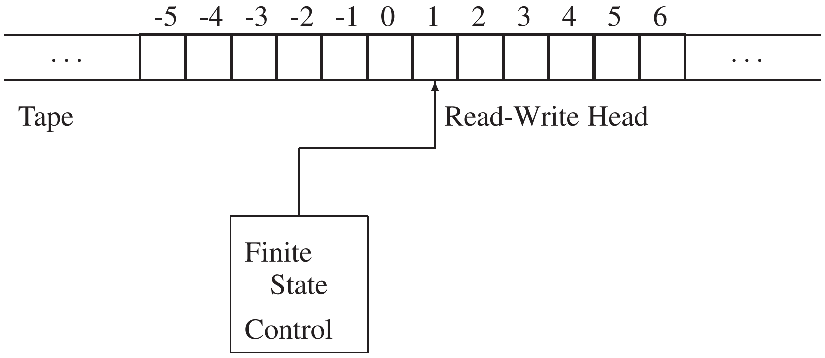
1.1 Relations between optimization problem and decision problem

It is easy to see that if you can solve the optimization problem, then you can also solve the decision problem. The interesting part is that in many cases the opposite is also true. For example, if you consider the problem of finding the shortest path between two given nodes s, t in a given graph $G = (V, E)$, the decision problem is: "given G, s, t, and integer k, is there a path of length at most k from s to t?" if we have an oracle (procedure) to solve this problem then we can solve the optimization problem (shortest path) using polynomially many calls to this oracle. One can easily find the *length* of the shortest path by trying all possible values of $1 \leq k \leq n$ (or just do binary search) to find the smallest k for which the oracle says yes and no for larger values.

Exercise 1: How can you extend this to actually find the shortest path itself?

1.2 Deterministic Turing Machine and class P

We give a model of computation to solve decision problems. The abstract model of computation we consider is a Turing Machine.



A deterministic Turing Machine (DTM) consists of three main components: 1) a finite state control, 2) a read-write head, and 3) a (two-way, infinite, position labeled) tape. You can think of the FSC as the program. This FSC works by reading the letter written on the tape at the position currently the head is pointing to. Then based on the value of that and the current state it specifies the next state, also the letter should be written back to the tape, and to move the head one step to left or right. We use Σ to denote the set of alphabet we are working with and there is a special character, b, used to denote blank. Let's call $\Gamma = \Sigma \cup \{b\}$. FSC has a finite state set Q which includes 3 special states: q_0 for start, q_Y and q_N for halt (accept and reject states). A transition function specifies the next move of the machine at any given state: δ: $(Q - \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

Turing Machines can in fact be used to solve more than just decision problems. One can treat the final sequence left on the tape at the halting state as the solution returned by the Turing Machine. But for now we only focus on decision problems.

For a given input string x, the running time of a DTM M on x, denoted by $t_M(x)$, is the number of steps it takes to get into one of halting states. The size of input x, denoted by $|x|$, is the number of characters in x. The worst case running time of a DTM, M, is a function $T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ with

$$T_M(n) = \max_x \{t_M(x) | \text{ strings } x \text{ with } |x| = n\}$$

If this function is a polynomial in n then we say that M is a polynomial time DTM.

Class P: is the set of decision problems for which there is a polynomial time DTM that solves it.

As said earlier, if the TM needs to output a string we can view the string on the tape as the output string when the TM halts. In this case we say M computes a function $f : \Sigma^* \to \Sigma^*$ if for each $x \in \Sigma^*$, given x, M halts with output $f(x)$ on the tape.

Then we can define:

$FP = \{f : \Sigma^* \to \Sigma^* |$ there is polytime TM M that computes $f\}$

Polytime Thesis (Church-Turing thesis): If a problem can be solved (language can be accepted) or a function can be computed by some polytime algorithm, based on some reasonable notion of polytime-algorithm, then that language or function can be computed by a polytime TM.

Therefore, we can switch the model of computation to a RAM (random access machine) which is basically the computers we use. Reduction from a TM to a RAM is easy; the other way is also true. Therefore, when we say an algorithm (on a RAM) has polynomial running time, it means it is a function that is polynomial in size of the input. We should be careful as what is the size of an input. The size of an instance is the number of memory bits (or memory blocks) needed to represent the instance in binary (or larger bases). For example, if the input is a graph $G = (V, E)$ then the size of the instance is $O(|V| + |E|)$. If we are also given an integer W then the size of the instance is $O(|V| + |E| + \log W)$ since we only need $\log W$ bits to represent that integer. Therefore, for the problem of "given integer P, is P a prime?" the running time of algorithm should be polynomial in $\log P$ to be considered polynomial. If the running time is a function of the actual value of integers then it is pseudo-polynomial. If the input contains some numbers but the running time is only dependent on the number of them but not on their values then we say the algorithm is strongly polynomial.

Example 1: Given a list of n integers (each of which fits into one memory block), a running time of $O(n^c)$ for any fixed constant c is considered polynomial. If we are given a graph $G = (V, E)$ with $|V| = n$, $|E| = m$, and each edge having an integer weight of up to W then a running time of $O(n^2 m \log W)$ is considered polynomial but $O(nmW)$ is not polynomial. A running time of $O(n^3 m^2)$ is strongly polynomial.

1.3 Polynomial time Reductions, NP and NP-complete

The notion of class NP is defined using non-deterministic Turing machine. Basically, a problem belongs to NP if there is a NDTM for it with polynomial running time. Alternatively, we say a decision problem Π belongs to NP if for every yes instance x of the problem (i.e. if the answer to the question is yes) then there is a certificate y whose size is polynomial in x and can be verified in polynomial time (by a DTM). For example, the question of "Is given integer N a composite?" is clearly in NP because if the answer is yes then a certificate would be two integers $m, p > 1$ such that $N = m \cdot p$ and one can easily verify this (given m, p) in polynomial time. Another example could be: "given a graph $G = (V, E)$, does G have a Hamiltonian cycle (i.e. a simple cycle containing all the vertices)?". Clearly a certificate to a yes answer (which will be a ham-cycle) can be verified in polynomial time. To define class NP-complete we need to define notion of reduction.

Reduction: Given two decision problems Π_1 and Π_2, a reduction from Π_1 to Π_2 is a polynomial time computable function f which maps instances of Π_1 to instances of Π_2 such that $x \in \Pi_1 \Leftrightarrow f(x) \in \Pi_2$, that is the answer to instance x is yes if and only if the answer to instance $f(x)$ is yes. Thus, if we can solve Π_2 in polytime then we can solve Π_1 in polytime too.

It is easy to verify that the polytime reduction defined above is transitive. That is if there is a reduction from Π_1 to Π_2 and there is a reduction from Π_2 to another problem Π_3 then there is a polytime reduction from Π_1 to Π_2. Now we are ready to define class NP-complete.

A problem Π is NP-complete if:

1. Π is in NP
2. Every other problem $\Pi' \in NP$ can be reduced to Π.

In a sense, problems in NP-complete are the hardest problems of NP. The question becomes, how can one prove the second property? Fortunately, there is a nice theorem which proves the existence of a problem that is NP-complete. SAT is the problem of checking whether a given Boolean formula is satisfiable or not. Cook proved (in 1971) that SAT is NP-complete

Theorem 1.1 (Cook'71) *SAT is NP-complete.*

Once we have a non-empty class of NP-complete problems to start, to prove a problem Π is NP-complete, one only needs to show that it is in NP and then find a problem Π' that is already known to be NP-complete

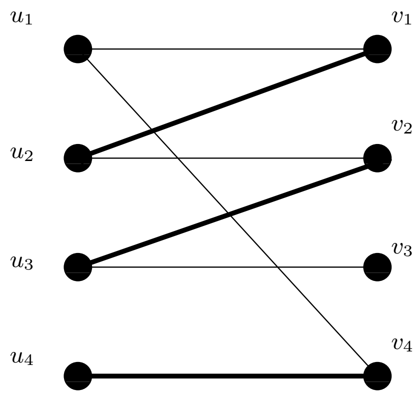


Figure 1: A maximal matching in a bipartite graph

and give a reduction from Π' to Π. By transitivity of reduction, it implies that every problem in NP can be reduced to Π and that proves the NP-completeness of Π.

2 Maximum Cardinality Bipartite Matching

A graph $G = (V, E)$ is bipartite if its vertex set can be partitioned into two sets A, B such that every edge $e \in E$ has one end-point in A and one end-point in B. The degree of a node v, is the number of edges incident to that node. It is easy to prove that:

Lemma 2.1 *A graph G is bipartite if and only if it does not have any odd cycles.*

Proof: Exercise. ∎

Definition 2.2 *A subset of edges $M \subseteq E$ is called a matching if each vertex of V is incident to at most one edge of M, i.e. edges of M do not share any end-points. Vertices of V that are incident to an edge of M are called* saturated*; the other vertices are called* unsaturated *or* exposed*.*

Therefore, every vertex of a graph in a matching has degree at most 1. A matching M is *maximal* if it cannot be extended to a larger matching by adding more edges. It is a *maximum* matching if it has the largest size among all possible matchings. We say it is a *perfect* matching if it saturates all the vertices. Therefore, in a bipartite graph $G = (A \cup B, E)$ to have a perfect matching one has to have $|A| = |B|$.

The maximum (unweighted) matching problem is: given a graph G, find a matching of maximum size. In this lecture we focus on this problem when the input graph is bipartite. This problem can be generalized to the weighted case where along with the graph we have an edge weight w_e for each edge and the goal is to find a matching whose total weight is the largest. The cardinality matching is the special case when all edge weights are 1. Figure 1 shows an example of a bipartite graph with a maximal matching (shown in bolder lines).

One natural algorithm one might try to find a maximum matching could be a greedy one. It is not hard to convince yourself that such an algorithm will fail (for example one could end-up in with a maximal matching such as the one in Figure 1 which is not maximum).

An important question we need to answer is: how to tell if a given matching M is maximum? In general, in designing algorithms for optimization problems an important task is to find good bounds for the optimum. For this case, we need to find good upper bounds on the size of a matching and hope that the smallest of the upper bounds is equal to the size of a matching we have found and therefore our solution is optimum (i.e.

Figure 2: Two graphs each with a vertex cover (the squared nodes)

maximum matching). For this purpose, we need to define another problem which will be the *dual* problem of matching.

Definition 2.3 *A vertex cover of a given graph $G = (V, E)$ is a set $C \subseteq V$ such that every edge of the graph has at least one end-point in C, i.e. C covers all the edges (see Figure 2).*

It follows from the definition of a vertex cover that there is no edge in $V - C$. This implies that the size of any matching is at most the size of any vertex cover (since for every matching M, any vertex cover must contain at least one vertex from every edge and all the edges of the matching are disjoint). Thus:

Fact: For any matching M and any vertex cover C: $|C| \geq |M|$.

This fact is true for any graph G (which may not be bipartite).

The vertex cover problem is: given a graph G, find the smallest vertex cover in G. This problem is the dual of matching. The above fact is called the weak duality: the size of the solution of the minimization problem (vertex cover) is an upper bound for the size of the maximization problem (maximum matching). We shall prove that this upper bound is tight if G is bipartite, i.e. we have a min-max theorem:

Theorem 2.4 (König) *For any bipartite graph G, the maximum size of a matching is equal to the minimum size of a vertex cover.*

What this theorem is showing is called strong duality. This is one of many examples of min-max theorems in combinatorial optimization. We will prove this theorem algorithmically, by presenting an algorithm that finds a maximum matching and a minimum vertex cover (of the same size).

Definition 2.5 *Given a matching M, an M-alternating path is a path that alternates between the edges of M and those in $E - M$.*

Definition 2.6 *An M-augmenting path is an M-alternating path in which the first and last vertex are exposed.*

For example, for the matching of Figure 1, both of the following paths are M-augmenting: $u_1 \rightarrow v_1, \rightarrow u_2 \rightarrow v_2 \rightarrow u_3 \rightarrow v_3$, and $u_1 \rightarrow v_4 \rightarrow u_r \rightarrow v_3$.

Note that if an M-augmenting path has k edges of M then it must have $k + 1$ edges of $E - M$. Also the end-points of such a path are in different parts. What is the purpose of the augmenting paths? As we prove in the following lemma, if we can find an augmenting path then we can make the matching larger by replacing the edges of M in the path by those of the path that are not in M.

Lemma 2.7 *If M is a matching and P is an M-augmenting path then $M \Delta P = (M - P) \cup (P - M)$ is a matching of size $|M| + 1$.*

Proof: It is easy to see that $M \Delta P$ is a matching because each vertex still has a degree of at most 1 (every saturated vertex of P still has degree 1 and the end-points are now also saturated and have degree 1). It is easy to see that the size is $|M| + 1$. ■

Thus, if we can find an augmenting path then we can enlarge the matching. The question is, is it true that if there is no augmenting path then the matching is maximum?

Theorem 2.8 *A matching M is maximum if and only if there is no M-augmenting path.*

Proof: Using Lemma 2.7, it is easy to see that if there is an augmenting path then M is not a maximum matching. We now prove the other direction.

Suppose that there is no augmenting path and there is a matching M' with $|M'| > |M$. Consider the graph $H = (V, M \cup M')$. Since H has maximum degree at most two every connected component of H is either a path or an even cycle (remember there are no odd cycles in a bipartite graph). Noting that in each cycle the edges are alternating between M and M' and so the cycles have an equal number of edges of M and M', there must be a component that is a path with more edges from M' than M. Thus, there must be an M-alternating path with more edges in M' than in M, this will be an M-augmenting path in H, contradicting the assumption that there is no M-augmenting path. ■

This theorem suggests an algorithm for finding a maximum matching in a given bipartite graph:

Maximum Bipartite Matching

> $M \leftarrow \emptyset$
> **while** there is an M-augmenting path P **do**
> $M \leftarrow M \Delta P$
> **return** M

We analyze this algorithm in the next lecture and provide more details on how to find an M-augmenting path efficiently and complete the proof of Theorem 2.4.