Bipartite Matching (17)

Michael L. Littman

October 29th, 1998

1 MATCHING PROBLEM

1.1 Teaching Assignments

Every semester, the CS faculty has to figure out who is going to teach what.
We try to assign faculty to teach all the courses (one course per professor, one professor per course), but if some important course would go untaught, we bring in a grad student or local researcher to help out, or have someone do double duty.
So, it's important to do as good a job as possible at assigning faculty to courses on the first try.

1.2 Example Semester

Here's a piece of the data from a previous semester. Some professors are capable of teaching more than one course and some courses can be taught by more than one professor.
Courses: CPS130, CPS108, CPS100E, CPS100

- Littman: CPS130
- Rodger: CPS130, CPS100E
- Astrachan: CPS108, CPS100E, CPS100
- Ramm: CPS100E
- Duvall: CPS100E

1.3 Bipartite Graphs

This problem is naturally expressed as a *bipartite* graph.
Such a graph $G = (V, E)$ has a set of nodes L and a set of nodes R such that $L \cap R = \emptyset$, $L \cup R = V$ (partition: mutually exclusive and exhaustive), and for all $(u, v) \in E$, $u \in L$ and $v \in R$.
Also natural for machines-tasks, classes-classrooms, and others?

Boys-girls, senders-receivers, servers-clients, clothes-bodyparts, meat-wine, etc.

1.4 Matching

A *matching* M in a graph $G = (V, E)$ is a subset of E such that there is no $u \in V$, $v_1 \in V$, $v_2 \in V$ such that $v_1 \neq v_2$ and either $(u, v_1) \in M$ and $(u, v_2) \in M$, or $(v_1, u) \in M$ and $(v_2, u) \in M$.
In other words, no node is linked to two other nodes. It's a monogamy thing.
I'll use "blue" edges to refer to those in the matching M and "white" for the others.
Example...

1.5 Max Matchings

A *maximal* matching M is a matching such that any other edge in E that might be added to M will violate the matching property (one or the other of the nodes is already matched).
A *maximum* matching is the largest matching out of all possible matchings.
Clearly a maximum matching is maximal. But is every maximal matching maximum?
Examples... Astrachan 100E, 108, 100/108

1.6 Finding a Max Matching

Easy to find a maximal matching. Just keep adding edges until we're full. Running time? (HW)
Unfortunately, because not every maximal matching is maximum, we can get stuck.
What we really want is a maximum matching. So how can we find one?

2 MIN-CUT MAX-FLOW

2.1 Augmenting Path

Turns out that we can actually take a non-maximum maximal matching and extend it by changing the pairing of some of the nodes.
An *augmenting path* starts at an unmatched node on the left, then alternates taking white (unmatched) and blue (matched) edges back and forth until an unmatched node on the right is reached.
Example...

2.2 Augmenting Path Facts

Augmenting paths are neat:

- An augmenting path has one more white edge than blue. Why?

  Start and end on white.

- Flipping the color of edges on the augmenting path results in a valid matching. Why?

Turning a blue edge white could never hurt. The new blue edges all emanate from nodes with no blue edges (or formerly one blue edge, now white).

- Therefore, after flipping the colors, our matching is bigger by one edge.

2.3 Algorithm

We're getting close to an algorithm.

```
Bipartite-Match(G)
M ← ∅
p ← Augmenting-Path(G, M)
while p ≠ nil
    do M ← M ∪ p[1]
       for i ← 1 to (|p| − 1)/2
           do M ← M − p[2i]
              M ← M ∪ p[2i + 1]
       p ← Augmenting-Path(G, M)
return M
```

Will get a matching that's maximal-plus-plus. But, there are still some unanswered questions. Like?

How do we find an augmenting path? Does no augmenting path imply maximum?

2.4 Cuts

Break left and right vertices into top and bottom subsets.
The *size* of a cut is the number of bottom left nodes x, plus the number of top right nodes y, plus the number of edges from top left to lower right z.
Lemma: The size $x + y + z$ of any cut is an *upper bound* on the size of the maximum matching.
Why? Picture.

2.5 Max-flow Min-cut Theorem

Given a bipartite graph G and a matching M, the following are equivalent:

(1) The current matching is maximum.

(2) There are no augmenting paths.

(3) The size of the matching is equal to the size of some cut.

Proof:

- (1) implies (2) because the existence of an augmenting path implies that the current matching is *not* maximum. So maximum implies no augmenting path.
- (3) implies (1) because of the lemma: the size of a cut upper bounds the size of the matching, so if we have a matching that meets this upper bound, it is maximum.
- (2) implies (3) is the interesting one...

2.6 Proof of Max-flow Min-cut

Construct the following cut: Let the top nodes be the unmatched left nodes along with all those nodes reachable by alternating white-blue paths from them. Let the bottom nodes be the others.
Facts:

- All y top right nodes are matched. What would it mean if a top right node were unmatched?

    Augmenting path.

- All x bottom left nodes are matched. By definition... all unmatched left nodes were put on top.
- There are $z = 0$ edges from top left to bottom right. This is because there can be no such edge. Why? Such an edge couldn't be white, because any right node with a white edge from the top is itself a top, by definition. But, such an edge couldn't be blue, either, because all top left nodes are either unmatched (the starter nodes), or matched to a top right node (the others).

So, the size of this cut is equal to the size of the matching. Therefore, we have a maximum matching, since cut sizes are upper bounds on matching sizes.

3 FORD-FULKERSON METHOD

3.1 Review

So, Bipartite-Match computes a maximum matching!
But, how do we find augmenting paths?

3.2 Augmenting-Path Algorithm

This is just another application of depth-first search (or breadth-first).

- Mark nodes as matched or unmatched.
- Start a search from each of the unmatched left nodes.

- Traverse white edges from left to right.
- Traverse blue edges from right to left.
- Stop successfully if an unmatched right node is reached.
- Stop with failure if search terminates without finding an unmatched right node.

Running time?

$$O((|V| + |M|) + (|V| + |E|)) = O(|V| + |E|).$$

3.3 Overall Running Time

Accounting:

- Time before loop: $O(|V| + |E|)$.
- Each iteration: $O(2|P| + (|V| + |E|)) = O(|V| + |E|)$.

Total time is time for initialization plus time for each iteration times maximum number of iterations. How many iterations?

$|V|/2$, since each augmenting path adds two nodes to the matching.

$O((|V| + |E|) + |V|(|V| + |E|)) = O(|V||E|)$ (best algorithm known is $O(\sqrt{|V|}|E|)$.

4 GENERALIZATIONS

4.1 Network Flow

We can generalize the bipartite matching problem quite a bit.
A beautiful generalization is network flow:

- source, sink, nodes, capacities, flow.

Can be solved efficiently using a generalization of augmenting paths: $O(|V||E|^2)$.
Matching is a special case: unit capacities, flow left to right.

4.2 Linear Programming

Even more general is linear programming (LP): $Ax \leq b$, maximize cx.
Example:

- One variable for each edge, constrained to be between zero (white) and one (blue).
- Constraint for each node: sum of value of variables on incident edges must be no more than one (matching property).
- Maximize sum of edges (number of edges in matching).

In spite of the lack of an integrality constraint, the value of this linear program is equal to the size of the maximum matching.
Can solve network flow problems this way, as well as some tricky generalizations.
Theoretically efficient LP algorithms (ellipsoid, interior point) are very complex. Practical LP algorithm (simplex) is inefficient in the worst case. Very good commercial implementations exist!