Washington University in St. Louis

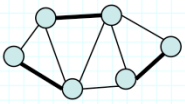
Engineering

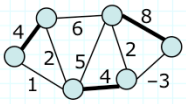
Matchings in Bipartite Graphs

Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst

Matchings in Graphs

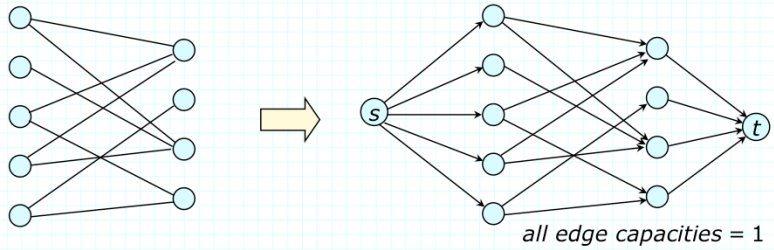




- A *matching M* in a graph *G* is a subset of the edges, with no two edges incident to a common vertex
- Two maximum matching problems
  - » *max size* – find matching with largest number of edges
  - » *max weight* – find matching with largest total edge weight
- Special case of bipartite graphs
  - » in *bipartite graph*, vertex set can be divided into two subsets such that every edge has an endpoint in each subset
  - » bipartite version arises frequently in practice
    - weighted bipartite matching also known as the *assignment problem*
  - » bipartite version can be solved more efficiently than general case

Unweighted Bipartite Matching

- If *G* is bipartite, the max size matching problem can be reduced to a max flow problem as illustrated below



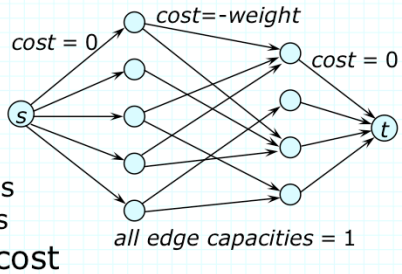
all edge capacities = 1

- For any integer flow on this network the saturated edges in the central part correspond to a matching
- Shortest augmenting path algorithm finds max flow on this graph in $O(mn)$ time
  - » can improve to $O(mn^{1/2})$ using a different max flow algorithm

3

Weighted Bipartite Matching

- Weighted bipartite matching can be solved using a variant of min cost, max flow
  - » construct flow graph as before
  - » assign zero cost to source/sink edges and $cost(u,v)=-w(u,v)$ for the others



- Find a flow with minimum total cost
  - » may not be a max value flow
  - » apply minimum cost augmentation to get sequence of augmenting paths of non-decreasing cost
  - » stop when augmenting path has positive cost
  - » flow at that point defines a maximum weighted matching
- Requires solution to at most n shortest path problems
  - » since edge costs can be negative, must use shortest path algorithm that can handle negative edge lengths

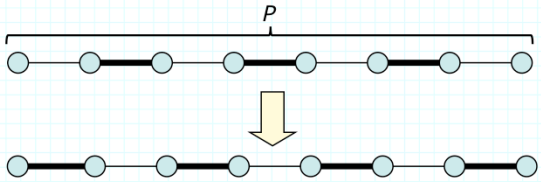
4

Review Questions

1. Draw a non-bipartite graph on with at least 10 vertices and 12 edges. Assign positive weights to all of the edges. By trial and error, find a maximum weight matching. Find a second matching that is a maximum size matching, but is not a maximum weight matching.
2. Draw a bipartite graph with at least 5 vertices in each part and at least 12 edges. For this part, the edges should not have weights. Construct the flow graph described on page 3 and find a maximum flow from the source to the sink. What is the matching that corresponds to this flow.
3. Add edge weights to your graph from question 2. Construct the flow graph for this graph (including edge costs). Find a series of mincost augmenting paths for this flow graph, stopping when you find an augmenting path of positive cost. What is the matching corresponding to this flow?

Alternating Paths

- Let M be a matching of $G=(V,E)$
  - » a *matching edge* is any edge in M; all other edges are *free*
  - » vertex is *matched* if incident to matching edge; *free* otherwise
  - » an *alternating path* or *cycle* is a simple path or cycle in which every other edge is in matching
  - » alternating path is an *augmenting path* if its end points are free
- If G contains an augmenting path P (w.r.t. M), we can get a larger matching by removing the matching edges in P from M and adding the non-matching edges.



Augmenting Path Method

- *Theorem 9.1*. Let M be a matching of G, M' be a max matching and $k=|M'|-|M|$; then G has k vertex disjoint augmenting paths with respect to M

  Proof. Let N be set of edges in M or in M' but not in both
  - » every vertex is incident to at most two edges of N, so the subgraph induced by N consists of paths and even length cycles that are alternating with respect to M
  - » N contains exactly k more edges from M' than from M, so at least k paths in N must begin and end with edges from M'
  - » these paths are vertex disjoint and augmenting (with respect to M) ■
- The *augmenting path method* initializes $M=\{\}$ and repeats following step until no augmenting paths left
  - » *Augmenting Step*: let P be an augmenting path w.r.t. M; remove matching edges in P from M & add non-matching edges
- At most $n/2$ augmenting steps to find a max matching

Augmenting Paths in Bipartite Graphs

- Each vertex is assigned one of three states, *odd, even* or *unreached*; for each matched vertex v, *mate*(v) is the vertex connected to v by a matching edge
- Algorithm builds forest, defined by parent pointers $p(v)$
  - » initially every matched vertex is unreached, every free vertex is even and every free vertex v has $p(v)$=**null**
- Repeat following step until path is found or every edge has been examined

  Choose unexamined edge $\{v,w\}$ with v even and examine it
  - if w is even, stop; path from root of tree containing v to root of tree containing w forms an augmenting path
  - if w is unreached and matched, make w odd, *mate*(w) even, $p(w)=v$ and $p(mate(w))=w$
- Can be implemented to run in $O(m)$ time

8

```
function path augpath(graph G, matching M);
    vertex u,v,w,x,y;
    mapping state:vertex→{unreached,even,odd};
    mapping matched:vertex→{true,false};
    mapping p, mate: vertex→vertex;
    for u∈V ⇒ state(u):=even; matched(u) := false rof;
    for {u,v}∈M ⇒
        state(u), state(v) := unreached;
        matched(u), matched(v) := true; mate(u):=v; mate(v):=u;
    rof;
    queue := [];
    for u∈[1,n] ⇒
        for {u,v}∈edges(u) ⇒
            if v>u and  (state(u)=even or state(v)=even) ⇒
                queue := queue & {u,v};
            fi;
        rof;
    rof;
```

```
    do queue ≠ [] ⇒
        {v,w} := queue(1); queue := queue[2..];
        if not even(v) ⇒ v ↔ w fi;
        if state(w) = unreached and  matched(w) ⇒
            x := mate(w);
            state(w) := odd; p(w):=v; state(x) :=even; p(x):=w;
            for {x,y}∈edges(x) – {x,w} ⇒
                if {x,y}∉queue ⇒ queue := queue & {x,y}; fi;
            rof;
          | state(w) = even ⇒
            path := []; x := v;
            do p(x) ≠ null ⇒ path := [p(x),x] & path; x := p(x); od;
            path := path & [v,w]; x := w;
            do p(x) ≠ null ⇒ path := path & [x,p(x)]; x := p(x); od;
            return path;
        fi;
    od;
end;
```

main **do**-loop executed at most *m* times
inner **for**-loop also executed at most *m* times
inner **do**-loops executed at most *n* times

10

Stable Matchings

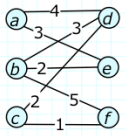
- Given two sets A and B with n elements each, and
  - » for each element $a \in A$, a function $rank_a(b)$ that assigns each $b \in B$ a unique integer in $[1,n]$, and
  - » for each element $b \in B$, a function $rank_b(a)$ that assigns each $a \in A$ a unique integer in $[1,n]$
- *Stable matching* is set P of n pairs (a,b) with $a \in A$, $b \in B$ such that for any two pairs $(a,b) \in P$ and $(\alpha,\beta) \in P$

  $(rank_a(b) < rank_a(\beta)$ or $rank_\beta(\alpha) < rank_\beta(a))$ and
  $(rank_b(a) < rank_b(\alpha)$ or $rank_\alpha(\beta) < rank_\alpha(b))$
- Gale-Shapley algorithm – to find stable matching start with no pairs and repeat following step until done
  - » select an unpaired element a of A and let b be its top-ranked member of B, among those it has not yet tried to match with
  - » if b has no current match, form pair (a,b); if b has a current match (a',b) and b ranks a above a', replace (a',b) with (a,b)

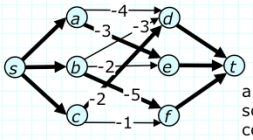
11

Exercises

1. Consider the weighted bipartite graph shown below.



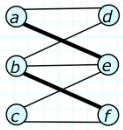
Construct the flow graph used to find a max weight matching in this graph and find a min-cost flow corresponding to a max wt matching.



all capacities are 1
source/sink edge
costs are zero

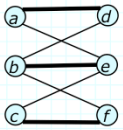
The heavy-weight edges correspond to a min-cost flow. The weight of the corresponding matching is 10.

2. Identify an augmenting path in the bipartite graph shown below, relative to the matching defined by the heavy weight edges.

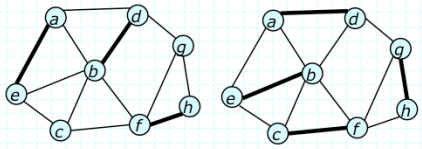


The path daebfc is an augmenting path.

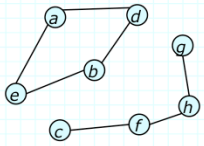
Show the matching obtained using this augmenting path.



3. The figure below shows two matchings in a graph.



Draw the graph defined by the edges that are in one matching or the other, but not both.



4. As described in the notes, the augmenting path algorithm performs a lot of redundant work each time it starts a new augmenting path search. Describe a version that would eliminate much of this overhead.

One simple optimization is to avoid doing the augmenting path search at all, when the number of edges in the matching is small. To implement this, just go through the edges one-by-one and add an edge to the matching so long as it doesn't conflict with any previously selected edge. This will give us a matching that is at least half as large as the maximum size matching.

After we find this initial matching, we can start applying the augmenting path algorithm. We can avoid much of the overhead of the search by maintaining a variable visit(u) for each vertex u. These are initialized to zero. When we reach a vertex u in the k-th augmenting path search, the value of visit(u) is set to k. We can use visit(u) to determine if u has been visited in the current search or not (this takes the place of the unreached state in the original search).

We can also save some overhead by maintaining a list of free (unmatched) vertices and a list of leaves in the set of trees in the current augmenting path search. The list of free vertices can be carried over from one search to the next (although it is must be modified at the end of each successful search). The list of leaves is initialized at the start of each search to contain all the free vertices. Then each step in the path search expands a leaf by examining its incident edges that are not in the matching. If the other endpoint of such an edge has not yet been reached in the search, we can expand the tree containing the current leaf. If the other endpoint has been reached, then the path to the root of its tree together with the path from the current leaf to its tree forms an augmenting path.

By eliminating about half the path searches and much of the overhead of each search, this version can potentially cut the time required to find a maximum matching by a factor of 2 or 3.