CSC 373 Lecture 14

- Review: The max flow-min cut theorem
- A simple consequence of the FF max flow algorithm: if all capacities are integral (or rational) then any FF implementation will terminate with an optimal integral max flow
- Ways to choose an augmenting path so as to insure polynomial time termination
- Immediate applications of max flow

The Ford Fulkerson scheme

- The format if the Ford Fulkerson scheme is:
  f := 0 ; G(f) = G %initialize
  While there is an augmenting path in *G(f)*
  Choose an augmenting path pi
  f':= f+f_pi; f:=f' % Note this also changes *G(f)*
  End While
- I call this a "scheme" rather than a well specified algorithm since we have not said how one chooses an augmenting path (as there can be many such paths)

The Theorem

- The following are equivalent:
- 1) *f* is a max flow
  2) There are no augmenting paths wrt flow *f*; that is, no *s-t* path in *G(f)*
  3) *val(f) = c(S,T)* for some cut *(S,T)* ; hence this *cut (S,T)* must be a min (capacity) cut since *val(f) <= c(S,T)* for all cuts.
- Hence the name max flow (=) min cut

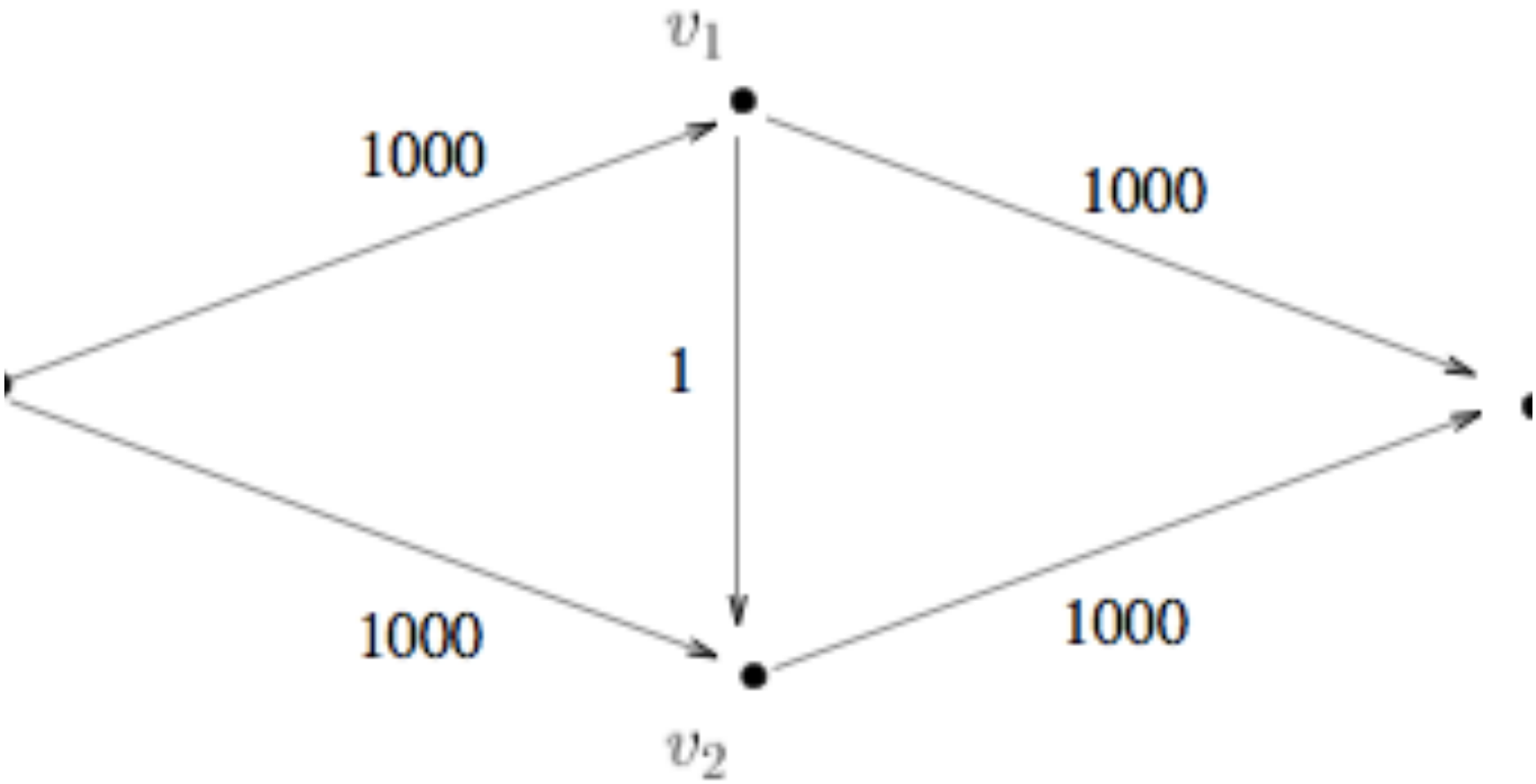
The proof

- 1) => 2) If there is an augmenting path (wrt *f* then f can be increased and hence not optimal
- 2) => 3) Consider all the nodes *S* reachable from s in the residual graph *G(f)*. Note that *t* cannot be in *S* and hence *(S,T) = (S,V-S)* is a cut and *c(S,T) =* val(f) since *f(u,v) = c(u,v)* for all edges *(u,v)* with *u* in *S, v* in *T*
- 3) =>1) Let *f'* be an arbitrary flow. We know *val(f') <=c(S,T)* for any *cut (S,T)* and hence *val(f') <= val(f)* for the cut constructed in 2).

Many ways to insure polynomial time

- As claimed, one can have bad ways to choose augmenting paths such that with non rational capacities, the FF algorithm will not terminate.
- The max flow min cut theorem along with the observation that each augmenting path has residual capacity at least one insures that with integral capacities FF must always terminate and the number of iterations is at most *C*, the sum of edge capacities leaving *s*. . However, as the example in the notes shows there are bad ways to choose augmenting paths so that FF does not terminate in polynomial time.

Bad example for naive FF



Some ways to achieve poly time

- Choose an augmenting path having shortest distance; this is the Edmonds Karp method and can be found in CLRS (page 661); it has running time *O(n m^2); n = |V|, m = |E|*
- There is a "weakly polynomial time" algorithm in Kleinberg and Tardos; here the number of arithmetic operations depends on the length of the (integral) capacities. It follows that always choosing the largest capacity augmenting path is (at least weakly) polynomial time.
- The method I like to present (although not as fast as the push pull method in the text) is Dinitz' method which has running time *O(m n^2)*. It is also a shortest augmenting path method based on the concept of a blocking flow and has some additional advantages beyond the somewhat better running time of Edmonds-Karp.

Dintiz' algorithm

- *L_f = (V' ,E')* where *V' = {v|* v reachable from *s* in *G_f}* and *(u,v)* in *E'* iff *level(v) = level(u) + 1*. Here *level(u)* = length of shortest path from *s* to *u*. *L_f* is the "levelled graph" wrt *G_f*.

  A blocking flow *f'* is a flow (in a flow network) such that every *s* to *t* path in *L_f* has a saturated edge.

- Dinitz's algorithm:

  Initialize *f(e) = 0* for all *e*.

  While *t* is reachable from *s* in *G_f* (else no augmenting path)

  Construct *L_f* corresponding to *G_f*

  Find a blocking flow *f'* wrt *L_f* and set f:= f + f'

  End While.

Proof sketch

- The algorithm halts in at most *n-2* iterations. The proof of this claim rests on the fact that for every node in *L_f', level'(v) >= level(v)* since every edge in *L_f'* is either an edge in *G_f* or the reverse of an edge in *L_f*; and *level'(t) > level(t)* since *f'* was a blocking flow.
- A blocking flow can be found in time *O(mn)*; the levelled graph can be computed in *O(m)* and using depth first search we can compute a blocking path in time *O(mn)*.

Maximum bipartite matching

- Recall that we do not know an efficient DP or any greedy optimal algorithm for computing a max bipartite matching.
- We ***transform*** the bipartite matching problem for graph *G = (V,E)* into a flow problem:

Let *G' = (V',E')* where *V' = V + {s,t}*; and *E' = E + {(s,x)| x \in X + {(y,t)| y in Y}*

We make this into a flow network *F_G* by making s and t the source and terminal nodes and setting *c(e) = 1* for all *e* in *E'*.

Transformation preserves solutions

- Every matching *M* in *G* gives rise to an integral flow *f_M* in *F_G* with *val(f_M) = |M|* and conversely every integral flow *f* in *F_G* gives rise to a matching M_f in G with *|M| = val(f)*.
- It follows that the run time is *O(mn)*; Dinitz' algorithm can be used to obtain *O(m sqrt(n))*.
- When we get to our next big topic (NP completeness), we will be focusing on decision problems and as a decision problem we have *|M| >=k iff val(f_M) >= k*.