```
===========================================================================
CSC 373                 Lecture Summary for Week  6             Winter 2019
===========================================================================

Choosing augmenting paths efficiently:

  * Example where it could take 2 * 10^100 augmentations to find max flow of
    2 * 10^100. In worst-case, Ford-Fulkerson takes time Theta(m f*), where
    f* is the value of a max flow -- this is not polytime.

  * Edmonds-Karp algorithm: use BFS on residual network to find augmenting
    paths; guaranteed to find an augmenting path with smallest # of edges in
    time O(m). Possible to prove that no more than O(mn) augmentations are
    required to find max. flow. Total time: O(nm^2) = O(n^5).

  * Dinic algorithm: perform complete BFS, use all augmenting paths w.r.t.
    BFS tree, then repeat. Worst-case time down to O(n^2m) = O(n^4).

  * "Preflow" algorithms: don't use augmenting paths; instead, push as much
    as possible along individual edges then go back to fix conservation.
    More complicated to explain and write down correctly, but cuts down time
    to O(n^3).

  * Why are we not concerned about details? Applications of network flow do
    not involve writing new algorithms: you always solve the same problem,
    so you can use an existing implementation that's already been debugged
    and optimized. Applications involve taking a problem, casting it in the
    guise of a network flow, solving the network flow problem, then casting
    the answer back to your problem.

Applications of network flows:

  * Maximum bipartite matching:
    Input: An undirected bipartite graph G=(V_1,V_2,E) -- one where every
        edge is between V_1 and V_2 (i.e., no edge has both endpoints in the
        same "side").
    Output: A disjoint subset of edges (a "matching") with maximum size
        ("disjoint" edges means no edge shares an endpoint with any other
        edge in the matching).

    Algorithm:
     1. Create network N from G: add source s, sink t, edges from s to every
        vertex in V_1, edges from every vertex in V_2 to t; make original
        edges directed from V_1 to V_2; set all edge capacities to 1.
     2. Find maximum flow in N.
     3. Output M = {every original edge with flow = 1}.

    Correctness?
      - Any matching in graph yields flow in network: set flow = 1 for graph
        edges in matching, 0 for graph edges not in matching; set flow = 1
        for new edges to/from matched vertices, 0 for new edges to/from
        unmatched vertices.
      - Any integer flow in network yields matching in graph: pick edges
        with flow = 1 (leave out edges with flow = 0). No edge can have flow
        larger than 1 because of capacities.
    For this correspondence, size of matching = value of flow. Hence, any
    max flow in network yields a maximum matching in graph (because a larger
    matching would give a larger flow).

Difference between Network Flow and other techniques:
```

* When solving optimization problem with "network flow techniques", what w
  are doing is actually a *reduction* or *transformation*: we take the inp
  to our problem and create a network from it, then use standard algorithm
  to solve the maximum flow problem on that network (it's always the same
  problem and always the same algorithm, only the input differs). Then, we
  use the solution to the network flow problem to reconstruct a solution t
  our problem.

  So the algorithm we end up with will always have the following structure

  1. Describe how to create network N from input x (to original problem)
  2. Find max flow f (or min cut X) in N, using standard algorithms
  3. Describe how to construct solution to x from f (or X)

  There are two ways this could go wrong:

  - The network we construct could fail to represent certain solutions
    to our original problem. We show this is *not* the case by arguing
    that every solution to the original problem yields a valid flow (or
    cut) in the network.
  - The network we construct could have solutions that don't correspond
    to anything in our original problem. We show this is *not* the case
    by arguing that every flow (or, depending on the problem, every
    integer flow, or every cut) in the network corresponds to a
    solution to the original problem.

  Sometimes those arguments can be very short, because the correspondence
  is obvious between both problems. But both arguments are important and
  must always be included.

Another application: Maximum Independent Set in a bipartite graph

Input: Bipartite graph G = (V_1,V_2,E).
Output: Independent Set I (_ V_1 u V_2 with |I| maximum -- vertex subset
I is "independent" iff no edge has both endpoints in I.

Algorithm:

1. Create network as for bipartite matching except c(u,v) = oo for each
   edge (u,v) in original graph.
2. Find cut X = (V_s,V_t) with minimum capacity c(X) -- first, find max
   flow, then run algo from proof of Ford-Fulkerson theorem to generate
   min-capacity cut.
3. Return I = (V_1 n V_s) u (V_2 n V_t).

Correctness?

- For any independent set I, let V_s = {s} u (V_1 n I) u (V_2 - I);
  V_t = V - V_s. Then the only edges across the cut X = (V_s,V_t) are
  those from s to V_1 - I and those from V_2 - I to t. So c(X) = |V_1|
  + |V_2| - |I|. So min cut capacity is at most |V_1| + |V_2| - |I|,
  where I is max independent set.
- For any cut X = (V_s,V_t) with finite capacity, let I = (V_1 n V_s)
  u (V_2 n V_t). Because c(X) is finite, there is no edge (u,v) from
  original graph with u in V_s, v in V_t, so I is an independent set.
  Moreover, the only edges across X are those from s to V_1 - I, and
  those from V_2 - I to t, which means c(X) = |V_1| + |V_2| - |I|.
  So max independent set has size at least |V_1| + |V_2| - c(X), where
  X is min-capacity cut.

Hence, max independent set size = min cut capacity and the algorithm
correctly returns an independent set of maximum size.

```
For Next Week
  * Readings: Sections 7.1
  * Self-Test: Exercises 7.17(a), 7.17(b)
```