15-451 Algorithms, Fall 2014, Gupta and Sleator

Lecture 11: Dinic's Algorithm   Wed. October 1, 2014

Dinic's algorithm is a way of implementing the Edmonds-Karp # 2 algorithm mentioned in the Avrim's Network Flows II lecture notes. (Please see those notes for the notation we use here.) The running time is $O(n^2m)$ which improves the $O(nm^2)$ of the original algorithm. The basic idea is to compute all the augmenting paths that have a certain number of edges at once, in what is called a blocking flow computation. Each iteration of this process increases the length of the shortest path from s to t by at least 1. So the number of iterations is at most n.

Here's the algorithm at a high level:

Dinic's algorithm for maximum network flow:

> Initialize G_R to be equal to G. Initialize the flow F to be zero in all edges.
>
> While there is an $s \to t$ path in G_R do:
>
> 1. Construct G_L, the level graph of G_R using a BFS.
> 2. Find a blocking flow f in G_L using the blocking flow algorithm below.
> 3. Augment the current flow F by f, and update the residual graph G_R accordingly.

When the algorithm terminates F is a maximum flow. As explained above the number of iterations is at most n. It remains to give a blocking flow algorithm that runs in time $O(nm)$.

Blocking Flows

Consider the level graph G_L that exists at some stage of Dinic's algorithm. The graph is constructed by doing a breadth-first search from s. Level 0 is $\{s\}$. Level 1 is eveything reachable from s by following an edge (of positive capacity) in G_R. The rest of the levels are defined similarly. Once t is reached, no more levels are generated. The level graph only contains edges between neighboring levels.

We're only going to be interested in augmenting paths in G_L which go through all the levels in increasing order. Call such an augmenting path *proper*. Each time we find a proper augmenting path and augment along it, we are going to saturate (i.e. delete) at least one edge from a level i to a level $i+1$, (and possibly add additional edges that go backwards from a level to the previous level). Therefore after enough such augmentations (at most m), we know that there can be no more proper augmenting paths. The flow in G_L constructed in this manner is called a *blocking flow*. (A blocking flow is not necessarily a maximum flow. Construct an example.)

It's easy to search for a proper augmenting path in $O(m)$ time. This results in an $O(m^2)$ time algorithm to compute a blocking flow. The trick of Dinic's algorithm is that by being a little bit careful, and trying to reuse what we've done already, we can get the running time down to $O(nm)$.

The Blocking Flow Algorithm

For each vertex we're going to keep the list of edges to the next level. We also keep track of a "current" edge that we're working on. Initially, the current pointer points to the first edge of each list. As the algorithm proceeds we advance the current pointers of the nodes. We never need to look at any of the edges that the current pointer has passed.

Blocking Flow Algorithm:

> Do a depth-first-search searching for a single path from s to t. When a recursive call to the DFS(v) function returns, it tells us if it has found a path from v to the sink. If it has found it, we immediately return to our caller saying "path found". If it has not found a path, we advance the current edge pointer and continue with a DFS call to the next neighbor. If there are no more neighbors, then we return "no path found".
>
> When the top level DFS(s) returns with "path found" we augment the flow along that path, and the process continues from the source again. If the top level DFS returns "no path found" then we have found a blocking flow.

Claim: *The algorithm finds a blocking flow.*

Proof: The argument is by induction. We want to claim that when the current edge pointer advances, then that edge is useless as far as being part of a path from s to t is concerned. So we assume that all prior edges passed over are useless. The only way that we declare an edge (u, v) to be useless (and pass it over) is when the DFS(v) returns "no path found". It only does this by virtue of all of the edges (v, w) being found useless. Thus it's clear that edge (u, v) must too be useless.

We need one more small observation to complete the proof. An edge can be passed (declared useless), then an augmentation can occur, which changes the graph. Why is it justified to claim that it is still useless? It is because the augmentation only removes edges from a level to the next level and adds edges going backwards. So an edge, once useless, is always useless. ∎

Claim: *The algorithm runs in time $O(nm)$*

Proof: All the work of the algorithm can be accounted for in the calls to the DFS(v) function. Actually, the work is constant for each DFS(v) call, so we just need to bound the number of such calls.

Each DFS(v) call returns either "no path found" or "path found". The former case can happen at most m times, becaues each time this happens an edge is declared useless, and there are at most m edges.

The "path found" calls come in groups of l where l is the number of levels in G_L. Every l of them will cause an edge to become saturated. So the total number of such calls is at most $lm \leq nm$. ∎

Ocaml Code for Dinic's Algorithm

```
(* Danny Sleator *)

let infinite_c = 1000000000 (* "infinite" capacity *)

let maxflow s t adj c =
  (* s and t are nodes (numbers). g is a graph (represented via an
     array of adjacency lists). c is a 1-d array of capacities.  The
     adjacency list on node v stores pairs like (w,i) where w is the
     other node, and i is the index into the capacity array for where
     this capacity is stored.  It's set up so that i lxor 1 is where the
     capacity of edge (w,v) is stored.  We compute the maximum flow in
     the same representation as the capacities.  *)

  let n = Array.length adj in
(*  let f = Array.make (Array.length c) 0 in *)
(*  let rc i = c.(i) - f.(i) in  (* residual capacity *) *)
  let alive = Array.make n [] in
  let level = Array.make n 0 in
  let queue = Array.make n 0 in
  let front = ref 0 in
  let back = ref 0 in

  let push w = queue.(!back) <- w; back := !back+1 in
  let eject() = let x = queue.(!front) in front := !front + 1; x; in

  let rec bfs () =
    let rec loop () =
      if !front = !back then false else
        let v = eject() in
          if level.(v) = level.(t) then true else (
            List.iter (
              fun (w,i) -> if level.(w) < 0 & c.(i) > 0 then (
                level.(w) <- level.(v) + 1;
                push w
              )
            ) adj.(v);
            loop ()
          )
    in
      level.(s) <- 0;
      back := 0; front := 0;
      push s;
      loop ();
  in
```

```
let rec dfs v cap =
  (* Does a DFS from v looking for a path to t.  It tries to push cap units
     of flow to t.  It returns the amount of flow actually pushed *)
  if v=t then cap else
    let rec loop total remaining li =
        match li with
          | [] ->
              alive.(v) <- [];
              total
          | (w,i)::tail ->
              if level.(w) = level.(v)+1 & c.(i) > 0 then (
                let p = dfs w (min c.(i) remaining) in
                  c.(i) <- c.(i) - p;
                  c.(i lxor 1) <- c.(i lxor 1) + p;
                  if remaining-p = 0 then (
                    alive.(v) <- li;
                    total+p
                  ) else loop (total+p) (remaining-p) tail
              ) else loop total remaining tail
    in loop 0 cap alive.(v)
in

let rec main_loop flow_so_far =
  for i=0 to n-1 do level.(i) <- -1; done;
  Array.blit adj 0 alive 0 n;
  if bfs () then main_loop (flow_so_far + (dfs s infinite_c)) else flow_so_far
in
  main_loop 0
```

C++ Code for Dinic's Algorithm

```
/* Neal Wu */
#include <cstdio>
#include <cstring>
#include <limits>
#include <algorithm>
using namespace std;
const int MAXV = 120002, MAXE = 155000, INF = 10000;

template <typename T> struct Dinic {
    int V, source, sink;
    int eind, eadj [2 * MAXE], eprev [2 * MAXE], elast [MAXV], start [MAXV];
    int front, back, q [MAXV], dist [MAXV];
    T CAPINF, ecap [2 * MAXE];
    int dfscount;

    inline Dinic () {
        V = -1;
        CAPINF = numeric_limits <T> :: max ();
    }

    inline void init (int v) {
        V = v; eind = 0;
        memset (elast, -1, V * sizeof (int));
        dfscount = 0;
    }

    inline void addedge (int a, int b, T cap1, T cap2) {
        eadj [eind] = b; ecap [eind] = cap1; eprev [eind] = elast [a]; elast [a] = eind++;
        eadj [eind] = a; ecap [eind] = cap2; eprev [eind] = elast [b]; elast [b] = eind++;
    }

    bool bfs () {
        memset (dist, 63, V * sizeof (int));
        front = back = 0;
        q [back++] = source; dist [source] = 0;

        while (front < back)
        {
            int top = q [front++];

            for (int i = elast [top]; i != -1; i = eprev [i])
                if (ecap [i] > 0 && dist [top] + 1 < dist [eadj [i]])
                {
                    dist [eadj [i]] = dist [top] + 1;
                    q [back++] = eadj [i];
                }
        }
        return dist [sink] < INF;
    }
    T dfs (int num, T pcap) {
      dfscount++;
        if (num == sink)
            return pcap;

        T total = 0;

        for (int &i = start [num]; i != -1; i = eprev [i])
            if (ecap [i] > 0 && dist [num] + 1 == dist [eadj [i]])
            {
                T p = dfs (eadj [i], min (pcap, ecap [i]));
                printf("push %d\n",p);
                ecap [i] -= p;
                ecap [i ^ 1] += p;
                pcap -= p;
                total += p;

                if (pcap == 0)
                    break;
            }

        return total;
    }

    T flow (int _source, int _sink) {
        if (V == -1)
            return -INF;

        source = _source; sink = _sink;
        T total = 0;

        while (bfs ())
        {
            memcpy (start, elast, V * sizeof (int));
            total += dfs (source, CAPINF);
        }
        printf("dfscount = %d\n", dfscount);
        return total;
    }
};
```