Washington University in St. Louis

Engineering

Max Flow Problem

Dinic's Algorithm

Jon Turner
Computer Science & Engineering
Washington University

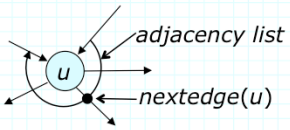
www.arl.wustl.edu/~jst

Dinic's Algorithm

- Shortest augmenting path algorithm often explores same "dead-ends" multiple times
  - » can avoid some redundant effort by ignoring edges that cannot possibly appear in augmenting paths of the "current length"
- Let f be flow function on G and let R be residual graph
  - » $level(u)$=number of edges in a shortest path from s to u in R
  - » only edges (u,v) for which $level(v)=level(u)+1$ can possibly be on a shortest augmenting path
- Dinic's algorithm restricts path search to those edges (u,v) for which $res(u,v)>0$ and $level(v)=level(u)+1$
  - » this can be done by explicitly constructing a subgraph of R with only these edges
  - » or, it can be done by ignoring edges that violate the condition during path searches

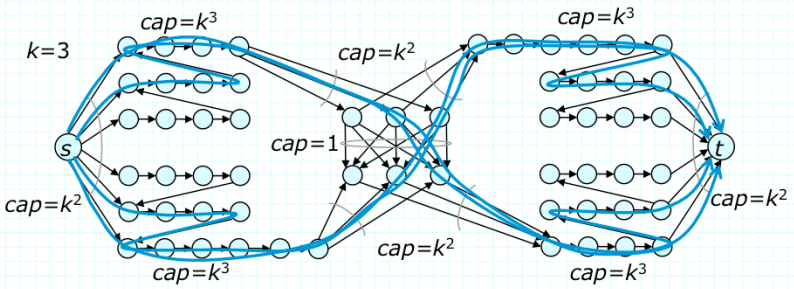
2

- Dinic's algorithm begins by setting $f(u,v)$ to zero for all edges in G, computes $level(u)$ for all vertices u and then repeats the following step as long as $level(t)<n$
  - » while there is an augmenting path using edges (u,v) with $level(v)=level(u)+1$, select such a path and saturate it
  - » recalculate $level(u)$ for all u
- The subroutine that finds augmenting paths uses depth-first search and maintains certain information across searches
  - » for each vertex u, maintain a pointer $nextedge(u)$ into adjacency list for u
  - » when a path search reaches u always continue search through $nextedge(u)$
  - » pointer is advanced past edges used for unsuccessful searches from u
  - » $nextedge(u)$ is reset to the start of the adjacency list every time *level* is recalculated



3

Performance on Hard Cases



- Shortest augmenting path algorithm examines most of the graph on every search – $\Omega(k^5)$ steps
- For each path length, Dinic does k^2 path searches, spending $O(k)$ time per search, so $O(k^4)$ altogether
  - » does *not* explore entire central subgraph on every search

Implementing Dinic's Algorithm

```
class dinic : public augPath { // inherits pEdge, augment()
public:
      dinic(Flograph&,int&);
      ~dinic();
private:
   int*  nextEdge;           // ignore edges before nextEdge[u]
   int*  level;              // level[u]=# of edges in path from source

   bool  findPath(vertex);   // find augmenting path
   bool  newPhase();         // prepare for a new phase
};

dinic::dinic(Flograph& fg1, int& floVal) : augPath(fg1,floVal) {
   level = new int[fg->n()+1];
   nextEdge = new edge[fg->n()+1];

   floVal = 0;
   while (newPhase()) {
      while (findPath(fg->src())) floVal += augment();
   }
}
```

at most *m* steps per phase

```
bool dinic::newphase() {
// Prepare for new phase. Return true if there is a source/sink path.
   vertex u,v; edge e;
   UiList q(G->n());

   for (u = 1; u <= fg->n(); u++) {
      level[u] = fg->n(); nextEdge[u] = fg->first(u);
   }
   q.addLast(fg->src()); level[fg->src()] = 0;
   while (!q.empty()) {
      u = q.first(); q.removefirst();
      for (e = fg->firstAt(u); e != 0; e = fg->nextAt(u,e)) {
         v = fg->mate(u,e);
         if (fg->res(u,e) > 0 && level[v] == fg->n()) {
            level[v] = level[u] + 1;
            if (v == fg->snk()) return true;
            q.addLast(v);
         }
      }
   }
   return false;
}
```

breadth-first search
$O(m)$ time

early return prevents useless exploration of vertices that are further from source than sink

```
bool dinic::findpath(vertex u) {
// Find a path to sink with positive residual capacity.
   while (nextEdge[u] != 0) {
      edge e = nextEdge[u];
      vertex v = fg->mate(u,e);
      if (fg->res(u,e) > 0 && level[v] == level[u] + 1 &&
           (v == fg->snk() || findpath(v))) {
         pEdge[v] = e; return true;
      }
      nextEdge[u] = fg->nextAt(u,e);
   }
   return false;
}
```

depth-first search

ignore ineligible edges

return path in *pEdge* and update *nextEdge*

on failure, nextEdge=0 blocking future searches

Analysis of Dinic's Algorithm

- Define a phase of Dinic's algorithm to be the period from one calculation of level function to the next
- *Theorem 8.4*. Dinic's algorithm executes $\leq n-1$ phases

  proof sketch. *Level*(t) increases by ≥ 1 after each phase; since no path is longer than $n-1$, there can be at most $n-1$ phases
- *Theorem 8.7*. Dinic's algorithm spends at most $O(mn)$ time per phase and finds a max flow in $O(mn^2)$ time

  Proof. Time per phase is $O(mn)$ excluding time for path searches. Let N_i be number of edges traced by i-th top level call to *findpath* in a phase. Time for all calls to *findpath* is proportional to $\Sigma_{1 \leq i \leq m} N_i$

  A *nextedge* pointer is advanced for every edge traced in i-th top level call that is not part of path returned by *findpath*. So pointers are advanced at least $(N_i - n)$ times during the i-th call. Thus,

  $$\Sigma_{1 \leq i \leq m}(N_i - n) \leq 2m \text{ and } \Sigma_{1 \leq i \leq m} N_i \text{ is } O(mn)$$

  So, $O(mn)$ time per phase and $O(mn^2)$ time overall ■

8

Dinic's Algorithm on Unit Networks

- In a *unit network* all edges have capacity one and every vertex other than s or t has either a single entering edge or a single outgoing edge
- *Theorem 8.5*. On unit network, Dinic's algorithm takes $O(m)$ time/phase and finds max flow in $O(mn^{1/2})$ time

  Proof. Time for i-th top level call to *findpath* in a phase is proportional to number of edges N_i that are traced

  A *nextedge* pointer is advanced for every edge traced in i-th call that is not part of the path returned by *findpath*

  So, if k_i is length of i-th augmenting path, *nextedge* pointers are advanced at least $(N_i - k_i)$ times during *findpath* and all k_i edges on path are saturated when flow is added

  Consequently, $\Sigma_i k_i \le m$ and $\Sigma_i(N_i - k_i) \le 2m$, so $\Sigma_i N_i \le 2m + \Sigma_i k_i \le 3m$ and thus, number of edges examined in all calls to *findpath* during a phase is $O(m)$

9

Now show there are at most $2\lceil (n-2)^{1/2} \rceil$ phases; suppose the we've completed k phases, and let f be current flow, f^* be a max flow and R be the residual graph for f.

The number of remaining phases is at most $|f^*|-|f|$, so the total number of phases is at most $k+|f^*|-|f|$. We can complete the analysis by finding an upper bound on this expression.

Note that f^*-f is a flow on R, R is a unit network and f^*-f is zero or one on every edge.

We can partition edges on which f^*-f is one into a collection of $|f^*|-|f|$ paths from s to t and possibly some cycles.

Since R is unit, any vertex other than s or t can be on at most one of these paths; hence, there is some augmenting path in R with at most $(n-2)/(|f^*|-|f|)+1$ edges

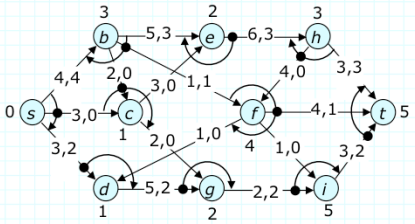
Since we've completed k phases, the next augmenting path will have at least $k+1$ edges, so

$$k+1 \le (n-2)/(|f^*|-|f|) + 1 \quad \text{or} \quad |f^*|-|f| \le (n-2)/k$$

Hence, the total number of phases is $\le k+(n-2)/k$, no matter what value k has. Choosing $k=\lceil (n-2)^{1/2} \rceil$ yields the desired result. ■

Exercises

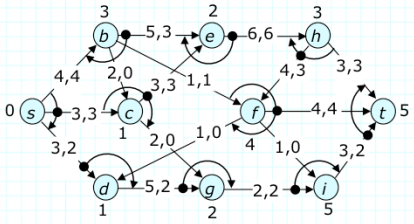
1. The diagram below shows an intermediate state in the execution of Dinic's algorithm. In the diagram, the numbers by the vertices represent the value of the level function. The arcs around each vertex indicate the order of the edges in the adjacency list, and the heavy dot on each arc shows the position of the next edge pointer. So, for example, the next edge at vertex *c* is (*b*,*c*).



Which augmenting path is selected next?
scehft

Show the state of the algorithm after flow is added to the next augmenting path.

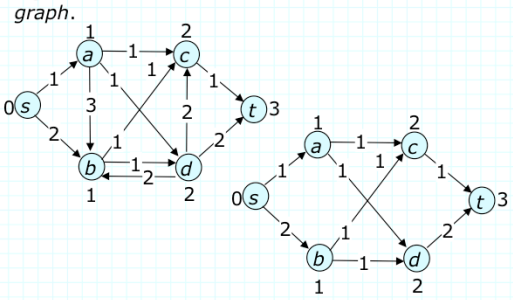
The new state appears below.



2. Explain how Dinic's algorithm can be used to compute a maximum size matching in a bipartite graph in $O(mn^{1/2})$ time.

We have seen earlier how a maximum size matching for a bipartite graph can be computed by solving a maximum flow problem. The flow problem used to solve the matching problem is a unit network. Therefore, if we solve it using Dinic's algorithm, the running time is $O(mn^{1/2})$ by the analysis on pages 9 and 10.

3. The left-hand diagram below shows a residual graph *R* for some flow. The number next to each vertex is its level value (that is, the number of edges in a shortest path from the source). The subgraph shown at right excludes the edges tat are ignored by Dinic's algorithm and is sometimes called the *level graph*.



We can view each phase of Dinic's algorithm as finding a *blocking flow* in the level graph. A flow is called a blocking flow, if every source-to-sink path contains some saturated edge. Find a blocking flow with a total value of 2 in the level graph. Show that this is not a maximum flow.

If we add one unit of flow to the path sact and one unit of flow to the path sbdt, we get a blocking flow.

To get a maximum flow, add one unit of flow to each of the three paths sadt, sbdt and sbct.