MAXFLOW: assign flows to edges that
  • equalize inflow and outflow at every vertex
  • maximize total flow through the network

MINCOST MAXFLOW: find the BEST maxflow

Mincost maxflow is important for two primary reasons
  it is a GENERAL PROBLEM-SOLVING MODEL
    • solves (through reduction) numerous practical problems
  it is TRACTABLE and PRACTICAL
    • we know fast algorithms that solve mincost flow problems
    • basic data structures play a critical role

One step closer to a single ADT for combinatorial problems

Distribution problem

SUPPLY vertices (produce goods)
DEMAND vertices (consume goods)
DISTRIBUTION points (transfer goods)

Feasible flow problem

  • Can we make supply to meet demand?

Distribution problem

  • Add costs, find the lowest-cost way

Ex: Walmart
Ex: McDonald’s

THM: Feasible flow reduces to maxflow
THM: Distribution reduces to mincost maxflow
Proof: Add source to provide supply, sink to take demand

Transportation problem

No distribution points

  • feasibility: is there a way?
  • transportation: find best way

Seems easier, but that is not the case (!)

THM: Maxflow reduces to maxflow for acyclic networks
THM: Transportation reduces to mincost maxflow
Mincost flow reductions

SHORTEST PATHS
MAXFLOW
DISTRIBUTION and TRANSPORTATION

ASSIGNMENT
Minimal weight matching in weighted bipartite graph

MAIL CARRIER
Find a cyclic path that includes each edge AT LEAST once

SCHEDULING (example)
Given a sport's league schedule, which teams are eliminated?

POINT MATCHING
Given two sets of N points, find minimal-distance pairing

ALL of these problems reduce to mincost flow

RESIDUAL NETWORK
for each edge in original network
• flow f in edge u–v with capacity c and cost x
define TWO edges in residual network
• FORWARD edge: capacity c–f and cost x in edge u–v
• BACKWARD edge: capacity f and cost –x in edge v–u

THM: A maxflow is mincost iff
there are NO negative-cost cycles in its residual network

GENERIC method for solving mincost flow problems:

start with ANY maxflow
REPEAT until no negative cycles are left
• increase the flow along ANY negative cycle

Implementation: use Bellman-Ford to find negative cycles

Cycle canceling example

Cycle canceling implementation

```
void addflow(link u, int d)
{ u->flow += d; u->dup->flow -=d; }

int GRAPHmincost(Graph G, int s, int t)
{ int d, x, w; link u, st[maxV];
  GRAPHmaxflow(G, s, t);
  while ((x = GRAPHnegcycle(G, st)) != -1)
  { u = st[x]; d = Q;
    for (w=u->dup->v; w != x; w=u->dup->v)
    { u = st[w]; d = ( Q > d ? d : Q ); } 
    u = st[x]; addflow(u, d);
    for (w=u->dup->v; w != x; w=u->dup->v)
    { u = st[w]; addflow(u, d); }
  }
  return GRAPHcost(G);
}
```
Cycle canceling analysis

No need to compute initial maxflow
  - use dummy edge from sink to source that carries maxflow

THM: Generic cycle canceling alg takes $O(VE^2CM)$ time
Proof:
  - each edge has at most capacity $C$ and cost $M$
  - total cost could be $ECM$
  - each augment reduces cost by at least 1
    Bellman-Ford takes $O(VE)$ time
There exist $O(VE^2 \log^2 V)$ cycle-canceling implementations
  - mincost maxflow is therefore TRACTABLE

EXTREMELY pessimistic UPPER bounds
  - not useful for predicting performance in practice
  - algs that achieve such bounds would be useless
  - algs are typically fast on practical problems

Network simplex algorithm

An implementation of the cycle-canceling algorithm

Identify negative cycles quickly by
  - maintaining a tree data structure
  - reweighting costs at vertices

Edge classification
  - EMPTY
  - FULL
  - PARTIAL

FEASIBLE SPANNING TREE
  - Any spanning tree that contains all the partial edges

VERTEX POTENTIALS
  - a set of vertex weights (vertex-indexed array $\phi$)

Network simplex concepts (continued)

REduced COST (rewighted edge cost)
  - $c^*(u, v) = c(u, v) - (\phi(u) - \phi(v))$
VALID vertex potentials for a spanning tree
  - all tree edges have reduced cost 0
ELIGIBLE EDGE
  - nontree edge that creates negative cycle with tree edges

THM: A nontree edge is eligible iff it is either
  - a full edge with positive reduced cost, or
  - an empty edge with negative reduced cost
Proof:
  - cycle cost equals cycle reduced cost
  - edge cost is negative of cycle reduced cost
    (since reduced costs of tree edges are all zero)

THEREFORE, it is easy to identify eligible edges

Network simplex algorithm

still a generic algorithm for the mincost flow problem

start with ANY feasible spanning tree
REPEAT until no eligible edges are left
  - ensure that vertex potentials are valid
  - add to the tree an eligible edge
  - increase the flow along the negative cycle formed
  - remove from the tree an edge that is filled or emptied

Problem: could have zero flow on cycle

THM: IF the algorithm terminates, it computes a maxflow

Implementation challenges
  - cope with zero-flow cycles
  - strategy to choose eligible edges
  - data structure to represent tree
Network simplex example

Computing vertex potentials (example)

Feasible spanning tree data structure

Operations to support

- compute valid vertex potentials
- find cycle created by nontree edge
- replace tree edge by nontree edge

use PARENT-LINK representation!

to compute vertex potentials

- start with root at potential 0
- for each vertex
  - follow parent links to vertex with known potential
  - recursively set each vertex potential on path
to make reduced edge costs 0

to follow cycle created by nontree edge u-v

- follow parent links from each to their LCA
to delete nontree edge that fills or empties
- REVERSE the parent links from u or v
Network simplex basic implementation

```c
#define R(u)  u->cost - phi[u->v]+phi[u->dup->v]
int GRAPHmincost(Graph G, int s, int t)
{ int v; link u, x, st[maxV];
  GRAPHinsertE(G, EDGE(t, s, M, 0, C));
  initialize(G, s, t, st);
  for (valid = 1; valid++; )
  {
    for (v = 0; v < G->V; v++)
      phi[v] = phiR(st, v);
    for (v = 0, x = G->adj[v]; v < G->V; v++)
      for (u = G->adj[v]; u!=NULL; u = u->next)
        if (R(u) < R(x)) x = u;
    if (R(x) == 0) break;
    update(st, augment(st, x), x);
  }
  return GRAPHcost(G);
}
```

Network simplex variations

**OBJECTIVES**
- guarantee termination
- reduce number of iterations
- reduce cost per iteration

**Eligible edge selection strategies**
- random
- find next
- queue of eligible edges

**Lazy vertex potential calculation**
**Tree representations**
- triply-linked, threaded

**Guided by practical performance, not worst-case bounds**
- DATA STRUCTURES are the key to good performance

**Different implementations for different reductions??**

**BOTTOM LINE**
- accessible code for powerful problem-solving model