Washington University in St. Louis

Engineering

Max Flow Problem

Dynamic Trees

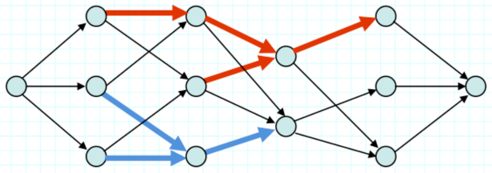
Jon Turner
Computer Science & Engineering
Washington University

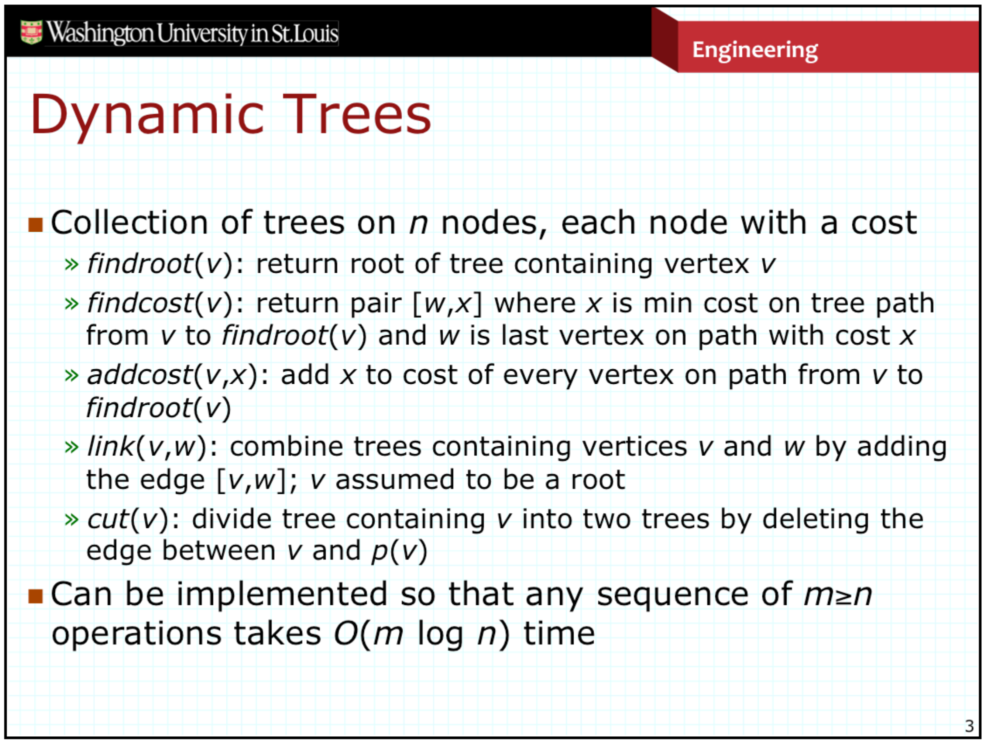
www.arl.wustl.edu/~jst

Washington University in St.Louis Engineering

Speeding Up Dinic's Algorithm

- Dinic's algorithm can waste time rediscovering path segments found previously
  - » Sleator & Tarjan showed how to maintain information about path segments with positive residual capacity across successive calls to *findpath*
- Partial paths stored in *dynamic trees* data structure
  - » represents a collection of trees
  - » used to represent subgraphs with unused residual capacity
  - » can largely eliminate retracing of previously discovered paths
  - » reduces time for each phase to $O(m \log n)$

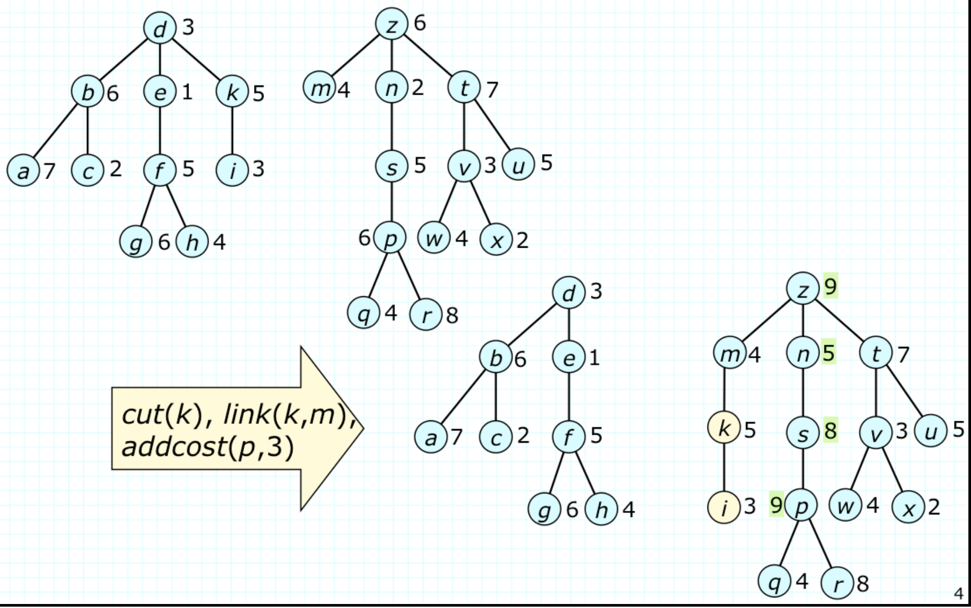




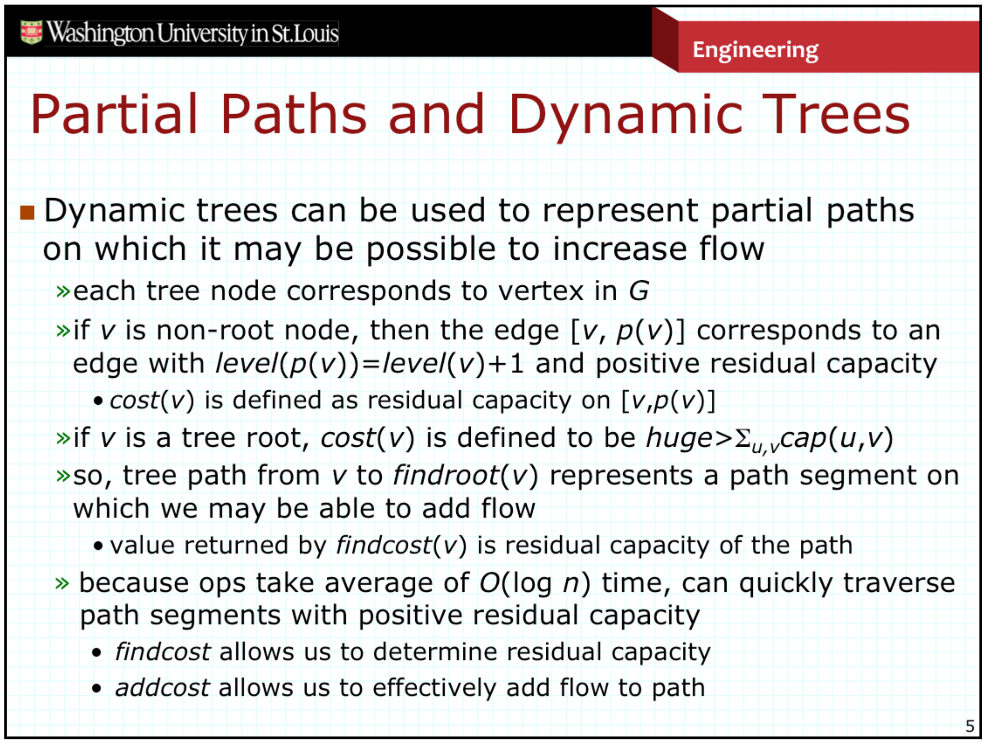
Dynamic Trees

- Collection of trees on *n* nodes, each node with a cost
  - » *findroot*(*v*): return root of tree containing vertex *v*
  - » *findcost*(*v*): return pair [*w*,*x*] where *x* is min cost on tree path from *v* to *findroot*(*v*) and *w* is last vertex on path with cost *x*
  - » *addcost*(*v*,*x*): add *x* to cost of every vertex on path from *v* to *findroot*(*v*)
  - » *link*(*v*,*w*): combine trees containing vertices *v* and *w* by adding the edge [*v*,*w*]; *v* assumed to be a root
  - » *cut*(*v*): divide tree containing *v* into two trees by deleting the edge between *v* and *p*(*v*)
- Can be implemented so that any sequence of $m \geq n$ operations takes $O(m \log n)$ time

Example Tree Ops

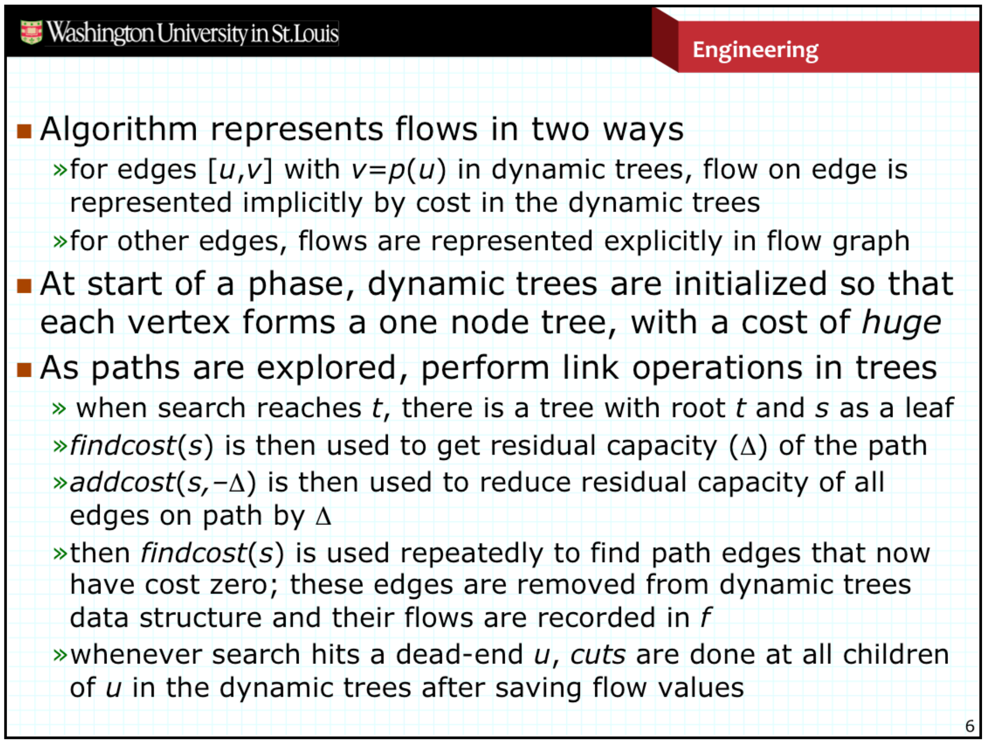


cut(*k*), *link*(*k*,*m*), *addcost*(*p*,3)



Partial Paths and Dynamic Trees

- Dynamic trees can be used to represent partial paths on which it may be possible to increase flow
  - » each tree node corresponds to vertex in *G*
  - » if *v* is non-root node, then the edge [*v*, *p*(*v*)] corresponds to an edge with *level*(*p*(*v*))=*level*(*v*)+1 and positive residual capacity
    - *cost*(*v*) is defined as residual capacity on [*v*,*p*(*v*)]
  - » if *v* is a tree root, *cost*(*v*) is defined to be $huge > \Sigma_{u,v} cap(u,v)$
  - » so, tree path from *v* to *findroot*(*v*) represents a path segment on which we may be able to add flow
    - value returned by *findcost*(*v*) is residual capacity of the path
  - » because ops take average of *O*(log *n*) time, can quickly traverse path segments with positive residual capacity
    - *findcost* allows us to determine residual capacity
    - *addcost* allows us to effectively add flow to path

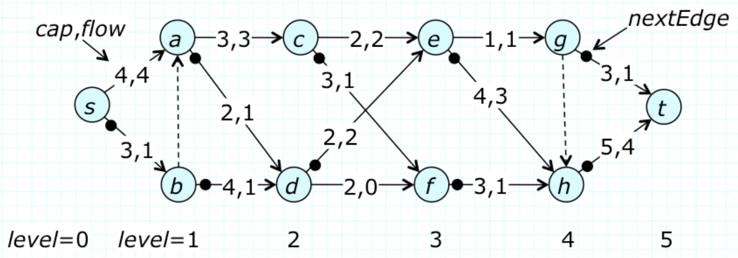


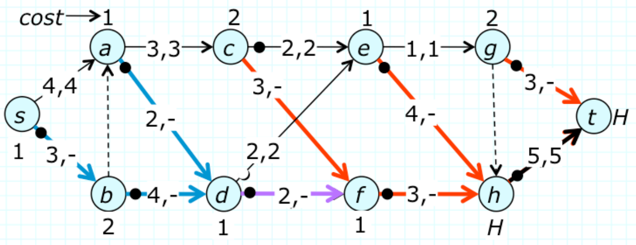
Washington University in St.Louis Engineering

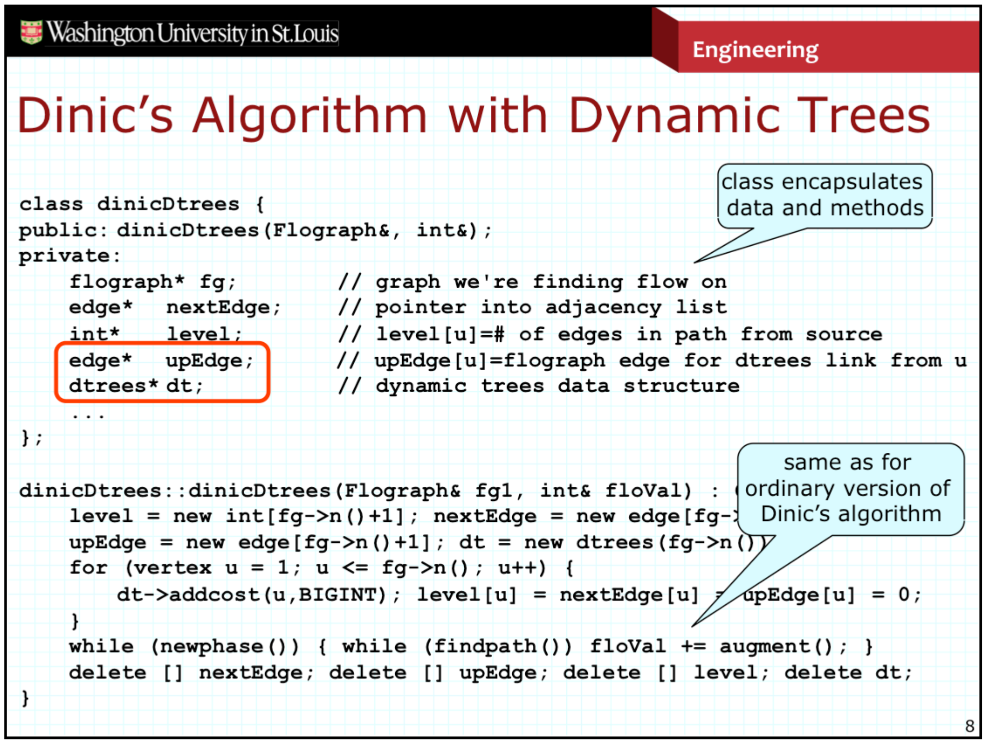
- Algorithm represents flows in two ways
  - »for edges [u,v] with v=p(u) in dynamic trees, flow on edge is represented implicitly by cost in the dynamic trees
  - »for other edges, flows are represented explicitly in flow graph
- At start of a phase, dynamic trees are initialized so that each vertex forms a one node tree, with a cost of *huge*
- As paths are explored, perform link operations in trees
  - » when search reaches t, there is a tree with root t and s as a leaf
  - »*findcost*(s) is then used to get residual capacity (Δ) of the path
  - »*addcost*(s,$-\Delta$) is then used to reduce residual capacity of all edges on path by Δ
  - »then *findcost*(s) is used repeatedly to find path edges that now have cost zero; these edges are removed from dynamic trees data structure and their flows are recorded in f
  - »whenever search hits a dead-end u, *cuts* are done at all children of u in the dynamic trees after saving flow values

6

Example


cap,flow
nextEdge
s
a
b
c
d
e
f
g
h
t
4,4
3,1
3,3
2,1
4,1
3,1
2,2
2,0
2,2
4,3
3,1
1,1
3,1
5,4
level=0
level=1
2
3
4
5


cost
1
2
1
2
s
a
b
c
d
e
f
g
h
t
H
4,4
3,-
3,3
2,-
4,-
3,-
2,2
2,-
2,2
4,-
3,-
1,1
3,-
5,5
1
2
1
1
H


Washington University in St. Louis

Engineering

Dinic's Algorithm with Dynamic Trees

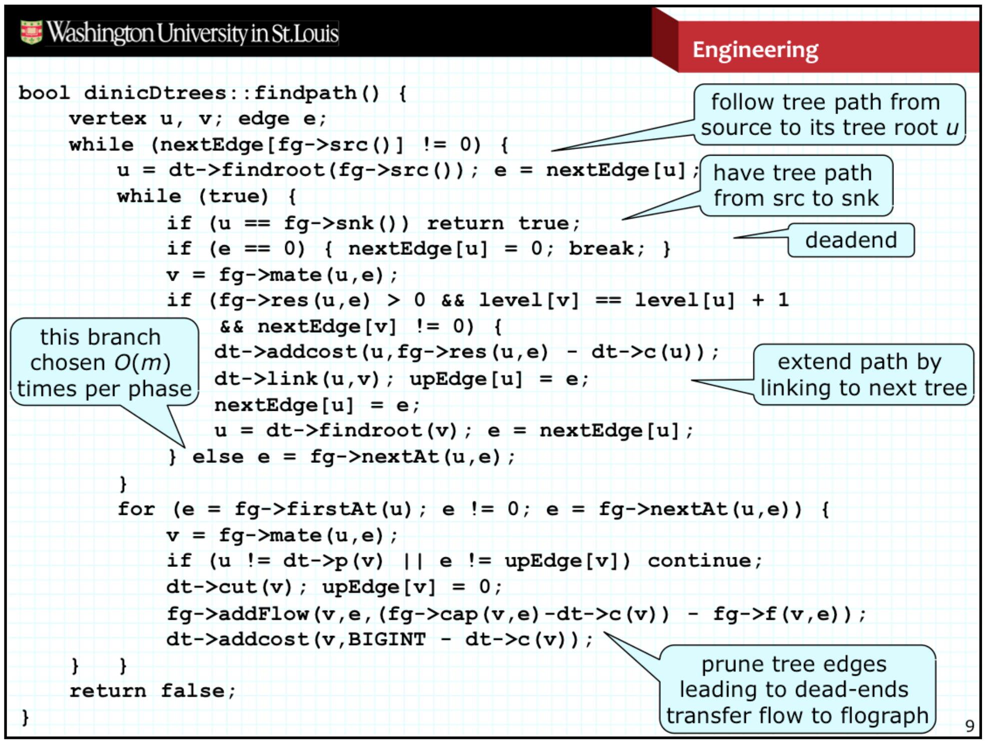
```
class dinicDtrees {
public: dinicDtrees(Flograph&, int&);
private:
    flograph* fg;          // graph we're finding flow on
    edge*   nextEdge;      // pointer into adjacency list
    int*    level;         // level[u]=# of edges in path from source
    edge*   upEdge;        // upEdge[u]=flograph edge for dtrees link from u
    dtrees* dt;            // dynamic trees data structure
    ...
};

dinicDtrees::dinicDtrees(Flograph& fg1, int& floVal) :
    level = new int[fg->n()+1]; nextEdge = new edge[fg->
    upEdge = new edge[fg->n()+1]; dt = new dtrees(fg->n())
    for (vertex u = 1; u <= fg->n(); u++) {
        dt->addcost(u,BIGINT); level[u] = nextEdge[u] = upEdge[u] = 0;
    }
    while (newphase()) { while (findpath()) floVal += augment(); }
    delete [] nextEdge; delete [] upEdge; delete [] level; delete dt;
}
```

class encapsulates data and methods

same as for ordinary version of Dinic's algorithm

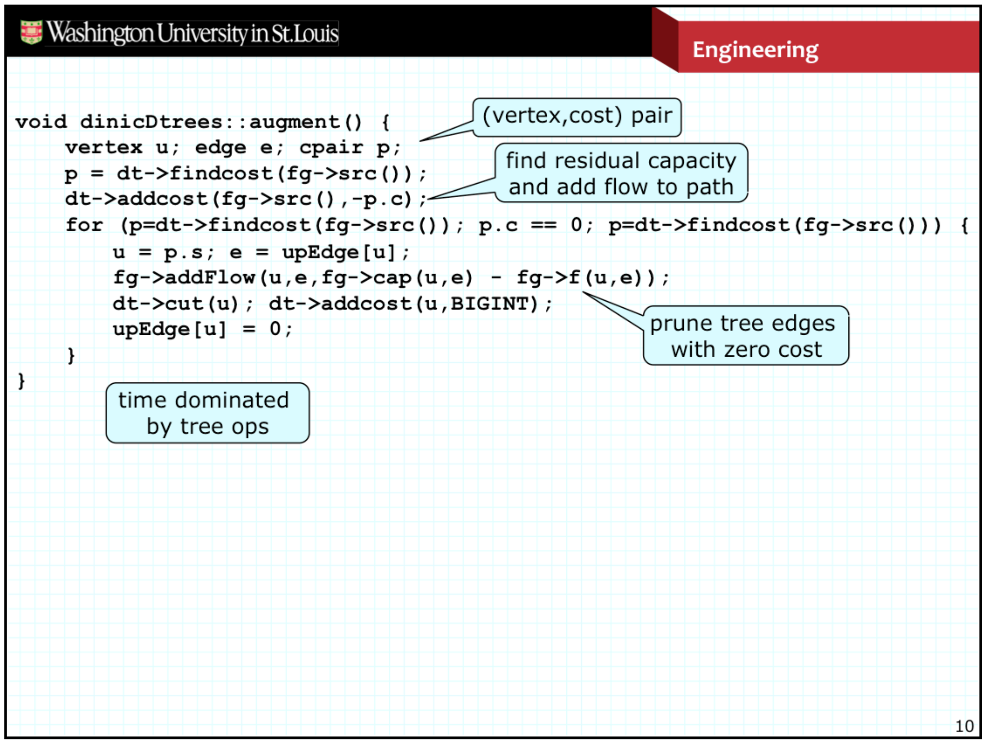
8



```
bool dinicDtrees::findpath() {
    vertex u, v; edge e;
    while (nextEdge[fg->src()] != 0) {
        u = dt->findroot(fg->src()); e = nextEdge[u];
        while (true) {
            if (u == fg->snk()) return true;
            if (e == 0) { nextEdge[u] = 0; break; }
            v = fg->mate(u,e);
            if (fg->res(u,e) > 0 && level[v] == level[u] + 1
                && nextEdge[v] != 0) {
                dt->addcost(u,fg->res(u,e) - dt->c(u));
                dt->link(u,v); upEdge[u] = e;
                nextEdge[u] = e;
                u = dt->findroot(v); e = nextEdge[u];
            } else e = fg->nextAt(u,e);
        }
        for (e = fg->firstAt(u); e != 0; e = fg->nextAt(u,e)) {
            v = fg->mate(u,e);
            if (u != dt->p(v) || e != upEdge[v]) continue;
            dt->cut(v); upEdge[v] = 0;
            fg->addFlow(v,e,(fg->cap(v,e)-dt->c(v)) - fg->f(v,e));
            dt->addcost(v,BIGINT - dt->c(v));
    }   }
    return false;
}
```

follow tree path from source to its tree root u
have tree path from src to snk
deadend
this branch chosen $O(m)$ times per phase
extend path by linking to next tree
prune tree edges leading to dead-ends transfer flow to flograph
9

rewrite to update nextEdge on each iteration. Use explicit loop test.



```
void dinicDtrees::augment() {
    vertex u; edge e; cpair p;
    p = dt->findcost(fg->src());
    dt->addcost(fg->src(),-p.c);
    for (p=dt->findcost(fg->src()); p.c == 0; p=dt->findcost(fg->src())) {
        u = p.s; e = upEdge[u];
        fg->addFlow(u,e,fg->cap(u,e) - fg->f(u,e));
        dt->cut(u); dt->addcost(u,BIGINT);
        upEdge[u] = 0;
    }
}
```

(vertex,cost) pair

find residual capacity and add flow to path

prune tree edges with zero cost

time dominated by tree ops

```
int dinicDtrees::newphase() {
    vertex u, v; edge e;
    list q(fg->n());
    for (u = 1; u <= fg->n(); u++) {
        nextEdge[u] = fg->first(u);
        if (dt->p(u) != 0) { // cleanup from previous pha
            e = upEdge[u];
            fg->addFlow(u,e,(fg->cap(u,e)-dt->c(u)) - fg->f(u,e));
            dt->cut(u); dt->addcost(u,BIGINT - dt->c(u));
            upEdge[u] = 0;
        }
        level[u] = fg->n();
    }
    q.addLast(fg->src()); level[fg->src()
    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; q.addLast(v);
                if (v == fg->snk()) return level[v];
    }   }   }
    return 0;
}
```

reset nextEdge to start of adjacency list

move flow information from dtrees to flograph, and clear dtrees

initialize level before new values computed

breadth-first search to initialize level values

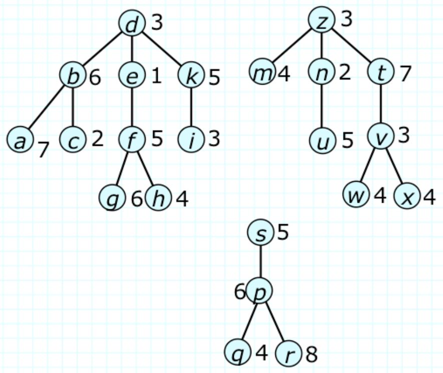
$O(m+n)$ time excluding tree ops
$O(n)$ tree ops

Analysis of Dinic with Dynamic Trees

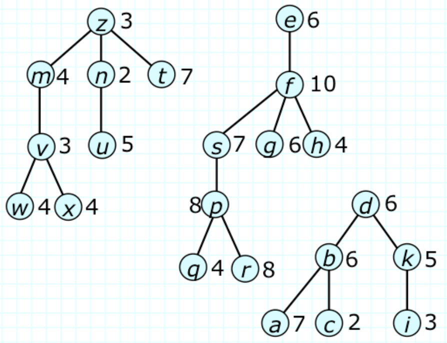
- Running time per phase is $O(m+\#\text{of tree ops})$ if we count each tree op as taking constant time
- Number of tree ops per phase: $O(\#$ of *links*$+\#$ of *cuts*$)$
  - for each edge (u,v), there is at most one link per phase and one cut per phase
  - thus, $O(m)$ tree ops per phase
- The dynamic tree operations can be implemented so that m operations take $O(m \log n)$ time
- *Theorem 8.10*. Dinic's algorithm with dynamic trees completes each phase in $O(m \log n)$ time and finds a max flow in $O(mn \log n)$ time

Exercises

1. The diagram below shows a collection of trees in a dynamic trees data structure.

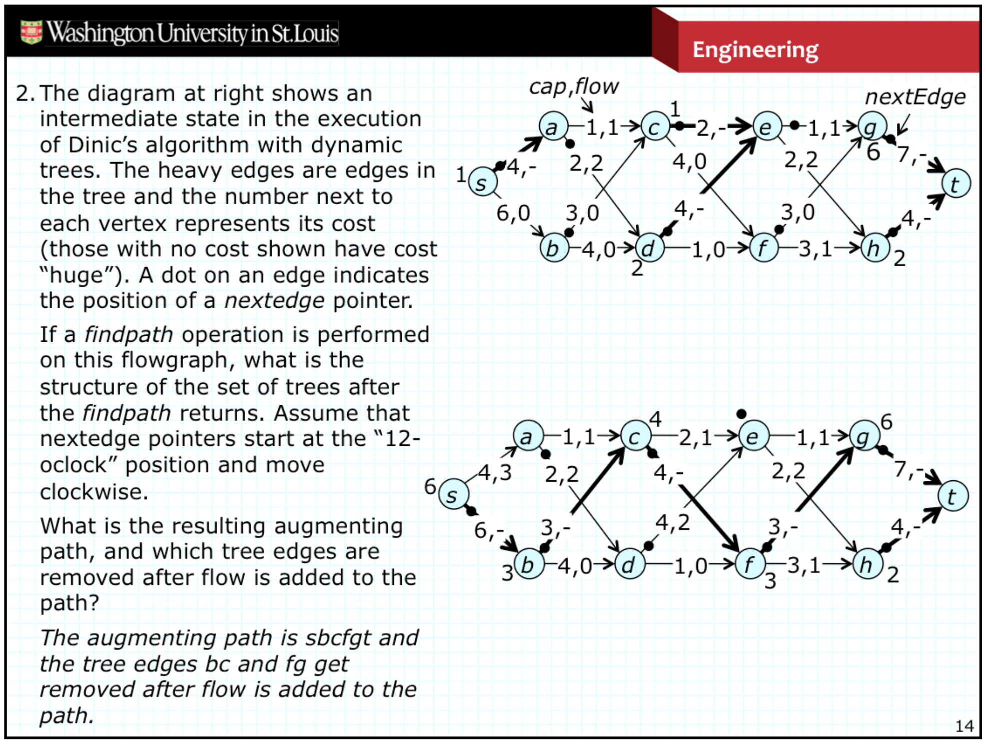


Show the new collection results after the following operations are performed. *Addcost*(*f*,3), *cut*(*v*), *cut*(*e*), *link*(*v*,*m*), *link*(*s*,*f*), *addcost*(*p*,2).



What value is returned if we now perform *findcost*(*x*)?

[*z*,3]



Washington University in St.Louis

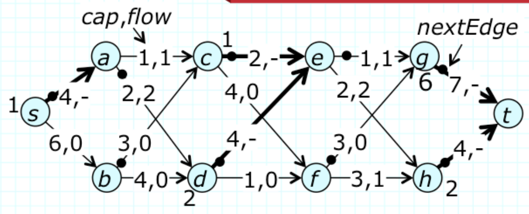
Engineering

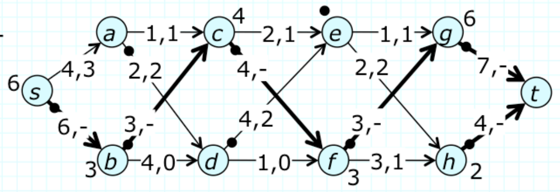
2. The diagram at right shows an intermediate state in the execution of Dinic's algorithm with dynamic trees. The heavy edges are edges in the tree and the number next to each vertex represents its cost (those with no cost shown have cost "huge"). A dot on an edge indicates the position of a *nextedge* pointer.

   If a *findpath* operation is performed on this flowgraph, what is the structure of the set of trees after the *findpath* returns. Assume that nextedge pointers start at the "12-oclock" position and move clockwise.

   What is the resulting augmenting path, and which tree edges are removed after flow is added to the path?

   The augmenting path is sbcfgt and the tree edges bc and fg get removed after flow is added to the path.





14

3. The diagram below is a bad case for the shortest augmenting path algorithm.

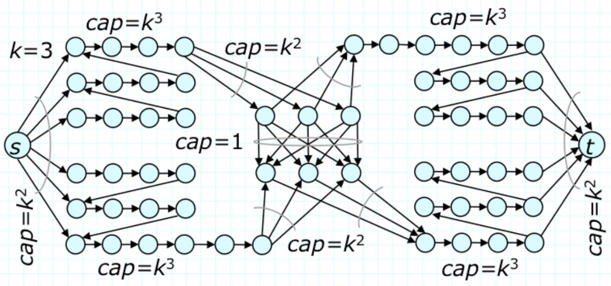
   If we apply Dinic's algorithm with dynamic trees to this graph, show which edges are in the set of trees after the first augmenting path is found?

   The edges in the dynamic trees are shown in bold at bottom right

   How does the dynamic tree structure change after the second augmenting path is found?

   The edge in the central bipartite graph drops out, and the next edge at the same vertex is added to the tree, along with the edge from its other endpoint to the first vertex in the lower right chain.

   In general, just a few edges change in each step within a phase.


$k=3$
$cap=k^3$
$cap=k^2$
$cap=k^3$
s
$cap=1$
t
$cap=k^2$
$cap=k^2$
$cap=k^3$
$cap=k^2$
$cap=k^3$