Washington University in St. Louis

Engineering

The Round Robin Algorithm for MSTs and Leftist Heaps

Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst

The Round-Robin Algorithm

- Start with *n* blue trees (one for each vertex) then merges trees using following rule
  - »*Coloring rule 3*: select a blue tree and a min cost edge incident to it; color the edge blue
- To minimize time, select blue trees using "round-robin" strategy

```
procedure minspantree(graph G=(V,E);  modifies set blue);
    vertex u,v; list queue;  partition(V); blue:={}; queue :=[];
    for u∈[1..n] ⇒ queue := queue & [u]; rof;
    do |queue|>1 ⇒
        Let {u,v} be a min cost edge incident to the tree that
        contains queue(1)
        blue := blue∪{{u,v}};  queue := queue–{find(u), find(v)};
        link(find(u),find(v)); queue := queue & [find(u)];
    od;
end;
```

Selecting Edges with Meldable Heaps

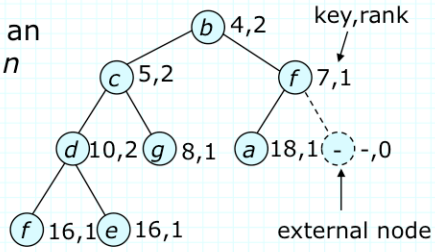
- To select min cost edge incident to a tree
  - » for each tree, maintain heap of all incident edges
  - » need fast way to combine heaps as we combine trees (meld)
  - » must remove "internal" edges created as heaps are combined

```
procedure minspantree(graph G=(V,E); modifies set blue);
   vertex u,v; list queue; mapping h:vertex→heap;
   partition(V); blue := {}; queue := [ ];
   for u∈[1..n] ⇒ queue:=queue & [u]; h(u):=makeheap(edges(u)); rof;
   do |queue| > 1 ⇒
        {u,v} := findmin(h(queue(1)));
        blue := blue∪{{u,v}}; queue:=queue−{find(u),find(v)};
        h(link(find(u),find(v))) := meld(h(find(u)),h(find(v)));
        Remove from heap all edges joining vertices in newly-formed tree
        queue := queue & [find(u)];
   od;
end;
```

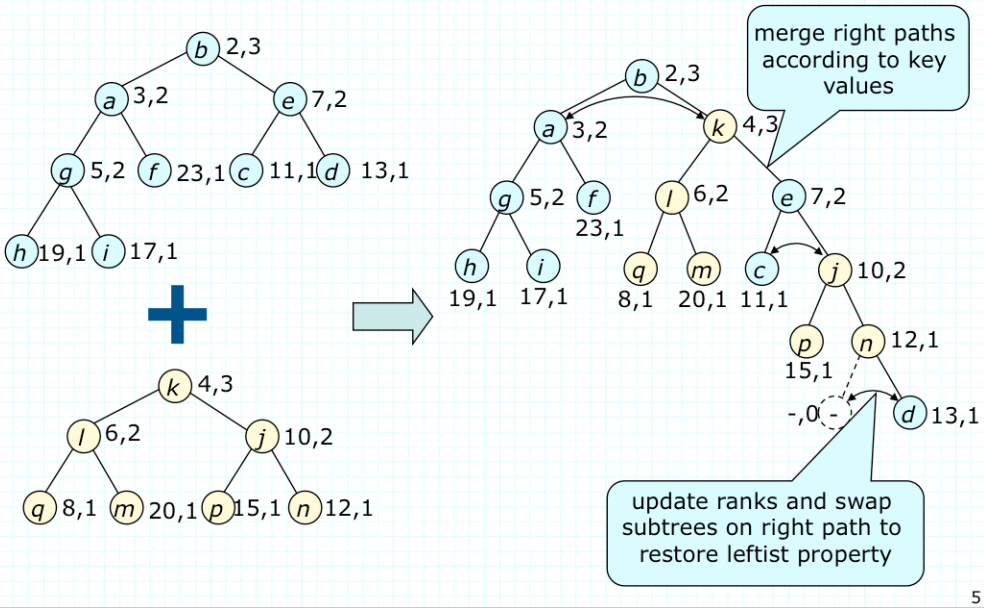
- $O(m \log \log n)$ time using "lazy" *leftist heaps*

Leftist Heaps

- Heap operation *meld*(h_1,h_2), combines the two heaps and returns resulting heap
- Can be implemented efficiently using *leftist heaps*
  - » if x is node in a *full binary tree*, define *rank*(x)=length of shortest path from x to a leaf that is a descendant of x
  - » a full binary tree is *leftist* if *rank*(*left*(x))≥*rank*(*right*(x)) for every internal node x
  - » the *right path* in a leftist tree is path from the root to the rightmost external node
    - » is a shortest path from root to an external node; has length ≤lg n
  - » a *leftist heap* is a leftist tree in heap order containing one item per internal node



Melding Leftist Heaps



Implementing Leftist Heaps

```
heap function meld(heap h1,h2);
    if h1 = null ⇒ return h2 | h2 = null ⇒ return h1 fi;
    if key(h1)>key (h2) ⇒ h1 ↔ h2; fi;
    right(h1) := meld(right(h1),h2);
    if rank(left(h1))<rank(right(h1)) ⇒ left(h1) ↔ right (h1) fi;
    rank(h1) := rank(right(h1))+1;
    return h1;
end;
```

note use of possibly null node $right(h_1)$

```
procedure insert(item i, modifies heap h);
    left(i) := null;  right(i) := null;  rank(i) := 1;
    h := meld(i,h);
end;

item function deletemin(modifies heap h);
    item i;  i := h;
    h := meld(left(h),right(h));
    return i;
end;
```

Heapify

- *heapify*(*q*) builds heap from heaps on list *q*

```
heap function heapify (list q);
    if q = [ ] ⇒ return null fi;
    do |q| ≥ 2 ⇒ q := q[3..] & meld(q(1),q(2)) od;
    return q(1)
end
```

- Time for heapify
  - » let *k*=number of heaps on *q* initially and let *r* be number of heaps on *q* after first ⌈*k*/2⌉ melds (*r*≤*k*/2)
  - » if n_i=size of *i*-th heap after first pass, the first pass time is $O(\lg n_1 + \ldots + \lg n_r) = O(r \lg(n/r))$ since 2≤n_i≤ *n* and Σn_i=*n*,
  - » *heapify* time is

$$O\left(\sum_{j=1}^{\lfloor \lg k \rfloor} (k/2^j)\lg(2^j n/k)\right) = O\left(k\sum_{j=1}^{\lfloor \lg k \rfloor} (j/2^j) + (1/2^j)\lg(n/k)\right)$$

$$= O\left(k(1+\lg(n/k))\right)$$

7

Makeheap and Listmin

- To build a heap in $O(n)$ time from a list of n items,

```
heap function makeheap(set s);
  list q;  q := [ ];
  for i∈s ⇒ left(i),right(i) := null;  rank(i):=1;  q:=q & [i]; rof;
  return heapify(q)
end;
```

- Operation *listmin*(x,h) returns a list containing all items in heap h with keys ≤ x

```
list function listmin(real x, heap h);
  if h = null or key(h) > x ⇒ return [ ]; fi;
  return [h] & listmin(x,left(h)) & listmin(x,right(h));
end;
```

Running time is proportional to number of items listed

Lazy Melding and Deletion

- It's often possible to improve performance of algorithms by postponing certain operations
  - »to implement lazy melding and deletion, add *deleted* bit to nodes
    - delete node by setting bit, meld two heaps by making them children of a dummy node with deleted bit set
  - »alternatively, call *deleted function* to determine node status
  - »remove deleted nodes during *deletemin* and *findmin* operations

```
item function deletemin(modifies heap h);
    item i; h := heapify(purge(h)); i := h; h := meld(left(h),right(h));
    return i
end;
list function purge(heap h);
    if h = null ⇒ return [ ];
     | h ≠ null and not deleted(h) ⇒ return [h]
     | h ≠ null and deleted(h) ⇒ return purge(left(h)) & purge(right(h))
    fi;
end;
```

purge() time
=*O*(1 + length of returned list)
=*O*(1 + # of calls to *deleted*())
assuming *deleted*() takes constant time

Analysis of Round Robin

- Use implicit deletion using deleted function

  predicate deleted(**edge** *e*); **return** find(*left*(*e*))=find(*right*(*e*)); **end**;
- Use deleted bit for lazy melding
- Divide the algorithm into passes as follows
  - » pass zero ends after every tree that was on the queue initially has been removed and combined with some other tree.
  - » pass j ends after every tree that was on the queue at the end of pass $j-1$ has been removed and combined with some other tree
- By induction on j, each tree that is on queue during pass j contains at least 2^j vertices, so $\leq \lfloor \lg n \rfloor$ passes
- Let m_i = # of edges in the heap selected in i-th step
- *Lemma 6.2.* $\sum_{i=1}^{n-1} m_i \leq (2m+n-1)\lfloor \lg n \rfloor$

  Proof. Trees chosen in same pass are vertex disjoint, so the number of edges in *all* the associated heaps is $\leq 2m+n-1$ ■

- *Findmin* time dominated by *heapify* time + # of find ops
  - » let k_i=# of nodes removed from heap by findmin in the i-th step
  - » excluding the *find*s the time for the i-th *findmin* is

$$O((k_i+1)(1+\lg(m_i/(k_i+1))))$$

- Call a *findmin small* if $k_i+1<m_i/(\lg n)^2$, else call it *large*
  - » time for all the small *findmin*s (excluding *find*s) is

$$O\left(\sum_{i=1}^{n-1}\frac{m_i}{(\lg n)^2}(1+\lg m_i)\right)=O\left(\sum_{i=1}^{n-1}\frac{m_i}{\lg n}\right)=O(m)$$

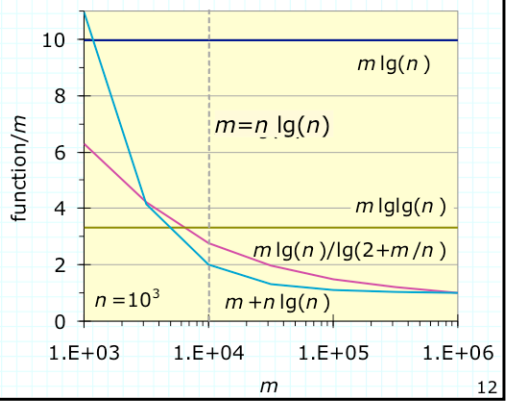
  - » time for all the large *findmin*s (excluding *find*s) is

$$O\left(\sum_{i=1}^{n-1}(k_i+1)\lg\frac{m_i}{m_i/(\lg n)^2}\right)=O\left(\sum_{i=1}^{n-1}k_i\lg\lg n\right)=O(m\lg\lg n)$$

- So, takes $O(m \lg \lg n)$ time excluding *find* ops
  - » so there are at most $O(m \lg \lg n)$ find operations
  - » by analysis of partition data structure, the find operations take $O((m \lg \lg n)\alpha(m \lg \lg n, n))=O(m \lg \lg n)$ time

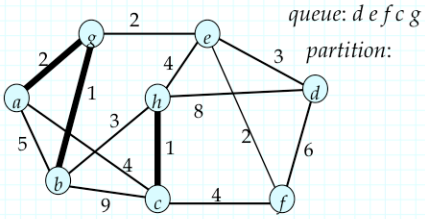
Implications for MST & Shortest Path

- Using Fheaps, Prim's algorithm and Dijkstra's algorithm take $O(m+n \log n)$ time
- Fheaps also enables multipass algorithm for MST that takes $O(m\beta(m,n))$ time where β grows very slowly
  - » not really a good option in practice
- Comparing MST options
  - » Prim's with F-heaps best at most densities
  - » round robin best at lowest densities
  - » other factors may dominate theoretical running time

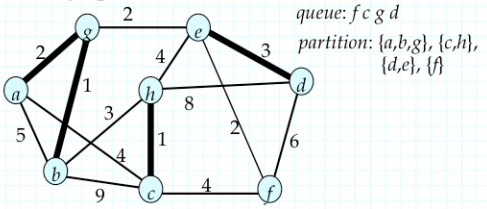


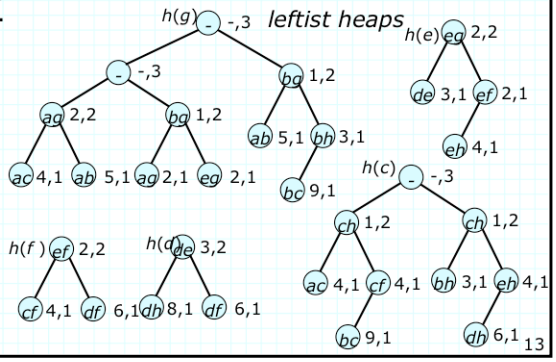
Exercises

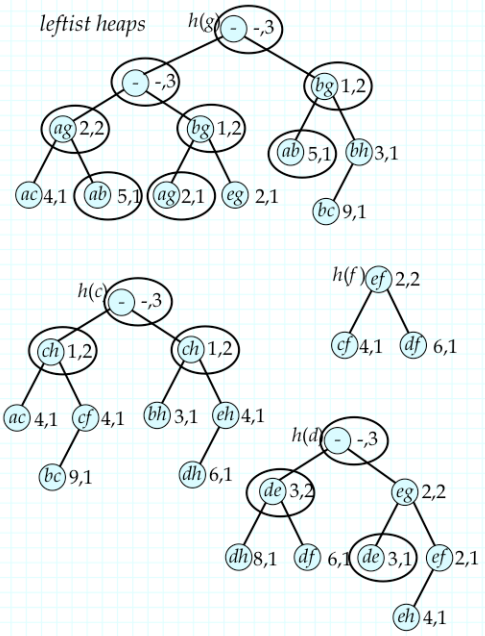
1. The figure below shows an incomplete representation of an intermediate state in the execution of the round-robin algorithm. Show the complete state of the algorithm after one more iteration, showing the state of the partition data structure as a collection of sets. Also, circle all the nodes in the leftist heaps that should be considered "deleted".



The graph, queue and partition are shown below. The leftist heaps are on the next page..







2. Let $f(n)$ be an integer function that satisfies the following property: for all integers i and j, the value of $f(j)$ does not lie above the line through $f(i)$ and $f(i+1)$. More precisely,

$f(j) \le f(i) + (j-i)\,D_i$ where $\Delta_i = f(i+1)-f(i)$

Show that if $n_1, n_2 \ge 0$ and $n_1 + n_2 = n$ then
$f(n_1) + f(n_2) \le f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil)$. Show that if
$n_1, \ldots, n_k \ge 0$ and $n_1 + \cdots + n_k = n$ then

$f(n_1) + \cdots + f(n_k) \le m\, f(\lceil n/k \rceil) + (k-m)\, f(\lfloor n/k \rfloor)$

where $m = n$ **mod** k.

To show the first part, note that the property satisfied by f implies that for all i,

$$f(i+2)-f(i+1) \le f(i+1)-f(i)$$

That is, successive differences in adjacent function values decrease (or at least, do not increase), which means that for all $i \le j$,

$$f(j+1)-f(j) \le f(i+1)-f(i)$$

This means, in particular, that if $n_1 < \lfloor n/2 \rfloor$

$$f(n_2)-f(n_2-1) \le f(n_1+1)-f(n_1)$$
$$f(n_1)+f(n_2) \le f(n_1+1)+f(n_2-1)$$

That is, the sum of the function values increases (or at least doesn't decrease) as we select pairs of function arguments that are more nearly equal. This implies $f(n_1) + f(n_2) \le f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil)$.

The second part follows from the same observation. For any pair $n_i < n_j$ we can only increase the sum, by replacing n_i and n_j with n_{i+1} and n_{j-1}. Applying this as long as possible yields

$$f(n_1) + \cdots + f(n_k) \le m\, f(\lceil n/k \rceil) + (k-m)\, f(\lfloor n/k \rfloor).$$

Let P be a partition on a set of r elements, with h subsets, $S_1, \ldots, S_h$. Suppose the running time of an algorithm on this partition is $g(|S_1|) + \cdots + g(|S_h|)$ where $g(n)=n^{1/2}$. Give an upper bound on the running time of the algorithm in terms of h and r.

If we treat g as an integer function, it clearly satisfies

$$g(j) \le g(i) + (j-i)\,\Delta_i \text{ where } \Delta_i = g(i+1)-g(i)$$

since its second derivative is negative. So,

$$g(|S_1|) + \cdots + g(|S_h|) \le m\, g(\lceil r/h \rceil) + (h-m)\, g(\lfloor r/h \rfloor) \le h\, g(\lceil r/h \rceil) \le h\, \lceil r/h \rceil^{1/2}$$

15

3. A portion of the C++ declaration of the leftist heap data structure is given below. List as many invariants as you can think of for this data structure.

```
typedef int keytyp, item;
class Lheaps {
public: ...
private:
  int n;
    struct node {
    keytyp keyf; int rankf;
    int leftf, rightf;
   } *vec;
};
```

$n \geq 0$
for $1 \leq i \leq n$, $0 \leq$ vec[i].left $\leq n$
for $1 \leq i \leq n$, $0 \leq$ vec[i].right $\leq n$
vec[0].leftf = vec[0].rightf = vec[0].rankf = 0
for $1 \leq i \leq n$, vec[vec[i].rightf].rankf $\leq$ vec[vec[i].leftf].rankf
for $1 \leq i \leq n$, vec[i].rankf = 1+ vec[vec[i].rightf].rankf
for $1 \leq i < j \leq n$, vec[i].leftf = vec[j].leftf $\Rightarrow$ vec[i].leftf = 0
for $1 \leq i < j \leq n$, vec[i].leftf = vec[j].rightf $\Rightarrow$ vec[i].leftf = 0
for $1 \leq i < j \leq n$, vec[i].rightf = vec[j].leftf $\Rightarrow$ vec[i].rightf = 0
for $1 \leq i < j \leq n$, vec[i].rightf = vec[j].rightf $\Rightarrow$ vec[i].rightf = 0
Let p(i)=j if i=vec[j].leftf or i = vec[j].rightf; if there is no such j, let p(i)=null. There is no sequence $i_1,\ldots,i_k$ such that $p(i_j)=p(i_{j+1})$ for $1 \leq j < k$ and $p(i_k)=p(i_1)$.