Enoch Hwang

✶ ehwang@cs.ucr.edu ✶

CS218 Design & Analysis of Algorithms

Notes

July 6, 1995

Sorting and Selection

Comparison Sorts

- Selection sort $O(n^2)$
- Insertion sort $O(n^2)$
- Heap sort O(n lg n)

  heap property - for every node *i* other than the root, A[parent(*i*)] ≥ A[*i*].

```
Heapsort(A)
Build-Heap(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
    heap-size[A] ← heap-size[A] - 1
    Heapify(A, 1)

Build-Heap(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1 do
    Heapify(A, i)

Heapify(A, i)
l ← left(i)
r ← right(i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i then
    exchange A[i] ↔ A[largest]
    Heapify(A, largest)
```

- Merge sort O(n lg n)
- Quicksort O(n lg n)

Linear Sorts

- Radix O(n)
- Bucket O(n)
- Counting O(n)

Recurrence (p 60)

- lg lg n lg n n n lg n n^2 n^3 2^n

$\lfloor$ + c $\rfloor$

$\lfloor$ + n $\rfloor$

• O(lg n)	$T(n) = T(n/3) + c$	
	$T(n) = T(n/2) + c$	
	$T(n) = T(2n/3) + c$	
• O(n)	$T(n) = 2T(n/2) + c$	
• $O(n^{\log_2 3})$	$T(n) = 3T(n/2) + c$	
• O(n)	$T(n) = T(n/2) + n$	$O(\lg n + n) \equiv O(n)$
	$T(n) = T(2n/3) + n$	
• O(n lg n)	$T(n) = 2T(n/2) + n$	$O(n \lg n + n) \equiv O(n \lg n)$
	$T(n) = T(n/3) + T(2n/3) + n$	
• $O(n^2)$	$T(n) = 4T(n/3) + n$	
	$T(n) = 4T(n/2) + n$	
	$T(n) = 2T(n/2) + n^2$	$O(n \lg n + n^2) \equiv O(n^2)$
• $O(n^{\log_2 7}) \equiv O(n^{2.8})$	$7T(n/2) + n^2$	
• $O(n^{\log_2 8}) \equiv O(n^3)$	$8T(n/2) + n^2$	

Medians and Order Statistics (p185)

- Selection problem can be solved in O(n)

Finding the minimum or maximum:

- upper bound of n -1 comparisons where n is the number of elements.

Finding the minimum and maximum simultaneously:

- upper bound of $3\lceil n/2 \rceil$ comparisons.

Randomized-Select returns the i^{th} smallest element of the array A[p..r]

worst-case running time is $\Theta(n^2)$

average running time is O(n)

```
Randomized-Select(A,p,r,i)      {A[p..r]}
if p = r
    then return A[p]
q := Randomized-Partition(A,p,r)                          {p ---- q ---- r}
                        {each element of A[p..q] < each element of A[q+1..r]}
k := q-p+1              {finds number k of elements in A[p..q]}
if i <= k
    then return Randomized-Select(A,p,q,i)
    else return Randomized-Select(A,q+1,r,i-k)
```

Deterministic-Select returns the i^{th} smallest of an input array of n elements

- worst-case running time is O(n)

Divide the n elements into $\lfloor n/5 \rfloor$ groups of 5 elements.
Find the median of each of the groups.
Use Select recursively to find the median x of the $\lceil n/5 \rceil$ medians found above
Partition the array around the median-of-median x
    Let k be the number of elements on the low side of the partition,
        n-k is the number of elements on the high side.
Use Select recursively to find the ith smallest element on the low side if $i \le k$
    or the (i-k)th smallest element on the high side if $i > k$.

Hashing

- Direct addressing O(1)

Collision resolution by chaining (p223)

- Given a hash table T with *m* slots that stores *n* elements.
- load factor α for T is *n*/*m*.
- worst-case time for searching if all n keys hash to the same slot -- $\Theta(n)$
- Assumption of simple uniform hashing (any given element is equally likely to hash into any of the m slots.):
- average time for an unsuccessful search is $\Theta(1+\alpha)$
- average time for a successful search is $\Theta(1+\alpha)$

Hash functions (p226)

The division method

$h(k) = k \bmod m$.

- m should be a prime that is far from a power of 2 number.

The multiplication method

$h(k) = \lfloor m(kA \bmod 1) \rfloor$.

$0<A<1$. A good A is $A \approx (\sqrt{5}-1)/2 = 0.6180339887...$

Advantage of this method is that the value of m is not critical.

"kA mod 1" means take the fractional part of kA.

Universal hashing

- Choose the hash function randomly.

Open addressing (p232)

- All elements are stored in the hash table itself.
- m is the size of the hash table T.

Linear probing

$h(k,i) = (h'(k) + i) \bmod m$

- Initial probe position determines the entire probe sequence.
- Only m distinct probe sequences are used.
- h'(k) is going to be a constant because you will do something like k mod 13 = c
- Problem of **primary clustering**. Long runs of occupied slots have a high probability to grow even larger.

Quadratic probing

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

- Initial probe position determines the entire probe sequence.
- Only m distinct probe sequences are used.
- Problem of **secondary clustering**. If two keys have the same initial probe position, then their probe sequences are the same.

Double hashing

$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

- The probe sequence depends in two ways upon the key k, the initial probe position and the offset.
- m^2 probe sequences are used ⇒ better than Linear or Quadratic.
- Choose m prime, and m' slightly less than m
  eg. $h_1(k) = k \bmod m$ $= k \bmod 13$
  $h_2(k) = 1 + (k \bmod m')$ $= 1 + (k \bmod 11)$

Binary Search Trees (p244)

Search(k)

```
while x != nil and k != key[x]
    if k < key[x]
        then x == left[x]
        else x == right[x]
return x
```

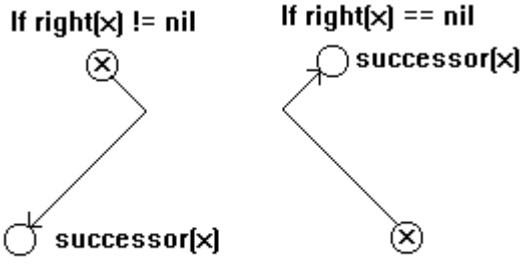
Tree_Minimum(x)

keep going LEFT until NIL.

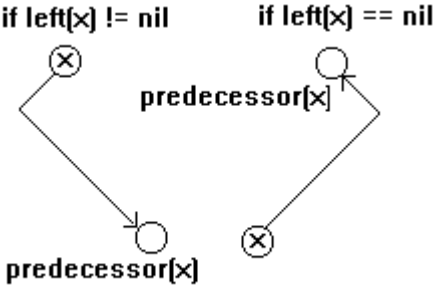
Tree_Maximum(x)

keep going RIGHT until NIL.

Tree_Successor(x)



Tree_Predecessor(x)



Insert(x)

Search(x)
Add to leaf

Delete(z)

3 cases:

1. z has no children, modify its parent p[z] to replace z with nil as its child
2. z has one child, splice out z by making a new link between its child and its parent.
3. z has two children, splice out z's successor y, (which has no left child) and replace the contents of z with the contents of y.

Time Complexity

Worst case is O(n) when tree has only 1 branch.
Average case is O(lg n) when tree is balanced.

Balanced binary search tree

Red-Black tree **O(lg n)** (p263)

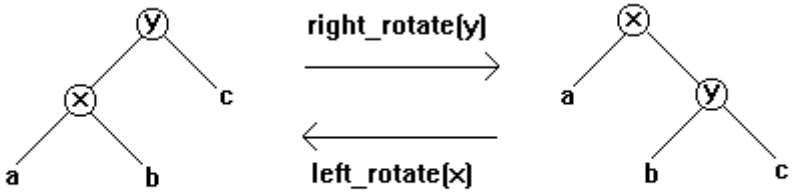
Definition: A binary search tree is a red-black tree if

1. Every node is either red or black.
2. Every leaf (NIL) is black.
3. If a node is red, then both its children are black.

4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

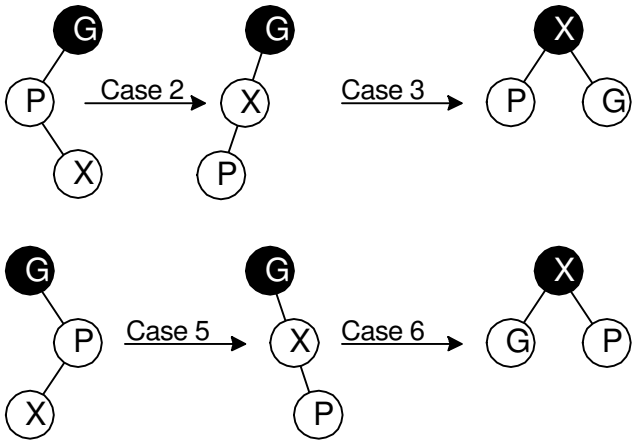
- black height of a node is the number of black nodes on the path from the node to the leaf.
- A rb tree with n internal nodes has height <= 2 lg n.
  Therefore worst case is O(lg n) for Search, Insert, & Delete

Rotation **O(1)**



Insertion(x)

Do ordinary binary search tree insert.
Color x red.
If parent(x) is black then done else
  Fix tree by doing rotations up the tree as follows. Six (3x2 symmetric) cases:
  while (x != root) and (parent(x) is red)
  At this point parent(x) is red, grandparent(x) is black, and all nils are black
  if parent(x) == left(grandparent(x))
    Case 1 uncle(x) is red.
      - color parent(x) black
      - color uncle(x) black
      - color grandparent(x) red
      - x = grandparent(x)
      - repeat while loop
    Case 2 uncle(x) is black
      x is the right child of parent(x)
      - x = parent(x)
      - left_rotate(x)
      - continue with case 3(6)
    Case 3 uncle(x) is black
      x is the left child of parent(x)
      - color parent(x) black
      - color grandparent(x) red
      - right_rotate(grandparent(x))
      - the while loop terminates
  if parent(x) == right(grandparent(x))
    Cases 4,5,6 same as 1,2,3 with "right" and "left" exchanged.
  end while
  - color root black



Disjoint sets (p441)

n = number of MakeSet operations, *m* = number of MakeSet, FindSet & Union operations.

Linked List Representation

- Make-Set O(1)
- Find-Set O(1)
- Union $O(m^2)$
- With weighted-union O(m + nlgn)

Tree Representation

- Make-Set O(1)
- Find-Set O(1)
- Union by rank and path compression $O(m\ \alpha(m,n))$ (p445)
  $\alpha(m,n)$ is the Ackermann's function ≤ 4 (a constant). Therefore, $\approx O(m)$.

```
Union(x, y)
    Link(Find-Set(x), Find-Set(y))

Link(x, y)
    if rank[x] > rank[y] then
        p[y] ← x
    else
        p[x] ← y
        if rank[x] = rank[y] then
            rank[y] = rank[y] + 1

Find-Set(x)
    if x ≠ p[x] then
        p[x] ← Find-Set(p[x])
    return p[x]
```

Graph Algorithms

Breath First Search (BFS) O(E+V) (p469)

- shortest path from *s*, use a queue, no recursion, one tree, not weighted.

```
Q = {s}
color all vertices white
while Q ≠ ∅ do
    u = head[Q]
    for each v ∈ Adj[u] do
        if color v = white then
            d[v] = d[u] + 1
            color[v] = gray
            Enqueue (Q, v)
    Dequeue(Q)
    color[u] = black
```

Depth First Search (DFS) O(E+V) (p477)

- use recursion, a stack, forest

```
DFS
    color all vertices white
    for each vertex u do
        if color[u] = white then
            DFS-Visit(u)

DFS-Visit(u)
    color[u] = grey
    starttime = time = time + 1
    for each v ∈ Adj[u] do
        if color[v] = white then
            DFS-Visit(v)
        else
            print "A backedge from v to u creating a cycle"
    color[u] = black
    finishtime = time = time + 1
```

Topological sort O(E+V) (p485)

- call DFS(G)
  when vertex is finished, insert to front of linked list.

Strongly connected components O(E+V) (p488)

- the maximal set of vertices such that every u and v are reachable from each other.

```
call DFS(G)
compute G^T     /* O(E+V) */     Transpose of G
call DFS(G^T)   /* in main loop of DFS, consider vertices in order of decreasing finish time */
output the vertices of each tree as strongly connected components
```

Minimum Spanning Tree problem

- shortest path that traverses all nodes.
- find a tree that spans the graph G where the sum of the edges is minimized.
- a Hamiltonian path where each vertex has a degree of at most 2.

Kruskal O(E lg E) (p504)

- a forest

```
A = 0
for each vertex v do
    Make-Set(v)
sort edges by increasing weight
for each edge (u,v) in order by increasing weight do
    if Find-Set(u) ≠ Find-Set(v) then
        A = A ∪ {(u,v)}
        Union(u,v)
return A
```

Prim **O(E lg V)** (p505)

- O(E + V lg V) if use Fibonacci heap
- a single tree
- like Dijkstra, idea like BFS

Single source shortest path

- shortest path from *s* to every other node.

Dijkstra **$O(V^2)$** (p527)

- weighted directed graph, weights cannot be negative, greedy, priority queue Q.
- use Relax

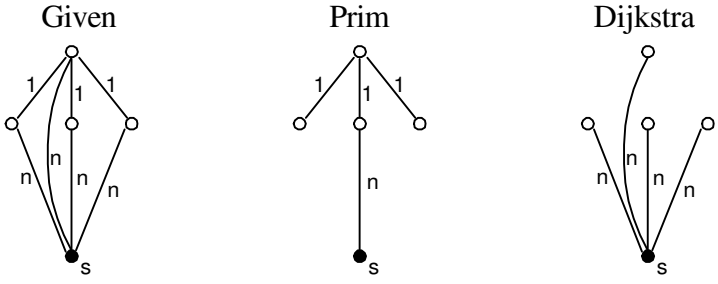
```
Dijkstra(G, w, s)
S = 0
Q = V[G]
while Q ≠ 0 do
    u = Extract-Min(Q)
    S = S ∪ {u}
    for each vertex v ∈ Adj[u]
        do Relax(u, v, w)

Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] ← d[u] + w(u, v)
        π[v] ← u
```

Bellman-Ford **O(VE)** (p532)

- edge weights can be negative

Difference between MST Prim and Dijkstra

- Prim is a MST - shortest path from *s* that traverses all nodes.
- Dijkstra is shortest path from *s* to every other node.
- The shortest path tree computed by Dijkstra is <u>not</u> a MST.
- If *T* is a MST, then the shortest path between any two vertices may <u>not</u> be in *T*.



All-pairs shortest-paths on a directed graph

Floyd-Warshall **$O(V^3)$** (p558)

- all pairs shortest path
- use matrixes, weighted, dynamic.

```
D(0) = w      // adjacency matrix of G
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            D(k)ij = min(D(k-1)ij , D(k-1)ik + D(k-1)kj)
```

Transitive closure $O(n^3)$ (p562)

- finds whether there is a path from i to j for all vertex pairs i,j

Flow Network (p580)

flow network G

- capacity constraint - each flow <= capacity of edge
- skew symmetry - flow into = - flow out of a vertex
- flow conservation - total net flow out of a vertex (other than s and t) is 0.
- source s, and sink t.
- value of a flow $|f|$ is the total net flow out of s.
- maximum-flow problem - find a flow of maximum value from s to t.
- residual capacity of an edge - $C_f(u,v) = c(u,v) - f(u,v) + f(v,u)$
- residual network G_f - network resulting from the residual capacity of all the edges
- augmenting path p is a simple path from s to t in the residual network G_f.
- residual capacity of p $C_f(p)$ is the minimum capacity of an edge in the augmenting path.
- for a given G, the flow of any cut is the same, however, the capacity is <u>not</u> the same.
- Max-flow min-cut theorem the following conditions are equivalent:
  1) f is a maximum flow in G.
  2) the residual network G_f contains no augmenting paths.
  3) $|f| = c(S, T)$ for some cut (S, T) of G = cut with minimum capacity.

Ford-Fulkerson(G,s,t) $O(E \times |f^*|) = O(E \times \text{maxflow})$ (p587)

- to find the max-flow problem

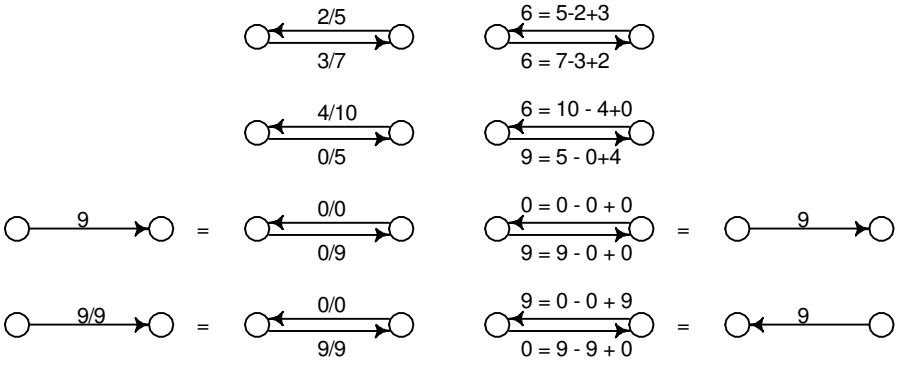
```
1   initialize flow f to 0
2   while there exists an augmenting path p do
3       augment flow f along p
```

```
1   for each edge (u,v)
1       f[u,v] = 0
1       f[v,u] = 0
2   while there exists a path p from s to t in the residual network Gf
3       cf(p) = min{cf(u,v) : (u,v) is in p}
3       for each edge (u,v) in p
3           f[u,v] = f[u,v] + cf(p)
3           f[v,u] = -f[u,v]
```

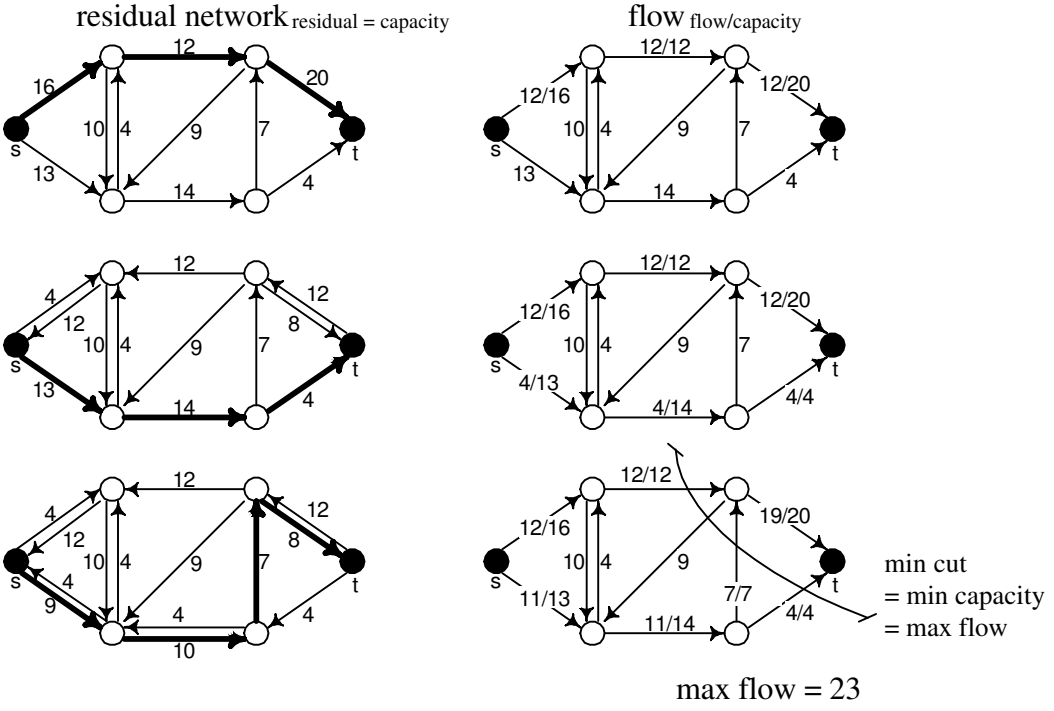
Edmonds-Karp algorithm $O(VE^2)$

- Same as Ford-Fulkerson except use breadth-first search in line 2 above, that is, pick the shortest augmenting path (from *s* to *t*) first, where each edge has unit distance (weight).

residual = capacity − flow + flow of other side



Example:



Design & Analysis techniques

- Strassen - matrix multiplication $O(n^{\lg 7}) = O(n^{2.8})$

Dynamic programming

- Matrix chain multiplication $O(n^3)$ (p307)
- CYK - recognition of context-free language $O(n^3)$
- Longest Common Subsequence problem $O(mn)$ (p314)
- Fibonacci numbers $O(n)$
- Tiling $O(n)$
- 0-1 Knapsack

Greedy Algorithms

- Activity-selection O(n lg n)
- Fractional Knapsack O(n lg n)

Number-Theoretic

- Prime & composit numbers
- Factorization - each number can be expressed in exactly one way as a product of primes.
- Euclid(a,b) - Greatest Common Divisor of a,b O(max(lg a, lg b)) (p809)

  if b = 0 then return a
  else return Euclid(b, a mod b)
- Relatively prime if gcd(a, b) = 1
- Groups (p814)

  Z_n (+ mod n) = {0, 1, .. n-1}
  $|Z_n| = n$
  Z_n^* (* mod n) = all numbers from 1 to n-1 that are relatively prime to n
  e.g. $Z_9^* = \{1,2,4,5,7,8\}$
  Euler function # = $|Z_n^*|$ = size of Z_n^* =

  n-1 if n is prime
  (p-1)(q-1) if n = pq, and p and q are prime
  - Euler Theroem: $a^{\text{Euler \#}} = 1 \pmod n$ for a in Z_n^*
  - Fermat's Theroem: if p is prime then $a^{p-1} = 1 \pmod p$ for all a in Z_p^*
  - Miller's Probabilistic Algorithm (p839)

Cryptography

- - P = public, (e,n) = $M^e \bmod n$; S = secret, (d,n) = $M^d \bmod n$
  - $P_B(M)$ - everyone can write but only Bob can read it
  - $S_B(M)$ - everyone can read but only Bob can write it = Bob's signature
  - to send secret message - send $P_B(M)$
  - authentication (did Bob sent it?) - send $[M, S_B(M)]$
  - to send secret signed message - send $[P_B(M, S_B(M))]$
- Key distribution problem
  - A, B agree on n and c where $1 < c < n$
  - A picks a secret number a B picks a secret number b
  - A sends $x = c^a \bmod n$ B sends $y = c^b \bmod n$
  - A computes key = $y^a \bmod n$ B computes key = $x^b \bmod n$
  - $y^a \bmod n = x^a \bmod n$ = our key

String Matching

Knuth-Morris-Pratt Algorithm O(m+n) (p869)

n = length(Text), m = length(Pattern)
use Prefix_Function π O(m)