Brian Chen
chen187
1002297034
University of Toronto, Scarborough Winter 2017 CSCB58

CSCB63 – Analysis and Design of Data Structures

Comparison Based Algorithms

Thinking about decision trees proved that our lower bound for comparison based algorithms is Omega(n log n) to sort n keys.
We can use this to find bounds for other things, such as:
Searching a Sorted Sequence:
Input: $A_1 \ldots A_n$ and k s.t. $k=A_i$ for some I, to find where the I is
We can just do a binary search, etc. but we can draw this as a decision tree.
Any decision tree that represents a problem needs as many leaves as there are possible outcomes
Therefore, it must have height at least $\log_2 n$ → conclusion, any comparison based algorithm for searching a sorted list requires at least omega(log n) time. Thus, binary search is optimal.
This is called the **Information Theoretic Argument for Lower Bounds**
Any decision tree that solves (does k exist in sorted list) must have at least $\log_2(2n+1)$ leaves.
Combining sorted sequences
The number of leaves is Choose(n+m, n) – height is $\log_2$(Choose(2n, n))
Stirling's Approximation – any comparison based algorithm to merge two sorted lists of length n requires >= 2n – 1/2$\log_2$n – 0.826 comparisons
Standard Algorithm for Merging two sorted lists – 2n – 1
Searching an unsorted sequence
Input: $A_1 \ldots A_n$ and k s.t. $k=A_i$ for some I, to find where the I is
ITLB should be at least n leaves and height should be at least $\log_2$n, at least $\log_2$n comparisons in the worst case.

Amortized Analysis

Instead of using worst case for one operation, we evaluate the average time for an operation on any data structure. We introduce amortized analysis.
Sequence Complexity -> $C_{(m)}$ the maximum number of steps that it takes to process m operations starting from some initial state.
$C_{(m, n)}$ more refined, i.e. n of the operations are insertions
Amortized Complexity – Take C_m/m and that's our amortized analysis.
Naïve Upper Bound – K(m) = cost of most expensive operation in the worst case sequence. Therefore A(m) <= K(m), but C(m) <= K(m) so that's not really useful. When does this work? Suppose we have an AVL tree, so we have
K(m) = Theta(log m) so ➔ A(m) = O(log m) ➔ C(m) = O(log m) which makes sense. Sometimes this will be the case.
This will not always be the case – we need to perform some lesser operations before we can get to a large and expensive operation.
Accounting Method (banker)
Credit Scheme – Associate # of credits with each type of operation. Some credits are used to pay for operation, some are stored in data structure.
Generalist stack example:
Push – 2 credits → One credit for the push, another stored in DS
Pop – 0 credits → Each pop is paid for by the stored credits by pushes
Adequate Credit Allocation Scheme: Allocated enough to the operation of each seq to pay for cost of most expensive
Credit Invariant: Statement about how many credits have been stored based on credits stored in the past
e.g. Every element in the stack has a credit with it
Dynamic Tables
Table expansion -- Table when we need more space we create a new table and transfer elements to new table **O(n)**
Table contraction – Table when we need less space, same as above, **O(n)**
Assume Insertion/Deletion not involving table expansion/contraction takes constant time
I/D involving E/C takes Theta(elements) time
Accounting Scheme –
Insert – 3 credits → use 1 credit to insert I, store 2 credits in table, After k insertion, we have 2k credits (expansion = 2x eles)
Deletion – 2 credits, 1 for time to delete we have enough to move n/4 elements into a table of n/2

Disjoint Sets

MAKESET(S) – creates a new set in the collection {x}, singleton – one element only
FIND(x) – returns the representative of the unique set that contains x
UNION(x, y) – replaces x and y with a new set that's the union of x and y → can also be UNION(Find(X), FIND(Y))
N = number of makeset operations (nodes in structure)
M = number of find operations
N-1 = max amount of union operations
Linked List Representation of a Disjoint Set –
Each set is a linked list
Each representative of a set is a pointer to element of each list
Each node has == (first, next, last)

```
Makeset(x) – first(x) = last(x) = x, next(x) = nil
Find(x) – first(x)
Union(x, y) =
```

Brian Chen
chen187
1002297034

```
T = x
While(t != nil) do
        First(t) = y
        T = next(t)
Next(last(y)) = x
Last(y) = last(x)
Return y
```

MS = theta(1)
F = theta(1)
U(x, y) = theta(len(x))
Amortized Analysis: T(n, m) = max cost to process n MS and n-1 U and m F ops.
$O(m + n^2)$
n + m + (n-1) takes no more than time proportional to n
n + m + (n-1)n = n + m + n^2 = n^2 + m as wanted
Weighted Union – Theta(n log n) – if we always append shorter list to longer list!!!
Another option – worse find, better union. For x's head, we make it y. All we need is first(x) = last(y), but now we just need to do two hops to find head(y). We also do not need the next pointer, so we can shake it and get a …
Tree Representation of a Disjoint Set –
Find is recursive – basically find until curr = parent (because root loops the pointer onto itself)
Collection of sets = **Forest**
Each set in collection is an '**Up Tree'** – root points to itself, node points to parent
Rep(set) is now root of the tree
T(n, m) = Omega(nm) – to avoid the worst case, we avoid putting tall trees under short trees ➔ Omega(m log n)
Union by weight – make smaller trees subtrees of larger trees (more nodes). We need to add weight information
In the forest of any sequence of MS/U/F the UbW rule, the weight of any tree wt(T) >= $2^{ht(T)}$
Path Compression – After having gone through a path, we shorten the path (i.e. a -> b -> c, we can just make a-> c and b -> c)
Using Union by Weight and Path Compression we can process any sequence of N, M, N-1 in **O(m+n log* n)** (practically constant)
Log* n is at most 5 usually. **Amortized cost O(log* n)**

Graphs

Graph G = (V, E) (vertices, edges)
Degree(u) = number of nodes adjacent
Strongly connected (digraph) = u-> v for every u, v
Connected (undigraph) = u->v for every u,v
Handshake lemma = degree(u) = |E| if G is directed, 2|E| if G is undirected
Adjacency Matrix – Need $O(n^2)$ space to have this matrix
Linked List Representation – Each node has a node of its neighbor nodes – need theta(n + SUM(degree(u))) space = theta(n+m)
If the graph is sparse, adjacency list is better.
Determine if (u, v) belongs to E (if they are connecte) theta(1) in adj matrix, theta(deg(u)) in adj list
Find all neighbors of node u theta(n) in adj matrix, theta(deg(u)) in adj list (deg is number of adjacent, n is total nodes)
Graph Searches
Breadth First Search(BFS) – Starting at node s, we store all neighbors in our queue.

```
BFS(G(graph), s(Searchee):
        For each node u!=s do d[u] := inf
        Q = empty queue
        ENQUEUE(Q, s)
        D(s) := 0; parent(s) = NIL
        While(Q != empty) do
                U:= dequeuer(Q)
                For each v in adjacency[u] do
                        If d(v) = inf then
                                ENQUEUE(Q, v); d(v) = d(u)+1 (can be + weight); par(v):=
```

Every node has a discovery and finish time.
Running Time for BFS:
n + sum(degrees of all nodes in G) = theta(n + m) = **Linear Time Algorithm for a graph**
Lemma 1) if d(v) = I then there is a path from s to v of length I
Lemma 2) for any node v that is enqueued, if v' is enqueued before v then d(v') <= d(v)
BFS theorem: if shortest path s->v has length I then d(v) = i
Depth First Search(BFS) –

```
DFS(G, s):
For each node u do:
        D(u):= f(u) := 0; par(u) = NIL
Time = 0
For each node u do:
        If d(u) = 0 then DFS-V(u)
DFS-V(u):
        Time = time+1
        D(u) = time    //discovery of u
        For each v E adj[u] do //explore (u, v)
                If d[v] == 0 then
```

Brian Chen
chen187
1002297034

```
                    Par(v) = u
                    DFS-V(v)
        Time = time+1
        F(u) = time // finished with u
```

Running time is **theta(m+n)** just like BFS.
We define a subgraph of G, G' = (G, E') such that E' is all the parent pointers (u,v) such that u is parent of v
Lemma 1) There exists x->y of length >=1 in G' ⇔ DFS-V(y) is called during DFS-V(x)
(x,y) classification when (x, y) is explored:
Forest ⇔ d(y) = 0
Back ⇔ d(x) >= d(y) != 0 and f(y) = 0
Forward ⇔ d(x) < d(y)
Cross ⇔ d(x) > d(y) and f(y) != 0
DFS Applications
Testing digraphs for **cycles** – if we detect a back-edge then theres a cycle! Otherwise, if we conclude DFS and no back-edge, no cycle
Lemma 2) For any path p in G, if p starts at u and when DFS-V(u), d(u) = 0, for all u on p, then DFS-V(v) is executed during DFS-V(u) for all nodes v on p.
Topological Sorts
Topological sort of Directed Acyclic Graph (DAG) G:: listing of all G's nodes such that if u-> v path then u is listed before
G must be acyclic for this toposort to be possible
If G is acyclic then this toposort exists

Minimum Spanning Trees (MST) –

Input: Undirected, connected graphs G = (V, E)
Output: A minimum weight spanning tree of the graph G
Free Tree – Connected, undirected, acyclic graph – MSTs are Free Trees
Kruskal's Algorithm – Greedy algorithm!
Start with trivial partial solution, empty set of edges → extend greedily the partial solution one edge at a time. We keep going until n-1 edges (smallest to form a tree) that's our MST.

Greedy Rule – Pick a minimum weight edge that does **not create a cycle** with our current partial solution
We use priority queues to prioritize the smallest weight edges

```
H:= heap containing (u, v, wt(u,v)) for all edges (u, v) E G
F := nothing
While |F| != |V|-1 do
        (x,y,t) = extractMin(h)
        X' = find(x), y' = find(y)
        If [x->y path using F in edges] then F:= F U {x,y}
                ^ if x' = y' (same rep, in same disjoint set)
```

Using disjoint sets, each set contains all nodes containing all connected nodes
We find what set x, y belongs to (if they belong to same set, then adding it creates a cycle)
Running Time: O(m log n)
Cut Properties
Prim's Algorithm –
Start with a specific edge, lets call it a, choose the smallest edge out of that tree. This is one tree we keep extending vs multiple smaller trees

```
R := {s}; F := nothing
While R != V do
        (u,v) = min wt edge connecting u in R to node v not in R
        R:= R u {v}
        F: F u {(u,v)}
Return F                    -- we can use a 'near' array to do the first statement in while
```

Running Time: O(n^2) (m (degree) could be as big as n^2 – could be if dense, so Kruskals could be worse)
Prims – Better for dense graphs
Kruskals – Better for sparse graphs