3100-Final-Review

Trees

Defination

Definition: a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle.

Sibling: Node B and Node C are siblings if they have the same parent.

Leaf: A node is called leaf if it has mo children.

Depth of node i: is the length of the unique path from the root to ni.

Height of node i: is the length of the longest path.

The height of a tree is equal to the height of the root, the depth of a tree is equal the depth of the deeppst leaf.

Tree can have one only parent, tree have no cycles.

```
// Implementation of a tree
class TreeNode{
  Object element;
  TreeNode firstchild; // arrow that points downward
  TreeNode nextsibling; // arrow that goes left from right.
}
```

Ordered Tree: There is a linear ordering defined for the children of each node. (can define the children of a node as the first, second, third).

Binary Tree: in which no node can have no more than two children. left node and right node could possibly be empty.

Strictly binary tree: Every non-leaf node in the tree has non-empty left and right sub-trees.

complete binary tree: a complete binary tree of depth d is strictly binary tree with all leaf nodes at level d.

Perfect binary tree: A Perfect Binary Tree(PBT) is a tree with all leaf nodes at the same depth. All internal nodes have degree 2.

BST Traversing Strategy

- Preoder: (depth first) `Visist Node --> Traverse the left --> Traverse the right`
- Inorder: `Traverse the left --> visit the node --> traverse the right`
- Postorder: `Traverse the left --> Traverse the right --> visit the node`

Using two sequence(preorder and postorder) to construct a tree, but there may not be uniquely defined by its preorder and postorder sequences.

Binary Search Tree: for every node, the key value in its left subtree are smaller, and the key value in its right subtree are lagrer than the key value of T.

- Support many dynamic set operations such as findMin, findMax, insert, predecessor, and so on.
- Running time of basic operations on binary
- seach trees: on average, $O_{(logn)}$ (The expected height of the tree is logn), worst case $O_{(n)}$(The tree is a linear chain).

Tree operations:

```
# Search:  find(x,k), The running time is O(h), where h represent the height of the tree
if x = NULL or find key[x]
  then return x
if k < key[x]
  then return find(left[x], k)
  else return find (right[x],k)

# Find Min: keep finding left from root, until reach the end
while left[x] != NULL
  do x <- left[x]
return x

# Find the Max: keep finding right from root, until reach the end
while right[x] != NULL
  do x <- right[x]
return x

# Successor
# def: the smallest key greater than x
  case 1: right x is non empty
    successor = the minium in right x
  case 2: right x us empty
    go up the tree until the current node is left child, successor = parent of current node
    if can not go further(and reach the root) x is the largest.

if right[x] != NULL
  return findMin(right[x])
y <- p[x]
while y != NULL and x = right[y]
  do x <- y
    y <- p[y]
  return y
```

```
# Predecessor
# def: the biggest key less than x
  case 1: left x is non empty
    predecessor = the maximum in left x
  case 2: left x is empty
    go up the tree until the current node is a right child, predecessor = parent of current
node
    if can not go further(and reach the root), x is the smallest

# Insert: Find the proper position and insert the value
y <- NULL
x <- root[T]
while x != NULL
  do y <- x
    if key[z] < key[x]
      then x <- left[x]
      else x <- right[x]

-----if Tree is empty
p[z] <- y
if y == NULL
  then root[T] <- z
  else if key[z] < key[y]
    then left[y] <- z
    else right[y] <- z

# Deletion
case 1: no children or one chilren. just delete
case 2: two chilren. find the successor and replace it.
```

Red-Black Tree

Definition:

- Binary search tree with an additional attribute for its node, (color, which can be red or black.)
- Restrict: the way nodes can be colored on any path from root to leaf
- Ensures that no path is more than twice as long as any other path.

Balanced binary search trees guarantee an O(logn) running time.

Propertites:

- Every node is either red or black
- the root is black
- every leaf(NIL) is black.
- if a node is red, then both its children are black, no two consecutive red nodes on a simple path form the root to a leaf.
- For each node, all paths from that node to descendant leaves contain the same number of black nodes.

In this waty to keep the tree in balanced .

For NIL[T]: use a sentinel NIL[T] to represent all the NIL nodes at the leafs, NIL[T] has the same fields as an ordianry node, color is black, and the other is not important.

Height of a node x: the number of edges in the **longest** path from the current node to a leaf.

Black-height of a node x: the number of black nodes(including NIL) on the path from x to a leaf, not counting x.

A RBT with n internal nodes has height at most $2log_{n+1}$.

The search opeartion is still O(h), which is O(nlogn) runtime.

The insert and delete will keep O(nlogn), but this time we have to maintain the tree order is still a red-black tree.

That's why we need rotation:

- together with some node re-coloring
- change some pointer of structure
- maintain the RBT property

Time complexity: `O(1)`. since a constant number of pointers are modified.

🔷**Insertion:**

- insert node z into the tree as for an ordinary binary search tree
- color the node red.
- Restore the RBT and do **Fix-Up** (modify the tree by recoloring nodes and performing rotation to preserve the RBT property).

Inserting the new element into the tree is O(logn), Fix-Up: The while loop repeats only if case 1 is executed, takes O(logn), The total running time is `O(logn)`

🔶**Deletion:**

- delete like ordinary RBT
- coloring and Fix-up

should preserve all the properties of RBT, and do the fix-up.

AVL Tree

Definition: A balanced BST where the height of the two subtrees of any node differs by at most one.

A BST is an AVL tree if for any node X: $BF(x) \in (-1, 0, 1)$.

Graph

Definition

A graph $G = (V, E)$, consist of a set of vertices, V, and a set of edges, E. each edge (arc) is a pair (v, w), where v,w $\in$ V. An edge may have a weight**(cost).**

If a pair is ordered, then the graph is directed-graph.

Vertex w is adjacent to v if and only if (v,w) $\in$ E.

Directed graph:

- The **in-degree** of a vertex v is the number of edges (u,v) entering v.
- The **out-degree** of a vertex v is the number of edges (v,u) leaving v

Undirected graph:

The degree of a vertex v is the number of edges connecting v. edges have no specific direction, edges always "two way".

Degree of a vertex: number of edges containing that vertex. (the number of adjacent vertices).

Self-edges: a loop edge is of the form (u,u)

The use/algorithm usually dictates if a graph has: no self edges, some self edges, all self edges.

Connectedness: a graph does not have to be connected.

if $(u,v) \in E$, then v is a neigjbor of u(ie, v is adjacent to u), but order matters in directed graph.

Weighted Graph: in a weighted graph, each graph has a weight or cost. Typically is numeric, some graph allows negative weights, but many do not.

Path and cycles: denote pathe exist from V0 to Vn if there is a list of vertices[V0,V1, ... , Vn] such that (Vi, Vi+1) belongs to E for all i belongs [0,n]. cycle is a path that begins and ends at the same node.

Path length: Number of edges in a path

Path cost: sum of the weights of each edge.

A *simple path* repeats no vertices(except the first may be the last)

A *cycle* is a path that ends where it begins

A **simple cycle** is a cycle and a simple path.

Undirected graph connectivity: An undircted graph is conneted if for all pairs of vertices u != v, **there exits a path from u to v.**

Is **full or complete** if for all pairs of vertices u!=v thre exist an edge from u to v.

Directed grah connectivity:

- Strongly connected: if there is a path from every vertex to every other vertex.
- Weekly connected: if there is a path from every vertex to every other vertex ignoring direction of edges.
- complete or fullt connected, if for all pairs of vertices u != v, there exits an edge from u to v.

Tree as graphs

Tree as graphs: tree is a graph that: undirected, acyclic, connected. **ALL the trees are graphs.**

Rooted trees: identify a unique root, edges are directed: parent --> children.

Directed acyclic graphs(DAGs): A DAG is a directed graph with **no directed cycles.**

- Every rooted directed tree is a DAG
- but not every DAG is a rooted directed tree.
- Every DAG is directed graph
- but no every direted graph is a DAG

Representation of graph: mainly need to answer: " what is the lowest cost path from x to y."

If we need to choose a data structure to represent graph, it may depends on the properties (dense, sparse..)

Adjacency Matrix and Adjacency List

Adjacency Matrix: A |V| |V| matrix of booleans(or 0 vs. 1)That means M [u] [v] == true means there is an edge from u to v.

Running time property:

- Get in/out edges (O|V|)
- decide if some edge exists (O1)
- insert/delete an edge (O1)

Space requirement: O(V*V)

Useful when graphs are dense

How will adjacency matrix vary for an undireted graph, will be symmetric about diagnoal axis. so If we save space by only using the half of the matrix, there will be a problem that can not get all neighbors.(search linear)

for the representation for weighted graphs, store the number instead of boolean, if we need some value for 'not an edge', the 0,-1,Inf may be used, or need some other filed.

Adjacency List:

Assign each node a number from 0 to |v| -1.

Then it is an array of lenghth |V| in which each entry stores a list of all adjacent vertices (linked list)

Running time:

- get out-edges: O(d)
- get in-edges: O(E) **(could keep a second adjacency list for that)**
- decide if some edge exist: O(d)
- Insert and delete: O(1)

Space requirement: O(|V|+|E|)

Useful for **sparse** matrix

Which one is better: graphs are often sparse, Adjacency lists should generally be the default choice, which **slower performance compensated by greater space savings.**

Topological Sort

An ordering of all vertices in a direted acyclic graph, such that if there is a path from vi to vj, then vj appears after vi in the ordering.

If there is no path between vi and vj, then any order between them is fine.

It is not possible if there is a cycle in the graph

A DAG has at least one topological ordering.

A simple algorithm:

- Compute the indegree of all vertices from the adjacency information of the graph
- Find any vertex with no incoming edges
- Print this vertex, and remove it, and its edges
- Apply this strategy to the rest of the graph.

An improved algorithm:

- Keep all the unasigned vertices of indegree 0 in a queue.
- while queue is not empty
- remove a vertex in the queue
- Decrement the indegree of all adjacent vertices
- If the indegree of an adjacent vertex becomes 0, enque the vertex

Runing time: O(|E| +|V|).

Shortest Path problem

Variants of shortest path:

- single-source shortest path: find a shortest path from a given source vertex s to each vertetx v belongs to V.
- single-destination shortest path: find shortest path to a given destination vertex t from each vertex v
- single-pair shortest path: find a shortest path from u to v for given vertices u and v
- All-pairs shortest path: find a shortest path from u to v for every pair of vertices u and v.

Shortest path can NOT contain cycles.

optimal substructure Theorem:

shortest path from vi to vk is p1, shortest path from vk to vj is p2, then the shortest path from vi to vj is the sum of theri two.

For unweighted shortest paths in single-source graphs algorithm:

- Mark the starting vertex, s
- Find and mark all unmarked vertices adjacent to s (vertices adjacent to s)
- Find and mark all unmarked vertices adjacent to the marked vertices
- Repeat step3 until no more vertices can be marked.

The strategy is Breadth-First Search

For each vertex, it is required to keep track of:

- whether the adjacent vertex has been marked
- its distance from s(dv)
- previous vertex of the path form s(pv)

can employ queue structure to implement.

Dijkstra's Algorithm:

- A greedy algorithm, solving a problem by stages by doing what appears to be the best thing at each stage.
- select a vertex v, which has the smallest dv among all the unknown vertices, and declare that the shortest path from s to v is knonw
- For each adjacent vertex, w, update dw = dv + Cv,w. if this new value for dw is an improvement. **(Relaxation part)**

Bellman-Ford Algorithm:

Allows negative edge weights can detect negative cycles.

Return True if no negative-wight cycles are reachable

Return false otherwise.

Key-Point: after |V| -1 iterations, d values will not be updated or can not be lower any more, store the measure of the shortest path.

Spanning Tree:

defination: given a connected graph G(v,E), a spanning tree T(V,E) is:

- Subgraph of G
- spans the graph
- Forms a tree(no cycle)
- E has |V| - 1 edges.

Minimum Spanning tree: edges are weighted, aim to find the minimum cost spanning treee.

可以看到一个包含3个顶点的完全图可以产生3颗生成树。对于包含n个顶点的无向完全图最多包含 n^{n-2} 颗生成树。比如上图中包含3个顶点的无向完全图，生成树的个数为： $3^{3-2}=3$.

Two algorithm:

Prime: build up the tree incrementally. (based on vertex)

pick lower cost edge connected to known(incomplete) spanning tree that does not create a cycle and expand to include it in the tree.

Kruskal: build forest that finish as a tree (based on edge)

pick lowest cost edge not yet in a tree that does not create a cycle, then expand the set of included edges to include it. (somewhere in the forest)

Detecting a cycle:

if the edge to be added (u,v) is such that vertices u and v belongs to the same tree, then by adding (u,v) would form a cycle.

therefore to check, `Find(u)` and `Find(v)`, if they are the same discard (u,v).

if thery are different `Union(Find(u),Find(v))`.

Properties of trees in K's algorithm:

- Vertices in different trees are disjoint (True at initialization and Union won't modify the fact for remainning trees)\
- Trees form equivalent classses under the relation "is conneted to"

Priority Queue && Heap

Priority queue: A queue maintain the priority, FIFO.

Three implementations:

- Simple linked list (with insert at O1 and delete at On)
- Sorted linked list(with insert at On and delete at O1)
- Sorted array(with Insert On and delete On)
- Binary heap(With **Ologn for insert and delete**)

Binary Heap:

A heap is a binary tree that completely filled. except at the bottom level, which is filled from left to right.

A complete binary tree of height h has between 2^h and $2^{h+1}-1$ Nodes.

Heap order property:

The value at any node should be smaller than(or equal to) all of its descendants (gurantee that the node with the minimum value at the root).

Binary Search Tree and Heap is **DIFFERENT in node ordering.**

Operations:

Insert: to insert an element X, Create a hole in the next avaliable location, If X can be placed in the hole without vilating heap order, insertion is complete. Otherwise slide the element that is in the hole's parent node into the hole, bubling the hole up towards the root. continue the process until X can be placed in the hole.

Deletion: The element at the root is to be remved, and a hole is created, fill the root with last node X, Percolate X down until the heap order is satisfied.

Both Wost case are in the O(logn).

Build up the heap: Time complexity: O(n).

d-Heaps: d-heap is exactly like a binary heap except that all nodes have d children.

Hash Table

Set: A collection that does not allow duplicates(dont have indices or any order)

Basic operations: insert, remove, search.

But different to check if a word is in the dictionary.

Hash Table: Table maintains b different "buckets", also have hash function maps elements to value in 0 to b -1. Use has to determine which bucket an element belongs in and only search/modify this bucket.

Look-up becoms constant time. Easy to search.

But Lose all ordering information like get min, get max, remove min so on.

Hash collision: the event that two hash table elements map into the same slot in the array.

collision solution: means for fixing collisions in a hash table.

Hash function for strings:

one possible way is treat first character as an int and hash on that.

Another possible way is treat each character as an int, sum them and hash on that.

A third option is polynomical accmuclation. Perform a weighted siumm of the letters, and hash on that.

Coming a great hash function is hard

Chainining: all keys that map to the same hash value are kept in a linked list.

Load factor: ratio of elements to capacity.

Search with chaning:

unsuccessful: λ (the average lenghth of a list at hash)

successful: $1 + \frac{\lambda}{2}$ (one node, plus half the average length of a list, not including the item)

Implementing set with hash table: each set entry adds an element to the table(based on the hash function)

Run time will be: insert, remove, search (O1)

MinHash & Locality Sensitive Hashing

Goal: Find near-neighbors in high dimension space.

Jaccard distance/similarity:

The Jaccard similarity of two sets is the size of their intersection divided by the size of their union.

Jaccard distance is 1 - Jaccard similarity.

If we want to check for similarity for billons of documents, keeping full text in the memory is not an option, hence we need to **find a shorter representation.** How do we pairwise comparision of billions of documents?

1. Shingling: shingling convert documents to sets

Difinition: A K-shingle (k-gram) for a document is a sequence of k tokens that appear in the document. (token can be character, words).

Then we can represent the Document D as a set of its k-shingles, equivalently, each document is a 0/1 vector in the space of k-shingles.

Warning: pick the K large enough, or most document will have most shigles. (such as k = 10).

2. **Minhash, LSH:** we need to find near-duplicate documents among N = 1 million documents, which cost a lot of time, so we need to reduce it.

Signatures: short integer vectors that represent the sets, and reflect their similarity.

Minihashing: convert large sets to short signatures, while preserve similarity.

Using bit vectors to formalized the problem, encode the sets using 0/1(bit or boolean), intersection as bitwise AND and set union as bitwise OR. And combine this vectore to a matrix, then we can transfer the problem to find the similar colomns.

Then, next goal is **similarity of colomns == similarity of signatures.**

3. Hashing colomns:

Key idea is that: hash each column C to a small signature h(c), suhc that, h(c) is small enough that the signature fits in RAM, and sim(c1,c2) is the same as the similarity of signatures h(c1) and h(c2). --> Find the Hash function.

The function depends on the similarity metric, not all simialrity metric have a suitable hash function.

The similarity of signatures is the fraction of the hash functions in which they agree. With multiple signatures we get a good approximation.

A more practical way: Instead of permuting the rows we will apply a hash function that maps the rows to a new (possible larger space),

4 .Locallity Sensitive Hashing:

Goal: Find the documents with Jaccard similarity at least s.

LSH general idea: use a function `f(x,y)` that tells whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated.

For mini-hash matrices: hash colums of signature matrix M to many buckets, each pair of documents that hashes into the same bucket is a candidate pair.

Sorting & complexity

Sorting classification: **In memory sorting** and **Externel sorting.**

In memory sorting: Comparision sorting(nlogn), specialized sorting.

external sorting: access the disk. (merge sort)

Sorting an array:

Bubble sort.

Insertion sort. (Keeping expanding the sorted portion by one)

Quick sort:

Pick pivot: Use the first element as pivot. (situation will be worse when array is reverse order)

if Choose the pivot randomly, generally is safe but also have overhead in generating.

Partition:

In-place partiton: if use additional array (not in-place), straightforward to code, but not efficient.

Merge sort:

divide and conquer algorithm. recursively divide each part in half, continuing until a part contains only one element(one is sorted), then combine them into one sorted list.