CSE 250: Midterm Review

Lecture 29

Nov 8, 2023

Reminders

- Midterm 2: Friday
  - Review today
  - Example midterms on class website

Sorting Algorithms

Algorithm	**Runtime**
BubbleSort	$O(N^2)$
MergeSort	Unqualified $O(\log N)$
QuickSort	Expected $O(\log N)$
HeapSort	Unqualified $O(\log N)$

Bound Guarantees

- $f(N)$ is a [Unqualified] Worst-Case Bound $(T(N) \in O(f(N)))$
  The algorithm **always** runs in at most $c \cdot f(N)$ steps.

- $f(N)$ is an Amortized Worst-Case Bound
  N **invocations** of the algorithm **always** run in at most $N \cdot c \cdot f(N)$ steps.

- $f(N)$ is an Expected Worst-Case Bound $(E[T(N)] \in O(f(N)))$
  The algorithm is **statistically likely** to run in at most $c \cdot f(N)$ steps.

Back to Sequence ADTs

- **Sequence**
  - `get(i)`, `set(i, v)`
- **List**
  - ... and `add(v)`, `add(i, v)`, `remove(i)`,
- **Stack**
  - `push(v)`, `pop()`, `peek()`
- **Queue**
  - `add(v)`, `remove()`, `peek()`

The Stack ADT

A stack of objects on top of one another.

- **Push**
  Put a new object on top of the stack.
- **Pop**
  Remove the object from the top of the stack.
- **Top**
  Peek at what's on top of the stack.

The Queue ADT

Outside of the US, "queueing" is lining up.

- **Enqueue** (`add(item)` or `offer(item)`)
  Put a new object at the end of the queue.
- **Dequeue** (`remove()` or `poll()`)
  Remove the object from the front of the queue.
- **Peek** (`element()` or `peek()`)
  Peek at what's at the front of the queue.

Queues vs Stacks

- **Queue**
  First in, First out (FIFO)
- **Stack**
  Last in, First out (LIFO, FILO)

Queues vs Stacks (Implementation)

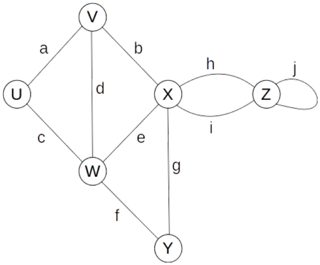
ADT	**Stack**		**Queue**	
using...	`Doub. L. List`	`Array`	`Doub. L. List`	`Array`
`add`	$O(N)$	Amortized $O(N)$	$O(N)$	Amortized $O(N)$
`remove`	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Graphs

A **graph** is a pair (V, E), where

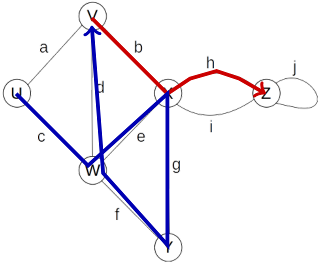
- V is a set of **vertices** (sometimes nodes)
- E is a set of vertex pairs called **edges**
- Edges and vertices may also store data (**labels**)

Graph Terminology



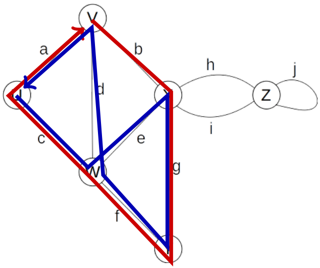
- **Endpoints of an edge**
  U, V are the endpoints of a.
- **Edges incident on a vertex**
  a, b, d are incident on V.
- **Adjacent Vertices**
  U, V are adjacent.
- **Degree of a vertex (# of incident edges)**
  X has degree 5.
- **Parallel Edges** (same endpoints)
  h, i are parallel.
- **Self-loop** (same vertex is start and end)
  j is a self-loop.
- **Simple Graph**
  A graph with no parallel edges or self-loops.

Paths



- **Path**
  A sequence of alternating vertices and edges
  - Begins with a vertex
  - Ends with a vertex
  - Each edge is preceded/followed by its endpoints
- **Simple Path**
  A path that never crosses the same vertex/edge twice
- **Examples**
  V, b, X, h, Z is a simple path.
  $U, c, W, e, X, g, Y, f, W, d, V$ is a path that is not simple.

Cycles



- **Cycle**
  A path that starts and ends on the same vertex.
  - Must contain at least one edge
- **Simple Cycle**
  A cycle where all of the edges and vertices are distinct (except the start/end vertex).
- **Examples**
  $V, b, X, g, Y, f, W, c, U, a, V$ is a simple cycle.
  $U, c, W, e, X, g, Y, f, W, d, V, a, U$ is a cycle that is not simple.

Notation

- N: The number of vertices
- M: The number of edges
- $\texttt{deg}(v)$: The degree of a vertex

Handshake Theorem

$$\sum_{v \in V} \texttt{deg}(v) = 2M$$

Proof (sketch): Each edge adds 1 to the degree of 2 vertices.

Edge Limit

In a directed graph with no self-loops and no parallel edges:

$$M \leq N \cdot (N - 1)$$

Proof (sketch):

- Each pair is connected at most once (no parallel edges)
- N possible start vertices
- $(N - 1)$ possible end vertices (no self-loops)
- $N \cdot (N - 1)$ distinct combinations possible

The Directed Graph ADT

Interfaces

- `Graph<V, E>`
  - `V`: The vertex *label* type.
  - `E`: The edge *label* type.
- `Vertex<V, E>`
  - ... represents a single element (like a LinkedListNode)
  - ... stores a single value of type `V`
- `Edge<V, E>`
  - ... represents an edge (a pair of vertices)
  - ... stores a single value of type `E`

Graph Data Structures

What do we need to store for a graph ((V, E))?

- A collection of vertices
- A collection of edges

Edge List

```java
class EdgeList<V, E> implements Graph<V, E>
{
  List<Vertex> vertices = new ArrayList<Vertex>();
  List<Edge>   edges    = new ArrayList<Edge>();

  /*...*/
}
```

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`: $O(M)$

Space Used: $O(N + M)$
(constant space per vertex, edge)

Improving on the Edge List

How can we avoid searching every edge in the edge list to find the incident edges?

Idea: Store each edges in/out edge list.

Adjacency List

```java
public class Vertex<V, E>
{
  Node<Vertex> node = null;
  List<Edge> inEdges = new BetterLinkedList<Edge>();
  List<Edge> outEdges = new BetterLinkedList<Edge>();
  /*...*/
}
```

Adjacency List Summary

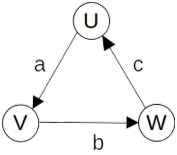
- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\texttt{deg}(v))$
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`: $O(\texttt{deg}(v))$

Space Used: $O(N + M)$
(constant space per vertex, edge)

The Adjacency Matrix Data Structure

Destination

Origin	U	V	W
U	-	a	-
V	-	-	b
W	c	-	-



Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Space Used: $O(N^2)$

A few more definitions

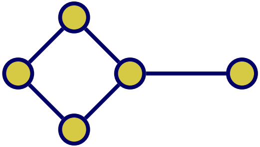
A graph is **connected** if. . .

- . . . there is a path between every pair of vertices.

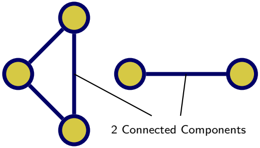
A **connected component** of G is a *maximal*, *connected* subgraph of G

- "*maximal*" means that adding any other vertices from G would break the connected property.
- Any subset of G's edges that makes the subgraph connected is fine.

Connected Graph



Disconnected Graph



Depth First Search (DFS)

Primary Goals

- Visit every vertex in graph $G = (V, E)$.
- Construct a spanning tree for every connected component.
  - **Side Effect**: Compute connected components.
  - **Side Effect**: Compute a path between all connected vertices.
  - **Side Effect**: Determine if the graph is connected.
  - **Side Effect**: Identify any cycles (if they exist).
- Complete in time $O(N + M)$.

Depth First Search (DFS)

`DFS(G)`

Input

- Graph $G = (V, E)$

Output

- Label every edge as a:
  - Spanning Edge: Part of the spanning tree
  - Back Edge: Part of a cycle

Depth First Search (DFS)

`DFSOne(G, v)`

Input

- Graph $G = (V, E)$
- Start vertex $v \in V$

Output

- A spanning tree, rooted at v, to every node in v's connected component.

Depth First Search (DFS)

DFSOne

1. Initialize Todo **Stack** with start vertex v (no edge)
2. Retrieve next todo vertex (or return if none left).
3. If the vertex is already visited[1], return to step 2.
4. Otherwise, mark this vertex as visited.
5. Mark the edge listed in the todo item as a spanning edge.
6. Add todo items for every unvisited, adjacent vertex (via the edge to the current vertex).
7. Return to step 2.

[1] It won't be for DFS or BFS, but bear with me...

Breadth First Search (BFS)

BFSOne

1. Initialize Todo **Queue** with start vertex v (no edge)
2. Retrieve next todo vertex (or return if none left).
3. If the vertex is already visited[2], return to step 2.
4. Otherwise, mark this vertex as visited.
5. Mark the edge listed in the todo item as a spanning edge.
6. Add todo items for every unvisited, adjacent vertex (via the edge to the current vertex).
7. Return to step 2.

[2] It won't be for DFS or BFS, but bear with me...

Dijkstra's Algorithm

Dijkstra One

1. Initialize Todo **Priority Queue** with start vertex *v* (no edge)
2. Retrieve next todo vertex (or return if none left).
3. If the vertex is already visited, return to step 2.
4. Otherwise, mark this vertex as visited.
5. Mark the edge listed in the todo item as a spanning edge.
6. Add todo items for every unvisited, adjacent vertex (via the edge to the current vertex).
7. Return to step 2.

Graph Traversal

	DFS	**BFS**	**Dijkstra's Algo**
Runtime	$O(N + M)$	$O(N + M)$	$O(N + M\log(M))$[3]
Visit Order	Last Visited	Closest by Edge Count	Closest by Total Edge Weight
Spanning Tree	Long paths	Fewest Vertices to Root	Shortest Edge Weight to Root

[3] With Heap as Priority Queue

New ADT: Priority Queue

`PriorityQueue<E>` (`E` must be `Comparable`)

- `public void add(E e)`: Add e to the queue.
- `public E peek()`: Return the *least* element added.
- `public E remove()`: Remove and return the *least* element added.

(Partial) Ordering Properties

A partial ordering must be...

- **Reflexive** $x \leq x$
- **Antisymmetric** **if** $x \leq y$ **and** $y \leq x$ **then** $x = y$
- **Transitive** **if** $x \leq y$ **and** $y \leq z$ **then** $x \leq z$

(Total) Ordering Properties

A total ordering must be...

- **Reflexive** — $x \leq x$
- **Antisymmetric** — **if** $x \leq y$ **and** $y \leq x$ **then** $x = y$
- **Transitive** — **if** $x \leq y$ **and** $y \leq z$ **then** $x \leq z$
- **Complete** — **either** $x \leq y$ **or** $y \leq x$ **for any** $x, y \in A$

Priority Queues

There are two mentalities...

- **Lazy**: Keep everything a mess.
- **Proactive**: Keep everything organized.
- **Balanced**: Keep everything a little sorted.

Lazy Priority Queue

Base Data Structure: Linked List

- `public void add(T v)` $O(1)$
  Append `v` to the end of the linked list.
- `public T remove()` $O(N)$
  Traverse the list to find the least value and remove it.

Proactive Priority Queue

Base Data Structure: Linked List

- `public void add(T v)` $O(N)$
  Traverse the list to insert `v` in sorted order.
- `public T remove()` $O(1)$
  Remove the head of the list.

Binary Min-Heaps

- **Directed** A directed edge in the tree means $\leq$
- **Binary** (max 2 children, easy to reason about)
- **Complete** (every 'level' except last is full)
  - For consistency, keep all nodes in the last level to the left.

This is a **Min-Heap**

Priority Queues

Operation	Lazy	Proactive	Heap
`add`	$O(1)$	$O(N)$	$O(\log(N))$
`remove`	$O(N)$	$O(1)$	$O(\log(N))$
`peek`	$O(N)$	$O(1)$	$O(1)$

Trees

- **Child**
  An adjacent node connected by an out-edge
- **Leaf**
  A node with no children
- **Depth** of a node
  The number of edges from the root to the node
- **Depth** of a tree
  The maximum depth of any node in the tree
- **Level** of a node
  The depth + 1

Tree Traversals

- Pre-order (top-down)
  - visit **root**, visit **left** subtree, visit **right** subtree
- In-order
  - visit **left** subtree, visit **root**, visit **right** subtree
- Post-order (bottom-up)
  - visit **left** subtree, visit **right** subtree, visit **root**

Binary Search Trees

- **Binary** Tree
  - Each element has (at most) 2 children.
- **Binary Search Tree** Constraint
  - Each node has a value.
  - Each node's value is greater than its left descendants
  - Each node's value is lesser than (or equal to) its right descendants
- **Set** Constraint [optional]
  - Each node's value is unique.

Binary Search Trees

Operation	**Runtime**
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

Balanced Search Trees

- General BST: $d = O(N)$
- **Balanced** BST: $d = O(\log(N))$
  - Complete Tree
  - AVL Tree Property
  - Red-Black Colorability

AVL Trees

- An AVL Tree (Adelson-Velsky and Landis) is a BST where every node is "depth balanced"
  - $|\mathbf{height}(\mathit{left}) - \mathbf{height}(\mathit{right})| \leq 1$

- $\mathbf{balance}(v) = \mathbf{height}(\mathit{left}) - \mathbf{height}(\mathit{right})$

  Maintain $\mathbf{balance}(v) \in \{ -1, 0, 1 \}$
  - $\mathbf{balance}(b) = 0 \rightarrow$ "v is balanced"
  - $\mathbf{balance}(b) = -1 \rightarrow$ "v is left-heavy"
  - $\mathbf{balance}(b) = 1 \rightarrow$ "v is right-heavy"

- $\mathbf{balance}(v) \in \{ -1, 0, 1 \}$ is the AVL tree property

AVL Trees

If $\mathbf{balance}(v) = \mathbf{height}(\mathit{left}) - \mathbf{height}(\mathit{right})$

Then $N > \mathbf{minNodes}(d) = \Omega(1.5^d)$

So $d \in O(\log(N))$

AVL Trees

If the tree starts off balanced:

- The tree can be re-balanced after an insertion in $\log(N)$ time.
- The tree can be re-balanced after a removal in $\log(N)$ time.

Red-Black Trees

A BST is Red-Black Colorable if...

- Every node can be assigned a color, either **Red** or **Black**.
- The root is **Black**.
- The parent of every **Red** node is **Black**.
- The number of **Black** nodes on every path from a null-leaf to the root is the same (the **Black**-depth).

Red-Black Trees

If a BST is red-black colorable...

Then the distance from the root to the shallowest null-leaf is at least half the distance from the root to the deepest null-leaf.

Then The upper "half" of the tree is *complete*.

Then $N > \mathbf{minNodes}(d) = \Omega(2^d)$ **and** $d \in O(\log(N))$

BST Overview

	General BST	**AVL Tree**	**R-B Tree**
`find`	$O(N)$	$O(\log(N))$	$O(\log(N))$
`insert`	$O(N)$	$O(\log(N))$	$O(\log(N))$
`remove`	$O(N)$	$O(\log(N))$	$O(\log(N))$

Note 1: R-B Trees are like AVL Trees, but with a better constant.