CS 5800 - Final - Review material

Old stuff

**Big-O notation** Though you won’t be quizzed directly on Big-O notation, you should be able to apply it to analyzing algorithms, e.g. ones that you produce a problem solutions.

**Recurrence relations** As with Big-O.

**Binary search** Never forget this.

**Merge sort, Partition in quicksort, Insertion sort** You really should be able to tell me about these ten years from now even if you don’t remember all the details.

**Priority queues - binary heaps** You really should be able to tell me that you remember having heard about these ten years.

**Greedy algorithms, Dynamic programming** Methods that you are likely to use throughout your careers.

---

New stuff

You should know these algorithms and how to analyze their running times.

- **Tables - Amortized algorithms**
- **Binary search trees**
- **Decomposition of Graphs**
  - Adjacency list and adjacency matrix representation
  - Depth-first search in undirected graphs
  - Depth-first search in directed graphs
    - Topological sort
  - Strongly connected components
- **Paths in Graphs**
  - Distances
  - Breadth-first search
  - Lengths on edges (weighted graphs)
  - Dijkstra's algorithm
  - NO – Shortest paths in the presence of negative edges
– Shortest paths in dags.

• Minimum spanning trees
  – Kruskal
  – Data structure for disjoint sets: Union/Find
  – Prim

• Flows in Networks
  – Flows and cuts

Graphs
A undirected graph $G = (V, E)$ where each edge $e \in E$ is 2-way and represented by a set $\{u, v\}$ with $u, v \in V$. We also write $(u, v)$ when we are talking about following the edge from $u$ to $v$.
A graph $G = (V, E)$ where each edge $e \in E$ is 1-way and represented by an ordered pair $(u, v)$ with $u, v \in V$. The edge $(u, v)$ goes from $u$ to $v$.

A graph can be represented by an $n \times n$ adjacency matrix where $n = |V|$. The $(i, j)$th entry is

$$a_{ij} = \begin{cases} 
1 & \text{if there is an edge from } v_i \text{ to } v_j \\
0 & \text{otherwise.}
\end{cases}$$

or an adjacency list that consists of $|V|$ linked lists, one per vertex. The linked list for vertex $u$ holds the names of vertices to which $u$ has an outgoing edge, i.e. vertices $v$ for which $(u, v) \in E$.

Depth-first search
You should understand procedures explore($G, v$) on page 84 and dfs($G$) on page 85 using the previsit and postvisit on page 87.

You should be able to perform depth-first search in a directed or undirected graph, label the vertices with pre and post numbers and label the edges as tree or back in an undirected graph, tree, forward, back, or cross in a directed graph.

Tree edges are actually part of the DFS forest.

Forward edges lead from a node to a non-child descendant in the DFS tree.

Back edges lead to an ancestor in the DFS tree.

Cross edges lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already post-visited).

Strongly connected components - the algorithm
1. Run DFS on $G^R$. #
2. Run DFS on $G$ from vertex in 1 with highest post number. Remove those vertices from $G$ (or their component from the dag) and repeat starting with the remaining vertex highest post number from step 1.

Breadth-first search

\[
\text{BFS}(G, s) \\
\text{for all } u \in V \\
\quad \text{dist}(u) = \infty \\
\text{dist}(u) = 0 \\
Q = [s] \text{ (queue containing just } s) \\
\text{while } Q \text{ is not empty} \\
\quad u = \text{ eject}(Q) \\
\quad \text{ for all edges } (u, v) \in E \\
\quad\quad \text{ if dist}(v) = \infty \\
\quad\quad\quad \text{ inject}(Q, v) \\
\quad\quad\quad \text{ dist}(v) = \text{ dist}(u) + 1
\]

Shortest Path

\[
\text{DIJKSTRA}(G, s) \\
\text{ Initialize dist}(s) = 0, \text{ other dist}(\cdot) \text{ values to } \infty, R = \{ \} \text{ (the “known region”) } \\
\text{ while } R \neq V: \\
\quad \text{ Pick the node } v \notin V \text{ with smallest dist}(\cdot) \\
\quad \text{ Add } v \text{ to } R \\
\quad \text{ for all edges } (v, z) \in E: \\
\quad\quad \text{ if dist}(z) > \text{ dist}(v) + l(v, z): \\
\quad\quad\quad \text{ dist}(z) = \text{ dist}(v) + l(v, z)
\]

Minimum spanning trees

Kruskal’s algorithm

Repeatedly add the next lightest edge that doesn’t produce a cycle.

\[
\text{KRUSKAL}(G, w) \\
\text{ Input: A connected undirected graph } G = (V, E) \text{ with edge weights } w \text{ } \\
\text{ Output: A minimum spanning three defined by the edges } X \\
\text{ for all } u \in V: \\
\quad \text{ MAKESET}(u)
\]

\[
X = \{ \} \\
\text{ sort the edges } E \text{ by weight} \\
\text{ for all edges } \{u, v\} \in E, \text{ in increasing order of weight:} \\
\quad \text{ if FIND}(u) \neq \text{ FIND}(v): \\
\quad\quad \text{ add edge } \{u, v\} \text{ to } X \\
\quad\quad \text{ UNION}(u, v)
\]
Data structure for disjoint sets

**Union by rank:**
store a set is as a directed tree. Nodes of the tree are elements of the set, in no particular order, and each has parent pointers that eventually lead up to the root of the tree. This root element is a *representative*, or *name*, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

```plaintext
MAKESET(x)
    π(x) = x
    rank(x) = 0

FIND(x)
    if x ≠ π(x): π(x) = find(π(x))
    return π(x)

UNION(x, y)
    r_x = find(x)
    r_y = find(y)
    if r_x = r_y return
    if rank(r_x) > rank(r_y)
        π(r_y) = r_x
    else
        π(r_x) = r_y
        if rank(r_x) = rank(r_y): rank(r_y) = rank(r_y) + 1
```

The *amortized cost* of a sequence of $n$ union find operations starting from an empty data structure averages $O(1)$ down from $O(\log n)$.

**Prims algorithm** for MST

The intermediate set of edges $X$ always forms a subtree, and $S$ is chosen to be the set of this tree's vertices.

On each iteration, the subtree defined by $X$ grows by one edge, namely, the lightest edge between a vertex in $S$ and a vertex outside $S$.

**Shortest path in dags**

initialize all $\text{dist}(\cdot)$ values to $\infty$

$\text{dist}(s) = 0$

for each $v \in V \setminus \{s\}$, in linearized order:

$\text{dist}(v) = \min_{(u,v) \in E} \{\text{dist}(u) + l(u,v)\}$

**Flow networks**

$G = (V, E)$ directed.

Each edge $(u, v)$ has a *capacity* $c(u, v) \geq 0$.

If $(u, v) \notin E$, then $c(u, v) = 0$. 
If \((u, v) \in E\), then reverse edge \((v, u) \notin E\).

(Can work around this restriction.)

Source = vertex \(s\), sink = vertex \(t\), assume \(s \sim v \sim t\) for all \(v \in V\), so every vertex lies on some path from source to sink.

\[
\begin{align*}
\text{Value of flow} &= \left| f \right| \\
&= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)
\end{align*}
\]

In the example above, value of flow \(f = \left| f \right| = 3\).

**Maximum-flow problem**
Given \(G, s, t, \) and \(c\), find a flow whose value is maximum.

**Cuts**
A cut \((S, T)\) of a flow network \(G = (V, E)\) is a partition of \(V\) into \(S\) and \(T = V - S\) such that \(s \in S\) and \(t \in T\).

The net flow across cut \((S, T)\) is

\[
f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u).
\]

**Capacity** of a cut

\[
c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v).
\]

A minimum cut of \(G\) is a cut whose capacity is minimum over all cuts of \(G\).

Maximum flow \(\leq\) capacity of minimum cut.

Given a flow \(f\) in network \(G = (V, E)\).

That’s the residual capacity,

\[
c_f(u, v) = \begin{cases} 
    c(u, v) - f(u, v) & \text{if } (u, v) \in E \\
    f(u, v) & \text{if } (v, u) \in E \\
    0 & \text{otherwise} (i.e., (u, v), (v, u) \notin E) \in E).
\end{cases}
\]

The residual network is \(G_f = (V, E_f)\), where

\(E_f = \{(u, v) \in V \times V : c_f(u, v) > 0}\).

**Ford-Fulkerson algorithm**
Keep augmenting flow along an augmenting path until there is no augmenting path.
Represent the flow attribute using the usual dot-notation, but on an edge: \((u, v)_f\).

\[
\text{FORD-FULKERSON}(G, s, t) \\
\text{for all } (u, v) \in G.E \\
(u, v)_f = 0 \\
\text{while there is an augmenting path } p \text{ in } F_f \\
\text{augment } f \text{ by } c_f(P)
\]

5