CSE101 Quiz2 Review

This review doc summarizes the essential algorithms covered in lectures. It follows the section order to textbook to organize everything. Created by M. and Yilin, feel free to collaborate.

Quiz 2 Topics Lists

- MST, Trees, union/find data structure, Binary heap, Dijkstra, Prim, Kruskal
    - T/F, short answers, multiple choices
- Prove greedy algorithm - modify the solution

Greedy Algorithm

- **Format**
    - Instance: input
    - Solution format: output
    - Constraint: output's property to count as solution
    - Objective function: Quantity are we trying to max/min

Method 1: Modify-the-solution Recursion

- **Claim**
    - Let g_1 be the first greedy choice. Let OS be any other solution that meet all requirements and does not include g_1. Then there is a solution OS' that includes g_1, meets all constraints and is at least as good as OS.

- **Exchange argument**
    - State what you know: g_1 meet the condition for the first choice. OS meets all of the constraints for the problem
    - Define OS' in terms of g_1 and OS to include g_1. There might be multiple cases.

- **Prove exchange argument**
    - Show that OS' meet all constraints
    - Compare objective function of OS and OS' → OS' same or better

- **Prove the algorithm by induction**
    - For any instance I of size N, GS(I) is an optimal solution
    - By strong induction, Base case: N = 0 or 1, trivial case.
    - Assume for every instance I_2 of size $0 \le n \le N - 1$, $GS(I_2)$ is optimal solution. $GS(I) = g_1 + GS(I_2)$. By MTS lemma, there is an optimal solution OS' that also includes g_1. $OS' = g_1 + OS_2$, for some other solution OS_2 of instance I_2. Then by IH, GS(I) is at least as good as OS'. Since OS' is optimal, GS(I) is optimal.
    - $OS(I) \ge OS' = g_1 + S(I_2) \ge g_1 + GS(I_2) = GS(I)$

Method 2: Modify the solution - Iterative format

Basic idea:

- Prove for all i ≥ 1
  - **Min -** value(GS_i) ≤ value(OS_i)
  - **Max -** cost(GS_i) ≤ cost(OS_i)

- **IterMTS:** Let g_1, g_2, … g_T be the decisions made in order by the greedy strategy. For each 0 ≤ i ≤ T, there is an optimal solution OS_i that includes g_1, g_2... , g_i.

- **Prove by induction:**
- *Base case*: For i = 0, we let OS_0 be any optimal solution. Since it doesn't have to agree with any greedy decisions.
- Assume that there is an optimal solution OS_{i-1} that includes g_1, g_2, g_{i-1}. If it also includes g_i, we set OS_i = OS_{l-1}. Otherwise,
  - Define OS_i in a way that leaves g_1 to g_{i-1} unchanged, but but changes i'th move of OS_{i-1} to g_i.
  - Prove that OS_i meets constraints
  - Compare obj(OS_i) and obj(OS_{i-1}).

Minimum Spanning Tree

- Given a graph G = (V, E), MST is a tree T = (V, E') that minimizes total weight of T. **Acyclic, connected.**

- Properties:
  - 1. Remove a cycle edge cannot disconnect graph.
  - 2. A tree on n nodes has **(n - 1)** edges.
  - 3. Any connected, undirected graph G = (V, E) with **|E| = |V| - 1 is a tree**.
  - 4. An undirected graph is a tree iff a unique path between any pairs of nodes.

- **Cut property**
  - Suppose edge X are part of a MST of G = (V, E). Pick any subset of nodes S for which X doesn't cross between S and V - S, and let e be the lightest edge across this partition. Then X U {e} is a part of some MST.

- **Union/Find data structure**
  - Make root of the shorter tree point to the root of larger tree
  - Properties:

- For any x, rank(x) < rank (P(x))
- A node of rank k has at least 2^k descendants. Rank k + 1: union 2k.
- If there are n elements, there can be at most n / 2^k nodes of rank k.

- Runtime: **find/union → O(log(n)).**
- Path compression: reduce runtime to near O(1).

- **Kruskal's algorithm**

```
For all u belongs to V:
      Makeset (u);
X : {}
Sort the edges E by weight;
For all edges (u, v) belongs to E, in increasing order of weight:
      If find(u) ≠ find(v):
            Add edge (u, v) to X.
            Union (u, v)
Return X
```

- **Prim's algorithm**

```
X = {}
Repeat until |X| = |V| - 1
      Pick a subset of V S for which X has no edge between S and V - S.
      Let e be the min edge between S and V - S
      X = X U {e}
Return X.
```

<u>Shortest Path in graph</u>

- Dijkstra's algorithm

```
For all u belong to V:
      dist(u) = infinity
      prev(u) = null
dis(s) = 0

H = makequeue(V) (using dist-values as keys)
While H is not empty:
      U = deletemin(H)
      For all edges (u, v) belongs to E:
            If dist(v) > dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```

- **Runtime: depends on priority queue implementation**

Implementation	`deletemin`	`insert/ decreasekey`	$\|V\| \times$ `deletemin` + $(\|V\| + \|E\|) \times$ `insert`
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
d-ary heap	$O(\frac{d \log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O((\|V\| \cdot d + \|E\|) \frac{\log \|V\|}{\log d})$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$ (amortized)	$O(\|V\| \log \|V\| + \|E\|)$

- **Binary Heap:**
  - Each level is filled from left to right,
  - Key (parent) < children.

Quiz 1 Review Doc

- **chapter 3.1,3.2**
- **chapter 3.3,3.4**
- **chapter 4.1,4.2,4.3**
- **chapter 4.4,4.5**
- **chapter 4.5,5.1**

Topics1

- Max bandwidth path
  - Path
    - Simple path
      - No two edges are the same
    - **A single vertex is a trivial path to itself**
  - Objective
    - Over all possible paths p between v and u, find max BW(p)
- Graph reachability
  - Given a graph **G** and starting vertex **s**, give all reachable vertices v from s
    - **X:** a set of explored vertices
    - **F:** a set of reached but not explored vertices
    - **U:** a set of unreached vertices
  - procedure GraphSearch (G: directed graph, s: vertex)
    -
    - `Initialize X = empty, F = {s}, U = V – F.`
    - `While F is not empty:`
      - `Pick v in F.`
      - `For each neighbor u of v:`
        - `If u is not in X or F:`
          - `move u from U to F.`
      - `Move v from F to X.`
    - `Return X.`
  - Time analysis: **O(|V| + |E|)**

- **chapter 3.1,3.2**

- How big is ur graph
  - |E|
    - As small as |V|, if smaller, then the graph degenerates - **sparse**
    - As large as $|V|^2$, all possible connections - **dense**
  - Adjacency matrix vs adjacency list
    - Matrix always takes $O(|V|^2)$ space, so if the graph is **sparse**, that would be wasteful.
    - List always takes O(|E|)
- DFS in **undirected graphs**

- **procedure** explore(G, v):
    - visited(v);
    - previsited(v):
    - for each edge (v, u) ∈ E:
        - If not visited(u): explore(G, u)
    - postvisit(v);

- **procedure** dfs(G):
    - for all v ∈ V:
        - visited(v) = false;
    - for all v ∈ V:
        - If not visited(v): explore(v)
- Two steps when considering the runtime
    - Mark each spot as visited - **O(|V|)**
    - A loop in which scanned all edges - **O(|E|)**

- **Connectivity**
    - Connected components:
        - Subgraph internally connected but has edges to remaining graph
        - Each time DFS called explore, one connected component is picked out
- Previsit and povisit ordering
    - Define a counter **clock, initialized as 1**
        - **procedure** previsit(v):
            - Pre[v] = clock;
            - Clock++;
        - **procedure** postvisit(v):
            - Post[v] = clock;
            - Clock++;
    - **Property:**
        - For any nodes u and v, the two intervals [pre[u], post[u]] and [pre[v], post[v]] are either disjoint or contained within the other.

- **chapter 3.3,3.4**

- **Types of edges**
    - Forward edges
        - Lead to a non-child descendant
        - If u is an ancestor of v, $\mathbf{[_u, [_v,]_v,]_u}$
    - Back edges
        - Lead to ancestor
        - $\mathbf{[_v, [_u,]_u,]_v}$
    - Cross edges
        - Lead to a node that has been explored.
        - $\mathbf{[_v,]_v, [_u,]_u}$
- Directed acyclic graphs

- **Property**
  - DFS reveals a back edge **iff** this G has a cycle
  - Every edges leads to a vertex with lower post number in a **DAG**
    - Sink: smallest post number
    - Source: highest post number
  - **Every DAG has at least one source and at least one sink**
    - Linearization:
      - Find a **source**, output it and delete from G
      - Repeat until the graph is empty

- **Strongly connected components (SCCs)**
  - Two nodes u and v are connected in a graph if there is a path from u to v and a path from v to u.
  - Property
    - Every directed graph is a DAG of its SCCs.
- Decomposition of SCCs
  - How to locate the **sink**
    - The SCC of highest post number must be a **source** SCC
    - We only need to reverse the whole graph, the **source SCC** of G' will be the **sink SCC** of the original graph.
  - How to continue once the first **sink** is discovered
    - Run DFS on G^R (step 1)
    - Run the directed connected components algorithm on G, and during DFS, process the vertices in decreasing order of their post numbers from step 1

- chapter 4.1,4.2,4.3

- Distances
  - The distance between two vertices is the length of the shortest path between them

- BFS
  - **procedure** bfs(G, s):
    - for all u ∈ V:
      - dist(v) = infinite
    - dist(s) = 0;
    - Q = [s] (queue containing only s)
    - while Q is not empty:
      - u = eject(Q)
      - for all edges (u, v) ∈ E:
        - If dist(v) = infinite:
          - inject(Q, v)

- dist(v) = dist(u) + 1
- Lengths on edges

- chapter 4.4,4.5

- **Dijkstra's algorithm**
  - All edges have positive length
  - Using priority queue
    - Insert
      - Add a new element to the set
    - Decrease key
      - Decrease the value of certain key
    - Delete min
      - Return the element with the smallest key,
      - remove it from the set
    - Make heap
      - Build a priority queue out of given elements
- Priority Queue Implementations
  - Array
    - M

- chapter 4.5,5.1