Pragmatic Programming

Session 3 - Graphs

Max Nilsson

9th November

Representing Graphs

There are two common ways of representing graphs. In these slides we will let n be the number of nodes and m be the number of edges in the graph.

- As a (adjacency) matrix

  ```
  int g[mxn][mxn];
  ```

  where `g[u][v]` denotes the weight of the edge between `u, v`.
  This weight could in turn represent the distance between `u, v`.
  Note that `g[u][v] =` ∞ should be interpreted as that there exists no edge between `u, v`.
  The downside is that it has $\mathcal{O}(n^2)$ memory complexity, which may pass over to the time complexity in many algorithms.
- As a sparse matrix (adjacency list)

  ```
  vii g[mxn];
  ```

  where each `g[u][i]` contains a pair `(v, d)` which represents that there exists an edge between `u, v` with weight `d`.

Depth and Breadth First Search

- Depth first search can be implemented recursively

```
vi g[mxn];
bool vis[mxn];
void dfs(int u) {
    vis[u] = true;
    for (int v :  g[u]) if (!vis[v])
        dfs(v);
}
```

  with time complexity $\mathcal{O}(m)$. Used for Topological sorting (topic for later session) and Kosaraju's algorithm (finding strongly connected components).
- Breadth first search is a bit more tricky. It can be implemented with a queue<int> as shown last session. The time complexity is $\mathcal{O}(m)$ and can be used for path-finding. BFS is a special case of Dijkstra's algorithm.

Minimum Spanning Tree (Kruskal's Algorithm)

A minimum spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together and with the minimum possible total edge weight.

- Without loss of generality, it contains no cycles.
- Initialize a disjoint set data structure `disj`. First sort all the edges with respect to its weights. Then go through one edge `(d, u, v)` at a time and check if

  ```
  disj.find(u) == disj.find(v).
  ```

  If that is true, then we ignore the edge, otherwise we add the edge to our (so far) incomplete spanning tree and do

  ```
  disj.Union(u, v).
  ```

- Time complexity is

  $$\mathcal{O}(m \log m + m\alpha(n)) = \mathcal{O}(m \log m) = \mathcal{O}(m \log n)$$

  where $\alpha : \mathbb{R} \to \mathbb{R}$ is the inverse Ackerman function which is less than 5 for all reasonable values (even for $2^{2^{2^{65536}}}$ which does not even fit into a `long long long` if that even was a thing).

Dijkstra's Algorithm

Finds the shortest path between `s, t` in a graph with nonnegative edge weights in $\mathcal{O}(m \log m)$.

```
vii g[mxn];
ll dist[mxn];
void dijkstra(int s) {
    rep(i, 0, n) dist[i] = LLONG_MAX;
    priority_queue<pii, vii, greater<pii>> pq;)
    dist[s] = 0; pq.push(pii(0, s));
    while(pq.size()) {
        pii p = pq.top(); pq.pop();
        ll d = p.first, u = p.second;
        for (pii q :  g[u]) if(d+q.first < dist[q.second])
            {dist[q.second] = d+q.first;
            pq.push(pii(d+q.first, q.second));}
    }
}
```

Shortest Path with Negative Weights (Bellman-Ford)

Finds the shortest path between `s, t` in a graph with arbitrary edge weights in $\mathcal{O}(nm)$.

- It basically repeats Dijkstra's algorithm $n-1$ times without using a priority queue

```
vii g[mxn];
ll dist[mxn];
void bellman_ford(int s) {
    rep(i, 0, n) dist[i] = LLONG_MAX;
    dist[s] = 0;
    rep(_, 0, n-1)) {
        rep(u, 0, n) for (auto p :  g[u])
            if(dist[u]+q.first < dist[q.second])
                dist[q.second] = dist[u]+q.first;
    }
}
```

Shortest Path between All Pairs of Nodes (Floyd-Warshall)

Finds the shortest path between all pairs of nodes in a graph with arbitrary edge weights in $\mathcal{O}(n^3)$. In the following code, the adjacency matrix will be modified to represent the shortest paths.

```
int g[mxn][mxn];
void floyd_warshall() {
    rep(i, 0, n) g[i][i] = 0;
    rep(k, 0, n) rep(i, 0, n) rep(j, 0, n)
        if (g[i][j] > g[i][k] + g[k][j])
            g[i][j] = g[i][k] + g[k][j];
}
```

Note that the if time complexity is viable then so is the space complexity of the adjacency matrix.

Strongly Connected Components (Kosaraju)

A strongly connected component is a subgraph such that every pair of nodes is reachable by some path. Kosaraju's algorithm finds all such such components of a graph and it uses a stack `st`, two depth first searches and a reversed graph `g_rev`, resulting in an $\mathcal{O}(m)$. See video!

```
void dfs1(int u) {
    vis1[u] = true;
    for (int v :  g[u]) if (!vis1[v]) dfs1(v);
    st.push(u) };
void dfs2(int u, int repr) {
    vis2[u] = true; who[u] = repr;
    for (int v :  g_rev[u]) if (!vis2[v]) dfs2(v, repr);
}
void kosaraju() {
    rep(u, 0, n) if (!vis1[u]) dfs1(u);
    while(q.size()) { int u = st.top(); st.pop();
    if (!vis2[u]) dfs2(u, u); } }
```

Trees and Prüfer Sequences

- A tree is a connected undirected graph with $n-1$ edges.
- Consider a labeled tree, with labels $1, 2, \ldots, n$, and construct the following sequence of $n-2$ numbers. Iterate the following procedure $n-2$ times: remove the leaf with the lowest label and append its connected node.
- The resulting sequence is called a Prüfer sequence.
- The British mathematician Arthur Cayley (1821-95) proved the amazing **Cayley's Formula:** *Prüfer sequences provide a bijection between the set of labeled trees on n vertices and the set of sequences of length $n-2$ on the labels 1 to n.*
- A corollary to this is that there exists n^{n-2} labeled trees on n vertices (up to isomorphism of labeled trees).

Eulerian Paths, Cycles and Live Coding

- An Eulerian path is a path in a graph that visits every edge exactly once and may visit the same node multiple times.
- An Eulerian cycle is an Eulerian path that is also a cycle! In other words, it must begin and end in the same node.
- Recall that the degree of a node in an undirected graph is the number of edges connected to it.
- **Euler's Theorem:** *In an undirected graph, there exists an Eulerian cycle if and only if every node has even degree.*
- **Proof:** ($\Longrightarrow$) Direct.
  ($\Longleftarrow$) Use Euler's characteristic $V - E + F = 2$ and some imagination. $\square$
- Real life application of this theorem: Christmas Gifts.

What we will save for later

- Dynamic programming methods in graph algorithms.
- Matchings, cuts and flows.
- Treaps and topological sorting.
- Check out the boost graph library.
- More advanced stuff?

This Week

- Skim read parts of chapter 12 in Sannemo[1] and chapter 7 in Laaksonen[2].
- Solve half of this weeks 12 problems.

[1] Johan Sannemo. "Principles of Algorithmic Problem Solving". In: *Draft version* (2018).

[2] Antti Laaksonen. *Guide to competitive programming*. Springer, 2020.