1 Graph-Traversal

1.1 Breadth-First Search

Given a graph $G = (V, E)$, we want to discover all nodes a distance $dis(s, v)$. For BFS, we maintain a set S of all the nodes that we have travelled to. We push all of the vertices in Adj[i]. Also, we maintain a the parent of every node we find the shortest path of. Since we only place each node once in the set S, in total our algorithm will take $O(n + m)$.

1.2 Max trees

If we run BFS algo twice from an arbitrary node, the greatest dis[d] will be one of the points on the pair (o_1, o_2). Why? If there exists a longer path than (o_1, o_{i2}), then there has to be some contradiction or cycle, which is not allowed by the definition of a tree.

1.3 DFS-Visit

In DFS, we move through the most recent currently unexplored node. As soon as we find a node that is not yet explored, we mark it as "discovered", and immediately move to explore its edges. When we finish exploring every node, we mark it as "finished". However, DFS-visit only acts on the nodes reachable from s.

1.4 DFS

If we want to act on all the nodes of the graph, we need to consider the nodes not reachable from s. For two nodes (u, v), if there is overlap between $[u.d, v.d]$and$[u.f, v.f]$, then one is the ancestor of the other. Why is DFS said to create a *forest*? Since not all the elements might be reached with just the initial call of DFS, we might need to call it on other nodes originally unreachable from our starting node.

1.5 Edges

Four types w.r.t. DFS-forest F

- **tree edges**: edges $(u, v) \in$ F.
- **back edges**: edges (u, v) connecting u to an ancestor.
- **forward edges**: edges (u, v) connecting node to descendant.
- **cross edges**: all other edges(b/w disjoint trees....)

1.6 Topological Sort

We try to find an ordering of ndoes in a DAG. We can do this by comparing the finishing times of the DFS on each node.

2 Minimum Spanning Trees

MSTs are ways of joining all the nodes so that the overall cost of visiting all of them is minimized. Why are they MSTs? *Tree* becuase the cost for every weight is > 0 $\forall$ e. *Spanning* becuase it includes all of the nodes in the tree. *Minimum* b/c $w(H)$ is minimal. We "cut" two from the set of included nodes and the set that we haven't included. Prim's and Kruskal's algorithms basically have two sets, S and the set of unjoined nodes. For Kruskal's algorithm, we join the lowest edges connecting a node in and outside the set S. For Prim's, we only one node from the set S and the yet unjoined nodes.

3 Shortest Paths

Why are negative cylces not allowed when looking for shortest paths? If we choose the shortest paths with neg cycles, then we would infinitely choose the negative cycle.

3.1 Bellman-Ford

With this algo, we can detect the existence of negative cycles by using the `Relax(G,u,v)` method. If after relaxing all the nodes we can still further relax them afterwards, then it means we still have a negative cycle, since we when we relax all edges, we are finding the shortest paths, there would have to be another one if we could still relax them further.

3.2 Djikstra's Algorithm

With Djikstra's Algorithm, we build on the idea of BFS, we keep a set of the sorted estimates for the distance from our source node, then, we relax the edges in Adj[u]. Runs in $O(n^2)$

3.3 All-Pairs Shortest Paths

Here, we just need to find the path from all the nodes that is the min. We can run Djikstras on all nodes. We can also use dynammic programming to solve it, since we have the equation $l_{uv}^{(t)} = l_{uk}^{(t-1)} + w_{kv}$, basically the minimum paths of lengths $l - 1$ plus the weights to v.

3.4 Floyd-Warshall Algorithm

Using the previous formuls, we can also calculate the shortest path between two nodes. We take the minimum of the paths that cross node k and the paths that does not cross node k. Since we have three nested for loops, takes $O(n^3)$.

4 Maximum Flow

Given a network of pipes, how do we make a graph such that it has maximum flow from source to sink? We can allow k flow from s to t. We can also build reverse pipes, which cancel the flow, if any, between a node u and v.

4.1 Ford-Fulkerson Method

For the Ford-Fulkerson Algorithm, we continue while there exists an *augmenting path* hrough the *residual graph.* Basically, while there exists some flow that allows more efficient use of the capacities at each node, we keep increasing the flow at each node. Since $f(b, a) = c(a, b) - f(a, b)$, then we can show that there might be a path that allows for more flow if there is residual in any node.

4.2 Max-flow min-cut theorem

Here, suppose f is max flow in G, $G_f$contains no augmenting path from s to t, and abs$(f) = c(S, T)$

4.3 Applications

bipartite graphs are graphs which can be separated into two sections so that for every edge, one endpoint is in one and the other endpoint is in another section.

5 linear programming

With linear programming, we try and maximize the constraints on a given equation. Becuase the equations are solvable in a specific region, we have to find the *feasible region.* The solution must lie in the edges of the feasible region, since those are the regions that will have the greatest or smallest numbers.

6 NP completeness

There are countably infinite many polynomial-time algorithms, but uncountably infinite many problems. Why? **P**:Class of problems whose output is either "Yes" or "No", and whose running time is polynomial. **NP**: A

problem $L \in NP$ iff: $\exists$ polynomial-time algo and we can check the answer in polunomial-time($\exists$ Certificate). Graph connectivity, paths, etc...
NP-hard: $\forall$ problems in NP, $\exists$ polynomial-time reduction to another problem L.