21 Disjoint Set Data Structure

Used, for example, to represent multiple strongly-connected components.

Since a strongly-connected component of a graph is the maximal set of vertices satisfying a property, a vertex/edge may only belong to one SCC.

Each object in a set needs to be able to identify if other elements belong to the same set. this is easy to do if each set has a representative object that all objects in the set point to.

```
procedure MakeSet(Object o)                              ▷ O(1)
    o.size ← 1
    o.parent ← 1
end procedure
```

```
procedure FindSet(Object o)                              ▷ O(lg* n)
    if o.parent = 0 then return 0
    end if
    o.parent ← FindSet(o.parent)                         ▷ path compression
    return o.parent
end procedure
```

```
procedure Union(Object o, Object p)                      ▷ O(lg* n)
    x ← FindSet(o)
    y ← FindSet(p)
    if x.size ≥ y.size then
        y.parent ← x
        x.size ← x.size + y.size
    else
        x.parent ← y
        y.size ← y.size + x.size
    end if
end procedure
```

22 Graphs

BFS: O(V+E). Use a queue. Add every vertex to the queue as it is discovered for the first time.

DFS: Use recursive calls to DFS. (note: many applications of DFS require keeping track of the discovery/finish times from calling DFS)

```
procedure DFS(Vertex v)                                          ▷ Θ(V+E)
    if v has been discovered then
        return
    end if
    discover v
    for all vertices u such that v has an edge to u do
        DFS(u)
    end for
    finish v
end procedure
```

In DFS(u):

- (u, v) edge is a back edge $\iff$ v is discovered but not finished.
- (u, v) edge is a forward edge $\iff$ v is discovered and finished and u is not finished.
- (u, v) edge is a cross edge $\iff$ v is finished before u was discovered.

Example: With these discovery/finish times $(^1_A (^2_B (^3_C A)^4_C (^5_D)^6_D (^7_E D)^8_E)^9_B D)^{10}_A$ (where '(' indicates discovery and ')' indicates finishing), we have:

- (C, A) is a back edge because it occurs after $(_A$ and $(_C$
- (E, D) is a cross edge because it occurs after $(_D$, $)_D$, and $(_E$
- (A, D) is a forward edge because it occurs after $(_A$, $(_D$, and $)_D$

Topological Sort: Θ(V+E). Call DFS(G). As each vertex finishes, add it to the beginning of a linked list. Return this list.

A strongly-connected component of G is a maximal set of vertices C $\subseteq$ V such that for every u, v $\in$ C, u $\rightsquigarrow$ v and v $\rightsquigarrow$ u.

To find strongly-connected components of graph G

```
procedure StronglyConnectedComponents                            ▷ O(Θ(V+E)
    DFS(any vertex of G)
    for all vertices v in decreasing order of finish time do
        G^T.DFS(v) ▷ All nodes discovered in this call to DFS belong to the
            same strongly-connected component
    end for
end procedure
```

An alternate form of topological sort exists, which is also $\Theta(V+E)$. In the case it is called on a graph with cycles, it produces a flawed ordering and has poorer failure modes.

```
procedure AltTopologicalSort(Graph G)                    ▷ Θ(V+E)
    Create a queue Topo
    Create a queue Q
    for all vertices v ∈ G do
        if v.indegree = 0 then
            Q.enqueue(v)
        end if
    end for
    while Q nonempty do
        vertex u ← Q.dequeue
        Topo.enqueue(u)
        for all vertices v such that (u, v) ∈ G do
            v.indegree ← v.indegree - 1
            if v.indegree = 0 then
                Q.enqueue(v)
            end if
        end for
    end while
    for all vertices v ∈ G do
        if v.indegree ≠ 0 then                ▷ Found a cycle. Uh-oh!
        end if
    end for
    return Topo
end procedure
```

23 Minimum Spanning Trees

Kruskal's Algorithm makes use of the disjoint set data structure.

Prim's Algorithm makes use of a min-Priority Queue to retrieve vertices in non-descending order of weight; and also so that their weights can be decreased.

24 Single-source Shortest Paths

It's often useful to figure out the shortest paths within a graph G, from vertex v to every other vertex in G. This general problem has several variants.

```
procedure KruskalsMST(Graph G)                                    ▷ O(E lg V)
    for all vertices v ∈ G do
        MakeSet(v)
    end for
    sort the edges of G in non-descending order by edge-weight
    for all edges (u, v) in non-descending order by weight do
        if FindSet(u) ≠ FindSet(v) then
            add edge (u, v) to MST
            Union(u, v)
        end if
    end for
end procedure
```

```
procedure PrimsMST(Graph G)                                       ▷ O(E lg V)
    for all vertices v ∈ G do                          ▷ initialize all vertices
        key[v] ← ∞
        π[v] ← null
    end for
    key[some initial vertex] ← 0
    PriorityQueue Q ← all vertices
    while Q not empty do
        vertex u ← ExtractMin(Q)
        for all vertices v such that (u, v) is an edge in G do
            if v ∈ Q and weight(u, v) ¡ key[v] then
                π[v] ← u
                key[v] ← weight(u, v)
            end if
        end for
        add edge (π[u], u) to MST
    end while
end procedure
```

- Single-destination shortest paths: This variant reduces to a SSSP problem if you reverse the direction of every edge in G.
- Single-pair shortest paths: If you need to find the shortest path between vertices u and v in G, solving the SSSP general problem will give you an answer. While it may seem like overkill, no algorithm is known that will solve this problem better than a SSSP algorithm.

Shortest paths have an interesting property: If a $\rightsquigarrow$ c is the shortest path from a to c, and passes through vertex b, then a $\rightsquigarrow$ b is the shortest path from a to b and b $\rightsquigarrow$ c is the shortest path from b to c.

Some gotchas:

- Some SSSP algorithms don't produce correct results (or even halt) if the graph contains negative edge-weights. In particular, cycles of negative weight can create a shortest path with weight $-\infty$.
- Cycles of weight 0 contribute no savings to overall path weight, yet add edge traversals. So no SSSP should contain a 0-weight cycle.
- Positive weight cycles have higher weight than the same path without the cycles, so no SSSP will contain one.

For Graph G and starting vertex s, SSSP algorithms use these common functions:

---

procedure InitializeSingleSource(Graph G, Vertex v) $\triangleright$ Θ(V)
    for all vertices v $\in$ G **do**
        distance[v] $\leftarrow \infty$
        parent[v] $\leftarrow$ *null*
    end for
    distance[s] $\leftarrow$ 0
end procedure

---

procedure Relax(Source Vertex u, Destination Vertex v, Weight Function w) $\triangleright$ O(1)
    if distance[v] > distance[u] + w(u, v) **then**
        distance[v] $\leftarrow$ distance[u] + w(u, v)
        parent[v] $\leftarrow$ u
    end if
end procedure

Here's the first SSSP algorithm:

```
procedure BellmanFord(Graph G, Weight Function w, Starting Vertex s)    ▷ O(VE)
    InitializeSingleSource(G, s)
    for i ← 1 to —V(G)— - 1 do
        for all edge (u, v) in G do
            Relax(u, v, w)
        end for
    end for
    for all edge (u, v) ∈ G do
        if distance[v] > distance[u] + w(u, v) then
            return false    ▷ found a negative-weight cycle
        end if
    end for
    return true    ▷ found no negative-weight cycles
end procedure
```

Though Bellman-Ford is slow, one benefit is that it can determine if there are negative-weight cycles in G, which are detectable if the shortest path keeps getting smaller past a fixed number of relaxation attempts.

DagShortestPaths only works on a DAG (directed acyclic graph), but is faster than Bellman-Ford.

```
procedure DagShortestPaths(Graph G, Weight Function w, Starting Vertex s)    ▷ Θ(V+E)
    T ← a list of the vertices of G, sorted topologically
    InitializeSingleSource(G, s)
    for all vertices u ∈ T do
        for all vertices v such that (u, v) is an edge in G do
            Relax(u, v, w)
        end for
    end for
end procedure
```

Dijkstra's/Dantzig's algorithm finds SSSPs in much the same way as Prim's algorithm finds MSTs. Whereas Prim's uses a Min-Priority Queue to keep track of minimum edge weights to a vertex, Dijkstra's associates with each vertex the minimum path weight to reach that vertex.

Dijkstra's algorithm will give the wrong answer if the graph it's used on has negative weights.

```
procedure DIJKSTRA(Graph G, Weight Function w, Starting Vertex s)  ▷ O(E lg V)
    INITIALIZESINGLESOURCE(G, s)
    MinPriorityQueue Q ← all vertices ∈ G
    while Q is not empty do
        Vertex u ← EXTRACTMIN(Q)
        for all vertices v such that (u, v) ∈ G do
            RELAX(u, v, w)
        end for
    end while
end procedure
```

25 All-Pairs Shortest Paths

The Floyd-Warshall algorithm is similar to the Bellman-Ford SSSP algorithm, but with more work and storage. Whereas B-F stored, for each destination vertex, what the parent and path-cost is on a path originating at some source vertex, F-W must store those two facts for each source vertex × destination vertex pair.

Basically, you start with an adjacency matrix. Then, for each vertex, { v_1, v_2, ..., v_n } you derive a matrix from it that indicates the shortest paths from each vertex to each other vertex by storing only information about vertices that precede it on the path.

```
procedure FLOYDWARSHALL(n × n Matrix W)  ▷ Θ(V^3) time, O(V^2) memory
    n ← rows(W)
    D^0 ← W
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                d^k_{i,j} ← min(d^{k−1}_{i,j}, d^{k−1}_{i,k} + d^{k−1}_{k,j})
            end for
        end for
        Destroy D^{k−1}
    end for
    return D^n
end procedure
```

i $\rightsquigarrow$ k $\rightsquigarrow$ j implies that i $\rightsquigarrow$ k and k $\rightsquigarrow$ j. So if you're running Floyd-Warshall, you can save a lot of arithmetic by applying these rules:

1. if a row has ∞ in column k, then nothing in this row will change during

this iteration of the outer loop. i.e., i $\rightsquigarrow$ j does not pass through k if i has no path to k.

2. if a column has ∞ in row k, then nothing in that column will change during this k. This is because i $\rightsquigarrow$ j $\rightsquigarrow$ k only if j $\rightsquigarrow$ k.