Connected Components

A connected component is defined as a subgraph where there exists a path between any two vertices in it. Graph $G$ is made up of separate connected components and it may be useful to be able to classify each vertex by which connected component it belongs to.

For undirected graph $G$, executing a BFS or DFS starting from a vertex $v$ will visit every other vertex in the same connected component as $v$. We can mark every vertex visited from a BFS/DFS from $v$ as being “owned” by $v$. As we iterate through all the vertices, we execute a BFS/DFS starting from a vertex if it has no owner (i.e. it is part of an undiscovered connected component) and mark all the vertices visited in that BFS/DFS. After iterating through all the vertices, each vertex will be marked by its owner, representing which connected component it is a part of. In summary, the algorithm is the following:

1. For each vertex $v$ in undirected graph $G$
   
   (a) If $v$ has no owner, it is part of an undiscovered connected component. Execute BFS or DFS starting from $v$ and mark all the vertices as being owned by $v$
   
   (b) Else, if $v$ has an owner, it is part of a connected component we’ve already discovered. Ignore $v$ and move on to the next vertex.

The runtime of this algorithm is $O(|V| + |E|)$ since each vertex is visited twice (once by iterating through it in the outer loop, another by visiting it in BFS/DFS) and each edge is visited once (in BFS/DFS).
**Strongly Connected Components**

The algorithm above does not work with directed graphs. For undirected graphs, finding a path from \( u \) to \( v \) implies that there exists a path from \( v \) to \( u \). This is not the case for directed graphs. We can still separate the directed graphs into **strongly connected components**, which are components in directed graphs where any two vertices has a path in between each other. Note that this is the same definition as connected components above, but applied to directed graphs.

The intuition that will help us separate a directed graph into strongly connected components is realizing that a strongly connected component with its edges’ directions reversed is still a strongly connected component. We will introduce \( G^T \), which is the transpose of directed graph \( G \). \( G^T \) and \( G \) are the same graph except the edge directions are reversed in \( G^T \), i.e. if edge \((u,v)\) is in \( G \), then the edge \((v,u)\) is in \( G^T \). An algorithm to find strongly connected components goes as follows:

1. Execute DFS on \( G \) (starting at an arbitrary starting vertex), keeping track of the finishing times of all vertices
2. Compute the transpose, \( G^T \)
3. Execute DFS on \( G^T \), starting at the vertex with the latest finishing time, forming a tree rooted at that vertex. Once a tree is completed, move on to the unvisited vertex with the next latest finishing time and form another tree using DFS and repeat until all the vertices in \( G^T \) are visited
4. Output the vertices in each tree formed by the second DFS as a separate strongly connected component

![Graph](image)

We can reduce a directed graph \( G \) to a graph of its strongly connected components, as seen above. Note that the graph of \( G \)’s strongly connected components cannot contain a cycle, since
a cycle of strongly connected components can itself be reduced into a single strongly connected component. We call a directed graph with no cycles a **dag**, short for directed acyclic graph. We can thus say that every directed graph $G$ can be reduced to a dag of its strongly connected components.

### Notation: Shortest Paths

Bellman-Ford and Dijkstra both solve the problem of finding a shortest path from a source vertex to each other vertex in the graph. We will designate the source vertex as $s$. Every vertex $v$ in the graph is augmented with the following parameters:

- **$v.d$** - The weight of the current shortest path from $s$ to $v$. This is initialized to be $\infty$ for all vertices besides the source vertex but decreases as paths are found and shorter paths are discovered. At the end of the algorithms, this will be the weight of the shortest path. $s.d$ is initialized to be 0.

- **$v.\pi$** - The parent vertex of $v$ in the current shortest path. This is initialized to be NIL but gets set to a vertex once a path is discovered from $s$ to $v$. As shorter paths to $v$ are discovered, the parent updates to reflect the change. At the end of the algorithms, this will be the parent of $v$ in the shortest path to $v$. $s.\pi$ will always be NIL.

Also,

- **$w(u, v)$** is the weight of the edge from vertex $u$ to vertex $v$

- **$\delta(u, v)$** is the weight of the shortest path from vertex $u$ to vertex $v$

### Relaxation

Initializing the algorithms involves setting $v.d$ to $\infty$ and $v.\pi$ to NIL. Throughout the course of the algorithms, we will need to update these values to find shortest paths.

The idea is that if we found a path costing $u.d$ from $s$ to $u$ and there is an edge from $u$ to $v$, then the upper bound on the weight of a shortest path from $s$ to $v$ is $u.d$ plus the weight of the edge between $u$ and $v$. We can thus compare $u.d + w(u, v)$ to $v.d$ and update $v.d$ if $u.d + w(u, v)$ is smaller than the current $v.d$. In pseudocode, relaxing the edge $(u, v)$ is:

```plaintext
RELAX(u, v):
    if v.d > u.d + w(u, v) ## if we find a shorter path to v through u
        v.d = u.d + w(u, v) ## update current shortest path weight to v
        v.pi = u ## update parent of v in current shortest path to v
```

Both Bellman-Ford and Dijkstra use relaxation to discover shortest paths. The difference between the two is the order in which edges are relaxed.
Properties of Shortest Paths

Using our definitions of shortest paths and relaxations, we can come up with several properties. These can all be found in CLRS in chapter 24.

**Triangle inequality:** For any edge \((u, v)\), we have \(\delta(s, v) \leq \delta(s, u) + w(u, v)\). In english, the weight of the shortest path from \(s\) to \(v\) is no greater than the weight of the shortest path from \(s\) to \(u\) plus the weight of the edge from \(u\) to \(v\).

**Optimal substructure:** Let \(\{v_1, v_2, v_3, ..., v_k\}\) be a shortest path that goes from \(v_1\) to \(v_k\) through the vertices \(v_2\) through \(v_{k-1}\). Any subpath \(\{v_i, v_{i+1}, ..., v_{j-1}, v_j\}\) must be a shortest path from \(v_i\) to \(v_j\). That is, a shortest path is constructed of shortest paths between any two vertices in the path.

**Upper-bound property:** We always have \(v.d \geq \delta(s, v)\) for all vertices \(v\). Once \(v.d = \delta(s, v)\), it never changes.

**No-path property:** If there exists no path from \(s\) to \(v\), \(v.d\) will always be \(\infty\).

**Convergence property:** If a shortest path from \(s\) to \(v\) contains the edge \((u, v)\) and \(u.d = \delta(s, u)\) before relaxing edge \((u, v)\), then \(v.d = \delta(s, v)\) at all times after relaxing edge \((u, v)\).

**Path-relaxation property:** Let \(\{v_1, v_2, v_3, ..., v_k\}\) be a shortest path that goes from \(v_1\) to \(v_k\). If the edges are relaxed in the order \((v_1, v_2), (v_2, v_3), \text{etc.}\), then \(v_k.d = \delta(s, v_k)\) once the whole path is relaxed.

**Predecessor-subgraph property:** Once \(v.d = \delta(s, v)\) for all vertices \(v\), the predecessor subgraph is a shortest-paths tree rooted at \(s\). The predecessor subgraph is the subgraph of \(G\) that contains all the vertices with a finite distance from \(s\) (i.e. reachable from \(s\)) and only the edges that connect \(v\) to \(v.\pi\).

Shortest Path and BFS

In the past, we were able to use breadth-first search to find the shortest paths between a source vertex to all other vertices in some graph \(G\). The reason it worked is that each edge had equal weight (e.g., 1) so the shortest path between two vertices was the one that contained the fewest edges. Now, we introduce edge weights so the cost of traveling through edges can differ from edge to edge. The shortest path between two vertices is defined to be the path whose sum of edge weights is the least. BFS will not work on weighted graphs since the path with the fewest edges may not be the shortest if the edges it contains are expensive.

However, if all the weights are integers and they are bounded by a small number, say \(k\), we
can still use BFS. To do this, for each edge \((u, v)\), we split it into \(w(u, v)\) edges with weight 1 connecting \(u\) to \(v\) through some dummy vertices. Then we do BFS on the new graph to find the shortest path. The time complexity will be \(O(kE)\) if the original graph has \(|E|\) edges.

Graph Transformation

Shortest path with even or odd length

Given a weighted graph \(G = (V, E, w)\), suppose we only want to find a shortest path with odd number of edges from \(s\) to \(t\). To do this, we can make a new graph \(G'\). For every vertex \(u\) in \(G\), there are two vertices \(u_E\) and \(u_O\) in \(G'\): these represent reaching the vertex \(u\) through even and odd number of edges respectively. For every edge \((u, v)\) in \(G\), there are two edges in \(G'\): \((u_E, v_O)\) and \((u_O, v_E)\). Both of these edges have the same weight as the original. Constructing this graph takes linear time \(O(V + E)\). Then we can run shortest path algorithms from \(s_E\) to \(t_O\).