Plan:

- Strongly Connected Components
- Shortest Path

---

Algorithm for Biconnected Components

Maintain dfs and low numbers for each vertex.

The edges of an undirected graph are placed on a stack as they are traversed.

When an articulation point is discovered, the corresponding edges are on the top of the stack.

Therefore, we can output all biconnected components during a single DFS run.

---

Algorithm for Biconnected Components

for all v in V do dfs[v] = 0;
for all v in V do if dfs[v]==0 BCC(v);

k = 0; S - empty stack;

BCC(v) {
    k++; dfs[v] = k; low[v]=k;
    for all w in adj(v) do
        if dfs[w]==0 then
            push((v,w), S);
            BCC(w);
            low[v] = min( low[v], low[w] );
        else if dfs[w] < dfs[v] && w ∈ S then
            push((v,w), S); low[v] = min( low[v], dfs[w] );
    }
}

---

DFS on Directed Graphs

Strongly connected vs. weakly connected
**Strongly Connected Components**

G is strongly connected if every pair (u, v) of vertices is reachable from one another.

A strongly connected component (SCC) of G is a maximal set of vertices C ⊆ V such that for all vertices in C are reachable.

**Equivalent classes**

partitioning of the vertices

Two vertices v and w are equivalent, denoted u=v, if there is a path from u to v and one from v to u.

The relation = is an equivalence relation.

Reflexivity v = v. A path of zero length exists.

Symmetry if v = u then u = v. By definition.

Transitivity if v = u and u = w then v = w.

Join two paths to get one from v to w.

The equivalent class of = is called a strongly connected component.

**DAG of SCCs**

Choose one vertex per equivalent class.

Two vertices are connected if the corresponding components are connected by an edge.

The resulting graph is a DAG.

Applications... social networks

**Preamble**

Def. low[v] is the smallest dfs-number of a vertex reachable by a back edge from the subtree of v.

Def. A vertex is called a base if it has the lowest dfs number in the SCC.

Lemma 1. Let b be a base in a component X, then any v∈X is a descendant of b and all they are on the path b-v.

Lemma 2. A vertex is a base iff dfs[v] = low[v].

**The Algorithm**

for all v in V do dfs[v] = 0;
for all v in V do if dfs[v]=0 SCC(v);
k = 0; S - empty stack;
SCC(v) {
  k++; dfs[v] = k; low[v] = k; push(v, S);
  for all w in adj(v) do
    if dfs[w]=0 then
      SCC(w); low[v] = min( low[v], low[w] );
    else if dfs[w] < dfs[v] && w ∈ S then
      low[v] = min( low[v], dfs[w] );
  if low[v]=dfs[v] then //base vertex of a component
    pop(S) where dfs(u_stack) ≥ dfs(v); // output
**The Algorithm**

Store vertices on a stack as you run DFS

![Graph](image)

**Correctness**

**Theorem.** After the call to SCC(v) is complete it is a case that
(1) low[v] has been correctly computed
(2) all SCCs contained in the subtree rooted at v have been output.

Proof by induction on calls.

First we prove 1) and then 2).

(1) low[v] correctly computed

for all w in adj(v), do
  if dfs[w]=0 then
    SCC(w); low[v] = min( low[v], low[w] );
  else if dfs[w] < dfs[v] && w ∈ S then
    low[v] = min( low[v], dfs[w] );

Case a) w ∈ S. Then there is a path w-v. Combining this path with edge (v,w) assures that v and w in the same component.

Case b) w ∉ S. Then the rec. call to w must have been completed.

(2) all SCCs contained in the subtree rooted at v have been output.

if low[v] = dfs[v] then //base vertex of a component
  pop(S) where dfs(u_stk) ≥ dfs(v); / output

By lemma 2, v is a base vertex.

We have to make sure that we pop only vertices from the same component.

Let be another base vertex b that descends from v.

Let assume that there is w (in the same component as v) that descends from both v and b.

There must be a path w-v.

By lemma 1 there is a path v-b. And also b-w.

Cycle w-v-b-w. So, v and b are in the same component.

**Lemma 1.** Let b be a base vertex in a component X, then any v∈X is a descendant of b and all they are on the path b-v.

**Proof.** We know that either
(1) v descends from b, or
(2) b descends from v, or
(3) neither of the above.

(2) is impossible since b has the lowest dfs-num.

Suppose (3). There is a path b-v (same component)

Find the least common ancestor r of all vertices on b-v path. We claim path goes through r.

**Case 1.**

Since dfs[b]-dfs[v], T_b and T_v are disjoint - there are not an edge between them.

**Case 2.** b and v in the same DFS tree.

b-v path must touch at least two DFS trees, (r is the least)

It follows, b-v path starts in one tree, goes through one or more another subtrees and come back.

Impossible to come back, since dfs-num in one tree is less than in another.

Find the least common ancestor r of all vertices on b-v path. We claim path goes through r.

**Case 1.**

Since dfs[b]-dfs[v], T_b and T_v are disjoint - there are not an edge between them.

**Case 2.** b and v in the same DFS tree.

b-v path must touch at least two DFS trees, (r is the least)

It follows, b-v path starts in one tree, goes through one or more another subtrees and come back.

Impossible to come back, since dfs-num in one tree is less than in another.

Exercise: Consider a directed graph G with vertices A, B, C, D, and E. The edges are as follows:

- A → B
- B → C
- C → D
- D → E
- E → A

The algorithm is run on this graph using a depth-first search (DFS) order. The order of visiting vertices is A, B, C, D, E. The algorithm marks the vertices as visited and assigns a discovery time (dfs) and a low time (low) to each vertex. The discovery time is the time when the vertex is first visited, and the low time is the minimum of the discovery time and the low time of its children.

**Exercise: What are the discovery times and low times for all vertices in the graph?**

**Solution:***

- **A:** dfs = 1, low = 1
- **B:** dfs = 2, low = 2
- **C:** dfs = 3, low = 3
- **D:** dfs = 4, low = 4
- **E:** dfs = 5, low = 5

The graph is strongly connected, and the algorithm correctly identifies all connected components.
Lemma 2.

A vertex is a base iff \( \text{dfs}[v] = \text{low}[v] \).

A proof left as an exercise...

The Shortest Path Problem

Given a positively weighted graph \( G \) with a source vertex \( s \), find the shortest path from \( s \) to all other vertices in the graph.

Greedy approach

When algorithm proceeds all vertices are divided into two groups:
- vertices whose shortest path from the source is known
- vertices whose shortest path from the source is NOT known

Move vertices (shortest distance) one at a time from the unknown set to the known set.

Maintain a PQ of distances from the source to a vertex

Complexity

\[ O(V \log V + E \log V) \]

Let \( D(v) \) denote a length from the source \( s \) to vertex \( v \). We store distances \( D(v) \) in a PQ.

\[ O(V) \quad \text{PQ has } V \text{ vertices} \]

\[ \text{INIT: } D(s) = 0; \quad D(v) = \infty \text{ } O(\log V) \]

\[ \text{LOOP: } O(\log V) \]

Delete a node \( v \) from PQ using deleteMin()

Update \( D(w) \) for all \( w \) in \( \text{adj}(v) \) using decreaseKey()

\[ O(\log V) \]

We do \( O(E) \) updates

\[ D(w) = \min(D(w), D(v) + c(v, w)) \]
Assume that a unsorted array is used instead of a priority queue. What would the algorithm’s running time in this case?

$O(V^2 + E)$

**PQ is a linear array**

findMin takes $O(V)$ - for one vertex
findMin takes $O(V^2)$ - for all vertices

Updating takes $O(1)$ - for one edge
total edge adjustment $O(E)$

the algorithm running time $O(E + V^2)$

Why Dijkstra’s algorithm does not work on graphs with negative weights?

The Bellman-Ford algorithm (1958)

repeat $V - 1$ times:
for all $e$ in $E$:
update($e$)

The Bellman-Ford Algorithm

for ($k = 0; k < V; k++$) dist[$k$] = INFINITY;

Queue $q$ = new Queue();
dist[$s$] = 0; $q$.enqueue($s$);
while ($q$.isEmpty())
{
    $v$ = $q$.dequeue();
    for each $w$ in adj($v$) do
        if (dist[$w$] > dist[$v$] + weight[$v$,$w$]) {
            dist[$w$] = dist[$v$] + weight[$v$,$w$];
            if (!$q$.isInQueue($w$)) $q$.enqueue($w$);
        }
}

What is the worst-case complexity of the Bellman-Ford algorithm?

for ($k = 0; k < V; k++$) dist[$k$] = INFINITY;

Queue $q$ = new Queue();
dist[$s$] = 0; $q$.enqueue($s$);
while ($q$.isEmpty())
{
    $v$ = $q$.dequeue();
    for each $w$ in adj($v$) do
        if (dist[$w$] > dist[$v$] + weight[$v$,$w$]) {
            dist[$w$] = dist[$v$] + weight[$v$,$w$];
            if (!$q$.isInQueue($w$)) $q$.enqueue($w$);
        }
}
Graph with a negative cycle?

How would you apply the Bellman-Ford algorithm to find out if a graph has a negative cycle?

Dynamic programming approach

For each node, find the length of the shortest path to \( t \) that uses at most 1 edge, or write down \( \infty \) if there is no such path.

Suppose for all \( v \) we have solved for length of the shortest path to \( t \) that uses \( k - 1 \) or fewer edges. How can we use this to solve for the shortest path that uses \( k \) or fewer edges?

We go to some neighbor \( x \) of \( v \), and then take the shortest path from \( x \) to \( t \) that uses \( k - 1 \) or fewer edges.

All-Pairs Shortest Paths (APSP)

Given a weighted graph, find a shortest path from any vertex to any other vertex.

Note, no distinguished vertex

All-Pairs Shortest Paths

One approach: run Dijkstra's algorithm using every vertex as a source.

Complexity: \( O(V E \log V) \)

- sparse: \( O(V^2 \log V) \)
- dense: \( O(V^3 \log V) \)

But what about negative weights...

APSP: Bellman-Ford's

Complexity: \( O(V^2 E) \)

Note, for a dense graph we have \( O(V^4) \).
**APSP:**
Dynamic programming approach

Floyd-Warshall, $O(V^3)$

See lecture 5

---

**APSP: Johnson's algorithm**

Complexity: $O(VE + VE \log V)$

for a dense graph -- $O(V^3 \log V)$.

for a sparse graph -- $O(V^2 \log V)$.

---

**Johnson's Algorithm**

It improves the runtime only when a graph has negative weights.

A bird’s view:
- Reweight the graph, so all weights are nonnegative (by running Bellman-Ford’s)
- Run Dijkstra’s on all vertices

---

**Johnson’s Algorithm: intuition**

The way to improve the runtime is to run Dijkstra’s from each vertex.

But Dijkstra’s does not work on negative edges.

So what about if we change the edge weight to be nonnegative?

We have to be careful on changing the edge weight... to preserve the shortest path.

---

**Wrong reweighting (adding the fix amount)**

The actual shortest path to $X$ is $S$-$B$-$C$-$X$

Let us add 3 to all edges

---

**Wrong reweighting (adding the fix amount)**

Shortest path to $X$ is $S$-$A$-$X
Adding the fix amount does not work, since every shortest path has a different number of edges.

**Johnson's Algorithm: reweighting**

Every edge \((v, u)\) with the cost \(w(v, u)\) is replaced by

\[
w'(v, u) = w(v, u) + p(v) - p(u)
\]

where \(p(v)\) will be decided later.

\[
w'(v, u) = 2 + (-2) - 1 = -1
\]

**Theorem.** All paths between the same two vertices are reweighted by the same amount.

**Proof.**
Consider path \(v = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n = u\)
Then we have

\[
w'(v,u) = w'(v_1, v_2) + \ldots + w'(v_{n-1}, v_n)
= w(v_1, v_2) + p(v_1) - p(v_2) + \ldots
\]

Telescoping sum

\[
w'(v, u) = w(v, u) + p(v) - p(u)
\]

**Find vertex labeling \(P(v)\)**

First we need to create a new vertex and connect it to all other vertices with zero weight.
Note this change in the graph won’t change the shortest distances between vertices.

**Running SSSP**

Next we run Bellman-Ford’s starting at vertex \(s\).

Shortest Path \(s-a\) is 0
\(s-b\) is -2
\(s-c\) is -3
and so on…

Now we define \(p(v)\) as the shortest distance \(s-v\).
Johnson's Reweighting

Here we redraw the example by using

\[ w^*(v, u) = w(v, u) + p(v) - p(u) \]

Edge (a,b): \((-2+0)(-2) = 0\)
Edge (b,c): \((-1+(-2))(-3) = 0\)
Edge (z,x): \(1+0-(-1) = 2\)

New graph

After Johnson's reweighting we get a new graph with non-negative weights.
Remember, Johnson's reweighting preserves the shortest path.

Now we can use Dijkstra's

Johnon's Algorithm:

1. Add a new vertex \(s\) and connect it with all other vertices.
2. Run Bellman-Ford's algorithm from \(s\) to compute \(p(v)\).
   Note that Bellman-Ford's algorithm will correctly report if the original graph has a negative cost cycle.
3.Reweight all edges: \(w^*(v,u) = w(v,u) + p(v) - p(u)\)
4. Run Dijkstra's algorithm from all vertices
5. Compute the actual distances by subtracting \(p(v) - p(u)\)

Theorem. After reweighting every edge has a nonnegative cost.
Proof. Consider edge \((v, u)\)
\(p(v)\) is the shortest distance from \(s\) to \(v\)
\(p(u)\) is the shortest distance from \(s\) to \(u\)
\(p(u) \leq p(v) + w(v, u)\)
since the shortest path \(s-u\) cannot be longer then \(p(v) + w(v, u)\).

Complexity

1. Add a new vertex \(s\) and connect it with all other vertices. \(O(V)\)
2. Run Bellman-Ford's algorithm from \(s\) to compute \(p(v)\). \(O(V E)\)
3. Reweight all edges: \(w^*(v,u) = w(v,u) + p(v) - p(u)\) \(O(E)\)
4. Run Dijkstra's algorithm from all vertices \(O(V E \log V)\)
5. Compute the actual distances by subtracting \(p(v) - p(u)\) \(O(E)\)
Total: \(O(V E \log V)\)

Johnson's Algorithm

1. Add a new vertex \(s\) and connect it with all other vertices.
2. Run Bellman-Ford's algorithm from \(s\) to compute \(p(v)\).
   Note that Bellman-Ford's algorithm will correctly report if the original graph has a negative cost cycle.
3. Reweight all edges: \(w^*(v,u) = w(v,u) + p(v) - p(u)\)
4. Run Dijkstra's algorithm from all vertices
5. Compute the actual distances by subtracting \(p(v) - p(u)\)

It shines for sparse graphs with negative edges

\(O(V^2 \log V)\)

Better than Floyd-Warshall's, which is \(O(V^3)\)