1 Articulation Points Detection Algorithm


In a graph $G = (V, E)$, $v$ is an articulation point if:

- removal of $v$ in $G$ results in a disconnected graph
- If $v$ is an articulation point, then there exist distinct vertices $w$ and $x$ such that $v$ is in every path from $w$ to $x$.

Finding articulation points can be nicely done by using Depth-First Search. In a DFS tree of an undirected graph, a node $u$ is an articulation point, for every child $v$ of $u$, if there is no back edge from $v$ to a node higher in the DFS tree than $u$. That is, every node in the decedent tree of $u$ have no way to visit other nodes in the graph without passing through the node $u$, which is the articulation point. Thus, for each node in DFS traversal, we calculate $dfsn(v)$ and $LOW(v)$. The definition of $LOW(v)$ is the lowest $dfsn$ of any vertex that is either in the DFS subtree rooted at $v$ or connected to a vertex in that subtree by a back edge. Then, in DFS, if there is no more nodes to visit, we back up and update the values of $LOW$ as we return from each recursive call.

Global initialization: $v.dfsnum ← −1$, for all $v$.

**Algorithm 1 ArticPointDFS**

```plaintext
Require: Vertex $v$

1: $v.dfsnum ← dfsCounter + +$
2: $v.low ← v.dfsnum$
3: for all edge $(v, x)$ do
4: if $x.dfsnum = −1$ then // x is undiscovered
5: $x.dfslevel ← v.dfslevel + +$
6: $v.numChildren ← v.numChildren + +$
7: stack.push edge $(v, x)$ // add this edge to the stack
8: ArticPointDFS($x$) // recursively perform DFS at children nodes
9: $v.low ← \min(v.low, x.low)$
10: if $v.dfsnum = 1$ then
11: // Special Case for root :
12: // Root is an artic. point iff there are two or more children
13: if $v.numChildren \geq 2$ then
14: articPointList.add($v$)
15: // Retrieve all edges in a biconnected component
16: while stack.top $\neq (v, x)$ do
17: bccEdgeList.add(stack.pop)
18: else
19: if $x.low \geq v.dfsnum$ then
20: // v is artic. point separating x.
21: // Children of $v$ cannot climb higher than $v$ without passing through $v$.
22: articPointList.add($v$)
23: while stack.top $\neq (v, x)$ do // Retrieve all edges in a biconnected component
24: bccEdgeList.add(stack.pop)
25: else if $x.dfslevel < v.dfslevel − 1$ then
26: // x is at a lower level than the level of $v$'s parent, equivalent to $(v, x)$ is a back edge
27: $v.low ← \min(v.low, x.dfsnum)$
28: stack.push edge $(v, x)$ // add the back edge to the stack
```

2 Shortest paths

2.1 Single source

Dijkstra’s algorithm supposes \( w(u, v) \geq 0 \), whereas Bellman-Ford’s doesn’t. Moreover Bellman-Ford’s algorithm returns false if a negative weighted cycle is found.

Algorithm 2 Dijkstra

**Require:** Graph \( G = (V, E) \), source

1: \textbf{for all} vertex \( v \in G \) do // Initializations
2: \hspace{1em} \texttt{dist}[v] \leftarrow \infty // Unknown distance function from source to \( v \)
3: \hspace{1em} \texttt{previous}[v] \leftarrow \texttt{null} // Previous node in optimal path from source
4: \hspace{1em} \texttt{dist}[source] \leftarrow 0 // Distance from source to source
5: \hspace{1em} Q \leftarrow V // All nodes in the graph are unoptimized - thus are in \( Q \)
6: \textbf{while} \( Q \neq \emptyset \) do
7: \hspace{2em} \( u \leftarrow \) vertex \( \in Q \) with smallest \texttt{dist}[{}]
8: \hspace{2em} \textbf{remove} \( u \) from \( Q \)
9: \hspace{2em} \textbf{for all} neighbor \( v \) of \( u \) \textbf{do} // where \( v \) has not yet been removed from \( Q \)
10: \hspace{3em} \texttt{alt} \leftarrow \texttt{dist}[u] + w(u, v) // be careful in 1st step - \texttt{dist}[u] is \infty yet
11: \hspace{3em} \textbf{if} \texttt{alt} < \texttt{dist}[v] \textbf{then} // Relax \((u,v,a)\)
12: \hspace{4em} \texttt{dist}[v] \leftarrow \texttt{alt}
13: \hspace{4em} \texttt{previous}[v] \leftarrow u
14: \hspace{2em} \textbf{return} \texttt{previous}[{}]

Algorithm 3 Bellman-Ford

**Require:** list vertices, list edges, vertex source

1: // This implementation takes in a graph, represented as lists of vertices
2: // and edges, and modifies the vertices so that their distance and
3: // predecessor attributes store the shortest paths.
4: 4: \textbf{Step 1: Initialize graph}
5: 5: \textbf{for all} vertex \( v \in V \) do
6: 6: \hspace{1em} \textbf{if} \( v \) is source then
7: 7: \hspace{2em} \texttt{dist}[v] \leftarrow 0
8: 8: \textbf{else}
9: 9: \hspace{1em} \texttt{dist}[v] \leftarrow \infty
10: 10: \hspace{1em} \texttt{previous}[v] \leftarrow \texttt{null}
11: //Step 2: relax edges repeatedly
12: \textbf{for all} \( i \) from 1 to \( |V| - 1 \) do
13: \hspace{1em} \textbf{for all} edge \((u, v) \in E\) do
14: \hspace{2em} \textbf{if} \texttt{dist}[v] > \texttt{dist}[u] + w(u, v) \textbf{then}
15: \hspace{3em} \texttt{dist}[v] \leftarrow \texttt{dist}[u] + w(u, v)
16: \hspace{3em} \texttt{previous}[v] \leftarrow u
17: // Step 3: check for negative-weight cycles
18: \textbf{for all} edge \((u, v) \in E\) do
19: \hspace{1em} \textbf{if} \texttt{dist}[v] > \texttt{dist}[u] + w(u, v) \textbf{then}
20: \hspace{2em} \textbf{Error} “Graph contains a negative-weight cycle”
2.2 All pairs of vertices

Assume that $w(u,v)$ is the cost of the edge from $u$ to $v$ ($\infty$ if there is none). Also assume that $n$ is the number of vertices and $w(u,u) = 0$.

Let $path$ be a 2-dimensional matrix. At each step in the algorithm, $path[u][v]$ is the shortest path from $u$ to $v$ using intermediate vertices ($1 \ldots k - 1$). Each $path[u][v]$ is initialized to $w(u,v)$ or $\infty$ if there is no edge between $u$ and $v$.

Algorithm 4 Floyd-Warshall

1: for $k = 1$ to $n$ do
2:  for $u \in 1, \ldots, n$ do
3:    for $v \in 1, \ldots, n$ do
4:      $path[u][v] \leftarrow \min(path[u][v], path[u][k] + path[k][v])$

Also see Johnson’s algorithm which is faster for sparse graphs.

3 Flood-fill

Algorithm 5 Recursive flood fill

Require: node, target_color, replacement_color

1: if the color of node is not equal to target_color then
2:   return
3: if the color of node is equal to replacement_color then
4:   return
5: Set the color of node to replacement_color
6: Perform Flood-fill (one step to the west of node, target_color, replacement_color)
7: Perform Flood-fill (one step to the east of node, target_color, replacement_color)
8: Perform Flood-fill (one step to the north of node, target_color, replacement_color)
9: Perform Flood-fill (one step to the south of node, target_color, replacement_color)
10: return

4 Minimum Spanning Tree

Find a minimum spanning tree (MST) for a connected weighted graph, i.e., a spanning tree with minimum weight.

Also see Prim’s algorithm.
Algorithm 6 Kruskal

**Require:** Graph $G$

1: **for all** vertex $v \in G$ **do**
2: Define an elementary cluster $C(v) \leftarrow \{v\}$.
3: Initialize a priority queue $Q$ to contain all edges in $G$, using the weights as keys.
4: Define a tree $T \leftarrow \emptyset$ // $T$ will ultimately contain the edges of the MST
5: // $n$ is total number of vertices
6: **while** $T$ has fewer than $n - 1$ edges **do**
7: // edge $(u, v)$ is the minimum weighted route from/to $v$
8: $(u, v) \leftarrow Q.removeMin()$
9: // prevent cycles in $T$. add $(u, v)$ only if $T$ does not already contain a path between $u$ and $v.$
10: // Note that the cluster contains more than one vertex only if an edge containing a pair of
11: // the vertices has been added to the tree.
12: Let $C(v)$ be the cluster containing $v$, and let $C(u)$ be the cluster containing $u$.
13: **if** $C(v) \neq C(u)$ **then**
14: Add edge $(v, u)$ to $T$
15: Merge $C(v)$ and $C(u)$ into one cluster, that is, $C(v) \cup C(u)$.
16: **return** tree $T$
5 Maximum Flow

The maximum flow problem is to find a feasible flow through a single-source, single-sink flow network that is maximum.

We denote by $c(u, v)$ the capacity of the link $(u, v)$ between $u$ and $v$, and $f(u, v)$ the flow on this link. We define the residual network $G_f = (V_f, E_f)$ to be the network with capacity $c_f(u, v) = c(u, v) - f(u, v)$ and no flow.

**Algorithm 7** Ford-Fulkerson

**Require:** Graph $G = (V, E)$ with flow capacity $c$, source node $s$ and sink node $t$

1: for all $(u, v) \in E$ do
2: \hspace{1em} $f(u, v) \leftarrow 0$
3: while $\exists$ path $p$ from $s$ to $t$ in $G_f$, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$ do
4: \hspace{1em} Find $c_f(p) \leftarrow \min \{ c_f(u, v) : (u, v) \in p \}$
5: \hspace{1em} for all edge $(u, v) \in p$ do
6: \hspace{2em} $f(u, v) \leftarrow f(u, v) + c_f(p)$ \hspace{1em} // Send flow along the path
7: \hspace{2em} $f(v, u) \leftarrow f(v, u) - c_f(p)$ \hspace{1em} // The flow might be “returned” later

The algorithm maintains the following invariants:

- $f(u, v) \leq c(u, v)$ the flow does not exceed the capacity
- $f(u, v) = -f(v, u)$
- $\sum_v f(u, v) = 0 \Leftrightarrow f_{in}(u) = f_{out}(u)$ for all nodes $u$, except the source $s$ and the sink $t$.

Finding a path (line 3 in Algorithm 7) can be found using a DFS or BFS.