
Algorithm 1: *GirvanNewman*($G, modBound$)

Input: Graph G – an undirected graph, Float $modBound$ – a bound for the modularity so the algorithm knows when it is complete

Output: An undirected graph composed of all vertices and some edges of G , likely with more connected components representing communities.

```
1  $G2 \leftarrow copy(G)$ ;  
2  $connectedComponents \leftarrow DetectConnectedComponents(G, G.edges)$ ;  
3 while  $Modularity(G, connectedComponents) < modBound$  do  
4    $betweennessCentrality \leftarrow CalculateEdgeBetweennessCentrality(G2)$ ;  
5    $maxEdge \leftarrow$  edge with max betweenness centrality in  $G2$ ;  
6   Remove  $maxEdge$  from  $G2$ ;  
7   foreach  $nodeSet \in connectedComponents$  do  
8     if  $maxEdge.node1 \in nodeSet$  then  
9        $currComponent \leftarrow nodeSet$ ;  
10      break;  
11    $newComponents \leftarrow DetectConnectedComponents(G, currComponent)$ ;  
12   if  $newComponents.length == 2$  then  
13     Remove  $currComponent$  from  $connectedComponents$ ;  
14     Add  $newComponents$  to  $connectedComponents$ ;  
15 return  $G2$ ;
```

Algorithm 2: *DetectConnectedComponents*($G, allNodes$)

Input: Graph G – an undirected graph, Node Set $allNodes$ – a set of nodes of G to split into sets of nodes representing connected components

Output: A list of sets of nodes in G , each set representing one connected component

```
1  $components \leftarrow$  empty list;  
2 while  $allNodes$  is not empty do  
3    $visited \leftarrow$  empty set;  
4    $stack \leftarrow$  arbitrary element removed from  $allNodes$  inserted into an empty stack;  
5   while  $stack$  is not empty do  
6      $currNode \leftarrow stack.pop()$ ;  
7     add  $currNode$  to  $visited$ ;  
8     Remove  $currNode$  from  $allNodes$ ;  
9     foreach Node  $n$  adjacent to  $currNode$  in  $G$  do  
10      if  $n$  is not in  $stack$  and  $n$  is not in  $visited$  then  
11         $stack.push(n)$ ;  
12   append  $visited$  to  $components$ ;  
13 return  $components$ ;
```

Algorithm 3: *Modularity($G, components$)*

Input: Graph G – an undirected graph, List of Sets *components* – a list of sets of nodes representing the communities to use for G in the modularity calculation

Output: Integer – the modularity of G

```
1  $sum \leftarrow 0$ ;  
2  $m \leftarrow$  number of edges in  $G$ ;  
3 foreach Node  $n1$  in  $G.nodes$  do  
4   foreach Node  $n2$  in  $G.nodes$  do  
5     if  $n1$  is in the same component as  $n2$  then  
6        $A \leftarrow 1$  if  $G$  has edge  $(n1, n2)$ ;  
7        $d1 \leftarrow$  degree of  $n1$ ;  
8        $d2 \leftarrow$  degree of  $n2$ ;  
9        $sum \leftarrow sum + A - ((d1 * d2) / (2 * m))$ ;  
10 return  $sum / (2 * m)$ ;
```

Algorithm 4: CalculateEdgeBetweennessCentrality(G)

Input: Graph G – an undirected graph
Output: Dictionary containing the the edges and their related betweenness centrality value

```
1  $EBC \leftarrow$  empty dictionary;
2 foreach Edge  $e$  in  $G.edges$  do
3    $EBC[e] \leftarrow 0$ ;
4 foreach Node  $n$  in  $G.nodes$  do
5    $distanceDict \leftarrow$  new Dictionary;
6    $distanceDict[n] \leftarrow 0$ ;
7    $queue \leftarrow$  new Queue;
8    $queue.enqueue(n)$ ;
9    $visited \leftarrow$  empty set;
10  while  $queue$  is not empty do
11     $currNode \leftarrow queue.dequeue()$ ;
12    add  $currNode$  to  $visited$ ;
13    foreach Node  $u$  adjacent to  $currNode$  in  $G$  do
14      if  $u$  not in  $visited$  and  $u$  not in  $queue$  then
15         $distanceDict[u] \leftarrow distanceDict[currNode] + 1$ ;
16         $queue.enqueue(u)$ ;
17   $numShortestPaths \leftarrow$  new Dictionary;
18  foreach Node  $u$  in  $G.nodes$  do
19     $numShortestPaths[u] \leftarrow 0$ ;
20   $numShortestPaths[n] \leftarrow 1$ ;
21   $visited \leftarrow$  empty set;
22   $distances \leftarrow$  sort list of (key, value) pairs in  $distanceDict$  by value;
23  foreach ( $key, d$ ) in  $distances$  do
24    foreach Node  $pred$  adjacent to  $key$  in  $G$  do
25      if  $distanceDict[pred] == d - 1$  then
26         $numShortestPaths[key] \leftarrow numShortestPaths[key] + numShortestPaths[pred]$ ;
27   $edgeWeights \leftarrow$  new Dictionary;
28  foreach Edge  $e$  in  $G$  do
29     $edgeWeights[e] \leftarrow 0$ ;
30  foreach ( $key, d$ ) in  $distances$  do
31     $sumIncoming \leftarrow 0$ ;
32    foreach Node  $succ$  adjacent to  $key$  in  $G$  do
33      if  $distanceDict[succ] == d + 1$  then
34         $sumIncoming \leftarrow sumIncoming + edgeWeights[(key, succ)]$ ;
35    foreach Node  $pred$  adjacent to  $key$  in  $G$  do
36      if  $distanceDict[pred] == d - 1$  then
37         $edgeWeights[(n, pred)] \leftarrow$ 
38           $(1 + sumIncoming) * (numShortestPaths[pred] / numShortestPaths[key])$ ;
39  foreach Edge  $e$  in  $edgeWeights$  do
40     $EBC[e] \leftarrow EBC[e] + edgeWeights[e]$ ;
41   $EBC[e] \leftarrow EBC[e] / 2$ ;
42 return  $EBC$ ;
```
