Caio Brighenti
COSC 302 - Analysis of Algorithms – Spring 2019
Assignment 4

1. **Articulation points**

(a) Give an algorithm to find if graph G is connected
A connected graph G is a graph for which there exists a between every pair of vertices in the graph. That is to say, every vertex is reachable from a traversal starting at any arbitrary vertex. We leverage this fact to devise a DFS-based algorithm to determine whether a graph G is connected.
Algorithm:

```
isConnected(G){
  v = any vertex in G
  seen[] = empty array
  explore(G,v)
  if length(seen) == length(G.V):
    return true
  else:
    return false
}

explore(G,v){
  visited(v) = true
  add v to seen
  for each edge(v,u)∈ G.E:
    if not visited(u):
      explore(u)
}
```

This algorithm simply explores the graph G starting at any vertex in V, and returns true if all other vertices are reachable from that arbitrary vertex. As this graph is undirected, if is enough to check one single vertex, as any path v, u in the graph is reflexive giving us path u, v. Thus, all nodes u reachable by one node v are also reachable by each of the nodes u.

(b) Give an algorithm to find all the articulation points of a graph in time $O(V(V + E))$.
We leverage the algorithm provided in the previous problem to devise an algorithm to locate articulation points. In short, this algorithm will, for each vertex, remove that vertex and check if is still a connected graph. This assumes the input G is a connected graph.
Algorithm:

```
findArticulationPoints(){
  art_points[] = empty array
  for each vertex v ∈ G.V:
    G' = G - v
    if !isConnected(G'):
      add v to art_points
  return art_points
}
```

It is left to show that this runs in $O(V(V + E))$ time. Let us first show the running time of the *isConnected*() algorithm. In the case of a connected graph, it is clear that the connected algorithm explores each edge and each vertex a single time, in the order that they are reachable from the starting vertex. Thus, the running time of this is $O(V + E) + O(1) \in O(V + E)$. The constant term comes from both the operations within explore and within isConnected, but are clearly dominated by the other term. The articulation points algorithm includes a loop running once for each vertex in G. On each iteration, the loop calls *isConnected*. Thus, the overall running time must be $O(V(V+E)) + O(1) \in O(V(V+E))$.

(c) Prove that an internal node of G is an articulation point if and only if there exists a subtree rooted at a child of u that has no back edges to an ancestor of u.

Let node u be an internal node in graph G such that node u has at least one child. Let tree T be a non-empty tree rooted at node t, which is a child of u. Let G'' be the subgraph of G that includes all nodes of G excluding u and T. We must show that u is an articulation point of g if and only if T has no back edge to an ancestor of G. We start by showing that if T does have a back edge, then u cannot be an articulation point.
Let T have a back edge to an ancestor of u. As all nodes that are not u or not $\in T$ are in G', then it must be that $\forall\ ancestors(u) \in G''$. Thus, there is a back edge from a node in T to G''. By the definition of a back edge, this means that there exists an edge (a, b) such that $a \in T \wedge b \in G''$. Also by the definition of a back edge, we know that this is a direct edge, and not a path passing through u. Thus, if all subtrees rooted at a child of u include a back edge to G'', u *cannot* be an articulation point of G as removing it would not disconnect its subtrees from the rest of the graph. It remains to be shown that if u is an articulation point, its subtrees must not have back edges to any ancestors of u.
We proceed by contradiction. Let G, G'', T, t and u be defined as above, with the added constraint that u is an articulation point. Therefore, removing u from the graph G would make the graph disconnected, making any children of u unreachable from G'', including tree T. However, as tree T includes a back edge to G', there must be an edge connecting a node in T to G''. As tree T is rooted at a child of u, then all nodes in T are descendants of u, and must be disconnected from G''. Thus, we have a contradiction, and have shown that if u is an articulation point, all subtrees rooted at a child of u must not have back edges to an ancestor of u. This, in combination with our previous conclusion, allows us to show that node u is an articulation point if and only if all subtrees rooted at a child of u do not have back edges to an ancestor of u.

(d) Prove that the root of G' is an articulation point if and only if it has at least two children in G'.

As this is a two-sided implication, we must show each separately. First, we show that if the root of G' is an articulation point, it must have at least two children.
Let u be the root of G', and let u be an articulation point of G. Thus, removing u from G should disconnect G, making it so that path no longer exists between every node. Let u have a single child, v. We proceed by contradiction.
As G is connected, then there exists a path in G to all nodes from u. As u has only a single child, and is the root of the DFS, then there exists a path from u to all other nodes which passes through v. By implication, this means that there exists a path form v to all other nodes in G which do not pass through u. Thus, removing u does not make the tree unconnected, as all nodes are still reachable. This is impossible, as u is an articulation point, and thus we have a contradiction. This would not be the case if u had multiple children, as not all paths from u to each other node would necessarily pass through v. With this, we conclude that u being an articulation point of G $\implies$ u must have more than a single child.
Next, we show that if the root of G', u, has at least two children, it must be an articulation point. We proceed by direct proof based on the definition of DFS. Let T_1 and T_2 be two trees rooted at children of u. Assume that T_1 is rooted at the first child, t_1, of u explored by DFS. Necessarily, T_1 will include all nodes accessible by t_1, and consequently by any descendants of t_1. For u to have a second tree T_2, then it must be that no elements of T_2 are acessible from T_1. If this was not the case, then the nodes of T_2 would be have been explored as part of the DFS through T_1. Thus, all paths between T_1 and T_2 *must* pass through u, which means removing u would leave us with two unconnected components. Thus, if the root of G' has at least two children, then it must be an articulation point.
We can therefore come to the overall conclusion that the root of G' is an articulation point $\iff$ it has at least two children.

(e) Give a biconditional statement for leaves of G' to be an articulation point of G.

It is definitionally impossible for a leaf of G' to be an articulation point of G. In a DFS graph, a leaf is a node from which no new nodes can be explored that have not yet been visited by the search. Consequently, this means that there are no nodes that can only be reached through a path passing through that leaf. Therefore, a articulation point in G *cannot* ever be a leaf in G'. We write this in biconditional form as follows. A node $u \in G$ is a leaf in G' and an articulation point in $G \iff \mathit{false}$, as the left side cannot ever be true.

(f) Give a better algorithm for finding articulation points of a connected graph using the biconditional statements in the above problems.

We leverage the conclusions above to devise an algorithm that checks for articulation points in G as it builds the DFS tree G'. This algorithm keeps track of the node with the lowest pre value that can be reached by a direct edge in the subtree of each node.
Algorithm:

```
find_art_points_DFS(G){
  // initialize data
  for vertex v in G.V:
    parent(v) = null
    low(v) = null
    art_points[] = empty array

  // DFS starting at any vertex
  root = any v in G
  explore(r,G)

  // root case
  if root has 2 or more children:
    add root to art_points

  return art_points
}

explore(v,G){
  // mark as visited and set low
  visited(v)=true
  previsit(v)
  low(v) = prev[v]

  // loop through children
  for each edge (v,u) ∈ G.E:
    if not visited(u):
      parent(u) = v
      explore(u,G)
      // if child u has path to back edge, so does v
      low(v) = min(low(u),low(v))
      // check if articulation point
      if low(u) ≥ low(v):
        add v to art_points
    else if u != parent(v) \\ must be back edge
      low(v)=min(low(v),prev[u])
}
```

This algorithm relies on the biconditional statements for a root or internal node of G' being an

articulation point. The root case is checked on line 13-14. As we have shown, if the root has two or more children, it is an articulation point. This is precisely what we check. For the internal node, we have shown that if no trees rooted at a child of internal node u have a back edge to an ancestor of u, then u must be an articulation point. This is checked by maintaining the $low()$ data structure. For each node u, $low(u)$ holds the lowest *prev* number reachable by a direct edge from a subtree rooted at u. This allows us to check easily whether a child of u reaches an ancestor of u, which we do on lines 33-34. It is unnecessary to check the leaf case, as leafs cannot be articulation points.
It is left to show the running time of this algorithm. This algorithm has two main loops. Firstly, the loop on lines 3-6 that initializes the data structures. This has $O(V)$ time. Next, we have the running time of *explore*. As was the case before, *explore* visits each vertex and each edge once, though we have some added operations on each execution. Fortunately, these operations are in constant time. Therefore, the running time of explore is unchanged and is $O(V + E)$. The overall running time of this algorithm is $O(V) + O(V + E) \in O(V + E)$, as $V + E \geq V$.

2. Give an algorithm for topological sort other than what you studied

   Instead of using DFS, we devise an algorithm that leverages a linked-list implementation of a directed graph. In this implementation, we assume that the input G is an array of linked lists, one for each node containing the children of that node. For instance, $G[u]$ returns a linked list containing all children v of u.
   Algorithm:

```
topo_sort(G){
  order[] = empty array
  q = new queue

  for node u in G:
    num_parents(u) = 0

  for node u in G:
    for node v in G[v]:
      num_parents(v)++

  for node u in G:
    if num_parents(u) == 0
      enqueue(q,u)
      break

  while q not empty:
    u = dequeue(q)
    add u to order
    for each node v in G[u]:
      num_parents(v)--
      if num_parents(v) == 0:
        enqueue(q,v)

  return order
}
```

   The principle behind this algorithm is simple. A node cannot be added to the topological sort before any of its parents. Therefore, at each point we maintain the number of parents yet to be added to the ordering for each node. Any node that has 0 parents not yet in the order can be added. When a node is added is necessary to the order, we then update each of its' children to decrease the number of parents yet to be added.

In terms of running time, this algorithm at first glance appears to be highly inefficient given the number of for loops, some of which are nested. However, we will show that this algorithm actually runs in $O(V+E)$ time by calculating the running time of each component. The first two lines (2-3) are clearly constant time, thus $O(1)$. The loop on lines 5-6 iterates once for each node in G, thus it is clearly $O(V)$.
The second loop, on lines 8-10, is actually a nested for loop. However, it is clear that this iterates once per node, and each node visits the corresponding children. Thus, we can think of this as visiting each node in V, and each edge in E. The running time of this nested loop is $O(V+E)$.
The third loop, on lines 12-15, is similar to the first. It loops until it finds the first vertex in G that has no parents . While in the best case this is constant, in the worst the loop runs through until it reaches the last node in G. Thus, it is $O(V)$ overall.
The while loop on lines 17-23 is the most complicated. Let us first consider the parent loop, then the inner loop. As the while loop iterates until the queue is empty, and each node gets added to the queue once, then it must execute once per node. Thus the while loop is $O(V)$. The running time of the inner loop depends on the number of children of the current node. However, it should be clear that in total across all executions of the while loop, the inner loop will traverse each edge once. Thus, the overall nested loop is the same as loop 2 as its running time is$O(V+E)$, although with a higher constant factor.
We can thus calculate the overall running time for the algorithm by adding each of the components. We then have $O(1)+O(V)+O(V+E)+O(V)+O(V+E) = O(1)+2O(V)+2O(V+E) \in O(V+E)$, as the $O(V+E)$ term dominates and we can ignore the constant factors.

3. Modify BFS so that it can determine if a given undirected graph is bipartite in linear time. Analyze it!

   A graph is bipartite if it can be split into two disjoint sets V_1 and V_2 such that for any two nodes $u, v \in V_1 \vee V_2$, then there does not exist an edge between u and v. We can rephrase this in by stating that for any node $u \in G$, the neighbors of u must be in the opposite set. We leverage this fact to implement a BFS algorithm that assigns each visited node to a group, and checks whether any two neighbors are in the same group. If so, the graph cannot be bipartite. We assume that the graph G is implemented in linked-list form.
   Algorithm:

```
bipartite_BFS(G){
  for all vertex v in G:
    visited(v) = false

  for all v in G:
    if visited(v) == false
      V1(v) = true
      is_bipartite = explore(G,v)

      if is_bipartite == false:
        return false

  return true
}

explore(G,v){
  q = new queue
  enqueue(q,v)
  visited(v) = true

  while q not empty:
    s = dequeue(q)
23      for vertex u in G[s]:
24        if visited(u) == false:
25          V1(u)=!V1(s)
26          enqueue(q,u)
27          visited(u) = true
28        else:
29          if V1(s) == V1(u):
30            return false
31
32    return true
33  }
```

Explanation:
For the most part, this algorithm is a simple BFS algorithm. The *bipartiteBFS* function first marks all nodes as unvisited, then proceeds to call *explore* on each unvisited node until all connected components explored. However, *explore* will check and return whether the current component is bipartite. If it is not, the overall algorithm returns negative. If all calls of explore reach the end, and return true, then *bipartiteBFS* will return true, based on the conclusion that it has no components that are *not* bipartite, and thus G is bipartite.
Let us discuss the functionality of *explore*. The function initializes a queue starting at the node passed to the function, and iterates while the queue is not empty. At each iteration, the algorithm explores all neighbors of the current node, adding them to the queue. Additionally, each node is assigned to either group V_1 or V_2. This is done by setting $V1(v)$ to true or false for each node. When *explore* is first called, the node passed is set to belong to group V_1. From there, on each iteration of the while loop, each unvisited neighbor of the current node is set to belong to the opposite group. If a neighbor has already been visited, then it must already belong to a group. Here, the algorithm checks whether this visited neighbor belongs to the same group as the current node. If so, then the graph cannot be bipartite, and thus the algorithm returns false.

Running time:
We calculate the overall running time of this algorithm by calculating each of its component pieces individually, and then adding them together. We start with the *explore* function. This function includes several constant time operations, followed by a while loop with a nested for loop. The while loop will execute until the queue is emptied, and as each element in the connected component is added to the queue once, then the while loop will run V' times, where V' is the number of vertices in the component. The number of iterations of the inner loop depends on the current vertex, as it iterates through all neighbors of that vertex. While this would usually lead to the conclusion that the overall loop is $O(V' + E')$, iterating throgh each vertex and each edge once, this situation is slightly different. This is because perform operations for *all* neighbors of each vertex, not only the unvisited ones. Thus, each edge u, v is considered twice, once for node u and once for node v. Thus, the running time the loop is $O(V' + 2E')$. This is, however, simply a difference of a constant factor, making our overall running time for *explore* $O(1) + O(V' + 2E') \in O(V' + E')$.
Next we consider the overall running time of *bipartiteBFS*. As we have seen, *explore* will run in $O(V' + E')$, where V' and E' are the vertices and edges respectively of the component passed to *explore*. These calls to explore, while each different, can be elegantly combined given the loop on lines 5-11 simply calls *explore* once for each connected component. Thus, the running time for this loop is $O(1) + O(V + E) \in O(V + E)$, as each component's edges and vertices are disjoint and add up to compose the entire graph's vertices and edges. Finally, we consider the loop on lines 2-3. This loop, running once per each vertex in G, is clearly $O(V)$. Thus, the overall running time for our *bipartiteBFS* algorithm is $O(V) + O(V + E) \in O(V + E)$, as the second term dominates.