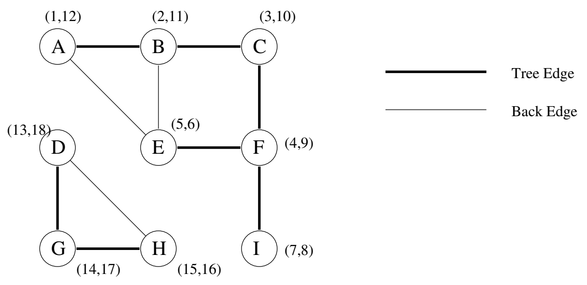
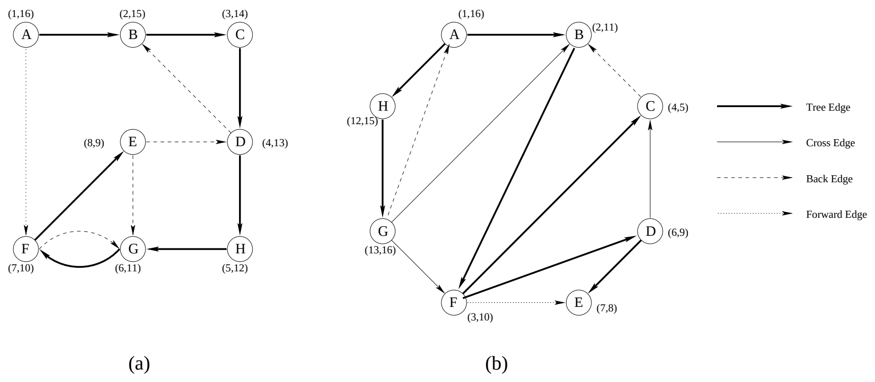
Chapter 3 – Solutions

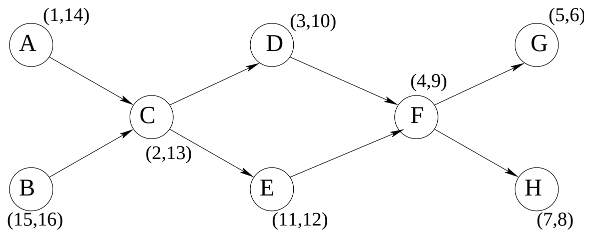
3.1 The figure below gives the `pre` and `post` numbers of the vertices in parentheses. The tree and back edges are marked as indicated.



3.2 The figure below shows `pre` and `post` numbers for the vertices in parentheses. Different edges are marked as indicated.



3.3 (a) The figure below shows the `pre` and `post` times in parentheses.



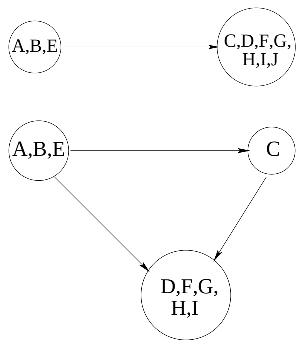
(b) The vertices A, B are sources and G, H are sinks.

(c) Since the algorithm outputs vertices in decreasing order of `post` numbers, the ordering given is B, A, C, E, D, F, H, G.

(d) Any ordering of the graph must be of the form $\{A, B\}, C, \{D, E\}, F, \{G, H\}$, where $\{A, B\}$ indicates A and B may be in any order within these two places. Hence the total number of orderings is $2^3 = 8$.

3.4 (i) The strongly connected component found first is $\{C, D, F, G, H, I, J\}$ followed by $\{A, B, E\}$. $\{C, D, F, G, H, I, J\}$ is a source SCC, while $\{A, B, E\}$ is a sink SCC. The metagraph is shown in the figure below. It is easy to see that adding 1 edge from any vertex in the sink SCC to a vertex in the source SCC makes the graph strongly connected.

(ii) The strongly connected components are found in the order $\{D, F, G, H, I\}$, $\{C\}$, $\{A, B, E\}$. $\{A, B, E\}$ is a source SCC, while $\{D, F, G, H, I\}$ is a sink. Also, in this case adding one edge from any vertex in the sink SCC to any vertex in the source SCC makes the metagraph strongly connected and hence the given graph also becomes strongly connected.



3.5 Create a new (empty) adjacency list for the reverse. We go through the list of G and if in the neighborhood of u, we find the vertex v, add u in the list of v in G^R as the *first* element in the list. Note that it would have been difficult to insert u at the end of the list, but insertion in the first position takes only $O(1)$ time.

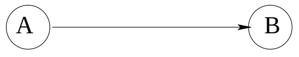
3.6 (a) Note that each edge (u, v) contributes 1 to $d(u)$ and 1 to $d(v)$. Hence, each edge contributes exactly 2 to the sum $\sum_{v \in V} d(v)$, which gives $\sum_{v \in V} d(v) = 2|E|$.

(b) Let V_o be the set of vertices with odd degree and V_e be the set of vertices with even degree. Then

$$\sum_{v \in V_o} d(v) + \sum_{v \in V_e} d(v) = 2|E| \implies \sum_{v \in V_o} d(v) = 2|E| - \sum_{v \in V_e} d(v)$$

The RHS of this equation is even and the LHS is a sum of odd numbers. A sum of odd numbers can be even, only if it is the sum of an *even number of odd numbers*. Hence, the number of vertices in V_o (equal to the number of people with odd number of handshakes) must be even.

(c) No. The following graph provides a counter-example as only one vertex (B) has odd indegree.



3.7 (a) Let us identify the sets V_1 and V_2 with the colors red and blue. We perform a DFS on the graph and color alternate levels of the DFS tree as red and blue (clearly they must have different colors). Then the graph is bipartite iff there is no monochromatic edge. This can be checked during DFS itself as such an edge must be a back-edge, since tree edges are never monochromatic by construction and DFS on undirected graphs produces only tree and back edges.

(b) The "only if" part is trivial since an odd cycle cannot be colored by two colors. To prove the "if" direction, consider the run of the above algorithm on a graph which is not bipartite. Let u and v be two vertices such that (u, v) is a monochromatic back-edge and u is an ancestor of v. The path length from u to v in the tree must be even, since they have the same color. This path, along with the back-edge, gives an odd cycle.

(c) If a graph has exactly one odd cycle, it can be colored by 3 colors. To obtain a 3-coloring, delete one edge from the odd cycle. The resulting graph has no odd cycles and can be 2-colored. We now add back the deleted edge and assign a new (third) color to one of its end points.

3.8 (a) Let $G = (V, E)$ be our (directed) graph. We will model the set of nodes as triples of numbers (a_0, a_1, a_2) where the following relationships hold: Let $S_0 = 10$, $S_1 = 7$,$S_2 = 4$ be the sizes of the corresponding containers. a_i will correspond at the actual contents of the i^{th} container. It must hold $0 \leq a_i \leq S_i$ for $i = 0, 1, 2$ and at any given node $a_0 + a_1 + a_2 = 11$ (the total amount of water we started from). An edge between two nodes (a_0, a_1, a_2) and (b_0, b_1, b_2) exists if both the following are satisfied :

– the two nodes differ in exactly two coordinates (and the third one is the same in both).

  - if i, j are the coordinates they differ in, then either $a_i = 0$ or $a_j = 0$ or $a_i = S_i$ or $a_j = S_j$.

  The question that needs to be answered is wether there exists a path between the nodes $(0, 7, 4)$ and $(*, 2, *)$ or $(*, *, 2)$ where $*$ stands for any (allowed) value of the corresponding coordinate.

(b) Given the above description, it is easy to see that a DFS algorithm on that graph should be applied, starting from node $(0, 7, 4)$ with an extra line of code that halts and answers 'YES' if one of the desired nodes is reached and 'NO' if all the connected component of the starting node is exhausted an no desired vertex is reached.

(c) It is easy to see that after a few steps of the algorithm (depth 6 on the dfs tree) the node (2,7,2) is reached, so we answer 'YES'.

3.9 First, the degree of each node can be determined by counting the number of elements in its adjacency list. The array `twodegree` can then be computed by initializing `twodegree` to 0 for every vertex, and then modifying `explore` as follows to add the degree of each neighbor of a vertex u to `twodegree`$[u]$.

```
explore($G, u$) {
    visited($u$) = true
    previsit(u)
    for each edge $(u, v) \in E$:
        twodegree[$u$] = twodegree[$u$] + degree[$v$]
        if not visited($v$):  explore($v$)
    postvisit($u$)
}
```

3.10 We use an extra variable `top` which refers to the top element in the stack. We assume that `top` is automatically updated when some element is pushed or popped. Also, we need to keep track of how many neighbors of a vertex have been already checked. For this we also need the outdegree of every vertex (which is part of the adjacency list). We assume that `neighbor`(v, i) gives the ith neighbor of the vertex v[1].

The code for the `explore` function is as follows:

```
explore($G, u$) {
    visited($u$) = true
    previsit(u)
    neighbors_checked($u$) = 0
    push $u$
    while stack is not empty {
        $w$ = top
        for $i$ from neighbors_checked[$w$]+1 to outdegree[$w$] {
            $v$ = neighbor($w$, $i$)
            if not visited($v$)
                neighbors_checked[$w$] = $i$
                push($v$)
                break
        }
        if neighbors_checked[$w$] = outdegree[$w$]
            pop($w$)
            postvisit($w$)
    }
}
```

3.11 Let $e = (u, v)$. The graph has a cycle containing e if and only if u and v are in the same connected component in the graph obtained by deleting e. This can be easily checked by a DFS on this graph.

[1]We have a list and cannot actually access the ith neighbor, but i changes sequentially in the loop here and hence we allow ourselves some abuse of notation.

3.12 There are two cases possible: $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$ or $\texttt{pre}(v) < \texttt{post}(v) < \texttt{pre}(u) < \texttt{post}(u)$. In the first case, u is an ancestor of v. In the second case, v was popped off the stack without looking at u. However, since there is an edge between them and we look at all neighbors of v, this cannot happen. So, the given statement is true.

3.13 (a) Consider the DFS tree of G starting at any vertex. If we remove a leaf (say v) from this tree, we still get a tree which is a connected subgraph of the graph obtained by removing v. Hence, the graph remains connected on removing v.

(b) A directed cycle. Removing any vertex from a cycle leaves a path which is not strongly connected.

(c) A graph consisting of two disjoint cycles. Each cycle is individually a strongly connected component. However, adding just one edge is not enough as it (at most) allows us to go from one component to another but not back.

3.14 The algorithm is to always pick a source, delete it from the graph and then recurse on the resulting graph. A source is just a vertex with indegree 0. Thus, we can find all the sources in the initial graph by performing a DFS and computing the indegree of all the vertices. We add all the vertices with indegree 0 to a list L.

At each step, the algorithm then removes an element from L (a source) and reduces the indegree of each of its neighbors by 1 (this corresponds to deleting it from the graph). If this changes the indegree of any vertex to 0, we add it to L. The removed element is assigned the next position in the ordering.

Computing all indegrees in the first step only requires a DFS which takes linear time. Subsequently, we only look at each edge (u, v) at most once when we are removing u from L. Hence, the total time is linear in the size of the graph.

3.15 (a) We view the intersections as vertices of a graph with the streets being directed edges, since they are one-way. Then the claim is equivalent to saying that this graph is strongly connected. This is true iff the graph has only one strongly connected component, which can be checked in linear time.

(b) The claim says that starting from the town hall, one cannot get to any other SCC in the graph. This is equivalent to saying that the SCC containing the vertex corresponding to the town hall is a sink component. This can be easily done in linear time by first finding the components, and then running another DFS from the vertex corresponding to the town hall, to check if any edges go out of the component[2].

3.16 The graph of the prerequisite relation must be a DAG, since there can be no circular dependencies between courses. The number of courses essentially corresponds to the length of the longest path in this DAG.

We first linearize this graph to obtain an ordering $c_1, c_2, \ldots, c_n$ of the courses, such that c_i is a possibly a prerequisite only for c_j with $j > i$. Any path ending at c_j can now only pass through c_i for $i < j$. If l_j is the length of the longest path ending at c_j, then

$$l_j = 1 + \max_{(i,j)\in E} \{l_i\}$$

which depends only on l_i for $i < j$. The required solution is $\max_i\{l_i\}$. This is essentially a use of dynamic programming.

Note however, that here we need all edges coming *into* c_j rather going out, which is what the adjacency list stores. This can be handled by computing the reverse of the graph (see problem 3.5 - in fact, it is even easier for a linearized DAG) or by modifying the algorithm above so that each c_i "updates" the maximum value at its neighbors, when it computes its own l_i value.

[2]In fact, it can even be done *while* decomposing the graph into SCCs by noting that the algorithm for decomposing progressively removes sinks from the graph at every stage. A component found by the algorithm is a sink if and only if there are no edges going out of the component into any component found *before* it.

3.17 (a) For the sake of contradiction, assume that there are two different vertices $u, v \in Inf(p)$ such that u and v belong to different strongly connected components, say C_1 and C_2 respectively. But since u occurs both before and after v in the trace, there must be a path from u to v and also a path from v to u. This would imply that there is a path from every vertex in C_1 to every vertex in C_2 and vice-versa, which is a contradiction.

(b) The argument in the previous part shows that any infinite trace must be a subset of a strongly connected component. It is also easy to see that any strongly connected component of size greater than 1 has an infinite trace since we can always pick two vertices in the same component and go from one to another infinitely often.

However, a graph that has all SCCs of size 1 must be a DAG and hence the problem reduces to checking if the given directed graph has a cycle. This can be done using DFS since the graph has a cycle if and only if DFS finds a back edge.

(c) Let $v \in Inf(p)$ be a good vertex visited infinitely often by the trace p. Also, there must be at least one more vertex in the component of v for the trace to be infinite. Hence, the problem reduces to checking if the graph contains a strongly connected component of size more than 1, which contains a good vertex. This can be done by decomposing the graph into its SCCs.

(d) Let p be a trace such that $Inf(p) \subseteq V_G$. Then there must exist a number N such that $v_n \in V_G$ for all $n \geq N$ (since no bad vertex is visited infinitely often). Then $Inf(p)$ must itself form a strongly connected subgraph since after time N, the trace does not take any path passing through a bad vertex. Hence, a graph contains an infinite trace which visits only good vertices infinitely often if and only if the subgraph induced by V_G contains a strongly connected component of size greater than 1 (We argued only the "only if" part above, but the other direction is trivial). This can be checked by proceeding as in part (b) with the subgraph induced by V_G.

3.18 Do a DFS on the tree starting from r and store the previsit and postvisit times for each node. Since the given graph is a tree, and we started at the root, the DFS tree is the same as the given tree. Thus, u is an ancestor of v if and only if $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$.

3.19 We modify the explore procedure so that explore called on a node returns the maximum `x` value in the corresponding subtree. The parent stores this as its `z` value, and returns the maximum of this and its own `x` value.

```
explore($G, u$) {
     visited($u$) = true
     z($u$) = $-\infty$
     temp = 0
     for each edge $(u, v) \in E$:
         If not visited($v$): {
             temp = explore($G, v$)
             if temp $>$ z($u$): z($u$) = temp
         }
     postvisit($u$)
     return max{z($u$), x($u$)}
}
```

3.20 We maintain the labels of all the vertices currently on the stack, in a a separate array. Since a path can have at most n vertices, the length of this array is at most n. The labels are modified using this array in the `previsit` and `postvisit` procedures.

```
– previsit($v$) {
       current_depth = current_depth + 1
       labels[current_depth] = $l(v)$
  }
– postvisit($v$){
       ancestor_depth = max{0, current_depth - $l(v)$}
```

```
        $l(v)$ = labels[ancestor_depth]
        current_depth = current_depth - 1
    }
```

ancestor_depth identifies the level at which $p^{l(v)}$ is present in the path and stores the appropriate label in the current node. Since we add only a constant number of operations at each step of DFS, the algorithm is still linear time.

3.21 Consider the case of a strongly connected graph first. The case of a general graph can be handled by breaking it into its strongly connected components, since a cycle can only be present in a single SCC. We proceed by coloring alternate levels of the DFS tree as red and blue. We claim that the graph has an odd cycle if and only if there is an edge between two vertices of the same color (which can be checked in linear time).

If there is an odd cycle, it cannot be two colored and hence there must be a monochromatic edge. For the other direction, let u and v be two vertices having the same color and let (u, v) be an edge. Also, let w be their lowest common ancestor in the tree. Since u and v have the same color, the distances from w to u and v are either both odd or both even. This gives two paths p_1 and p_2 from w to v, one through u and one not passing through u, one of which is odd and the other is even.

Since the graph is strongly connected, there must also be a path q from v to w. Since the length of this path is either odd or even, q along with one of p_1 and p_2 will give an odd length tour (a cycle which might visit a vertex multiple times) passing through both v and w. Starting from v, we progressively break the tour into cycles whenever it intersects itself. Since the length of the tour is odd, one of these cycles must have odd length (as the sum of their lengths is the length of the tour).

3.22 Let us call a vertex from which all other vertices are reachable, a *vista* vertex. If the graph has a vista vertex, then it must have only one source SCC (since two source SCCs are not reachable from each other), which must contain the vista vertex (if it's in any other SCC, there is no path from the vista vertex to the source SCC). Moreover, in this case *every* vertex in the source SCC will be a vista vertex.

The algorithm is then simply to a DFS starting from any node and mark the vertex with the highest post value. This must be in a source SCC. We now again run a DFS from this vertex to check if we can reach all nodes. Since the algorithm just uses decomposition into SCCs and DFS, the running time is linear.

3.23 We start by linearizing the DAG. Any path from s to t can only pass through vertices between s and t in the linearized order and hence we can ignore the other vertices.

Let $s = v_0, v_1, \ldots, v_k = t$ be the vertices from s to t in the linearized order. For each i, we count the number of paths from s to v_i as n_i. Each path to a vertex i and an edge (i, j), gives a path the vertex j and hence

$$n_j = \sum_{(i,j)\in E} n_i$$

Since $i < j$ for all $(i, j) \in E$, this can be computed in increasing order of j. The required answer is n_k.

3.24 Start by linearizing the DAG. Since the edges can only go in the increasing direction in the linearized order, and the required path must touch all the vertices, we simply check if the DAG has an edge $(i, i+1)$ for every pair of consecutive vertices labelled i and $i + 1$ in the linearized order. Both, linearization and checking outgoing edges from every vertex, take linear time and hence the total running time is linear.

3.25 Start by linearizing the DAG. Let $v_1, \ldots v_n$ be the linearized order. Then the following algorithm finds the cost array in linear time.

```
find_costs() {
    for $i = n$ to 1:
        cost[$v_i$] = $p_{v_i}$
        for all $(v_i, v_j) \in E$:
```

```
        if cost[$v_j$] < cost[$v_i$]:
            cost[$v_i$] = cost[$v_j$]
}
```

The time for linearizing a DAG is linear. For the above procedure, we visit each edge at most once and hence the time is linear. For a general graph, the `cost` value of any two nodes in the same strongly connected component will be the same since both are reachable from each other. Hence, it is sufficient to run the above algorithm on the DAG of the strongly connected components of the graph. For a node corresponding to component C, we take $p_C = \min_{u \in C}\{p_u\}$.

3.26 (a) We first prove the "only if" direction. Suppose we have an Eulerian tour for a graph G. Let u be any vertex in the graph. Suppose that we "enter" u k times during the tour. Since it is a cycle, we must also leave u exactly k times and all these edges must be distinct. Hence, the degree of u must be $2k$ which is even. Since this is true for any vertex u, the claim follows.

For the other direction we use induction on the number of vertices in the graph. First note that if $|V| = 2$, the trivially if the degree of both the vertices is even then the graph has an Eulerian tour. Let the statement be true for all graphs with $|V| = n$.

We consider a graph G on $n + 1$ vertices such that all its vertices have even degrees. Let u be a vertex in this graph having neighbors $i_1, i_2, \ldots i_{2k}$. Consider a graph G' where we remove u and add edges (i_1, i_2), $(i_3, i_4), \ldots, (i_{2k-1}, i_{2k})$ to G. Since G' has n vertices and the degree of each vertex is the same as in G (and thus even), G' must have an Eulerian tour. Replace every occurence of the extra edges of the form (i_{t-1}, i_t) that we inserted, by (i_{t-1}, u) followed by (u, i_t). This gives an Eulerian tour of G.

(b) To have an Eulerian path, exactly two of the vertices in the graph must have odd degree, while all the remaining ones must have even degree.

(c) A directed graph has an Eulerian tour iff the number of incoming edges at every vertex is equal to the number of outgoing edges.

3.27 By problem 3.6, we know that the number of vertices with an odd degree in an undirected graph, must be even. Suppose the number of such vertices is $2k$. We arbitrarily pair up these vertices and add an edge between each pair so that all vertices now have even degree.

By problem 3.26, each connected component of this new graph must have an Eulerian tour. Removing the k edges we added from this set of tours, breaks it into k paths with the two ends of each path being vertices of odd degree. Furthermore, all these paths are edge-disjoint, since an Eulerian tour uses each edge exactly once. Thus, taking the two ends of each path as a pair gives the required pairing.

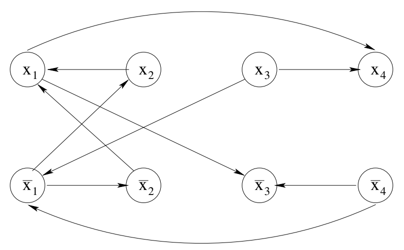
3.28 a) The formula has two satisfying assignments:

$$(x_1, x_2, x_3, x_4) = (\texttt{true}, \texttt{false}, \texttt{false}, \texttt{true})$$

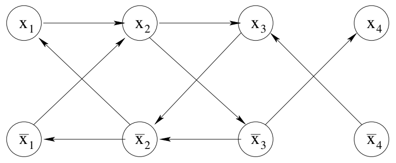
$$(x_1, x_2, x_3, x_4) = (\texttt{true}, \texttt{true}, \texttt{false}, \texttt{true})$$

b) An example is $(\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee \bar{x}_2) \wedge (x_1 \vee x_2) \wedge (x_3 \vee x_4)$

c) The graphs for the given formula and the example in part b) are given below.



d) Notice that a path in the directed graph from x to y means that $x \implies y$. If x and $\bar{x}$ are in the same storngly connected componenent, then we have $x \implies \bar{x}$ and $\bar{x} \implies x$. Then there is no way to assign a value to x to satisfy both these implications.



e) We recursively find sinks in the graph of strongly connected components and assign `true` to all the *literals* in the sink component (this means that if a sink component contains the literal $\bar{x}_2$, then we assign $\bar{x}_2 = \texttt{true} \equiv x_2 = \texttt{false}$). We will then remove all the variables which have been assigned a value from the graph.

When we set the literals in a sink component to `true`, we set their negations to `false`. However, by the symmetry of the way we assigned edges, if we reverse the directions of all the edges and replace each literal x by $\bar{x}$, the graph should remain unchanged. Hence, the negations of all the variables in a sink component *must form a source component.*

Now, we claim that we did not violate any implication while doing this. Any implication of the form $l_1 \implies l_2$ is violated only when $l_1 = \texttt{true}$ and $l_2 = \texttt{false}$. All the literals in the sink are `true` so this never happens. All the literals in the source are set to `false`, so no edge going out of them can be violated. We can now recurse on the remaining graph. Since the values assigned at each stage are consistent, we end up with a satisfying assignment.

f) To perform the operations in the previous part, we only need to construct this graph from the formula, find its strongly connected components and identify the sinks - all of which can be done in linear time.

3.29 We represent all elements of the set S as a graph with an edge from x to y if $(x, y) \in R$. Since the relation is symmetric, the corresponding graph is undirected. We claim that the connected components of the graph give the required partition of S into groups $S_1, S_2, \ldots, S_k$.

Since there are no edges between different connected components, elements in different groups are definitely not related and it only remains to check that all elements in the same component are related to each other. However, this follows directly from transitivity since there is a path between any two elements in the same component.

3.30 The relation is symmetric by definition. It is also reflexive since there is a path from every vertex u to itself of length 0. To verify transitivity, if $(u, v) \in R$ and $(v, w) \in R$, then there is a path from u to v and v to w, which gives a path from u to w (this might not be a simple path, but we can remove any cycles). Similarly, we get a path from w to u.

By problem 3.29, this partitions the graph into disjoint groups such that for every two vertices u and v in the same group, there is a path from u to v and v to u. Also, this is not true for two vertices not in the same group. However, this is precisely the definition of the strongly connected components.

3.31 (a) The relation is reflexive and symmetric by definition. To verify transitivity, let $(e_1, e_2) \in R$, both being contained in a cycle C_1 and $(e_2, e_3) \in R$, contained in cycle C_2. Then, if either of C_1 and C_2 contains all three of e_1, e_2 and e_3, we are done as then $(e_1, e_3) \in R$.

If not, then neither of e_1 and e_3 can be common to both the cycles. Also, C_1 and C_2 must share some edges. Starting from e_3, we move in both directions on C_2, until we reach a vertex in C_1. This gives a (possibly small) path P in C_2, which contains e_3 and has both endpoints (say v_1 and v_2) in C_1. Now, C_1 has two simple paths between v_1 and v_2, exactly one of which contains e_1. Removing the other gives a simple cycle containing both e_1 and e_3. Hence $(e_1, e_3) \in R$.

(b) The biconnected components in the given graph are $\{AB, BN, NO, OA\}$, $\{BD\}$, $\{CD\}$, $\{DM\}$ and $\{LK, KJ, JL, KH, IJ, IF, FG, GH, HI, FH\}$. The bridges are BD, CD and DM and the separating vertices are B, D and L.

(c) We first argue that two biconnected components must share at most one vertex. For the sake of contradiction, assume that two components C_1 and C_2 share two vertices u and v. Note that there must be a path from u to v in both C_1 and C_2, since in each component, there is a simple cycle containing one edge incident on u and one edge incident on v. The union of these two paths gives a cycle containing some edges from C_1 and some from C_2. However, this is contradiction as this would imply that an edge in C_1 is related to an edge in C_2.

We now need to prove that if two biconnected components intersect in exactly one vertex, then it must be a separating vertex. Let the common vertex be u. Let (u, v_1) and (u, v_2) be the edges corresponding to u in the two components. Then we claim that removing u disconnects v_1 and v_2. If not, then there must be a path between v_1 and v_2, which does not pass through u. However, this path, thogether with (u, v_1) and (u, v_2), gives a simple cycle containing one edge from each component which is a contradiction.

(d) The graph can in fact, be a forest. However, there cannot be a cycle as this would imply a cycle involving edges from two different biconnected components, which cannot happen since edges in different equivalence classes cannot be related.

(e) If the root has only one child, then it is effectively a leaf and removing the root still leaves the tree connected. The DFS from the first child explores every vertex reachable through a path not passing through the root. Also, since the graph is undirected, there can be no edges from the subtree of the first child to that of any other child. Hence, removing the root disconnects the tree if it has more than one children.

(f) If there is a backedge from a descendant of every child v' to an ancestor of v, each child can reach the entire tree above v the graph is still connected after removing v. If there is a child v' such that none of its descendants have a backedge to an ancestor of v, then in the graph after removing v, there is no path between an ancestor of v and v' (note that there cannot be any cross edges since the graph is undirected).

(g) While exploring each vertex u, we look at all the edges of the form (u, v) and can store at u, the lowest `pre(`v`)` value for all neighbors of u. `low(`u`)` is then given by the minimum of this value, `pre(`u`)` and the `low` values of all the children of u. Since each child can pass its `low` value to the parent when it's popped off the stack, the entire array can be computed in a single pass of DFS.

(h) A non-root node u is a separating vertex iff `pre`$(u) <$ `low`(v) for *any* child v of u. This can be checked while computing the array. Also, if u is a separating vertex and v is a child such that `pre`$(u) <$ `low`(v), then the entire subtree with v as the root must be in different biconnected components than the ancestors or other children of u. However, this subtree itself may have many biconnected components as it might have other separating vertices.

Hence, we perform a DFS pushing all the edges we see on a stack. Also, when we explore a child v of a separating vertex u such that the above condition is met, we push an extra "mark" on the stack (to mark the subtree rooted at v). When DFS returns to v, i.e. when v is popped off the stack (of vertices), we can also pop the subtree of v from the stack of edges (pop everything till the mark). If the subtree had multiple biconnected components, they would be already popped off before the DFS returned to v.