An Approach to Prove Correctness of Graph Algorithm

Vinay Kumar*

Abstract

An algorithm is said to be correct if for every input data that satisfies some conditions-called the precondition of the algorithm, the output data satisfy a certain predefined condition-called the post condition of the algorithm. A graph algorithm depends upon number of nodes and edges in a graph and inter/intra relationship between these two features. In this paper an approach based on incremental induction on edge or on node or on both is described to mathematically prove or disprove the correctness of a graph algorithm. This is required in view of the complex nature of graph theory world and changing properties of graph with changes in the number of nodes, edges and their relations in the graph.

Keywords: Algorithm, Correctness, Complexity, Deterministic, Heuristic, Vertex, Edge, Degree, Invariance, Pre-condition, Post-condition, Loop Invariant

1. Introduction

While teaching and learning a graph algorithm, it is felt that proper way of proving its correctness is essentially required. In most of the paper/text more emphasis is given on computing complexity [2, 7, 19] of an algorithm rather than on proving correctness of the graph algorithm. An algorithm is any well defined computational procedure that takes set of values as input and produces some set of values as output [8]. An algorithm consists of explicit and unambiguous finite steps which when carried out for a given set of initial conditions, produces the corresponding output and terminates in a finite time [12]. An algorithm is said to be correct if for every input instance, it halts with correct output [8]. The proof of correctness of an algorithm has two parts:

- Partial correctness: If the algorithm terminates then it will give the right result (the result will satisfy the post condition).
- Termination: Proof that the algorithm terminates.

An algorithm contains one or more of the constructs like: sequential, conditional, loops and recursive [16]. There are algorithms for finding shortest path [3, 6, 11], minimum

*Scientist 'E', NIC, Lodhi Road, New Delhi, India

spanning tree [1, 17] graph connectivity, graph traversal besides other problems in a graph. Some of the algorithms are greedy and some are based on dynamic programming concept [14]. Graph algorithms are also developed to determine whether a graph is planar, whether it is Hamiltonian [5, 10, 20, 21], whether it is an Euler graph and to determine others such characteristics. Graph algorithms are designed and implemented either iteratively or recursively or in both ways. A graph has two basic components: node and edge. Invariably, all graph algorithms are designed in terms of either edge or node or both.

Proving correctness of an algorithm is not similar to program verification. Program verification usually refers to a formal proof that examines the actual code implementing some algorithm in a step-like fashion and shows that the program actually achieves the goal. Whereas proving correctness of algorithm implies that the algorithm terminates and when it terminates it produces correct output for a given input [1, 13]. A predicate that describes the initial state before execution of an algorithm is called pre-condition of the algorithm. Similarly a predicate that describes the final state after execution of the algorithm is called post-condition of the algorithm. Our endeavor is to formulate a process to determine the followings from a graph algorithm:

i) Pre conditions of the algorithm
ii) Post conditions of the algorithm
iii) Termination condition
iv) Progress conditions and updating of states that leads to termination of algorithms and intermediate states that lead to post conditions

And then to prove finally that algorithm terminates and when it terminates the post condition of the algorithm is well satisfied given that precondition was satisfied.

In this paper, an approach is presented to prove the correctness of a graph algorithm. The paper is organized is five sections. Section 2 describes method to formulate problem. How to convert the given algorithm in algebraic equations is described in Section 3 with an example. In Section 4, the process of proving correctness of algorithm is mentioned. The paper is concluded in Section 5.

2. Problem Formulation

Let G = (V, E) be a graph where V is set of nodes and E is set of edges in the G and P(G) be a graph problem and Algo(P(G)) be an algorithm to solve the problem. The Algo(P(G)) may be sliced into sequential parts as shown in the Figure 1(a). A sequential part may be a single assignment, a loop, or it may be a condition as shown in the Figure 1(a), (b) and (c) respectively. A graph algorithm may consist of some functions (or subroutines). A function may be recursive. Control flow corresponding to a simple function call and for a recursive function is shown in the Figure 2 (a) and (b) respectively.

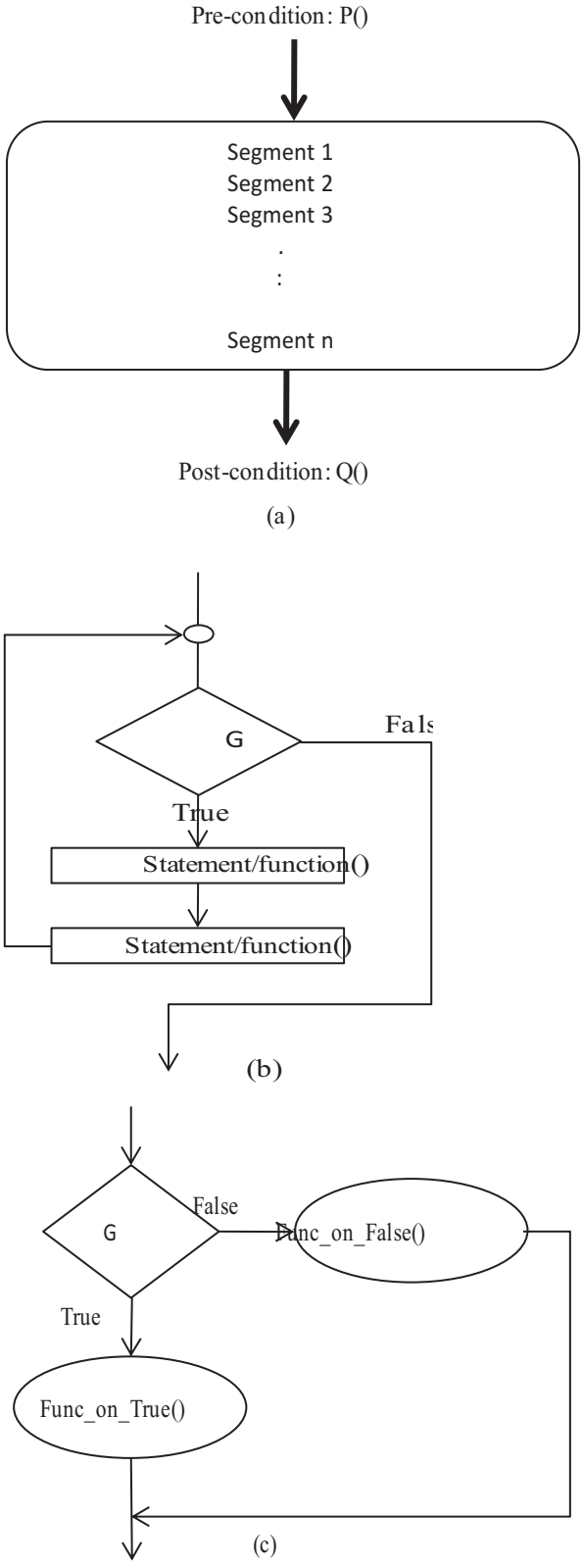


Fig. 1: (a) Sequential Slicing (b) Loop (c) Condition

An algorithm is said to be correct if it can be proved that if the pre-condition is true, the post-condition must be true i.e.

Pre-condition ⇒Post-condition

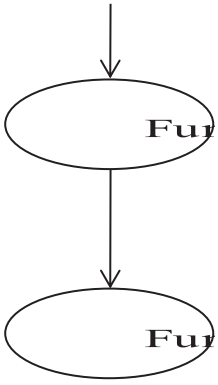
Assignment operations (statements) are fairly straightforward to deal with. Number of statements in a sequential slicing is always

finite; therefore the proof is carried out at each statement in that order. A conditional statement facilitates selection of one segments out of two segments (if....else) or out of multiple segments (switch). The correctness of conditional statement depends on the 'Guard G', as shown in the Figure 1(c) and on success or failure of the execution of individual statement as control flows automatically to next statement in the sequence. The case of a loop, however, is different. In this case we write a loop invariant that captures the purpose of the loop. A loop invariant, say I(n), is an end predicate where the variable, n, represents the number of times the loop has iterated.

We consider a 'while' loop with pre-condition, post-condition and a guard G as shown in the Figure 1(b). To prove that the loop is "correct", we do the followings [1, 8, 12]:

- Write a loop invariant I(n) that will be true after n iterations of the loop.
- Prove that the pre-condition implies the loop invariant I(0). We refer I(0) as basis step.
- Prove that I(k) I(k+1) i.e. when I(K) and guard G are true then I(k+1) is true. This step we refer as inductive step.
- Prove that Guard G eventually becomes false and loop terminates.
- If the loop iterates for a maximum of n times, then I(n) implies post-condition.

A 'repeat until' kind of loop may also be translated in this form. Many graph algorithms are developed in recursive form [4]. A recursive algorithm is proved correct using mathematical induction. In many cases strong induction is not needed, all we needed was correctness for k in order to prove correctness for k+1. But the case of graph is somehow different. A graph G may possess a property with n nodes and m edges, but it may not possess the same property when either n or m or both changes. Therefore, in case of a graph, we need strong induction [9, 11].



(

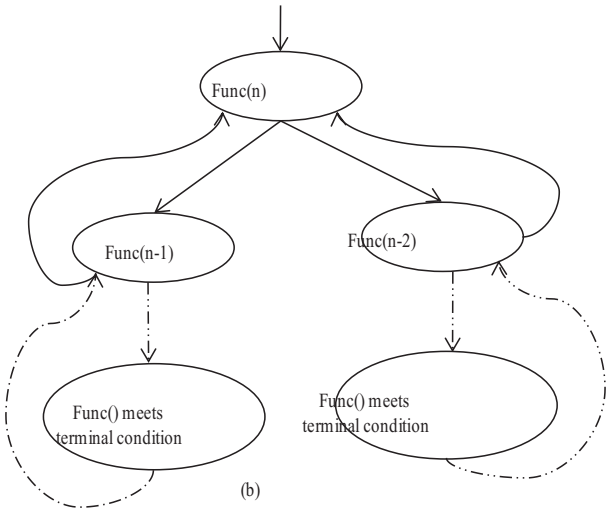


Figure 2: Depicts control flow in (a) Non recursive function (b) recursive function call

Before proceeding to construct proof for a graph algorithm, it is essential to translate the respective sections: assignment, condition, loop, function both explicit and recursive part into corresponding algebraic equations and / or propositions. It is discussed in the following section.

3. Algorithm to Algebraic Equation

In order to prove the correctness of graph algorithm, we translate the algorithm to mathematical/propositional relations. Let Algo (P(G)) be the algorithm ArticPointDFS (Vertex v) to find articulation point, if any, in a connected graph G(V, E). This example is taken to illustrate the working of the concepts that has been formulated. Vertex and node are synonymous.

An articulation point is a node [1, 15] whose removal makes the graph disconnected. An articulation point in the graph G can be determined by using Depth-First Search (DFS) method [1]. In a DFS tree of an undirected graph, a node *u* is an articulation point, if for every child *v* of *u* 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 has 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 dfsnum (v) and LOW (v) [1]. LOW(v) is the lowest *dfsnum* of any vertex that is either in the DFS sub-tree rooted at v or connected to a vertex in that sub-tree by a back edge. The Artic Point DFS (Vertex v) is a recursive algorithm. Then, in DFS, if there are no more nodes to visit, we back up and update the values of LOW as we return from each recursive call. The algorithm also determines any biconnected component, if any, in the graph. A biconnected component in a graph is defined as the sub-graph which is a connected sub-graph with no articulation points.

When we process an edge (u, x) - either by a recursive call on vertex x from vertex u, or (u, x) is back edge, we push that edge to a stack. Later, if we identify u as an articulation point, then all edges from the top of the stack down to (u, x) are the edges of one biconnected component. So, we pop edges out of the stack until the top of the stack is (u, x). Those edges belong to a biconnected

component. The pseudo code of the algorithm is given below. Global variables like v.dfslevel, v.numChildren and low value of the nodes are initialised to 0 for all vertices v.

3.1 Algorithm: Finding articulation point in a graph G.

```
1.  Global initialization: v.dfsnum := -1 for all v.
2.  dfsCounter := 1
    ArticPointDFS (Vertex v) {
3.  v.dfsnum := dfsCounter++
4.  v.low := v.dfsnum
5.  for each edge (v, x) do {
6.  if (x.dfsnum == -1) then do { // x is unvisited
7.  x.dfslevel := v.dfslevel + 1
8.  v.numChildren++
9.  stack.push_edge (v, x) // add this edge to the stack
10. ArticPointDFS (x) // recursively perform DFS at
    children nodes
11. v.low := min(v.low, x.low)
12. if (v.dfsnum == 1) then do{// Special Case for root
    // Root is an articulation point iff there are two or
    more children
13. if (v.numChildren >=2) then articPointList.add (v)
    // Retrieve all edges in a biconnected component
14. while (stack.top != (v, x))
    bccEdgeList.add (stack.pop)
    } //end of if in line 12
15. else {
16. if (x.low >= v.dfsnum) then do {
    // v is artic. point seperating x. Children of v cannot
    // climb higher than v without passing through v.
17. articPointList.add (v)
    // Retrieve all edges in a biconnected component
18. while (stack.top != (v, x))
    bccEdgeList.add (stack.pop)
    } //end of if in Line 16
    } //end of else in Line 15
    } // end of if in Line 6
19. else if (x.dfslevel < v.dfslevel - 1) then do {
    // x is at a lower level than the level of v's parent,
    // equiv. (v, x) is a back edge
20. v.low := min(v.low, x.dfsnum)
    // add the back edge to the stack
21. stack.push_edge (v, x)
    } // end of if in line 19
    } // end of for each in line 5
    } // end of function in
```

The algorithm consists of 'for' loop in line 5, 'while' loop in line 14, 18, 'if … else' in lines 6, 12, 15, 16 and 19, recursive call of the function Artic Point DFS () in line 10 and assignment statements elsewhere. The algorithm also includes function calls at line 9, 11, 13, 14, 17, 18, 20 and 21. Let us first coverts the assignment statements into corresponding algebraic equations.

3.2 Assignment

Let number of nodes in graph G be N. Equations corresponding to initialization at line 1 and 2 are given in (1) and (2) respectively.

v_j.dfsnum = -1 for j=1, 2, … N ..... (1)

dfsCounter = 1 ..... (2)

Assignment in line 3 has two parts: one is initialization and other is increment in that order. Therefore the corresponding equations are given in equations (3) and (4). The assignment in line 4 is converted into equation (5).

v_j.dfsnum = dfsCounter ..... (3)

dfsCounter = dfsCounter + 1 ..... (4)

v_j.low = v_j.dfsnum ..... (5)

Equations corresponding to other assignments in lines 7, 8, 11 and 20 are in equations (6, 7), (8), (9) and (10) respectively.

v_k.dfslevel = v_j.dfslevel ..... (6)

v_j.dfslevel = v_j.dfslevel + 1 ..... (7)

v_j.numChildren = v_j.numChildren + 1 ..... (8)

v_j.low = a ..... (9)

v_j.low = b ..... (10)

If Then else

Now while dealing with if-else structures, let us see how this structure works. Consider a statement of the form

S = if <condition> then B_1

Let p_1 denotes the part of the algorithm appearing before S, and q_1 denotes the part of the algorithm appearing after S. The following rule will be applicable:

$$(p_1 \,/\backslash\, \text{condition})\ \{B_1\}\ q_1$$

It implies, "if p_1 and the <condition> are true when B_1 starts and B_1 terminates, and when B_1 terminates then q_1 is true." And, if p_1 is true and the condition is false, then q_1 is true (doesn't rely on B_1) i.e.

$$(p_1 \,/\backslash\, \sim\text{condition}) \rightarrow q_1$$

Therefore,

$$p_1\ \{S\}\ q_1 \quad \text{..... (11)}$$

For example "if a < b then b:= a" is correct with initial assertion p_1 = "T" (true) and final assertion q_1 = "b $\leq$ a". Further, let us see how to deal with 'if-then-else'. Suppose S is of the form

S = if <condition> then B_2 else B_3

Then we have,

$$(p_2 \,/\backslash\, \text{condition})\ \{B_2\}\ q_2$$
$$(p_2 \,/\backslash\, \sim\text{condition})\ \{B_3\}\ q_2$$

Therefore,

$$p_2\ \{S\}\ q_2 \quad \text{..... (12)}$$

In the given algorithm, 'if ... then ...' in line 6 and its 'else ...' in line 19 is modeled as per equation (12). Similarly, 'if ... then ...' in line 12 and its 'else ...' in line 15 is modeled as per equation (12). The 'if ... then ...' in line 16 and 19 are modeled as per equation (11). In any algorithmic design another important construct is 'loop'. Let us see how we deal with loop, while proving correctness of algorithm.

Loop Invariant

Consider a statement of the form

S = while <condition> B_4

A loop invariant is a statement p_3 that remains true each time B_4 is executed i.e.

$$(p_3 \,/\backslash\, \text{condition})\{B_4\}p_3 \quad \text{..... (13)}$$

is true. It is a Hoare triple. A Hoare triple has three parts, a

precondition P, a program statement or series of statements S, and a post condition Q. It's usually written in the form

$$\{P\}\ S\ \{Q\}$$

The meaning is "if P is true before S is executed, and if the execution of S terminates, then Q is true afterwards". Note that the triple does not assert that S will terminate. It requires a separate proof. Let us consider a simple algorithm of computing factorial of a positive integer as an example to better understand the concept of loop invariant before actually using this for proving correctness of a graph algorithm.

```
i := 1
factorial := 1
while (i < n) do
      i := i+1
      factorial := factorial*i
end while
```

In the above algorithm, "condition" is "i < n" and B_4 consists of the body of the loop {i := i+1; factorial := factorial*i} and loop invariant p3 is "(factorial = i! and i <= n)". Thus the following Hoare triple is true.

```
((factorial = i! and i <= n) /\ (i < n))
{i := i+1; factorial := factorial*i}
(factorial = i! and i <= n)
```

In the given algorithm, the 'for' loop of lines 5 and while loops in line 14, 18 are modeled as per the Hoare triple in equation (12). Now let us see how to deal with function call in an algorithm while proving its correctness.

3.3 Subprograms

Suppose program S consists of subprogram S_j followed by another subprogram S_k, denoted $S = S_j; S_k$ and p, q, r be predicates then the situation can be modeled as

$$p\ \{S_j\}\ q$$

Which implies that if p is true when S_j starts and S_j terminates, and when S_j terminates then q is true and

$$q\ \{S_k\}\ r$$

implying that if q is true when S_k starts and S_k terminates, and when S_k terminates then r is true. Combining these two formulations, we get

$$p\ \{S\}\ r \qquad \text{..... (14)}$$

which indicates that if p is true when S starts and S terminates, and when S terminates then r is true. The function calls at line 9, 11, 13, 14, 17, 18, 20 and 21 are modeled according to equation (14). There is one recursive function call at line number 10. Correctness proof of a recursive function is handled differently. Let us deal with this situation in the following subsection.

3.4 Recursive Subprograms

Correctness of a recursive algorithm is proved using mathematical induction. In a graph, induction is applied either on number of nodes or on the number of edges. The methodology is used to show that a recursive sub-program is partially correct and it terminates. Let us consider a simple recursive algorithm of computing a^n.

```
procedure power (a: nonzero real, n: nonnegative int)
{
if (n = 0) then return 1
else return a * power(a; n – 1)
}
```

Let P(n) be "for all nonzero real numbers a, power(a; n) correctly computes a^n". Using mathematical induction on n, it can be shown that the algorithm is correct for n 0 i.e. P(n) is true for all n 0. In the basis step, if n = 0, the first step of the algorithm tells us that a^0 =1. And therefore P(0) is true because a^0 = 1 for every nonzero real number a. In inductive step, when P(k) is true i.e. power (a; k) = a^k, then power (a; k + 1) = a * power (a; k) = a^{k+1} i.e. P(k+1) is true. Finally, the algorithm terminates because for any non negative integer n, successive decrement by 1 leads to zero – the terminating condition of the algorithm. Therefore the algorithm is correct.

In general, the progress of algorithm is simulated through inductive step which helps in proving partial correctness of the algorithm. And basis step helps in proving that ultimately recursive algorithm terminates. In the following section, a method for construction of prove is formulated.

4. Construction of Proof

Assignment statements are fairly straightforward to deal with. While proving the correctness, bind the variable with values so that preconditions and post-conditions are clearly highlighted and visualized. Continuing with the example taken in the paper, equation (1) and (2) perform the global initialization. This is the pre-condition of the algorithm wherein dfsnum (depth first number) of all nodes in graph is assigned value –1. It means, all nodes in the graph are unvisited and dfsCounter (depth first counter) is equal to one. Also v.dfslevel, v.numChildren and low value of all the nodes are initialised to 0. Therefore, preconditions of the algorithm are:

v_j.dfsnum = –1 for j = 1, 2, ….N and
dfsCounter = 1
v_j.low = 0
v_j.numChildren = 0
v_j.dfslevel = 0

Besides the list of biconnected component (bccEdgeList), list of articulation point (articPointList) and stack containing tree edges are empty in the precondition.

The post conditions of the algorithm are:
v_j.dfsnum = Order in which the nodes are visited starting from the first node v1,
dfsCounter = N+1, where N is the number of nodes in graph G,
v_j.low = Number of edges required to reach the node from root node,
v_j.numChildren = Number of nodes at next dfslevel,
v_j.dfslevel = Level at which the node is from the root node. Level of root node is 0,
The bccEdgeList contains biconnected component, if any, of the graph. The articPointList contains all the articulation points, if any, in the graph. And the stack is again empty in the post condition. To illustrate the working of process, we have taken a graph, which and its corresponding DFS tree is shown in the Figure 3 (a) and (b) respectively. The corresponding post conditions are listed in the Table 1.

First, convert assignment statements into atomic propositions. Similarly, a function call is converted into atomic proposition because the correctness of the algorithm of that function is proved elsewhere and it is assumed that the function being called is tested and works well whenever it is called with suitable parameters. Let *p* represents assignment statement in line 1 and 2, *q* represent that in 3 & 4, *r* represents that in 7 & 8 and s is atomic propositions for assignment statement in 11 and 20. Let S_1, S_2, S_3 and S_4 be propositions representing functions stack.push_edge (v, x), min (v.low, x.low), articPointList.add(v) and bccEdgeListadd(stack.pop) respectively. After successful execution of a function, control returns to the calling function. Let the situation be represented by the proposition T. Further, let S be the proposition for the function ArticPointDFS(). Now, as per equation (14), we can write propositions to prove correctness of S_1, S_2, S_3 and S_4 respectively as follows.

r {S_1} T

r S {S_2} s

(T ^ (v.numChildren >=2)) {S_3} T, for function call at line 13 and
T ^ (x.low >= v.dfsnum)) {S_3} T, for function call at line 17

((Stack not empty) ^ (Partial list of edges of biconnected component) ^ (stack.top != (v, x))) {S_4} (A list of edges of biconnected Component) T

These propositions are to prove correctness of functions call at lines 9, 11, 13, 14, 17, 18, 20 and 21. Let S_5, S_6, S_7, S_8 and S_9 be propositions for 'if … then … else' at line 6, 12, 15, 16 and 19. Then these propositions can be written as follows.

S_5: (T ^ (x.dfsnum == -1)) {Line 7 to Line 18} T
(T ^ (x.dfsnum == -1) ^ (x.dfslevel < v.dfslevel - 1)) {Line 20 to Line 21} T

The second part of S5 also includes the 'if … then' statement of line 19 i.e. S_9.

S_6: (T ^ (v.dfsnum == 1)) {Line 13 to Line 14} T
(T ^ (v.dfsnum == 1) ^ (x.low >= v.dfsnum)) {Line 17 to Line 18} T
The second part of S_5 also includes the 'if … then' statement of line 16 i.e. S_7.

Truthiness of propositions S_1 to S_9 ensures correctness of corresponding parts of the algorithm. A 'for' and a 'while' loop iterates as long as the condition is true. The correctness is proved using the concept of loop invariant. The loop invariant for while loop at line no. 14 and 18 is given in the expression outlined for proposition S4 above in this section. The loop invariant for 'for' loop in the ensuing example is derived according to the concept outlined in the Section 3. Let number of nodes adjacent to node v be m and number of visited nodes at a time before completion of loop be k, then 'loop invariant', condition and body of the 'for' loop at line 5 are as follow:

Loop invariant:
((number of visited node = k) (k <= m))

Condition: (k < m))
Body: {Line 6 to Line 21}
Thus the Hoare triple for the 'for' loop is
((number of visited node = k) ^ (k <= m) ^ (k < m))
{Line 6 to Line 21}
((number of visited node = m) ^ (k <= m) ……. (15)

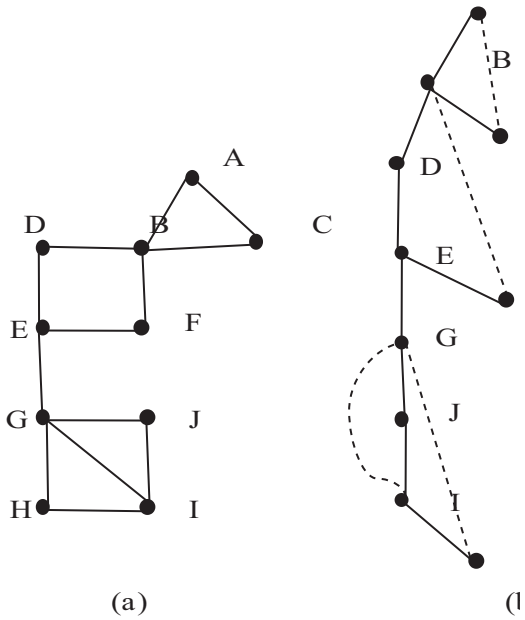


Fig. 3: (a) A connected graph (b) DFS tree of the graph in

Here truthiness of this Hoare Triple ensures the correctness of 'for' loop. Now, the correctness of the recursive function ArticPointDFS() needs to be proved. The function is recursively called in line 10. Let A(n) be "for a connected graph G with n vertices, ArticPointDFS() correctly produces outputs similar to that in table 1 and finds all biconnected components, if any".

Table 1: Showing values assigned to nodes in DFS tree of graph in figure 3 (a)

Sl. No.	Node	Dfsnum	dfslevel	low_ value	No. of Children	Articulation Point
1.	A	1	0	1	1	
2.	B	2	1	2 (1)	2	B is an articulation point and (B, D), (D, E), (E, F), (F, B) is a biconnected component.
3.	C	10	2	10 (1)	0	
4.	D	3	2	3 (2)	1	
5.	E	4	3	4 (2)	2	E is an articulation point and (E, G) is a biconnected component.
6.	F	9	4	9 (2)	0	
7.	G	5	4	5	1	G is an articulation point and (G, H), (G, J), (H, I), (G, I), (I, J) is a biconnected component.
8.	H	8	7	8 (5)	0	
9.	I	7	6	7 (5)	1	
10	J	6	5	6 (5)	1	

Here, we use incremental induction to prove the correctness of this algorithm. A(1) is trivially true. Let A(k) be true i.e. ArtticPointDFS() works correctly for for a connected graph with k nodes. Let x be new node in A(k+1). G is a connected graph, therefore x is adjacent to at least one node, say u. Thus (u, x) is an edge and therefore for loop at line 5, iterates for one more time. Since x is unvisited, 'if' statement at line 6 will be true and ArtticPointDFS() will be called for node x. Statements at line 7, 8, 9, 10, 3 and 4 are execute in that order on node x. ArtticPointDFS() terminates because there will be no unvisited node left in the graph G. Propositions S_4 to S_9 are true and hence post conditions of the algorithm are satisfied. It is now proved that the post-condition is true whenever the algorithm terminates, therefore the algorithm is partially correct. Finally, let us prove that the algorithm terminates. Termination of 'while' loops are explained earlier while explaining S_4.Termination of algorithm depends on (i) termination of recursion and (ii) termination of for loop. As explained in Hoare triple in equation (15), for loop terminates. Once all the nodes in a graph is visited, there is no recursive call of ArtticPointDFS() and recursion terminates finally.

5. Conclusion

The paper describes an approach to prove correctness of a graph algorithm. The process is based on mathematical concepts. An algorithm is classified into sections like assignment, conditional selection (if…else) loop, function call and recursive function. The concept of recursion is important to many fields of study, especially when many applications rely on computer software for data analysis and prediction. Each section is translated into corresponding algebraic equations or propositional equations. These equations are then used to prove correctness of a graph algorithm. The concept can be further extended to both restricted [1] and unrestricted backtracking.

The approach is useful for teacher and student who want to learn how to prove an algorithm correct. It is useful in the area of algorithmic research as well for proving correctness of an algorithm. The example taken is only for illustrative purpose. Since an algorithm consists one or more or all of the logical constructs: sequence, conditional selection, loop, recursive function, normal function call, the general approach derived is applicable to any graph algorithm Algo(P(G)).

Further, the method presented in this paper is useful in proving correctness of greedy algorithm, dynamic programming based algorithm, recursive and iterative algorithm. Many graph algorithms are either greedy algorithm or based on dynamic programming concept. The difference between proving an algorithm correct and verification is also explained in this paper.

Through years of conveying the concept of correctness proof of algorithm to students in Computer Science, the authors have found that learning correctness proof of graph algorithm is nothing more than the old saying: 'practice makes perfect' [Anonymous]. However, just as in most learning environments, an adequate learning approach is the key to success. The approach provided in this paper is reflective of one of the goals of the teaching and learning in which -the faculty frame and systemically investigate questions related to pre-condition/post-condition -the conditions under which the algorithm is being investigated [18].

6. Acknowledgement

Existence of a problem inspires one to find its solution. One has to look around to know the presence of something worth noticing. This research and subsequent implementation of the concept made possible by systemic evaluation and testing of the concept on different algorithms. To put a solution in place, resources are required. I am very grateful to all those who have been constantly encouraging and providing required resources for such scientific research work besides the regular work of the departments. Special thanks to some of my students who always chant for something new.

7. References

1. *Aho, A.V., Hopcroft, J.E. and Ullman, J.D., The Design and Analysis of Computer Algorithm. Reading MA: Addison-Wesley, 1994.*
2. *Akiyama, T., Nishizeki, T. and Saito, N., 'NP-completeness of the Hamiltonian cycle problem for bipartite graphs'. J. Inf. Process, 1980, Vol. 3, No. 2, pp.73–76.*
3. *Berge, C., Theory of Graphs and its Application. London: Methuen Press, 1962.*
4. *Blazewicz, J., Hertz, A., Kobler, D. and de Werra, D., 'On some properties of DNA graphs', Discrete Applied Mathematics, 15 November, 1999, Vol. 98, Nos. 1–2, pp.1–19.*
5. *Bollobas, B., 'Almost all regular graphs are Hamiltonian', European Journal of Combinatorics, 1983, Vol. 4, pp.97–106.*
6. *Bondy, J.A. and Murty, U.S.R., Graph Theory with Applications. New York: North-Holland, 1976.*
7. *Bringsjord, S. and Taylor, J., 'P = NP'. Department of Cognitive Science, Department of Computer Science, RAIR Lab, RPI, Troy, NY, 2004.*
8. *Cormen, T.H., Leiserson, C.E. and Rivest, R.L., Introduction to Algorithms. PHI, 1998.*
9. *Corneil, D.G., Lerchs, H. and Stewart Burlingham, L., 'Complement reducible graphs', Discrete Appl. Math., 1981, Vol. 3, No. 3, pp.163–174.*
10. *Deogun, J.S. and Steiner, G., 'Polynomial algorithms for Hamiltonian cycle in comparability graphs', SIAM Journal on Computing, 1994, Vol. 23, No. 3, pp.520–552.*
11. *Dirac, G.A., 'Some theorems on abstract graphs', Proc. London. Math. Soc. (3) 2, 1952, pp. 69-81.*
12. *Dromey, R. G., How to solve it by Computer. New Delhi: PHI, 2002.*
13. *Garey, M.R. and Johnson, D.S., Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: W. H. Freeman, 1979 pp.199–200.*
14. *Held, M. and Karp, R.M., 'A dynamic programming approach to sequencing problems'. SIAM Journal on Applied Mathematics, 1962, Vol. 10, No. 1, pp.196–210.*
15. *R. M. Karp and J. M. Steele., 'Probabilistic analysis of heuristics'. In E. L. Lawler et ah, editor, The Travelling Salesman Problem, Essex: John Wiley & Sons, 1985, pp 181-205.*
16. *Knuth, D.E., The art of computer programming, Fundamental Algorithm. Vol. 1, Reading Mass: Addison-Wesley, 1973.*
17. *Kumar, V., Graph Theory, Chapter 7, Discrete Mathematics, 1 ed., BPB Publication, New Delhi, India, 2002.*
18. *Lawler, E.L., Combinatorial Optimization: Networks and Matroids. New York: Holt, Rinehart and Winston, 1976.*
19. *Lewis, H. and Papadimitriou, C., 'The efficiency of algorithms'. Scientific American, 1978, Vol. 238, No. 1, pp.96–109.*

20. *Mark K. J., 'Finding Hamiltonian circuits in interval graphs'. Inform. Process. Lett., 1985, Vol. 20, No. 4, pp.201–206.*
21. *Shamir, E., 'How many edges are needed to make a random graph Hamiltonian?' Combinatorica, 1983, Vol. 3, pp. 123-132.*