ISSN:0976-4933
Journal of Progressive Science
Vol.09, No.01 & 02, pp 44-56 (2018)

A study of Graph-Theoretic Algorithms

Sudhir Prakash Srivastava
IET, Dr.Ram Manohar Lohia Avadh University, Ayodhya,
Uttar Pradesh-224001, India
Email- sudhir_ietfzd@yahoo.com

Abstract

A study of Graph-theoretic algorithm is discussing about Graph theory and combinatorial analysis of arrangements, ordering, selection of discrete objects etc. First of all, we describe some basic about Graph theory. Then we focus on major algorithm to describe different type of case which is very important part of Graph-Theoretic algorithm. This study is very significant for interested reader to find it fruitful sources and discover for himself this wealth of knowledge.

Keywords- Spanning tree, Shortest Path, Adjacency Matrix, Algorithm

Introduction

Graph theory and combinatorial analysis involves the study of arrangements, ordering, selection of discrete objects etc. The questions normally asked are those of existence or of enumeration. With the advent of digital computer, a new type of investigation has gained importance. Not only "does the arrangement exist?", "how many arrangements are there?" are the question of interest," what is the best arrangement?" "how does one find all the arrangements satisfying a particular property?" are becoming matters of concern to the combinatorial lists. Interestingly enough the digital computer has itself created technical problems of combinatorial nature. Research in computer design, the theory of computation, application of computer to numerical and non-numerical problems have required new methods, new approaches, and new mathematics insights.

From one view point the problems of "what is the best arrangement" etc. are trivial since there are a finite number of feasible solutions to graph-theoretic problems. For example, the problem of finding the lowest weight Hamiltonian circuit in a weighted complete graph of n vertices can be solved by just listing the ½ (n-1)! different

Hamiltonian circuits and then picking the one of lowest weight. These brute force techniques won't work. If the computer is programmed to examine each of these feasible Hamiltonian circuits at the rate of one each nanosecond, it will finish its task for n=21 in about 400 years, for n = 22 in about 8,400 years and so on. Clearly the brute force enumeration technique is not "effective" for the Hamiltonian circuit problem. How does one evaluate the effectiveness of a solution procedure or an algorithm? One standard which is now most accept is that of "Polynomial bounded". An algorithm is considered "good" if the required number of elementary computational steps is bounded by a polynomial in the size of the problem (Lawler, 1976), (Srivastava, 2016). Number of questions crop up, what is an elementary computational step? What is meant by the "size" of a problem and why the polynomial bound?

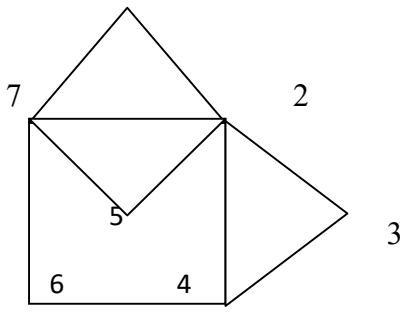
Polynomial bound, essentially because a polynomial function grows less rapidly than an exponential function. An exponential function grows much less rapidly than a factorial function. The issues of computational step and size of the problem are somewhat inter-related and for our purposes is representation dependent.

Representations

A graph G = (V, E) where V is the set of vertices and E is the set of edges has the most familiar representation on paper by dots and line segments. In a computer the graph must be represented in a discrete way. The following are the most common methods.

a. Adjacency Matrix

An adjacency matrix of G = (V, E) is a |V| ×|V| matrix. If A = $[a_{ij}]$ in which a_{ij} = 1 if there is an edge from vertex i to vertex j in G, otherwise a_{ij} = 0. Figure 1 shows a directed graph and its adjacency matrix.



$$A=\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

b. List of edges

Sometimes it may be enough to list the edges in the graph as pairs of vertices implemented by two arrays. g = {g_1, g_2, …… $g_{[E]}$} and h = {h_1, h_2, ……., $h_{[E]}$}. Each entry is a vertex label, and the i^{th} edge in G goes g_i to h_i. for the digraph of Fig. 1 a representation could be

g = (1, 2, 2, 2, 2, 3, 4, 5, 5, 5, 6)

h = (7, 3, 4, 5, 7, 4, 6, 1, 6, 7, 7)

c. Adjacency structures or lists

A vertex y in a directed graph is called a successor of another vertex if there is an edge directed from x to y; vertex x is then called the predecessor of y. In an undirected graph two vertices are called neighbors of each other if there is an edge between them. A graph can be described by the list of all successors (neighbors) of each vertex; Adj (v) is list of successors (neighbors) of v. for the digraph of figure 1 an adjacency structure is as follows:

v		Adj (v)
1	:	7
2	:	3, 4, 5, 7
3	:	4
4	:	6
5	:	1, 6, 7
6	:	7
7	:	-

If the space needed to store an integer is the unit of space then one can easily see that using list of edges or adjacency structures one can represent the graph G in no more than $0(|V|+|E|)$ space (f(x) = 0 (g(x)) as $x \longrightarrow x_o$ If and only if there exists a constant c such that

$$\lim_{x \longrightarrow x_o} \sup \left|\frac{f(x)}{g(x)}\right| = 0.$$

We say that the function grows no faster than g(x)). Adjacency matrix on the other hand requires $0(|V|^2)$ space. The unit of computation step may be roughly defined as the amount of time needed to look at an edge in the graph under this criterion. Most algorithms would require at least $0(|V|^2)$ time if the graph is represented by the adjacency

matrix and $0(|V|+|E|)$ time if list of edge or Adjacency structures are used. This is because for most non-trivial problems at least each edge in the graph would need to be looked upon at least once.

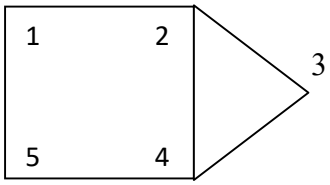
Some Basic Algorithms

The most general question about graphs concern connectivity, paths and distances. We may want to find out whether the graph is connected; if it is connected then what is the shortest distance between a specified pair of vertices. If it is disconnected, we may be interested in determining its connected components. Algorithms for answering some of these questions and related matters will be discussed in the following.

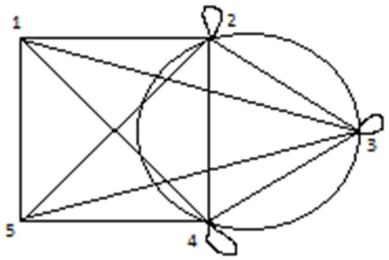
1- Spanning trees

Spanning trees of a graph G are sub graphs of G and contain every vertex of G. If G is not connected then the set consisting of a spanning tree for each component is called a spanning forest of G. In a weighted graph, i.e. a graph which has weights (real numbers) associated with each edge it is often of interest to determine a spanning tree (forest) of minimum weight (i.e. sum of the weights of all edges is minimum). An interesting algorithm for finding a minimum spanning tree is by Kruskal (1956). The algorithm is as follows: List all the edge of graph G in the order of non-decreasing weight. Next, selected the smallest edge of G. Then for each successive step select (from all remaining edges of G) another smallest edge that makes on circuit with the previously selected edges. Continue until $|V|$-1 edgeshave been selected, and there edges will constitute the desired minimum spanning tree T. This process is known as the greedily algorithm. That the greedy algorithm produces the minimum spanning tree can be proved as follows. Suppose T were not a minimum spanning tree. Let the greedy algorithm add edges to T in the order e_1, e_2, ……., e_n so that these edges are in the order of non-decreasing weights. Let T_{min} be a minimum spanning tree that contains edge e_1, e_2, ……., e_{i-1} for the largest possible i. Clearly, $i \geq 1$, and since T is not a minimum spanning tree, $i \leq n$. Adding e_i to T_{min} causes a cycle that must include an edge $x \neq e_j$, $1 \leq j \leq i$. Since x and e_1, e_2, ……., e_{i-1} are in T_{min}, which is acyclic, and since the greedy algorithm adds the edge of least cost that does not cause a cycle, we know that the weight of x is at least the weight of e_i (otherwise x would have been added to e_1, e_2, ……., e_{i-1} instead of e_i). If the weight of x is more than the weight of e_i , then

$T_{min} = (T_{min} - \{x\})\ \{ e_i \}$ is a spanning tree of less total weight then T_{min} , contradiction. Thus, the weight of x and the weight of e_i must be equal, making a T'_{min} a minimum spanning tree and contradicting the assumption that i was as large as possible. Therefore, T must be a minimum spanning tree.



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$



$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

It can be shown that the greedy algorithm has time requirement of 0($|E|$ log $|E|$).

Often it is of interest to examine all the spanning tree of a graph particularly in electrical networks analysis when the networks are interpreted as graphs. Since the number of spanning trees grows exponentially with the number of vertices the problem becomes infeasible even for moderate size graphs until case is taken to do the generation efficiently. The most successful algorithm is by Minty. The set of all spanning trees of 0 is divided into two classes, those that contain a specific edge (u,v) and those that do not contain (u,v) the spanning trees that contain (u,v) are the one consisting of (u,v) and a spanning tree of the graph $G_{u,v}$ obtained from G by merging vertices u and v into a "super vertex" (removing any self-loops that might result). Other spanning trees are the spanning trees of the graph G (u,v), obtained by deleting edge (u,v) form G.

Notice that $G_{u,v}$ and G – (u,v) are smaller than G. Thus successive applications of this basic step reduce the graphs until either all n vortices are merged together into one super-vortex or the graphs become disconnected and have no spanning trees. If properly implemented, this algorithm requires 0 ($|V| + |E| + |E|$ .t) operations, Where t is the number of spanning trees of G. Figure 2 illustrates the process on a small example. Large circles denote "super-vortices".

2- Transitive Closure

Suppose that G = (V, E) is a directed graph represented as a $|V| \times |V|$djacency matrix A = $[a_{ij}]$. We would like to compute the connectivity matrix A* = $[a *_{ij}]$ defined by $a *_{ij}$ =1 if there is a path in G fro i to j and $a *_{ij}$= 0 if not. If we view E, the edge of G as a binary relation of V, the vortices of G, then $A *$ is the adjacency matrix for the graph G*=(V, E*) where E* is the transitive closure of the binary relation E. Figure 3 show a graph G, its adjacency matrix A, its transitive closure G* and its adjacency matrix A*.

The method for computing A* from A given below is due to Warshall (1962). Its importance as the same method with some modification is also the best-known method for computing the shortest distance between all pairs of vortices in a graph.

A* is computed from A by defining a sequence of matrices

$A^{(O)} = \left[a_{ij}^{(O)}\right], A^{(1)} = \left[a_{ij}^{(1)}\right], \ldots., A^{|V|} = \left[a_{ij}^{|V|}\right]$ as follows:

$$a_{ij}^{(O)} = a_{ij},$$

$$a_{ij}^{(l)} = a_{ij}^{(l-1)} \mathrm{V} a_{ij}^{(l-1)} \cap a_{ij}^{(l-1)}$$

It is not difficult to show, using induction, that $a_{ij}^{(l)}$=1 if and only if there is a path from l ε V to j ε V with intermediate vertices on the path chosen only from {1, 2…..,l} V. For l = O it is obvious. If it is true for l-1, then $a_{ij}^{(l)} = a_{ij}^{(l-1)} \mathrm{V} a_{ij}^{(l-1)} \cap a_{ij}^{(l-1)}$ is 1 if and only if either =1 (there is a path from I to j using only vertices in {1,2,……, l-1) or both $a_{il}^{(l-1)}$and $a_{il}^{(l-1)}$ are 1 (then one paths from i to l and l to j using only vertices {1, 2, ……., l-1}). Thus it is true for l.

3-Shortest Paths

Given a directed graph G = (V, E) let u_{ij} be a weight associated with each are (i, j) denoting the length of (i, j). Problem of finding the shortest path between two specified vertices with no repeated nodes are possibly the most fundamental and important of all combinatorial optimization problems.

There does not seen to be a really good method for finding the length of a shortest path from a specified origin to a specified destination without, in effect, finding the lengths of shortest paths from the origin to all other nodes. We shall, therefore, view the problem so such.

Let

u_{ij}= the finite length of the arc (i, j) if there is such an are = ∞ otherwise

u_j= the length of the shortest path form origin origin to node j.

If the origin is numbered 1, the rest of the nodes are numbered 2, 3, ……, n and there are no directed cycles of negative then $u_1 = 0$. For each node j, $j \neq 1$, there must be some final are (k, j) in the shortest path from 1 to j. from the principle of optimality if is clear that $u_j = u_k + u_{kj}$. since there are only a finite number of choices for k, i.e. k = 1, 2, ……, j-1, j+1, ….., n, k must be a node for which $u_k + u_{kj}$ is as small as possible, therefore the shortest path must satisfy the following equations.

$u_1 = 0$

$u_j = \min_{k \neq j} \{u_k + u_{kj}\}, (j = 2,3, \ldots, n)$

These equations are known as Bellman-Ford equations and are necessary and sufficient to determine the lengths of the shortest paths. If there are no positive cycles in the networks then it can be shown that the solution to the Bellman-Ford equations is unique. As the equation an non-linear the equation do not lend themselves to solutions as they stand. We shall discuss cases when the solution is easy to obtain and how the problems of non-linearity are overcome.

A situation which is easy to solve is when all the arc lengths are positive. $0(|V|^2)$ algorithm that will not be described is due to Dijkastra (1959), (Srivastava, 2016) Dijkstras algorithm labels the vertices of the given digraph. At each stage of the algorithm some vertices have permanent labels and the others temporary labels. Permanent label of a vertex represents the true length of the shortest path to the node. Temporary label represents an upper bound on the length of a shortest path.

Initially, the only permanently labeled node is the origin, which is given the label $u_1 = 0$; each of the other nodes j is given the tentative label $u_j = a_{ij}$. The general step is as follows. Find the tentatively labeled vertex k for which is u_k minimal. Declare k to be permanently labeled, and revise the remaining temporary labels u_j by comparing u_j with $u_k + a_{kj}$, and replacing u_j by the smaller of the two values. Procedure terminates when all nodes are permanently labeled.

The above method will not work if some of the edges have negative weights. This is because in the above method once a vertex gets a permanent label, its label cannot be altered. The method can be modified provided all the cycles have positive weight. The modification is the obvious one, i.e., at each iteration every vertex with a permanent label also gets a new temporary label if this temporary label turns out to be smaller than the final label. However, this is most inefficient. The algorithm which is discussed next is

duo to Floyd (1962) and find shortest paths between all pairs of vertices. It is strikingly similar to warshalls method for transitive closure and has the same complexity.

Given W = $[W_{ij}]$ the weight matrix of G = (V, E) we want to compute W* = [W_{ij}^{*}], in which W_{ij}^{*} is the length of the shortest path from I to j in G. Analogous to Warshalls method we define a sequence of matrices

$W^{l} = \left[w_{ij}^{(l)}\right]$ as follows.

$w_{ij}^{(o)} = w_{ij}$

$w_{ij}^{(l)}$ = mi ($w_{ij}^{(l-1)}, w_{ij}^{(l-1)} + w_{ij}^{(l-1)}$)

Assuming that the weight of a nonexistent edge is ∞. The value of $w_{ij}^{(l)}$ is the length of the shortest path from i to j with intermediate vertices chosen only from { 1, 2, ……., l} V.

An O (V^3) implemented on of the above idea is given below in a more formal language.

for l = 1 to $|V|$ do

for i = 1 to $|V|$ do

if $w_{il} \neq \infty$ then for j = 1 to $|V|$ do

$$w_{ij} - \min(w_{ij}, w_{il} + w_{ij})$$

4-Travelling salesman problem

There are a number of combinatorial problem requiring answers to questions as "list all possible….." "How many way are there to ……" which require exhaustive search of the set of all potential solutions.

Generating all spanning trees of a graph is one such example.

One general technique for organizing such a search is back tracking. Batch tracking works by continually trying to extend partial solutions. At each stage of the search, if an extension of the current partial solution is not possible, we "back track to a shorter partial solution and try again. This search procedure can be represented by a tree called to search tree. Let us assume that the solution to the problem consists of a vector $(a_1, a_2 \ldots\ldots)$ of finite but undetermined length satisfying certain constraints. The search tree representing a systematic search for such a solution would look as in figure 4.

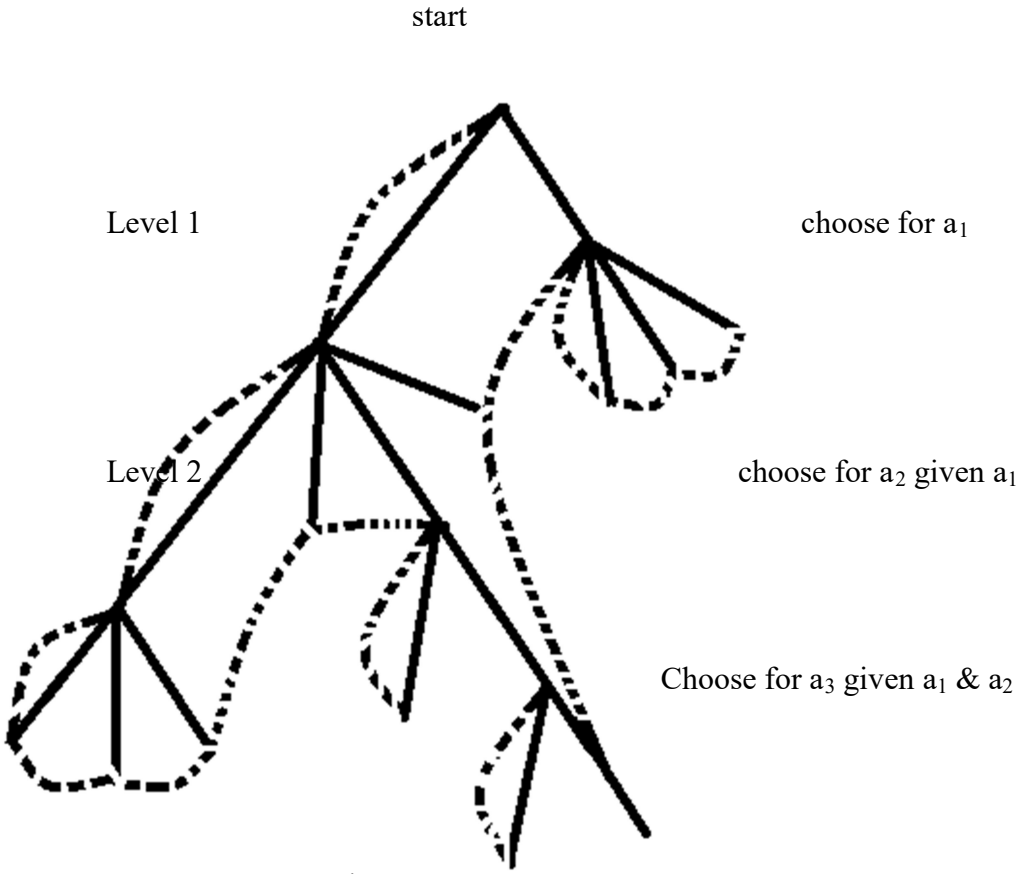


Figure 4

The root of the tree is the null vector. Its sons are the choices of a_1, and, in general the nodes at the level are the choices for a_k give the choice mode for $a_1, a_2, \ldots\ldots\ldots\ldots, a_{k-1}$ as indicated by the of these nodes, in the tree of figure 4 back track traverses are shown by dashed lines.

It should not be difficult to imagine that the search for solution if the solution lies somewhere on the search tree can become extremely time consuming if the search is not directed at each stage by using some information about the nature of the problem. One such method of "pruning" the search tree is called branching and bounding. This method of pruning or preclusion is based on the assumption that each solution has a cost associated with it and the optimal solution (the one of least cost is to be found). In order for branch and bound to be applicable the costs must be well defined on partial solution; and for all partial solutions $(a_1, a_2, \ldots\ldots\ldots a_{k-1}) \leq cost\ (a_1, a_2, \ldots\ldots, a_{k-1}, a_k)$.

When the cots have these properties then we can discard a partial solution $(a_1, a_2, \ldots\ldots\ldots a_{k-1})$ if its cost is greater than or equal to the cost of a previously computed solution.

Travelling salesman problem in a complete graph is one of the problems that can be solved with branch and bound techniques.

The algorithm that we now discuss for the travelling salesman problem is due to Little *et al.* (1953). It utilizes branching and bounding and illustrates the technique of tree re-arrangement such that near optimal solutions are found as early as possible. The tree re-arrangement technique used is to split the remaining solutions into two groups at each stage one including a particular are and those that exclude that are. The arc is chosen according to the heuristic given below.

∞	3	93	13	33	9	47
4	∞	77	42	21	16	34
45	17	∞	36	16	28	25
39	90	80	∞	56	7	91
28	46	88	33	∞	25	57
3	88	18	46	92	∞	7
44	26	33	27	84	39	∞

Figure 5

∞	3	93	13	33	9	47
4	∞	77	42	21	16	34
45	17	∞	36	16	28	25
39	90	80	∞	56	7	91
28	46	88	33	∞	25	57
3	88	18	46	92	∞	7
44	26	33	27	84	39	∞

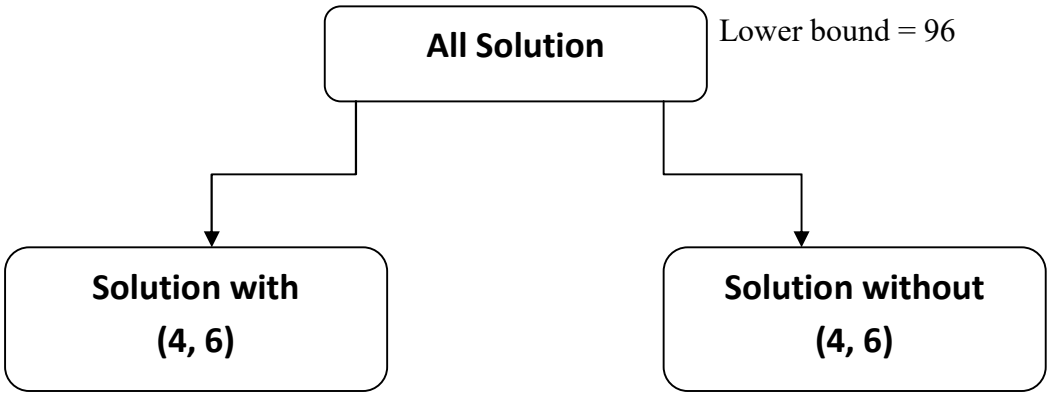
figure 6.

Consider the cost matrix of a 7 × 7 travelling salesman problem given in Figure 5. The heuristic uses the fact that if a constant is subtracted from any row or any column of the cost matrix, the optimal solution does not change. The cost of the optimal solution changes but not the path itself. The cost diminishes by the amount subtracted from the

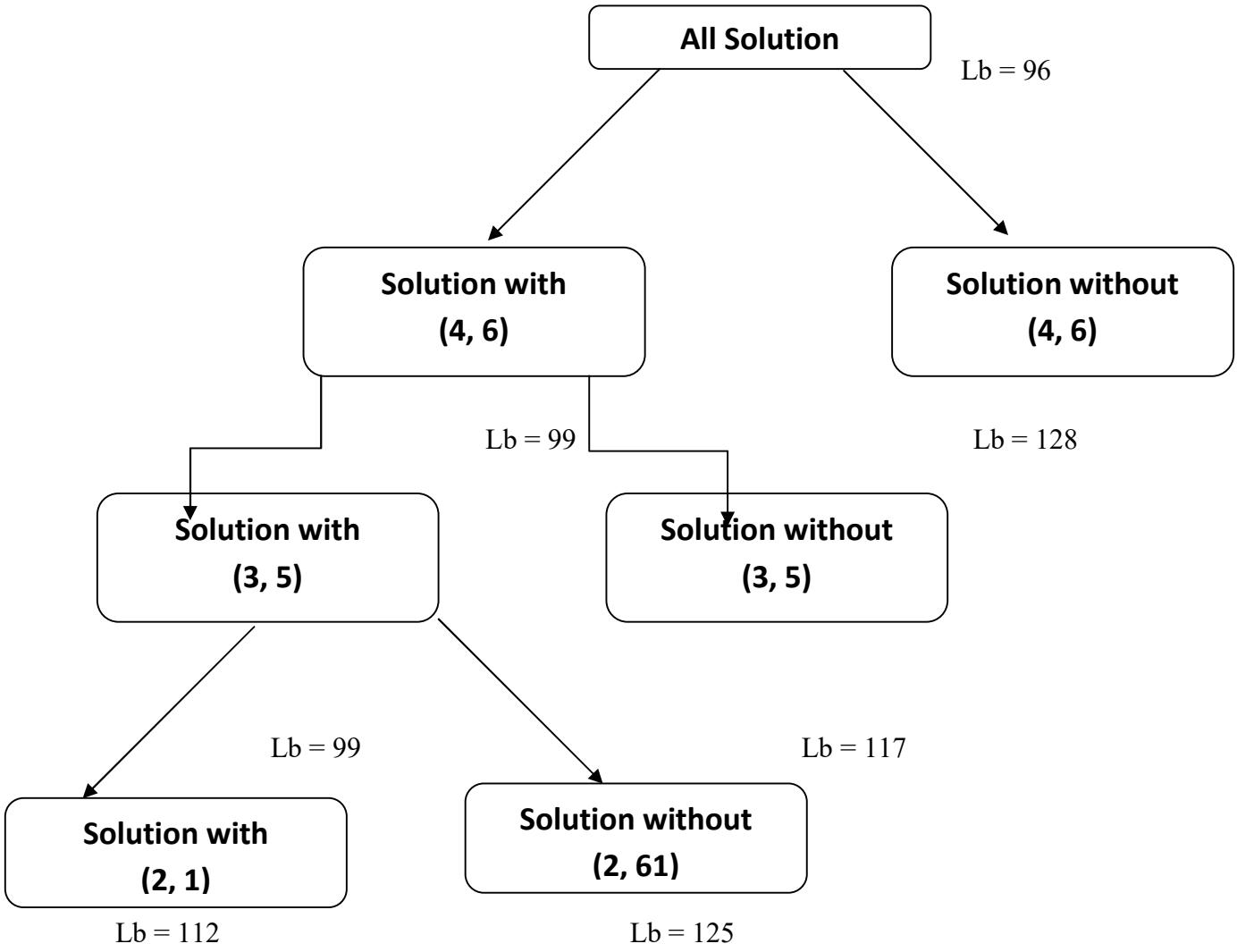
row or the column. If such a subtraction results in each row end column containing a zero with all the remaining non-negative then the total amount subtracted will be a lower bound on the cost of any solution. In the cost matrix of figure 5, 3, 4, 16, 7, 25, 3 and 26 can be subtracted from rows 1 through 7, and 7, 1, 4 from columns 3, 4 and 7 to result in the reduced matrix of A total of 96 was subtracted and so 96 is a lower bound on the cost of any solution. The root of the binary search tree, therefore, is labeled as follows:

Lower bound = 96

Suppose we choose they are (4, 6) to split the root. Right subtree will contain all solutions that exclude they are (4, 6) and so we can set w_{46} =∞. The resulting matrix can that have 32 subtracted from the fourth row, and thus the right subtree has a lower bound of 96 + 32 = 128. As the left subtree will contain all solution that include (4, 6) we can delete the fourth row and the sixth column. This is because we can never go form 4 to anywhere else nor arrive at 6 from anywhere else. The resulting matrix is of one less dimension. Also in this matrix (6, 4) is no longer usable and so we set w_{46} =∞. We can also subtract 3 from the fourth row of the resulting matrix giving the left subtree a lower bound of 96 + 3 = 99. The binary search tree at this stage would look like.



They are (4, 6) chosen to split the root because, of all arcs, it caused the greatest increase in the lower bound of the right subtree. This rule is used because we would prefer to find the solution following the left edges rather than the right edges. The left edge reduce the dimension of the problem whereas the right edges only add another ∞ to the matrix. At any stage we expand the node with the lowest lower bound. It should be noted that the first solution found need not be the optimal solution. This will be if some unexpanded node in the binary search tree has a lower bound which is smaller than the solution just found. The solution will be optimal if the lower bound of all unexpanded nodes is larger than the cost of the solution. A partial search tree for the travelling salesman problem discussed above is given below:



Experimental evidence indicates that the number of nodes examined is $0(1.26^n)$ for random n × n matrices.

Conclusions

Algorithmic graph theory is a fast-growing field. Number of good books has appeared. Some of these are listed in reference given below. What has been explored here does not eve probe the surfaces of the known results extensively. The interested reader will find it fruitful to go to the sources and discover for himself this wealth of knowledge.

References

1. Lawler, E. (1976). Combinatorial Optimistation; Networks and Matroids, Holt-Rineheart.
2. Kruskal, J.B. (1956). On the shortest spanning subtree of a Graph and the Travelling salesman Problem", Proc. American Math. Soc., 7: 48-50.
3. Minkty, G.J. (1965). A Simple Algorithm for listing All the Trees of a Graph", I EEE Trans. Circuit Theory, 12: 120.
4. Warshall, S. (1962). A Theorem on Boolean Matrices, J. ACM, 9:11-12.
5. Dijksta, E. (1959). Two Problems in Connexion with Graphs", Num. Math., 1: 269-271.
6. Floyed, R.W. (1962). Algorithm 97: Shortest Path, Comm. ACM, 5:345.
7. Little, J.D.C., K.G. Murty, D.W. Sweency, and O., Karel (1963). An Algorithm for the Travelling Salesman Problem", Operations Research, 11:977-989.
8. Reingold, E.M., J. Nievergelt, and N. Deo (1977). Combinatorial Algorithms", Preatice-Hall.
9. Christifides, N. (1974). Algorithmic Graph Theory", Academic Press.
10. Srivastava, S.P. (2010). Study of Dirac and Charval Theorem for Hamiltonian Graph, Research Analysis and Evaluation, 1(14 November);112-115
11. Srivastava, S.P. (2016). Study of Different algorithm in Euler Graph, International Journal on Recent and Innovation Trends in Computing and communication, 4(12): 308-311.

Received on 18.8.2018 and accepted on 20.9.2018