A Fast Scheduling Algorithm in AND-OR Graphs

GEORGE M. ADELSON-VELSKY
Department of Mathematics and Computer Science,
Bar-Ilan University, Ramat Gan, ISRAEL

ALEXANDER GELBUKH
Center for Computing Research,
National Polytechnic Institute, Mexico City, MEXICO

EUGENE LEVNER
Department of Computer Science,
Holon Institute of Technology, Holon, ISRAEL

Abstract. We present a polynomial-time algorithm for scheduling tasks in AND-OR graphs. Given the number p of arcs and n of nodes, the complexity of the algorithm is $O(np)$, which is superior to the complexity of previously known algorithms.

Key-Words: AND-OR graphs, scheduling, routing, polynomial-time algorithms.

1 Introduction

We consider a task scheduling problem in weighted directed AND-OR graphs in which arcs are identified with tasks while nodes represent their starting and finishing endpoints. A starting point of a task is represented by an *AND-node* if its execution can be started after all its preceding tasks have been solved, and by an *OR-node* if it can be started as soon as any one of its preceding tasks is solved. The time needed to execute a task is represented by an *arc length*. The problem that emerges is to implement all the tasks in the graph in minimum time.

Scheduling problems in AND-OR graphs have many real-world applications. For example, De Mello and Sanderson [5] have applied the scheduling problems for the planning of robotic assembling systems; Gillies and Liu [8] and Adelson-Velsky and Levner [1] employed AND-OR graphs for real-time scheduling of tasks in computer communication systems. Along with various technological applications, the scheduling problems in AND-OR graphs arise in mathematical analysis of extremal problems in context-free grammars [9], hypergraphs [4], and games [11].

The problem considered generalizes the classical shortest-path and critical-path problems in graphs. While vast literature is devoted to the shortest-path problem and the critical-path problem in standard graphs (see, e.g., [3] and the numerous references therein), not much have been done for efficiently solving the path-finding problems in AND-OR graphs. A special case in which arc lengths are strictly positive has been elegantly solved by Dinic [6]. Another special case—in which AND-OR graphs are bipartite, arc lengths are non-negative, and the arcs leading to the OR-nodes have zero lengths—has been investigated by Adelson-Velsky and Levner [1, 2]. Independently, Mohring *et al*. [10], studied a subcase in which only one arc led from each OR-node. The latter two algorithms are of the same complexity $O(pp')$, where p is the total number of arcs and p' the number of arcs entering AND-nodes.

In this paper, we extend the polynomial-time algorithms of [1, 2, 10] to the general (i.e., non-bipartite) AND-OR graphs with non-negative arc lengths and at the same time improve their complexity. Given the total number n of nodes, the complexity of the new algorithm is $O(np)$, which is superior to the complexity of the previous algorithms.

The paper is organized as follows: In Section 2 we define the problem. Section 3 presents a new polynomial-time algorithm. Section 4 analyzes its properties. Section 5 concludes the paper.

2 Problem Formulation

The input to the scheduling problem under consideration is $<G, s, t>$, where $G = (V, E)$ is a directed graph, V is the node-set, $|V| = n$, E is the arc-set, $|E| = p$; $t = t(v_i, v_j)$ is an arc length function, and s is a node called the *start* whose occurrence time is given by $t(s) = t_0$.

We assume that $V = A \cup O \cup \{s\}$, A being the set of AND-nodes and O the set of OR-nodes. The problem is to find the earliest *starting times* $t(v_j)$, for all $v_j \in V$, satisfying the following conditions:

$$t(s) = t_0, \tag{1}$$

$$t(v_j) \geq \max_{v_i \in P(v_j)} (t(v_i) + t(v_i, v_j)) \text{ if } v_j \in A, \tag{2}$$

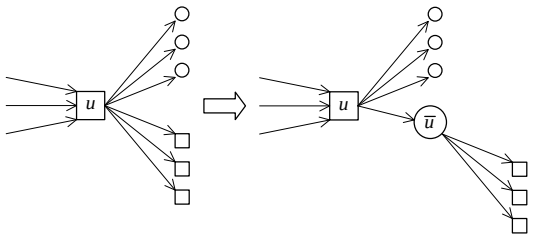
$$t(v_j) \geq \min_{v_i \in P(v_j)} (t(v_i) + t(v_i, v_j)) \text{ if } v_j \in O, \tag{3}$$

$$t(v_j) \geq t_0, \text{ for all } v_j. \tag{4}$$

Here $P(v)$ denotes the set of nodes that are immediate predecessors to v. Without the loss of generality, we assume that $P(v)$ is non-empty for any node v, $v \neq s$ (otherwise, we would have several start nodes which could be glued together into a single start node). We will denote this problem by **P**. The problem turns into the critical path problem if O is empty, and into the shortest path problem if A is empty.

Conditions (2) and (3) are represented in the graph G by the arcs from v_i to v_j of length $t(v_i, v_j)$. We start the solution process with the following graph transformations that permit us to present the constraints (4) in graph form (this does not violate the problem size order):

(a) if the graph has an OR-node u having immediate successors v_j of OR type, then we add a new AND-node $\bar{u}$ with $t(u, \bar{u}) = 0$, and the arcs (u, v_j) are replaced by the arcs $(\bar{u}, v_j)$ with $t(\bar{u}, v_j) = t(u, v_j)$ (see the figure below, where the squares denote OR-nodes and the circles AND-nodes);



(b) we add arcs (s, v_j) of zero length leading from the start s to all AND-nodes v_j in G (including those added in the previous transformation).

Due to these preliminary transformations, we may assume that any OR-node in G has a preceding AND-node or s; in particular, G does not contain zero-length cycles consisting only of OR-nodes. Note that in contrast to the graph model considered in [1, 2], in this paper the arcs entering OR-nodes are allowed to be of non-zero length.

DEFINITIONS. A set of values $\{t(v_j)\}$, $j = 1, \ldots, n$, satisfying inequalities (1) to (4) is called a *feasible solution* to the problem **P**. The feasible solution providing the minimum values $t(v_j)$ for all v_j among all feasible solutions, is called *optimal*, or *earliest, starting times*, and is denoted by $\{t^*(v_j)\}$.

Denote the graph obtained after the transformations by $\tilde{A}$, and the problem of finding the optimal occurrence times $\{t^*(v_j)\}$ in the new graph $\tilde{A}$ subject to (1) to (4) by **Đ**. Obviously, the problems **P** and **Đ** are equivalent.

3 Algorithm

The new algorithm is based on the previous algorithm suggested by the first and third authors of this paper in [1,2], differing from the latter in the following two aspects: (i) it uses another labeling procedure which permits us to improve the algorithm complexity; (ii) it uses a refined graph reduction procedure which permits to treat general AND-OR graphs rather than only bipartite graphs.

Our algorithm works iteratively. At each iteration, the algorithm finds a node with the smallest starting time (at this point, the algorithm is similar to the classic Dijkstra algorithm). However, in contrast to Dijkstra's [7], Knuth's [9] or Dinic's [6] algorithms, our algorithm is *not* greedy: it first discovers and labels all nodes with *not-minimal* current starting times, and only after that it reveals that the remaining (i.e., not-yet-labeled) nodes gain minimum-time labels. At termination, the algorithm either provides the minimal starting time $t^*(v_j)$ for all nodes, or announces that the problem **Đ** has no feasible solution.

For every node $v \in V$, the algorithm assigns a time label $t(v)$ and maintains a status $St(v) \in \{$*uncolored, red, black*$\}$. All nodes start out *uncolored* and later become *red* or *black*. Initially, $t(s) = t_0$.

At each iteration, the graph is reduced to a smaller one. Let $\tilde{A}_h$ denote the graph derived at the end of the h-th iteration (h = 1, 2, …).

Each iteration consists of four procedures: Node_Painting, Sorting, and Graph_Reduction.

The main idea behind the painting procedure is to guarantee that in the $\tilde{A}_h$ all nodes painted *red* will have the time labels (i.e., the occurrence times) *greater* than the earliest occurrence time in $\tilde{A}_h$; the nodes labeled *black* will have the time labels $t(v)$ *equal* to the *earliest* occurrence time among the nodes of $\tilde{A}_h$.

Consider an iteration h. Let $\mathbf{F}(s)$ denote the set of all nodes v which are the heads of the arcs leaving s.

First (before the first iteration starts) we sort all nodes v_i in $\mathbf{F}(s)$ in non-decreasing order of their weights $t(s,v_i)$. In addition, for each OR-node, v_j, we compute its in-degree and assign it to a variable r_j.

Next, we use Node_Painting consisting of two steps, S1 and S2. During these operations, yet *uncolored* nodes are painted *red*, until at some instant no *uncolored* node can be labeled *red*.

Step S1. For each positive-length arc (v_i,v_j), $v_i \neq s$, consider its head v_j. If it is an AND-node then paint it *red*. If it is an OR-node then decrement r_j by 1; when r_j becomes 0, paint v_j *red*.

Step S2. For each zero-length arc (v_i,v_j) such that its tail v_i is *red*, consider its head v_j. If it is an AND-node then paint it *red*. If it is an OR-node then decrement r_j by 1; when r_j becomes 0, paint v_j *red*. Note that when a node is painted *red*, all its leaving arcs of zero length are added to a queue to be processed in the same manner later at this step.

Two cases are possible at this point:

Case **C1**. All nodes are painted *red*. This means that the initial graph has a cycle of positive length, and moreover, the time labels of some nodes in the cycle will be infinitely large. In this case, the problem has no feasible solution, so the algorithm stops.

Case **C2**. Some nodes (or, possibly, all of them) are not painted *red*. Consider this case in detail.

Note that all nodes must be reachable from the *start* s through directed paths in G. Indeed, for the AND-nodes this trivially follows from the preliminary transformation (b) described above; for the OR-nodes it follows from the fact that any such node is preceded by an AND-node or s (this follows from the transformation (a)). Therefore, $\mathbf{F}(s)$ contains unpainted nodes.

Among the unpainted nodes in $\mathbf{F}(s)$, we choose the node v^{**} with the maximal length $t(s,v_i)$: $t^{**} = t(s,v^{**}) = \max_{i\in \mathbf{F}(s)} t(s,v_i)$. If v^{**} is of AND type then we paint it *red*. If v^{**} is of OR type then we decrement its $r_{v^{**}}$ by 1. If the $r_{v^{**}}$ becomes 0 then we paint v^{**} *red*. Otherwise, i.e., if $r_{v^{**}} > 0$ then the arc (s, v^{**}) is removed (i.e., v^{**} is removed from $\mathbf{F}(s)$) since this arc is not critical.

If the node v^{**} was painted *red*, then we paint not-yet-painted nodes, as described in Step S2, starting from the node v^{**}.

If in $\mathbf{F}(s)$ there are still not painted nodes, then the operation described in the previous two paragraphs is executed with the next maximal node in $\mathbf{F}(s)$, i.e., choose the (next) maximal not-yet-painted node in $\mathbf{F}(s)$, paint it or decrease its $r_{v^{**}}$, and apply Step S2. This descend is repeated until all nodes in $\mathbf{F}(s)$ become painted *red*.

When all nodes in $\mathbf{F}(s)$ become *red*, we select the last node considered as v^{**} and denote it as v^*. It has the minimum time $t(v^*)$ in $\tilde{A}_h$. This node is painted *black* (in order to simplify the further proofs). This finishes the procedure Node_Painting.

Then Graph_Reconstruction is applied: The node v^* and the arcs incident to it are removed from $\tilde{A}_h$. Each immediate successor v of v^* is made connected with s by an arc (s,v) whose length is defined as follows. If there is no arc (s, v), then such an arc is inserted in the graph (instead of the removed arc (v^*, v)), with

$$t(s, v) := t(s, v^*)+\tau(v^*, v).$$

Otherwise, the length of the arc (s, v) is recalculated:

$$t(s, v) := \max(t(s, v),\ t(s, v^*)+\tau(v^*, v))$$
if v is an AND-node;

$$t(s, v) := \min(t(s, v),\ t(s, v^*)+\tau(v^*, v))$$
if v is an OR-node.

As a result, some new nodes may enter $\mathbf{F}(s)$ and others may change their position in the ordering of $\mathbf{F}(s)$. This finishes Graph_Reconstruction.

If after the reconstruction the resulting graph is empty, the algorithm finishes. All nodes have been removed from the graph, with their time labels having been defined. Otherwise, a new iteration (starting from Step S1) is executed with the reduced graph, with all its nodes made again unpainted.

4 Algorithm Analysis

At each iteration, a node is deleted from the graph, so that the number of nodes (and arcs) in the graph $\tilde{A}_h$ decreases by one. Though the number of nodes in **F**(s) changes dynamically from iteration to iteration, it never exceeds the total number n of nodes.

To summarize, after each iteration, the following states are possible:

(1) either all of the nodes are painted *red* after the steps S1 and S2 of Node_Painting; then the problem has no solution, or

(2) all nodes of the initial graph G have been removed (so that their minimum times have been defined); it means that the problem has been solved, or

(3) after Graph_Reconstruction terminates, the graph is not yet empty; then the algorithm goes to the next iteration.

Observe that this construction does not mimic the structure of Dijkstra's shortest-path algorithm.

Consider the properties of the algorithm in more detail.

DEFINITIONS. Let $t(v_j)$ be the optimal solution to the problem **Ð**. Any arc (v_i, v_j) such that $t(v_j) = t(v_i) + t(v_i,v_j)$ where $v_i \in P(v_j)$, is called *critical*, or *binding*. A path is called *critical* if it either consists of a single node or consists of critical arcs. Given the optimal solution to the problem **Ð** and a node v_j, a *critical sub-network* $N_{cr}(v_j)$ originating at v_j is defined recursively as follows: (1) $v_j \in N_{cr}(v_j)$, and (2) if $v_k \in N_{cr}(v_j)$ and $t(v_j) = t(v_i) + t(v_i,v_j)$, then $v_i \in N_{cr}(v_j)$ and $(v_i, v_k) \in N_{cr}(v_j)$.

We will see that the critical path in the problem **Ð** may be neither the shortest nor the longest path in $\tilde{A}$.

The following theorem establishes the relationships between the critical paths and the optimal values of the decision variables, $t^*(v_j)$.

THEOREM 1 [2]. 1. *If $v_i \in N_{cr}(v_j)$, then there exists a simple (i.e., acyclic) critical path $L=(v_i,\ldots,v_j)$, starting at v_i and terminating at v_j, in which all nodes and arcs belong to $N_{cr}(v_j)$.*

2. *The length $\ddot{e}(v_i,\ldots,v_j)$ of the critical path L, defined as $\sum_{s=i}^{j-1} t(v_s, v_{s+1})$, equals $t^*(v_j) - t^*(v_i)$.*

3. *If the problem **Ð** has a feasible solution, then the starting node s belongs to all critical sub-networks $N_{cr}(v_j)$, $j = 1,\ldots, n$.*

The proof, which follows directly from the definitions of the critical sub-network and the critical path, is skipped here. (The detailed proof can be found in [2]).

Consider now an iteration h. Assign the labels {*red, black*} to the nodes of $\tilde{A}_h$ as described in the previous section, and define the ranks of the *red*-labeled nodes as follows:

The nodes that are labeled *red* during the initial labeling receive the rank $k = 1$. When a node v is painted red at Step S2 using an arc (u,v), then the rank of v is set to the rank of u plus 1. The following two lemmas establish useful properties of the *black*- and *red*-labeled nodes.

LEMMA 1 ("The time of any *red*-labeled node is not minimal"). *If the optimal solution to Problem **Ð** exists, then the earliest occurrence time of any red-labeled node is greater than $t_0 + t^* = t_0 + \min_{i\in F(s)} t(s, v_i)$, where **F**(s) is the set of all heads of the arcs leaving the start node s in $\tilde{A}_h$..*

Proof is by induction on the rank k. Consider any iteration, say h. Let $\ddot{E}_k$ denote the set of all *red*-labeled nodes of the rank k at that iteration.

Let us first verify the result for $k = 1$. For the nodes $v_i \in \ddot{E}_1$, we have:

If $s \in P(v_j)$, and $t(s, v_j) > t^* = t(s,v_1) = \min_{i\in F(s)} t(s, v_i)$, then $t(v_j) \geq t(s) + t(s,v_j) > t_0 + t(s,v_1)$.

If $v_i \in P(v_j)$, and $t(v_i, v_j) > 0$, then $t(v_j) \geq t(s) + t(s,v_1) + t(v_i, v_j) > t_0 + t(s,v_1)$.

Suppose that the required result is true for all the ranks not greater than k, that is, for the earliest occurrence times of the nodes v_i from $\cup_{s\leq k}\ddot{E}_s$, $t(v_i) > t_0 + t(s,v_1)$.

Let $v_j \in \ddot{E}_{k+1}$. Then we have:

If $v_j \in A$ and $v_i \in P(v_j) \cap \ddot{E}_k$ then $t(v_j) \geq t(v_i) + t(v_i, v_j) \geq t(v_i) > t_0 + t(s,v_1)$;

If $v_j \in O$ and $v_i \in P(v_j) \cap (\cup_{s\leq k}\ddot{E}_s)$, then $t(v_j) = \min_{v_i\in P(v_j)} t(v_i) > t_0 + t(s,v_1)$. ð

Remark (the structure of the sets of predecessors for the *black*-labeled nodes). Let $\boldsymbol{U}$ be the set of the *black*-labeled nodes, and v_j is a node from $\boldsymbol{U}$. Then:

(1) If $v_j \in A$ then $P(v_j) \subset \boldsymbol{U}\cup\{s\}$ (i.e., all immediate predecessors to any *black*-labeled AND-node are *black*-labeled, or coincide with the starting node s).

(2) If $s \in P(v_j)$ then $t(s, v_j) = t^* = t(s,v_1) = \min_{i\in \mathbf{F}(s)} t(s, v_i)$ (i.e., if the starting node s enters $P(v_j)$ then

the arcs leading from it to the *black*-labeled AND-nodes are of minimally possible length).

(3) If $v_i \in P(v_j) \cap O$, and $v_j \in A$, then $t(v_i, v_j) = 0$ (i.e., if an immediate predecessor to the *black*-labeled AND-node is *black*-labeled then the arc linking these two nodes is of zero length).

(4) If $v_j \in O$ then $P(v_j) \cap U \neq \varnothing$ (i.e., among the immediate predecessors to any *black*-labeled OR-node, there is a *black*-labeled node).

Indeed, if any one of the conditions above does not hold, then the node v_j will be labeled *red*.

LEMMA 2 ("The time of any black-labeled node is minimal"). Let $\tilde{A}_h$ be a graph obtained at the *h*-th iteration of the algorithm. If a feasible solution to **Ð** in graph $\tilde{A}_h$ exists, then the earliest occurrence time of any black-labeled node is equal to $t_0 + \tau^* = t_0 + \tau(s,v_1) = t_0 + \min_{i \in \mathbf{F}(s)} \tau(s, v_i)$.

Proof. Let $\boldsymbol{t} = \{t(v_j)\}$ be a feasible solution to Problem **Ð** in graph $\tilde{A}_h$, and consider a new solution, $t' = \{t'(v_j)\}$, defined by $\boldsymbol{t}$ as follows:

$$t'(v_j) = \begin{cases} t(v_j), & \text{if } v_j \in \cup \ddot{E}_k, \\ t(s, v_1), & \text{if } v_j \in U. \end{cases}$$

Consider any node v_j labeled *red*. Due to Lemma 1, $t(v_j) > t_0 + t(s,v_1)$, the times $t'(v_i)$ for $v_i \in P(v_j)$ are not greater than $t(v_i)$, and, hence, the occurrence times $t'(v_i)$ of the *red*-labeled nodes satisfy conditions (1)-(4).

Consider now the nodes labeled *black*. In view of Remark about the structure of the predecessors for the *black*-labeled nodes, we have, for nodes $v_j \in A$:

$$t'(v_j) = \max_{v_i \in P(v_j) \subset O \cup \{s\}} (t'(v_i) + t(v_i, v_j)) =$$

$$\left\{ \begin{array}{l} t(s) + t(s, v_1) + 0 \\ (\text{if } v_i \in P(v_j) \subset O), \\ t(s) + t(s, v_j) + t(s) + t(s, v_1) \\ (\text{if } P(v_j) = \{s\}), \\ \max(t(s, v_1) + 0, t(s) + t(s, v_j)) \\ (\text{if } P(v_j) \cap O \neq \varnothing, \text{ and } s \in P(v_j)) \end{array} \right\} =$$

$$t_0 + t(s,v_1).$$

Next, for the nodes $v_j \in O$, we have $P(v_j) \cap \boldsymbol{U} \neq \varnothing$, and, in view of the condition $t(v_i) > t(s,v_1)$, which is proved for the *red*-labeled nodes v_i, we obtain:

$$t'(v_j) = \min_{v_i \in P(v_j)} t'(v_i) =$$

$$\min (\{t'(v_i) = t_0 + t(s,v_1)\} \mid_{v_i \in P(v_j) \cap U}, \{t'(v_i) = t(v_i) > t_0 + t(s,v_1)\} \mid_{v_i \in P(v_j) \setminus (P(v_j) \cap U)}) = t_0 + t(s,v_1).$$

Therefore, if, in a feasible solution, some of the values $t(v_i)$ are greater than $t(s,v_1)$, then this solution is not optimal due to the uniqueness of the optimal solution to Problem **Ð**. The claim is proved.

The suggested algorithm is a modification of the algorithm in [1, 2] that solved the above problem (1)-(4) in a special case of bipartite graph with some specific requirements to the arc lengths. Our aim is to extend that algorithm for the general AND-OR graphs with cycles. Similar to the algorithm in [1, 2], the new algorithm operates with two main procedures, Node_ Labeling and Graph_Reduction. The Node_Labeling - just as in [1,2] - has the complexity O(*p*). However, after each run of Node_Labeling at least one *node* of *G* may be removed, so the total number of runs of the internal cycle can reach *n*, where *n* is the number of nodes in *G*; this is a point of departure from the algorithm in [1, 2], where O(*p*) iterations is required in the internal cycle in the worst case. By this way, we are able to improve the algorithm complexity.

THEOREM 3. *The complexity of the algorithm is O(np).*

Proof. We first estimate how many operations each iteration *h* requires. For each node v_i in $\tilde{A}_h$, the input contains the list of arcs (v_i, v_j) leaving the node. In addition, for each OR-node, v_j, we compute its in-degree r_j.

At step S1, the labeling procedure includes examining all arcs leaving the *start* node and all arcs of positive length in $\tilde{A}_h$. The required time is O(*p*) (*p* = the number of all arcs).

For Step S2 of Node_Painting (in total, during all its invocations at the given iteration *h*), the labeling procedure can be implemented by examining each arc in $\tilde{A}_h$ not more than *once*. Indeed, at each run of Step S2, the algorithm will examine in turn each arc, say (v_i, v_j), that leaves a *red*-labeled node v_i (and that has not yet been examined). During each examination, for an OR-node the algorithm will update the value r_j for the head-node v_j decrementing it by 1, when the new $r_j = 0$, the node v_j is painted *red*; for an AND-node, the node will be painted. Having been once examined at some run of Step S2, the arc (v_i, v_j)

is not examined anymore at further runs of this step at the considered iteration. Thus, at any given iteration, during all repetitions of Step S2, the labeling procedure takes O(p) operations.

At each iteration, graph $\tilde{A}_h$ is reduced by one (*black*) node so that the total number of iterations is O(n), where n is the number of nodes in the initial graph G. Thus, the overall time of the labeling procedure during all iterations is O(np).

Now we estimate how many operations the reconstruction of the graph requires. It is sufficient to scan only the arcs leading to the immediate successors of the nodes from the set **F**(s), and each arc is examined only once; their number is at most O(p). Since the total number of iterations is n, the graph transformations by Graph_Reduction add at most O(np) operations.

When a new node appears in **F**(s) or a node changes its position in the ordered **F**(s), the order can be maintained in O(n) operations (and even in O(log n) if to use the balanced-tree data structure). This may happen only when an arc is removed from the graph in Graph_Reduction, i.e., it happens at most p times. Thus, the total cost of maintaining **F**(s) ordered is not greater than O(np). Therefore, the total complexity of the suggested algorithm is O(np). The claim is proved.

The algorithm can be made faster if at some stage the reduced graph becomes acyclic, or if all arc lengths in the reduced graph become positive.

5 Concluding Remarks

When arc lengths in the AND-OR graph in the considered scheduling problem are of arbitrary sign, the proposed algorithm is inapplicable. Although it is easy to construct a dynamic programming algorithm that yields the optimal solution for this problem in pseudo-polynomial time, the question whether the problem is polynomial solvable, is still open. A challenging problem would be to find a polynomial-time algorithm for this scheduling problem, or, otherwise, to prove its polynomial unsolvability.

Acknowledgement. The work was done under partial support of INTAS (Brussels, Belgium) for the first and third authors, and SNI and CONACyT (Mexico) for the second author.

References

1. Adelson-Velsky, G.M., and E. Levner (1999). Routing information flows in networks: A generalization of Dijkstra's algorithm, *Proceedings of the International Conference "Distributed Computer Communication Networks"*, November 9-13, 1999, Tel-Aviv University, Israel, Tel-Aviv - Moscow, IPPI RAN Press, pp.1-4.
2. Adelson-Velsky, G.M., and E. Levner (1999). *Finding extremal paths in AND-OR graphs. A generalization of Dijkstra's algorithm.* Technical report, Holon Academic Institute of Technology, Holon, Israel, 35 pp.
3. Ahuja, R.K., T.L. Magnanti, J.B. Orlin (1993). *Network Flows. Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs.
4. Ausiello, G., A. D'Atri, D. Sacca (1983), Graph algorithms for functional dependency manipulation, *Journal of ACM*, 30, 752-766.
5. De Mello, L.S.H., and A.C. Sanderson (1990). AND/OR graph representation of assembly plans, *IEEE Transactions on Robotics and Automation*, vol. 6, no.2, 188-199.
6. Dinic, E.A. (1990). The fastest algorithm for the PERT problems with AND- and OR-nodes. *Proceedings of the Workshop on Combinatorial Optimization*, Waterloo, University of Waterloo Press, Waterloo, 185-187.
7. Dijkstra, E.W. (1959). A note on two problems in connexion with graphs, *Numerische Mathematik*, 1, 269-271.
8. Gillies, D. and J. Liu (1995), Scheduling tasks with AND/OR precedence constraints, *SIAM Journal on Computing* 24(4), 787-810.
9. Knuth, D. (1977), A generalization of Dijkstra's algorithm, *Information Processing Letters*, 6,1-5.
10. Mohring, R.H, M. Skutella and F. Stork (2000), *Scheduling with AND/OR Precedence Constraints*, Technical Report No. 689/2000, Technische Universitat Berlin, August 2000, 26 pp.
11. Zwick, U., and M. Patterson (1996), The complexity of mean payoff games on graphs, *Theoretical Computer Science*, 158, 343-359.