An $O(|V||E|)$ Algorithm for AND/OR Scheduling Problem: General Non-Negative Weight Case

Paz Carmi[*] Yefim Dinitz[†] Shahar Golan[‡] Guy Rozenwald[§]

May 1, 2006

Abstract

The paper is devoted to scheduling with AND/OR precedence constraints. The events are nodes of a certain graph, while the precedence relation is defined by its edges. The non-negative edge weights represent delays. An event of type AND may happen only after *all* its preceding events have happened, with the corresponding delays after them. An event of type OR may happen only after *at least one of* its preceding events has happened, with the corresponding delay after it. The early schedule is in question. We present an algorithm for the general AND/OR scheduling problem, where cycles consisting of zero weighted edges only are allowed and the precedence graph is arbitrary. The running time is $O(|V||E|)$, for a graph $G = (V, E)$.

[*]Department of Computer Science, Ben-Gurion University of the Negev, POB 653, Beer-Sheva 84105 Israel. `E-mail: carmip@cs.bgu.ac.il`.

[†]Department of Computer Science, Ben-Gurion University of the Negev, POB 653, Beer-Sheva 84105 Israel. `E-mail: dinitz@cs.bgu.ac.il`.

[‡]HyperRoll LTD, Omer, Israel. `E-mail: golansha@hotmal.com`.

[§]`E-mail: guyrozenwald@hotmail.com`.

1 Introduction

The classical PERT scheduling problem may be described as follows. There is a directed graph with a non-negative edge weight function d. Its nodes represent events, and its edges represent the precedence relation between events: if there is an edge from a node u to a node v (we say: *u precedes v*), then the event v may happen not earlier than by the delay $d(u, v)$ after u happens. The problem is to find the earliest non-negative times for all events which obey the precedence relation; they are called the *early times*, for events. It is accepted that if it is impossible to find a feasible time, for some event, then the time *infinity* is assigned to it. The classic motivation for PERT is finding the early time schedule for a large project, where the nodes are intermediate events (e.g., the starting or finishing time of a project component).

The basic case is the case of positive weights. A linear time algorithm for this case, using so called topological sorting of nodes, is one of the classic computer science algorithms. For the basic case, it is well known that a finite solution exists if and only if the precedence graph is acyclic. For a graph with cycles, all nodes lying on a cycle are assigned the infinite early time. For the case when edge weights are non-negative but each cycle has a positive weight, the same algorithm works. For the general case, when cycles consisting of zero weight edges only (henceforth called "zero cycles") are allowed, the algorithm must be extended.

The PERT problem is generalized to the AND/OR case in [5, 4], as follows. Any node with precedence conditions as above is considered as an *AND node*. Another type is an *OR node*, v, such that for *at least one* node u preceding it, the time of v is not earlier than by $d(u, v)$ after that of u.

A natural motivation for this AND/OR setting of PERT is the following New-Product-New-Technology problem [5, 4]. There are new technologies, where each one supplies certain new products to the market. The event represented by an AND node is launching of a new technology; this needs presence of certain new products at the market. The event represented by an OR node is the first appearance of the corresponding new product at the market, produced by some new technology. The edge weights are the corresponding minimal delays. Early schedule of this technology-product system is in question. (The relaxing assumption is that the volume of any new product, at the market, is not taken into account.) Suppose that some new technologies have a-priory lower time bounds. This is modeled by introducing an auxiliary AND node TIME-0, which has no predecessors and hence is assigned time 0, and the properly weighted edges from it to the corresponding AND nodes. Suppose that a new product can be obtained, at a certain time moment, without a help of any new technology (e.g., by exporting it). This a-priory upper time bound is modeled by using a similar auxiliary AND node EXPORT, and a properly weighted edge from it to the corresponding OR node. Note that in this application, the graph is bipartite, except for the node TIME-0, while in general, the precedence graph may be arbitrary (e.g., the classic PERT has AND nodes only).

The sense of a zero delay precedence is “not earlier than”; these precedences are common, in input data. Specifics of such precedence is a certain simultaneity: in some cases, for a group of events with mutual zero delay relations between them, all these events must happen simultaneously.

More motivation and applications for the AND/OR scheduling problem can be found in [6, 2, 7]. In the latter paper, a wide investigation related to this problem is made.

By definition, the pure case of AND nodes only is PERT. It is interesting that the case of OR nodes only, except for the “source” AND node without incoming edges, is another corner-stone optimization problem: finding the shortest path lengths from the source node to the other nodes. The basic algorithm for it, due to Dijkstra, finds the optimal solution if the graph has no zero cycle. It runs in time quadratic in the number of nodes, while using appropriate data structures leads to its versions with almost linear time bounds, in the size of the precedence graph (see e.g., [3]).

The two above algorithms are of somewhat similar and of somewhat different nature; both their common structure and their distinct approaches are studied in all basic courses in Algorithms. Surprisingly, for the hybrid AND/OR problem as above, there exists a fast algorithm, which is composed of these two classic algorithms and runs in the summary time. Such a hybrid algorithm is suggested in [5, 4]. This algorithm is also correct if the precedence graph has no zero cycle. This paper presents the correctness proof for this algorithm (it lacks in [5, 4]).

A similar algorithm was suggested earlier by Knuth [6] (the algorithm of [5, 4] was found independently). However, types of the problems considered in [5, 4] and [6] are not the same. Mutual reductions are not always possible, and if possible, do not preserve the correctness proofs and the time bounds.

The question of existence of a polynomial time algorithm for the general case of the AND/OR problem, when zero cycles are allowed, was open for about a decade. Polynomial algorithms for its *special settings* were suggested in [2] and in [7]. Let us denote the number of nodes by n, and that of edges by m. The algorithm of [2] runs in time $O(m^2) = O(n^4)$. For algorithm [7], the time bound is $O(nm) = O(n^3)$. As mentioned by E. Levner, it is possible to make a reduction from the general case to the special case considered there, and this implies the $O(m^2) = O(n^4)$ time bound for the general case.

Our algorithm solves the AND/OR problem in the general case, and runs in time $O(nm)$. It was formulated and proven at the course Advanced Algorithms, given by the second author to the rest of the authors and other students, in the spring 2001. Another algorithm, for the general case and with the same time bound, is suggested in [1]. We would like to mention that it was found after our time bound has been announced to one of its authors. Besides, the paper [1] has some lacks. More discussion on specifics in the previous papers, justifying relevance of our paper, see in Section 6.

2 Background

Let us give the formal problem setting. There is a directed (precedence) graph $G = (V, E, d)$ given, where d is a non-negative function on the edge set E. The node set V is divided into the non-intersecting subsets V^{AND} and V^{OR}. A non-negative (time) function T on E is feasible if it satisfies the following conditions:

$$T(v) \geq \max_{(u,v)\in E}\{0, T(u) + d(u,v)\}, \text{ for all } v \in V^{AND} \tag{1}$$

$$T(v) \geq \min_{(u,v)\in E}\{\infty, T(u) + d(u,v)\} \geq 0, \text{ for all } v \in V^{OR} \tag{2}$$

We stress that if there are no edges incoming to an OR node v, then $T(v) = \infty$. Notice that giving infinite values to all nodes, we arrive at a feasible solution. Let us define function $T^* = T^*(G)$ by defining its value at any node $v \in V$ as infimum of $T(v)$, over all feasible solutions T.

Claim 2.1 (i) *The solution T^* is feasible.*
(ii) *The solution T^* satisfies all conditions* (8) *and* (10) *as equations.*

The proof is omitted. The solution $T^*(G)$ will be referred as the *optimal solution*, and the conditions (8) and (10), w.r.t. it, as *equations*. The problem goal is, given graph G, to find the optimal solution. Let us remind first the algorithms for the pure AND and OR cases. The sense of S is a set of nodes with the T^{final} label fixed.

Algorithm AND

```
compute in-degree $indeg(v)$, for all $v \in V$
$T(v) \leftarrow 0$, $T^{final}(v) \leftarrow \infty$, for all $v \in V$
**while** (there is a node $u \in V$ with $indeg(u) = 0$)
    /* when the labels are fixed for all predecessors of $u$,
    the label of $u$ is well defined */
    $T^{final}(u) \leftarrow T(u)$
    **for each** edge $(u, v) \in E$
        $indeg(v) \leftarrow indeg(v) - 1$
        $T(v) \leftarrow \max\{T(v), T(u) + d(u,v)\}$
```

```
Algorithm OR
$T(v) \leftarrow \infty$, for all $v \in V$
$S \leftarrow \emptyset$
$T^{final}(s) \leftarrow T(s) \leftarrow 0$, for the source node $s$
while $(S \neq V)$
    set $u$ to be the node with the minimum value of $T$ in $V \setminus S$
    $T^{final}(u) \leftarrow T(u)$, $S \leftarrow S \cup \{u\}$
    for each edge $(u, v) \in E$
        $T(v) \leftarrow \min\{T(v), T(u) + d(u, v)\}$
```

The algorithm of [5, 4] is as follows:

```
Routine Scan(u)
for each edge $(u, v) \in E$
    $indeg(v) \leftarrow indeg(v) - 1$
    if $v \in V^{AND}$
        $T(v) \leftarrow \max\{T(v), T(u) + d(u, v)\}$
    else $T(v) \leftarrow \min\{T(v), T(u) + d(u, v)\}$
```

```
Algorithm AND/OR
/* Initialization */
compute in-degree $indeg(v)$, for all $v \in V$
$T(v) \leftarrow 0$, for all $v \in V^{AND}$
$T(v) \leftarrow \infty$, for all $v \in V^{OR}$
$T^{final}(v) \leftarrow \infty$, for all $v \in V$
$S \leftarrow \{\emptyset\}$

/* Main part */
$m \leftarrow 0$
while $(m \neq \infty)$

    /* Phase "AND" */
    while (there is a node $u \in V$ with $indeg(u) = 0$)
        $T^{final}(u) \leftarrow T(u)$, $S \leftarrow S \cup \{u\}$
        /* this is correct for both node types AND and OR */
        Scan(u)

    /* Phase "OR" */
        let $m$ be $\min_{v \in V^{OR} \cap (V \setminus S)} T(v)$, or infinity, if $V^{OR} \setminus S = \emptyset$
        if $m \neq \infty$
            choose $u : T(u) = m$
            $T^{final}(u) \leftarrow T(u)$, $S \leftarrow S \cup \{u\}$
            Scan(u)
```

/* Reminder: for all nodes remaining in $V \setminus S$, $T^{final} = \infty$ */

Remark: The name "AND" is given to the first phase of the iteration because of the method used.

Now, let us prove the correctness of Algorithm AND/OR.

Theorem 2.2 *For any graph without zero cycles, Algorithm AND/OR finds $T^{final} = T^*$. Its running time bound is the sum of running time bounds for Algorithm AND and Algorithm OR, for graphs of the same size.*

Proof: We prove, by induction on the moment of inserting a node u into S, that the value assigned then to $T^{final}(u)$ is equal to the optimal value $T^*(u)$ (note that neither $T^{final}(u)$, nor $T(u)$ change after this moment).

If a node u is inserted into S in a Phase "AND", then for all its predecessors holds $T = T^*$, by construction and by the induction hypothesis. Since the current value $T(u) = T^{final}(u)$ satisfies the appropriate condition (8) or (10) as an equation, and so does $T^*(u)$, they coincide, by Claim 2.1(ii), as required.

Let us assume, to the contrary, that there exist nodes assigned wrong values of T^{final} at a Phase "OR". Let v_0 be the first node, in the execution of the algorithm, assigned a wrong value, say, $m0$. The solution T^{final} found by the algorithm is feasible, by construction; hence the only case is $T^*(v_0) < T^{final}(v_0) = m0$. As a consequence, $T^*(v_0) \neq \infty$. Since v_0 is an *OR node*, there exists a node, v_1, such that $T^*(v_1) + d(v_1, v_0) = T^*(v_0)$. Recall that, by the algorithm and induction hypothesis,

$$m0 = \min_{(u,v)\in E,\ u\in S,\ v\in V^{OR}\cap(V\setminus S)} \{T^*(u) + d(u,v)\}\ .$$

Since $T^*(v_1) + d(v_1, v_0) < m0$, the edge (v_1, v_0) does not participate in computing the latter minimum as above; hence, $v_1 \in V \setminus S$. If v_1 is an *OR node*, we analyse it as above, arriving at a node v_2 with similar properties. Let v_1 be an *AND node*. Then, there must exist some its predecessor, v_2, in $V \setminus S$, by the finishing condition of the **while** loop at the previous Phase "AND". Since $T^*(v_2) + d(v_2, v_1) \leq T^*(v_1)$, holds $T^*(v_2) < m0$, similarly to v_0 and v_1.

Let us continue to build, as above, a sequence $v_0, v_1, v_2, \ldots$, with non-increasing $T^*(v_i)$. Since V is finite, we must arrive at a cycle. Evidently, T^* is a constant, at this cycle. Hence, all edges of this cycle have the weight zero, a contradiction to the assumption of Theorem.

Finally, let us assume, to the contrary, that there exist nodes with the wrong ∞ value, remaining in $V \setminus S$ when the algorithm finishes. An analysis similar to that made above for Phase "OR" brings us to a contradiction.

Initialization is done in linear time. The algorithm operations at its Phases "AND" are similar to those of Algorithm "AND", and can be given the same linear time bound. Its operations at Phases "OR" are similar to those of Algorithm "OR". It is easy to see that interleaving by Phases "AND" is so independent that cannot contradict using the data structures which fasten the execution of Algorithm "OR". Hence, the time bounds for Algorithm "OR" are appliable to the Phases "OR" together. This justifies the summary time bound. □

3 Algorithm for the General Case

Let us consider the general AND/OR scheduling problem, allowing zero cycles. As mentioned in [7, 2], its nature does not allow to extend S—the self-supporting set of nodes with optimal labels—node by node, as in Phases AND and OR of the previous algorithms. Let us consider V as divided into *levels*, where level x is the set of nodes with the optimal label T^* equal to x. It may be that some nodes in a level support each other and can be added to S only together. Notice that, in particular, there are more a-priory members of level 0: not only AND nodes without incoming edges, but also the nodes of any zero cycle consisting of OR nodes only. Indeed, nodes of such a "zero OR cycle" support each other, and hence they can be given label 0.

Let us present Algorithm AND/OR General, working for any precedence graph with non-negative edge weights. The purpose of its iteration is to extend S by the *entire next level of nodes*, lowest in $V \setminus S$. (Thus, the levels are added to S in the increasing order of their label.) At any iteration, we use auxiliary node labels. Their final values isolate the nodes of the next level as the nodes with the *minimal* auxiliary label; moreover, this minimal auxilary label coincides with their optimal label. Let $G(W)$, $W \subseteq V$, denote the induced subgraph on the node subset W. Let $G^{OR,zero}$ denote the graph on V^{OR}, with the zero edges from E, between its nodes. The algorithm description is followed by an explanation of its intuition.

```
**Routine Scan(u)**
**for each** edge $(u, v) \in E$
    $indeg(v) \leftarrow indeg(v) - 1$
    **if** $v \in V^{AND}$
        $T(v) \leftarrow \max\{T(v), T(u) + d(u, v)\}$
    **else** $T(v) \leftarrow \min\{T(v), T(u) + d(u, v)\}$
```

```
**Algorithm AND/OR General**
**/* Initialization */**
compute in-degree $indeg(v)$, for all $v \in V$
$T(v) \leftarrow 0$, for all $v \in V^{AND}$
$T(v) \leftarrow \infty$, for all $v \in V^{OR}$
$T^{final}(v) \leftarrow null$, for all $v \in V$
$S \leftarrow \{\emptyset\}$
```

```
/* Zero OR Cycles */
find the strongly connected components of $G^{OR,zero}$
for each node $u$ of any non-singleton strongly connected component found
    /* any node of such a component lies on a zero OR cycle */
    $T^{final}(u) \leftarrow T(u) \leftarrow 0$, $S \leftarrow S \cup \{u\}$
    Scan(u)

while ($S \neq V$)
    /* Iteration */
    $auxT(v) \leftarrow T(v)$, for all nodes $v \in V \setminus S$
    $auxindeg(v) \leftarrow indeg(v)$, for all nodes $v \in V \setminus S$
    for each edge $(u,v)$ with $d(u,v) > 0$ in $G(V \setminus S)$
        if $v \in V^{AND}$
1           $auxT(v) \leftarrow \infty$
2       else $auxindeg(v) \leftarrow auxindeg(v) - 1$

    /* we maintain list $Q$ of non-processed nodes in $V \setminus S$,
    with $auxT$ non-increasing */
    while ($Q$ is non-empty)
        pop the first node $u$ from $Q$
        if $u$ is not an OR node with $auxindeg(u) > 0$
            $m \leftarrow auxT(u)$
            for each edge $(u,v)$ with $d(u,v) = 0$

                if $v \in V^{AND}$
3                   $auxT(v) = \max\{auxT(v), auxT(u)\}$
                    /* if $auxT(v)$ increases, $v$ is put at the head of $Q$ */
                else /* $v \in V^{OR}$ */
                    $auxindeg(v) \leftarrow auxindeg(v) - 1$
                    if $auxindeg(v) = 0$
4                       $auxT(v) = \min\{auxT(v), auxT(u)\}$
                        /* if $auxT(v)$ decreases, $v$ is put at the head of $Q$ */

    for each node $u$ with $auxT(u) = m$
        $T^{final}(u) \leftarrow auxT(u)$, $S \leftarrow S \cup \{u\}$
        Scan(u)
```

Let us explain an intuition behind Algorithm AND/OR General. Let us recall that the current label $T(v)$, at the beginning of any iteration, equals to:

$$\max_{u \in S, (u,v) \in E} \{0, T^*(u) + d(u,v)\}, \text{ for each } v \in V^{AND} ,$$

$$\min_{u \in S, (u,v) \in E} (\infty, T^*(u) + d(u,v), \text{ for each } v \in V^{OR} .$$

Therefore, for any OR node v, $T(v)$ is an upper bound for $T^*(v)$, and for any AND node, $T(v)$ is a lower bound for $T^*(v)$. Moreover, these bounds express the entire influence of nodes in S on the optimal label of v.

Let t_{min} be the minimal optimal label in $V \setminus S$, at some iteration. Let L_{min} denote the lowest level in $V \setminus S$, i.e., the set of nodes with the optimal label t_{min}. It is sufficient to show that each iteration finds $m = t_{min}$ and extends S by L_{min}. At any iteration, we process the nodes in a non-increasing order of their auxiliary labels $auxT$, level by level, w.r.t. $auxT$. Our aim is to show that (i) any non-last level is provably disjoint from L_{min}, and (ii) that all the nodes in the last level can be given the label t_{min}, which suffices.[1]

Let us analyze L_{min}. For any OR node v, let us call "defining" any edge that brings the minimum to $T^*(v)$, in equation (10).

Observation 3.1 (i) *For any AND node in L_{min}, no edge from $(V \setminus S) \setminus L_{min}$ enters it, while any edge from L_{min} entering it is a zero edge, unless $t_{min} = \infty$;*

(ii) *For any OR node in L_{min}, any defining edge entering it comes from S or from L_{min}; in the latter case, it is a zero edge, unless $t_{min} = \infty$.*

Based on this, we show that operations 1–4, crucial in the algorithm, do not contradict to our aim.

1) No non-zero edge can enter an AND node in L_{min}, unless $t_{min} = \infty$. So, we may label any such node by ∞. (*See line 1 in the algorithm.*)

2. No non-zero edge can be defining, for an OR node in L_{min}, unless $t_{min} = \infty$. So, we may delete it. (*See line 2 in the algorithm.*)

3. If an AND node v is a successor of a node u, it cannot be in L_{min}, unless u is in L_{min}. So, we may relabel v by the auxiliary label of u, if the current auxiliary label of v is lesser. (*See line 3 in the algorithm.*)

4. For any OR node in L_{min}, if the currently considered incoming edge is not the last one, in this iteration, then we may be sure that edges considered later will bring a not worse value. So, we may delete this edge. (*See line 4 in the algorithm.*)

Let V_m be the set of nodes with the minimal auxiliary label m. As a consequence from the above items, $(V \setminus S) \setminus V_m$ is disjoint from L_{min}. By construction, (i) no edge entering an AND node in V_m goes from $(V \setminus S) \setminus V_m$, and (ii) for any OR node $v \in V_m$, either there is at least one edge entering it from V_m, or $T(v) = m$. Hence, we may give the final label m to the nodes in V_m, being sure of feasibility. Therefore, $m \geq t_{min}$.

[1] This is similar to what sculptures say: "It is very simple to sculpture: you need only to cut off the extra pieces of marble."

On the other hand, if we trace the iteration algorithm, keeping in mind the nodes in L_{min} labeled t_{min}, we see that there cannot be any cause to give to them a strictly greater auxiliary label. Hence, $m = t_{min}$ and $V_m = L_{min}$.

4 Correctness Proof

The algorithm finishes when all the nodes are given the final label. The algorithm is finite since each iteration inserts at least one new node into S.

It is well known that in any non-singleton strongly connected graph, each node lies on a cycle. So, assigning the zero final label at Zero OR Cycles phase is correct. Clearly, after this phase, there is no zero OR cycle in $G(V \setminus S)$.

Correctness of the entire algorithm is implied by the following statements.

Lemma 4.1 *If in a solution, for every node v belonging to a zero OR cycle $T(v) = 0$, and*

$$T(v) \geq \max_{(u,v)\in E}\{0, T(u) + d(u,v)\}, \text{ for all } v \in V^{AND}, \tag{3}$$

and

$$T(v) \geq \min_{(u,v)\in E}\{\infty, T(u) + d(u,v)\} \geq 0, \text{ for all } v \in V^{OR}, \tag{4}$$

Then the solution is optimal.

Proof: Let us assume that for some node v, $T(v)$ is higher than its optimal value. this means that one of v's predecessors v' must also have $T(v')$ higher than its optimal value with $T(v') \leq T(v)$. Since the graph is finite, there must be a cycle of nodes that can be improved with $T(v) \leq T(v') \leq ... \leq T(v)$. This implies that all of these nodes are part of an independent zero cycle. But we assumed that this kind of nodes have zero labels, so they cannot be improved. $\square$

Proposition 4.2 *Each iteration arrives at $m = t_{min}$, while the nodes with auxT equal to m form L_{min}.*

Proof: Let us consider any iteration. It is discussed as processing $G(V \setminus S)$.

Lemma 4.3 *All nodes are processed. Hence, all edges are processed.*

Proof: Clearly, all AND nodes are processed. An OR node v_0 could be not processed, if there remains a non-processed zero edge, (v_1, v_0), entering it, when the iteration finishes. Then, also the OR node v_1, and some zero edge, (v_2, v_1), are not processed. Continuing in this way, we arrive at a zero OR cycle, — a contradiction to inserting all nodes of such cycles into S at the phase Zero OR Cycles. $\square$

Lemma 4.4 *At the beginning of every iteration we have:*

$$T(v) = \max_{(u,v)\in E, u\in S}\{0, T(u) + d(u,v)\}, \ \textit{for all } v \in (V \setminus S) \cap V^{AND}. \tag{5}$$

$$T(v) = \min_{(u,v)\in E, u\in S}\{\infty, T(u) + d(u,v)\} \geq 0, \ \textit{for all } v \in (V \setminus S) \cap V^{OR}. \tag{6}$$

Proof: The value of $T(v)$ for nodes that are not in S is defined only by the call to scan. The scan is performed for every node in S. The scan procedure maintains the desired value of $T(v)$ for every node v that is not in S. □

Lemma 4.5 *During a single iteration, if u is the currently processed node, then during the rest of the iteration:*

$$auxT(u) \geq \max\{auxT(v)\}, \ \textit{for all } v \in Q. \tag{7}$$

Proof: At the beginning of the processing of u this is trivially correct. During the iteration, the only way an $auxT$ label is changed is by steps 3 and 4. In both these steps $auxT(v) = auxT(u)$, so at the end of the processing of u the lemma is true. Since the next node that will be processed, u', is from Q, we get $auxT(u) \geq auxT(u')$. We can use the same reasoning again and see that $auxT(v)$ will not rise over $auxT(u)$ for any v in Q. □

Lemma 4.6 *During a single iteration, after a node is processed, its $auxT$ label is not increased.*

Proof: For every OR node v, since $auxT(v)$ is not increased in any place in the iteration, this lemma is trivially true. Let us assume that after an AND node, v, is processed its label is increased. This can happen only in step 3 of the algorithm. This means that the currently processed node u has a higher label than v, which was already processed. This contradicts lemma 4.5 □

Corollary 4.7 *After a node u is processed $auxT(u)$ stays unchanged until the end of the iteration.*

Proof: For an AND node u, $auxT(u)$ is not decreased in the iteration. The last Lemma states that it is not increased.

For an OR node u, u is processed only after all of its predecessors have been processed and $auxindeg(u) = 0$. since the only way to update a node is by its predecessor, $auxT(u)$ will not be updated until the end of the iteration. □

Lemma 4.8 *At the beginning of every iteration we have:*
For every AND node, $v \in S$, all predecessors of v are in S, and:

$$T(v) \geq \max_{(u,v)\in E}\{0, T(u) + d(u,v)\}. \tag{8}$$

For every OR node, $v \in S$:

$$T(v) \geq \min_{(u,v)\in E, u\in S}\{\infty, T(u) + d(u,v)\}. \tag{9}$$

Proof: We will prove the lemma by induction on the number of iterations. At the beginning of the algorithm S is empty so the claim is trivially true. after "Zero OR Cycles" there are only OR nodes in S and every node $v \in S$ has $T(v) = 0$. Every node $v \in S$ has a predecessor $u \in S$ such that $T(u) = 0$ and $d(u,v) = 0$, so the lemma is true.
Let us assume that at the beginning of some iteration the lemma is true. The labels of the nodes in S do not change throughout the iteration, so the only way the lemma can be contradicted is by a node v that has been added in the iteration. There are four cases that might contradict the lemma:

Case 1 : *v is an AND node and one of its predecessors u is not in S.* Since all nodes with the minimal final $auxT$ are added to S, this means that $final\ AuxT(u) > final\ AuxT(v)$. From Lemma4.5 we get that we scanned u before we scanned v. When we scan u we should have updated $auxT(v)$ to be at least $auxT(u)$ and this leads us to a contradiction.

Case 2 : *v is an AND node and $T(v) < T(u) + d(u,v)$ for some predecessor, u, of v.* If u was in S at the beginning of the iteration, we get from Lemma 4.4 that $T(v) = final\ auxT(v) \geq auxT(v) \geq T(u) + d(u,v)$. If u was not in S we get from case 1 that it is entered to S with v. This means that $T(u) = final\ auxT(u) = final\ auxT(v) = T(v)$ which is a contradiction.

Case 3 : *v is an OR node with $T(v) < \infty$, and none of its predecessors is in S.* At the beginning of the iteration the value of $auxT(v)$ is equal to $T(v)$. from Lemma 4.4 we get that at the beginning of the iteration $auxT(v) = T(v) = \infty$. Since at the end of the iteration $final\ auxT(v) = T(v) < \infty$, it must be that $auxT(v)$ have been updated by step 4 of the algorithm. This means that during the processing of some predecessor u of v with $d(u,v) = 0$ we executed $final\ auxT(v) = auxT(u)$. from Lemma 4.7 we get that $final\ auxT(u) = final\ auxT(v)$. Therefore u will be also added to S, contradiction.

Case 4 : *v is an OR node and $T(v) < T(u) + d(u,v)$ for all predecessor of v in S.* By Lemma 4.4, at the beginning of the iteration

$$T(v) = \min_{\infty,(u,v)\in E, u\in S}\{\infty, T(u) + d(u,v)\}. \tag{10}$$

This means that during the iteration $auxT(v)$ decreased. similarly to the last case this means that there is a node u with $d(u,v) = 0$ which was added to S with v. $\Box$

Lemma 4.9 *The $auxT$ labels produced at the iteration, together with the T labels on S, form a feasible solution.*

Proof:

At any node labeled infinity, the problem conditions are trivially satisfied. So, we may concentrate on the AND nodes without positive entering edges, and on the OR nodes with at least one zero edge or edge from S entering it. That is, we need to check the conditions over edges from S and zero edges from $V \setminus S$ only.

Sketch: Notice that we scan zero edges in the non-increasing order of the $auxT$ label of their initial nodes. For an AND node v, any finite label is given by the first scanned edge entering it. After that, all edges entering v bring equal or lesser values. Hence, the maximum condition (8) at v is satisfied. The label for an OR node v is defined by the last considered entering edge, so all other edges bring to it greater or equal values. Hence, the minimum condition (10) at v is satisfied. $\Box$

By Lemma 4.9, the $auxT$ labels given by the algorithm are greater or equal than the optimal labels. As a consequence, for any node v in $(V \setminus S) \setminus V_m$ holds $auxT(v) \geq T^*(v) > t_{min}$. Now, it would be sufficient to prove that each node $v \in L_{min}$ is labeled by $auxT(v) = t_{min}$. Assume, to the contrary, that this is not so. Then, t_{min} is finite. Choose v to be the first node in L_{min}, in the chronological order, given the final $auxT$ value greater than t_{min}.

Case 1 : *v is an AND node.* By Observation 3.1(i), all edges entering v are from S or from L_{min}. Let (u,v) be the edge bringing the wrong value to v, while scanning u. By the induction hypothesis, nothing wrong can come from S, so u belongs to L_{min}. By Observation 3.1(i), (u,v) is a zero edge. Hence, by our assumption, the label given (previously) to u is greater than t_{min}. A contradiction to the choice of v.

Case 2 : *v is an OR node.* Let (u,v) be the edge bringing the wrong value to v. By the induction hypothesis, the influence of S is correct. Hence, u belongs to $V \setminus S$, and it has been given a label greater than t_{min}.

Notice that the edge bringing the optimal value to v, in the optimal solution, cannot be from S; indeed, otherwise, the same edge would bring the record value also in the solution built by the algorithm. Hence, by Observation 3.1(ii), there is at least one zero edge, say (w,v), coming to v from L_{min}. By our assumption on (u,v) being the last edge entering v considered at the iteration, all edges coming to v from L_{min} were scanned previously, in particular, the edge (w,v). Thus, w was been given its label earlier than u. Hence, the label given to w is at least that given to u, i.e., greater than t_{min}. A contradiction to the choice of v. $\Box$

5 Data Structure and Time Complexity

Initialization: Finding the strongly connected components can be done in a linear time, $O(|E|)$ (see, for example, [3]). The other operations are also linear.

The number of iterations: At each iteration, at least one node enters S. Hence, there are at most $|V|$ iterations.

Data Structure: We maintain a priority queue of the nodes in $V \setminus S$ w.r.t. T, during the entire execution. Let us choose a data structure with the constant time pop query and the logarithmic update query. The queue initialization, at the beginning of the algorithm costs $O(|V|)$, since any OR node has label ∞ and any AND node has label 0. The T values change only in the routine Scan, at most once for each edge, i.e., $O(|E|)$ times totally. Hence, the priority queue maintenance costs $O(|E| \log |V|)$.

Iteration: The $auxT$ labels and the sorted list Q are initialized in time $O(|V|)$, by copying them from the above priority queue. These labels change while processing edges, at most once for each edge. When processing a non-zero edge, (u, v), v is put at the head of Q, in $O(1)$ time. During the **while** loop, when edge (u, v) is processed, if a new value is given to $auxT(v)$, it is always equal to the current value of m. Then, v is put at the head of Q, in $O(1)$ time. Therefore, the total time of updates is $O(|E|)$. Clearly, the other operations are done in linear time, i.e., each iteration costs $O(|E|)$.

Summarizing, the total time of the algorithm is $O(|V||E|)$.

6 Related Work and Discussion

1. In the paper of Knuth [6], the graphs considered are not general. The graph is bipartite, where the parts are the sets of AND and OR nodes, and from each AND node, there is a single out-coming edge. A transformation of a non-bipartite graph to an equivalent bipartite one without expanding the problem is suggested in [1]. Taking care of AND nodes with multiple out-coming edges is more heavy. Any reduction of the general case to the case considered, known to us, leads to a substantial increase of the graph size. One of the ways is as follows. Let v be an AND node with out-coming edges $(v, w_1), \ldots, (v, w_k)$. We add new nodes w, $v_1, \ldots v_k$, add the edge (v, w), and replace each edge (v, w_i) by the sequence of two edges (w, v_i) and (v_i, w_i). As a result, the number of AND nodes of the new graph increases by the total number of *edges* out-coming from AND nodes, in the original graph.

Another difference, from our setting, is related to the essence of the application: The events, in a solution, are essentially ordered by the causal relation. This property is used in proofs, in [6], which prevents extending these proofs to our setting.

The setting of Knuth is more general than our one, in the definition of the precedence restriction for an AND node. Instead of the maximum function only, a more general class of so called “superior” functions is considered.

2. In the paper of Moehring et al. [7], the class of graphs considered is analogous to the class considered in [6], but is symmetric to it: from each OR node out-comes a single edge. The similar expansion, as above, happens when reducing a general graph to a graph of this kind. As a result, the number of OR nodes increases by the number of *edges* out-coming from OR nodes in the original graph. Since the number of OR nodes is a factor in the time bound, the bound becomes $O(|E|^2)$, where E is the set of edges of the original graph.

3. In the paper of Adelson-Velsky et al. [1], an algorithm for the general case is suggested. It is more complicated than our algorithm, but has the same time bound as that achieved in this paper. We have found a few substantial gaps in the proof of the algorithm correctness, given in [1]. The proof is based on Lemma 1 and Lemma 2. The crucial one is Lemma 2. We found out that neither its statement nor its proof is correct.

Remark: In January 2002, our algorithm was presented at the Seminar of the CS Dept. of HAIT, Holon, Israel; the talk was hosted by E. Levner. We would like to mention that the version of [1] that we cite here is the latest one before this presentation.

References

[1] G. M. Adelson-Velsky, A. Gelbukh, and E. Levner. A fast scheduling algorithm in AND-OR graphs. Manuscript, submitted to *Mathematical Problems in Engineering* in autumn 2001.

[2] G. M. Adelson-Velsky, and E. Levner. Project scheduling in AND/OR graphs: A generalization of Dijkstra’s algorithm. Technical Report, Dept. of Computer Science, Holon Academic Institute of Technology, Holon, Israel, November 1999.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, McGraw-Hill, 1990.

[4] E. A. Dinic. The fastest algorithm for the PERT problem with AND- and OR-nodes (the new-product-new technology problem). In R. Kannan and W. R. Pulleyblank (Eds.), *Integer Programming and Combinatorial Optimization, Proceedings of a conference held at the University of Waterloo, May 28-30, 1990*, Mathematical Programming Society, pp. 185-187.

[5] E .A. Dinic, A. B. Merkov, and I. A. Tejman. Coordination analysis and computing of early periods of launching for a set of new technologies, in: *Models and Methods for Forecast of the Science-Technology Progress*, V. V. Tokarev ed., Moscow, 1984, 125–131 (in Russian).

[6] D. E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters* **6**, no.1, pp.1-5.

[7] R. H. Moehring, M. Skutella, and F. Stork. Scheduling with AND/OR precedence constraints. Technical Report No. 689/2000, Technische Universitat Berlin, August 2000, 26 p. (Accepted to *SIAM J.of Computing* in 2002.)