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

Paz Carmi* Yefim Dinitz† Shahar Golan† Guy Rozenwald‡

March 11, 2008

Abstract

The paper studies scheduling with AND/OR precedence constraints. The events are vertices of a non-negatively weighted graph $G = (V, E, d)$; the precedence relation is defined by its edges, whose weights represent delays. An event of type AND should be scheduled after *all* its preceding events, with the corresponding delays after them. An event of type OR should be scheduled after *at least one of* its preceding events, with the corresponding delay after it. The early schedule is in question. A few algorithms solving various restricted cases of this problem are known. We present an algorithm for the general case, where the precedence graph may have an arbitrary structure and zero weight cycles are allowed. Its running time is $O(|V||E|)$.

*????

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

‡E-mail: guyrozenwald@hotmail.com.

1 Introduction

The classic PERT scheduling problem (see e.g. [**?**]) is defined by a directed graph $G = (V, E, d)$, where d is an edge weight function. Its vertices represent events, and its edges represent the precedence relation between them: if there is an edge from vertex u to vertex v (we say: u *precedes* v), 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 time-schedule, which obeys the precedence relation (the *early times* for events). It is accepted that if for some event no feasible time exists, then the time *infinity* is assigned to it. The motivation for PERT is finding the early time-schedule for a large project, where vertices are intermediate events (e.g., the starting or finishing time of a project component).

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

The AND/OR setting of PERT with non-negative delays is defined in [5, 4] as follows. The given digraph may be arbitrary, with two vertex types. Any vertex with precedence conditions as above is considered as an *AND vertex*. For an *OR vertex* v, its time should be not earlier than by $d(u, v)$ after *at least one* vertex u preceding it.

A motivation in [5, 4] is the following New-Product-New-Technology problem. There are new technologies, which supply some new products to the market and need certain such products. An AND vertex represents the event of launching of a new technology; at this time, each required new raw product should have being present at the market for a certain time period. An OR vertex represents the first appearance of a new product at the market, which happens after a certain delay from launching an arbitrary new technology producing it. Early schedule of this technology-product system is in question. (A relaxing assumption is that the amount of any new product at the market is not taken into account.) A-priory lower time bounds for new technologies are modeled by introducing an auxiliary AND vertex TIME-0,

with no predecessors, and the properly weighted edges from it. If some new products can be obtained without a help of any new technology (e.g., by exporting it), at certain time moments, this is modeled by using a similar auxiliary AND vertex EXPORT and edges from it. In this application, the graph is almost bipartite, where vertex TIME-0 is an exception.

The common sense of a zero delay precedence is "not earlier than". Specifics of such precedence is that some group of events with mutual zero delay precedences between them may be scheduled simultaneously, supporting each other. (Note that in future, quantum computation may become an additional source of precedence relations with zero delay, since for quantum objects it is natural to appear in groups.)

More motivation and applications can be found in [6, 2, 7]. The latter paper presents an extensive study of (a restricted case of) the non-negative weight AND/OR PERT problem.

By definition, the pure case of AND vertices only is the classic PERT. It is interesting that the case of OR vertices only, except for the AND vertex "source" without incoming edges, is essentially equivalent to another cornerstone optimization problem: finding the shortest path lengths from "source" to the other vertices. The basic variant of Dijkstra algorithm [**?**] solves it in time quadratic in $|V|$, while using appropriate data structures provides time bounds almost linear in the size of G (see e.g., [3]).

The two above algorithms, studied in all basic courses in algorithms, are of somewhat similar and of somewhat different nature. Surprisingly, for the hybrid AND/OR problem as above with no zero cycle in the precedence graph, there exists an algorithm, which is combined from these two classic algorithms and runs in the summary time. Such a hybrid algorithm was suggested independently in [6] and in [5, 4]. The problem considered by Knuth in [6] is presented in another language; it is similar but not equilvalent to the AND/OR scheduling problem. Mutual reductions are not always possible, and if possible, do not preserve the correctness proofs and the time bounds (see a discussion in Appendix). Presentation in [5, 4] lacks the correctness proof. Such a proof is provided in this paper.

The question of existence of a polynomial time algorithm for the general non-negative weight AND/OR problem, when zero cycles are allowed, was open for about a decade. Adelson-Velsky and Levner suggested an algorithm solving it in time $O(|E|^2) = O(|V|^4)$ [2]. Our algorithm solves it in time $O(|V||E|) = O(|V|^3)$. It was formulated and proven at the course Advanced

Algorithms, given by the second author in the spring of 2001, when the rest of the authors were its students; it was first presented in [**?**].

The related work is as follows. Moehring et al. suggested in 2000 an algorithm solving a special case of the problem in time $O(|V||E|)$ [7]. As observed by E. Levner, there is a reduction from the general case to their case, so that the running time increases to $O(|E|^2) = O(|V|^4)$ (see a discussion in Appendix). An algorithm for the general case of the problem, with the time bound equal to ours, was suggested by Adelson-Velsky et al. in [1]; it is more complicated than our algorithm, and its correctness is not proved.

2 Solution to the Zero-Cycle-Free Case

The formal problem setting includes a directed weighted graph $G = (V, E, d)$, where d is a non-negative function on the edge set E and the vertex set V is divided into two non-intersecting subsets V^{AND} and V^{OR}. A non-negative (time) function T on E is feasible if it satisfies the following constraints:

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

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

Let us define function T^* by assigning to each vertex $v \in V$ the infimum of $T(v)$ over the set of all feasible solutions T (since $T \equiv \infty$ is feasible, this set is non-empty).

Claim 2.1 (i) *The solution T^* is feasible.*
(ii) *The solution T^* satisfies all conditions* (1) *and* (2) *as equations.*

Proof sketch: Item (i) is valid since the minimum of two feasible solutions is feasible. If item (ii) is wrong at a vertex v, then we can decrease $T^*(v)$ by a small amount without violating any constraint—a contradiction to T^* being the infimum.

The solution T^* will be referred to as the *optimal solution*, and the constraints (1) and (2), w.r.t. it, as *equations*. The problem goal is, given a graph G, to find the optimal solution.

In this section we consider the case when G has no zero cycle. Let us first remind the algorithms for the pure AND or OR cases, mentioned in Introduction. In this paper, all algorithms output T^{final}.

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 vertex $u \in V$ with $indeg(u) = 0$)
    $T^{final}(u) \leftarrow T(u)$
    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 vertex s
while $(S \neq V)$
    set u be the vertex 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], for the general no-zero-cycle case, 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
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 $v \in V$
$S \leftarrow \{\emptyset\}$; $m \leftarrow 0$

while $(m \neq \infty)$
    /* Phase AND */
    while (there is a vertex $u \in V$ with $indeg(u) = 0$)
        $T^{final}(u) \leftarrow T(u)$, $S \leftarrow S \cup \{u\}$
        Scan(u)

/* comment: works for both vertex types AND and OR */

/* Phase OR */

set m to $\min_{v \in V^{OR} \cap (V \setminus S)} T(v)$, or to ∞, if $V^{OR} \subseteq S$

if $m \neq \infty$

choose $u : T(u) = m$

$T^{final}(u) \leftarrow T(u)$, $S \leftarrow S \cup \{u\}$

Scan(u)

Theorem 2.2 *For any graph without zero cycles, Algorithm AND/OR finds the optimal solution. Its running time bound is the sum of those for Algorithm AND and Algorithm OR for graphs of a similar size.*

Proof: We prove first that the value of m does not decrease. Let m be assigned value m_1 at some Phase OR, at a moment t. By the finishing condition of the while loop at the previous Phase AND, no AND vertex will be added to S without scanning previously an edge in-coming it from a vertex u currently in $V \setminus S$. So, tracing back any chain of updates of T after t arrives inevitably at the T label of an OR vertex non-scanned at time t, which is least m_1. Hence, any new value of T will be at least m_1, which suffices.

We conclude that vertices are added to S and scanned with non-decreasing labels $T^{final} = T$ (except maybe those at the first phase AND). As a consequence, $T(u)$ does not change after adding u to S, for any $u \in V$. Observe, in addition, that incremental updating at Scan maintains the properties:

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

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

At the end of the algorithm execution, all vertices not in S have $T^{final} = \infty$. Therefore, by (3) and (4), the output solution is feasible.

We now prove, by induction on the moments of adding vertices v to S, that $T^{final}(v)$ equals $T^*(v)$. If v is added to S during some Phase AND, then all its predecessors are in S, and by the induction hypothesis, for all of them holds $T = T^{final} = T^*$. Since the current value of $T(v) = T^{final}(v)$ satisfies the appropriate condition (3) or (4), and by Claim 2.1(ii), also $T^*(v)$ satisfies the similar one, $T^{final}(v)$ and $T^*(v)$ coincide.

Let us now assume, to the contrary, that there exist vertices assigned wrong values of T^{final} at a Phase OR. Let $v_0 \in V \setminus S$ be assigned *first* a wrong value, $T^{final}(v_0) = m_0$. By (4) and the induction hypothesis,

$$m_0 = \min_{(u,v)\in E,\ u\in S,\ v\in V^{OR}\cap(V\setminus S)} \{T^*(u) + d(u,v)\} \ .$$

Since T^{final} is feasible, $T^*(v_0) < T^{final}(v_0) = m_0$. Recall that $T^*(v_0) < \infty$ satisfies (2) as an equation, i.e., there exists a vertex v_1, such that $T^*(v_1) + d(v_1, v_0) = T^*(v_0)$. Since $T^*(v_0) < m_0$, edge (v_1, v_0) does not participate in computing m_0 as above; hence, $v_1 \in V \setminus S$. If v_1 is an *OR vertex*, we analyse it as above, arriving at a vertex v_2 with similar properties. Let v_1 be an *AND vertex*. By the finishing condition of the while loop at the previous Phase AND, there exists its predecessor v_2 in $V \setminus S$. Since $T^*(v_2) + d(v_2, v_1) \leq T^*(v_1)$, we have $T^*(v_2) < m_0$, similarly to v_0 and v_1.

Let us continue to build in $V \setminus S$, 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 on this cycle. Hence, all its edges have weight zero—a contradiction to the assumption of the theorem.

A similar analysis brings to a contradiction also the assumption that there exist vertices with the wrong ∞ value, remaining in $V \setminus S$ when the algorithm finishes.

Initialization is done in a linear time. The algorithm operations at its Phases AND require a linear time similarly to those of Algorithm AND. The operations at Phases OR are similar to those of Algorithm OR. It is easy to see that the specifications for a data structure coincide with those for algorithm Dijkstra: to answer ExtractMin query $|V|$ times, while updating by DecreaseKey query $|E|$ times. Hence, the same time bounds are appliable. □

3 Algorithm for the General Case

Let us consider the general AND/OR scheduling problem, permitting zero cycles. Its nature does not allow to extend S—the self-supporting set of vertices with optimal labels—vertex by vertex, as in Algorithm AND/OR. It may happen that some vertices with the same T^* label support each other, and hence may be added to S only together. In particular, the vertices lying on any zero cycle consisting of OR vertices only ("zero OR cycle") support

each other, and thus have T^* equal to 0. Let us denote the set of such vertices by V^{OR-ZC}, and the graph on the vertex set V^{OR} with the zero edges from E between its vertices by G^{OR-Z}.

Lemma 3.1 *The set V^{OR-ZC} consists of all the vertices of the non-singleton strongly connected components of G^{OR-Z}.*

Proof: On one hand, any zero OR cycle is contained in G^{OR-Z}, and should belong entirely to the same strongly connected component. On the other hand, it is known that any two vertices of the same strongly connected component lie together on some cycle. □

Let us consider $V \setminus V^{OR-ZC}$ as divided into *levels*, where level L_x is the set of vertices with the optimal label T^* equal to x. The following Algorithm AND/OR General works for any precedence graph with non-negative edge weights. It initializes the set S of vertices with the final label by V^{OR-ZC}. Each its iteration extends S by the *entire next level of vertices* (i.e., that lowest in $V \setminus S$). Each iteration uses auxiliary labels $auxT$ at vertices of $V \setminus S$. At its beginning, $auxT$ are initialized by T. At its end, the vertices of the next level are distinguished as those with the minimal value of $auxT$; moreover, that minimal value coincides with their optimal label T^*.

We denote the subgraph of G induced by a vertex set $W \subseteq V$ by $G(W)$. Some steps in the algorithm description are given numbers, for their referencing in the correctness proof.

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
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 $v \in V^{OR}$
$S \leftarrow \emptyset$

/* Zero OR Cycles */

find the strongly connected components of $G^{OR,zero}$
for all vertices u of non-singleton strongly connected components
    $T^{final}(u) \leftarrow T(u) \leftarrow 0$
    $S \leftarrow S \cup \{u\}$
    Scan(u)

while $(S \neq V)$
    /* Iteration */
    $auxT(v) \leftarrow T(v)$, for all vertices $v \in V \setminus S$
    $auxindeg(v) \leftarrow indeg(v)$, for all vertices $v \in V \setminus S$
    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** $auxindeg(v) \leftarrow auxindeg(v) - 1$

    /* the non-processed vertices in $V \setminus S$ are maintained
    as the priority queue Q by non-increasing $auxT$ */
    $Q \leftarrow$ all vertices in $V \setminus S$, sorted by non-increasing $auxT$
    while (Q is non-empty)
        pop the first vertex u from Q
        if u is not an OR vertex with $auxindeg(u) > 0$
            $m \leftarrow auxT(u)$
            for each edge (u, v) with $d(u, v) = 0$

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

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

4 Correctness Proof

This section proves the correctness of **Algorithm AND/OR General**. By Lemma 3.1, assigning the zero final label at **Zero OR Cycles** phase is correct. Clearly, after this phase, there is no zero OR cycle in $G(V \setminus S)$.

Let us analyze any iteration of the outer **while** loop. We denote by t_{min} the minimum value of T^* in $V \setminus S$. The rest of the proof shows that at the iteration end, m is given the value t_{min}, and exactly $L_{t_{min}}$ is added to S, which will suffice for the algorithm correctness.

Observation 4.1 *At the beginning of any iteration we have:*

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

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

Indeed, the value of $T(v)$ for vertices that are not in S is changed only by the calls to **Scan**. Procedure **Scan** is performed for every vertex in S. So, the stated values of T are mainained for every vertex in $V \setminus S$.

The iteration is analyzed as processing $auxT$ labels at $G(V \setminus S)$. At the beginning, $auxT$ is initialized by the values of T as in equations (5) and (6). Note that by the algorithm description, during the iteration $auxT$ label of any AND vertex can only increase, and that of any OR vertex can only decrease. We denote the value of $auxT(v)$ at the iteration end by $final\_auxT(v)$.

Lemma 4.2 *All vertices in $V \setminus S$ are processed.*

Proof: By the algorithm, all AND vertices are processed. An OR vertex v_0 could be not processed, if there remains a non-processed zero edge entering it, (v_1, v_0). Then, also the OR vertex 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 vertices of such cycles into S at phase **Zero OR Cycles**. □

As a consequence of the lemma, all edges in $G(V \setminus S)$ are processed.

Lemma 4.3 *After we process a vertex u with $auxT(u) = l$, every vertex v in Q will have $auxT(v) \leq l$ for the rest of the iteration.*

Proof: At the beginning of the processing of u the lemma is correct, since u is at the head of Q. During processing u, the only way to change $auxT$ is at steps 3 and 4. In both these steps $auxT(v) \leftarrow auxT(u)$, so at the end of processing u the lemma is true. Since the next vertex u' to be processed is taken from Q, its $auxT$ label is at most l. Using the same reasoning repeatedly, we see that $auxT$ will never rise over l for vertices in Q. □

By the lemma, the values $auxT(u)$ in the while loop of the iteration are non-increasing.

Lemma 4.4 *After a vertex is processed, its auxT label is not changed.*

Proof: Any OR vertex u is processed only after all of its predecessors have been processed. Since the only way to update a vertex is by processing its predecessors, $auxT(u)$ will not be updated since then.

For any AND vertex u, the value of $auxT(u)$ is changed in the inner **while** loop only at step 3. Lemma 4.3 implies that for any vertex v processed after vertex u, holds $auxT(v) \leq auxT(u)$. Hence, $auxT(u)$ will not increase after u is processed. Hence, it will not change, since u is an AND vertex. □

Summarizing, the following statement holds:

Proposition 4.5

1. *All vertices in $V \setminus S$ are processed, with non-increasing values of $auxT$.*
2. *For any v in $V \setminus S$, $final\_auxT(v)$ is equal to the value of $auxT(v)$ at the time of processing v.*

Corollary 4.6 *For any vertex $v \in V \setminus S$, the label $auxT(v)$ is updated at most once.*

Proof: The label $auxT(v)$ is updated only by steps $1, 3, 4$ in the iteration. If the label of v is changed to ∞ by step 1, then v is an AND vertex, and thus $auxT(v)$ cannot decrease. Assume now that the label of v is changed during processing u at step 3 or 4. This means that $auxT(v)$ is set to $auxT(u)$, which is maximal in Q. By Lemma 4.3, only vertices with the *same* $auxT$ label might be processed earlier than v. Therefore, $auxT(v)$ will remain unchanged up to its processing. By Proposition 4.5, the label of v will not change also after that. □

Lemma 4.7 *When a vertex v is inserted into S, at the iteration end:*

1. *If v is an AND vertex, then* all *its predecessors are in S, and* (1) *is satisfied at it as an equality.*
2. *If v is an OR vertex with a finite label, then there* exists *its predecessor in S, satisfying* (2) *at it as an equality.*

Moreover, these properties are kept up to the end of the algorithm.

Proof: Let v be an AND vertex. If at the iteration beginning all predecessors of v are in S, (1) is satisfied at v as equality, by (5). In this case, the value of $auxT(v)$, and hence that of $T(v)$ cannot be changed during the iteration, so the property is kept up to its end.

Otherwise, (1) will be satisfied at v as an inequality w.r.t. its predecessors in the current S, by (5) and since $T(v)$ may only increase during the iteration, at the iteration end. Let u be an arbitrary predecessor of v beyond S at the beginning of the iteration (there exists such a one). The description of processing edge (u, v), the fact that the $auxT$ label of an AND vertex can only increase, and Proposition 4.5 ensure that $final\_auxT(v) \geq final\_auxT(u) + d(u, v) \geq final\_auxT(u)$. Since v is added to S, the value of $final\_auxT(v)$ is minimal, and hence the above two inequalities hold as equalities. Therefore, also $final\_auxT(u)$ is minimal, and thus also u is added to S at this iteration. By the above equalities, $T(v) = T(u) + d(u, v)$. This means that (1) will be satisfied at v as an equality at the iteration end.

Let now v be an OR vertex with a finite label. If $T(v)$ does not change during the iteration, the statement of the lemma follows from (6). Otherwise, at step 4 when processing some vertex u, $auxT(v)$ is changed to be equal to $auxT(u)$, and $d(u, v) = 0$. Since $final\_auxT(v)$ is minimal, by Corollary 4.6 and Proposition 4.5(2), also $final\_auxT(u)$ is minimal. Thus, also u is added to S at that iteration, ensuring the statement of the lemma.

These properties keep to hold up to the end of the algorithm, since S only grows and the T labels of its vertices never change. □

Lemma 4.8 *The $auxT$ labels produced at the iteration, together with the T labels on S, form a feasible solution.*

Proof: After **Zero OR Cycles**, S is initialized by the vertices in V^{OR-ZC} with zero labels. By definition, each vertex $v \in S$ has a predecessor $u \in S$,

such that $T(u) = 0$ and $d(u, v) = 0$. Hence before the first iteration, the T labels on S form a feasible solution on $G(S)$. [[??? where (1) and (2) are satisfied as equalities.]]

We prove the lemma by induction on the number of iterations of the outer **while** loop. For any non-first iteration I, let us assume that the claim of the lemma is correct at the previous iteration. By the induction hypothesis and Lemma 4.7, also at the beginning of I the T labels on S form a feasible solution on $G(S)$. [[??? where (1) and (2) are satisfied as equalities.]] In particular, in the boundary case of an OR vertex in S labeled by infinity, (2) is satisfied trivally.

It is easy to see that by Lemma 4.7, for each vertex belonging to S at the beginning of the iteration, the validity of the lemma(???) is maintained. Indeed, for an AND vertex nothing changes, while the right part of (2) can only decrease.

For every vertex $v \in S$ at the beginning of the iteration, it is easy to see that the validity of the lemma is maintained. Indeed, from Lemma 4.7 we see that for every AND vertex that was in S, all of its predecessors are in S, and for every OR vertex there exists at least one predecessor in S. Since at the beginning of the iteration the labels of vertices in S had feasible values, changing the $auxT$ labels of vertices in $V \setminus S$ cannot change the feasibility of these vertices. ??????? OR vertices ??????????

Let us prove that for every vertex $v \notin S$ at the beginning of the iteration, the lemma is valid at the end of the iteration. At any vertex labeled infinity, the problem conditions are trivially satisfied. So, we may concentrate on the AND vertices without positive entering edges, and on the OR vertices 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.

By Proposition 4.5, we scan zero edges in the non-increasing order of the $auxT$ label of their initial vertices. If v is an AND vertex, the $auxT$ label of v is given by the first scanned edge entering v. After that, all edges entering v bring equal or lesser values. Hence, the maximum condition (1) at v is satisfied. If v is an OR vertex, the $auxT$ label of v is defined by the last considered entering edge, so all other edges bring to it greater or equal values. Hence, the minimum condition (2) at v is satisfied. □

Now we prove the main statement.

Proposition 4.9 *Each iteration arrives at $m = t_{min}$, while the vertices*

inserted into S with $auxT$ equal to m form L_{min}.

Proof:

By Lemma 4.8, the $auxT$ labels given by the algorithm are greater or equal than the optimal labels. As a consequence, for any vertex 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 vertex $v \in L_{min}$ 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 vertex in L_{min}, in the chronological order, given the final $auxT$ value greater than t_{min}.

Case 1 : *v is an AND vertex.* By Observation **??**(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 **??**(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 vertex.* 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 **??**(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. □

5 Data Structure and Time Complexity

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

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

Data Structure: We maintain a priority queue of the vertices 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 vertex has label ∞ and any AND vertex 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 Appendix: Discussion on Related Work

Graphs considered by Knuth in [6] are bipartite, where the parts are the sets of AND and OR vertices, and such that there is a single out-coming edge from each AND vertex. A transformation of a general graph to an equivalent bipartite one without expanding the problem is suggested in [1]. Taking care of AND vertices with multiple out-coming edges is more heavy. Any reduction from the general case to the case of [6], known to us, leads to a substantial increase of the graph size, and thus of the time bound. One of the ways is as follows. Let v be an AND vertex with out-coming edges $(v, w_1), \ldots, (v, w_k)$. We add new vertices w and $v_1, \ldots v_k$, 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 vertices of the new graph increases by the total number of *edges* out-coming from AND vertices in the original graph.

Another specifics of Knuth's setting is related to the essence of the application in [6]: the events in any solution are essentially ordered by the causal relation. This property is used in proofs in [6], which prevents applying these proofs to our setting.

Note that the setting of Knuth is more general than that in [5, 4] in the definition of the precedence restriction for an AND vertex. Instead of

the maximum function only, a more general class of so called "superior" functions is considered.

The class of graphs considered by Moehring et al. in [7] is symmetric, in a sense, to the class considered in [6]: there is a single edge out-coming from each OR vertex. For a general graph, a reduction similar to that described above may be applied. As a result, the number of OR vertices increases by the number of edges out-coming from OR vertices in the original graph. Since the number of OR vertices is a factor in the time bound provided in [7], the bound becomes $O(|E|^2)$. We are not aware of a better time bound of the algorithm [7] in a general case.

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. (conference paper ????)

[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, R. Rivest and C. Stein. *Introduction to Algorithms*, McGraw-Hill, 2001.

[4] E. A. Dinic. The fastest algorithm for the PERT problem with AND- and OR-vertices (the new-product-new technology problem). In *Proc. of the Mathematical Programming Society Conference on Integer Programming and Combinatorial Optimization (IPCO'90), Waterloo, Canada*, R. Kannan and W. R. Pulleyblank eds., Univ. of Waterloo Press, 1990, 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. (Accepted to *SIAM J.of Computing* in 2002.) (???? - reference to SIAM)