Simplifying Activity-On-Edge Graphs

David Eppstein

University of California, Irvine, CA, United States
eppstein@uci.edu

Daniel Frishberg

University of California, Irvine, CA, United States
dfrishbe@uci.edu

Elham Havvaei

University of California, Irvine, CA, United States
ehavvaei@uci.edu

Abstract

We formalize the simplification of *activity-on-edge* graphs used for visualizing project schedules, where the vertices of the graphs represent project milestones, and the edges represent either tasks of the project or timing constraints between milestones. In this framework, a timeline of the project can be constructed as a leveled drawing of the graph, where the levels of the vertices represent the time at which each milestone is scheduled to happen. We focus on the following problem: given an activity-on-edge graph representing a project, find an equivalent activity-on-edge graph—one with the same critical paths—that has the minimum possible number of milestone vertices among all equivalent activity-on-edge graphs. We provide an $O(mn^2)$-time algorithm for solving this graph minimization problem.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases directed acyclic graph, activity-on-edge graph, critical path, project planning, milestone minimization, graph visualization

Digital Object Identifier 10.4230/LIPIcs.SWAT.2020.24

Related Version A related version of the paper is available at `https://arxiv.org/abs/2002.01610`.

Funding This work was supported in part by NSF grants CCF-1618301 and CCF-1616248.

1 Introduction

The *critical path method* is used in project modeling to describe the tasks of a project, along with the dependencies among the tasks; it was originally developed as PERT by the United States Navy in the 1950s [18]. A dependency graph is used to identify bottlenecks, and in particular to find the longest path among a sequence of tasks, where each task has a required length of time to complete (this is known as the *critical path*).

In this paper we consider a phase in planning a given project in which we do not yet know the time lengths of each task. We are interested in the problem of visualizing an abstract timeline of the potential critical paths (i.e., paths that could be critical depending on the lengths of the tasks) of the project, represented abstractly as a partially ordered set of tasks. The most common method of visualizing partially ordered sets, as an *activity-on-node graph* (a transitively reduced directed acyclic graph with a vertex for each task) is unsuitable for this aim, because it represents each task as a point instead of an object that can extend over a span of time in a timeline. To resolve this issue, we choose to represent each task as an edge in a directed acyclic graph. In this framework, the endpoints of the task edges have a natural interpretation, as the *milestones* of the project to be scheduled. Additional *unlabeled edges* do not represent tasks to be performed within the project, but constrain certain pairs of milestones to occur in a certain chronological order. The resulting *activity-on-edge graph*

17th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2020).
Editor: Susanne Albers; Article No. 24; pp. 24:1–24:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

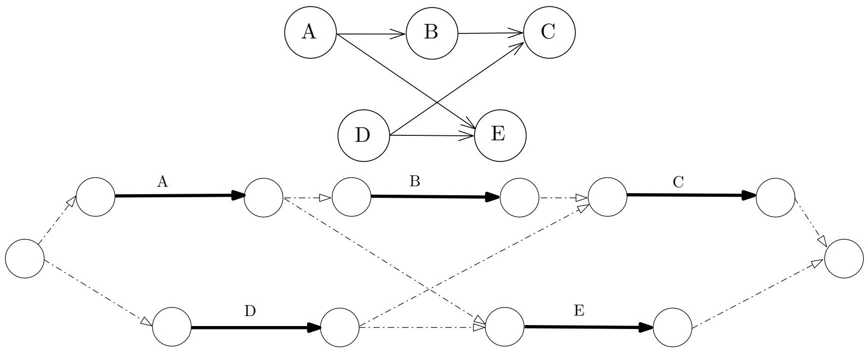


Figure 1 An activity-on-node graph, above, and its naively expanded activity-on-edge graph, below, with solid arrows as task edges and empty arrows as unlabeled edges.

can then be drawn in standard upward graph drawing style [1, 6, 8, 10, 11]. Alternatively, once the lengths of the tasks are known and the project has been scheduled, this graph can be drawn in leveled style [12, 16], where the level of each milestone vertex represents the time at which it is scheduled.

It is straightforward to expand an activity-on-node graph into an activity-on-edge graph by expanding each task vertex of the activity-on-node graph into a pair of milestone vertices connected by a task edge, with the starting milestone of each task retaining all of the incoming unlabeled edges of the activity-on-node graph and the ending milestone retaining all of the outgoing edges. It is convenient to add two more milestones at the start and end of the project, connected respectively to all milestones with no incoming edges and from all milestones with no outgoing edges. The size of the resulting activity-on-edge graph is linear in the size of the activity-on-node graph. An example of such a transformation is depicted in Figure 1.

However, the graphs that result from this naive expansion are not minimal. Often, one can merge some pairs of milestones (for instance the ending milestone of one task and the starting milestone of another task) to produce a simpler activity-on-edge graph (such as the one for the same schedule in Figure 2). Despite having fewer milestones, this simpler graph can be equivalent to the original, in the sense that its potential critical paths (maximal sequences of tasks that belong to a single path in the graph) are the same. By being simpler, this merged graph should aid in the visualization of project schedules. In this paper we formulate and provide an $O(mn^2)$-time algorithm (where n is the number of milestones and m is the number of unlabeled edges) for the problem of optimal simplification of activity-on-edge graphs.

1.1 New Results

We describe a polynomial-time algorithm that, given an activity-on-edge graph (i.e., a directed acyclic graph with a subset of its edges labeled as tasks), produces a directed acyclic graph that preserves the potential critical paths of the graph and has the minimum possible number of vertices among all critical-path-preserving graphs for the given input. Our algorithm is agnostic about the weights of the tasks. In more general terms, the resulting graph has the following properties:

- The task edges in the given graph correspond one-to-one with the task edges in the new graph.

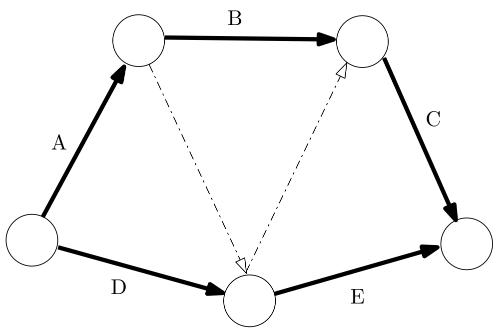


Figure 2 A simplification of the graph from Figure 1.

- The new graph has the same dependency (reachability) relation among task edges as the original graph.
- The new graph has the same potential critical paths as the original graph.
- The number of vertices of the graph is minimized among all graphs with the first three properties.

Our algorithm repeatedly applies a set of local reduction rules, each of which either merges a pair of adjacent vertices or removes an unlabeled edge, in arbitrary order. When no rule can be applied, the algorithm outputs the resulting graph.

We devote the rest of this section to related work and then describe the preliminaries in Section 2. We then present the algorithm in Section 3 and show in Section 4 that its output preserves the potential critical paths of the input, and in Section 5 that it has the minimum possible number of vertices. We also show that the output is independent of the order in which the rules are applied. We discuss the running time in Section 6 and conclude with Section 7.

1.2 Related work

Constructing clear and aesthetically pleasing drawings of directed acyclic graphs is an old and well-established task in graph drawing, with many publications [5,6,13,19]. The work in this line that is most closely relevant for our work involves upward drawings of unweighted directed acyclic graphs [1,8,10,11] or leveled drawings of directed acyclic graphs that have been given a level assignment [12,16] (an assignment of a y-coordinate to each vertex, for instance representing its height on a timeline).

Although multiple prior publications use activity-on-edge graphs [3,7,15,17] and even consider graph drawing methods specialized for these graphs [20], we have been unable to locate prior work on their simplification. This problem is related to a standard computational problem, the construction of the transitive reduction of a directed acyclic graph or equivalently the covering graph of a partially ordered set [2]. We note in addition our prior work on augmenting partially ordered sets with additional elements (preserving the partial order on the given elements) in order to draw the augmented partial order as an upward planar graph with a minimum number of added vertices [9].

The PERT method may additionally involve the notion of "float", in which a given task may be delayed some amount of time (depending on the task) without any effect on the overall time of the project [4,14]. We do not consider constraints of this form in the present work, although the unlabeled edges of our output can in some sense be seen as serving a similar purpose.

2 Preliminaries

We first define an activity-on-edge graph. The graph can be a multigraph to allow tasks that can be completed in parallel to share both a start and end milestone when possible.

▶ **Definition 1.** *An* activity-on-edge graph (AOE) *is a directed acyclic multigraph $G = (V, E)$, where a subset of the edges of E, denoted $\mathcal{T}$, are labeled as* task edges*. The labels, denoting tasks, are distinct, and we identify each edge in $\mathcal{T}$ with its label.*

▶ **Definition 2.** *Given an AOE G with tasks $\mathcal{T}$, for all $T \in \mathcal{T}$, let $\mathrm{St}_G(T)$ be the start vertex of T, and let $\mathrm{End}_G(T)$ be the end vertex of T.*

When the considered graph is clear from context, we omit the subscript G and write $\mathrm{St}(T)$ and $\mathrm{End}(T)$. It may be that $\mathrm{St}(T) = \mathrm{St}(T')$, or $\mathrm{End}(T) = \mathrm{End}(T')$, or $\mathrm{End}(T) = \mathrm{St}(T')$ with $T \neq T'$.

To define potential critical paths formally, we introduce the following notation.

▶ **Definition 3.** *Given an AOE G with tasks $\mathcal{T}$, for all $T, T' \in \mathcal{T}$ with $T \neq T'$, say that T* has a path to T' *in G if there exists a path from $\mathrm{End}(T)$ to $\mathrm{St}(T')$, or if $\mathrm{End}(T) = \mathrm{St}(T')$, and write $T \rightsquigarrow_G T'$.*

▶ **Definition 4.** *Given an AOE G with tasks $\mathcal{T}$, a* potential critical path *is a sequence of tasks $P = (T_1, \ldots, T_k)$, where for all $i = 1, \ldots, k-1$, $T_i \rightsquigarrow_G T_{i+1}$, and where P is not a subsequence of any other sequence with this property.*

Our algorithm will apply a set of transformation rules to an input AOE of a *canonical* form.

▶ **Definition 5.** *A* canonical AOE *is an AOE which is naively expanded from an activity-on-node graph.*

Every AOE G can be transformed into a canonical AOE with the same reachability relation on its tasks. First, we start by computing the reachability relation of the tasks. The transitive closure of the resulting reachability matrix gives an activity-on-node graph (which is quadratic, in the worst case, in the size of the original AOE). Then, this activity-on-node graph can be converted to a canonical AOE as described in Section 1.

▶ **Definition 6.** *Two AOE graphs G and H are* equivalent*, i.e. $G \equiv H$, if G and H have the same set of tasks—i.e., there is a label-preserving bijection between the task edges of G and those of H—and, with respect to this bijection, G and H have the same set of potential critical paths.*

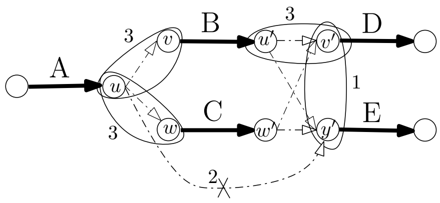
▶ **Definition 7.** *An AOE G is* optimal *if G minimizes the number of vertices for its equivalence class: i.e., if for every AOE $H \equiv G$, $|V(H)| \geq |V(G)|$.*

We now formally define our problem.

Problem 1. *Given a canonical AOE G, find some optimal AOE H with $H \equiv G$.*

3 Simplification Rules

Our algorithm takes a canonical AOE and greedily applies a set of rules until no more rules can be applied. Given an AOE $G = (V, E)$ and given two distinct vertices $u, v \in V$, the simplification rules used by our algorithm are:



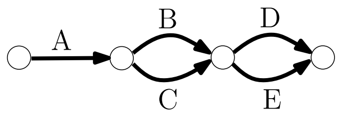


Figure 3 On the left, an AOE in which each of rules 1-3 can be applied, and on the right, the corresponding graph output by the algorithm.

1. if u and v have no outgoing task edges and have precisely the same outgoing neighbors, merge u and v. Symmetrically, if u and v have no incoming task edges and have precisely the same incoming neighbors, merge u and v.
2. If u has an unlabeled edge to v, and u has another path to v, remove the edge (u, v).
3. If u has an unlabeled edge to v and the following conditions are satisfied, merge u and v:
   - rule 2 is not applicable to the edge (u, v).
   - if u has an outgoing task, then v has no incoming edge other than (u, v).
   - if v has an incoming task, then u has no outgoing edge other than (u, v).
   - every incoming neighbor of v has a path to every outgoing neighbor of u.

Figure 3 depicts an AOE graph and the graph output by the algorithm after applying all possible rules. Vertices u and v can be merged by rule 3, since there is no other path from u to v to apply rule 2 (satisfying the first condition in the application of rule 3), u and v have no outgoing and no incoming task, respectively, and v has no incoming neighbor other than u. Therefore, the conditions of rule 3 are true (the second and third hold vacuously). Further, vertices u' and v' can be merged since the first three conditions for applying rule 3 are satisfied and there exists a path from w' to y', satisfying the last condition.

It will be convenient for the proofs in Section 5 to give a name to the output of the algorithm:

▶ **Definition 8.** *An* output AOE*, denoted $\mathcal{A}$, is any AOE obtained from a canonical AOE G by a sequence of applications of rules 1, 2, and 3, to which none of these rules can still be applied.*

We will show (Theorem 19) that $\mathcal{A}$ does not depend on the order in which the rules are applied.

4 Correctness

In this section we prove the correctness of our algorithm (its output graph is equivalent to its input graph).

We begin with preserving potential critical paths. We show that the rules never change the existence or nonexistence of a path from one task to another, and that this implies preservation of potential critical paths.

▶ **Lemma 9.** *Given two AOEs G and H with the same set of tasks $\mathcal{T}$, G and H have the same reachability relation $\rightsquigarrow$ on the tasks if and only if $G \equiv H$.*

Proof. Trivially, we have $T \rightsquigarrow_G T'$ (or $T \rightsquigarrow_H T'$) if and only if T is earlier than T' in some potential critical path of G (or H). Therefore, preservation of potential critical paths is equivalent to preservation of the reachability relation. ◀

▶ **Lemma 10.** *The output of the algorithm is equivalent to its input.*

Proof. We show the invariant that given tasks T and T', $T \rightsquigarrow T'$ at a given iteration of the algorithm if and only if $T \rightsquigarrow T'$ at the next iteration. From this, and from the fact that the rules never change the set of tasks, it follows that the output of the algorithm has the same reachability relation on its tasks as the input, and then the lemma follows from Lemma 9.

The invariant is true because merging a pair of vertices (rules 1 and 3) never disconnects a path, and no edge is ever removed (by rule 2) between two vertices unless another path exists between the two vertices. In particular, the end vertex of T still has a path to the start vertex of T' after the application of any of the rules.

For the other direction, removing an edge never introduces a new path. Furthermore, if vertices u and v are merged by applying rule 1, and if some vertex w has a path to some vertex z through the newly merged uv, then the condition of rule 1 ensures that w has a path, through u or v, to z before the merge. Similarly, suppose u and v are merged by applying rule 3. Then if w has a path to z through uv, then (abusing notation) either $w \rightsquigarrow u$ and $v \rightsquigarrow z$ before the merge, so $w \rightsquigarrow z$ (via the edge (u, v)), or for some incoming neighbor x of v and outgoing neighbor y of u, $w \rightsquigarrow x$ and $y \rightsquigarrow z$. In this case, by the conditions of the rule, $w \rightsquigarrow z$ before the merge. ◀

▶ **Lemma 11.** *Any intermediate graph that results from applying rules of the algorithm to an input canonical AOE graph, is acyclic.*

Proof. Given Definition 1 and Definition 5, the canonical AOE input G is acyclic. Now we show none of the rules can create a cycle after being applied to an intermediate acyclic graph G'. This is obvious for rule 2 as it removes edges. Suppose for a contradiction that merging vertices u and v creates a cycle. The cycle must involve the new vertex resulting from the merge. For rule 1, this implies the existence of a cycle in G' either from u or v to itself which is a contradiction. For rule 3, it implies the existence of a cycle in G' including the unlabeled edge (u, v) or a cycle including an incoming neighbor of v and an outgoing neighbor of u, which is a contradiction. ◀

▶ **Corollary 12.** *Any graph $\mathcal{A}$ output by the algorithm is acyclic.*

5 Optimality

In this section we prove the optimality of our algorithm: it uses as few vertices as possible. Let $\mathcal{A}$ be any output AOE. Let *Opt* be any optimal AOE such that $\mathcal{A} \equiv Opt$. Our proof relies on an injective mapping from the vertices of $\mathcal{A}$ to the vertices of *Opt*. The existence of this mapping shows that $\mathcal{A}$ has at most as many vertices as *Opt*, and therefore has the optimal number of vertices. Once we have identified the vertices of $\mathcal{A}$ with the vertices of *Opt* in this way, we show that, for a given input, any two graphs output by the algorithm (but not necessarily *Opt*) must have the same unlabeled edges. Since the task edges are determined, and since the injective mapping to *Opt* determines the vertices, determining the unlabeled edges implies the order-independence of our algorithm's choice of simplification rules.

Before defining the mapping between $\mathcal{A}$ and *Opt*, we establish some facts about the structure of $\mathcal{A}$.

▶ **Lemma 13.** *For every unlabeled edge (u, v) in any output AOE $\mathcal{A}$, there exist tasks T and T' such that $u = \mathrm{End}(T)$ and $v = \mathrm{St}(T')$.*

Proof. By Definition 8, $\mathcal{A}$ is produced by the algorithm from some canonical AOE G. This property holds for G by Definition 5. As every rule of the algorithm either removes an unlabeled edge or merges two vertices, and never creates a new edge or vertex, the proof is complete. ◀

▶ **Corollary 14.** *Every vertex in an output AOE $\mathcal{A}$ has an incident task edge.*

We can now define a mapping from the vertices of $\mathcal{A}$ to those of *Opt*:

▶ **Definition 15.** *Given an output AOE $\mathcal{A}$ with task set $\mathcal{T}$, and given an optimal AOE Opt with $\mathcal{A} \equiv Opt$, let $M : V(\mathcal{A}) \to V(Opt)$ be the following mapping: for every $v \in V(\mathcal{A})$:*

- *Let $M(v) = \mathrm{St}_{Opt}(T)$, for some $T \in \mathcal{T}$ for which $v = \mathrm{St}_{\mathcal{A}}(T)$, if such a task exists.*
- *Let $M(v) = \mathrm{End}_{Opt}(T)$, where $v = \mathrm{End}_{\mathcal{A}}(T)$, otherwise.*

As shown in Corollary 14, every vertex in $\mathcal{A}$ has an incident task edge, and by Definition 6, $\mathcal{A}$ and *Opt* have the same set of tasks. Therefore, this mapping is well-defined (up to its arbitrary choices of which task to use for each v). To prove that M is injective, we will use the fact that since $\mathcal{A} \equiv Opt$, $\mathcal{A}$ and *Opt* have the same reachability relation (by Lemma 9 and Lemma 10).

The heart of the proof that M is injective lies in showing that if two tasks do not share a vertex in $\mathcal{A}$, the corresponding tasks also do not share the corresponding vertices in *Opt*. From this it follows that M cannot map distinct vertices in $\mathcal{A}$ to the same vertex in *Opt*.

▶ **Lemma 16.** *Given an output AOE $\mathcal{A}$, and an optimal AOE $Opt \equiv \mathcal{A}$, with task set $\mathcal{T}$, let T and T' be two distinct tasks in $\mathcal{T}$. If $\mathrm{St}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, then $\mathrm{St}_{Opt}(T) \neq \mathrm{St}_{Opt}(T')$. If $\mathrm{End}_{\mathcal{A}}(T) \neq \mathrm{End}_{\mathcal{A}}(T')$, then $\mathrm{End}_{Opt}(T) \neq \mathrm{End}_{Opt}(T')$.*

Proof. Suppose for a contradiction that $\mathrm{St}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, but $\mathrm{St}_{Opt}(T) = \mathrm{St}_{Opt}(T')$ (the other case is symmetrical). Let $u = \mathrm{St}_{\mathcal{A}}(T)$ and $v = \mathrm{St}_{\mathcal{A}}(T')$. Consider the following (exhaustive) cases for u and v:

1. u and v have no incoming edges
2. u or v has an incoming unlabeled edge, but neither u nor v has an incoming task edge
3. u or v has an incoming task edge A

In case 1, applying rule 1 results in merging u and v. However, since $\mathcal{A}$ is the output of the algorithm, no rule can be applied to $\mathcal{A}$. This is a contradiction.

In case 2, u and v cannot have the same incoming neighbors or else rule 1 would apply. We may assume without loss of generality that there exist a vertex w and an unlabeled edge (w, u), such that the edge (w, v) does not exist. By Lemma 13, there exists a task A where $w = \mathrm{End}_{\mathcal{A}}(A)$. Since $A \rightsquigarrow_{\mathcal{A}} T$ and $\mathcal{A} \equiv Opt$ (by Lemma 10), then by Lemma 9, $A \rightsquigarrow_{Opt} T$, so $A \rightsquigarrow_{Opt} T'$, since $\mathrm{St}_{Opt}(T) = \mathrm{St}_{Opt}(T')$. Again by Lemma 9, $A \rightsquigarrow_{\mathcal{A}} T'$, so there is a path P from w to v. If $|P| = 1$, then this contradicts that (w, v) does not exist. Suppose $|P| > 1$. Then we show there exist some vertex $w' \neq w$ and an unlabeled edge (w', v). The following cases are exhaustive:

(a) P contains a path from u to v. As such a path to v exists and v has no incoming task edge, there exist a vertex w' and an unlabeled edge (w', v) $(w' \neq u)$, not belonging to P unless rule 3 can be applied to vertex v and its incoming neighbor in path P .

(b) P does not contain a path from u to v. As $|P| > 1$, an unlabeled edge (w', v) belonging to path P exists.

Given the existence of (w', v), by Lemma 13, there exists a task B where $w' = \mathrm{End}_{\mathcal{A}}(B)$. $B \rightsquigarrow_{\mathcal{A}} T'$, so by reasoning similar to the above, $B \rightsquigarrow_{\mathcal{A}} T$. Then, one can apply rule 2

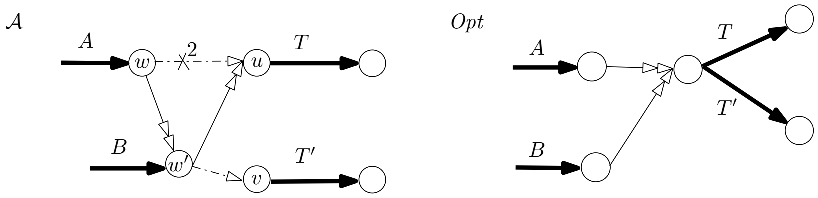


Figure 4 Lemma 16, case 2, subcase b (double arrows indicate paths).

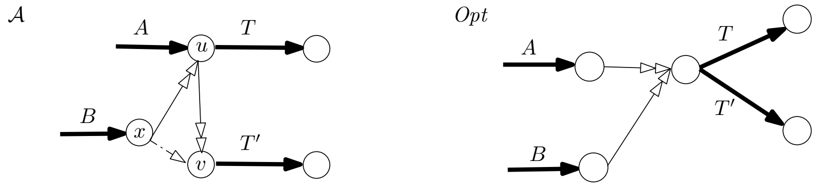


Figure 5 Lemma 16, case 3, subcase b.

and either remove edge (w', v) in case a or (w, u) in case b (Figure 4); this contradicts the definition of $\mathcal{A}$.

In case 3, we can assume without loss of generality that u has an incoming task A; consequently, $u = \mathrm{End}_{\mathcal{A}}(A)$. Then, by Lemma 9, we have $A \rightsquigarrow_{Opt} T'$ and thus $A \rightsquigarrow_{\mathcal{A}} T'$ via a path P. Consider the following cases for P:

(a) P contains a task edge B

(b) P is a sequence of unlabeled edges

In case a, by Lemma 9 $B \rightsquigarrow_{Opt} T'$, and thus $B \rightsquigarrow_{Opt} T$, and therefore $B \rightsquigarrow_{\mathcal{A}} T$. This creates a cycle between u and $\mathrm{End}_{\mathcal{A}}(B)$, contradicting Corollary 12.

In case b, illustrated in Figure 5, since rule 3 cannot be applied (if it could, this would contradict the definition of $\mathcal{A}$), there exist a vertex x not on the path from u to v, and an edge (x, v) (a task edge or an unlabeled edge). Therefore, there exists a task B where either $v = \mathrm{End}_{\mathcal{A}}(B)$ or by Lemma 13, $x = \mathrm{End}_{\mathcal{A}}(B)$. Considering *Opt* and applying Lemma 9, $B \rightsquigarrow_{Opt} T$ so $B \rightsquigarrow_{\mathcal{A}} T$. This path either creates a cycle in $\mathcal{A}$ or allows for removing edge (x, v) by rule 2, which is a contradiction.

Thus if $\mathrm{St}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, then $\mathrm{St}_{Opt}(T) \neq \mathrm{St}_{Opt}(T')$. ◀

▶ **Lemma 17.** *Given an output AOE $\mathcal{A}$, and an optimal AOE $Opt \equiv \mathcal{A}$, with task set $\mathcal{T}$, let T and T' be two distinct tasks in $\mathcal{T}$. If $\mathrm{End}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, then $\mathrm{End}_{Opt}(T) \neq \mathrm{St}_{Opt}(T')$.*

Proof. The proof, which is in Appendix A.1, uses essentially the same approach as the proof of Lemma 16: supposing that the two vertices are the same, then using the fact that $\mathcal{A}$ and *Opt* have the same reachability relation on their tasks, and the definition of $\mathcal{A}$ as having no more rules to apply, to derive a contradiction. ◀

There is one remaining technicality: we have defined an optimal AOE as being acyclic; the question arises whether one could reduce the number of vertices by allowing (unlabeled) cycles. However, this is not the case; it is easy to see that any unlabeled cycle can be merged into one vertex, reducing the number of vertices, without changing the reachability relation on the tasks.

We are ready to prove our main results.

▶ **Theorem 18.** *Given a canonical AOE G, the algorithm produces an optimal AOE $Opt \equiv G$.*

Proof. Let $\mathcal{A}$ be the output AOE produced by the algorithm on G. Given any optimal AOE Opt and $\mathcal{A}$, the mapping M in Definition 15 is injective: suppose for a contradiction that u and v are distinct vertices in $\mathcal{A}$, and $w = M(u) = M(v)$. Then by the definition of M, either u, v, and w have the same incoming task, or u, v, and w have the same outgoing task, or there exist tasks T and T' such that (without loss of generality) $u = \mathrm{End}_{\mathcal{A}}(T), v = \mathrm{St}_{\mathcal{A}}(T')$, and $\mathrm{End}_{Opt}(T) = w = \mathrm{St}_{Opt}(T')$. By Lemmas 16 and 17, all three of these cases imply that $u = v$.

Therefore, $|V(Opt)| = |V(\mathcal{A})|$. Furthermore, $\mathcal{A} \equiv G$, by Lemma 10. The theorem follows. ◀

▶ **Theorem 19.** *Given an input, the algorithm produces the same output regardless of the order in which the rules are applied.*

Proof. As stated earlier, all task edges of an input canonical AOE G are present in any output of the algorithm and the mapping determines the vertices. Therefore, it suffices to show that any two graphs output by the algorithm have the same set of unlabeled edges. Suppose for a contradiction that $\mathcal{A}_1$ and $\mathcal{A}_2$ are two distinct outputs of the algorithm, resulting from applying different sequences of rules. By Theorem 18, the algorithm always produces an optimal AOE. Therefore, $|V_{\mathcal{A}_1}| = |V_{\mathcal{A}_2}| = |V_{Opt}|$. Since $\mathcal{A}_1 \neq \mathcal{A}_2$, there is an unlabeled edge (u, v) in $\mathcal{A}_1$ (without loss of generality) that is not in $\mathcal{A}_2$. By Lemma 13, there exist task edges T and T' such that $u = \mathrm{End}_{\mathcal{A}_1}(T)$ and $v = \mathrm{St}_{\mathcal{A}_1}(T')$. We have $T \rightsquigarrow_{\mathcal{A}_1} T'$. Since by Lemma 9 and Lemma 10, $\mathcal{A}_1$ and $\mathcal{A}_2$ both preserve the reachability relation of the tasks of G, we have $T \rightsquigarrow_{\mathcal{A}_2} T'$. Consider the cases for path P from T to T' in $\mathcal{A}_2$:

1. There exists a task A in P other than T and T'.
2. Path P is a sequence of unlabeled edges.

In case 1, we have $T \rightsquigarrow_{\mathcal{A}_2} A \rightsquigarrow_{\mathcal{A}_2} T'$ and therefore, $T \rightsquigarrow_{\mathcal{A}_1} A \rightsquigarrow_{\mathcal{A}_1} T'$. Then by rule 2, one can remove the edge (u, v), which contradicts the definition of $\mathcal{A}_1$.

In case 2, the length of P is at least two, and P contains a vertex w. By Lemma 13, there exist tasks A and B where $w = \mathrm{End}_{\mathcal{A}_2}(A) = \mathrm{St}_{\mathcal{A}_2}(B)$. Now, since $\mathcal{A}_1 \equiv \mathcal{A}_2$, both graphs are optimal, and both graphs are outputs of the algorithm, Lemma 17 implies that $\mathrm{End}_{\mathcal{A}_1}(A) = \mathrm{St}_{\mathcal{A}_1}(B)$. Call this vertex x. Then there exists a path from u to v, through x, by Lemma 9, and one can remove the edge (u, v) by rule 2. This contradicts the definition of $\mathcal{A}_1$. ◀

It is tempting to imagine that Theorem 19 implies uniqueness of the optimal AOE. However, this is not the case: the unlabeled edges of an optimal AOE are not determined by our bijection. Figure 6 shows an optimal AOE graph that our algorithm cannot produce (because it violates Lemma 13). One way to see the optimality is to expand the graph naively into a canonical AOE, apply the algorithm, and verify that the resulting number of vertices is the same.

6 Analysis

Let n be the number of vertices in a canonical AOE (which is linear in the number of tasks), and m the number of unlabeled edges. (The number of task edges is $O(m)$.) There are at most $O(n + m)$ iterations in the algorithm, because each iteration either merges two vertices or removes an edge, by applying one of the three rules. This requires finding an edge to remove ($O(m)$ potential edges) or two vertices to merge ($O(n^2)$ potential pairs), then performing the merge or the removal. Intuitively, our algorithm runs in polynomial time as it takes polynomial time to find and apply a rule.

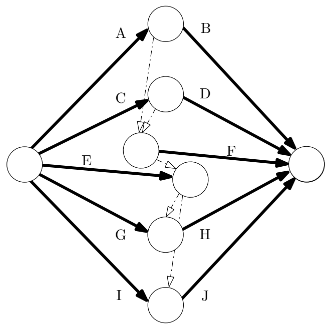


Figure 6 An optimal AOE graph that the algorithm cannot produce.

Algorithm 1 the proposed transformation algorithm.

Data: Canonical AOE G
Result: Optimal AOE *Opt*

while *true* **do**
    Initialize and compute the reachability matrix M;
    Remove, by rule 2, all unlabeled edges (u, v) where $M[u][v] = 2$;
    if *rule 1 applies* **then**
        apply rule 1;
    else if *rule 3 applies* **then**
        apply rule 3;
    else
        return the graph;

We provide a faster implementation of our algorithm than the naive approach. The algorithm transforms a canonical AOE graph G into an optimal AOE graph by applying rules 1, 2 or 3. For simplicity, we label the vertices $1, \ldots, n$. At each iteration, compute a reachability matrix M for the current graph. $M[u][v]$ indicates whether there exist zero, one, or more than one paths from u to v. In order to compute M, for all u and v initialize $M[u][v] = 1$ if the edge (u, v) exists. Then sort the vertices in topological order (such an ordering exists according to Lemma 11). For each vertex v in this order, and for each vertex u, set $M[u][v]$ to $\min(2, \sum_{w \in W}(M[u][w]))$, where W is the set of all vertices w such that either $w = v$ or there exists an edge (w, v). This procedure takes $O(nm)$ time. Algorithm 1 provides a summary.

Given the reachability matrix, an unlabeled edge (u, v) is removed by rule 2 in $O(1)$ time, if $M[u][v] \geq 2$. Therefore, checking rule 2 for all edges takes $O(m)$ time.

Without loss of generality, for rule 1, we only consider merges of pairs of vertices with the same outgoing neighbors. This requires, for each vertex u with no outgoing task edge, a sorted list of outgoing neighbors ($\mathrm{S}[u]$). To obtain such lists for all vertices, list unlabeled edges as pairs of vertices and sort all the pairs with two bucket sorts: first over the first elements of the pairs, then over the second elements. Breaking the sorted list into chunks of pairs with the same first element (say u), gives the outgoing neighbors of u, in the second elements of the pairs, in a numerically sorted order. This takes $O(m)$ time. Then find pairs of vertices to merge, if any exist: first, bucket sort vertices based on their out-degree. Vertices in different buckets cannot be merged by rule 1. For each bucket b containing vertices with degree d ($0 \leq d < n$), call MergeDetection(b, d):

```
Function MergeDetection(bucket a, i):
    if i = 0 then
        return bucket a
    else
        bucket sort vertices v of a based on S[v][i]
        foreach newly created bucket a' do
            MergeDetection(a', i − 1)
```

The vertices in each resulting bucket have the same outgoing neighbors and can be merged by rule 1. As each vertex with degree d appears in one bucket in each of $d+1$ iterations, this sort takes $O(\sum_v (deg(v))) = O(m)$ time. Upon merging vertices u and v, name the new vertex $\min(u, v)$.

To check rule 3, for each vertex v, compute $I(v)$: the intersection of the reachable sets of the incoming neighbors of v. This takes $O(mn)$ time.

Consider only those unlabeled edges (u, v) that meet the preconditions of rule 3 concerning the existence of outgoing and incoming tasks of u and v respectively. Test whether the last point in rule 3 applies to edge (u, v) by testing in $O(n)$ time whether all outgoing neighbors of u are in $I(v)$.

Computing the reachability matrix takes $O(mn)$ time, and using this matrix to check for rule 2 takes $O(m)$ time per iteration. Checking for rule 1 or 3 takes $O(mn)$ time per iteration. Further, the outer loop in Algorithm 1 runs at most n times as it either merges two vertices or returns the output. This gives a total complexity of $O(mn^2)$ for our algorithm.

7 Conclusion

Our algorithm reduces the visual complexity of an activity-on-edge graph, making it easier to understand bottlenecks in a project. The algorithm repeatedly applies simple rules and therefore can be implemented easily. We have shown that the algorithm runs in $O(mn^2)$ time. One question for future work is whether this analysis is tight. Another question is whether some other algorithm could achieve an optimal graph more efficiently.

Furthermore, one can measure the complexity of a graph in other ways. One question for future work is whether one can minimize the number of edges in an AOE graph in polynomial time. Another question is whether one can, in polynomial time, convert an AOE graph G into a graph that (i) has the same potential critical paths as G, and (ii) has a plane drawing with fewer edge crossings than all other graphs satisfying (i). It would also be interesting to implement this algorithm and run it on realistic graphs arising in project planning, and to evaluate the visual complexity of the resulting graphs in terms of the measures described above.

References

1 Sarmad Abbasi, Patrick Healy, and Aimal Rextin. Improving the running time of embedded upward planarity testing. *Information Processing Letters*, 110(7):274–278, 2010. doi:10.1016/j.ipl.2010.02.004.

2 A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972. doi:10.1137/0201008.

3 C. Alexander, D. Reese, and J. Harden. Near-critical path analysis of program activity graphs. In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 308–317. IEEE, 1994. doi:10.1109/mascot.1994.284406.

4 David Arditi and Thanat Pattanakitchamroon. Selecting a delay analysis method in resolving construction claims. *International J. Project Management*, 24(2):145–155, 2006. doi:10.1016/j.ijproman.2005.08.005.

5 Oliver Bastert and Christian Matuszewski. Layered drawings of digraphs. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*, pages 87–120. Springer-Verlag, 2001. doi:10.1007/3-540-44969-8_5.

6 Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall PTR, 1998.

7 Pranay Chaudhuri and Ratan K. Ghosh. Parallel algorithms for analyzing activity networks. *BIT*, 26(4):418–429, 1986. doi:10.1007/bf01935049.

8 Giuseppe Di Battista, Roberto Tamassia, and Ioannis G. Tollis. Area requirement and symmetry display of planar upward drawings. *Discrete and Computational Geometry*, 7(4):381–401, 1992. doi:10.1007/BF02187850.

9 David Eppstein and Joseph A. Simons. Confluent Hasse diagrams. *J. Graph Algorithms Appl.*, 17(7):689–710, 2013. doi:10.7155/jgaa.00312.

10 Ashim Garg and Roberto Tamassia. Upward planarity testing. *Order*, 12(2):109–133, 1995. doi:10.1007/BF01108622.

11 Ashim Garg and Roberto Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, 2001. doi:10.1137/S0097539794277123.

12 Patrick Healy, Ago Kuusik, and Sebastian Leipert. A characterization of level planar graphs. *Discrete Math.*, 280(1-3):51–63, 2004. doi:10.1016/j.disc.2003.02.001.

13 Patrick Healy and Nikola S. Nikolov. Hierarchical Graph Drawing. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 409–453. CRC Press, 2014.

14 Jerry L. Householder and Hulan E. Rutland. Who owns float? *J. Construction Engineering and Management*, 116(1):130–133, 1990. doi:10.1061/(ASCE)0733-9364(1990)116:1(130).

15 Wei Huang and Lixin Ding. Project-scheduling problem with random time-dependent activity duration times. *IEEE Transactions on Engineering Management*, 58(2):377–387, 2011. doi:10.1109/tem.2010.2063707.

16 Michael Jünger and Sebastian Leipert. Level Planar Embedding in Linear Time. In Jan Kratochvıl, editor, *Graph Drawing: 7th International Symposium*, volume 1731 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 1999. doi:10.1007/3-540-46648-7_7.

17 Vidyadhar G. Kulkarni. A compact hash function for paths in PERT networks. *Operations Research Letters*, 3(3):137–140, 1984. doi:10.1016/0167-6377(84)90005-1.

18 Donald G. Malcolm, John H. Roseboom, Charles E. Clark, and Willard Fazar. Application of a technique for research and development program evaluation. *Operations Research*, 7(5):646–669, 1959. doi:10.1287/opre.7.5.646.

19 Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Systems, Man, and Cybernetics*, 11(2):109–125, 1981. doi:10.1109/TSMC.1981.4308636.

20 Zhenming Xu. Automatic layout of information in the AOE network. In *2010 International Conference on Mechanic Automation and Control Engineering*. IEEE, June 2010. doi:10.1109/mace.2010.5536213.

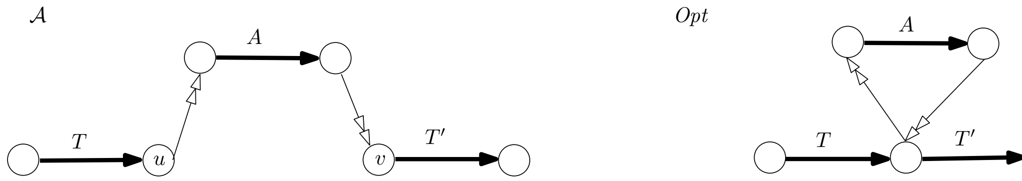


Figure 7 Lemma 17, case 1.

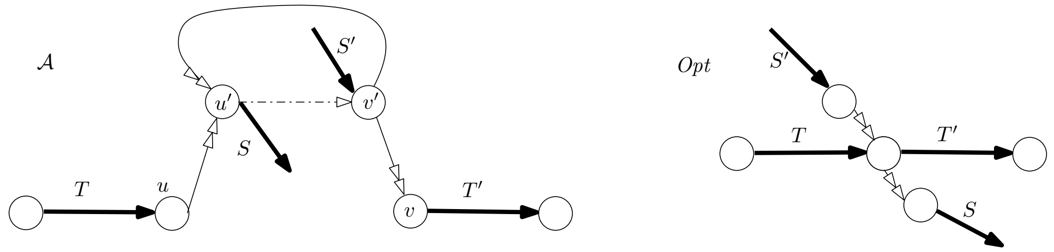


Figure 8 Lemma 17, case 2a.

A Appendix

A.1 Proof of Lemma 17

▶ **Lemma 17.** *Given an output AOE $\mathcal{A}$, and an optimal AOE $Opt \equiv \mathcal{A}$, with task set $\mathcal{T}$, let T and T' be two distinct tasks in $\mathcal{T}$. If $\mathrm{End}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, then $\mathrm{End}_{Opt}(T) \neq \mathrm{St}_{Opt}(T')$.*

Proof. Suppose for a contradiction that $u = \mathrm{End}_{\mathcal{A}}(T) \neq v = \mathrm{St}_{\mathcal{A}}(T')$, but $\mathrm{End}_{Opt}(T) = \mathrm{St}_{Opt}(T')$. Since $T \rightsquigarrow_{Opt} T'$, then by Lemma 9, $T \rightsquigarrow_{\mathcal{A}} T'$, so u has a path P to v. Consider the following possible cases for path P:

1. There exists a task A in P.
2. P only consists of unlabeled edges. Consider the cases for any unlabeled edge (u', v') in P:
   a. There exist incident tasks S and S', pointing away from and toward u' and v', respectively.
   b. There exists an incident unlabeled edge (u', w') pointing away from u' and an incident task S' pointing toward v'.
   c. There exists an incident task S pointing away from u' and an incident unlabeled edge (w', v') pointing toward v'.
   d. There exists an incident unlabeled edge (u', w') pointing away from u' and an incident unlabeled edge (x', v') pointing toward v'. Vertices u' and v' have no outgoing or incoming task edges, respectively.

These cases are exhaustive as path P either has a task or it is a sequence of unlabeled edges. Further, for case 2, suppose for an unlabeled edge (u', v'), none of the subcases of 2a, 2b, 2c and 2d holds. Then, by rule 3, one can merge vertices u' and v'; this contradicts the definition of $\mathcal{A}$.

In case 1, shown in Figure 7, we have $T \rightsquigarrow_{\mathcal{A}} A \rightsquigarrow_{\mathcal{A}} T'$ so by Lemma 9, $T \rightsquigarrow_{Opt} A \rightsquigarrow_{Opt} T'$. Therefore, there is a path in Opt (through A) from $\mathrm{End}_{Opt}(T)$ to $\mathrm{St}_{Opt}(T')$. Since $\mathrm{End}_{Opt}(T) = \mathrm{St}_{Opt}(T')$, this path creates a cycle in Opt. However, Opt is an AOE graph and is therefore acyclic by Definition 1.

In case 2a, shown in Figure 8, $S' \rightsquigarrow_{\mathcal{A}} T'$, so by Lemma 9, $S' \rightsquigarrow_{Opt} T'$, i.e. there is a path from S' to $\mathrm{St}_{Opt}(T')$. Similarly, $T \rightsquigarrow_{\mathcal{A}} T'$, so there is a path from $\mathrm{End}_{Opt}(T)$ to S. Since

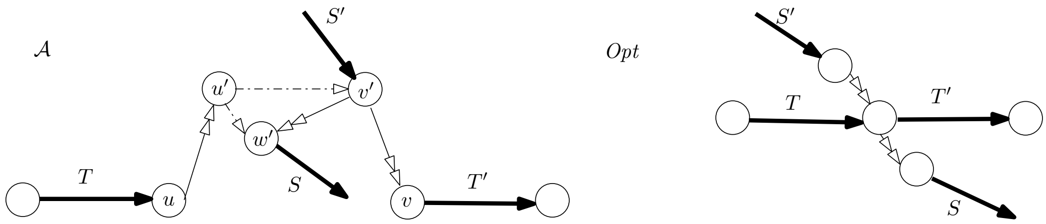


Figure 9 Lemma 17, case 2b.

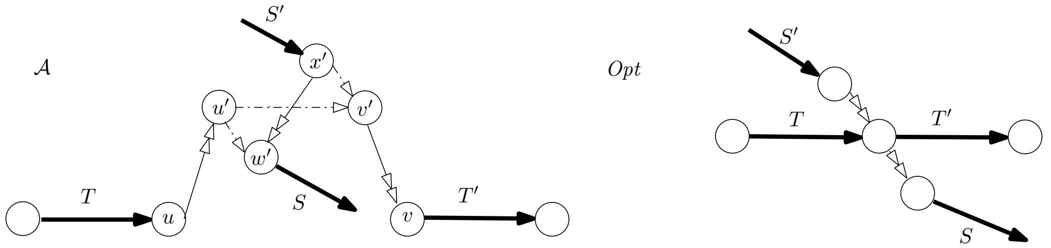


Figure 10 Lemma 17, case 2d.

$\mathrm{St}_{Opt}(T') = \mathrm{End}_{Opt}(T)$, this implies $S' \rightsquigarrow_{Opt} S$, so by Lemma 9, $S' \rightsquigarrow_{\mathcal{A}} S$. Therefore, there is a path in $\mathcal{A}$ from $\mathrm{End}_{\mathcal{A}}(S')$ to $\mathrm{St}_{\mathcal{A}}(S)$. This path creates a cycle in $\mathcal{A}$ and contradicts Corollary 12.

In case 2b, shown in Figure 9, by Lemma 13, there exists a task S where $w' = \mathrm{St}_{\mathcal{A}}(S)$. Since we have $T \rightsquigarrow_{\mathcal{A}} S$ and $S' \rightsquigarrow_{\mathcal{A}} T'$ and $\mathrm{End}_{Opt}(T) = \mathrm{St}_{Opt}(T')$, by Lemma 9, we have $S' \rightsquigarrow_{Opt} S$. Therefore, there is a path in $\mathcal{A}$ from $v' = \mathrm{End}_{\mathcal{A}}(S')$ to $w' = \mathrm{St}_{\mathcal{A}}(S)$. This path either creates a cycle between u' and v', contradicting Corollary 12 or by rule 2, one can remove edge (u', w'), which is a contradiction by the definition of $\mathcal{A}$.

Case 2c is almost identical to case 2b, and again leads to the existence of a path from S' to S (similarly defined), resulting in either a cycle or an application of rule 2 .

In case 2d, shown in Figure 10, by Lemma 13, there exist task edges S and S' where $w' = \mathrm{St}_{\mathcal{A}}(S)$ and $x' = \mathrm{End}_{\mathcal{A}}(S')$, and unlabeled edges (x', v') and (u', w'). We have $S' \rightsquigarrow_{Opt} S$, then by Lemma 9, $S' \rightsquigarrow_{\mathcal{A}} S$. Therefore, there is a path in $\mathcal{A}$ from $\mathrm{End}_{\mathcal{A}}(S')$ to $\mathrm{St}_{\mathcal{A}}(S)$. This path either creates a cycle between u' and v', contradicting Corollary 12 or by rule 3, one can merge u' and v' in $\mathcal{A}$, which is a contradiction by the definition of $\mathcal{A}$.

Thus if $\mathrm{End}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, then $\mathrm{End}_{Opt}(T) \neq \mathrm{St}_{Opt}(T')$. ◀