On Batch-Scheduling Dags for Internet-Based Computing

September 19, 2006

Grzegorz Malewicz
Dept. of Computer Science
University of Alabama
Tuscaloosa, AL 35487, USA
greg@cs.ua.edu

Arnold L. Rosenberg
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
rsnbrg@cs.umass.edu

Abstract

The process of scheduling computations for Internet-based computing presents challenges not encountered with more traditional platforms for parallel and distributed computing. The looser coupling among participating computers makes it harder to utilize remote clients well and also raise the specter of a kind of "gridlock" that ensues when a computation stalls because no new tasks are eligible for execution. This paper studies the problem of scheduling computation-dags in a manner that renders tasks eligible for allocation to remote clients (hence for execution) at the maximum possible rate. Earlier work has developed a framework for studying this problem when a new task is allocated to a remote client as soon as it returns the results from an earlier-allocated task. The proof in that work that many dags cannot be scheduled optimally within this scheduling paradigm demonstrated the need for a companion scheduling theory that addresses the scheduling problem for all computation-dags. A new, *batched,* scheduling paradigm for Internet-Computing is developed in this work. Although optimal batched schedules exist for every computation-dag, computing (successive steps of) such a schedule is shown to be NP-Hard, even for bipartite dags. In response, a polynomial-time algorithm is developed for producing optimal batched schedules for a rich family of dags that are constructed by "composing" tree-structured building-block dags. Finally, a fast heuristic schedule is developed for a class of "expansive" dags

1 Introduction

Earlier work [10, 12, 14] has developed the Internet-Computing (IC, for short) Pebble Game, a variant of the classical pebble games of [4, 11, 13], that abstracts the problem of scheduling computations having intertask dependencies for the several modalities of Internet-based computing—including Grid computing (cf. [1, 5, 6]), global computing (cf. [2]), and Web computing (cf. [9]). This Game was developed with the goal of formalizing the process of scheduling computations with intertask dependencies for Internet-based computing. The scheduling paradigm studied in [10, 12, 14] is that the IC-server allocates a task of the dag being computed to a remote client as soon as the task becomes eligible for allocation and the client becomes available for computation. The quality metric for schedules was to maximize the rate at which tasks were rendered eligible for allocation to remote clients, with the dual aim of (a) maximizing the utilization of available remote clients and (b) minimizing the likelihood of the "gridlock" that can arise when a computation stalls pending computation of already-allocated tasks. These three sources develop the conceptual framework of a theory of IC scheduling based on this scheduling paradigm.

The present study is motivated by the demonstration in [10] that there are simple computation-dags that do not admit any optimal IC schedule. (Intuitively, for such dags, any sequence of tasks that optimizes the number of eligible tasks after the first t steps of the computation is incompatible with every sequence that optimizes the number of eligible tasks after the first t' steps.) We respond in this paper by developing a companion scheduling theory (Section 2.2) in which every computation-dag admits an optimal schedule. This new theory is based on a *batched* scheduling paradigm, which relieves the Server from the chore of selecting a new task for allocation whenever a remote client becomes available for computation. Instead, we now assume that the Server collects requests for new tasks and then (either periodically or based on some trigger) allocates tasks for the collected requests in a batch. (This mode of operation may be inevitable if, say, tasks take extremely long to compute and enable many other tasks once completed.) The goal for the Server is to satisfy this batch of requests with a set of tasks whose execution will produce a maximal number of new eligible tasks. In contrast to the quality metric of [10, 12, 14], this new step-by-step metric can always be satisfied optimally. Moderating the news that optimality can always be achieved in the batched paradigm is our demonstration (in Section 3) that finding such a schedule for an arbitrary computation-dag—even a bipartite one—is NP-Hard, hence likely computationally intractable. We respond to this probable computational intractability (in Section 5) with a polynomial-time optimal algorithm for a rich family of dags that are constructed by "composing" certain tree-structured building-block dags. Since the preceding timing polynomial has high degree, we also develop (in Section 6) a fast heuristic schedule for a more restricted family of "expansive" dags, whose eligible-task production rate is within a factor of 4 of optimal.

Related work. The work most closely related to the present study appears in [7, 10, 12, 14]. The IC Pebble Game is introduced in [12, 14], and optimal schedules are identified for the structurally uniform computation-dags of Fig. 1. The framework for a theory of

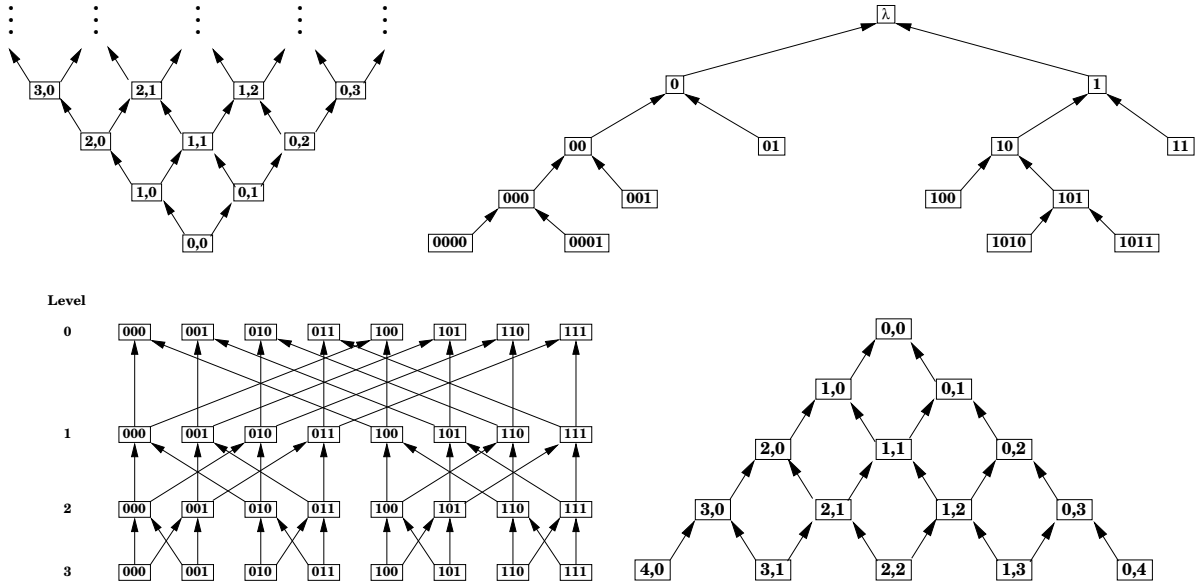

Figure 1: *Clockwise from upper left: an evolving (2-dimensional) mesh, a (binary) reduction-tree dag, the* 5*-level (*2*-dimensional) reduction-mesh, the* 4*-level FFT-dag.*

scheduling for IC is developed in [10], building on the principles that enable the optimal schedules of [12, 14]. Central to this framework are a formal method for composing simple computation-dags into complex composite ones, together with a relation that allows one to prioritize the execution order of the constituent building-block dags of a composite dag. The study in [7] develops an alternative direction, using a probabilistic pebble game to study the problem of executing tasks on unreliable clients. Our proof of the NP-Hardness of our batch-scheduling problem builds on tools developed in [7].

Although the goals and methodology of our study differ significantly from those of [4, 11, 13], we owe an intellectual debt to those pioneering studies of pebbling-based scheduling models.

Finally, the impetus for our study derives from the many exciting systems- and/or application-oriented studies of Internet-based computing, in sources such as [1, 2, 5, 6, 8, 9, 15].

2 A Model for Executing Dags on the Internet

2.1 Computation-Dags

2.1.1 Basic definitions

A *directed graph* $\mathcal{G}$ is given by a set of *nodes* $N_{\mathcal{G}}$ and a set of *arcs* (or, *directed edges*) $A_{\mathcal{G}}$, each having the form $(u \rightarrow v)$, where $u, v \in N_{\mathcal{G}}$. A *path* in $\mathcal{G}$ is a sequence of arcs that share adjacent endpoints, as in the following path from node u_1 to node u_n: $(u_1 \rightarrow u_2)$, $(u_2 \rightarrow u_3)$, $\ldots$, $(u_{n-2} \rightarrow u_{n-1})$, $(u_{n-1} \rightarrow u_n)$. A *dag* (*directed acyclic graph*) $\mathcal{G}$ is a directed graph that has no cycles; i.e., in a dag, no path of the preceding form has $u_1 = u_n$. When a dag $\mathcal{G}$ is used to model a computation, i.e., is a *computation-dag:*

- each node $v \in N_{\mathcal{G}}$ represents a task in the computation;
- an arc $(u \rightarrow v) \in A_{\mathcal{G}}$ represents the dependence of task v on task u: v cannot be executed until u is.

Given an arc $(u \rightarrow v) \in A_{\mathcal{G}}$, we call u a *parent* of v and v a *child* of u in $\mathcal{G}$. Each parentless node of $\mathcal{G}$ is called a *source (node)*, and each childless node is called a *sink (node)*; all other nodes are *internal*. A dag $\mathcal{G}$ is *bipartite* if:

1. $N_{\mathcal{G}}$ can be partitioned into subsets X and Y such that, for every arc $(u \rightarrow v) \in A_{\mathcal{G}}$, $u \in X$ and $v \in Y$;
2. each node of $\mathcal{G}$ is *incident* to some arc of $\mathcal{G}$, i.e., is either the node u or the node v of some arc $(u \rightarrow v) \in A_{\mathcal{G}}$. (It is convenient, but not necessary, to prohibit "isolated" nodes.)

The special class of bipartite dags that are *sums* of other bipartite dags plays a major role in our study. Let $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ be bipartite dags that are pairwise disjoint, in the sense that $N_{\mathcal{G}_i} \cap N_{\mathcal{G}_j} = \emptyset$ for all distinct indices i and j. The *sum* of $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$, denoted $\mathcal{G}_1 + \mathcal{G}_2 + \cdots + \mathcal{G}_n$, is the bipartite dag whose node-set and arc-set are, respectively, the unions of the corresponding sets of $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$.

We say that a dag is *connected* if, when one ignores the orientation of its arcs, then the resulting undirected graph is connected—i.e., there is an undirected path between any two distinct nodes. Every dag in Fig. 2 is connected; every bipartite dag is a sum of connected bipartite dags.

2.1.2 A sampler of building-block dags

The current study focuses on computation-dags that are built out of the following bipartite *building-block dags* by the operation of *composition* (defined in Section 2.1.3). We now present a sampler of building blocks that one can use to illustrate the theory we begin to develop in this study; see Fig. 2.

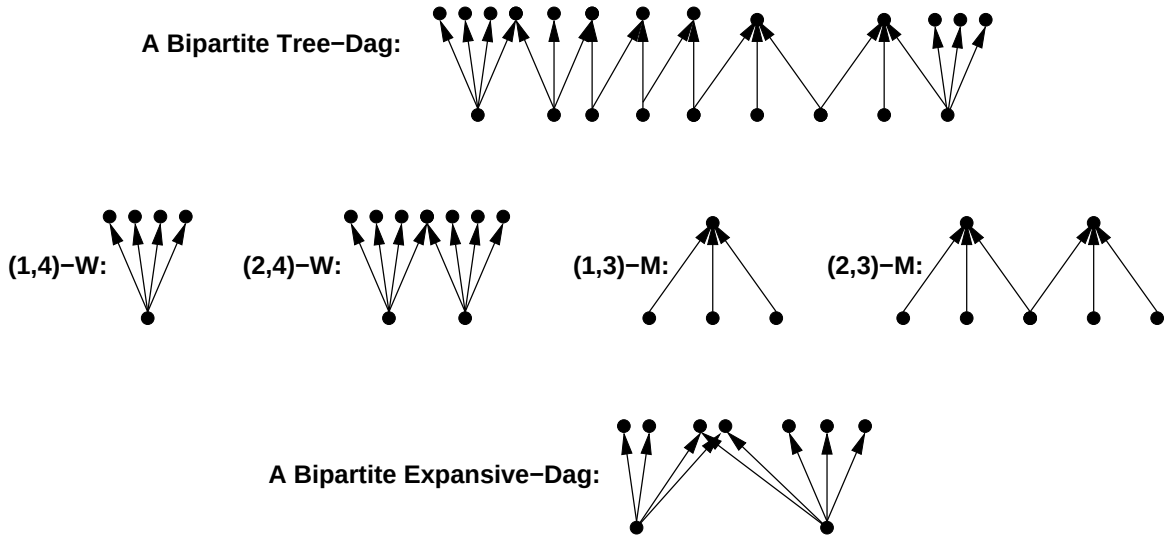

Figure 2: *Some bipartite building-block-dags.*

Bipartite tree-dags. Each such dag $\mathcal{T}$ is a bipartite dag such that, if one ignores the orientations of $\mathcal{T}$'s arcs, then the resulting graph is a tree.

The following two special classes of tree-dags generate important families of complex dags.

W-dags. For each integer $d > 1$, the $(1, d)$-*W-dag* $\mathcal{W}_{1,d}$ has one source node and d sink nodes; its d arcs connect the source to each sink. Inductively, for positive integers a, b, the $(a+b, d)$-W-dag $\mathcal{W}_{a+b,d}$ is obtained from the (a, d)-W-dag $\mathcal{W}_{a,d}$ and the (b, d)-W-dag $\mathcal{W}_{b,d}$ by identifying (or, merging) the rightmost sink of the former dag with the leftmost sink of the latter. W-dags epitomize "expansive" computations.

M-dags. For each integer $d > 1$, the $(1, d)$ *M-dag* $\mathcal{M}_{1,d}$ has d source nodes and 1 sink node; its d arcs connect each source to the sink. Inductively, for positive integers a, b, the $(a + b, d)$-M-dag $\mathcal{M}_{a+b,d}$ is obtained from the (a, d)-M-dag $\mathcal{M}_{a,d}$ and the (b, d)-M-dag $\mathcal{M}_{b,d}$ by identifying (or, merging) the rightmost source of the former dag with the leftmost source of the latter. M-dags epitomize "contractive" (or, "reductive") computations.

A large variety of significant computation-dags are compositions of W-dags and M-dags. Included are most of the computation-dags (all but the FFT dag) studied in [12, 14],

which are illustrated in Fig. 1: The evolving mesh is constructed from its source outward by "composing" a $(1,2)$-W-dag with a $(2,2)$-W-dag, then a $(3,2)$-W-dag, then a $(4,2)$-W-dag, and so on; the reduction-mesh is similarly constructed from its sources upward, using $(k,2)$-M-dags for successively decreasing values of k; the reduction-tree is constructed from its sources/leaves upward by "concatenating" independent collections of $(1,2)$-M-dags. One can, in polynomial time, craft optimal batched schedules for any such compositions.

The following additional class of building-block dags will be highlighted in Section 6, where we develop our heuristic that quickly produces "approximately optimal" batched schedules.

Bipartite expansive-dags. Each such dag $\mathcal{E}$ is a bipartite dag wherein each source v has an associated number $\varphi_v \geq 2$ such that: v has φ_v children that have no parent other than v and at most φ_v other children. Expansive-dags epitomize computations that are "expansive" but may have complex interdependencies.

One notes from Fig. 2 that bipartite expansive-dags need not be tree-dags.

2.1.3 Composing simple dags to build complex ones

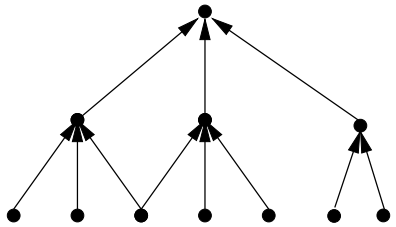

Figure 3: *A dag of type* $[[\mathcal{M}_{2,3} \Uparrow \mathcal{M}_{1,2}] \Uparrow \mathcal{M}_{1,3}]$.

The following inductive mechanism for *composing* a collection of connected bipartite dags to build complex dags is introduced in [10]; cf. Fig. 3.

- Start with a set $\mathcal{B}$ of connected bipartite dags; these will serve as a base set for the composition.
- Given dags $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{B}$—which could be copies of the same dag with nodes renamed to achieve disjointness—one obtains a composite dag $\mathcal{G}$ as follows.
  - Let the composite dag $\mathcal{G}$ begin as the sum, $\mathcal{G}_1 + \mathcal{G}_2$, of the dags $\mathcal{G}_1, \mathcal{G}_2$. Rename nodes to ensure that $N_{\mathcal{G}}$ is disjoint from $N_{\mathcal{G}_1}$ and $N_{\mathcal{G}_2}$.
  - Select some set S_1 of sinks from the copy of $\mathcal{G}_1$ in the sum $\mathcal{G}_1 + \mathcal{G}_2$, and an equal-size set S_2 of sources from the copy of $\mathcal{G}_2$ in the sum. (If $S_1 = \emptyset$, then the composition operation degenerates to the operation of forming a sum dag.)

– Pairwise identify (i.e., merge) the nodes in the sets S_1 and S_2 in some way. The resulting set of nodes is $\mathcal{G}$'s node-set; the induced set of arcs is $\mathcal{G}$'s arc-set.

- Add the dag $\mathcal{G}$ thus obtained to the base set $\mathcal{B}$.

Note the asymmetry of the composition operation: the first-named dag, $\mathcal{G}_1$, contributes some of its sinks, while the second-named dag, $\mathcal{G}_2$, contributes some of its sources. The reader should note the natural correspondence between the node-set of $\mathcal{G}$ and the node-sets of $\mathcal{G}_1$ and $\mathcal{G}_2$.

We denote the composition operation by $\Uparrow$ and refer to the resulting dag $\mathcal{G}$ as a composite dag *of type* $[\mathcal{G}_1 \Uparrow \mathcal{G}_2]$. The following lemma is of algorithmic importance, in that it allows one to ignore the order in which compositions are performed.

Lemma 2.1 ([10])**.** *The composition operation on dags is associative. That is, for all dags $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{G}_3$, a dag is composite of type $[[\mathcal{G}_1 \Uparrow \mathcal{G}_2] \Uparrow \mathcal{G}_3]$ if, and only if, it is composite of type $[\mathcal{G}_1 \Uparrow [\mathcal{G}_2 \Uparrow \mathcal{G}_3]]$.*

2.2 The Batched *Idealized* Internet-Computing Pebble Game

A number of so-called *pebble games* on dags have been shown, over the course of several decades, to yield elegant formal analogues of a variety of problems related to scheduling computation-dags. Such games use tokens called *pebbles* to model the progress of a computation on a dag: the placement or removal of the various available types of pebbles—which is constrained by the dependencies modeled by the dag's arcs—represents the changing (computational) status of the dag's task-nodes.

Our study is based on the Internet-Computing (IC, for short) Pebble Game of [12], whose structure derives from the "no recomputation allowed" pebble game of [13]. Based on studies of Internet-based computing in, for instance, [1, 8, 15], arguments are presented in [12, 14] that justify studying an idealized, simplified form of the Game. We refer the reader to the preceding sources for both the original IC Pebble Game and for the arguments justifying its simplification. We study the following idealized form of the Game here, adapted to a *batched* mode of computing.

2.2.1 The rules of the Game

The Batched IC Pebble Game on a computation-dag $\mathcal{G}$ involves one player S, the *Server*, who has access to unlimited supplies of two types of pebbles: ELIGIBLE pebbles, whose presence indicates a task's eligibility for execution, and EXECUTED pebbles, whose presence

indicates a task's having been executed. We now present the rules of the Game, which simplify those of the original IC Pebble Game of [12, 14].

The Rules of the Batch-IC Pebble Game

- S begins by placing an ELIGIBLE pebble on each unpebbled source node of $\mathcal{G}$.
  /*Unexecuted source nodes are always eligible for execution, having no parents whose prior execution they depend on.*/
- At each step t—when there is some number, say e_t, of ELIGIBLE pebbles on $\mathcal{G}$'s nodes—S is approached by some number, say r_t, of Clients, requesting tasks. In response, S:
  - selects $\min\{e_t, r_t\}$ tasks that contain ELIGIBLE pebbles,
  - replaces those pebbles by EXECUTED pebbles,
  - places ELIGIBLE pebbles on each unpebbled node of $\mathcal{G}$ all of whose parents contain EXECUTED pebbles.
- S's goal is to allocate nodes in such a way that every node v of $\mathcal{G}$ *eventually* contains an EXECUTED pebble.
  /*This modest goal is necessitated by the possibility that $\mathcal{G}$ may be infinite.*/

For brevity, we henceforth call a node ELIGIBLE (resp., EXECUTED) when it contains an ELIGIBLE (resp., an EXECUTED) pebble. For uniformity, we henceforth talk about executing nodes rather than tasks.

2.2.2 The Batch-IC Scheduling (BICSO) Problem

Our goal is to play the Batch-IC Pebble Game in a way that maximizes the number of ELIGIBLE pebbles on $\mathcal{G}$ after every move by the Server S. In other words: for each step t of a play of the Game on a dag $\mathcal{G}$ under a schedule Σ, if there are currently e_t ELIGIBLE tasks, and if r_t Clients request tasks, then we want the Server to select a set of $\min\{e_t, r_t\}$ ELIGIBLE tasks to execute that will result in the largest possible number of ELIGIBLE tasks at step $t+1$. We thus arrive at the following optimization problem.

Batched IC-Scheduling (Optimization version) (**BICSO**)

Instance: $\mathcal{I} = \langle \mathcal{G}, X, E; r \rangle$, where:
- $\mathcal{G}$ is a computation-dag;
- X and E are disjoint subsets of $N_{\mathcal{G}}$ that satisfy the following;

There is a step of some play of the Batched IC Pebble Game on $\mathcal{G}$ in which X is the set of EXECUTED nodes and E the set of ELIGIBLE nodes on $\mathcal{G}$.

- r is in the set[1] $[1, |E|]$.

Problem: Find a set $R \subseteq E$ of cardinality r that maximizes the number of ELIGIBLE nodes on $\mathcal{G}$ after executing the nodes in R, given that the nodes in X are already EXECUTED.

Note that the process of solving BICSO automatically carries with it a guarantee of optimality.

The significance of BICSO—as with the IC-Scheduling Problem of [10, 12, 14]—stems from the following intuitive scenarios. (1) Schedules that produce ELIGIBLE tasks fast may reduce the chance of the "gridlock" that could occur when remote clients are slow in returning the results of their allocated tasks—so that new tasks cannot be allocated pending the return of already assigned ones. (2) If the IC Server receives a batch of requests for tasks at (roughly) the same time, then a Batched IC-optimal schedule ensures that there are maximally many tasks that are ELIGIBLE at that time, hence maximally many requests can be satisfied. This enhances the exploitation of clients' available resources. See [12, 14] for more elaborate discussions of these scheduling criteria.

3 The General Intractability of BICSO Optimality

In this section, we show that BICSO is NP-hard even for bipartite dags. The nature of the theory of NP-Completeness demands that we focus on a decision-based reformulation of the Problem.

Batched IC-Scheduling (Decision version) (**BICSD**)

Instance: $\mathcal{I} = \langle \mathcal{G}, X, E; r, e \rangle$, where:

- $\mathcal{G}$ is a computation-dag;
- X and E are disjoint subsets of $N_{\mathcal{G}}$ that satisfy the following;

  There is a step of some play of the Batched IC Pebble Game on $\mathcal{G}$ in which X is the set of EXECUTED nodes and E the set of ELIGIBLE nodes on $\mathcal{G}$.

- r and e are integers, with $r \leq |E|$.

[1] $[a, b] = \{a, a+1, \ldots, b\}$.

Problem: Is there a subset $R \subseteq E$ of cardinality r such that, if one executes the nodes in R, given that all nodes in X are already EXECUTED, then there are at least $e + |E|$ ELIGIBLE nodes on $\mathcal{G}$?

We use the following auxiliary problem for our reduction.

Many Subsets with Small Union (**MSSU**).

Instance: Nonempty subsets $S_1, S_2, \ldots, S_n$ of $[1, n]$ whose union is $[1, n]$; integers $m \leq n$ and $b \leq n$.

Problem: Can one select m of these subsets whose union has cardinality at most b?

We invoke the following auxiliary results. The first tells us that the MSSU Problem can help us with the goal of the section.

Lemma 3.1 ([7])**.** *The Many Subsets with Small Union Problem is NP-Complete.*

The second result, which allows us to focus on a restricted class of schedules, is easily adapted from a result of [10].

Lemma 3.2 ([10])**.** *Let Σ be a schedule for a dag $\mathcal{G}$. If Σ is altered to execute all of $\mathcal{G}$'s non-sinks before any of its sinks, then the number of* ELIGIBLE *nodes produced by the resulting schedule is no smaller than Σ's.*

Theorem 3.1. *BICSO is NP-hard, even when restricted to bipartite dags.*

Proof. BICSD is in NP, by a simple argument which we leave to the reader. We concentrate, therefore, only on the Problem's NP-Hardness. To this end, we reduce MSSU to BICSD restricted to *bipartite* dags.

Let $I = \langle S_1, S_2, \ldots, S_n; m, b\rangle$ be an arbitrary instance of MSSU. We associate with I the following bipartite dag $\mathcal{G}_I$ which has n sources and n sinks. $\mathcal{G}_I$'s sources correspond to, and are named by, the n elements of $[1, n]$; its sinks correspond to, and are named by, the n subsets $S_1, \ldots, S_n$ of instance $\mathcal{I}$. $\mathcal{G}_I$ has an arc from source i to sink j just when $i \in S_j$. Note that:

- each sink is connected to at least one source because each S_i is nonempty;
- each source is connected to at least one sink because $S_1 \cup \cdots \cup S_n = [1, n]$.

We claim that I is a positive instance of MSSU—i.e., one can select m of the S_i whose union has cardinality at most b—if, and only if, there is a schedule for executing $\mathcal{G}_I$ that produces at least $m + n - b$ ELIGIBLE nodes (sources and sinks) after executing b nodes. Verifying

this claim will reduce MSSU to BICSD, thus demonstrating the NP-Completeness of the latter problem.

Assume first that I is a positive instance of MSSU. This means that in $\mathcal{G}_I$ there is a set S of b sources that collectively connect to some set T of m sinks. Say that one executes precisely the b sources of $\mathcal{G}_I$ that correspond to the set S: since sources start the Pebble Game being ELIGIBLE, one can always execute such a set. Then the resulting set of ELIGIBLE nodes of $\mathcal{G}_I$ include:

- the $n - b$ sources that have not been executed,
- the m sinks that are rendered ELIGIBLE by executing the chosen b sources (3.1)

(as well as possibly other sinks). This yields a total of at least $m + n - b$ ELIGIBLE nodes after executing the chosen b nodes.

Conversely, say that there is a set of $b \leq n$ nodes of $\mathcal{G}_I$ after whose execution there are at least $m+n-b$ ELIGIBLE nodes, for some integer $m \leq n$. The proof of Lemma 3.2 (in [10]) shows that we lose no generality by assuming that the b EXECUTED nodes are, in fact, *sources* of $\mathcal{G}_I$. (In short: replacing an EXECUTED sink by an EXECUTED source can only increase the resulting number of ELIGIBLE nodes.) The resulting snapshot of $\mathcal{G}_I$ is, thus, b EXECUTED sources, $n - b$ ELIGIBLE sources, and a total of at least $m - n - b$ ELIGIBLE nodes. Thus, there are at least m ELIGIBLE sinks. Given the way we constructed $\mathcal{G}_I$ from instance I of MSSU, the preceding snapshot of $\mathcal{G}_I$ betokens a set of (at least) m subsets chosen from the S_i whose union has cardinality b.

Since BICSD is, thus, NP-Complete, it follows that BICSO (the optimization version) is NP-Hard. □

4 Scheduling Composite Dags via Bipartite Dags

The computational intractability of solving BICSO (assuming, of course, that P $\neq$ NP) is a mandate for seeking significant special classes of dags for which one can solve BICSO in polynomial time. Our experience is that this goal is achievable for many classes of *bipartite* dags. While this is not a structural restriction of inherent interest, it turns out that we can sometimes use the operation of composition of dags to construct significant complex dags from bipartite building blocks. In this section, we show how to craft efficient solutions to BICSO for compositions of bipartite dags from efficient solutions to BICSO for bipartite dags obtained from the components.

Let us focus on a dag $\mathcal{G}$ that is a composition of the bipartite dags $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m$. This section is devoted to observations that help us solve BICSO for $\mathcal{G}$, given solutions to various subproblems for the $\mathcal{G}_i$.

We introduce a useful notion that delimits the "creation of bonuses" due to interdependencies in a bipartite dag. Given a bipartite dag $\mathcal{G}_i$, an *induced connected bipartite subdag* of the dag is an induced subdag[2] of $\mathcal{G}_i$ that is a connected bipartite dag. The utility of this notion is the following. It turns out that when searching for a solution to BICSO for a dag $\mathcal{G}$, it suffices to use solutions to a restricted version of BICSO for certain bipartite subdags of $\mathcal{G}$. In this restricted version—call it RBISCO—the bipartite subdags are connected, and all of their sources are ELIGIBLE, so the set E (of the instance of BICSO) comprises all sources of the subdag, and the set X is empty. The goal is to find an r-element subset of sources that maximizes the number of ELIGIBLE sinks. This goal is equivalent to solving BICSO for the restricted problem.

The following theorem states that if we can solve RBICSO for induced connected bipartite subdags of $\mathcal{G}$, then we can solve BICSO for $\mathcal{G}$.

Theorem 4.1. *Let the dag $\mathcal{G}$ be a composition of bipartite dags $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m$, and let $\Sigma_{\mathcal{D}_i}$, where $\mathcal{D}_i$ an induced connected bipartite subdag of $\mathcal{G}_i$, be a polynomial-time algorithm that solves RBICSO for $\mathcal{D}_i$. There is a polynomial-time algorithm Σ, that solves BICSO for $\mathcal{G}$, using the $\Sigma_{\mathcal{D}_i}$'s as subprocedures,*

Before proceeding with the proof of the theorem, we present an auxiliary algorithm that will help us develop Σ from the subprocedures $\Sigma_{\mathcal{D}_i}$. Fig. 4 presents a major building block of Σ, Procedure BBDP, which allows one to extend an efficient procedure for solving RBICSO for *individual* connected bipartite dags to an efficient procedure for solving BICSO for the composite dag $\mathcal{G}$. From a technical perspective, Procedure BBDP efficiently finds maxima on hyperplanes, as detailed in the following lemma.

Lemma 4.1. *Let us be given functions $f_i : \{m_i, \ldots, n_i\} \to \mathbb{Z}$, for $i \in [1, k]$, where each $m_i \leq n_i$. In time $O(k(n-m+1)^2)$ and space $O(k(n-m+1))$, Procedure BBDP determines, for all $r \in [m_1+\cdots+m_k,\ n_1+\cdots+n_k]$, the maximum value of $\sum_{i=1}^{k} f_i(r_i)$ over all sequences $r_1, \ldots, r_k$ such that $r_1 + \cdots + r_k = r$, and each $r_i \in [m_i, n_i]$.*

It is a clerical matter to add extra bookkeeping to Procedure BBDP, that finds the values of r_i's that achieve the maximum.

Proof of Theorem 4.1. Let us be given an instance $\mathcal{I} = \langle \mathcal{G}, X, E; r \rangle$ of BICSO, where $\mathcal{G}$ is as described in the theorem. Since we never jeopardize the number of ELIGIBLE nodes produced by a schedule by executing the sinks of $\mathcal{G}$ only after having executed all non-sinks (Lemma 3.2), let us focus on the modified goal of finding R among $\mathcal{G}$'s non-sinks; that is, we assume that E contains non-sinks only.

As the first step toward our goal, for each $i \in [1, m]$:

[2] A subdag $\mathcal{H}$ of $\mathcal{G}$ is *induced* if (*a*) $N_{\mathcal{H}} \subseteq N_{\mathcal{G}}$; (*b*) $A_{\mathcal{H}}$ comprises *all* arcs of $\mathcal{G}$ that have both endpoints in $N_{\mathcal{H}}$.

Procedure BBDP$(k, f_1, \ldots, f_k, m_1, \ldots, m_k, n_1, \ldots, n_k)$

```
01. local variables: integer array B[1, ..., k][∑_{j=1}^{k} m_j, ..., ∑_{j=1}^{k} n_j];
                     integers i, r, r_a, r_b, max
02. for r := m_1 to n_1
03.       B[1][r] := f_1(r)
04. for i := 2 to k
05.       for r := m_1 + ··· + m_i to n_1 + ··· + n_i
06.             max := −∞
07.             for r_b := m_i to n_i
08.                   r_a := r − r_b
09.                   if m_1 + ··· + m_{i−1} ≤ r_a ≤ n_1 + ··· + n_{i−1}
10.                         then if B[i − 1][r_a] + f_i(r_b) > max
11.                               max := B[i − 1][r_a] + f_i(r_b)
12.             B[i][r] := max
13. return vector B[k]
```

Figure 4: *A building-block dynamic program used by Algorithm* Σ_{DP}.

- let X_i (resp., E_i) be the sources of $\mathcal{G}_i$ that correspond (in the natural manner emerging from the definition of composition) to the set X (resp., the set E);
- restate the goal of finding the set R as the goal of finding m sets R_i, where each $R_i \subseteq E_i$ and $|R_1| + |R_2| + \cdots + |R_m| = r$.

Since after composing the $\mathcal{G}_i$ to form $\mathcal{G}$, all sources of the $\mathcal{G}_i$ become non-sinks of $\mathcal{G}$, our simplification honors Lemma 3.2. Moreover, by restating a lemma of [10], we can calculate the number of ELIGIBLE nodes on $\mathcal{G}$ by looking at the numbers of ELIGIBLE sinks of the $\mathcal{G}_i$, when the sources X_i of $\mathcal{G}_i$ are its only EXECUTED nodes.

Lemma 4.2 ([10])**.** *$\mathcal{G}$ has $|S| - |X| + \sum_{i=1}^{m} e_i(X_i)$ ELIGIBLE nodes, where:*

- *S is the set of sources of $\mathcal{G}$;*
- *$e_i(X_i)$ is the number of sinks of $\mathcal{G}_i$ that are ELIGIBLE when the only EXECUTED nodes of $\mathcal{G}_i$ are the sources X_i.*

Lemma 4.2 suggests a simplification of BICSO's goal of finding the set R. Since the sets E_i consist entirely of sources, the sets $R_i \subseteq E_i$ consist of non-sinks. Since the nodes of each R_i correspond to nodes of R, the latter set also consists of non-sinks; hence, we can so restrict our search for R.

A second simplification emerges by noting that, when any E_i is empty, the corresponding R_i is also empty, so that $e_i(X_i \cup R_i) = e_i(X_i)$. In this case, $\mathcal{G}_i$ can be ignored in our quest for R; hence, we can henceforth restrict attention to $\mathcal{G}_i$ for which $E_i \neq \emptyset$.

A subtler simplification emerges from the demonstration that, when calculating $e_i(X_i \cup R_i)$, we may restrict attention to a certain induced subdag of $\mathcal{G}_i$, which is isolated as follows. Let S_i be the sources and T_i the sinks of $\mathcal{G}_i$. The set T_i can be partitioned into:

1. the set $T_i^{(X)}$ of sinks all of whose parents are EXECUTED; these sinks are ELIGIBLE;
2. the set $T_i^{(E,X)}$ of sinks all of whose parents are either ELIGIBLE or EXECUTED and at least one of whose parents is ELIGIBLE; these sinks are not ELIGIBLE (but they may become so when we execute the nodes of the R that we choose);
3. the set $T_i \setminus (T_i^{(X)} \cup T_i^{(E,X)})$ of sinks that have at least one parent that is neither EXECUTED nor ELIGIBLE; these sinks are not ELIGIBLE (and will not become so when the nodes of R are executed).

The subdag of each $\mathcal{G}_i$—call it $\mathcal{S}_i$—that is induced by the sources E_i and the sinks $T_i^{(E,X)}$ is of special interest to us. We treat $\mathcal{S}_i$ as though its node-set is disjoint from $N_{\mathcal{G}_i}$ (which can be achieved via renaming), but we retain the natural correspondence between the nodes of $\mathcal{G}_i$ and $\mathcal{S}_i$. Note that each $u \in N_{\mathcal{S}_i}$ that corresponds to a node of $T_i^{(E,X)}$ is a child of a $v \in N_{\mathcal{S}_i}$ that corresponds to a node of E_i; however, the converse may be false: there may be childless nodes. Thus, each $\mathcal{S}_i$ is a sum of (possibly zero) isolated nodes that correspond to sources of $\mathcal{G}_i$ and (possibly zero) connected bipartite dags $\mathcal{S}_{i,1}, \mathcal{S}_{i,2}, \ldots, \mathcal{S}_{i,h_i}$. Let J_i be the set of these isolated nodes, and let s_i be the aggregate number of sources in these connected bipartite dags.

Each $\mathcal{S}_i$ helps simplify our calculation of $e_i(X_i \cup R_i)$. If we execute a set $R_i \subseteq E_i$ of nodes of $\mathcal{G}_i$, then some sinks of $\mathcal{G}_i$ may become ELIGIBLE. The parents of any such sink v must come either from X_i or from R_i; at least one parent must come from R_i. Thus, $v \in T_i^{(E,X)}$, hence corresponds to a sink of $\mathcal{S}_i$; moreover, v became ELIGIBLE when sources of $\mathcal{S}_i$ that correspond to R_i were executed. Importantly, now the converse is also true! If we execute any subset R_i of sources of $\mathcal{S}_i$, and a sink v becomes ELIGIBLE, then the corresponding sink in $\mathcal{G}_i$ also becomes ELIGIBLE when sources that correspond to R_i are executed (in addition, of course, to sources of X_i that are already EXECUTED.) Therefore, $e_i(X_i \cup R_i) = e_i(X_i) + \sum_{j=1}^{h_i} e_{i,j}(R_{i,j})$, where:

- $R_{i,j}$ comprises the sources of $\mathcal{S}_{i,j}$ that correspond to nodes from R_i;
- $e_{i,j}(R_{i,j})$ is the number of sinks of $\mathcal{S}_{i,j}$ that are ELIGIBLE when only the sources of $\mathcal{S}_{i,j}$ from the set $R_{i,j}$ are EXECUTED.

The preceding discussion implies that, in order to find a set R that maximizes $e(X \cup R)$, it is sufficient to consider two cases.

Case 1. If $r \geq s_1 + s_2 + \cdots + s_m$, the we maximize the number of ELIGIBLE nodes by executing:

- the nodes of $\mathcal{G}$ that correspond to all sources of the $\mathcal{S}_{i,j}$, and
- $r - (s_1 + s_2 + \cdots + s_m)$ nodes that correspond to arbitrary nodes from the "isolated-node" sets J_1 through J_m.

Case 2. If $r < s_1 + s_2 + \cdots + s_m$, then we maximize the number of ELIGIBLE nodes by executing the r nodes of $\mathcal{G}$ that correspond to the r sources of the $\mathcal{S}_{i,j}$ that maximize the number of ELIGIBLE sinks on the $\mathcal{S}_{i,j}$.

In the former case, we can trivially solve BICSO. In the latter case, since each $\mathcal{S}_{i,j}$ is an induced connected bipartite subdag of $\mathcal{G}_i$, we can use Procedure $\Sigma_{\mathcal{S}_{i,j}}$ to determine the sought maximum, for all possible numbers of EXECUTED sources of the $\mathcal{S}_{i,j}$. Then, we use Procedure BBDP to combine these individual solutions to obtain a solution to BICSO. □

5 Tractable BICSO Optimality for Composite Trees

This section develops a polynomial-time algorithm that solves BICSO for the family $\mathbf{T}$ of all computation-dags that are obtained from *bipartite tree-dags* via composition. Stated formally:

Theorem 5.1. *There is a polynomial-time algorithm Σ_{tree} that solves BICSO for any composite tree-dag $\mathcal{T} \in \mathbf{T}$.*

Proof. It suffices to develop a dynamic programming algorithm Σ_{DP} that solves RBICSO for any bipartite tree-dag. An invocation of Theorem 4.1 allows us to extend Σ_{DP} to Σ_{tree}, because when the $\mathcal{G}_i$'s of that theorem are bipartite tree-dags, then so also are the $\mathcal{S}_{i,j}$'s. Our goal of developing Σ_{DP} is stated in the subsequent lemma.

Lemma 5.1. *Algorithm Σ_{DP} will, in polynomial time, solve any instance of RBICSO in which the dag $\mathcal{T}$ of the instance is a bipartite tree-dag.*

Proof. Any bipartite tree-dag $\mathcal{T}$ arises from "folding" a tree T and orienting its edges. (Note that T is: (a) undirected: its edges lack orientations; (b) unrooted: no node of T is designated as root.) For convenience of reference, we label the nodes of T as "sources" and "sinks" according to their roles in the dag $\mathcal{T}$. Clearly each sink of T is adjacent to

only sources, and each source to only sinks. The key idea of Algorithm Σ_{DP} is that we can find the maximum number of ELIGIBLE sinks for a tree of a given height by appropriately evaluating maxima for shorter trees.

Our first step in developing Σ_{DP} is to recursively decompose T into subtrees. We choose some source w of T and let it act as a root for T; call the resulting rooted tree T_w. (We prefer rooted trees in this development is that they have *heights*.) We next perform a breadth-first traversal of T_w starting from its root w. Each time we descend from a sink v of T_w to a source u during this traversal, we naturally produce a subtree, call it T_u, of T_w. (T_u is a copy of the subtree of T_w rooted at u.) Although the node-sets of T_w and T_u are disjoint, there is a natural correspondence between them, and we refer to corresponding nodes using the same name. In the manner described, our traversal of T_w produces a sequence of subtrees (beginning with T_w), such that each subtree includes a certain number of shorter ones, which occur later in the sequence. Σ_{DP} processes the subtrees in the *reverse* order of this sequence, which ensures that certain values that need to be computed for a subtree, which depend recursively on values for shorter subtrees, can be computed, because the values for shorter subtrees are already available.

Algorithm Σ_{DP} chooses the nodes to execute by recursively calculating the following functions. Pick any subtree T_u; say that it has s sources.

- For any $r \in [1, s]$, let $E_1(T_u, r)$ be the maximum number of ELIGIBLE sinks on T_u when the root u and some other $r - 1$ of its sources are EXECUTED.

$E_1(T_u, r)$ is easy to calculate when T_u has height 0 or 1, for then T_u has only one source, so that $E_1(T_u, r)$ is just the number of sinks that are connected only to u.

- For any $r \in [0, s - 1]$, let $E_0(T_u, r)$ be the maximum number of ELIGIBLE sinks on T_u when the root u is *not* EXECUTED but some r other of its sources are EXECUTED.

$E_0(T_u, r) = 0$ when T_u has height 0 or 1. For any $r \in [0, s]$, the maximum number of ELIGIBLE sinks in T_u when r of its sources are EXECUTED can easily be calculated using E_0 and E_1. In particular, Σ_{DP} achieves its goal by computing $E_0(T_w, r)$ and $E_1(T_w, r)$ for any r in the range $0 \leq r \leq$ (the number of sources in T), as follows. As just noted, the challenge is to calculate these values for subtrees of heights ≥ 2. We accomplish this by decomposing trees in the manner depicted in Fig. 5. Focus on a subtree T_u of height ≥ 2; say that T_u has s sources. Consider all sinks of T_u that are linked to the source u. Now, some k of these sinks—call them $v_1, \ldots, v_k$—are also linked to some source other than u, while some h of these sinks are not. Since T_u has height ≥ 2, we know that $k \geq 1$; it is possible that $h = 0$. For any $i \in [1, k]$, sink v_i is connected to some $g_i \geq 1$ sources other than u—call them $u_{i,1}, \ldots, u_{i,g_i}$. Consider the subtrees $T_{u_{i,j}}$, for $i \in [1, k]$ and $j \in [1, g_i]$;

each has height strictly smaller than T_u's. Let $s_{i,j}$ be the number of sources in $T_{u_{i,j}}$, so that $s = 1 + \sum_{i=1}^{k} \sum_{j=1}^{g_i} s_{i,j}$. We can calculate the functions E_0 and E_1 for T_u recursively from the corresponding functions for the $T_{u_{i,j}}$.

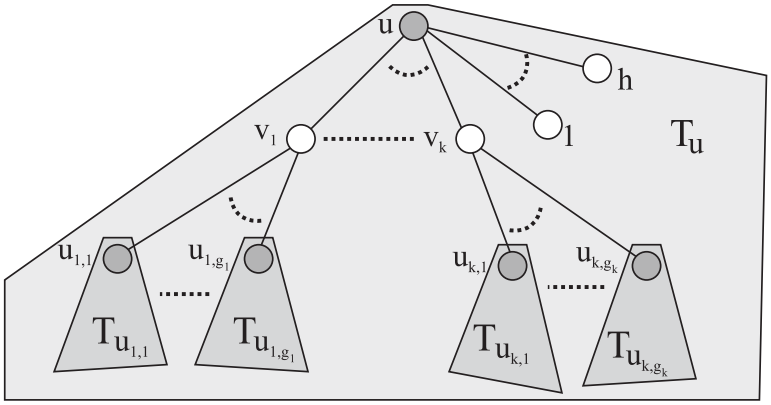

Figure 5: *Decomposing T_u: shaded nodes are sources; blank nodes are sinks.*

We begin with E_0, since it is easier to calculate. Pick any $r \in [0, s-1]$. Since (by assumption) u is not EXECUTED, none of the $k+h$ sinks connected to u is ELIGIBLE. Therefore, $E_0(T_u, r)$ is just the maximum number of ELIGIBLE sinks in the subtrees $T_{u_{i,j}}$, when a total of r sources are EXECUTED in these trees. Since these trees are shorter than T_u, Σ_{DP} will have computed the values of E_0 and E_1 for each of them; hence, using Procedure BBDP, it can calculate $E_0(T_u, r)$.

We decompose the goal of calculating E_1 into subgoals. Pick any $r \in [1, s]$, and pick the r sources that include u and that maximize the number of ELIGIBLE sinks in T_u. Let us count these sinks. Since one of the r EXECUTED sources is u, all h sinks that are connected only to u are ELIGIBLE. The remaining $r-1$ EXECUTED sources belong to the subtrees $T_{u_{i,j}}$ connected to sinks $v_1, \ldots, v_k$. Let r_i be the aggregate number of EXECUTED sources within subtrees $T_{u_{i,1}}, \ldots, T_{u_{i,g_i}}$, so that $r-1 = r_1 + \cdots + r_k$. These EXECUTED sources make sink v_i ELIGIBLE if, and only if, the sources $u_{i,1}, \ldots, u_{i,g_i}$ are EXECUTED. It follows that the number of ELIGIBLE sinks among sink v_i and the sinks in $T_{u_{i,1}}, \ldots, T_{u_{i,g_i}}$ is precisely the number of ELIGIBLE sinks in the subtrees, *plus*:

- 1, if all of $u_{i,1}, \ldots, u_{i,g_i}$ are EXECUTED;
- 0, if at least one of $u_{i,1}, \ldots, u_{i,g_i}$ is not EXECUTED.

We thus have k subgoals, the ith being to maximize the number of ELIGIBLE sinks among sink v_i and the sinks in $T_{u_{i,1}}, \ldots, T_{u_{i,g_i}}$, under the constraint that r_i sources from these subtrees are EXECUTED. If we can calculate this maximum for any $r_i \in [0,\ s_{i,1} + \cdots + s_{i,g_i}]$, then Σ_{DP} can use Procedure BBDP here also, to find the value of E_1.

The desired subgoal can be achieved as the larger of two "sub"-maxima. (1) When all sources $u_{i,1}, \ldots, u_{i,g_i}$ are EXECUTED, we use Procedure BBDP, as applied to the functions E_1 for the subtrees $T_{u_{i,1}}, \ldots, T_{u_{i,g_i}}$, to compute the maximum number of ELIGIBLE sinks for all $r_i \in [g_i,\ s_{i,1} + \cdots + s_{i,g_i}]$. Since then all sources connected to v_i are EXECUTED, each maximum gets increased by 1. It follows that the bonus of 1 is properly accounted for. (2) When at least one of the sources $u_{i,1}, \ldots, u_{i,g_i}$ is not EXECUTED, we know that sink v_i cannot be ELIGIBLE. Focusing on the case $r_i \leq s_{i,1} + \cdots + s_{i,g_i} - 1$, Σ_{DP} can now use Procedure BBDP to consider every $j \in [1, g_i]$: (i) using the function E_0 that was calculated for subtree $T_{u_{i,j}}$; (ii) for the other subtrees, using the function of the maximum number of ELIGIBLE sinks as a function of the number of EXECUTED sources in this subtree.

This completes the description of Algorithm Σ_{DP}. □

We now apply Lemma 5.1 in Theorem 4.1, which completes the proof of Theorem 5.1. □

6 Solving BICSO Efficiently for Expansive Dags

Algorithm Σ_{tree} is computationally rather inefficient, despite being polynomial-time, since its timing polynomial has high degree. We therefore have studied the problem of solving BICSO for nontrivial classes of computation dags *approximately* optimally as long as they solve the problem much faster than Σ_{tree}. The initial result of our quest is the following algorithm that approximates a solution to BICSO for the family $\mathbf{E}$ of composite expansive dags.

Algorithm Σ_{exp} implements the following natural, fast heuristic for scheduling an expansive dag $\mathcal{E}$. For each source v of $\mathcal{E}$, say that there are φ_v nodes that have v as their sole parent and ψ_v nodes that have other parents in addition to v. (By definition, $\varphi_v \geq \psi_v$). Say that $\mathcal{E}$ has $|E|$ ELIGIBLE nodes and that we are requested to execute the best r of these. Σ_{exp} selects the r nodes that have the largest associated integers φ_v. Of course, this greedy heuristic may deviate from optimality because it ignores the "bonuses" that may arise when one executes ELIGIBLE nodes that are siblings in $\mathcal{E}$.

We claim that Algorithm Σ_{exp} solves BICSO to within a factor of 4 of optimally for the family $\mathbf{E}$. Stated formally:

Theorem 6.1. *For any instance $\mathcal{I} = \langle \mathcal{E}, X, E; r\rangle$ of BICSO, where $\mathcal{E} \in \mathbf{E}$, Algorithm Σ_{exp} will, in time $O(|E|)$, find a solution to the Problem, whose increase in the number of* ELIGIBLE *nodes is at least one-fourth the optimal increase.*

Proof. Focus on instance $\mathcal{I} = \langle \mathcal{E}, X, E; r\rangle$ of BICSO, where $\mathcal{E} \in \mathbf{E}$. We consider in turn the speed of Algorithm Σ_{exp} and the quality of the schedule it produces.

Timing. It is easy to implement Σ_{exp} so that it operates in time $O(|E|)$. One way to achieve this is the following, referring to algorithms found in, say, [3].

1. Use the linear-time selection algorithm to find the node $v^\star \in E$ whose associated integer $\varphi_{v^\star}$ is the rth largest.
2. Use the PARTITION procedure of QUICKSORT with $v^\star$ as pivot to identify r nodes of E that have $\varphi_v \geq \varphi_{v^\star}$.

Quality. Our analysis of Algorithm Σ_{exp} considers two cases.

(1) Say first that r is no smaller than the number of non-sinks in E. In this case, Σ_{exp} actually provides a solution to instance $\mathcal{I}$ of BICSO. This is because sinks are the only nodes whose associated integers φ_v are zero, so Σ_{exp} will select them last, hence will maximize its production of ELIGIBLE nodes.

(2) Say next that r is smaller than the number of non-sinks in E, so that Σ_{exp} selects only non-sinks of $\mathcal{E}$. For any set $R \subseteq E$ that is composed of non-sinks, we can calculate the *impact* of set R, denoted $\iota(R)$, that is the difference between

N_{after}: the number of nodes that are ELIGIBLE when all nodes in $X \cup R$ are EXECUTED;
N_{before}: the number of nodes that are ELIGIBLE when the nodes in X are EXECUTED.

By our prior observations,

$$\iota(R) \stackrel{\text{def}}{=} N_{\text{after}} - N_{\text{before}} = -|R| + \sum_{i=1}^{m} \sum_{j=1}^{h_i} e_{i,j}(R_{i,j}).$$

We note that the $\mathcal{S}_{i,j}$ of Section 4 are expansive dags and that their associated "isolated" sets J_i are empty. Let $R_{\text{opt}} \subseteq E$ be a set of nodes that leads to a solution to BICSO for instance $\mathcal{I}$, and let R_Σ be the set selected by Σ_{exp}. Observe that, upon execution, each source v of an $\mathcal{S}_{i,j}$ can render no more than $\varphi_v + \psi_v \leq 2\varphi_v$ new sinks ELIGIBLE. Thus, $\iota(R_{\text{opt}}) \leq 2 \sum_{v \in R_{\text{opt}}} \varphi_v$, while

$$\iota(R_\Sigma) \geq -|R_\Sigma| + \sum_{v \in R_\Sigma} \varphi_v \geq 1/2 \sum_{v \in R_\Sigma} \varphi_v,$$

the last inequality following since $\varphi_v \geq 2$. Since Σ_{exp} selects nodes for execution that have the largest associated integers φ_v, $\iota(R_{\text{opt}}) \leq 4 \cdot \iota(R_\Sigma)$, as claimed in the statement of the theorem. □

Acknowledgment. A portion of the research of G. Malewicz was done while visiting the Univ. of Massachusetts Amherst. The research of A. Rosenberg was supported in part by NSF Grant CCF-0342417. The authors thank M. Yurkewych for valuable discussions.

References

[1] R. Buyya, D. Abramson, J. Giddy (2001): A case for economy Grid architecture for service oriented Grid computing. *10th Heterogeneous Computing Wkshp.*

[2] W. Cirne and K. Marzullo (1999): The Computational Co-Op: gathering clusters into a metacomputer. *13th Intl. Parallel Processing Symp.*, 160–166.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein (1999): *Introduction to Algorithms* (2nd Edition). MIT Press, Cambridge, Mass.

[4] S.A. Cook (1974): An observation on time-storage tradeoff. *J. Comp. Syst. Scis. 9*, 308–316.

[5] I. Foster and C. Kesselman [eds.] (2004): *The Grid: Blueprint for a New Computing Infrastructure* (2nd edition), Morgan-Kaufmann, San Francisco.

[6] I. Foster, C. Kesselman, S. Tuecke (2001): The anatomy of the Grid: enabling scalable virtual organizations. *Intl. J. Supercomputer Applications.*

[7] L. Gao and G. Malewicz (2004): Internet computing of tasks with dependencies using unreliable workers. *8th Intl. Conf. on Principles of Distributed Systems*, to appear.

[8] D. Kondo, H. Casanova, E. Wing, F. Berman (2002): Models and scheduling mechanisms for global computing applications. *Intl. Parallel and Distr. Processing Symp.*

[9] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky (2000): SETI@home: massively distributed computing for SETI. In *Computing in Science and Engineering* (P.F. Dubois, Ed.) IEEE Computer Soc. Press, Los Alamitos, CA.

[10] G. Malewicz, A. L. Rosenberg, M. Yurkewych (2005): Toward a scheduling theory for Internet-based computing. *IEEE Trans. Comput. 55*, 757–768.

[11] M.S. Paterson, C.E. Hewitt (1970): Comparative schematology. *Project MAC Conf. on Concurrent Systems and Parallel Computation*, ACM Press, 119–127.

[12] A.L. Rosenberg (2004): On scheduling mesh-structured computations for Internet-based computing. *IEEE Trans. Comput. 53*, 1176–1186.

[13] A.L. Rosenberg and I.H. Sudborough (1983): Bandwidth and pebbling. *Computing 31*, 115–139.

[14] A.L. Rosenberg and M. Yurkewych (2005): Guidelines for scheduling some common computation-dags for Internet-based computing. *IEEE Trans. Comput. 54*, 428–438.

[15] X.-H. Sun and M. Wu (2003): GHS: A performance prediction and task scheduling system for Grid computing. *IEEE Intl. Parallel and Distributed Processing Symp.*