

Data Races and the Discrete Resource-time Tradeoff Problem with Resource Reuse over Paths

Rathish Das
Stony Brook University
radas@cs.stonybrook.edu

Jayson Lynch
MIT
jaysonl@mit.edu

Joseph S. B. Mitchell
Stony Brook University
joseph.mitchell@stonybrook.edu

Shih-Yu Tsai
Stony Brook University
shitsai@cs.stonybrook.edu

Esther M. Arkin
Stony Brook University
esther.arkin@stonybrook.edu

Steven Skiena
Stony Brook University
skiena@cs.stonybrook.edu

Sharmila Duppala
Stony Brook University
sduppala@cs.stonybrook.edu

Rezaul Chowdhury
Stony Brook University
rezaul@cs.stonybrook.edu

ABSTRACT

A determinacy race occurs if two or more logically parallel instructions access the same memory location and at least one of them tries to modify its content. Races are often undesirable as they can lead to nondeterministic and incorrect program behavior. A data race is a special case of a determinacy race which can be eliminated by associating a mutual-exclusion lock with the memory location in question or allowing atomic accesses to it. However, such solutions can reduce parallelism by serializing all accesses to that location. For associative and commutative updates to a memory cell, one can instead use a reducer, which allows parallel race-free updates at the expense of using some extra space. More extra space usually leads to more parallel updates, which in turn contributes to potentially lowering the overall execution time of the program.

We start by asking the following question. Given a fixed budget of extra space for mitigating the cost of races in a parallel program, which memory locations should be assigned reducers and how should the space be distributed among those reducers in order to minimize the overall running time? We argue that under reasonable conditions the races of a program can be captured by a directed acyclic graph (DAG), with nodes representing memory cells and arcs representing read-write dependencies between cells. We then formulate our original question as an optimization problem on this DAG. We concentrate on a variation of this problem where space reuse among reducers is allowed by routing every unit of extra space along a (possibly different) source to sink path of the DAG and using it in the construction of multiple (possibly zero) reducers along the path. We consider two different ways of constructing a reducer and the corresponding duration functions (i.e., reduction time as a function of space budget).

We generalize our race-avoiding space-time tradeoff problem to a discrete resource-time tradeoff problem with general non-increasing duration functions and resource reuse over paths of the given DAG.

For general DAGs, we show that even if the entire DAG is available to us offline the problem is strongly NP-hard under all three duration functions, and we give approximation algorithms for solving the corresponding optimization problems. We also prove hardness of approximation for the general resource-time tradeoff problem and give a pseudo-polynomial time algorithm for series-parallel DAGs.

CCS CONCEPTS

• Parallel and Distributed Algorithms; • Multi-Core Architectures; • Resource Management and Awareness; • Scheduling Problems;

KEYWORDS

Parallel and Distributed Algorithms, Multi-Core Architectures, Resource Management and Awareness, Scheduling Problems

ACM Reference Format:

Rathish Das, Shih-Yu Tsai, Sharmila Duppala, Jayson Lynch, Esther M. Arkin, Rezaul Chowdhury, Joseph S. B. Mitchell, and Steven Skiena. 2019. Data Races and the Discrete Resource-time Tradeoff Problem with Resource Reuse over Paths. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19)*, June 22–24, 2019, Phoenix, AZ, USA. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3323165.3323209>

PAGE DISTRIBUTION

Page Type	Page Numbers	#Pages
Title	1	1
Main Text	2, 4, 5, 7 – 10, 12, 15, 16	10
Figures	3, 6, 11, 13, 14, 17	6
Bibliography	18	1
Appendices	19 – 20	2

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6184-2/19/06...\$15.00

<https://doi.org/10.1145/3323165.3323209>

1 INTRODUCTION

A determinacy race (or a general race) [12, 24] occurs if two or more logically parallel instructions access the same memory location and at least one of them modifies its content. Races are often undesirable as they can lead to nondeterministic and incorrect program behavior. A data race is a special case of a determinacy race which can be eliminated by associating a mutual-exclusion lock with the memory location in question or allowing only atomic accesses to it. Such a solution, however, makes all accesses to that location serial and thus destroys all parallelism. Figure 1 shows an example.

One can use a reducer [7, 13, 27] to eliminate data races on a shared variable without destroying parallelism, provided the update operation is associative and commutative. Figure 2 shows the construction of a simple recursive binary reducer. For any integer $h > 0$ such a reducer is a full binary tree of height h and size $2^{h+1} - 1$ with the shared variable at the root. Each nonroot node is associated with a unit of extra space initialized to zero. All updates to the shared variable are equally distributed among the leaves of the tree. Each node has a lock and a waiting queue to avoid races by serializing the updates it receives, but updates to different nodes can be applied in parallel. As soon as a node undergoes its last update, it updates its parent using its final value. In fact, such a reducer can be constructed using only 2^h units of extra space because if a node completes before its sibling it can become its own parent (with ties broken arbitrarily) and the sibling then updates the new parent. Assume that the time needed to apply an update significantly dominates the execution time of every other operation the reducer performs and each update takes one unit of time to apply. Then a reducer of height h can correctly apply n parallel updates on a shared variable in $\lceil \frac{n}{2^h} \rceil + h + 1$ time provided at least 2^h processors are available. Hence, for large n , the speedup achieved by a reducer (w.r.t. serially and directly updating the shared variable) is almost linear in the amount of extra space used.

To see how extra space can speed up real parallel programs consider the iterative matrix multiplication code PARALLEL-MM shown in Figure 3 which multiplies two $n \times n$ matrices $X[1..n][1..n]$ and $Y[1..n][1..n]$ and puts the results in another $n \times n$ matrix $Z[1..n][1..n]$; that is, it sets $Z[i][j] = \sum_{1 \leq k \leq n} X[i][k] \times Y[k][j]$ for $1 \leq i, j \leq n$. Since every $Z[i][j]$ value can be computed independently of others, all iterations of the loops in Lines 1 and 2 can be executed in parallel without compromising correctness of the computation. However, the same is not true for the loop in Line 4 because if parallelized, for fixed values of i and j , all iterations of that loop will update the same memory location $Z[i][j]$ giving rise to data races and thus producing potentially incorrect results. Use of a mutual-exclusion lock or atomic updates for each $Z[i][j]$ will ensure correctness but in that case even with an unbounded number of processors, the code will take $\Theta(n)$ time to multiply the two $n \times n$ matrices. Now if we put a reducer of height h (integer $h \in [1, \log_2 n]$) at the top of each $Z[i][j]$ the time to fully update each $Z[i][j]$ and thus the overall running time of the code will drop to $\Theta\left(\frac{n}{2^h} + h\right)$ at the cost of using $n^2 \times 2^h$ units of extra space. Observe that when $h = 1$, the running time of the code almost halves using $2n^2$ units of extra space, and when $h = \lfloor \log_2 n \rfloor$, the running time drops to $\Theta(\log n)$ using $\Theta(n^3)$ extra space.

In order to analyze a program P with data races, we capture those races in a directed acyclic graph (DAG) $D(P)$, assuming that there are no cyclic read-write dependencies among the memory locations accessed by P . Figure 4 shows an example. We restrict P to the set of programs that perform $\mathcal{O}(1)$ other operations between two successive writes to the memory, e.g., PARALLEL-MM in Figure 3. We assume that an update operation is significantly more expensive than any other single operation performed by P and hence the costs of those operations can be safely ignored. Each node x of $D(P)$ represents a memory location, and a directed edge from node x to node y means that y is updated using the value stored at x . The in-degree $d_x^{(in)}$ of node x gives the number of times x is updated. With x we also associate a *work* value w_x and set $w_x = d_x^{(in)}$. Assuming that each update operation requires unit time to execute and each node has a lock and a wait queue to serialize the updates, the w_x value represents the time spent updating x (excluding all idle times). The w_x value also represents an upper bound on the time elapsed between the trigger time of any incoming edge of x and the time the edge completes updating x . We assume that updates along all outgoing edges of x trigger as soon as all incoming edges complete updating x . One can then make the following observation.

OBSERVATION 1.1. *The running time of P with an unbounded number of processors is upper bounded by the makespan of $D(P)$ ¹.*

Then one natural question to ask is the following.

QUESTION 1.1. *Given a fixed budget of units of extra space to mitigate the cost of data races in P , which memory locations should be assigned reducers and how should the space be distributed among those reducers in order to minimize the makespan of $D(P)$?*

Figure 5 shows how to minimize the makespan of the DAG in Figure 4 using two units of extra space.

The question above ignores the possibility that space can be reused among reducers in $D(P)$. Indeed, after node x reaches its final value (i.e., updated $w_x = d_x^{(in)}$ times) it can release all (if any) space it used for its reducer which can then be reused by some other node y . A global memory manager can be used by the nodes to allocate/deallocate space for reducers. The following modified version of Question 1.1 now allows space reuse.

QUESTION 1.2. *Repeat Question 1.1 but allow for space reuse among nodes of $D(P)$ by putting all extra space under the control of a global memory manager that each node calls to allocate space for its reducer right before its first update and to deallocate that space right after its last update.*

The problem with a single global memory manager is that it can easily become a performance bottleneck for highly parallel programs. Though better memory allocators have been developed for multi-core or multi-threaded systems [1–3, 5, 30], we can instead use an approach often used by recursive fork-join programs which avoids repeated calls to an external memory manager altogether along with the overhead of repeated memory allocations/deallocations. A single large segment of memory is allocated before the initial

¹To see why this is true start from the sink node and move backward toward the source by always moving to that predecessor y of the current node x that performed the last update on x and noting that after edge (y, x) was triggered it did not have to wait for more than $d_x^{(in)}$ time units to complete applying y 's update to x .

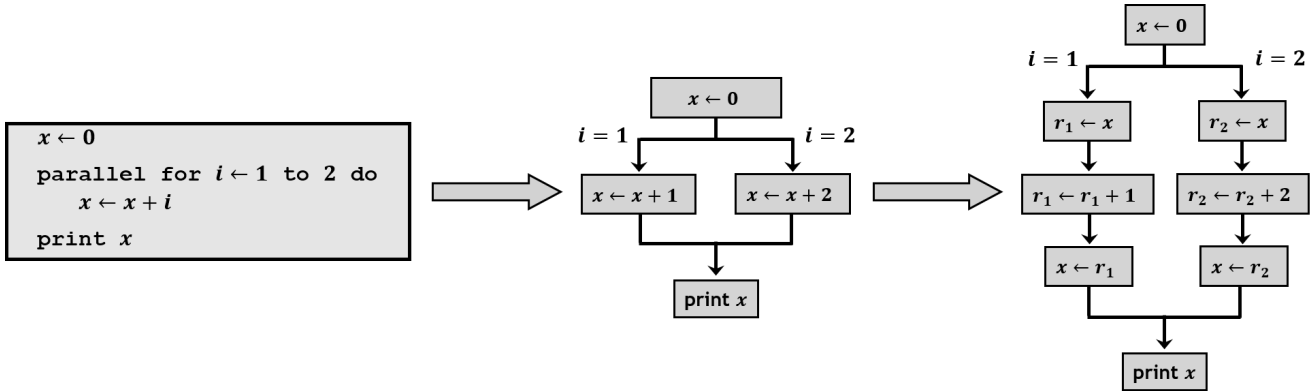


Figure 1: This figure shows a race on global variable x caused by two parallel threads trying to increment x , where r_1 and r_2 are local registers. The value printed by the 'print' statement depends on how the two threads are scheduled. Unless the two threads are executed sequentially, the print statement will print an incorrect result (either 1 or 2 depending on which thread updated x last).

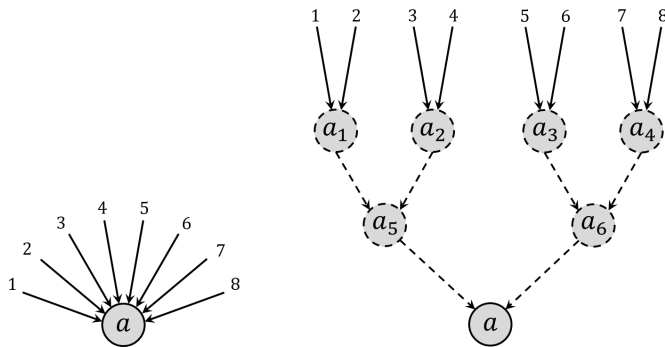


Figure 2: [LEFT] A memory location a with eight updates using an associative and commutative operator. [RIGHT] The same location a with a recursive binary reducer of height two on top of it.

```

PARALLEL-MM( $Z, X, Y, n$ )
(1) parallel for  $i \leftarrow 1$  to  $n$  do
(2)   parallel for  $j \leftarrow 1$  to  $n$  do
(3)      $Z[i][j] \leftarrow 0$ 
(4)     for  $k \leftarrow 1$  to  $n$  do
(5)        $Z[i][j] \leftarrow Z[i][j] + X[i][k] \times Y[k][j]$ 

```

Figure 3: Parallel code that multiplies two $n \times n$ matrices $X[1..n][1..n]$ and $Y[1..n][1..n]$, and puts the result in $Z[1..n][1..n]$.

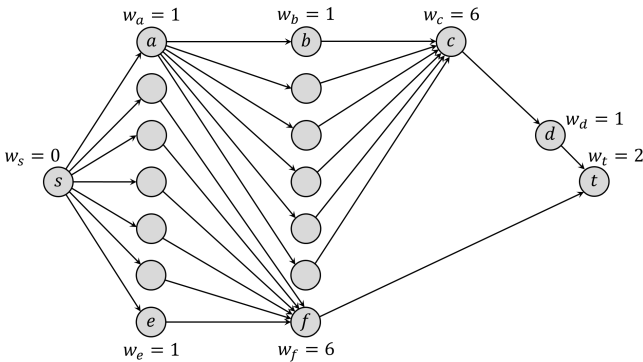


Figure 4: A DAG in which each node's work value is set to its in-degree. The makespan of this DAG is 11, and path $s \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow t$ achieves it.

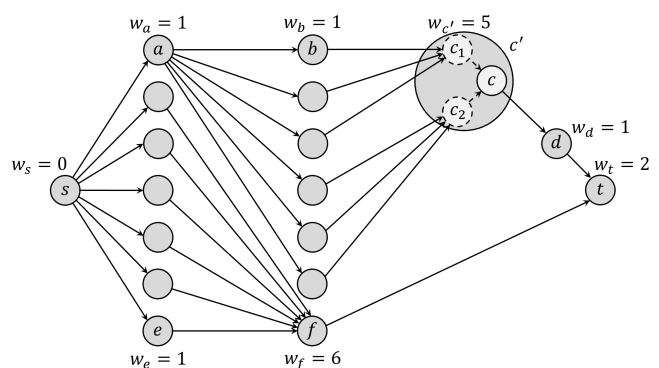


Figure 5: Node c from the DAG in Figure 4 has been replaced with a supernode c' in this figure which is nothing but node c with a reducer of height 1 on top. The makespan of this reduced DAG is 10, and path $s \rightarrow a \rightarrow b \rightarrow c_1 \rightarrow c \rightarrow d \rightarrow t$ achieves it.

recursive call is made and a pointer to that segment is passed to the recursive call. Each recursive call splits and distributes its segment among its child recursive calls and reclaims the space when the children complete execution. So, we will assume that all the given extra space initially reside at the source node (i.e., node with in-degree zero). Then they flow along the edges toward the sink node (i.e., node with outdegree zero) possibly splitting along outgoing edges and merging at the tip of incoming edges as they flow. Each unit of space reaching node x moves out of x along some outgoing edge as soon as x becomes fully updated and those edges trigger. Every unit of space may participate in the construction of multiple reducers (possibly zero) along the path it takes.

QUESTION 1.3. *Repeat Question 1.1 but now allow for space reuse among nodes of $D(P)$ by flowing each unit of space along a source to sink path and using it in the construction of zero or more reducers along that path.*

While several existing results [9, 10, 17, 32] can be extended to answer Questions 1.1 and 1.2, to the best of our knowledge, Question 1.3 had not been raised before. In this paper we investigate answers to Question 1.3 by extending it to a more general resource-time tradeoff question posed on a DAG in which nodes represent jobs (not necessarily of updating memory locations), resources (not necessarily space) flow along source to sink paths, and an general duration function (i.e., time needed to complete a job as a function of the amount of resources used) is specified for each node. We consider the following three duration functions: general non-increasing function for the general resource-time question, and recursive binary reduction and multiway (k -way) splitting for the space-time case.

For general DAGs, we show that even if the entire DAG is available to us offline the problem is strongly NP-hard under all three duration functions, and we give approximation algorithms for solving the corresponding optimization problems. We also prove hardness of approximation for the general resource-time tradeoff problem and give a pseudo-polynomial time algorithm for series-parallel DAGs. Our main results are summarized in Table 1.

Related Work

While several prior works either directly or indirectly address Questions 1.1 (nonreusable resources) and 1.2 (globally reusable resources), to the best of our knowledge, Question 1.3 (reusable along flow paths) has not been considered before.

The well-known time-cost tradeoff problem (TCTP) is closely related to our nonreusable resources question. In TCTP, some activities are expediated at additional cost so that the makespan can be shortened. Deadline and budget problems are two TCTP variants with different objectives. While the deadline problem seeks to minimize the total cost to satisfy a given deadline, the budget problem aims to minimize the project duration to meet the given budget constraint [4]. Most researchers consider the tradeoff functions to be either linear continuous or discrete giving rise to linear TCTP and discrete TCTP, respectively.

Linear TCTP was formulated by Kelley and Walker in 1959 [19]. They assumed affine linear and decreasing tradeoff functions. In 1961, linear TCTP was solved in polynomial time using network

flow approaches independently by Fulkerson [14] and Kelley [18]. Phillips and Dessouky [26] later improved that result.

In 1997, De et al. [9] proved that discrete TCTP is NP-hard. For this problem, Skutella [32] proposed the first approximation algorithm under budget constraints which achieves an approximation ratio of $O(\log r)$, where r is the ratio of the maximum duration of any activity to the minimum one. Discrete TCTP can also be used to approximate the TCTP with general time-cost tradeoff functions, see, e.g., Panagiotakopoulos [25] and Robinson [28]. For details on discrete TCTP see De et al. [8].

Our problem with globally reusable resources (Question 1.2) is very similar to the problem of scheduling precedence-constrained malleable tasks [33]. In 1978, Lenstra and Rinnooy Kan [20] showed that no polynomial time algorithm exists with approximation ratio less than $\frac{4}{3}$ unless $P = NP$. About 20 years later, Du and Leung [10] showed that the problem is strongly NP-hard even for two units of resources. In 2002, under the monotonous penalty assumptions of Blayo et al [6], Lepère et al. [21] first proposed the idea of two-step algorithms – computing an allocation first, and then scheduling tasks, and used this idea [22] to design a algorithm that achieve an approximation ratio of ≈ 5.236 . In the first phase, they approximate an allocation using Skutella’s algorithm [32]. Similarly, based on Skutella’s approximation algorithm, Jansen and Zhang [17] devised a two-phase approximation algorithm with the best-known ratio of ≈ 4.730598 and showed that the ratio is tight when the problem size is large. For more details on the problems of scheduling malleable tasks with precedence constraints, please check Dutot et al. [11].

There are memory allocators based on global memory manager for multi-core or multi-threaded systems such as scallocc [3], Hoard [5], lllallocc [2], Streamflow [30], and TCMalloc [1]. They use thread-local space for memory allocation and a global manager for memory deallocation/reuse. For the global manager, they use concurrent data structures. However, these data structures can not completely avoid the need for synchronization [3, 16, 31] without compromising correctness.

2 PRELIMINARIES, PROBLEM FORMULATION

In general, the option to use reducers to trade off between extra space and the time to complete race-free writing operations leads to a *discrete resource-time tradeoff problem*, where, here, the valuable “resource” is the space that is added, in order to reduce the time necessary for the write operations. By investing in additional space, we can reduce the time it takes to do conflict-free write operations.

We formalize the discrete resource-time tradeoff problem. Consider a DAG, $D = (V, E)$, whose vertices V correspond to jobs, and whose edges represent precedence relations among jobs. Without loss of generality, we assume that the DAG has a single source and a single sink vertex. The duration of a job depends on how much resource it receives. For each job $v \in V$, there is a non-increasing duration function $t_v(r)$ that denotes the time required to complete job v using r units of resources. We call $\langle r, t_v(r) \rangle$ a *resource-time tuple* associated with job (vertex) v . We consider three classes of duration functions – general non-increasing step functions, k -way splitting functions, and recursive binary splitting functions.

Duration function	Hardness	Hardness of Approximation	Approximation Results
General non-increasing	strongly NP-hard	<ul style="list-style-type: none"> • makespan < 2 OPT with resources fixed • resource $< \frac{3}{2}$ OPT with makespan fixed 	$(\frac{1}{\alpha}, \frac{1}{1-\alpha})$ bi-criteria (resource, makespan), $0 < \alpha < 1$
Recursive binary	strongly NP-hard	-	<ul style="list-style-type: none"> • makespan ≤ 4 OPT with resources fixed • $(\frac{4}{3}, \frac{14}{5})$ bi-criteria (resource, makespan)
Multiway splitting	strongly NP-hard	-	makespan ≤ 5 OPT with resources fixed

Table 1: Our main results on resource-time tradeoff problems in which resources are routed along source to sink paths (i.e., related to Question 1.3 and its generalization).

General non-increasing step function. Let l_v be the number of resource-time tuples associated with job v . The i -th resource-time tuple is $\langle r_{v,i}, t_v(r_{v,i}) \rangle$ where $1 \leq i \leq l_v$. Then, the duration function $t_v(r)$ is a step function with l_v steps described as follows:

$$t_v(r) = \begin{cases} t_v(r_{v,i}), & \text{if } r_{v,i} \leq r < r_{v,i+1}, 1 \leq i < l_v, \\ t_v(r_{v,l_v}), & \text{if } r_{v,l_v} \leq r, \end{cases} \quad (1)$$

where $r_{v,1} = 0, r_{v,j} < r_{v,j+1}$ and $t_v(r_{v,j}) \geq t_v(r_{v,j+1})$ for $1 \leq j < l_v$.

k -way splitting. A k -way split reducer utilizes k units of extra space, $S_v = \{s_1, s_2, \dots, s_k\}$, associated with a vertex v , with $2 \leq k \leq d_v^{(in)}$, such that the write operations associated with incoming edges at v are distributed among the vertices S_v , which then have edges linking each s_i to v . The duration function that results from k -way split reducers is given by

$$t_v(r) = \begin{cases} t_v(0), & \text{if } k \in \{0, 1\} \\ \lceil t_v(0)/k \rceil + k, & \text{if } 2 \leq k \leq \lfloor \sqrt{t_v(0)} \rfloor \\ t_v(\lfloor \sqrt{t_v(0)} \rfloor), & \text{if } \lfloor \sqrt{t_v(0)} \rfloor < k. \end{cases} \quad (2)$$

Recursive binary splitting. The duration function that results from a recursive binary split reducer is given by a step function, as follows. The resource-time tuples are defined for $r = 0$ and 2^i where $0 \leq i \leq k$ and $k = \lfloor \log_2 t_v(0) - \log_2 \log_2 e \rfloor$. The duration function $t_v(2^k) = \lceil t_v(0)/2^k \rceil + k + 1$ is minimized when $k = \lfloor \log_2 t_v(0) - \log_2 \log_2 e \rfloor$ (by differentiating $t_v(2^k)$ w.r.t. k).

$$t_v(r) = \begin{cases} t_v(0), & \text{if } r = 0, 1 \\ \lceil t_v(0)/2^i \rceil + i + 1, & \text{if } r = 2^i, 2 \leq i \leq k \\ t_v(2^i), & \text{if } 2^i \leq r < 2^{i+1}, 2 \leq i \leq k \\ t_v(2^k), & \text{if } i > k \end{cases} \quad (3)$$

When utilizing a reducer, extra space serves as the limited resource and the time taken for race-free writing at a vertex v is the duration of the job corresponding to v . Both the k -way splitting duration function and the recursive binary splitting duration function are special cases of general non-increasing function.

We consider jobs whose duration functions are of the types described above, and we distinguish between two optimization problems, depending on the objective function:

Minimum-Makespan Problem. Given a resource budget of B , assign the resources to the vertices V such that the makespan of the project is minimized. Resources can be reused over a path.

Minimum-Resource Problem. Given a makespan target of T , minimize the amount of resources to achieve target makespan. Resources can be reused over a path.

Finally, we remark that instead of jobs corresponding to vertices of the DAG, we can transform the DAG D into another DAG D' in which jobs correspond to edges of D' , and the precedence relations among jobs are enforced by introducing dummy edges, as follows: For each node v in D , we introduce an edge $e_v = (a_v, b_v)$ in D' (which then has the corresponding duration function, specified, e.g., by resource-time tuples). For each edge (u, v) of D , we introduce a dummy edge, $e = (b_u, a_v)$ in D' , from the endpoint b_u of edge $e_u = (a_u, b_u)$ to the origin a_v of edge $e_v = (a_v, b_v)$, with resource-time function $t_e(r) = 0$ for all valid resource levels r .

3 APPROXIMATION ALGORITHMS

3.1 Bi-criteria Approximation for Non-increasing Duration Functions

We use linear programming in our approximation algorithms. First, we relax the discrete duration function to a linear one. We transform the DAG so that a relaxed linear non-increasing duration function can be used. The transformation happens in two steps.

Activity on arc reduction. We reduce the input DAG D into an equivalent DAG D' with activities on arcs instead of nodes. This is a simple transformation described earlier in Section 2.

Activity with two tuples. Following [32], we create a DAG D'' from D' such that all activities in D'' are still on arcs and each such activity has at most 2 resource-time tuples as shown in Figure 6(b). Let j be a job with $l_j \geq 2$ resource-time tuples $\langle r_{j,i}, t_j(r_{j,i}) \rangle, 1 \leq i \leq l_j$ with $0 = r_{j,1} < r_{j,2} < \dots < r_{j,l_j}$ and $t_j(r_{j,1}) \geq t_j(r_{j,2}) \geq \dots \geq t_j(r_{j,l_j})$ (following Equation 1). Let edge (u, v) of D' represent job j . We add l_j parallel chains, each consisting of two edges in D'' (Figure 6). For $1 \leq i \leq l_j$, we create a chain of two edges (u, u_i) and (u_i, v) . We create a job j_i for arc (u, u_i) and associate two resource-time tuples with it. For $1 \leq i < l_j$, job j_i can be finished either using 0 resource in $t_j(r_{j,i})$ units of time or using $(r_{j,i+1} - r_{j,i})$ units of resource in 0 unit of time. The logic is that job j 's duration can be reduced from $t_j(r_{j,i})$ to $t_j(r_{j,i+1})$ provided the resource difference $(r_{j,i+1} - r_{j,i})$ is allocated to j_i . Thus the duration function is $t_{j_i}(0) = t_j(r_{j,i})$ and $t_{j_i}(r_{j,i+1} - r_{j,i}) = 0$. Job j_{l_j} 's (bottom most edge in the l_j parallel edges for job j) duration cannot be further improved from $t_j(r_{j,l_j})$ units of time by using extra resources. The resource-time tuple at edge (u_i, v) is $\langle 0, 0 \rangle$ where $1 \leq i \leq l_j$.

There is a canonical mapping of resource usages and durations for jobs j_i to that of job j . Let x_i be the units of resource used for job j_i , then for job j , $\sum_{i=1}^{l_j} x_i$ units of resource are used. The time taken to finish job j is $\max\{t_j(x_i) \mid 1 \leq i \leq l_j\}$. Without loss of generality, if we use 0 unit of resource for job j_i if $t_{j_i}(0) \leq$

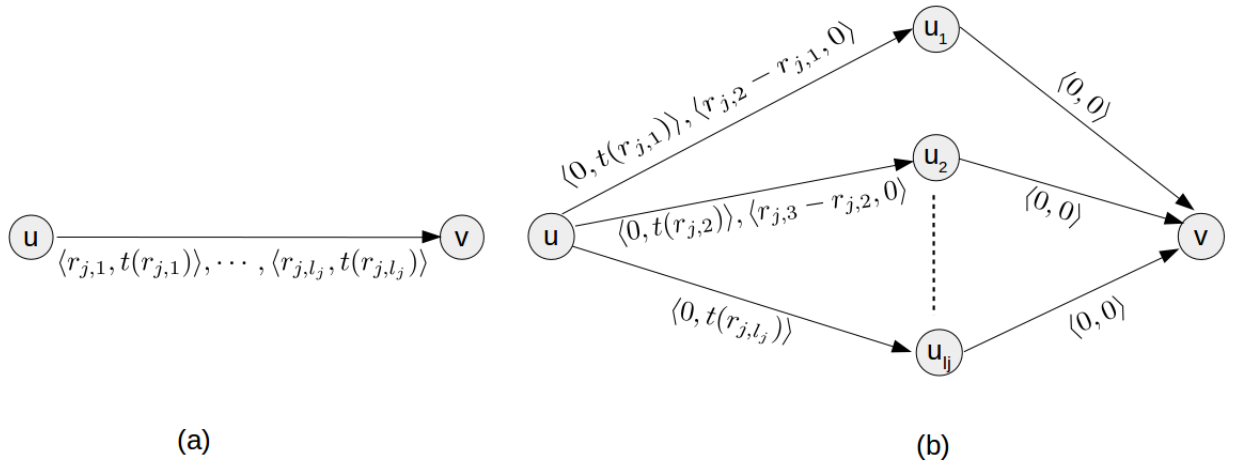


Figure 6: Transforming (a) a DAG with $l_j \geq 2$ resource-time tuples on each arc into (b) one with at most two resource-time tuples on each arc (Section 3.1)

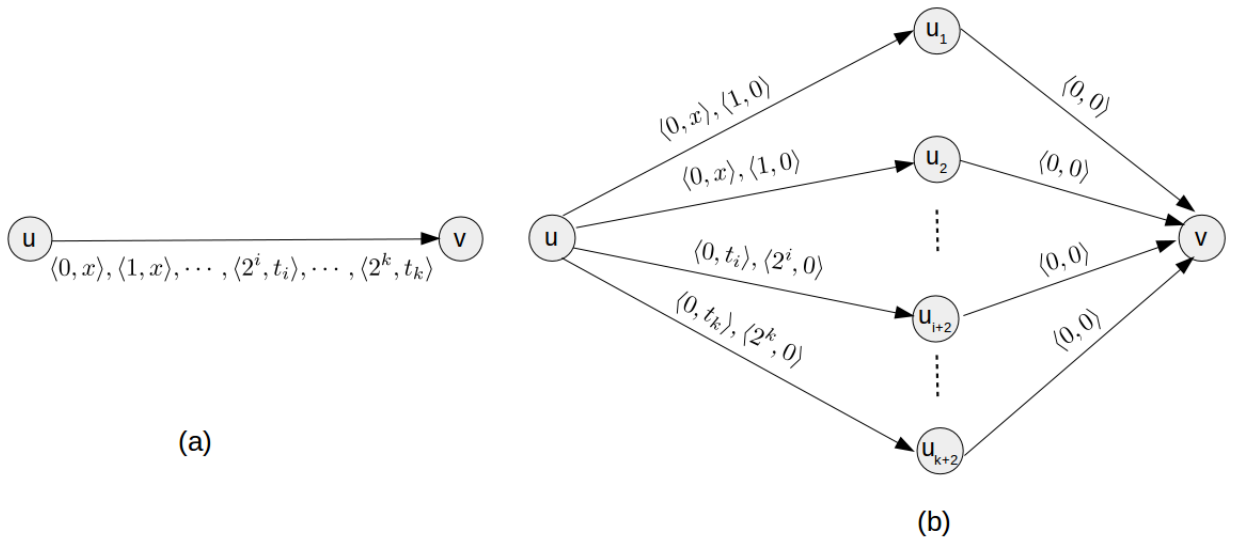


Figure 7: Transforming (a) a DAG with $(k+1)$ resource-time tuples on each arc based on the recursive binary splitting function into (b) one with at most two resource-time tuples on each arc (Section 3.3)

$\max\{t_{j,1}(x_1), t_{j,2}(x_2), \dots, t_{j,i-1}(x_{i-1})\}$, then this mapping is bijective. Thus we get the following lemma.

LEMMA 3.1. *Any approximation algorithm \mathcal{A} on DAG D'' (activity on edge and each edge has at most two resource-time tuples) with an approximation ratio α implies an approximation algorithm with the same approximation ratio α on general DAG D (activity on vertex and each job can have more than two resource-time tuples).*

From now on, we will only consider DAGs whose edges represent jobs, with each edge having at most two resource-time tuples.

Linear relaxation. In D'' , any edge (u, v) can have either two resource-time tuples $\{\langle 0, t_{(u,v)}(0) \rangle, \langle r_{(u,v)}, 0 \rangle\}$ or a single resource-time tuple $\{\langle 0, t_{(u,v)}(0) \rangle\}$. With linear relaxation, $r \in [0, r_{(u,v)}]$ units of resource can be used to reduce the completion time of the job corresponding to edge (u, v) that has two resource-time tuples. The corresponding duration function $t_{(u,v)}(r)$ is as follows:

$$t_{(u,v)}(r) = \frac{t_{(u,v)}(0)}{r_{(u,v)}} r \text{ for } r \in [0, r_{(u,v)}] \quad (4)$$

The linear duration function $t_{(u,v)}(r)$ for the job (u, v) with single resource-time tuple is as follows:

$$t_{(u,v)}(r) = t_{(u,v)}(0) \text{ for all } r \geq 0 \quad (5)$$

Linear programming formulation. Since we are allowed to reuse resources over a path we can model the problem as a network flow problem where resources are allowed to flow from the source to the sink in the DAG D'' . Let E be the set of edges in D'' . Let $f_{(u,v)}$ denote the amount of resources that flow through the edge (u, v) . Using linear relaxation on edge (u, v) , the time taken to finish the activity is $t_{(u,v)}(f_{(u,v)})$. Let the vertices in D'' denote events. From now onwards, we use a vertex and its corresponding event synonymously. Let $E_v = \{(x, v)\}$ be the set of edges that are incident on vertex v . Event v occurs if and only if all the jobs corresponding to the edges in set E_v are finished. Let T_v denote the time when event v occurs. Let s and t denote the source vertex and the sink vertex, respectively. For source vertex s , we assume $T_s = 0$. All variables are non-negative.

Constraints:

$$f_{(u,v)} \leq r_{(u,v)}, \quad \forall (u, v) \text{ with two resource-time tuples.} \quad (6)$$

$$T_u + t_{u,v}(f_{(u,v)}) \leq T_v, \quad \forall (u, v) \in E \quad (7)$$

$$\sum_w f_{(v,w)} + \sum_u f_{(u,v)} = 0, \quad \forall v \notin \{s, t\} \quad (8)$$

$$\sum_k f_{(s,k)} \leq B \quad (9)$$

Objective function:

$$\min T_t \quad (10)$$

Inequality 6 upper bounds the resource flow variable $f_{(u,v)}$ for edges with two tuples. This ensures that these variables remain in the range $[0, r_{(u,v)}]$ and the duration function is linear in this range. Note that there is no such upper bound on the edges with single resource-time tuple (except the trivial total resource budget B upper bound). This allows the flow of more resources over an edge

that can be used later on a path. Equation 8 is a flow conservation constraint for all the vertices $v \notin \{s, t\}$. Inequality 9 constrains the flow of resources from source s to be upper bounded by the resource budget.

Solving the LP and rounding. We first solve the LP described above. This might give solution as fractional flow f_e^* and duration $t_e(f_e^*)$ at edge $e = (u, v)$. Let the resource-time tuples at edge e be $\{\langle 0, t_e(0) \rangle, \langle r_e, 0 \rangle\}$. The range of feasible duration of activity e is $[0, t_e(0)]$. We divide this range into two parts $[0, \alpha t_e(0)]$, $[\alpha t_e(0), t_e(0)]$ where $0 < \alpha < 1$. If $t_e(f_e^*) \in [0, \alpha t_e(0)]$ we round it down to 0, otherwise, we round it up to $t_e(0)$. Observe that in the first case, the resource requirement at e can be increased by at most a factor of $1/(1 - \alpha)$. In the second case, the completion time can be increased at most by a factor of $1/\alpha$. Let f'_e denote the rounded integer resource requirement at edge e .

Computing min-flow. After rounding the LP solution, we get an integral resource requirement $f'_e \in \{0, r_e\}$ for every edge e . We now compute a min-flow through this DAG where f'_e serves as the lower bound on the flow through (or resource requirement at) edge e .

Constraints:

$$f_{(u,v)} \geq f'_{(u,v)}, \quad \forall (u, v) \in E \quad (11)$$

$$\sum_w f_{(v,w)} + \sum_u f_{(u,v)} = 0, \quad \forall v \notin \{s, t\} \quad (12)$$

Objective function:

$$\min \sum_k f_{(s,k)} \quad (13)$$

Let, f and f^* be the optimal solutions of LP 11–13 and LP 6–10, respectively.

LEMMA 3.2. $f^*/(1 - \alpha)$ is a feasible solution of min-flow LP 11–13.

PROOF. Let f_e^* be the optimal solution of LP 6–10. We know that $f'_e \leq f_e^*/(1 - \alpha)$. Hence, $f^*/(1 - \alpha)$ is a feasible solution of that LP as it meets the resource requirement f'_e at every edge e . \square

LEMMA 3.3. f is an integral flow and $f \leq f^*/(1 - \alpha)$, where $0 < \alpha < 1$.

PROOF. The minflow problem has integral optimality. If f is the optimal solution then it is an integral flow. From lemma 3.2 we know that $f^*/(1 - \alpha)$ is a feasible solution of LP 11–13. Since f is optimal and $f^*/(1 - \alpha)$ is a feasible flow, we have, $f \leq f^*/(1 - \alpha)$. \square

Bi-criteria approximation. We now summarize our bi-criteria approximation result for general non-increasing duration functions:

THEOREM 3.4. *For any $\alpha \in (0, 1)$, there is a $(1/\alpha, 1/(1 - \alpha))$ bi-criteria approximation algorithm for the discrete resource-time trade-off problem with an general non-increasing duration function which allows resource reuse over paths.*

PROOF. First, we know from lemma 3.3 that f is an integral flow and $f \leq f^*/(1 - \alpha)$, where $0 < \alpha < 1$.

Second, we claim that the makespan of the DAG used in the minflow LP 11–13 is at most a factor of $1/\alpha$ away from that of the LP 6–10 solution. Let us consider any $s - t$ path \mathcal{P} . The makespan

is at least the sum of completion times of the edges in \mathcal{P} . Now, after rounding the LP 6–10 solution, the completion time of an edge may increase at most by a factor of α . Hence, the sum of duration of edges along any path is increased at most by a factor of α , thus the makespan will be increased by at most a factor of α . \square

3.2 Single-criteria Approximation for k -Way and Recursive Binary Splitting

First, observe the prior section gives us a bi-criterion approximation for both k -way and recursive binary splitting. Setting $\alpha = 1/2$ in Theorem 3.4, we obtain a $(2, 2)$ bi-criteria approximation. Now, after LP rounding, say a job j uses \bar{r}_j units of resource and takes \bar{t}_j units of time. Then the optimal solution uses $r_j^* \geq \bar{r}_j/2$ units of resource and takes $t_j^* \geq \bar{t}_j/2$ units of time for job j . Recall that job j consists of l_j parallel jobs j_i where $1 \leq i \leq l_j$. Hence, \bar{r}_j is the sum of the resource (after rounding) used by l_j parallel jobs and \bar{t}_j is the maximum time (after rounding) taken by l_j parallel jobs.

Approximation algorithm for k -way splitting. To obtain a single-criteria approximation, in the case of k -way splitting, we use at most r_j^* units of resource for job j . If $\bar{r}_j > r_j^*$, we reduce \bar{r}_j to k (a nonnegative integer) units of resource such that $k \leq r_j^*$. Using k units of resource, job j takes $t_j(k)$ units of time to complete.

LEMMA 3.5. $[d/k] + k \leq 2.5\bar{t}_j$ for $\bar{r}_j > 3$ where $d = t_j(0)$ and $k = \lfloor \bar{r}_j/2 \rfloor$.

PROOF. Since $k = \lfloor \bar{r}_j/2 \rfloor \geq \bar{r}_j/2.5$ for $\bar{r}_j > 3$, we have $[d/k] \leq d/k + 1 \leq 2.5d/\bar{r}_j + 1 \leq 2.5[d/\bar{r}_j] + 1$. Also since $k = \lfloor \bar{r}_j/2 \rfloor \leq \bar{r}_j + 1$ and $2.5\bar{r}_j \geq \bar{r}_j + 2$ for $\bar{r}_j > 3$, we have $[d/k] + k \leq 2.5[d/\bar{r}_j] + 1 + \bar{r}_j + 1 \leq 2.5([d/\bar{r}_j] + \bar{r}_j)$. Hence, $t_j(k) \leq 2.5\bar{t}_j$. \square

LEMMA 3.6. If $\bar{r}_j > 3$ then $t_j(k) \leq 5t_j^*$.

PROOF. We know $t_j(k) = [d/k] + k$ as $k \geq 4$. Also in lemma 3.5, we prove $t_j(k) \leq 2.5\bar{t}_j$. However, we show that $\bar{t}_j \leq 2t_j^*$. Hence, combining these two results we get $t_j(k) \leq 5t_j^*$. \square

LEMMA 3.7. If $t_j^* = d/4$ then $r_j^* \geq 2$.

PROOF. Recall that in D'' , job j is represented as l_j parallel jobs j_i where $1 \leq i \leq l_j$. The resource-time tuples of jobs j_1 and j_2 are $\{(0, d), \langle 2, 0 \rangle\}$ and $\{(0, [d/2] + 2), \langle 1, 0 \rangle\}$, respectively. To attain $d/4$ duration, j_1 requires at least $3/2$ units of resource and job j_2 requires $1/2$ unit of resource (applying linear relaxation). Hence, $r_j^* \geq (3/2 + 1/2) = 2$ units of resource to achieve $t_j^* = d/4$. \square

LEMMA 3.8. If $\bar{r}_j \leq 3$ then $t_j(k) \leq 4t_j^*$.

PROOF. If $\bar{r}_j \leq 3$ and $r_j^* < 2$, then we round down \bar{r}_j to $k = 0$. So, from Lemma 3.7 it follows that after rounding down to 0 unit of resource, job j takes $d \leq 4t_j^*$ units of time.

If $\bar{r}_j \leq 3$ and $r_j^* \geq 2$, then we round \bar{r}_j to $k = 2$. It is true that $t_j(2) \leq 2t_j(3)$ because $([d/2] + 2) \leq 2([d/3] + 3)$. Also, $t_j(3) \leq t_j(\bar{r}_j) \leq 2t_j^*$. Combining this two results we get $t_j(2) \leq 4t_j^*$. \square

So, now we have the following result.

THEOREM 3.9. *There is a 5-approximation algorithm for the minimum-makespan problem with k -way splitting duration function.*

PROOF. Combining Lemmas 3.8 and 3.6 we get $t_j(k) \leq 5t_j^*$ for all valid \bar{r}_j . This proves that the makespan is at most 5 times the optimal solution. We now calculate the total amount of resource required to flow from the source of D' . We compute a min-flow in D' where k is the resource requirement for job j . Note that we are now working on D' that does not have l_j parallel chains for job j . Let f be the min flow from the source of D' such that all the resource requirements are met. The flow f^* from the LP solution before rounding is also a valid flow for the resource requirement k for job j as $k \leq r_j^*$. We know that min-flow gives an optimal integral solution. Hence, $f \leq f^*$. \square

Approximation algorithm for recursive binary splitting. We have the following result.

THEOREM 3.10. *There is a 4-approximation algorithm for the minimum-makespan problem with recursive binary splitting function.*

PROOF. As in the case of k -way splitter, to get a single-criteria approximation, we use no more than r_j^* units of resource for job j . If $\bar{r}_j > r_j^*$, we reduce \bar{r}_j to $\bar{r}_j/2$. We know that $t_j(\bar{r}_j/2) \leq 2t_j(\bar{r}_j)$ from the properties of the recursive binary splitting function. Thus, $t_j(\bar{r}_j/2) \leq 2t_j(\bar{r}_j) \leq 4t_j(r_j^*) = 4t_j^*$. \square

3.3 Improved Bi-criteria Approximation for Recursive Binary Splitting Functions

Putting $\alpha = 3/4$ in Theorem 3.4 we obtain a $(4/3, 4)$ bi-criteria approximation algorithm for general non-increasing duration functions. Hence, if we use $4/3$ times more resources than OPT (i.e., the optimal solution), we are guaranteed to get a makespan within factor of 4 of OPT. In this section we show that the bound can be improved to $(4/3, 14/5)$ for recursive binary splitting functions.

For a node with in-degree x , the resource-time tuples based on the recursive binary splitting function are as follows: $\{(0, x), \langle 1, x \rangle, \langle 2, t_1 \rangle, \dots, \langle 2^i, t_i \rangle, \langle 2^{i+1}, t_{i+1} \rangle, \dots, \langle 2^k, t_k \rangle\}$ where $t_j = \lceil x/2^j \rceil + j + 1$ for $j \geq 2$ and $k = \lfloor \log_2 x - \log_2 \log_2 e \rfloor$ is the largest value of j for which t_j decreases with the increase of j . See Figure 7.

After solving LP 6–10 from Section 3.1, we sum up the (possibly fractional) resources allocated to all the l_j parallel edges corresponding to job j . Let r be that sum. Let t be the maximum among the time values given by the LP solution for the l_j parallel edges. Thus, the LP takes t units of time for job j .

We round r to an integer \bar{r} based on the following criteria.

$$\bar{r} = \begin{cases} 0, & \text{if } r < 1 \\ 2^i & \text{if } 2^i \leq r < (2^i + 2^{i+1})/2, 0 \leq i \leq k \\ 2^{i+1}, & \text{if } (2^i + 2^{i+1})/2 \leq r < 2^{i+1}, 0 \leq i \leq k \end{cases}$$

We want to find a constant ρ , such that if $t = t_i/\rho$, then the LP must use at least $(2^i + 2^{i+1})/2 = 3(2^{i-1})$ units of resources. We compute r as follows. In Figure 7(b), each of the top two edges (u, u_1) and (u, u_2) requires $(1 - (1/x)t)$ units of resource to finish in time t . Each edge (u, u_{j+2}) for $1 \leq j \leq i + 1$ requires $(2^j - (2^j/t_j)t)$ units of resource to finish in time t . Summing over all these edges,

we get the expression of r

$$r = 2 \left(1 - \frac{1}{x} t \right) + \sum_{j=1}^{i+1} \left(2^j - \frac{2^j}{t_j} t \right) = 8 \cdot (2^{i-1}) - \frac{t_i}{\rho} \left(2/x + \sum_{j=1}^{i+1} \frac{2^j}{t_j} \right)$$

Since we want to have $r \geq 3(2^{i-1})$, we want to find the smallest value of ρ such that

$$\frac{t_i}{\rho} \left(2/x + \sum_{j=1}^{i+1} \frac{2^j}{t_j} \right) \leq 5 \cdot (2^{i-1}) \Rightarrow \rho \geq 1/5 \left(\frac{t_i}{2^{i-2}x} + \sum_{j=1}^{i+1} \frac{t_i}{2^{i-j-1}t_j} \right).$$

Now,

$$\begin{aligned} \frac{t_i}{2^{i-2}x} + \sum_{j=1}^{i+1} \frac{t_i}{2^{i-j-1}t_j} &= \frac{\lceil \frac{x}{2^i} \rceil + i + 1}{x(2^{i-2})} + \sum_{j=1}^{i+1} \frac{\lceil \frac{x}{2^i} \rceil + i + 1}{(\lceil \frac{x}{2^j} \rceil + j + 1)2^{i-j-1}} \\ &< \frac{\frac{x}{2^i} + i + 2}{x(2^{i-2})} + \sum_{j=1}^{i+1} \frac{\frac{x}{2^j} + i + 2}{(\frac{x}{2^j} + j + 1)2^{i-j-1}} \\ &= \frac{1}{2^i} \frac{1}{2^{i-2}} + \frac{i + 2}{x(2^{i-2})} + \sum_{j=1}^{i+1} \frac{\frac{1}{2^{i-j}} (\frac{x}{2^j} + j + 1) + i + 2 - \frac{j}{2^{i-j}} - \frac{1}{2^{i-j}}}{(\frac{x}{2^j} + j + 1)2^{i-j-1}} \\ &\leq \left(\frac{i + 2}{x} \frac{1}{2^{i-2}} + \sum_{j=1}^{i+1} \frac{i + 2}{(\frac{x}{2^j} + j + 1)2^{i-j-1}} \right) + \left(\frac{1}{2^i} \frac{1}{2^{i-2}} + \sum_{j=1}^{i+1} \frac{1}{2^{i-j}} \frac{1}{2^{i-j-1}} \right) \\ &= \left(\frac{i + 2}{x} \frac{1}{2^{i-2}} \right) + \left(\sum_{j=1}^{i+1} \frac{i + 2}{(\frac{x}{2^j} + j + 1)2^{i-j-1}} \right) + \left(\frac{32}{3} + \frac{1}{3} \frac{1}{4^{i-1}} \right) \end{aligned}$$

Let, $A = \frac{i+2}{x} \frac{1}{2^{i-2}}$, $B = \sum_{j=1}^{i+1} \frac{i+2}{(\frac{x}{2^j} + j + 1)2^{i-j-1}}$ and $C = 32/3 + \frac{1}{3} \frac{1}{4^{i-1}}$.

Note that $i + 2 = (i + 1) + 1 \leq (\log_2 x - \log_2 \log_2 e) + 1$, since $i + 1 \leq k$. Hence,

$$A \leq \frac{(\log_2 x - \log_2 \log_2 e) + 1}{x} \frac{1}{2^{i-2}} \leq \frac{2}{e} \frac{1}{2^{i-2}}.$$

Now, $x/2^j + j + 1 \geq (\log_2 x - \log_2 \log_2 e + \frac{1}{\ln 2})$ and hence,

$$\begin{aligned} B &\leq \sum_{j=1}^{i+1} \frac{(\log_2 x - \log_2 \log_2 e) + 1}{(\log_2 x - \log_2 \log_2 e + \frac{1}{\ln 2} + 1)} \frac{1}{2^{i-j-1}} \\ &< \sum_{j=1}^{i+1} \frac{1}{2^{i-j-1}} = 2 - \frac{1}{2^{i-2}}. \end{aligned}$$

Thus, $A + B + C < \frac{2}{e} \frac{1}{2^{i-2}} + 2 - \frac{1}{2^{i-2}} + 32/3 + \frac{1}{3} \frac{1}{4^{i-1}} \leq 14$.

Therefore, $(t_i/x) \frac{1}{2^{i-2}} + \sum_{j=1}^{i+1} \frac{t_i}{t_j} \frac{1}{2^{i-j-1}} < 14$.

So, by setting $\rho = 14/5$, we get $\rho > 1/5 \left((t_i/x) \frac{1}{2^{i-2}} + \sum_{j=1}^{i+1} \frac{t_i}{t_j} \frac{1}{2^{i-j-1}} \right)$. Summarizing, we get the following lemmas from the computation above.

LEMMA 3.11. *To achieve a duration of $t = t_i/(14/5)$ for any job j , the LP solution uses at least $3(2^{i-1})$ units of resources for $0 \leq i \leq k$.*

Lemma 3.11 implies the following.

LEMMA 3.12. *If the LP uses $2^i \leq r < 3(2^{i-1})$ units of resources and we round r down to $\bar{r} = 2^i$ where $0 \leq i \leq k$, then $t_i \leq (14/5)t$ where t is the duration from the LP solution.*

LEMMA 3.13. *With $r < 1$ units of resource, the LP cannot achieve a duration of $t < x/2$ for job j .*

PROOF. The first edge has resource-time tuples $\{(0, x), (1, 0)\}$. To achieve a duration of $x/2$, the LP has to use $1/2$ unit of resource on the first edge. The second edge also has the same resource-time tuples $\{(0, x), (1, 0)\}$, and it also takes $1/2$ unit of resource. Thus, the first two edges alone need 1 unit of resource to achieve a duration of $x/2$ for all l_j parallel edges of job j . \square

Lemma 3.13 implies the following.

LEMMA 3.14. *If the LP uses $r < 1$ unit of resource and we round r down to 0, then $t_i \leq 2t$, where t is the duration from the LP solution.*

LEMMA 3.15. *If r rounded to \bar{r} then $\bar{r} \leq (4/3)r$*

PROOF. When we use $\bar{r} = 2^{i+1}$ units of resource after rounding, the LP uses at least $3(2^{i-1}) \leq r \leq 2^{i+1}$ units. Thus, $\bar{r} \leq (4/3)r$. \square

From Lemma 3.12 and Lemma 3.15, we get the following theorem.

THEOREM 3.16. *There is a $(4/3, 14/5)$ bi-criteria approximation algorithm for the discrete resource-time tradeoff problem with resource reuse along paths when the recursive binary duration function is used.*

3.4 Exact Algorithm for Series-Parallel Graphs

We consider now the special case in which the underlying DAG D is a series-parallel graph. A series-parallel graph G can be transformed into (and represented as) a rooted binary tree T_G in polynomial time by decomposing it into its atomic parts according to its series and parallel compositions (see, e.g., [23]). In T_G , the leaves correspond to the vertices of G . Internal nodes of T_G are labeled as “s” or “p” based on series or parallel composition. We associate each internal node v of T_G with the series-parallel graph G_v , induced by the leaves of the subtree rooted at v .

Let $T(v, \lambda)$ denote the makespan of G_v using $0 \leq \lambda \leq B$ units of resources where B is the resource budget. We want to solve for $T(s, B)$, where s is the root of T_G . This can be done using dynamic programming, solving for the leaves first, and then progressing upward to the root of T_G . We compute $T(v, \lambda)$ as follows which assumes that node v corresponds to job j if it is a leaf, otherwise it has two children v_1 and v_2 .

$$T(v, \lambda) = \begin{cases} t_j(\lambda) & \text{if } v \text{ is a leaf} \\ T(v_1, \lambda) + T(v_2, \lambda) & \text{if } v \text{ is an internal node with label "s"} \\ \min_{0 \leq i \leq \lambda} \left\{ \max \left\{ \begin{array}{l} T(v_1, i), \\ T(v_2, \lambda - i) \end{array} \right\} \right\} & \text{if } v \text{ is an internal node with label "p"} \end{cases}$$

There are $O(m)$ nodes in T_G if G has m edges. For each node v we compute $T(v, \lambda)$ for $0 \leq \lambda \leq B$. Computing $T(v, \lambda)$ for any particular value of λ takes $O(\lambda)$ time, since, if the node is a “p” node, then for $0 \leq i \leq \lambda$ we need to look up values $T(v_1, i)$. Thus, for any internal node v , it takes $\sum_{\lambda=0}^B O(\lambda) = O(B^2)$ time. As there are $O(m)$ nodes in T_G , the (pseudo-polynomial) time complexity of the algorithm is $O(mB^2)$.

4 NP-HARDNESS

In this section we give a variety of NP-hardness and inapproximability results related to the discrete time-resource tradeoff problem in the offline setting (i.e., when the entire DAG is available offline). All problems consider the version where there is resource reuse over paths, but they vary the cost-function, graph structure, and minimization goal. Section 4.1 gives several reductions from 1-in-3SAT. Theorem 4.1 gives a base reduction for the problem with general non-increasing duration function which will provide the ideas and structure for later more complex proofs. Theorems 4.3 and 4.4 adapt this proof to give constant factor inapproximability for the minimum-resource and minimum-makespan problems. Section 4.2 adapts the NP-hardness proof to apply when the cost function is restricted to be the recursive binary splitting and the k -way splitting.

Section 4.3 considers the problem in bounded treewidth graphs. We show weak NP-hardness by a reduction from Partition.

4.1 Reuse Over a Path with General Non-increasing Duration Function

THEOREM 4.1. *It is (strongly) NP-hard to decide if there exists a solution to the (offline) discrete resource-time tradeoff problem, with resource reuse over paths and a non-increasing duration function, satisfying a resource bound B and a makespan bound T .*

Our proof is based on a polynomial-time reduction from the strongly NP-hard problem 1-in-3SAT [29]: Given n variables ($V_i, 1 \leq i \leq n$) and m clauses ($C_j, 1 \leq j \leq m$), with each clause a disjunction of three literals, is there a truth assignment to the variables such that each clause has exactly one true literal?

Variable gadget. The gadget for variable V consists of nodes $V^{(1)}, V^{(2)}, V^{(3)}, V^{(4)}, V^{(5)}$, and $V^{(6)}$ as shown in Figure 8(a). We show in the hardness proof that a variable gadget will get exactly one unit of extra resource, otherwise the makespan will be greater than the target makespan of 1. Sending one unit of resource to node $V^{(2)}$ (Figure 8(a)) corresponds to setting the variable V to TRUE and sending the unit of resource to $V^{(3)}$ corresponds to setting V to FALSE. The remaining vertices ensure the extra resource is used in the variable and not transferred into one of the clauses.

Clause gadget. The gadget corresponding to clause C has 10 vertices $C^{(i)}$ ($1 \leq i \leq 10$) as shown in Figure 8(b). Arcs $(C^{(1)}, C^{(2)})$, $(C^{(2)}, C^{(4)})$, $(C^{(1)}, C^{(3)})$ and $(C^{(3)}, C^{(4)})$ have resource-time pairs as $\{(0, 1), \langle 1, 0 \rangle\}$. If clause C has three literals V_i, V_j and V_k , then vertex $C^{(5)}$ is connected to the vertices $V_i^{(3)}, V_j^{(3)}$ and $V_k^{(2)}$. These vertices correspond to $\neg V_i, \neg V_j$ and V_k respectively. Vertex $C^{(6)}$ is connected to $V_i^{(3)}, V_j^{(2)}$ and $V_k^{(3)}$. These vertices correspond to $\neg V_i, V_j$ and $\neg V_k$. Vertex $C^{(7)}$ is connected to $V_i^{(2)}, V_j^{(3)}$ and $V_k^{(3)}$. These vertices correspond to $V_i, \neg V_j$ and $\neg V_k$. Arcs $(C^{(5)}, C^{(8)})$, $(C^{(6)}, C^{(9)})$, and $(C^{(7)}, C^{(10)})$ have resource-time pairs as $\{(0, 1), \langle 1, 0 \rangle\}$. The part of the clause gadget consisting of $C^{(1)}, C^{(2)}, C^{(3)}$ and $C^{(4)}$ demand at least two units of memory be allocated there and then these units of resource go to satisfy two of $C^{(5)}, C^{(6)}$ and $C^{(7)}$. There is still one of these lines that has no allocated resource so its cost is 1. Thus the corresponding variable must have had its path length reduced (by setting it true).

Figure 9 shows the complete construction of $(V_1 \vee \neg V_2 \vee V_3) \wedge (\neg V_1 \vee V_2 \vee V_3)$ as an example.

LEMMA 4.2. *There exists a solution to the input instance of 1-in-3SAT iff there exists a valid flow of resources through the DAG achieving a makespan of 1 under a resource bound of $B = n + 2m$.*

PROOF. Forward direction. We prove that if there is a solution to the 1-in-3SAT instance with n variables and m clauses, then the reduced DAG has a solution of makespan 1 with $(n + 2m)$ units of resource. If a variable V 's truth assignment is TRUE, then we allow one unit of resource to flow through vertex $V^{(2)}$ along the path $\langle S, V^{(1)}, V^{(2)}, V^{(4)}, V^{(5)}, V^{(6)}, T \rangle$, otherwise we allow one unit of resource to flow through vertex $V^{(3)}$ along the path $\langle S, V^{(1)}, V^{(3)}, V^{(4)}, V^{(5)}, V^{(6)}, T \rangle$. For every clause C , we allow one unit of resource to flow through the path $\langle S, C^{(1)}, C^{(2)}, C^{(4)} \rangle$ and another unit of resource through the path $\langle S, C^{(1)}, C^{(3)}, C^{(4)} \rangle$. Thus, 2 units of resource can be flowed from vertex $C^{(4)}$. In a valid assignment of 1-in-3SAT, for each clause C , exactly 2 vertices of $C^{(5)}, C^{(6)}$ and $C^{(7)}$ will have the earliest start time of 1 and the other one will have 0 (Table 2).

Also, if only one literal is true in a clause, exactly two vertices among $C^{(5)}, C^{(6)}$ and $C^{(7)}$ need one unit of extra resource each to meet the makespan requirement (from Table 2). We are allowed to flow 2 units of resource from vertex $C^{(4)}$. Thus the project makespan is 1 using $(n + 2m)$ units of resource.

Backward direction. Now, we prove that if there exists a solution of makespan 1 using $(n + 2m)$ units of resource in the reduced DAG, then there also exists a solution to the 1-in-3SAT instance. To achieve a makespan of 1, every variable gadget needs 1 unit of resource and each clause gadget needs 2 units of resource, otherwise the makespan would be greater than 1. Also, any resource that is used in a variable gadget cannot be used further in any other variable or clause gadget because the resource can be reused over a path only. Similarly, any resource that is used in any clause gadget, cannot be reused in any other gadget. Only one vertex that is either $V^{(2)}$ or $V^{(3)}$, will have the earliest start time 0. Both cannot be 0, as there is only 1 unit of resource per variable gadget. Both cannot be 1 as in a clause C where the literal V or $\neg V$ is present, each of $C^{(5)}, C^{(6)}$ and $C^{(7)}$ would have earliest starting time of 1. This requires use of 3 units of resource in the clause gadget C to achieve a makespan of 1. However, each clause gadget can have exactly 2 units of resource. Thus, for every variable, it has to be a valid assignment (V is set to either TRUE or FALSE). From Table 2, if a clause has exactly one TRUE literal, then the clause gadget requires 2 units of resource to achieve a makespan of 1. Otherwise, the clause gadget would have a makespan of 2 with the same amount of resource or would require more resource to achieve the target makespan of 1. Thus, each clause has exactly one TRUE literal. This satisfies the 1-in-3SAT instance. \square

We also prove hardness of approximation, both for the minimum-makespan problem and for the minimum-resource problem. We begin with the minimum-makespan problem.

THEOREM 4.3. *The minimum-makespan discrete resource-time tradeoff problem that allows resources to be reused only over paths cannot have a polynomial-time approximation algorithm with approximation factor less than 2 unless $P = NP$.*

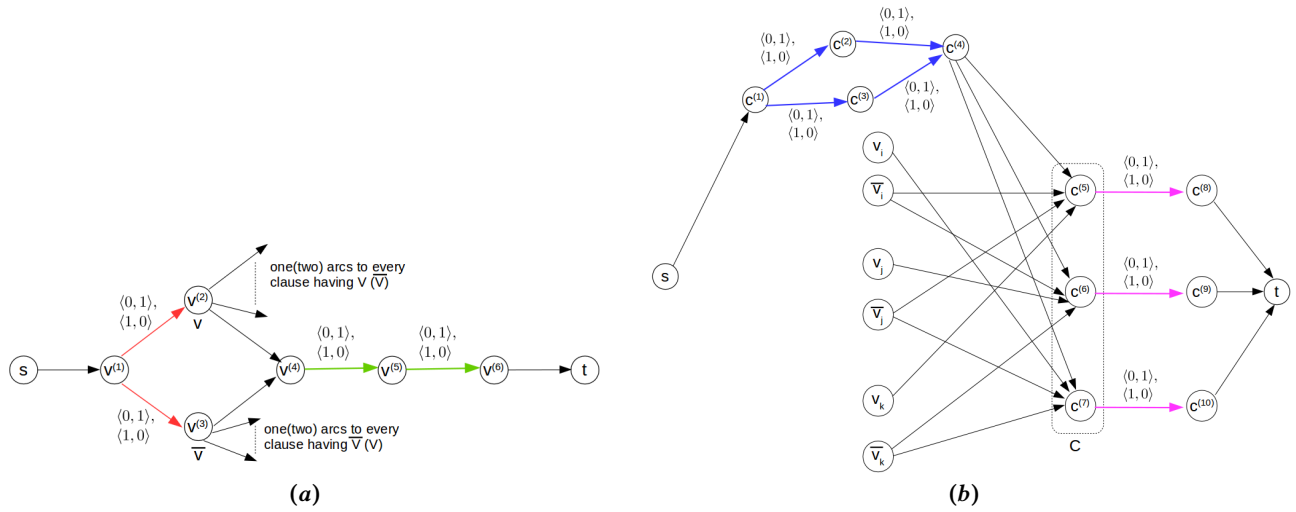


Figure 8: (a) Gadget for variable V , and (b) gadget for clause $C = (V_i \vee V_j \vee V_k)$ (Section 4.1).

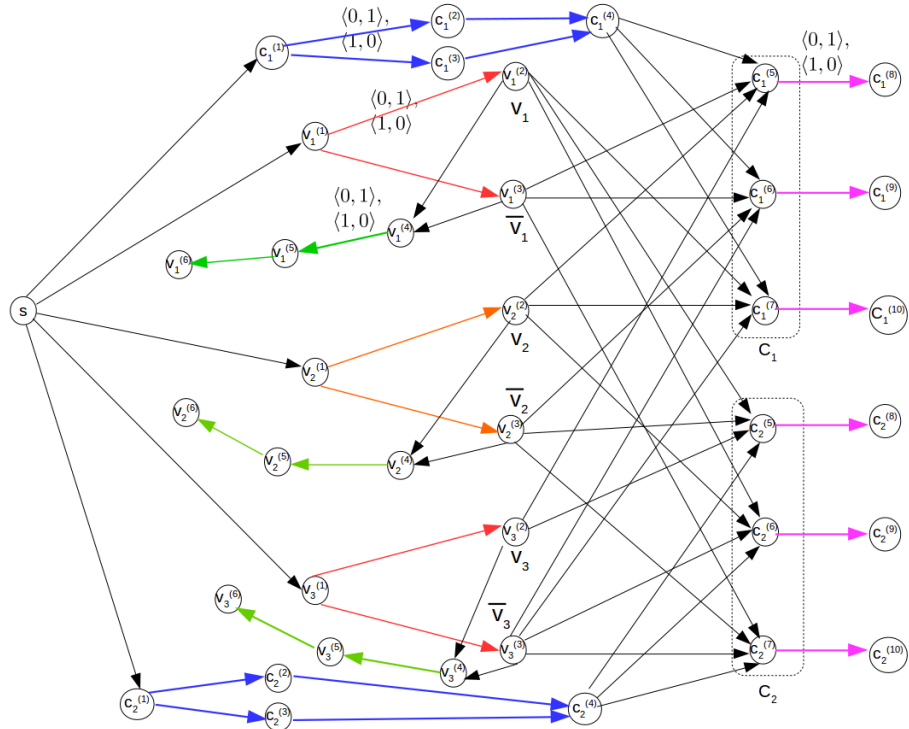


Figure 9: The complete construction for $(V_1 \vee \neg V_2 \vee V_3) \wedge (\neg V_1 \vee V_2 \vee V_3)$ is satisfiable with the truth assignment: $V_1 = \text{TRUE}, V_2 = \text{TRUE}, V_3 = \text{FALSE}$ (Section 4.1).

V_i	V_j	V_k	$C^{(5)}$	$C^{(6)}$	$C^{(7)}$
True	True	True	$\max(1, 1, 0) = 1$	$\max(1, 0, 1) = 1$	$\max(0, 1, 1) = 1$
False	True	True	$\max(0, 1, 0) = 1$	$\max(0, 0, 1) = 1$	$\max(1, 1, 1) = 1$
True	False	True	$\max(1, 0, 0) = 1$	$\max(1, 1, 1) = 1$	$\max(0, 0, 1) = 1$
True	True	False	$\max(1, 1, 1) = 1$	$\max(1, 0, 0) = 1$	$\max(0, 1, 0) = 1$
False	False	True	$\max(0, 0, 0) = 0$	$\max(0, 1, 1) = 1$	$\max(1, 0, 1) = 1$
False	True	False	$\max(0, 1, 1) = 1$	$\max(0, 0, 0) = 0$	$\max(1, 1, 0) = 1$
True	False	False	$\max(1, 0, 1) = 1$	$\max(1, 1, 0) = 1$	$\max(0, 0, 0) = 0$
False	False	False	$\max(0, 0, 1) = 1$	$\max(0, 1, 0) = 1$	$\max(1, 0, 0) = 1$

Table 2: Makespan at vertices $C^{(5)}$, $C^{(6)}$ and $C^{(7)}$ for different truth value assignments to V_i , V_j and V_k in Figure 8(b).

PROOF. We prove the theorem by contradiction. Let’s assume that there is a polynomial time approximation algorithm with factor less than 2. Given a formula with n variables and m clauses, we construct the reduced DAG as described in the proof of Lemma 4.2. If the formula is a valid 1-in-3SAT instance, then OPT (i.e., the optimal solution) has a makespan of 1 using $(n + 2m)$ units of resource in the reduced DAG. The approximation algorithm will return a schedule with makespan less than 2 using $(n + 2m)$ units of resource. If the formula is not a valid 1-in-3SAT instance, then OPT’s makespan is greater than or equal to 2. So, the approximation algorithm will have a schedule with makespan greater than or equal to 2. Thus, using a polynomial time algorithm one can solve a strongly NP-hard problem. This is a contradiction. Hence, there exists no polynomial time approximation algorithm for resource-time-reuse-path problem with factor less than 2 unless $P = NP$. \square

Now, we turn attention to the minimum-resource problem:

THEOREM 4.4. *The minimum-resource discrete resource-time tradeoff problem that allows resources to be reused only over paths cannot have a polynomial-time approximation algorithm with approximation factor less than $3/2$ unless $P = NP$.*

PROOF. (Sketch) The proof uses a reduction from 1-in-3SAT; the construction is similar to that in the proof of Theorem 4.1, but has several key differences that make it considerably more intricate.

First, for each variable x_i we have a gadget similar to before (Figure 8(a)), with the option to send one unit of resource on one of two two-edge paths via a vertex, with the choice of which path indicating whether the variable is set to true or to false. Unlike the previous construction, we chain the variable gadgets together into a path of gadgets, from a source s to a sink t . Refer to Figure 10. A single unit of resource will be moved along the path, using one of each pair of two-edge paths, according to the truth assignments of the variables. A single directed edge, with options $\langle 1, 0 \rangle$ and $\langle 0, M \rangle$, links variable x_i gadget to variable x_{i+1} gadget. Node s is connected to the variable x_1 gadget with an edge with $\langle 0, 0 \rangle$. A property of this construction is that the entry node of the x_i gadget is reached by the unit of resource at exactly time $i - 1$, and the exit node of this gadget is reached at time exactly i . At time n the one unit of resource that traverses the path of variable gadgets emerges at time n . Finally, there is also an edge directly from s to t with options $\langle 1, n \rangle$ and $\langle 0, M \rangle$. In total, two units of resource will be moved through this part of the DAG: one will follow a path through the variable gadgets, according to the truth assignments of the variables, and the other will go directly along the edge (s, t) . Both units of resource will arrive at t at time n .

The clause gadget consists of three vertices, each representing a literal. Each clause has an entry vertex and an exit vertex, and they are chained into a path of gadgets, with clauses ordered in a specific way, as described below. Refer to Figure 11. The exit vertex of one clause has an edge connecting it to the next clause in the order; these edges have specially chosen duration values in order to serve as “buffers”, as described below. The variable portion of the DAG feeds into the path of clause gadgets, with the 2 units of resource that arrive at t at time n moving along an edge that feeds into the first of the sequence of clause gadgets. Each of the three vertices of a clause gadget corresponds to a literal; each has an input edge coming from one of the two vertices of the variable gadget corresponding to the literal, according to whether the variable appears positively or negatively in the clause. These incoming edges have durations that are carefully chosen, so that the timing is as follows: For a clause with variables x_i , x_j , and x_k , the two units of resource (which came through the variable portion of the DAG before entering the path of clause gadgets) will arrive at the entry to the clause at exactly time $n + i + j + k$. The incoming edges from variables to the clause literals have durations chosen just so that the precedence constraints are satisfied “just in time”, for the two units of resource to pass through the clause gadget literals that are *not* true (using edges with duration 0, based on the resource of 1), while the one true literal vertex (who was reached within the clause gadget via an edge of duration 1, instead of 0, since there was no resource associated with it) is reached 1 unit of time sooner (from the variable gadget), to compensate. The net result is that both units of resource emerge out of a clause at time $n + 1 + i + j + k$, ready to pass into the buffer and the next clause gadget. The buffers are selected carefully.

Then, we claim that we can achieve makespan A using just the 2 units of resource if and only if the variables are assigned to satisfy the 1-in-3SAT. If the variables are assigned in a way that does not yield all clauses to be true, then we will need at least 3 units of resource to achieve the target makespan. Thus, it is NP-hard to distinguish between needing 2 units and needing 3 units of resource. This implies that it is NP-hard to achieve an approximation ratio better than factor $3/2$. \square

4.2 Reuse Over a Path with Recursive Binary Splitting and k -Way Splitting

We have seen a (strong) NP-hardness proof (Theorem 4.1) for the discrete resource-time tradeoff problem with general non-increasing duration functions. In this subsection we strengthen this result by showing that the problem remains hard even when the duration functions arise from recursive binary split reducers and k -way split reducers. The proof uses the same general technique as in Section 4.1, but requires more complex gadgets to deal with the restricted duration functions.

Composite node. A composite node v of order k is a gadget of $(k + 2)$ nodes as shown in Figure 12. A composite node can have only one incoming edge and only one outgoing edge. Without using any extra resource, a composite node of order k takes $(k + 2)$ units of time to finish its activities. This is because there is one write operation on vertex v_1 , one write operation on vertex v_i ($2 \leq i \leq k + 1$) and k write operations on vertex v_{k+2} . Using 2 units

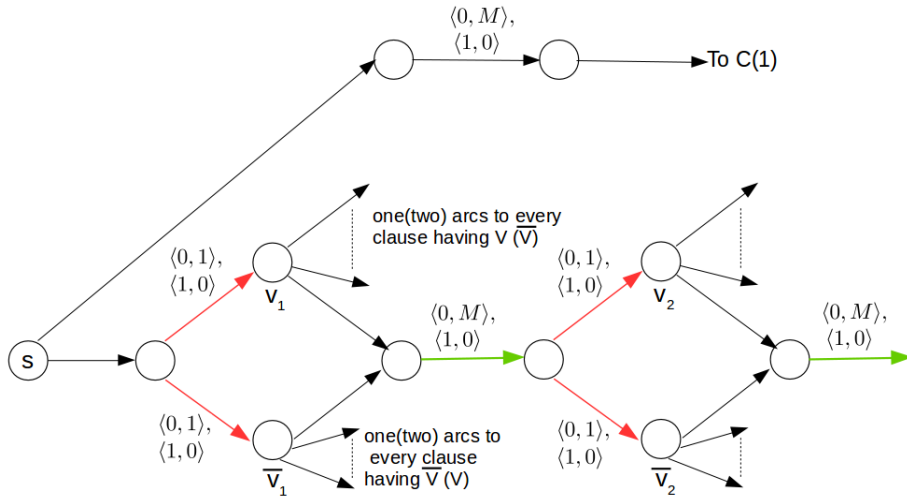


Figure 10: The variable gadgets chained together for the hardness of approximation of the minimum-resource problem (Theorem 4.4).

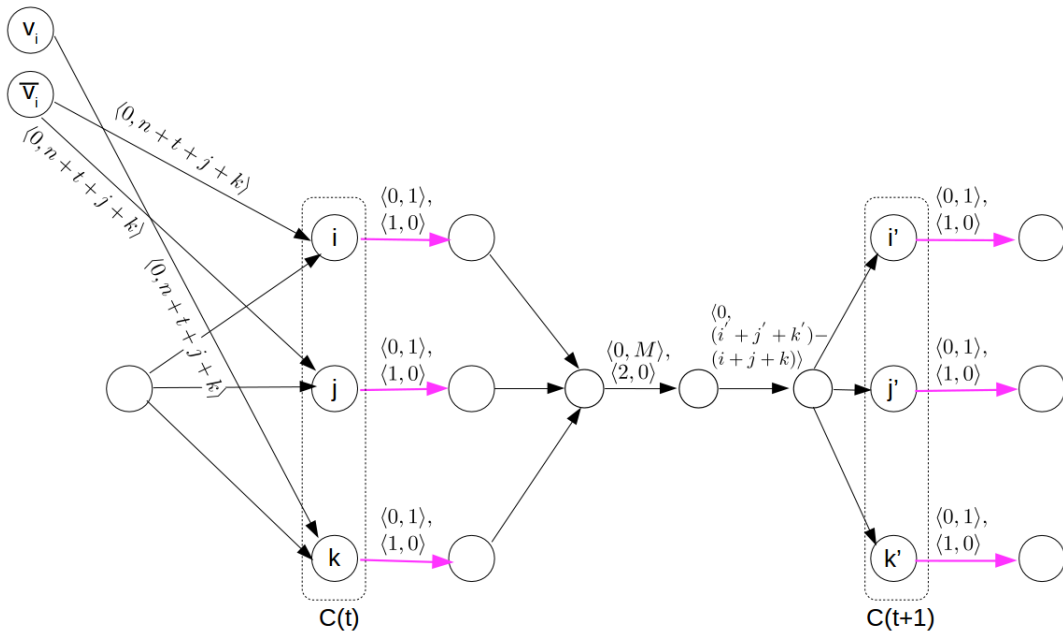


Figure 11: The clause gadgets chained together for the hardness of approximation of minimum-resource problem.

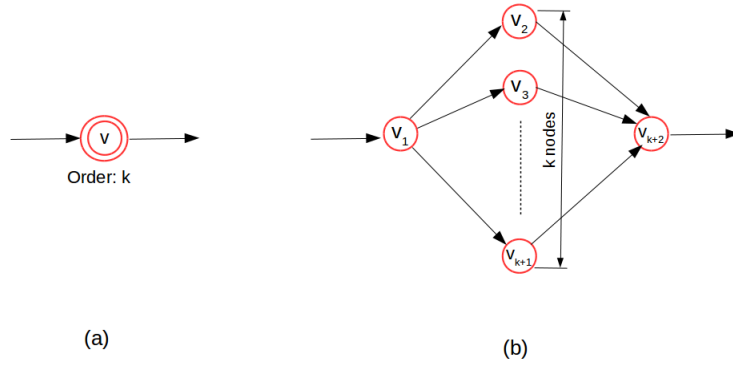


Figure 12: Composite node (Section 4.2).

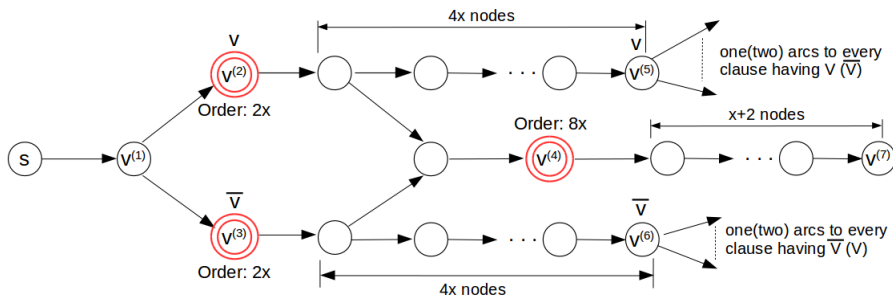


Figure 13: Gadget for variable V (Section 4.2).

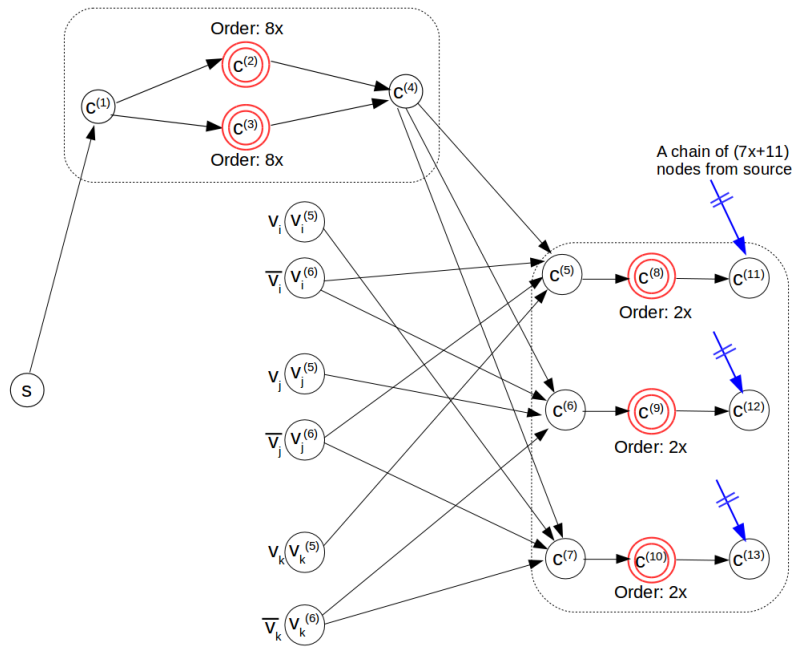


Figure 14: Gadget for clause $C = (V_i \vee V_j \vee V_k)$ (Section 4.2).

of resource with the k -way splitting function, all activities can be completed in $(2 + k/2 + 2) = (k/2 + 4)$ time. Similarly using 2 units of resource with recursive binary splitting function, all activities will be completed in $(2 + k/2 + \log 2 + 1) = (k/2 + 4)$ time. Thus using 2 units of resource, composite node v takes $(k/2 + 4)$ units of time using either function.

Variable gadget. The gadget for variable V consists of 3 composite nodes and other nodes as shown in Figure 13. Composite nodes $V^{(2)}$ and $V^{(3)}$ are of order $2x$. Composite node $V^{(4)}$ is of order $8x$. There is a chain of $4x$ nodes from $V^{(2)}$ to $V^{(5)}$ inclusive. Similarly there is a chain of $4x$ nodes from $V^{(3)}$ to $V^{(6)}$ inclusive. We will see that unless a variable gadget gets exactly 2 units of resource, its makespan will be greater than $(7x + 2y + 12)$ which we will use as the target makespan later in our hardness proof. The values of x and y will be described shortly. Sending 2 units of resource to node $V^{(2)}$ (Figure 13) corresponds to setting the variable V to TRUE and sending 2 units of resources to $V^{(3)}$ corresponds to setting V to FALSE. We will see that sending one unit of resource to $V^{(2)}$ and one unit of resource to $V^{(3)}$ will make the makespan greater than the target makespan.

Clause gadget. The gadget corresponding to clause C has 13 vertices $C^{(i)}$ ($1 \leq i \leq 13$) as shown in Figure 14. Vertices $C^{(2)}$ and $C^{(3)}$ are composite nodes each of order $8x$. If clause C has three literals V_i, V_j and V_k , then vertex $C^{(5)}$ is connected to the vertices $V_i^{(6)}, V_j^{(6)}$ and $V_k^{(5)}$. These vertices correspond to $\neg V_i, \neg V_j$ and V_k respectively. Vertex $C^{(6)}$ is connected to $V_i^{(6)}, V_j^{(5)}$ and $V_k^{(6)}$. These vertices correspond to $\neg V_i, V_j$ and $\neg V_k$. Vertex $C^{(7)}$ is connected to $V_i^{(5)}, V_j^{(6)}$ and $V_k^{(6)}$. These vertices correspond to $V_i, \neg V_j$ and $\neg V_k$. There are 3 composite nodes $C^{(8)}, C^{(9)}$ and $C^{(10)}$ each of order $2x$. There is a chain of $7x + 11$ vertices from s to each vertex in $\{C^{(11)}, C^{(12)}, C^{(13)}\}$. We define the “earliest finish time” of a node v as the time when all the write operations at v are finished.

In a valid assignment of 1-in-3SAT, we show that for each clause C , exactly 2 vertices of $C^{(5)}, C^{(6)}$ and $C^{(7)}$ will have earliest finish time of $(6x + 5)$ and the other one will have earliest finish time of $(5x + 8)$. (Table 3)

Value of x . There is only one vertex ($V^{(7)}$) with out-degree zero in every variable gadget V . Also, in every clause gadget C , there are three vertices $C^{(11)}, C^{(12)}$ and $C^{(13)}$, each with zero out-degree. So, if we connect all such vertices to the sink vertex t , then in-degree at t will be $(n + 3m)$. Let k be the smallest power of 2 such that $k \geq (n + 3m)$. We perform a recursive binary splitting at vertex t . Let y be the height of the binary splitting at t where $y = \log k$. To make $8x > (7x + 2y + 12)$, we define $x = \max((2y + 13), 8)$. Hence, the path from any vertex from $\{V^{(7)}, C^{(11)}, C^{(12)}, C^{(13)}\}$ to sink t will take time $2y$.

Truth value assignment. Setting variable V to TRUE implies sending 2 units of resource through composite vertex $V^{(2)}$. The corresponding earliest finish time at vertex $V^{(5)}$ is $1 + (x + 4) + 4x = 5x + 5$ and at vertex $V^{(6)}$ is $1 + (2x + 2) + 4x = 6x + 3$. Similarly, setting variable V to FALSE implies sending 2 units of resource through

vertex $V^{(3)}$. The corresponding earliest finish time at vertex $V^{(5)}$ is $1 + (2x + 2) + 4x = 6x + 3$ and at vertex $V^{(6)}$ is $1 + (x + 4) + 4x = 5x + 5$.

LEMMA 4.5. *There exists a solution to the input instance of 1-in-3SAT iff there exists a valid flow of resource through the reduced DAG achieving a makespan of at most $7x + 2y + 12$ using at most $2n + 4m$ units of resource.*

PROOF. Forward direction. We now prove that if there is a solution to the 1-in-3SAT instance with n variables and m clauses, then the reduced DAG has a makespan of $7x + 2y + 12$ with $2n + 4m$ units of resource.

If a variable V is set to TRUE, then we allow 2 units of resource to flow through vertex $V^{(2)}$ along the path $\langle S, V^{(1)}, V^{(2)}, V^{(4)} \rangle$, otherwise, we allow 2 units of resource to flow through vertex $V^{(3)}$ along the path $\langle S, V^{(1)}, V^{(3)}, V^{(4)} \rangle$. Assigning TRUE to variable V implies that the earliest finish times at vertex $V^{(5)}$ and $V^{(6)}$ are $5x + 5$ and $6x + 3$, respectively. Also, the earliest finish time at vertex $V^{(7)}$ is $1 + (2 + 2x) + 1 + 2 + (4x + 4) + x + 2 = 7x + 12$. In Figure 14, there are 3 writers from variable gadgets that write on each of the nodes in $\{C^{(5)}, C^{(6)}, C^{(7)}\}$. If there are multiple writers ready to write to the same vertex at the same time, we serialize the write operations. For example, if $V_i = TRUE, C_j = FALSE$ and $V_k = FALSE$, then the writer from variable gadget V_i is ready to write at time $5x + 5$. The writers from V_j and V_k are ready to write at time $6x + 3$. Hence, all three write operations can be completed at time $\max\{5x + 6, 6x + 4, 6x + 5\} = 6x + 5$. From Table 3, it is evident that in clause C , if only one literal is TRUE and the other two are FALSE, then among $C^{(5)}, C^{(6)}$ and $C^{(7)}$ only one vertex has an earliest finish time of $5x + 8$ and the other two have $6x + 5$. The vertex with starting time $5x + 8$, can finish the activity corresponding to composite node (one of $C^{(8)}, C^{(9)}$ and $C^{(10)}$) of order $2x$, in another $2x + 2$ units of time without using any resource. Hence, it will finish at time $5x + 8 + 2x + 2 = 7x + 10$. Each of the other two vertices with earliest finish time of $6x + 5$ takes 2 units of resource flowing from vertex $C^{(4)}$ and finishes the composite node’s activity at time $(6x + 5) + (x + 4) = 7x + 9$. There is a chain of $7x + 11$ nodes from the source vertex to each of the vertices in $\{C^{(11)}, C^{(12)}, C^{(13)}\}$. Thus, the earliest finish time at each of those three vertices is $7x + 12$. Together, with $2y$ units of time to sink vertex t , the total makespan is $7x + 2y + 12$.

Backward direction. To achieve a makespan of $7x + 2y + 12$, every variable gadget requires 2 units of resource and each clause gadget requires 4, otherwise the makespan will be $8x$ which is larger than $7x + 2y + 12$ because $x > 2y + 12$. Also, any resource used in a variable gadget cannot be used further in any other variable or clause gadget because the resource can be reused over a path only. Similarly, any resource used in any clause gadget cannot be reused in any other gadget. Only one vertex that is either $V^{(5)}$ or $V^{(6)}$, will have the earliest finish time of $5x + 5$. Both cannot be $5x + 5$, as there is only 2 units of resource per variable gadget. Both cannot be $6x + 3$ as in a clause C where the literal V or $\neg V$ is present, there is an edge from either $V^{(5)}$ or $V^{(6)}$ to each of $C^{(5)}, C^{(6)}$ and $C^{(7)}$. This requires clause gadget C to get 6 units of resource to achieve a makespan $\leq 7x + 2y + 12$. But each clause gadget can have exactly 4 units of resource. Thus, for every variable V , for it to be a valid

assignment, V is set to either TRUE or FALSE. From Table 3, if a clause has exactly one TRUE literal, then one of the vertices from $C^{(5)}, C^{(6)}$ and $C^{(7)}$ has the earliest finish time of $5x + 8$ and the other two have $6x + 5$. This requires to have 4 units of resource to achieve the earliest finish time $\leq 7x + 10$ at each of the vertices from $\{C^{(8)}, C^{(9)}, C^{(10)}\}$. This can be achieved by assigning 2 units of resource to those two composite nodes (from $C^{(8)}, C^{(9)}$ and $C^{(10)}$) that start executing at time $6x + 5$. The composite node that can start at time $5x + 8$ does not use any extra resource. If the clause does not have exactly one TRUE literal, then the clause gadget would require 6 units of resource to achieve the target makespan. However, we just argued that each clause gadget can have exactly 4 units of resource. Thus, each clause has exactly one TRUE literal and the 1-in-3SAT instance is also satisfied. \square

V_i	V_j	V_k	$C^{(5)}$	$C^{(6)}$	$C^{(7)}$
T	T	T	$\max(a, a+1, b) = a+1$	$\max(a, b, a+1) = a+1$	$\max(b, a, a+1) = a+1$
F	T	T	$\max(b, a, b+1) = a$	$\max(b, b+1, a) = a$	$\max(a, a+1, a+2) = a+2$
T	F	T	$\max(a, b, b+1) = a$	$\max(a, a+1, a+2) = a+2$	$\max(b, b+1, a) = a$
T	T	F	$\max(a, a+1, a+2) = a+2$	$\max(a, b, b+1) = a$	$\max(b, a, b+1) = a$
F	F	T	$\max(b, b+1, b+2) = b+2$	$\max(b, a, a+1) = a+1$	$\max(a, b, a+1) = a+1$
F	T	F	$\max(b, a, a+1) = a+1$	$\max(b, b+1, b+2) = b+2$	$\max(a, a+1, b) = a+1$
T	F	F	$\max(a, b, a+1) = a+1$	$\max(a, a+1, b) = a+1$	$\max(b, b+1, b+2) = b+2$
F	F	F	$\max(b, b+1, a) = a$	$\max(b, a, b+1) = a$	$\max(a, b, b+1) = a$

Table 3: Earliest start time at vertices $C^{(5)}, C^{(6)}$ and $C^{(7)}$ for different assignment of truth values of variable V_i, V_j and V_k in Figure 14, where $a = (6x + 4)$ and $b = (5x + 6)$.

4.3 Underlying Bounded Treewidth Graph

Let $G(D)$ be the undirected graph obtained by ignoring the directedness of the edges of a given DAG D . In the case that $G(D)$ is a graph of bounded treewidth,² we show that the offline minimum-makespan and minimum-resource problems on D are (weakly) NP-hard. (Note that Theorem 4.1 proving the strong NP-hardness of the problems does not assume that the underlying undirected graph is of bounded treewidth.)

THEOREM 4.6. *It is weakly NP-hard to decide if there exists a solution to the (offline) discrete resource-time tradeoff problem, with resource reuse over paths and a non-increasing duration function, satisfying a resource bound B and a makespan bound T , provided the undirected graph obtained by ignoring the directedness of the edges of the input DAG is of bounded treewidth.*

The proof of this theorem is based on a reduction from PARTITION[15]. The construction is shown in Figure 15. The input instance is a set $S = \{s_1, s_2, \dots, s_n\}$ of n positive integers; let $B = \sum_{i=1}^n s_i$. The PARTITION problem asks if there is a partition of S into subsets S_1 and S_2 such that the sums of the values in the two subsets are the same (i.e., exactly $B/2$). In this construction we have a total of B resources to allocate in our program. The value M is chosen to be greater than $B/2$, the target makespan, ensuring that memory resources must be allocated to these nodes. This ensures that at least s_i units of resource pass through each $v_i^{(1)}$, constructing our

²Recall that a tree decomposition of a graph $G = (V, E)$ is a tree T with nodes $X_1, X_2, \dots, X_n, X_i \subseteq V$, satisfying: (1) $\bigcup_i X_i = V$; (2) For edge $(u, v) \in E$ there exists a X_i with $u, v \in X_i$; (3) For any two nodes, X_i and X_j , in T , if node X_k is in the (unique) path between X_i and X_j in T , then $X_i \cap X_j \subseteq X_k$. The *width* of the tree decomposition is $\max_i |X_i| - 1$, and the *treewidth* of G is the minimum width over all tree decompositions of G .

numbers. From each $v_i^{(1)}$ there are two choices of nodes, $v_i^{(2)}$ and $v_i^{(3)}$, to pass the resources onto each of which will either utilize s_i resources or increase the makespan on that path by s_i . The pair also funnel the resources into a sink vertex \bar{v}_0 with a potential makespan cost of M which ensures that their resources cannot be passed along to nodes $v_j^{(2)}$ and $v_j^{(3)}$ to the right (i.e., $j > i$). Thus the top and bottom paths represent our two sets and for each v_i we must allocate s_i makespan to either the top or the bottom path. Thus a total makespan of $B/2$ can only be achieved iff there is a partition of the s_i 's into two sets such that each set sums to $B/2$.

To see that the constructed graph has bounded treewidth, let $V_i = \{v_i^{(j)}\}$, where $1 \leq j \leq 7$. Vertices $v_i^{(j)}$ for $1 \leq i \leq n$ are connected to the sink vertex \bar{v}_0 . Then G has a tree decomposition T with nodes $S_i, 1 \leq i \leq n$, as shown in Figure 16, with S_i defined as follows: $S_1 = \{v_0, \bar{v}_0\} \cup V_1$; $S_i = \{v_0, \bar{v}_0\} \cup V_{i-1} \cup V_i$, for $2 \leq i \leq n$. We claim that T is a valid tree decomposition. It is evident that $\bigcup_{1 \leq i \leq n} S_i = V$. From the construction of S_j ($1 \leq j \leq n$), it is clear that, for each edge (u, v) of the graph G , there exists a node S_j with $u, v \in S_j$. For any S_i and S_j , with $j > i + 1$ and $1 \leq i \leq (n - 2)$, we have $S_i \cap S_j = \{v_0, \bar{v}_0\}$, and, for any node S_k ($i < k < j$), on the path between S_i and S_j , we have $v_0 \in S_k$ and $\bar{v}_0 \in S_k$, so that $S_i \cap S_j \subseteq S_k$. Thus, T is a valid tree decomposition, and it has width 15 ($\max_i |S_i| - 1 = 15$), so the treewidth of G is at most 15.

5 CONCLUSION

In this paper we introduce the discrete resource-time tradeoff problem with resource reuse in which each unit of resource is routed along a source to sink path and is possibly used and reused to expedite activities encountered along that path. We consider two different objective functions: (1) optimize makespan given a limited resource budget and (2) optimize resource requirement given a target makespan.

Our original motivation came from a desire to mitigate the cost of data races in shared-memory parallel programs by using extra space to reduce the time it takes to perform conflict-free write operations to shared memory locations. We consider three duration functions: general non-increasing function for the general resource-time question, and recursive binary reduction and multiway (k -way) splitting for the space-time case.

We present the first hardness and approximation hardness results as well as the first approximation algorithms for our problems. We show that the makespan optimization problem is strongly NP-hard under all three duration functions. When the duration function is general non-increasing we also show that it is strongly NP-hard to achieve an approximation ratio less than 2 for the makespan optimization problem and less than $\frac{3}{2}$ for the resource optimization problem. We give a $(\frac{1}{\alpha}, \frac{1}{1-\alpha})$ bi-criteria (resource, makespan) approximation algorithm for that same duration function, where $0 < \alpha < 1$. We present improved approximation ratios for the recursive binary reduction function and the multiway (k -way) splitting functions.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants CCF-1439084, CCF-1526406, CNS-1553510, IIS-1546113 and US-Israel Binational Science Foundation grant number 2016116.

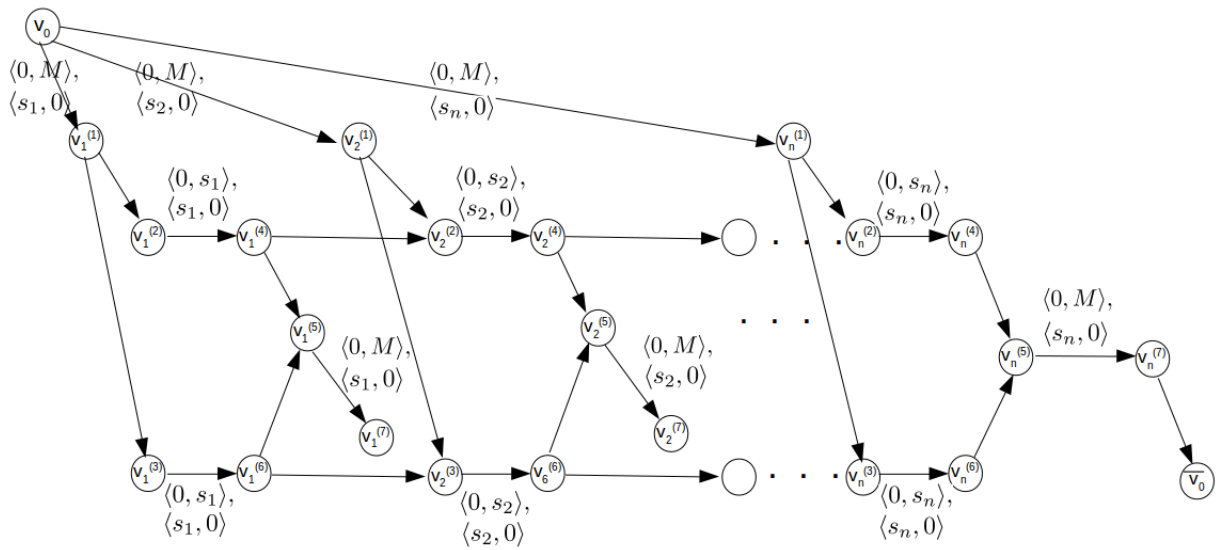


Figure 15: Construction for (weak) NP-hardness proof for graphs with bounded treewidth (Section 4.3).

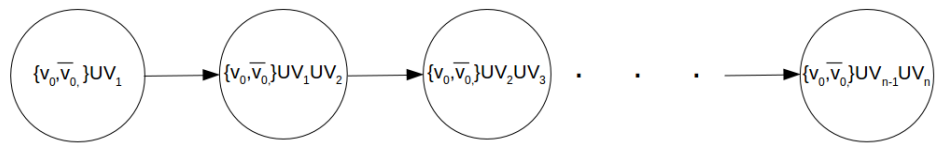


Figure 16: Tree decomposition of graph G (Section 4.3).

REFERENCES

- [1] Google gperftools. Fast, multi-threaded malloc() and nifty performance analysis tools. <http://code.google.com/p/gperftools/>.
- [2] lmalloc. Lockless memory allocator. <http://locklessinc.com/>.
- [3] Martin Aigner, Christoph M Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 451–469.
- [4] Can Akkan, Andreas Drexl, and Alf Kimms. 2005. Network decomposition-based benchmark results for the discrete time–cost tradeoff problem. *European Journal of Operational Research* 165, 2 (2005), 339–358.
- [5] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 117–128.
- [6] Eric Blayo, Laurent Debreu, Gregory Mounie, and Denis Trystram. 1999. Dynamic load balancing for ocean circulation model with adaptive meshing. In *European Conference on Parallel Processing*. Springer, 303–312.
- [7] OpenMP Architecture Review Board. 1997. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. *White Paper* (1997). url: <http://www.openmp.org/specs/mp-documents/paper/paper.ps>.
- [8] Prabuddha De, E James Dunne, Jay B Ghosh, and Charles E Wells. 1995. The discrete time-cost tradeoff problem revisited. *European Journal of Operational Research* 81, 2 (1995), 225–238.
- [9] Prabuddha De, E James Dunne, Jay B Ghosh, and Charles E Wells. 1997. Complexity of the discrete time-cost tradeoff problem for project networks. *Operations research* 45, 2 (1997), 302–306.
- [10] Jianzhong Du and Joseph Y-T Leung. 1989. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics* 2, 4 (1989), 473–487.
- [11] Pierre-François Dutot, Grégory Mounié, and Denis Trystram. 2004. Scheduling parallel tasks: Approximation algorithms.
- [12] Mingdong Feng and Charles E Leiserson. 1999. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.
- [13] Matteo Frigo, Pablo Halpern, Charles E Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *Proceedings of the twenty-first annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 79–90.
- [14] Delbert R Fulkerson. 1961. A network flow computation for project cost curves. *Management science* 7, 2 (1961), 167–178.
- [15] Michael R Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.
- [16] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative relaxation of concurrent data structures. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 317–328.
- [17] Klaus Jansen and Hu Zhang. 2006. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Transactions on Algorithms (TALG)* 2, 3 (2006), 416–434.
- [18] James E Kelley Jr. 1961. Critical-path planning and scheduling: Mathematical basis. *Operations research* 9, 3 (1961), 296–320.
- [19] James E Kelley Jr and Morgan R Walker. 1959. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. ACM, 160–173.
- [20] Jan Karel Lenstra and AHG Rinnooy Kan. 1978. Complexity of scheduling under precedence constraints. *Operations Research* 26, 1 (1978), 22–35.
- [21] Renaud Lepère, Grégory Mounié, and Denis Trystram. 2002. An approximation algorithm for scheduling trees of malleable tasks. *European Journal of Operational Research* 142, 2 (2002), 242–249.
- [22] Renaud Lepere, Denis Trystram, and Gerhard J Woeginger. 2002. Approximation algorithms for scheduling malleable tasks under precedence constraints. *International Journal of Foundations of Computer Science* 13, 04 (2002), 613–627.
- [23] Rolf H Möhring. 1989. Computationally tractable classes of ordered sets. In *Algorithms and order*. Springer, 105–193.
- [24] Robert HB Netzer and Barton P Miller. 1992. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 1 (1992), 74–88.
- [25] D Panagiotakopoulos. 1977. A CPM time-cost computational algorithm for arbitrary activity cost functions. *INFOR: Information Systems and Operational Research* 15, 2 (1977), 183–195.
- [26] Steve Phillips Jr and Mohamed I Dessouky. 1977. Solving the project time/cost tradeoff problem using the minimal cut concept. *Management Science* 24, 4 (1977), 393–400.
- [27] James Reinders. 2007. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. O’Reilly Media, Inc.
- [28] Don R Robinson. 1975. A dynamic programming solution to cost-time tradeoff for CPM. *Management Science* 22, 2 (1975), 158–166.
- [29] Thomas J Schaefer. 1978. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*. ACM, 216–226.
- [30] Scott Schneider, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. 2006. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*. ACM, 84–94.
- [31] Nir Shavit. 2011. Data structures in the multicore age. *Commun. ACM* 54, 3 (2011), 76–84.
- [32] Martin Skutella. 1998. Approximation algorithms for the discrete time-cost tradeoff problem. *Mathematics of Operations Research* 23, 4 (1998), 909–929.
- [33] John Turek, Joel L Wolf, and Philip S Yu. 1992. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the fourth annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 323–332.

APPENDIX

A ALTERNATE HARDNESS PROOF FROM NUMERICAL 3D MATCHING

We give a polynomial-time reduction from the numerical 3-dimensional matching problem to the discrete resource-time tradeoff problem (with resource reuse over paths and a non-increasing duration function).

Numerical 3-dimensional matching problem: Given $A = \{a_1, a_2, \dots, a_n\}$, $B = \{b_1, b_2, \dots, b_n\}$, and $C = \{c_1, c_2, \dots, c_n\}$, partition $A \cup B \cup C$ into n triples $S_i \in A \times B \times C$ of equal sum $T = (\sum A + \sum B + \sum C)/n$.

Given an instance of the numerical 3D matching problem, we create a DAG D with source s and sink t as shown in Figure 18. For each $a_i \in A$, there is an edge (s, a_i) in D . The space-time tradeoff function at edge (s, a_i) is $\{(0, \infty), \langle n, a_i \rangle\}$. Recall that, this means that with zero resource, it takes infinite time to finish the activity (s, a_i) and with n units of resource it finishes in time a_i . We create a gadget that has n incoming edges and n outgoing edges. We call the gadget a *bipartite matcher* (Figure 17) as it matches (a 1 : 1 mapping) the incoming edges to the outgoing edges. We describe the bipartite matcher in the next paragraph. For each $b_i \in B$, there is an edge (b_i, b'_i) in D . The tradeoff function at edge (b_i, b'_i) is $\{(0, \infty), \langle n, b_i \rangle\}$. We put all the n edges (b_i, b'_i) to a bipartite matcher as its incoming edges. For each $c_i \in C$, there is an edge (c_i, t) in D . The tradeoff function at edge (c_i, t) is $\{(0, \infty), \langle n, c_i \rangle\}$.

The bipartite matcher gadget. The gadget has n incoming edges at vertices $\{x_1, x_2, \dots, x_n\}$ and n outgoing edges from $\{z_1, z_2, \dots, z_n\}$. It maps the vertices from $\{x_1, x_2, \dots, x_n\}$ to those in $\{z_1, z_2, \dots, z_n\}$. The mapping is one to one. This works as follows. There are n units of incoming resource at each vertices x_i . Every outgoing edge (x_i, y_i^j) from x_i ($1 \leq j \leq n$) has a tradeoff function $\{(0, \infty), \langle 1, 0 \rangle\}$. Hence, each of the outgoing edges (x_i, y_i^j) from x_i gets one unit of resource. The tradeoff function at edge (y_i, z_i) is $\{(0, \infty), \langle 1, 0 \rangle\}$ which forces y_i^j to send one unit of resource to y_i . The tradeoff function at edge (y_i^j, z'_j) is $\{(0, M), \langle 1, 0 \rangle\}$. Thus, if y_i^j sends one unit of resource to y_i , it cannot send any resource to z'_j forcing the activity (y_i^j, z'_j) to take M units of time to finish. Here, $M > \max_{1 \leq i \leq n}(a_i) + \max_{1 \leq i \leq n}(b_i) + \max_{1 \leq i \leq n}(c_i)$. The tradeoff function at edge (z'_j, z_j) is $\{(0, \infty), \langle n-1, 0 \rangle\}$. There are n incoming edges (y_i^j, z'_j) to z'_j . Out of these n incoming edges, $(n-1)$ edges flow $n-1$ units of resource to z'_j which are then used for the activity at (z'_j, z_j) .

We now show the mapping through an example. Suppose x_1 is mapped to z_3 . Then the corresponding flow is as follows: one unit of resource flows from y_1^3 to y_1 . As the total incoming flow of resource at vertex y_1^3 is one, no resource flows from y_1^3 to z'_3 . However, one unit of resource flows from each y_i^3 except y_1^3 to z'_3 . The earliest start time (*EST*) along path $\langle x_1, y_1^3, z'_3 \rangle$ is $EST(x_1) + M$ while that along path $\langle x_i, y_i^3, z'_3 \rangle$ for $i \neq 1$ is $EST(x_i)$. This makes the earliest start time at z'_3 , $EST(z'_3) = \max\{EST(x_1) + M, EST(x_i)\} = EST(x_1) + M$. This holds true because $M > \max_{1 \leq i \leq n}(a_i) + \max_{1 \leq i \leq n}(b_i) +$

$\max_{1 \leq i \leq n}(c_i)$. Also, $n-1$ units of resource flow to z'_3 and they are used for the activity (z'_3, z_3) to finish in time 0. Observe that no y_1^i except y_1^3 can send resource to y_1 . The gadget has a total resource-inflow of n^2 . Each of (z'_i, z_i) requires $n-1$ units of resource that sums up to $n^2 - n$ units of resource. Each of (y_i, z_i) requires one unit of resource, that sum up to n units of resource. If two of y_1^i sends a unit of resource each to y_1 , then the total resource left to be used by all (z'_i, z_i) is at most $n^2 - n - 1$. Thus at least one of (z'_i, z_i) won't get $n-1$ units of resource and will take infinite time. Hence, mapping x_i to y_j corresponds to flowing one unit of resource from y_j^i to y_i and vice-versa; this makes a one-to-one mapping from $\{x_1, x_2, \dots, x_n\}$ to $\{z_1, z_2, \dots, z_n\}$.

LEMMA A.1. *There exists a solution to a input instance of numerical 3D matching if and only if there exists a valid flow of resource in the DAG such that the makespan is $2M + T$ with resource bound $B = n^2$.*

PROOF. If there is a solution in the input instance of numerical 3D matching, then there are n sets, each of type $\{a_i, b_j, c_k\}$ such that $a_i + b_j + c_k = T$. We use first bipartite matcher gadgets to map a_i to b_j and the second bipartite matcher to map b'_j to c_k . Each bipartite matcher contributes M in the makespan. $(s, a_i), (b_j, b'_j)$ and (c_k, t) adds T to the makespan. Thus the makespan is exactly $2M + T$.

If the reduced DAG admits a makespan of $2M + T$ using n^2 units of resource, then there is also a solution to the input instance of numerical 3D matching. From the construction of bipartite 3D matching, there is a one-to-one mapping from a_i to b_j and from b'_j to c_k . As the makespan is $2M + T$ and each bipartite matcher contributes M to the makespan, this gives a solution to numerical 3D matching. \square

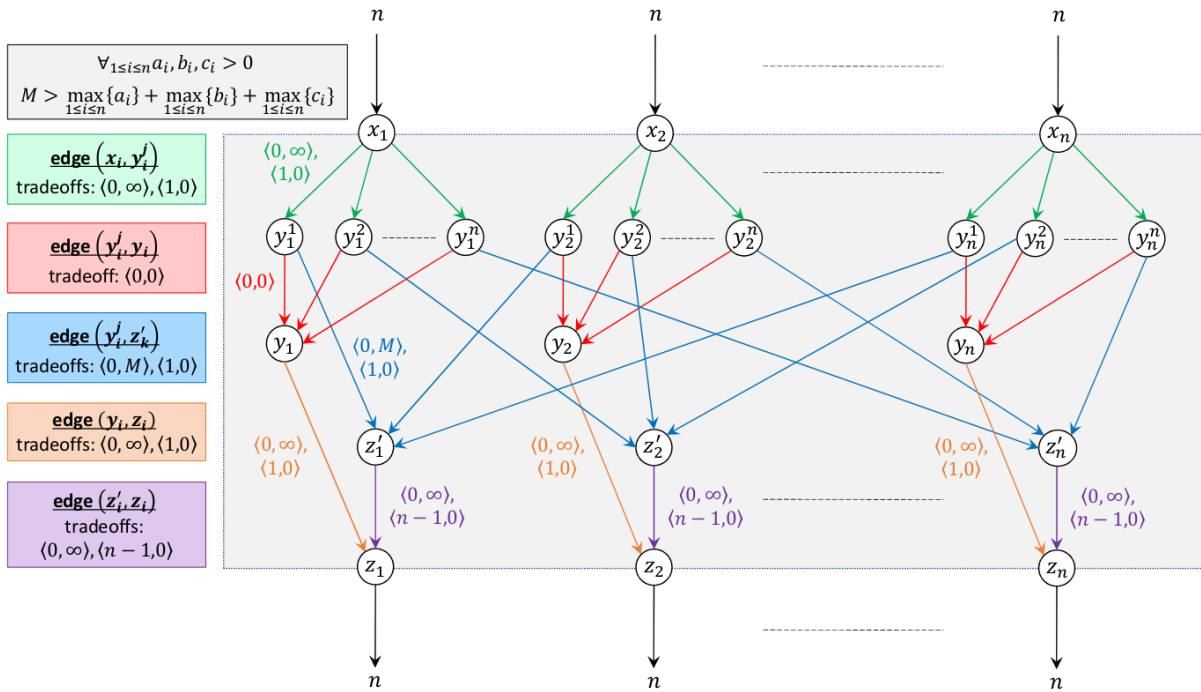


Figure 17: Bipartite matcher gadget (Section A).

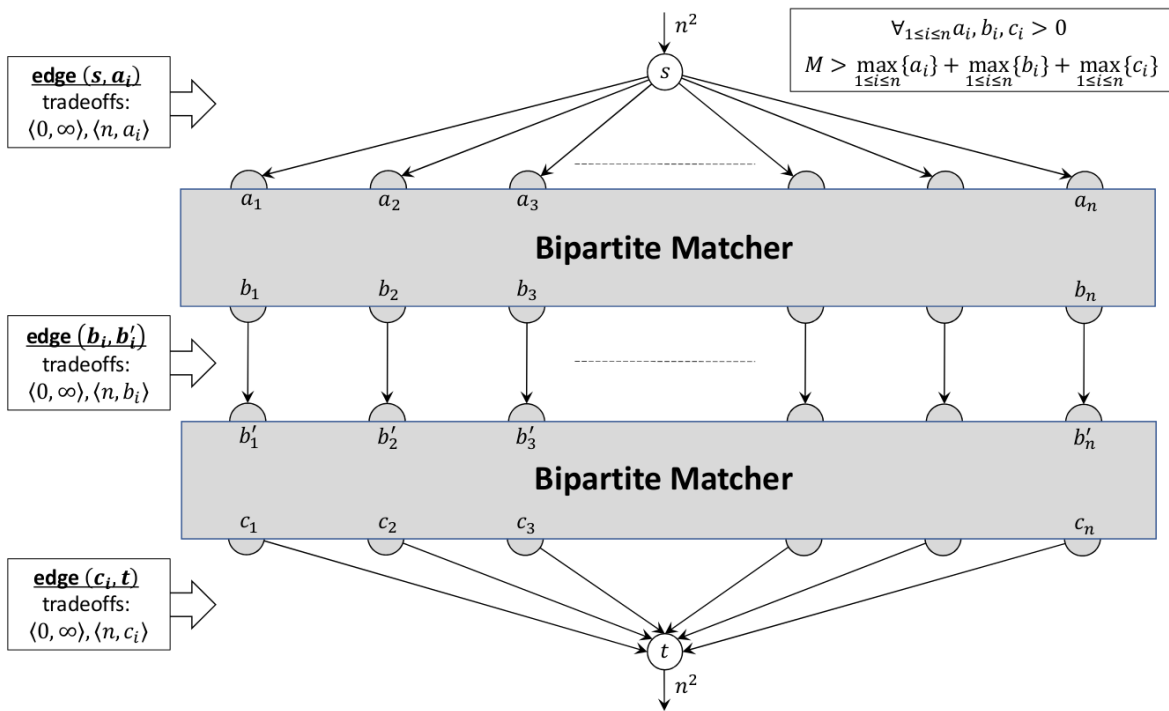


Figure 18: Reduced DAG from a numerical 3D matching instance (Section A).