Constraint Propagation on GPU: A Case Study for the Cumulative Constraint

Fabio Tardivo[1][00000−0003−3328−2174], Agostino Dovier[2][0000−0003−2052−8593], Andrea Formisano[2][0000−0002−6755−9314], Laurent Michel[3][0000−0001−7230−7130], and Enrico Pontelli[1][0000−0002−7753−1737]

[1] New Mexico State University {ftardivo,epontell}@nmsu.edu
[2] University of Udine, INdAM-GNCS
{agostino.dovier,andrea.formisano}@uniud.it
[3] Synchrony Chair in Cybersecurity, University of Connecticut ldm@uconn.edu

Abstract. The *Cumulative* constraint is one of the most important global constraints, as it naturally arises in a variety of problems related to scheduling with limited resources. Devising fast propagation algorithms that run at every node of the search tree is critical to enable the resolution of a wide range of applications. Since its introduction, numerous propagation algorithms have been proposed, providing different tradeoffs between computational complexity and filtering power.
Motivated by the impressive computational power that modern GPUs provide, this paper explores the use of GPUs to speed up the propagation of the *Cumulative* constraint. The paper describes the development of a GPU-driven propagation algorithm, motivates the design choices, and provides solutions for several design challenges. The implementation is evaluated in comparison with state-of-the-art constraint solvers on different benchmarks from the literature. The results suggest that GPU-accelerated constraint propagators can be competitive by providing strong filtering in a reasonable amount of time.

Keywords: Constraint Propagation · Cumulative · Parallelism · GPU

1 Introduction

Industrial scheduling problems are derivatives of the so-called "Resource Constrained Project Scheduling Problem" (briefly, RCPSP) in which one must *order* non-preemptible activities of fixed duration to minimize the makespan, i.e., the project duration. Activities use a fixed amount of resources to execute and each resource has a fixed capacity. Unsurprisingly, industrial scheduling readily benefits from any improvements to solve the classic RCPSP problem.

The last three decades witnessed the development of multiple techniques to prune the search tree of such an NP-hard problem [16,3]. The most prominent techniques are Edge-Finding [33,29], Time-Tabling [26], Not-First/Not-Last [33,40], and Energetic-Reasoning [28]. Edge-Finding was developed for cumulative instances through a series of contributions and it deduces precedences between activities that must be satisfied in time $O(n^2k)$ [29] where n is the number of activities and k is the number of distinct capacity requirements of the activities. Time-Tabling focuses on resource usage profile. Several techniques based on *line-sweep* methods were proposed with an $O(n^2)$ [17] solution that separates profile building and inference phases. The core of the inference mechanism rests on the ability to deduce, from the mandatory part of the profile, whether an activity must be postponed or not. Not-First/Not-Last performs an orthogonal pruning with respect to the other approaches by deducing unfeasible precedences between activities in time $O(n^2 \log n)$ [39] using a Θ-tree data structure. Energetic-Reasoning calculates the resource usage in specific time intervals to check and adjust the activities so that there is no over-consumption. Its standard algorithm has $O(n^3)$ time complexity, that can be reduced to $O(n^2 \log n)$ [36] by using Monge matrices.

In practice, most CP solvers employ Edge-Finding or Time-Tabling techniques that exhibit a lower time complexity at each node of the search tree, despite the strength of the filtering one might benefit from with Energetic-Reasoning. This paper revisits this design decision and considers the use of an Energetic-Reasoning propagator in a CP solver. Specifically, the paper advocates that the high computational complexity cost at each fixpoint can be mitigated with the use of a Graphics Processing Unit (GPU) and deliver, overall, faster computation times, or better solutions within a given time horizon. The fundamental assumption is that the energetic filtering rule is easily parallelized on this class of hardware and that the benefits from the derived filtering can be significant.

This paper is organized as follows. Section 2 offers some general background. Section 3 discusses the design of the proposed solution. Section 4 discusses empirical results that pitch a GPU-based Energetic-Reasoning against multiple solvers using Edge-Finding and Time-Tabling techniques. Section 5 concludes the paper.

2 Background

This section establishes the required background knowledge on Constraint Satisfaction and Optimization [2,37], Cumulative Scheduling [6,3] and General-Purpose computing on Graphics Processing Units (GPGPU) [38,10].

2.1 Constraint Satisfaction/Optimization Problem

A *Constraint Satisfaction Problem* (CSP) is a triple $P = \langle V, D, C\rangle$, where $V = \{V_1, \dots, V_n\}$ is a finite set of *variables*, $D = \{D_1, \dots, D_n\}$ is the set of *finite domains*, and C is a collection of *constraints* on variables in V. Each constraint $c \in C$, defined over a set of variables $vars(c) = \{V_{i_1}, \dots, V_{i_m}\} \subseteq V$, defines a relation on $D_{i_1} \times \cdots \times D_{i_m}$, namely $c \subseteq D_{i_1} \times \cdots \times D_{i_m}$. A *solution* of $\langle V, D, C\rangle$ is an assignment $\sigma : V \to \bigcup_{i=1}^{n} D_i$ such that:

- $\sigma(V_i) \in D_i$ for each $i = 1, \dots, n$
- $\forall\, c \in C$ then $\langle \sigma(V_{i_1}), \dots, \sigma(V_{i_m})\rangle \in c$, where $vars(c) = \{V_{i_1}, \dots, V_{i_m}\}$.

The set of solutions for the CSP $\langle V, D, C\rangle$ is denoted $S(\langle V, D, C\rangle)$. Given a CSP $\langle V, D, C\rangle$, a *constraint solver* searches for one or more solutions. A solver alternates two types of steps: (1) constraint propagation and (2) non-deterministic choice. The latter is used to select the next variable to be assigned and to select non-deterministically a value to be given to such a variable (drawn from its current domain). Constraint propagation uses the constraints to prune the domain of the variables, removing values that provably do not belong to a solution.

The propagation algorithm uses a queue to schedule the constraints that must be reconsidered when the domain of a variable changes. Namely, whenever the domain of a variable $x \in vars(c)$ for some $c \in C$ changes, the constraint c is added to the queue. The filtering algorithms of a constraint that shrinks domains enforces a level of consistency, such as *domain consistency* [37]. An m-ary constraint c on the variables $vars(c) = \{V_{i_1}, \dots, V_{i_m}\}$ is *domain consistent* if $\forall\, j \in \{1, \dots, m\}$ the following holds:

$$\forall a_j \in D_{i_j} : \exists a_1 \in D_{i_1} \cdots \exists a_{j-1} \in D_{i_{j-1}} \exists a_{j+1} \in D_{i_{j+1}} \cdots \exists a_m \in D_{i_m} : (a_1, \dots, a_m) \in c$$

A CSP is domain consistent if all constraints in $\mathcal{C}$ are domain consistent. For binary constraints (i.e., $m = 2$) domain consistency is known as *arc consistency.* Without loss of generality, a *Constraint Optimization Problem* is specified with $\langle V, D, C, f\rangle$ where $\langle V, D, C\rangle$ is a CSP and $f : D_1 \times \cdots \times D_n \to \mathbb{R}$ is an objective function to be minimized. The goal is to find a solution of $\langle V, D, C\rangle$

$$\sigma^* = \underset{\sigma \in S(\langle V,D,C\rangle)}{argmin} f(\sigma)$$

that minimizes $f(\sigma(V_1), \cdots, \sigma(V_n))$.

2.2 Cumulative

The *Cumulative* constraint is one of the most used constraints in CP. It makes it easy to model and solve a variety of real-world problems, contributing to the success of CP in scheduling applications.

In detail, it models the Cumulative Scheduling Problem (CuSP): given a set A of activities that use a resource of capacity u and where the goal is to schedule the activities so that the last activity finishes as soon as possible and no more than u units of the resources are used at any time. Formally, each activity $a \in A$ is defined by its start time s_a, its processing time p_a and, its resource usage h_a. The end time of activity a is $e_a = s_a + p_a$ and the problem is defined as follows:

$$\begin{array}{lll} \text{minimize} & \underset{a \in A}{max}(e_a) & \\ \text{subject to} & \displaystyle\sum_{\{a\,:\,a \in A,\, s_a \le t < e_a\}} h_a \le u & t \in \mathbb{N} \end{array}$$

Since its introduction in [1], the *Cumulative* constraint has been the subject of many studies to improve its efficiency. The result is a collection of propagation algorithms with different trade-offs between filtering capability and computational

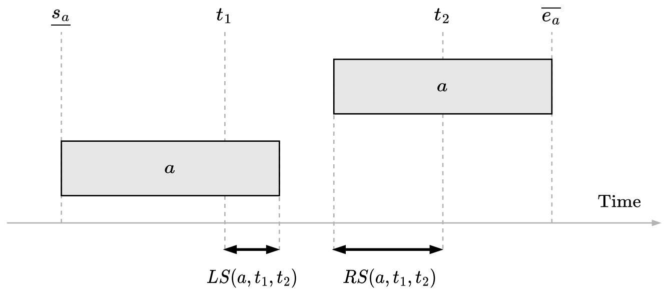

Fig. 1: Left shift and right shift for the activity a with respect to $[t_1, t_2)$.

complexity. Such algorithms are commonly classified by the filtering technique they employ: Time-Tabling, Edge-Finding, Not-First/Not-Last, and Energetic-Reasoning. A complete description of these approaches is out of the scope of this work, interested readers can refer to [6]. This section describes the core ideas that characterize each method and points to the relevant literature for details.

Edge-Finding. This approach considers subsets of activities, determining if an activity must start before or end after the rest. It was introduced in [33], corrected in [29] and improved in different ways in [47,48,24,35,20].

Time-Table. This method consists of computing the minimal resource usage at every time and adjusting the starting time of the activities so that there is no over-consumption of the resource. It first appeared in [26] and was successively refined in [7,27,17].

Not-First/Not-Last. This approach considers subsets of activities and determines whenever an activity cannot be the first/last to be executed. Introduced in [33], it was corrected in [40] and improved in [39,23,22].

Energetic-Reasoning. This method checks some critical time intervals and adjusts the starting time of the activities so that there is no over-consumption of the resource. Introduced in [28], it was refined and improved in [4,13,45,36,46].

Energetic-Reasoning is one of the strongest propagators for the *Cumulative* constraint, dominating both the Time-Table and Edge-Finding approaches [6]. Such filtering examines $O(n^2)$ time intervals for a total complexity of $O(n^3)$, too costly to be used in practice [13,36,46].

Preliminaries Before proceeding to formalize the Energetic-Reasoning, we introduce some notation: $[t_1, t_2)$ denotes an open time interval. The lower and upper bounds of (the domain of) a variable x are denoted by $\underline{x}$ and $\overline{x}$, respectively. Given a time interval $[t_1, t_2)$ and an activity a, their *minimal intersection* is $MI(a, t_1, t_2) = \min(LS(a, t_1, t_2), RS(a, t_1, t_2))$ where LS and RS are the *left shift* and *right shift* (see Figure 1):

$$LS(a, t_1, t_2) = \max(0, \min(\underline{e_a}, t_2) - \max(\underline{s_a}, t_1))$$
$$RS(a, t_1, t_2) = \max(0, \min(\overline{e_a}, t_2) - \max(\overline{s_a}, t_1))$$

```
foreach $[t_1, t_2) \in RI$ do
    $w = \sum_{a \in A} h_a \cdot MI(a, t_1, t_2)$
    if $w < c \cdot (t_2 - t_1)$ then
        foreach $a \in A$ do
            $r = c \cdot (t_2 - t_1) - w + h_a \cdot MI(a, t_1, t_2)$
            if $r < h_a \cdot LS(a, t_1, t_2)$ then
                $\underline{s_a} = \max(\underline{s_a}, t_2 - \frac{r}{h_a})$
            if $r < h_a \cdot RS(a, t_1, t_2)$ then
                $\overline{e_a} = \min(\overline{e_a}, t_1 + \frac{r}{h_a})$
    else
        Fail
```

Algorithm 1: Energetic-Reasoning propagation algorithm.

The Energetic-Reasoning propagator considers the intervals whose extremes are related to the beginning/end of an action [4,13]. We define the set of *relevant intervals* as

$$RI = \bigcup_{(i,j) \in A \times A} O(i, j)$$

where:

$$\begin{aligned} O(i,j) = \{[t_1, t_2) &: t_1 < t_2, t_1 \in O_1(i), t_2 \in O_2(j)\} \cup \\ \{[t_1, t_2) &: t_1 < t_2, t_1 \in O_1(i), t_2 \in O_3(j, O_1(i))\} \cup \\ \{[t_1, t_2) &: t_1 < t_2, t_1 \in O_3(i, O_2(j)), t_2 \in O_2(j)\} \end{aligned}$$

and $O_1(a) = \{\underline{s_a}, \overline{s_a}\}$, $O_2(a) = \{\underline{e_a}, \overline{e_a}\}$, $O_3(a, T) = \{\underline{s_a} + \overline{e_a} - t : t \in T\}$. The consistency condition of is:

$$\forall [t_1, t_2) \in RI : \sum_{a \in A} h_a \cdot MI(a, t_1, t_2) \leq u \cdot (t_2 - t_1)$$

From such condition one adjusts the start time of activities to prevent over-usage as listed in Algorithm 1. Note that it could be worth to consider a set $O(i, j)$ only if s_i or s_j changed in the current propagation phase. The rational of this heuristics is to avoid to check consistent intervals and despite that it makes the filtering weaker, it proved to be effective for the sequential implementation (see Section 4).

2.3 GPUs and CUDA

Modern Graphical Processing Units (GPUs) are massively parallel architectures where thousands of computing units can process large amounts of data. Such power allows for the solution of problems that are out of reach with contemporary multi-core CPU technology. Recent research shows that the use of GPUs can

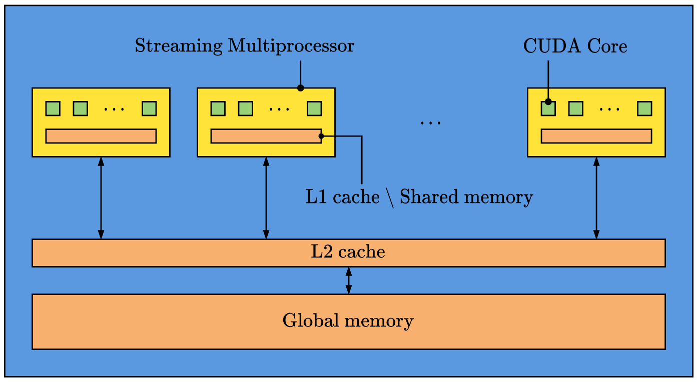

Fig. 2: High level GPU architecture.

be beneficial for speeding up basic Computational Logic tasks. See, among many, [12,11] for SAT, [14,15] for ASP, and [44] for CP. However, accessing the computational power offered by GPUs demands specific techniques and algorithms that proficiently exploit the peculiarities of the GPU architecture. To support developers and researchers, NVIDIA introduced *CUDA* (Computing Unified Device Architecture) [34], a C/C++ Application Programming Interface (API) that allows to ignore the underlying graphical concepts in favor of parallel computing concepts. A typical CUDA program is composed of parts executed by the CPU, the *host*, and parts designed to be executed on the GPU, the *device*. The host parts contain instructions for data movement and computation offloading, while the device parts contain the code that performs the computation.

A current NVIDIA GPU contains up to one hundred *Streaming Multiprocessors* (SM), each containing up to one hundred computational units called CUDA Cores (see Figure 2). The main GPU memory is called *global memory*, and it can be tens of GB large. Between global memory and SMs there is a *L2 cache* of a few MB. Finally, each SM is equipped with some tens of KB of fast memory used as *L1 cache* and/or scratchpad memory, in which case it is referred to as *shared memory*.

The CUDA computational model is defined as *Single-Instruction Multiple-Thread* (SIMT). In this model, each thread executes the same C/C++ function, named *kernel*, and uses its unique index to identify the data fragments to fetch or the control flow. The case where different threads take different control flows is called *thread divergence*, and it is handled by executing the threads one after the other. Such a behavior may cause serious performance degradation. From a programmer's perspective, threads are logically grouped in *blocks* and blocks are organized in a *grid*. Blocks are dispatched to the Streaming Multiprocessors, that run the threads using their CUDA Cores. Threads in the same block can share data using the shared memory, while threads of different blocks can only share data through the global memory.

```
include "cumulative.mzn";
include "minicpp.mzn";
...
int: m; % Number of resources
set of int: RESOURCE = 1..m;
...
constraint forall(r in RESOURCE)
  (cumulative(...) ::gpu );
...
```

Listing 1.1: MiniZinc annotation to use the GPU-accelerated propagator.

To take full advantage of GPU architecture, one has to adhere to specific programming directives to distribute the workload among the cores, avoid thread divergence and optimize memory accesses. This usually involves exploiting the shared memory to reduce costly global memory accesses.

3 Design and Implementation

This section describes the process of developing a constraint solver which supports the GPU-accelerated propagation of *Cumulative*. The first part is about the constraint solver and can be used to estimate the effort necessary to integrate our ideas into an existing solver. The second part focuses on a GPU-accelerated propagator and can be useful to evaluate how effectively other propagators can be parallelized.

Solver The exploitation of a GPU-based propagator within a solver has some caveats. The first is that the solver is *open-source* because intimate modifications of internals might be needed. Second, it is preferable that the solver is written in *C/C++* to facilitate the interaction with CUDA. Finally, it is convenient that the solver supports the *MiniZinc* language so that the GPU-accelerated propagator is easily accessible by the community.

We choose to work with MiniCP [30] because it is open-source and it is reasonably simple to modify thanks to the comprehensive documentation and the straightforward mapping between its architecture and the theory. In particular we used MiniCPP [19], a C++ implementation of MiniCP. The integration of the GPU-accelerated propagator is the same as any other propagator, but it requires modifying the build process to properly handle CUDA code. The addition of a FlatZinc frontend [42], few variable/value selection heuristics and some constraints were sufficient to obtain a solver compatible with MiniZinc [41]. To provide a simple mechanism to use the GPU-accelerated propagator, we introduced a new MiniZinc annotation. Specifically, a constraint annotated with `::gpu` is enforced using the GPU-accelerated propagator in place of the CPU implementation (see Listing 1.1).

Propagator The GPU can be used to enhance constraints propagation according to two strategies: speed-up the fastest algorithms, or lower the computational price of strong filtering algorithms. The first strategy has different downsides: offloading to a GPU introduces an overhead that may overshadow the speed-up, and it may not be obvious to parallelize the best (sequential) algorithm because of its data structures. On the contrary, strong filtering algorithms may expose enough parallelizable work to make it convenient to offload the computation, but it may still be too slow to be beneficial.

Let us consider prior implementations proposed for Edge-Finding, Time-Tabling, etc. to single out the most promising one for GPU parallelization. We evaluate them based on the data structures they use, preferring plain data structures since *pointer chasing* (i.e., a sequence of irregular memory accesses following chains of pointers) is quite harmful on a GPU. The Time-Table propagator, as proposed in [17], seems to be a good candidate since other implementations are impeded by the use of heap data structures. Among the Edge-Finding propagators found in the literature, the most promising are those proposed in [29,24], as other approaches heavily rely on *linked* data-structures (trees, queues, lists) and involve pointer chasing. All the Not-First/Not-Last approaches are equally dependent on *linked* data-structures. The standard Energetic-Reasoning [4] is a strong filtering candidate which uses only array-like data structure. We decided to base our GPU-accelerated propagator on Energetic-Reasoning for both its GPU-friendly data structures, and because on typical instances and with "enough" GPU cores, it is possible to generate and check all the $O(n^2)$ intervals in parallel, reducing the running time from $O(n^3)$ to $O(n)$.

3.1 Parallelization

This section describes and motivates the developing of a parallel Energetic-Reasoning propagator. The first part introduces the notions of occupancy and latency, two fundamental concepts of GPU computing. The second part details how we parallelized the filtering algorithm, while the final part is about overhead reduction.

Performance considerations Propagators are called thousands of times and run for a few milliseconds. To derive a speedup, a GPU-accelerated propagator must *maximize occupancy* and *minimize latency.*

Occupancy refers to how many and how effectively GPU cores are used. A good algorithm uses fine-grain parallelism to engage many GPU cores, and relies on cache-friendly data structures to mitigate memory stalls. Latency refers to the time used to transfer data –and control– to the GPU as well as the time to retrieve results and recover control back to the CPU. Such operations are *very* expensive, so it is crucial to minimize both their duration and frequency.

Data layout It is convenient to specify how data is organized on the GPU. All the data are stored in the global memory using dynamically allocated and statically sized vectors. Such vectors are triples (p, s, c), where p is a *pointer* to the

```
$propagationFailed = False$
initStartingTimesFromDomains($S$)
memcpyCpuToGpu([$propagationFailed$, $S$])            /* Asynchronous API */
calcIntervalsKernel($S$, $RI$)                       /* Asynchronous launch */
updateBoundsKernel($S$, $RI$, $propagationFailed$)  /* Asynchronous launch */
memcpyGpuToCpu([$propagationFailed$, $S$])            /* Asynchronous API */
waitGpu()
if $\neg propagationFailed$ then
    updateDomains($S$)
else
    fail()
```

Algorithm 2: Pseudocode of the parallel Energetic-Reasoning propagator.

allocated memory block, s is the current *size*, and c is the maximum *capacity*. This representation does not rely on links of pointers and can be allocated when the constraint is created. Specifically, four vectors are kept in the GPU memory: P containing the processing time of activities, H containing the resource usage of activities, RI[4] containing the relevant intervals (pairs of integers), and S containing pairs of integers representing the earlier/latest starting time of activities.

Parallel algorithm The parallelization of Algorithm 1 begins with the parallel computation of RI. A GPU kernel named *calcIntervalsKernel* calculates and merges the sets $O(i, j)$ of each $(i, j) \in A \times A$. Then, the outer loop is parallelized by a kernel named *updateBoundsKernel* that processes the intervals $[t_1, t_2) \in RI$. The resulting parallel propagator is listed in Algorithm 2 and available in the `gpu` branch of [41] .

Let $\#SM$ be the number of Streaming Multiprocessors, and $\#CS$ be the number of CUDA Cores per Streaming Multiprocessors. We maximize the occupancy of *calcIntervalsKernel* by running it with $\#SM$ blocks, each of $\#CS$ threads so that each thread is responsible for about $\frac{|A \times A|}{\#SM \cdot \#CS}$ pairs of activities. In details, each thread generates some elements of RI in shared memory and stores them in RI, that is in global memory. To store the elements, each thread first reserves enough space in RI and then writes the elements. The reservation is done with a single atomic increment on the size of the vector. Such increment is the only point where threads might be serialized. The occupancy of *updateBoundsKernel* is maximized by launching it with $\#SM$ blocks, each of $\#CS$ threads so that each thread is responsible for about $\frac{|RI|}{\#SM \cdot \#CS}$ intervals (see Figure 3). To retain correctness, each update of S must be *atomic*. Because of the massive number of threads concurrently accessing S, such atomic operations, if performed on global memory, would cause contention and slow down. Shared memory can be used instead by creating a copy S' of S for each block to reduce contention. Once all threads complete their computations, S is updated using S'. Naturally, updates

[4] From now on we will use RI to refer both to the set and to the relative vector.

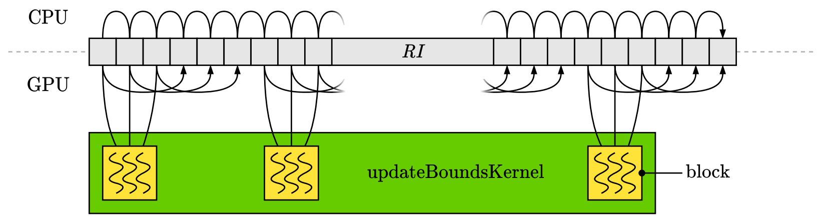

Fig. 3: Sequential (top) vs parallel (bottom) processing of RI.

of S's are still atomic, but since their scopes are single blocks, different blocks do not interfere.

Overhead reduction The first step to reduce the overhead is to minimize the volume of data transferred to/from the GPU. Since vectors P and H are constant, it suffices to copy them to the GPU when the constraint is posted. The only data that the host has to communicate to the GPU is the vector S, while it has to retrieve both the updated S and the Boolean *propagationFailed*. A possibility consists in using *CUDA Unified Memory* to exchange data between CPU and GPU. In this case, the CUDA runtime autonomously copies the data between host and device through a paging mechanism that, unfortunately, introduces a not negligible overhead. Hence, we packed S and *propagationFailed* into a structure and explicitly copy it to/from the GPU as a single block of data when needed. In Figure 4 such transfers are represented in cyan and magenta.

Another source of overhead originates from CUDA asynchronous calls. The bottom part of Figure 4 illustrates on a timeline the latency one experiences when multiple such calls occur. The alternative is to use *CUDA Graphs* to organize all kernel launches and memory operations in a dependency graph in such a way that they can be launched by means of a *single* API call.

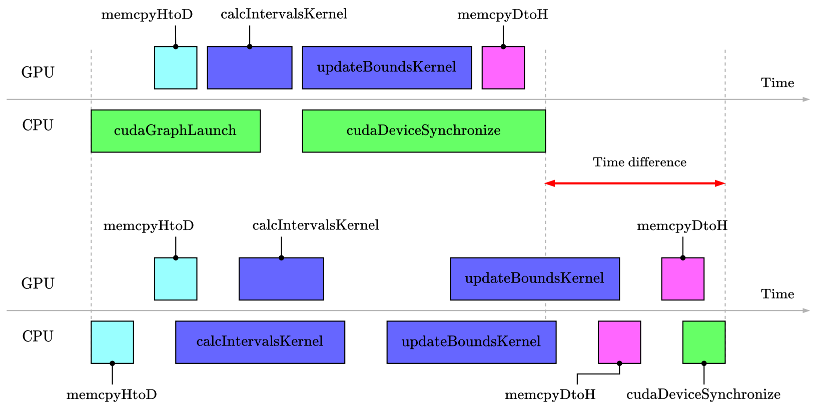

Fig. 4: Propagation with (top) and without (bottom) the use of CUDA Graph.

A final note is about the possibility to offload the propagation of multiple constraints at the same time. Parallel constraint propagation on GPU is possible and we are currently exploring it. Further investigation on such topic is warranted, and will be the subject of future work.

4 Experiments

This section presents the result of a comparison between the GPU-accelerated propagator, the CPU implementation and state-of-the-art solvers on different sets of instances from the literature. Moreover, it shows the benefits of the heuristics introduced in Section 2.2. We used the RCPSP as a benchmark. It is a generalization of CuPS with multiple resources and precedences between activities. In CP it is usually modeled with multiple *Cumulative* and linear constraints. Hence, it is particularly well-suited to evaluate the performance of a *Cumulative* propagator. The evaluation considered three established sets of instances, for a total of 299 instances:

PSPLib. Introduced in [25], it is the most popular benchmark for RCPSP. It contains synthetic instances of 30, 60, 90, and 120 activities. Instances are classified by their generation parameters, for a total of 204 classes, each of 10 instances. To have a reasonable benchmark time, we considered only the first instance of each class.

BL. Introduced in [5], it is part of a study about solving highly disjunctive and highly cumulative instances. It contains 40 highly cumulative synthetic instances, with 20 and 25 activities.

Pack. Introduced in [8], it is part of a study that uses sharp makespan's lower bounds to solve the RCPSP. It contains 55 highly cumulative synthetic instances, with 17 to 35 activities.

For a detailed description of the benchmarks, the reader can refer to [3]. Both the model and instances are from the MinZinc Benchmark Suite [32] and make use of `smallest` as variable selection heuristics. The model, instances and benchmark scripts are available at [43]. The system used in the experiments is equipped with an Intel Core i7-10700K, 32GB of RAM, and a NVIDIA GeForce RTX 3080. It runs Ubuntu 22.04 with CUDA 11.8.

The solvers included in the comparison are the top two open-source not-LCG (Lazy Clause Generation) solvers of the MiniZinc Challenge 2022 [31]: Jacop [21], and Gecode [18]. We focused on open-source solvers because MiniCPP is open-source, and on not-LCG solvers because we wanted to assess the specific benefits of parallelizing the propagator. However, there is nothing that precludes the use of GPU-accelerated propagators in a LCG solver. Note that neither Jacop nor Gecode provide Energetic-Reasoning propagators, so we compared ours with their Time-Tabling and Edge-Finding propagators. Since it is not possible to select a specific propagator from the MiniZinc model, we did it by modifying the source code of Jacop (version 4.9.0) and Gecode (version 6.3.0). We called such solvers `Jacop-TT`, `Jacop-EF`, `Gecode-TT`, and `Gecode-EF`, while

`MiniCPP-ER` and `MiniCPP-ER-GPU` stand for MiniCPP using the sequential and the GPU-accelerated Energetic-Reasoning propagators, respectively.

4.1 Results and Analysis

To give an effective and concise presentation, we focus on the instances for which a reasonable comparison of the solvers is possible. Namely, to be selected, an instance must satisfy one of the following criteria:

1. It has been solved by at least one solver, and at least half of the solvers had spent more than 10 seconds on the search. In this way, we rule out easier instances.
2. It was not solved by any solver and at least half of the solvers reported a solution after 10 seconds. This way we filter out instances for which there is not enough information on the progress of the search.

With 30 minutes timeout, these criteria select 31 instances in category 1 and 50 in category 2.

Category 1 The results of the instances in category 1 are illustrated in Figure 5 and summarized in Table 2. The plots use logarithmic scale on the time axis, while each entry of the table is the sum of the corresponding statistic among all the instances. Overall, `MiniCPP-ER-GPU` results are compelling. It is the only approach that completed the search for all the instances and has the smallest total search time. The BL benchmark offers a direct comparison between `MiniCPP-ER` and `MiniCPP-ER-GPU` since both solved all its instances. In this case the GPU-accelerated solver is an order of magnitude faster. For the highly cumulative BL and Pack, Energetic-Reasoning leads to a smaller search tree that translates in a smaller search time only for `MiniCPP-ER-GPU`. For the PSPLib instances, Energetic-Reasoning leads to bigger search trees and `Jacop-TT` results the fastest solver. A similar outcome was observed in [9].

Category 2 The results of the instances in category 2 are reported in Table 3. The number of optimally solved instances is replaced with the number of solutions and the search time with the Area Under the Curve (AUC). There are no instances of BL for category 2. The numbers confirm what was observed for category 1: `MiniCPP-ER-GPU` is the best solver for the Pack benchmark, having the smaller AUC and the bigger number of solutions, while `Jacop-TT` is the best on the PSPLib instances. Naturally, it remains possible to add Time-Tabling propagators alongside the Energetic-Reasoning propagators to get an additional boost, yet this remains a topic for future research.

To analyze the effects of the heuristics proposed at the end of Section 2.2 we implemented it in `MiniCPP-ER*`/`MiniCPP-ER*-GPU`, and test them on all the 299 instances. The results are summarized in Table 1. It reports the aggregate statistics of the instances optimally solved by all the Energetic-Reasoning implementations within the timeout of 30 minutes. The heuristics leads to an increment of the search tree size of less than 1% in the worst case (i.e., PSPLib), while improving the search time of the CPU implementation by at least 2.85 times (i.e. Pack). On the contrary, the GPU-accelerated implementation is barely affected, with a slowdown of less than 0.1%. That confirms the effectiveness of the parallelization and shows the penalty incurred in having *thread divergence*.

5 Conclusions

This paper revisited cumulative scheduling and offered a GPU-based implementation of the Energetic-Reasoning propagator. Energetic-Reasoning, while having one of the strongest filtering power, has often been sidelined because of its prohibitive runtime. The advent of GPU computing offers massive parallelism that opens the door to reconsider such design decisions and make this stronger contender viable. This paper reviewed Energetic-Reasoning and detailed key considerations for its implementation on GPUs. The empirical evaluation demonstrated that this is a worthwhile technique that is competitive, scales well and, should be part of CP toolkits.

Acknowledgements

Agostino Dovier and Andrea Formisano are partially supported by Interdepartment Project on AI and by INdAM-GNCS projects CUP E55F22000270001 and CUP E53C22001930001. Laurent Michel is partially supported by Synchrony.

Benchmark	Solver	Optimal	Time (s)	Nodes (M)	Failures (M)	Depth
BL	`MiniCPP-ER`	9	3072	3.31	1.10	349
	`MiniCPP-ER*`	9	985	3.32	1.11	353
	`MiniCPP-ER-GPU`	9	228	3.31	1.10	349
	`MiniCPP-ER*-GPU`	9	237	3.32	1.11	353
Pack	`MiniCPP-ER`	3	2277	2.17	0.72	147
	`MiniCPP-ER*`	3	799	2.19	0.73	147
	`MiniCPP-ER-GPU`	3	148	2.17	0.72	147
	`MiniCPP-ER*-GPU`	3	159	2.19	0.73	147
PSPLib	`MiniCPP-ER`	6	2138	0.26	0.09	558
	`MiniCPP-ER*`	6	535	0.30	0.10	566
	`MiniCPP-ER-GPU`	6	28	0.26	0.09	558
	`MiniCPP-ER*-GPU`	6	30	0.30	0.10	566

Table 1: Aggregate statistics for the proposed Energetic-Reasoning heuristics.

Benchmark	Solver	Optimal	Time (s)	Nodes (M)	Failures (M)	Depth
BL	`Gecode-TT`	15	8436	297.55	148.78	799
	`Gecode-EF`	0	30600	187.34	93.67	791
	`Jacop-TT`	13	11551	2307.47	1153.74	1123
	`Jacop-EF`	17	1022	22.76	11.38	787
	`MiniCPP-ER`	17	3163	3.46	1.15	605
	`MiniCPP-ER-GPU`	17	233	3.46	1.15	605
Pack	`Gecode-TT`	2	7812	222.34	111.17	308
	`Gecode-EF`	0	10800	40.62	20.31	272
	`Jacop-TT`	0	10782	2458.11	1229.05	814
	`Jacop-EF`	3	7219	177.93	88.97	463
	`MiniCPP-ER`	3	7674	5.37	1.79	286
	`MiniCPP-ER-GPU`	6	2509	23.76	7.92	307
PSPLib	`Gecode-TT`	4	8085	137.33	68.66	507
	`Gecode-EF`	0	14400	32.12	16.06	564
	`Jacop-TT`	8	404	34.01	17.00	502
	`Jacop-EF`	8	3506	33.40	16.70	496
	`MiniCPP-ER`	3	10049	11.97	3.99	485
	`MiniCPP-ER-GPU`	8	3142	43.88	14.63	492

Table 2: Aggregate statistics for the instances in category 1.

Benchmark	Solver	Solutions	AUC (K)	Nodes (M)	Failures (M)	Depth
Pack	`Gecode-TT`	41	16.99	493.47	246.74	520
	`Gecode-EF`	0	1157.40	60.17	30.09	458
	`Jacop-TT`	38	17.88	4112.02	2056.01	1646
	`Jacop-EF`	41	14.01	343.27	171.63	978
	`MiniCPP-ER`	28	45.66	14.22	4.74	669
	`MiniCPP-ER-GPU`	45	10.62	238.52	79.51	806
PSPLib	`Gecode-TT`	344	146.02	443.49	221.74	7613
	`Gecode-EF`	4	11176.49	25.44	12.72	9739
	`Jacop-TT`	412	22.70	4960.48	2480.23	6570
	`Jacop-EF`	382	84.48	861.81	430.90	5918
	`MiniCPP-ER`	312	213.83	7.15	2.38	4744
	`MiniCPP-ER-GPU`	389	76.45	354.61	118.20	5549

Table 3: Aggregate statistics for the instances in category 2.

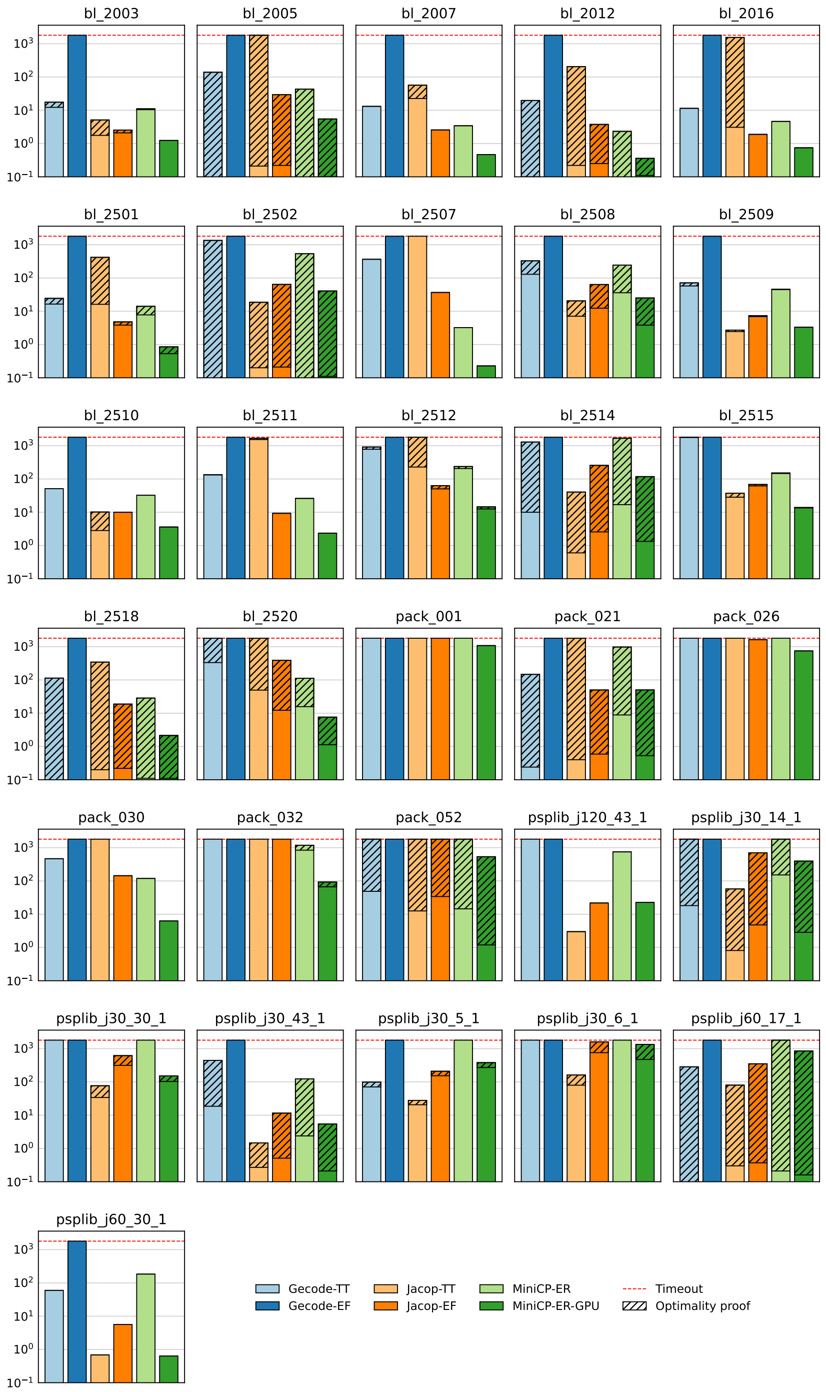

Fig. 5: Search time (in seconds) for all the instances in category 1.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling pp. 57–73 (1993). https://doi.org/10.1016/0895-7177(93)90068-a
2. Apt, K.: Principles of Constraint Programming. Cambridge University Press (aug 2003). https://doi.org/10.1017/cbo9780511615320
3. Artigues, C., Demassey, S., Nron, E. (eds.): Resource-Constrained Project Scheduling. Iste (2008). https://doi.org/10.1002/9780470611227
4. Baptiste, P., Le Pape, C., Nuijten, W.: Satisfiability tests and time-bound adjustments for cumulative scheduling problems. Annals of Operations Research **92**(0), 305–333 (1999). https://doi.org/10.1023/a:1018995000688
5. Baptiste, P., Le Pape, C.: Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. In: Principles and Practice of Constraint Programming-CP97, pp. 375–389. Springer Berlin Heidelberg (1997). https://doi.org/10.1007/bfb0017454
6. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling. Springer US (2001). https://doi.org/10.1007/978-1-4615-1479-4
7. Beldiceanu, N., Carlsson, M.: A new multi-resource cumulatives constraint with negative heights. In: Lecture Notes in Computer Science, pp. 63–79. Springer Berlin Heidelberg (2002). https://doi.org/10.1007/3-540-46135-3_5
8. Carlier, J., Néron, E.: On linear lower bounds for the resource constrained project scheduling problem. European Journal of Operational Research **149**(2), 314–324 (2003). https://doi.org/10.1016/s0377-2217(02)00763-4
9. Cauwelaert, S.V., Lombardi, M., Schaus, P.: Understanding the potential of propagators. In: Integration of AI and OR Techniques in Constraint Programming. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-18008-3_29
10. Cheng, J., Grossman, M., McKercher, T.: Professional CUDA C Programming. EBL-Schweitzer, Wiley (2014), https://www.wiley.com/en-us/Professional+CUDA+C+Programming-p-9781118739310
11. Collevati, M., Dovier, A., Formisano, A.: GPU parallelism for SAT solving heuristics. In: Calegari, R., Ciatto, G., Omicini, A. (eds.) Proceedings of the CILC'22. CEUR Workshop Proceedings, vol. 3204, pp. 17–31. CEUR-WS.org (2022)
12. Dal Palù, A., Dovier, A., Formisano, A., Pontelli, E.: CUD@SAT: SAT solving on GPUs. J. Exp. Theor. Artif. Intell. **27**(3), 293–316 (2015). https://doi.org/10.1080/0952813X.2014.954274
13. Derrien, A., Petit, T.: A new characterization of relevant intervals for energetic reasoning. In: Lecture Notes in Computer Science, pp. 289–297. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-10428-7_22
14. Dovier, A., Formisano, A., Pontelli, E.: Parallel answer set programming. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 237–282. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3_7
15. Dovier, A., Formisano, A., Vella, F.: GPU-Based Parallelism for ASP-Solving. In: Hofstedt, P., Abreu, S., John, U., Kuchen, H., Seipel, D. (eds.) Declarative Programming and Knowledge Management. Lecture Notes in Computer Science, vol. 12057, pp. 3–23. Springer (2019). https://doi.org/10.1007/978-3-030-46714-2_1
16. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., USA (1990)

17. Gay, S., Hartert, R., Schaus, P.: Simple and scalable time-table filtering for the cumulative constraint. In: Lecture Notes in Computer Science, pp. 149–157. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-23219-5_11
18. Gecode Team: GECODE, https://github.com/Gecode/gecode
19. Gentzel, R., Michel, L., van Hoeve, W.J.: HADDOCK: A language and architecture for decision diagram compilation. In: Lecture Notes in Computer Science, pp. 531–547. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-58475-7_31
20. Gingras, V., Quimper, C.G.: Generalizing the edge-finder rule for the cumulative constraint. In: Kambhampati, S. (ed.) Proceedings IJCAI 2016. pp. 3103–3109. IJCAI/AAAI Press (2016)
21. Jacop Team: JaCoP, https://github.com/radsz/jacop
22. Kameugne, R., Betmbe Fetgo, S., Gingras, V., Ouellet, Y., Quimper, C.G.: Horizontally elastic not-first/not-last filtering algorithm for cumulative resource constraint. In: Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 316–332. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-93031-2_23
23. Kameugne, R., Fotso, L.P.: A cumulative not-first/not-last filtering algorithm in $\mathcal{O}(n^2 \log n)$. Indian Journal of Pure and Applied Mathematics **44**(1), 95–115 (2013). https://doi.org/10.1007/s13226-013-0005-z
24. Kameugne, R., Fotso, L.P., Scott, J., Ngo-Kateu, Y.: A quadratic edge-finding filtering algorithm for cumulative resource constraints. In: Principles and Practice of Constraint Programming - CP 2011, pp. 478–492. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_37
25. Kolisch, R., Sprecher, A.: PSPLIB - a project scheduling problem library. European Journal of Operational Research **96**(1), 205–216 (1997). https://doi.org/10.1016/s0377-2217(96)00170-1
26. Lahrichi, A.: Scheduling: the notions of hump, compulsory parts and their use in cumulative problems. Comptes Rendus De L Academie Des Sciences Serie I-mathematique **294**(6), 209–211 (1982)
27. Letort, A., Beldiceanu, N., Carlsson, M.: A scalable sweep algorithm for the cumulative constraint. In: Lecture Notes in Computer Science, pp. 439–454. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_33
28. Lopez, P.: Energy-based approach for task scheduling under time and resource constraints. Ph.D. thesis, Université Paul Sabatier-Toulouse III (1991)
29. Mercier, L., Van Hentenryck, P.: Edge finding for cumulative scheduling. INFORMS Journal on Computing **20**(1), 143–153 (2008). https://doi.org/10.1287/ijoc.1070.0226
30. Michel, L., Schaus, P., Van Hentenryck, P.: MiniCP: a lightweight solver for constraint programming. Mathematical Programming Computation pp. 133–184 (2021). https://doi.org/10.1007/s12532-020-00190-7
31. MiniZinc Team: MiniZinc Challenge 2022 Results, https://www.minizinc.org/challenge2022/results2022.html
32. MiniZinc Team: The MiniZinc Benchmark Suite, https://github.com/MiniZinc/minizinc-benchmarks
33. Nuijten, W.: Time and resource constrained scheduling: a constraint satisfaction approach. Ph.D. thesis, Eindhoven University of Technology (1994)
34. Nvidia Team: CUDA, https://developer.nvidia.com/cuda-toolkit
35. Ouellet, P., Quimper, C.G.: Time-table extended-edge-finding for the cumulative constraint. In: Lecture Notes in Computer Science, pp. 562–577. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_42

36. Ouellet, Y., Quimper, C.G.: A $\mathcal{O}(n \log^2 n)$ checker and $\mathcal{O}(n^2 \log n)$ filtering algorithm for the energetic reasoning. In: Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 477–494. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-93031-2_34
37. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2. Elsevier (2006), https://www.sciencedirect.com/science/bookseries/15746526/2
38. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Pearson Education (2010), https://developer.nvidia.com/cuda-example
39. Schutt, A., Wolf, A.: A new $\mathcal{O}(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints. In: Principles and Practice of Constraint Programming - CP 2010, pp. 445–459. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-15396-9_36
40. Schutt, A., Wolf, A., Schrader, G.: Not-first and not-last detection for cumulative scheduling in $\mathcal{O}(n^3 \log n)$. In: Lecture Notes in Computer Science, pp. 66–80. Springer Berlin Heidelberg (2006). https://doi.org/10.1007/11963578_6
41. Tardivo, F.: fzn-minicpp, https://bitbucket.org/constraint-programming/fzn-minicpp
42. Tardivo, F.: libfzn, https://bitbucket.org/constraint-programming/libfzn
43. Tardivo, F.: MiniCPP-Benchmarks, https://bitbucket.org/constraint-programming/minicpp-benchmarks
44. Tardivo, F., Dovier, A., Formisano, A., Michel, L., Pontelli, E.: Constraints propagation on GPU: a case study for AllDifferent. In: Calegari, R., Ciatto, G., Omicini, A. (eds.) Proceedings of CILC'22. CEUR Workshop Proceedings, vol. 3204, pp. 61–74. CEUR-WS.org (2022)
45. Tesch, A.: A nearly exact propagation algorithm for energetic reasoning in $\mathcal{O}(n^2 \log n)$. In: Lecture Notes in Computer Science, pp. 493–519. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-44953-1_32
46. Tesch, A.: Improving energetic propagations for cumulative scheduling. In: Lecture Notes in Computer Science, pp. 629–645. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-98334-9_41
47. Vilím, P.: Edge finding filtering algorithm for discrete cumulative resources in $\mathcal{O}(kn \log n)$. In: Principles and Practice of Constraint Programming - CP 2009, pp. 802–816. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_62
48. Vilím, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 230–245. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-21311-3_22