A $O(n \log^2 n)$ Checker and $O(n^2 \log n)$ Filtering Algorithm for the Energetic Reasoning

Yanick Ouellet[1] and Claude-Guy Quimper[1]

Université Laval
Québec City, Canada
yanick.ouellet.2@ulaval.ca
claude-guy.quimper@ift.ulaval.ca

Abstract. Energetic reasoning is a strong filtering technique for the CUMULATIVE constraint. However, the best algorithms process $O(n^2)$ time intervals to perform the satisfiability check which makes it too costly to use in practice. We present how to apply the energetic reasoning by processing only $O(n \log n)$ intervals. We show how to compute the energy in an interval in $O(\log n)$ time. This allows us to propose a $O(n \log^2 n)$ checker and a filtering algorithm for the energetic reasoning with $O(n^2 \log n)$ average time complexity. Experiments show that these two algorithms outperform their state of the art counterparts.

1 Introduction

There exist many filtering rules for the CUMULATIVE constraint. Among them, the energetic reasoning rule [3, 7, 14] dominates the overload check [8, 23], the time-tabling [4], the edge-finder [15], and the time-tabling-edge-finder [22]. To apply the energetic reasoning, one needs to process $O(n^2)$ time intervals, which might be too slow in practice.

We introduce a technique based on Monge matrices to explicitly process only $O(n \log n)$ of the $O(n^2)$ intervals. The remaining intervals are processed implicitly. This allows us to propose the first subquadratic checker for the energetic reasoning, with a $O(n \log^2 n)$ running time. We also propose a new filtering algorithm that filters all tasks with an average running time complexity of $O(n^2 \log n)$ and a worst case running time complexity of $O(n^2 \log^2 n)$. However, we do not know whether the bound $O(n^2 \log^2 n)$ is tight as we did not succeed in finding an instance requiring that much time to filter.

The next section formally presents the CUMULATIVE constraint and the energetic reasoning rule. Section 3 presents some algorithmic background, including the Monge matrices that we use to design our algorithms. Section 4 introduces an adaptation of the range trees that is used in Section 5 to compute the energy in a time interval. Section 6 presents the subquadratic checker, and Section 7, the $O(n^2 \log n)$ filtering algorithm. Section 8 shows the performance of these algorithms on classic benchmarks.

2 Scheduling Background

Let $\mathcal{I} = \{1, \ldots, n\}$ be the set of task indices. Each task i is defined with four integer parameters: the *earliest starting time* est_i, the *latest completion time* lct_i, the processing

time p_i, and the resource consumption rate h_i. From these parameters, one can compute the *earliest completion time* $\mathrm{ect}_i = \mathrm{est}_i + p_i$, the *latest starting time* $\mathrm{lst}_i = \mathrm{lct}_i - p_i$, and the energy $e_i = p_i h_i$ of the task. The horizon spans from time $\mathrm{est}_{\min} = \min_{i \in \mathcal{I}} \mathrm{est}_i$ to time $\mathrm{lct}_{\max} = \max_{i \in \mathcal{I}} \mathrm{lct}_i$. A task i starts executing at time S_i and executes for p_i units of time without interruption. The starting time S_i is unknown but must belong to the time interval $\mathrm{dom}(S_i) = [\mathrm{est}_i, \mathrm{lst}_i]$. The task is necessarily executing during the time interval $[\mathrm{lst}_i, \mathrm{ect}_i)$ if $lst_i < \mathrm{ect}_i$. This time interval is called the *compulsory part*. A cumulative resource can simultaneously execute multiple tasks as long as the total resource consumption rate of the tasks executing at any time t does not exceed the capacity C of the resource. The constraint CUMULATIVE [1] ensures that the starting times of the tasks do not overload the capacity of the resource.

$$\textsc{Cumulative}([S_1, \ldots, S_n], [p_1, \ldots, p_n], [h_1, \ldots, h_n], C) \iff \forall t \sum_{i : S_i \leq t < S_i + p_i} h_i \leq C$$

Deciding whether the constraint CUMULATIVE is satisfiable is NP-Complete. For that reason, there are no polynomial time filtering algorithms that can achieve bounds consistency for this constraint. However, there exist multiple filtering rules that partially remove the inconsistent values from the domains of the starting variables. The overload check [8, 23], the time-tabling [4], the edge-finder [15], the time-table-edge-finder [22], and the not-first not-last [10, 17, 18] are popular filtering rules. With the exception of not-first not-last, all these rules are dominated by the energetic reasoning that detects more inconsistencies and filters more values [3]. The energetic reasoning is incomparable to the not-first not-last.

A task i that starts at its earliest starting time spends, during the time interval $[l, u)$, an amount of energy equal to $LS(i, l, u) = h_i \max(\min(\mathrm{ect}_i - l, p_i, u - l), 0)$. This energy is called the *left-shift*. If task i starts at its latest completion time, it spends, during the time interval $[l, u)$, an amount of energy equal to $RS(i, l, u) = h_i \max(\min(u - \mathrm{lst}_i, p_i, u - l), 0)$. This energy is called the *right-shift*. Finally, regardless of its starting time, a task i must spend during the time interval $[l, u)$ an amount of energy called *left-shift/right-shift* and denoted $LSRS(i, l, u)$.

$$LSRS(i, l, u) = \min(LS(i, l, u), RS(i, l, u)) \tag{1}$$
$$= h_i \max(\min(\mathrm{ect}_i - l, u - \mathrm{lst}_i, p_i, u - l), 0) \tag{2}$$

By abuse of notation, we define the *left-shift/right-shift* for a set of tasks Θ: $LSRS(\Theta, l, u) = \sum_{i \in \Theta} LSRS(i, l, u)$. The *slack* $S(\Theta, l, u)$ is the amount of remaining energy, for a cumulative resource of capacity C over an interval $[l, u)$, after spending the left-shift/right-shift of a set of tasks Θ.

$$S(\Theta, l, u) = C \cdot (u - l) - LSRS(\Theta, l, u) \tag{3}$$

The energetic reasoning tests [3], for every time interval $[l, u)$, whether the slack is non-negative: $S(\mathcal{I}, l, u) \geq 0$.

Baptiste et al. [3] showed that not all time intervals $[l, u)$ need to be tested. Let O_1, O_2, and $O(t)$ be such that

$$\begin{aligned} O_1 &= \{\mathrm{est}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{ect}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{lst}_i \mid i \in \mathcal{I}\} \\ O_2 &= \{\mathrm{lct}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{ect}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{lst}_i \mid i \in \mathcal{I}\} \\ O(t) &= \{\mathrm{est}_i + \mathrm{lct}_i - t \mid i \in \mathcal{I}\} \end{aligned}$$

Only the time intervals $[l, u)$ that fall into one of these three situations are considered of interest: 1) $l \in O_1$ and $u \in O_2$; 2) $l \in O_1$ and $u \in O(l)$; 3) $l \in O(u)$ and $u \in O_2$.

The energetic reasoning filtering consists in increasing est_i and decreasing lct_i to ensure that the energetic reasoning check would pass if the tasks was executed at its earliest starting time or latest starting time. The filtering rule for the est states that if the left-shift of the task i is greater than the slack of the remaining tasks in an interval $[l, u)$, est_i must be adjusted to $\left\lceil u - \frac{S(\mathcal{I}\setminus\{i\},l,u)}{h_i} \right\rceil$.

$$S(\mathcal{I} \setminus \{i\}, l, u) < LS(i, l, u) \implies est'_i = \left\lceil u - \frac{S(\mathcal{I} \setminus \{i\}, l, u)}{h_i} \right\rceil \tag{4}$$

Derrien and Petit [7] show that it is sufficient to test a subset of the intervals of Baptiste et al. to reach the fix point. To filter the est of a task i, one has to apply the filtering rule on all intervals in $O_C \cup L_i$. The set O_C contains at most two intervals for each pair of tasks and thus has a cardinality in $O(n^2)$. The set L_i contains $2n + 1$ intervals. Similarly, to filter the lct of a task, one has to apply the filtering rules on intervals in $O_C \cup R_i$, where R_i is symmetric to L_i. The definitions of O_C, L_i, and R_i are based on a long enumeration of cases, but are straightforward to compute. By lack of space, we refer the reader to [7] for a complete definition of these sets.

While the energetic reasoning achieves a strong level of filtering, it is not commonly used in practice due to its slow computation time. Baptiste et al. [3] proposed a checker with a running time complexity of $O(n^2)$. Their algorithm asymptotically remains the fastest checker in the literature. Nevertheless, Derrien and Petit [7] reduced the number of intervals to check by a constant. This improvement led to a checker with equivalent running time complexity, but faster in practice.

Baptiste et al. [3] also present an algorithm in $O(n^3)$ to filter the constraint. Bonifas [5] introduced an algorithm that filters at least one task in $O(n^2 \log n)$ time. Tesch [21] presents an algorithm that achieves a weaker level of filtering in $O(n^2 \log n)$ time which is later improved to perform an exact filtering in $O(n^2 \log^2 n)$ time [20].

3 Algorithmic Background

We present algorithms and data structures that will be used to design a subquadratic checker for the Energetic Reasoning.

3.1 Partial sums

Let $A[1..n]$ be an array of n integers. A partial sum query is defined such as

$$\texttt{Partial-Sum}(A, i, j) = \sum_{k=i}^{j} A[k] \tag{5}$$

To efficiently answer such a query, one preprocesses in $O(n)$ time the array A by creating an array $B[0..n]$ such that $B[0] = 0$ and $B[i] = B[i-1]+A[i]$. $\texttt{Partial-Sum}(A, i, j)$ returns $B[j] - B[i-1]$ in constant computation time.

3.2 Range trees

Consider a set of n weighted points P in a two-dimensional Cartesian plan. Each point i has two coordinates, x_i and y_i, and a weight w_i. A sum query $Q_{points}(\chi, \gamma, P)$ computes the weighted sum of all points delimited by the quater-plane $\chi \leq x$ and $y \leq \gamma$.

$$Q_{points}(\chi, \gamma, P) = \sum_{\{i \in P | \chi \leq x_i \wedge y_i \leq \gamma\}} w_i \tag{6}$$

Such queries can be answered by two-dimensional range-trees [6] . If the fractional cascading technique is used, each query can be answered online in $O(\log |P|)$ time after a $O(|P| \log |P|)$ pre-processing time of P is completed.

Each node of a range tree is associated to a set of points that serves as its label. The root P of a range tree contains all the points in P. The set of points of a node v is partitioned into two subsets left(v) and right(v), one for the left subtree and one for the right subtree. For each node v, the points contained in the left child have an abscissa smaller than or equal to the abscissa of the points contained in the right child: $i \in \text{left}(v) \wedge j \in \text{right}(v) \Rightarrow x_i \leq x_j$. Each node v of a range-tree has an attribute x_v^{mid} such $x_v^{mid} = \max_{i \in \text{left}(v)} x_i$ is the largest abscissa of a point in the left subtree.

Each node v of the range-tree has a vector Y_v of dimension $|v|$ which contains the ordinates y_i of the points $i \in v$ sorted in non-decreasing order. Three other vectors characterize the nodes. The vector W_v is a partial sum such that $W_v[i]$ is the sum of the weights of the i points in v with the smallest ordinates. The vector L_v and R_v link the points in v with the points in the left and right subtrees. There are $L_v[i]$ points in left(v) whose ordinate is no greater than $Y_v[i]$. Similarly, there are $R_v[i]$ points in right(v) whose ordinate is no greater than $Y_v[i]$.

If v is a leaf, v contains a single point i. Thus, the vectors Y_v, W_v, L_v, and R_v have length 1. The vectors $Y_v = [y_i]$ and $W_v = [w_i]$ contain the ordinate and the weight of that point. The vectors $L_v = R_v = [0]$ are the null vectors. The value x_v^{mid} is undefined.

Range-trees are built using a bottom-up approach similar to the merge sort. By definition, the leaves of range-tree are sorted in non-decreasing order of abscissa x_i. Therefore, one sorts the points in P by abscissa which gives the leaves of the tree. Then the upper level is computed by merging the vectors Y_v, W_v, L_v, and R_v from the lower level. Since creating the node v can be done in $O(|v|)$ time, building the range-tree is done in $O(|P| \log |P|)$ times.

The query $Q_{points}(\chi, \gamma, P)$ can be answered by traversing the tree from the root to a leaf. Let v be the current node initialized to the root. Let i be an index such that $Y_v[i] \leq \gamma < Y_v[i+1]$. This index is initialized by doing a binary search over the vector Y_P. If $\chi > x_v^{mid}$, then $Q_{points}(\chi, \gamma, v) = Q_{points}(\chi, \gamma, \text{right}(v))$. The current node v becomes right(v) and the index i becomes $R_v[i]$. If $\chi \leq x_v^{mid}$, $Q_{points}(\chi, \gamma, v) = Q_{points}(\chi, \gamma, \text{left}(v)) + W_{\text{right}(v)}[R_v[i]]$. The current node v becomes left(v) and the index i becomes $L_v[i]$. When the current node $v = \{j\}$ is a leaf, we return w_j if $x_j \geq \chi$ and $y_j \leq \gamma$ and zero otherwise. This computation is done in $O(\log |P|)$ time.

3.3 Monge matrices

A *Monge matrix* M is an $n \times m$ matrix such that for any pair of rows $1 \leq i_1 < i_2 \leq n$ and any pair of columns $1 \leq j_1 < j_2 \leq m$, the inequality (7) holds.

$$M[i_2, j_2] - M[i_2, j_1] \leq M[i_1, j_2] - M[i_1, j_1] \tag{7}$$

An *inverse Monge matrix* satisfies the opposite inequality: $M[i_2, j_2] - M[i_2, j_1] \geq M[i_1, j_2] - M[i_1, j_1]$.

Consider the functions: $f_i(x) = M[i, x]$. Inequality (7) imposes the slopes of these functions to be monotonic. By choosing $i_1 = i$, $i_2 = i + 1$, $j_1 = x$, and $j_2 = x + 1$ and substituting in (7), one can observe the monotonic behavior of the slopes.

$$\frac{f_{i+1}(x+1) - f_{i+1}(x)}{(x+1) - x} \leq \frac{f_i(x+1) - f_i(x)}{(x+1) - x}. \tag{8}$$

It follows that the functions of two distinct rows of a Monge matrix cross each other at most once. Monge matrices satisfy many more properties [19].

Property 1. The submatrix obtained from a subset of rows and columns of a Monge matrix is a Monge matrix.

Property 2. The transpose of a Monge matrix is a Monge matrix.

Property 3. If M is a Monge matrix, $\boldsymbol{v}$ and $\boldsymbol{w}$ are two vectors, then $M'[i, j] = M[i, j] + \boldsymbol{v}[i] + \boldsymbol{w}[j]$ is a Monge matrix.

Properties 1 to 3 also hold for inverse Monge matrices.

Property 4. M is a Monge matrix if and only if $-M$ is an inverse Monge matrix.

Sethumadhavan [19] presents a survey about Monge matrices.

The *envelope* of a Monge matrix M is a function $l^*(j) = \arg\min_i M[i, j]$ that returns the row i on which appears the smallest element on column j. The envelope $l^*(j)$ of a (inverse) Monge matrix is non-increasing (non-decreasing).

A *partial Monge matrix* is a Monge matrix with empty entries. Empty entries are not subject to the inequality (7) and are ignored when computing the envelope. In this paper, we only consider partial (inverse) Monge matrices where $M[i, j]$ is empty if and only if $i > j$. The envelope of such an $n \times m$ partial (inverse) Monge matrix is non-increasing (non-decreasing) on the interval $[1, i]$ and non-decreasing (non-increasing) on the interval $[i, m]$ where $M[i, l^*(i) - 1]$ is empty. Kaplan et al. [11] compute the envelope of a $n \times m$ partial (inverse) Monge matrix in $O(n \log m)$ time. Their algorithm uses binary searches over the columns to find the intersection of the row functions $f_i(x)$.

4 Adapting the Range-trees

We adapt the range-tree data structure to perform a query on a set of weighted segments $\mathcal{S}$ instead of a set of points. Such an adaptation is required for the algorithms we present in the next sections. A segment i of weight w_i, noted $\langle x_i, x'_i, y_i, w_i\rangle$, spans from coordinates x_i to x'_i on the abscissa, at y_i on the ordinate. The query $Q_{segments}(\chi, \gamma, \mathcal{S})$ computes the weighted sum of all segment parts inside the quater-plane of the query. A segment accounts for its weight times the length of the sub-segment that is within the query range.

$$Q_{segments}(\chi, \gamma, \mathcal{S}) = \sum_{\substack{\langle x_i x'_i, y_i, w_i\rangle \in \mathcal{S} \\ y_i \leq \gamma}} w_i(\max(x'_i - \chi, 0) - \max(x_i - \chi, 0))$$

We can simplify the problem by replacing each segment by two rays i' and i'': $\langle -\infty, x'_i, y_i, w_i\rangle$ and $\langle -\infty, x_i, y_i, -w_i\rangle$. Since ray i'' cancels ray i' when i begins, the result of the query is unchanged.

We adapt the range-tree data structure to answer queries on weighted rays instead of weighted points. We add an attribute $\underline{x}_v$ to each node v of the tree that represents the smallest abscissa of a ray $\langle -\infty, x_i, y_i, w_i\rangle$ in v. In other words, each ray in v ends at $\underline{x}$ or after. We also add a vector Σ_v to each node v such that $\Sigma_v[i] = Q_{segments}(\underline{x}_v, Y_v[i], v)$ is a precomputed result of a query. If rays in v are sorted by ordinates, we have $\Sigma_v[i] = \Sigma_v[i-1] + w_i(x_i - \underline{x}_v)$.

Similar to the original range-tree, the query $Q_{segments}(\chi, \gamma, \mathcal{S})$ is computed by traversing the tree from the root $\mathcal{S}$ to a leaf. Let v be the current node initialized to the root. Let i be an index such that $Y_v[i] \leq \gamma < Y_v[i+1]$. This index is initialized by doing a binary search over the vector $Y_{\mathcal{S}}$. If $\chi > x_v^{mid}$, then $Q_{segments}(\chi, \gamma, v) = Q_{segments}(\chi, \gamma, \text{right}(v))$. The current node v becomes right(v) and the index i becomes $R_v[i]$. If $\chi \leq x_v^{mid}$, $Q_{segments}(\chi, \gamma, v) = Q_{segments}(\chi, \gamma, \text{left}(v)) + \Sigma_v[R_v[i]] + (\underline{x} - \chi) \cdot W_v[R_v[i]]$. The current node v becomes left(v) and the index i becomes $L_v[i]$. When the current node $v = \{j\}$ is a leaf, we return $(x_j - \chi) \cdot w_j$ if $y_i \leq \gamma$ and 0 otherwise. This computation is done in $O(\log |\mathcal{S}|)$ time.

5 Computing the left-shift right-shift in $O(\log n)$ time

We want to preprocess n tasks in $O(n \log n)$ time in order to compute the left-shift/right-shift $LSRS(\mathcal{I}, l, u)$ of any time interval $[l, u)$, upon request, in $O(\log n)$ time. We decompose the problem as follows. For every task i, we let $c_i = \max(0, \text{ect}_i - \text{lst}_i)$ be the length of the task's compulsory part. For every task i, we define p_i weighted semi-open intervals partitioned into two sets: the c_i compulsory intervals CI_i that lie within the compulsory part $[\text{lst}_i, \text{ect}_i)$ of the task and the $p_i - c_i$ free intervals FI_i that embed the compulsory part. The weight of all intervals is h_i.

$$CI_i = \{\langle [\text{lst}_i + k, \text{lst}_i + k + 1), h_i\rangle \mid 0 \leq k < c_i\} \tag{9}$$

$$FI_i = \{\langle [\text{est}_i + k, \text{lct}_i - k), h_i\rangle \mid 0 \leq k < p_i - c_i\} \tag{10}$$

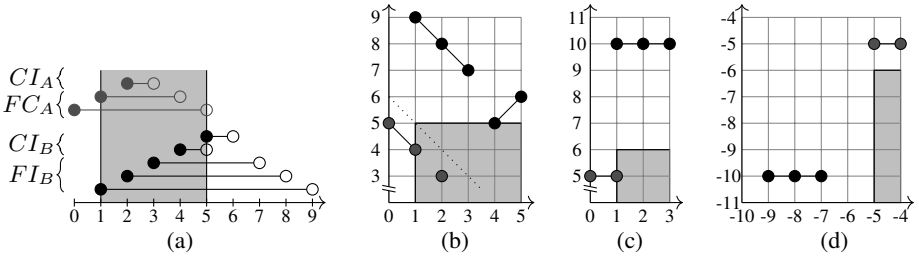

Fig. 1: a) The intervals of two tasks: task A in gray with $\mathrm{est}_A = 0$, $\mathrm{lct}_A = 5$, $p_A = 3$, $h_A = 1$ and task B in black with $\mathrm{est}_B = 1$, $\mathrm{lct}_B = 9$, $p_B = 5$, $h_B = 1$. b) The representation of the intervals on a Cartesian plan with lower bounds on the abscissa and upper bounds on the ordinate. c) First transformation. d) Second transformation. In all figures, the gray rectangle represents the query $[l, u) = [1, 5)$ that contains exactly 3 intervals of weight 1, hence $LSRS(\{A, B\}, 1, 5) = 3$.

For a set of tasks, we have: $CI_\Omega = \bigcup_{i \in \Omega} CI_i$ and $FI_\Omega = \bigcup_{i \in \Omega} FI_i$. Computing $LSRS(\mathcal{I}, l, u)$ consists of counting the weight of the intervals nested in $[l, u)$.

$$LSRS(\mathcal{I}, l, u) = \sum_{\substack{\langle [a,b), w \rangle \in CI_\mathcal{I} \\ [a,b) \subseteq [l,u)}} w + \sum_{\substack{\langle [a,b), w \rangle \in FI_\mathcal{I} \\ [a,b) \subseteq [l,u)}} w \tag{11}$$

Figure 1a shows two tasks and their intervals as well as a request interval $[l, u)$ shown in gray. The number of intervals nested in $[l, u)$ gives the amount of processing time the tasks must spend executing in $[l, u)$. When the number of intervals is weighted by the task heights, we obtain the left-shift/right-shift. Compulsary intervals have length 1 since the time at which compulsary energy is spent is known. Free intervals are longer and nested into $[l, u)$ only if the unit of processing time it corresponds to belongs both to the left-shift and the right-shift of the task. Equation (11) counts all intervals nested in $[l, u)$ and weighted by the task heights.

Figure 1b represents an interval $[a, b)$ by a point (a, b) on the Cartesian plane with the corresponding weight. The sums in (11) can be computed with the queries $Q_{points}(l, u, CI_\mathcal{I})$ and $Q_{points}(l, u, FI_\mathcal{I})$ as explained in Section 3.2. These queries are represented as gray rectangles on Figures 1a and 1b. Since the points associated to the intervals in FI_i form segments with a slope of -1 and that points in CI_i form a segment on the line $y = x + 1$, we can design efficient algorithms to compute these queries with $O(n \log n)$ processing time and $O(\log n)$ query time. Section 5.1 shows how to achieve these time bounds when computing the first summation in (11) that we call the *compulsory energy* while Section 5.2 shows how to compute the second summation that we call the *free energy*.

5.1 Computing the compulsory energy

We compute the compulsory energy that lies within an interval $[l, u)$ as follows. Let $T = \{\mathrm{ect}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{lst}_i \mid i \in \mathcal{I}\}$ be the sorted time points where the compulsory

energy can increase or decrease over time. Let Y_t be the amount of compulsory energy spent at time T_t. We compute Y_t using an intermediate vector Y'_t initialized to zero. For each task i, we increase by h_i the component Y'_t such that $T_t = \mathrm{lst}_i$ and decrement by h_i the component Y'_t such that $T_t = \mathrm{ect}_t$. We obtain these relations:

$$Y'_t = \sum_{i \in \mathcal{I} | \mathrm{lst}_i = T_t} h_i - \sum_{i \in \mathcal{I} | \mathrm{ect}_i = T_t} h_i, \qquad Y_0 = Y'_0, \qquad Y_t = Y_{t-1} + Y'_t.$$

Let Z_t be the amount of compulsory energy in the time interval $[T_t, T_{t+1})$, i.e. $Z_t = Y_t(T_{t+1} - T_t)$. Let t_1 and t_2 be such that $T_{t_1-1} < l \leq T_{t_1}$ and $T_{t_2} \leq u < T_{t_2+1}$. The amount of compulsory energy within the time interval $[l, u)$ is given below.

$$Q_{points}(l, u, CI_{\mathcal{I}}) = \sum_{\substack{\langle [a,b), w \rangle \in CI_{\mathcal{I}} \\ [a,b) \subseteq [l,u)}} w = Y_{t_1-1}(T_{t_1} - l) + \sum_{t=t_1}^{t_2} Z_t + Y_{t_2}(u - T_{t_2}) \quad (12)$$

Once the tasks are sorted, in $O(n \log n)$ time, by ect and lst, the vector Y' can be computed in linear time. The vectors Y, and Z can also be computed in linear time. The vector Z is preprocessed as a partial sum in linear time (see Section 3.1). Overall, the preprocess time is $O(n \log n)$.

To answer a query $Q_{points}(l, u, CI_{\mathcal{I}})$, a binary search finds the indices t_1 and t_2 in $O(\log n)$ time. Equation (12) is computed in constant time since the vector Z was preprocessed as a partial sum. Overall, the query time is $O(\log n)$.

5.2 Computing the free energy

We use our adaptation of range trees to answer the query $Q_{points}(l, u, FI_{\mathcal{I}})$. Since range trees process segments that are parallel to the x-axis, we use a geometric transformation to align the points in a set FC_i with the x-axis.

We transform an interval $[a, b) \in FI_{\mathcal{I}}$ into an interval $[a, a + b)$. The weights of the intervals remain unchanged. The intervals in $FI_{\mathcal{I}}$, when transformed, form the following weighted segments.

$$T^1_{\mathcal{I}} = \{\langle \mathrm{est}_i, \mathrm{est}_i + p_i - c_i, \mathrm{est}_i + \mathrm{lct}_i, h_i \rangle \mid i \in \mathcal{I}\}$$

We proceed to a second transformation where each interval $[a, b) \in FI_{\mathcal{I}}$ is transformed into $[-b, -a - b)$. The intervals from $FI_{\mathcal{I}}$ become the following weighted segment.

$$T^2_{\mathcal{I}} = \{\langle -\mathrm{lct}_i, -\mathrm{lct}_i + p_i - c_i, -\mathrm{est}_i - \mathrm{lct}_i, h_i \rangle \mid i \in \mathcal{I}\}$$

Figures 1c and 1d show the first and second transformations.

Lemma 1. $Q_{points}(l, u, FI_{\mathcal{I}}) = Q_{segments}(l, l + u, T^1_{\mathcal{I}}) + Q_{segments}(-u, -l - u - 1, T^2_{\mathcal{I}})$

Proof. Sketch: $Q_{segments}(l, l + u, T^1_{\mathcal{I}})$ computes the weights of the points inside the gray box and under the dotted line in Figure 1b or inside the gray box in Figure 1c. The query $Q_{segments}(-u, -l - u - 1, T^2_{\mathcal{I}})$ computes weights of the points inside the gray box and above the dotted line in Figure 1b or inside the gray box in Figure 1d.

From Lemma 1, it follows that two range trees can be constructed in $O(n \log n)$ time with the segments in $T^1_{\mathcal{I}}$ and $T^2_{\mathcal{I}}$. Computing the free energy within the interval $[l, u)$ is performed online in $O(\log n)$ time.

6 A Checker That Analyzes $O(n \log n)$ Time Intervals

We show that even though the energetic reasoning can fail in any of the $O(n^2)$ time intervals mentioned in Section 2, this number of intervals can be reduced to $O(n \log n)$ during the online computation. After analyzing a subset of $O(n \log n)$ intervals, it is safe to conclude whether the check passes for all $O(n^2)$ intervals.

We define the matrix $\mathbf{E}$ such that $\mathbf{E}[l, u]$ is the left-shift/right-shift energy contained in the interval $[l, u)$ for $\mathrm{est}_{\min} \leq l < u \leq \mathrm{lct}_{\max}$. If the interval $[l, u)$ is reversed ($l > u$), the left-shift/right-shift is null. The matrix $\mathbf{S}[l, u]$ contains the slack for the interval $[l, u)$, i.e. the remaining amount of energy in that interval.

$$\mathbf{E}[l, u] = \sum_{i \in \mathcal{I}} LSRS(i, l, u) \qquad \mathbf{S}[l, u] = C \cdot (u - l) - \mathbf{E}[l, u]$$

Theorem 1. *The matrix $\mathbf{E}$ is a Monge matrix.*

Proof. Let $\mathrm{est}_{\min} \leq l_1 < l_2 \leq \mathrm{lct}_{\max}$ and $\mathrm{est}_{\min} \leq u_1 < u_2 \leq \mathrm{lct}_{\max}$. The quantity $\mathbf{E}[l_2, u_2] - \mathbf{E}[l_2, u_1]$ is the amount of left-shift/right-shift energy that we gain by enlarging the interval $[l_2, u_1)$ to $[l_2, u_2)$. By analyzing (2), we deduce that the quantities $u - \mathrm{lst}_i$ and $u - l$ increase at the same rate when enlarging $[l_2, u_1)$ to $[l_2, u_2)$ than when enlarging $[l_1, u_1)$ to $[l_1, u_2)$. However, the terms $\mathrm{ect}_i - l$ and p_i might prevent the left-shift/right-shift to increase when the interval is enlarged. It turns out that $\mathrm{ect}_i - l$ increases and p_i remains constant as l decreases. Consequently, the increase of energy from $[l_1, u_1)$ to $[l_1, u_2)$ is less limited than when enlarging $[l_2, u_1)$ to $[l_2, u_2)$. Hence $\mathbf{E}[l_2, u_2] - \mathbf{E}[l_2, u_1] \leq \mathbf{E}[l_1, u_2] - \mathbf{E}[l_1, u_1]$. □

Corollary 1. *The matrix S is an inverse Monge matrix.*

Proof. Follows from $\mathbf{S}[l, u] = v[u] - v[l] - \mathbf{E}[l, u]$ where $v[i] = iC$, Theorem 1, Property 4, and Property 3. □

The energetic reasoning test fails if and only if there exists a non-empty interval $[l, u)$ such that $\mathbf{S}[l, u] < 0$. Inspired from [11], we design an algorithm that finds the smallest entry $\mathbf{S}[l, u]$ for any $l < u$ by checking only $O(n \log n)$ entries in $\mathbf{S}$. The algorithm assumes that the matrix $\mathbf{S}$ is not precomputed, but that any entry can be computed upon request. Since in Section 5, we show how to compute $\mathbf{S}[l, u]$ for any interval $[l, u)$ in $O(\log n)$ time, we obtain an algorithm with a running time complexity of $O(n \log^2 n)$. Moreover, we do not need to process the whole matrix $\mathbf{S}$ but submatrices containing a subset of rows and columns from $\mathbf{S}$. These submatrices contain all intervals of interest described in Section 2 and by Property 1, are inverse Monge matrices.

We need to execute Algorithm 1 twice to correctly apply the energetic reasoning rule. The first execution processes all intervals of the form $O_1 \times O_2 \cup \bigcup_{l \in O_1} O(l)$. To do so, we call Algorithm 1 with the parameters O_1, O_2, and $O' := O(0) = \{\mathrm{est}_i + \mathrm{lct}_i \mid$

Algorithm 1: MongeChecker(S, O_1, O_2, O')

$P_1 \leftarrow \emptyset$, $P_2 \leftarrow \{\langle[\min O_2, \max O_2], \min O_1\rangle\}$, $F \leftarrow \emptyset$;
for $l \in O_1 \setminus \{\min O_1\}$ *in increasing order* **do**
  $\langle[u_1, u_2], l^*\rangle \leftarrow \text{Top}(P_2)$;
1 **while** $S(l, u_2) \leq S(l^*, u_2)$ **do**
2 Pop(P_2);
  $\langle[u_1, u_2], l^*\rangle \leftarrow \text{Top}(P_2)$;
3 $b \leftarrow \max(\{u \mid u_1 \leq u \leq u_2 \wedge u \in O_2 \wedge S(l, u) < S(l^*, u)\} \cup \{-\infty\})$;
4 $c \leftarrow \max(\{u \mid b \leq u \leq \min(u_2, \text{succ}(b, O_2)) \wedge u + l \in O' \wedge S(l, u) < S(l^*, u)\} \cup \{-\infty\})$;
  $u \leftarrow \max(b, c)$;
  $u_l \leftarrow \min(\text{succ}(u, O_2), \text{succ}(u + l, O') - l)$;
  $u_{l^*} \leftarrow \min(\text{succ}(u, O_2), \text{succ}(u + l^*, O') - l^*)$;
5 $d \leftarrow \left\lfloor \frac{(u_l - u)(S(l^*,u)\cdot(u_{l^*} - u) - u(S(l,u_{l^*}) - S(l^*,u))) - (u_{l^*} - u)(S(l,u)(u_l - u) - u(S(l,u_l) - S(l,u)))}{(u_{l^*} - u)(S(l,u_l) - S(l,u)) - (u_l - u)(S(l^*,u_{l^*}) - S(l^*,u))} \right\rfloor$;
  if $d > \min(u_l, u_{l^*})$ **then** $d \leftarrow \max(b, c)$;
  if $d > l + 1$ **then** PushInterval(P_1, P_2, l, d) ;
for $\langle[\underline{u}, \overline{u}], l\rangle \in P_1 \cup P_2$ *in increasing order* **do**
  $u_1 \leftarrow \underline{u}$;
  if $S(l, u_1) < 0$ **then** $F \leftarrow F \cup \{[l, u_1)\}$;
  $u_3 \leftarrow \min\{u \in O_2 \mid u > u_1\}$;
  while $u_3 \leq \overline{u}$ **do**
    if $S(l, u_3) < 0$ **then** $F \leftarrow F \cup \{[l, u_3)\}$;
6 Find u_2 such that $u_2 + l \in O'$, $u_1 < u_2 < u_3$, and $S(l, u_2 - 1) \geq S(l, u_2) < S(l, u_2 + 1)$;
    if *such a* u_2 *exists and* $S(l, u_2) < 0$ **then** $F \leftarrow F \cup \{[l, u_2)\}$;
    $u_1 \leftarrow u_3$;
    $u_3 \leftarrow \min\{u \in O_2 \mid u > u_1\}$;
if $F = \emptyset$ **then return** (Success, $\emptyset$) **else return** (Fail, F) ;

$i \in \mathcal{I}\}$. Moreover, we pass the function $S := (l, u) \mapsto S(\mathcal{I}, l, u)$ that returns the slack in the interval $[l, u)$. The sets O_1, O_2, and O' contain $O(n)$ elements (see Section 2) and are computed in linear time. For the second execution, we execute the checker on the reversed problem, processing intervals of the form $O_1 \cup O' \times O_2$. Thus, the algorithm is called with $S := (l, u) \mapsto S(\mathcal{I}, -u, -l)$, $O_1 := \{-u \mid u \in O_2\}$, $O_2 := \{-l \mid l \in O_1\}$, and $O' := \{-(\text{est}_i + \text{lct}_i) \mid i \in \mathcal{I}\}$. If neither execution leads to a failure, the constraint is consistent according to the energetic reasoning rule.

Algorithm 1 is built around the data structure P that encodes the envelope of the inverse Monge matrix $\mathbf{S}[l, u]$. The algorithm proceeds in two phases. The first phase initializes the data structure P while the second phase uses it to perform the check.

Let P be a set of tuples such that $\langle[\underline{u}, \overline{u}], l\rangle \in P$ indicates that the smallest element on any column $u \in [\underline{u}, \overline{u}]$ occurs on row l (we ignore rows greater than or equal to u as they correspond to empty time intervals). The intervals $[\underline{u}, \overline{u}]$ in P are sorted, disjoint,

and contiguous. Upon the insertion of a tuple $\langle[\underline{u}, \overline{u}], l\rangle$, the intervals in P must be altered in order to be disjoint from $[\underline{u}, \overline{u}]$. Intervals in P that are nested in $[\underline{u}, \overline{u}]$ must be deleted from P. Intervals that partially overlap with $[\underline{u}, \overline{u}]$ must be shrunk. An interval that embeds $[\underline{u}, \overline{u}]$ needs to be split.

Consider the sequence of tuples $\langle[\underline{u}, \overline{u}], l'\rangle \in P$ sorted by intervals. By property of the envelope of an inverse Monge matrix, the rows l' in the sequence increase up to a maximum and then decrease. We store in a stack P_1 the first tuples of the sequence up to the tuple with the largest row (exclusively). We store in a stack P_2 the remaining tuple, i.e. the decreasing slice of the sequence. The ends of the sequence are at the bottom of the stacks, the tuple with the largest row l' is at the top of P_2 and the tuple before is at the top of P_1. Algorithm 2 details the process of inserting an interval in the data structure while maintaining the invariant. Lines 1-2 move intervals smaller than the current row l from P_2 to P_1. Lines 3-4 remove overlapping intervals in P_2. The remainder of the algorithm splits the top interval of P_2 and insert the new interval between it. A tuple is always pushed onto P_2 before being moved to P_1 and is never moved once in P_1. Since Algorithm 2 pushes two tuples onto P_2, it has a constant time amortized complexity.

The intervals inserted into P are computed as follows. First, all columns of $\mathbf{S}$ are associated to the first row $\min O_1$. Therefore, P is initialized to $\{\langle[\mathrm{est}_{\min}, \mathrm{lct}_{\max}], \min O_1\rangle\}$. We process the next rows in increasing order in the first for loop. Each time we process a row, we update the envelope function l^* encoded with the data structure P. Let $f_l(x) = S(\mathcal{I}, l, x)$ and $f_{l^*}(x) = S(\mathcal{I}, l^*(x), x)$ be two functions. Because $\mathbf{S}$ is an inverse Monge matrix, we know that these two functions intersect at most once. We search for the greatest value d where $f_l(d) < f_{l^*}(d)$. Once the value d is computed for a row l, if $d > l + 1$, we insert the tuple $\langle[l + 1, d], l\rangle$ in P. If $d \leq l + 1$, the functions do not intersect or intersect on an empty interval $[l, d)$ which is not of interest.

We compute d by proceeding in four steps. The while loop on line 1 searches the tuple $\langle[\underline{u}, \overline{u}], l^*\rangle$ in P_2 such that $f_l(x)$ and $f_{l^*}(x)$ intersect in $[\underline{u}, \overline{u}]$. This tuple can not be in P_1 because l is greater than all intervals in P_1 and we want d to be greater than l. The intervals in which $f_l(x)$ is smaller than the functions of the previous rows are removed from P on line 2. On line 3, we perform a binary search over the elements of O_2 within $[\underline{u}, \overline{u}]$ to find the greatest column $b \in O_2$ for which $f_l(b) < f_{l^*}(b)$. Let $\mathrm{succ}(a, A) = \min\{a' \in A \mid a' > a)$ be the successor of $a \in A$ when A is sorted in increasing order. Once b is found, we narrow the search for the intersection of the functions $f_l(x)$ and $f_{l^*}(x)$ to the interval $[b, \min(u_2, \mathrm{succ}(b, O_2))]$. On line 4 we find the greatest column $c \in O(l)$ that lies in $[b, \min(u_2, \mathrm{succ}(b, O_2))]$ where $f_l(c) < f_{l^*}(c)$. In order not to compute $O(l)$ for each row l, we perform the search in $O' = O(0)$. We have that $c \in O(l)$ if and only if $c + l \in O'$. Therefore, rather than searching for the greatest $c \in O(l)$, line 4 searches for the greatest $c + l \in O'$.

Using the values b and c, we find the value d where $f_l(x)$ and $f_{l^*}(x)$ intersect. The function $f_l(x)$ is piecewise linear with inflection points in $O_2 \cup O(l)$. The function $f_{l^*}(x)$ is piecewise linear with inflections points in $O_2 \cup O(l^*)$. Let $\bar{d}_l = \mathrm{succ}(\max(b, c), O_2 \cup O(l))$ and $\bar{d}_{l^*} = \mathrm{succ}(\max(b, c), O_2 \cup O(l^*(\max(b, c)))$. We know that $\max(b, c) \leq d \leq \min(\bar{d}_l, \bar{d}_{l^*})$, that $f_l(x)$ is linear over the interval $[\max(b, c), \bar{d}_l)$, and that $f_{l^*}(x)$ is linear over $[\max(b, c), \bar{d}_{l^*})$. We let d be the intersection point of these segments, or let

Algorithm 2: PushInterval(P_1, P_2, l, d)

$\langle [\underline{u}, \overline{u}], l^* \rangle \leftarrow$ Top(P_2);
1 **while** $l + 1 > \overline{u}$ **do**
    Push(P_1, Pop(P_2));
2   $\langle [\underline{u}, \overline{u}], l^* \rangle \leftarrow$ Top(P_2);
if $d > \overline{u}$ **then**
3   $u' \leftarrow \underline{u}$;
    $l' \leftarrow l^*$;
    while $d > \overline{u}$ **do**
        Pop(P_2);
4       $\langle [\underline{u}, \overline{u}], l^* \rangle \leftarrow$ Top(P_2);
    Push($P_1, \langle [u', l], l' \rangle$);
else
    $\langle [\underline{u}, \overline{u}], l^* \rangle \leftarrow$ Pop(P_2);
    Push($P_1, \langle [\underline{u}, l], l^* \rangle$);
Push($P_2, \langle [d + 1, \overline{u}], l^* \rangle$);
Push($P_2, \langle [l + 1, d], l \rangle$);

$d = \max(b, c)$ if that intersection point does not satisfy $\max(b, c) \leq d \leq \min(\bar{d}_l, \bar{d}_{l^*})$. Once d is computed, we insert $\langle [l + 1, d], l \rangle$ into P.

In the second part of the algorithm, we iterate on each tuple $\langle [a, d], l \rangle \in P$ found in the first part. The while loop processes the columns in O_2 that are within the current interval. After checking that the slack of two consecutive columns in O_2, u_1 and u_3, does not yield a negative slack, we try to find a column $u_2 \in O(l)$ such that u_2 is between u_1 and u_3 and has a negative slack.

Derrien and Petit [7] showed that the slope of the slack increases at elements in $O(l)$. Therefore, there is a global minimum between u_1 and u_3 that can be found using a binary search. If $\mathbf{S}[l, t] < \mathbf{S}[l, t + 1]$, the global minimum is before t. Otherwise, it is at t or after t. Hence, the binary search finds the element in $O(l)$ between u_1 and u_3 where the slope of the slack shift from negative to positive. If all columns in each interval are processed without causing a failure, we return success.

Lines 3, 4, and 6 are executed $O(n)$ times and perform binary searches over $O(n)$ columns of the matrix leading to $O(n \log n)$ comparisons. Each comparison requires the computation of two entries of the matrix $\mathbf{S}$ which is done in $O(\log n)$ times (see Section 5). This leads in an $O(n \log^2 n)$ running time complexity. The space complexity of the algorithm is dominated by the complexity of the range-trees, which is $O(n \log n)$.

7 Filtering algorithm

We present a filtering algorithm for the energetic reasoning with average complexity of $O(n^2 \log n)$ based on the checker we presented in Section 6 and inspired by Derrien and Petit's filtering algorithm [7]. We only show how to filter the est of the tasks. To filter the lct, one can create the reversed problem by multiplying by -1 the values in the domains.

Algorithm 3: MongeFilter($\mathcal{I}$)

$\text{est}'_i \leftarrow \text{est}_i \; \forall i \in \mathcal{I}$;
for $i \in \mathcal{I}$ **do**
  $S_i^1 \leftarrow (l,u) \mapsto S(\mathcal{I},l,u) + LSRS(i,l,u) - LS(i,l,u)$;
  $S_i^2 \leftarrow (l,u) \mapsto S(\mathcal{I},-u,-l) + LSRS(i,-u,-l) - LS(i,-u,-l)$;
  $(r_1,F_1) \leftarrow$ MongeChecker$(S_i^1, O_1, O_2, \{\text{est}_i + \text{lct}_i \mid i \in \mathcal{I}\})$;
  $(r_2,F_2) \leftarrow$ MongeChecker$(S_i^2, \{-u \mid u \in O_2\}, \{-l \mid l \in O_1\}, \{-(\text{est}_i + \text{lct}_i) \mid i \in \mathcal{I}\})$;
1 **for** $[l,u) \in F_1 \cup \{[l,u) \mid [-u,-l) \in F_2\}$ **do**
    $\text{est}'_i \leftarrow \min(\text{est}'_i, \lceil u - \frac{S(\mathcal{I},l,u)+LSRS(i,l,u)}{h_i} \rceil)$

Filtering the est in the reversed problem filters the lct in the original problem. We define the function $S_i^1(l,u)$ to be the amount of slack in the interval $[l,u)$ if i was assigned to its earliest starting time, i.e. $S_i^1(l,u) = S(\mathcal{I},l,u) + LSRS(i,l,u) - LS(i,l,u)$. Similarly, $S_i^2(l,u) = S_i^1(-u,-l)$ represents the same concept on the reversed problem.

Algorithm 3 filters each task i by running the checker presented in Section 6 with S_i^1 and S_i^2 rather than S. We execute Algorithm 1 twice for each task. The first execution handles intervals of the form $O_1 \times O_2 \cup \bigcup_{l \in O_1} O(l)$ and the second, intervals of the form $O_1 \cup \bigcup_{u \in O_2} O(u) \times O_2$. On line 1, for each interval $[l,u)$ whose slack is negative when i starts at its earliest starting time, we filter est_i using the energetic reasoning rule.

Algorithm 3 reaches the same fix point as [7]. A task i needs to be filtered if $S(\mathcal{I} \setminus \{i\},l,u) - LS(i,l,u)$ is negative on an interval $[l,u)$. If such intervals exist, our algorithm finds at least one since it necessarily processes the interval with the minimum value. If not all negative intervals are found or if the filtering creates new negative intervals, our algorithm processes them at the next iteration.

7.1 Running time analysis

Algorithm 3 makes $2n$ calls to the checker that each makes $O(n \log n)$ slack queries answered in $O(\log n)$ time, hence a running time in $O(n^2 \log^2 n)$. However, we use a memorization based on virtual initialization [13] with a space complexity of $O(\max_i \text{lct}_i^2)$. A hash table could also work with $O(n \log n)$ space. When $S(\mathcal{I},l,u)$ is computed, we store its value so that further identical queries get answered in $O(1)$. There are $O(n^2)$ intervals of interest. On line 5 of Algorithm 1, the slack for intervals that are not of interest is computed $O(n)$ times in the checker, hence $O(n^2)$ in the filtering algorithm. For these $O(n^2)$ intervals, a total of $O(n^2 \log n)$ time is spent thanks to memorization.

Line 4 performs a binary search over the elements $u \in O(l) \cap [b, \text{succ}(b,O_2)]$ and computes the slack $S(l^*,u)$ which might not be an interval of interest. The binary search could perform up to $O(\log n)$ slack evaluations and therefore lead to $O(n^2 \log n)$ distinct evaluations in the filtering algorithm. However, having n elements in $O(l)$ that occurs between two consecutive elements in O_2 (namely b and $\text{succ}(b,O_2)$) seldomly happens. Suppose that $|O_2| = 3n$ and $|O(l)| = n$ and that the time points in these sets are evenly spread. We obtain an average of $\frac{1}{3}$ elements in $O(l)$ between two consecutive elements in O_2. This number can vary depending on the cardinalities of O_2 and $O(t)$,

but in practice, we rather observe an average of 0.141 elements when $n = 16$ and this number decreases as n increases. To reach the worst time bound of $O(n^2 \log^2 n)$, one would need to construct an instance that triggers $O(n^2)$ binary searches on line 4, each time over $O(n)$ elements. We did not succeed to construct such an instance. Assuming that the number of elements in the search is bounded, in average, by a constant, we obtain an average running time of $O(n^2 \log n)$. Under these assumptions, the binary search might as well be substituted by a linear search.

7.2 Optimization

As Derrien and Petit in [7], we improve the practical performance of Algorithm 3 by processing, for each task i, only intervals in O_C and L_i. We partition O_C into two sets: O_C^1 contains the intervals in O_C with lower bound in O_1 and O_C^2 contains the intervals in O_C with lower bound in $\bigcup_{u \in O_2} O(u)$. Let $lb^s(A) = \{s \cdot a \mid [a, b) \in A\}$ and $ub^s(A) = \{s \cdot b \mid [a, b) \in A\}$ be the lower bound and upper bounds of the intervals in A, multiplied by s. We execute Algorithm 3 as usual, but the first call to the checker is done with parameters $O_1 := lb^1(O_C^1)$ and $O_2 := up^1(O_C^1)$ while the second call is made with parameters $O_1 := ub^{-1}(O_C^2)$ and $O_2 := lb^{-1}(O_C^2)$. The set O' is empty and S is unchanged. Since we do not process all elements in O_2, we can't use a binary search at line 6 of Algorithm 1 anymore and we must search linearly leading to a worse case complexity of $O(n^3 \log n)$ when there are $O(n^2)$ upper bounds in O_C. However, this seldom happens and the reduced constant outweighs the increased complexity.

To process intervals in L_i for $i \in \mathcal{I}$, we do as Derrien and Petit [7] by applying the filtering rule on each of the $O(n)$ intervals in L_i. Since we compute the slack in $O(\log n)$, we obtain a running time of $O(n^2 \log n)$ for all tasks.

8 Experiments

We implemented the algorithms in the solver Choco 4 [16] with Java 8. We ran the experiments on an Intel Core i7-2600 3.40GHz. We used both BL [2] and PSPLIB [12] benchmarks of the Resources Constrained Project Scheduling Problems (RCPSP). An instance consists of tasks, subject to precedences, that need to be simultaneously executed on renewable resources of different capacities. We model this problem using one starting time variable S_i for each task i and a makespan variable. We add a constraint of the form $S_i + p_i \leq S_i$ to model a precedence and use one CUMULATIVE constraint per resource. For these constraints, we do not use other filtering algorithms than the algorithms being tested. We minimize the makespan. All experiments are performed using the Conflict Ordering Search [9] search strategy with a time limit of 20 minutes.

Figure 2 shows the time to optimally solve instances of benchmark BL using the checker from Section 6 (on the y axis) and Derrien and Petit's checker [7] (on the x axis). The Monge Checker is faster for most instances and takes, in average, 77% of the time required by Derrien and Petit's checker to solve an instance to optimality.

Figure 3 shows the time to optimally solve instances using the filtering algorithm from Section 7 and Derrien and Petit's filtering algorithm [7], for BL (left) and PSPLIB (right) benchmarks. While our algorithm is only marginally faster on BL instances,

Benchmark	n	instances solved	% time
BL	20	20	0.92
BL	25	20	0.89
PSPLIB	30	445	0.94
PSPLIB	60	371	0.57
PSPLIB	90	369	0.44
PSPLIB	120	197	0.48

Table 1: Percentage of the time taken by Monge Filter to optimally solve instances of n tasks, compared to Derrien and Petit's filtering algorithm.

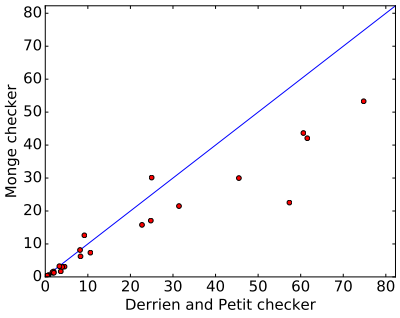

Fig. 2: Comparison of the time to optimally solve instances of the benchmark BL. Derrien et al.'s checker vs the Monge Checker.

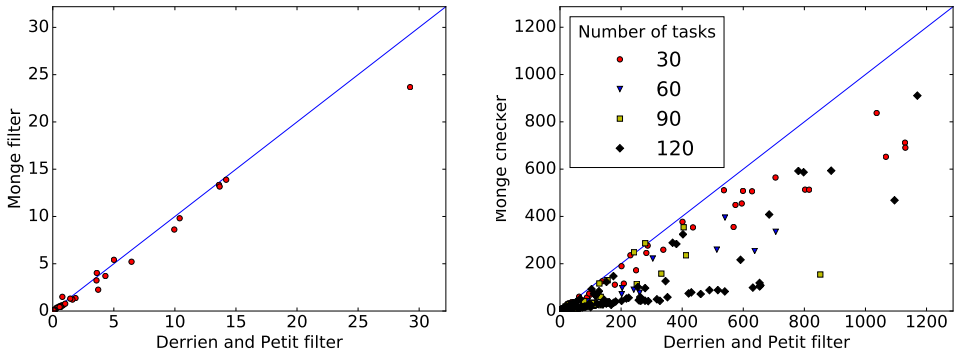

Fig. 3: Comparison of the time to solve to optimality instances of BL (left) and PSPLIB (right) benchmarks for Derrien and Petit's filtering algorithm and Monge Filter.

where the number of tasks is small (between 20 and 25), the difference significantly increases as the number of tasks increases, as shown in Table 1. This shows the impact of decreasing the complexity of the energetic reasoning from $O(n^3)$ to $O(n^2 \log n)$.

9 Conclusion

We introduced a new method to explicitly process only $O(n \log n)$ intervals for the energetic reasoning using Monge matrices. We showed how to compute the energy in an interval in $O(\log n)$. We proposed a checker in $O(n \log^2 n)$ and a filtering algorithm in $O(n^2 \log n)$. Experiments showed that these algorithms are faster in theory and in practice. Future work will focus on extending these algorithms to produce explanations.

Acknowledgment: In memory of Alejandro López-Ortiz (1967 – 2017) who introduced me to research, algorithm design, and even Monge matrices. – C.-G. Q

References

1. A. Aggoun and N. Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and computer modelling*, 17(7):57–73, 1993.
2. P. Baptiste and C. Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1-2):119–139, 2000.
3. P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling*. Kluwer Academic Publishers, 2001.
4. N. Beldiceanu and M. Carlsson. A new multi-resource cumulatives constraint with negative heights. *CP*, 2470:63–79, 2002.
5. N. Bonifas. A $O(n^2 \log(n))$ propagation for the energy reasoning. In *ROADEF 2016*, 2016.
6. M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.
7. A. Derrien and T. Petit. A new characterization of relevant intervals for energetic reasoning. pages 289–297, 2014.
8. H. Fahimi and C. G. Quimper. Linear-time filtering algorithms for the disjunctive constraint. In *AAAI*, pages 2637–2643, 2014.
9. S. Gay, R. Hartert, C. Lecoutre, and P. Schaus. Conflict ordering search for scheduling problems. In *International conference on principles and practice of constraint programming*, pages 140–148. Springer, 2015.
10. R. Kameugne and L. P. Fotso. A cumulative not-first/not-last filtering algorithm in $O(n^2 log(n))$. *Indian Journal of Pure and Applied Mathematics*, 44(1):95–115, 2013.
11. H. Kaplan, S. Mozes, Y. Nussbaum, and M. Sharir. Submatrix maximum queries in monge matrices and partial monge matrices, and their applications. *ACM Transactions on Algorithms (TALG)*, 13(2), 2017.
12. R. Kolisch and A. Sprecher. Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program. *European journal of operational research*, 96(1):205–216, 1997.
13. A. Levitin. *Introduction to the design & analysis of algorithms*. Pearson Education, Inc, Boston, 3rd ed. edition, 2012.
14. P. Lopez and P. Esquirol. Consistency enforcing in scheduling: A general formulation based on energetic reasoning. In *5th International Workshop on Project Management and Scheduling (PMS'96)*, 1996.
15. L. Mercier and P. Van Hentenryck. Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20(1):143–153, 2008.
16. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC,LS2N, CNRS UMR 6241 and COSLING S.A.S., 2017.
17. A. Schutt, , and A. Wolf. A new $O(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 445–459. Springer, 2010.
18. A. Schutt, A. Wolf, and G. Schrader. Not-first and not-last detection for cumulative scheduling in $O(n^3 \log n)$. In *INAP*, pages 66–80. Springer, 2005.
19. S. Sethumadhavan. A survey of monge properties. Master's thesis, Cochin University of Science and Technology, 2009.
20. A. Tesch. Exact energetic reasoning in $O(n^2 \log^2 n)$. Technical report, Zuse Institute Berlin, 2016.
21. A. Tesch. A nearly exact propagation algorithm for energetic reasoning in $O(n^2 \log n)$. In *Proceedings of the 22th International Conference of Principles and Practice of Constraint Programming (CP 2016)*, pages 493–519, 2016.

22. P. Vilım. Timetable edge finding filtering algorithm for discrete cumulative resources. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 6697:230–245, 2011.
23. A. Wolf and G. Schrader. $O(n \log n)$ overload checking for the cumulative constraint and its application. In *INAP*, volume 4369, pages 88–101. Springer, 2005.