ORIGINAL PAPER

Polygon stacks and time reference conversions

Christoph Lenzen[1] · Rüdiger Klaehn[2] · Sven Prüfer[1] · Maria Theresia Wörle[1]

Abstract

This paper describes how a time-based planning system, which supports resource constraints, may be extended such that a resource constraint interval does not have to refer to the start- or end-time of the underlying activity but to any linear combination thereof, such as the middle. This way, an activity with multiple resource constraints referring to different time intervals no longer has to be split into sub-activities, which may simplify the planning model and the algorithm. To be able to describe the necessary transformations, we introduce the concept of PolygonStacks and describe the operations which a typical planning engine requires to intersect the sets of consistent timeline entries of all constraints defined on an activity. We then introduce Sliders and Offsets, which allow specifying the constraint intervals in a more generic way as supported in current planning models. Based on this preparation, we can derive two lemmas, which provide the conversions required by Sliders and Offsets. We continue with several conversion examples and point out how to solve the issues which will occur during implementation. A short sketch of the complexity of our current implementation demonstrates that further work on performance should be considered, even though in practice we observe that the bottleneck of calculation remains within profile calculation rather than PolygonStack operations.

1 Introduction

Within the new planning library *Plains*, which is developed at the German Space Operations Center (GSOC), timelines are defined over the rational numbers to avoid introducing rounding issues due to a fix grid size or the use of IEEE floating point numbers. To make this possible, we are using the high-performance rational number implementation of the *Spire* algebra library, see [1]. One task of *Plains* is to determine the set of possible timeline entries of an activity from which the most suitable timeline entry may be chosen to be added to the timeline. In this way, *Plains* resembles other planning systems like most of those mentioned in [7]: APSI in [10], ASPEN in [6], EUROPA from [3], flexplan in [12, 13], Mexar2 from [5], MUSE in [16], Pinta/Plato from [8, 18] and SPIKE in [15, 22, ch.14]. As each timeline entry consists of a *start-time* and an *end-time*, timeline entries are

in fact points in the two-dimensional plane with *start-time* mapped to the *x-axis* and *duration = end-time − start-time* mapped to the *y-axis*. Selecting a good data structure to represent such sets of timeline entries therefore, is very important.

In the first section of this paper, we present the Polygon-Stack, which is used in *Plains* to efficiently represent such sets of timeline entries. The PolygonStack supports efficient Boolean operations to allow combining the impacts of multiple constraints, it supports checking whether a timeline entry belongs to it and it supports selecting an element according to a proper criterion. A useful property of the PolygonStack is that it uses a compact representation with a canonical choice, i.e. there is a preferable way to represent a given set of timeline entries. This has advantages for determinism of the calculation as well as the ability to speed up complex constraint computations using the process of memoization.

Contrary to existing concepts for representing two-dimensional polygons, a solution set for planning must be able to efficiently represent point solutions (an activity can be planned at exactly one point in time and duration), line solutions (e.g., an activity can be scheduled in a certain time range, but only with a fixed duration), area solutions (e.g.,

✉ Christoph Lenzen
  christoph.lenzen@dlr.de

1 DLR e.V. - German Aerospace Center, Münchener Straße 20, 82234 Weßling, Germany

2 Actyx AG, Ridlerstr. 31B, 80339 Munich, Germany

an activity can be scheduled within a time range and a duration range) or any arbitrarily complex combination thereof.

Following the approach of [20, ch. 5.1] (see also [23]), we associate with every point in the solution space a positive integer value. Set membership can, thus, be easily expressed using a convention like 0 for true and ≥ 1 for false. Additionally, having an integer value at each point in the solution space gives us more options to combine solution sets and simplifies implementing Boolean operations. Another benefit of [20 ch. 5.1], compared to most other existing concepts of polygons, such as [4, 11, 14, 24], is that we do not consider sorted sets of edges or vertices but just a non-sorted collection of rays. In contrast to [20, ch. 5.1], however, we need to distinguish between values within the inner part of a polygon and the values at its edges and vertices.

In the second section, we introduce the concept of *Sliders* and *Offsets*, which extends the capabilities of common planning modeling languages by allowing to formulate a constraint not only relative to the *start-time* or the *end-time* of a timeline entry but also relative to any time in between or even outside the timeline entry. Benefit of this new feature is that we may avoid splitting an activity into sub-activities in case multiple constraints refer to different time ranges relative to the activity. For example, a radar image acquisition may require power during its image observation phase; the on-board data compression mechanism, however, may block the on-board memory for twice the time of the image acquisition duration. With our new approach, both constraints may be defined on the same activity, even if the activity's duration is variable.

The downside of this new feature is that it creates additional work as for each constraint we need to convert the set of available timeline entries into the set of constraint interval *start-times* and *end-times* before we can apply the constraint restriction and thereafter convert back the result. The main part of this section, therefore, presents the generic formulas required to implement this conversion.

To clarify how these formulas may be applied, we present the implementation on PolygonStacks. Using selected examples for the conversion, we highlight what problems occur and how these can be solved. Thereafter, we present a worst case estimation for the complexity of the PolygonStack operations.

2 Polygon stacks

A timeline entry comprises a *start-time* and a *duration*, both of which shall take rational values. Therefore, one can identify a timeline entry with a rational point in the upper half plane $\mathbb{H}$, where the x-axis corresponds to the *start-time* and the y-axis to the *duration* of a timeline entry implying $y \geq 0$. To represent a set S of rational

timeline entries, we use functions $p : \mathbb{H} \cap \mathbb{Q} \times \mathbb{Q} \longrightarrow \mathbb{N}_0$ such that

$$E \in S \iff p(E) = 0.$$

For our purpose, we can further restrict to the future of some base time $X_0 \in \mathbb{Q}$ yielding a domain $D \subset \mathbb{H} \cap \mathbb{Q} \times \mathbb{Q}$ defined by

$$(x, y) \in D \iff \quad x \in \mathbb{Q}, \text{ s.t. } x \geq X_0 \text{ and} \tag{1}$$

$$y \in \mathbb{Q}, \text{ s.t. } y \geq 0. \tag{2}$$

We define functions $p : D \longrightarrow \mathbb{N}_0$, called *PolygonStack*, by finite sums

$$p = \sum_{r : \text{Ray}} r,$$

where each *ray* $r = \text{Ray}(x_r, y_r, s_r, d_{\text{At},r}, d_{\text{Above},r})$ with

$$x_r \in \mathbb{Q} \text{ (Time)} = \text{ time of } r\text{'s starting point}$$
$$y_r \in \mathbb{Q}_{\geq 0} \text{ (Duration)} = \text{ duration of } r\text{'s starting point}$$
$$s_r \in \mathbb{Q} \cup \{\infty\} = \text{ slope of } r$$
$$d_{\text{At},r} \in \mathbb{Z} = \text{ addend on } r\text{'s half line}$$
$$d_{\text{Above},r} \in \mathbb{Z} = \text{ addend above } r\text{'s half line}$$

defines a function $r : D \longrightarrow \mathbb{Q}$ as follows:

- If $s_r = \infty$, for any $E = (x, y) \in D$

$$r(E) = \begin{cases} d_{\text{At},r} & E = (x_r, y_r) \\ d_{\text{Above},r} & E = (x_r, y) \text{ with } y > y_r \\ 0 & \text{otherwise} \end{cases}$$

- If $s_r \in \mathbb{Q}$, for any $E = (x, y) \in D$

$$r(E) = \begin{cases} d_{\text{At},r} & x > x_r \text{ and } y_r + s_r(x - x_r) = y \\ d_{\text{Above},r} & x > x_r \text{ and } y_r + s_r(x - x_r) < y \\ 0 & \text{otherwise} \end{cases}$$

The difference to boost's version of the PolygonStack (see [20, ch. 5.1]) is that we distinguish in between d_{At} and d_{Above} and that we support a ray with infinite slope. This way we can specify different values for vertices, edges and inner parts of two-dimensional polygons.

2.1 Examples

Figure 1 shows a PolygonStack consisting of one ray with finite slope and one ray with infinite slope. Figure 2 shows a PolygonStack with the shape of a parallelogram.
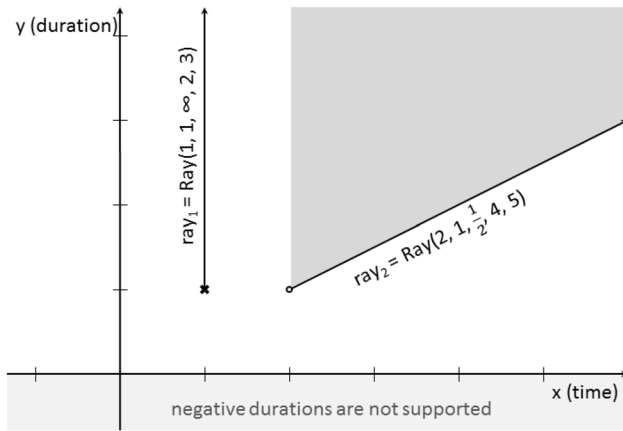
Fig. 1 Rays. $\mathrm{ray}_1(1,1) = 2$, $\mathrm{ray}_1(1,y) = 3$ for $y > 1$, $\mathrm{ray}_2(x,y) = 4$ for $y = 1 + \frac{1}{2}(x-2), x > 2$, $\mathrm{ray}_2(x,y) = 5$ for $y > 1 + \frac{1}{2}(x-2), x > 2$
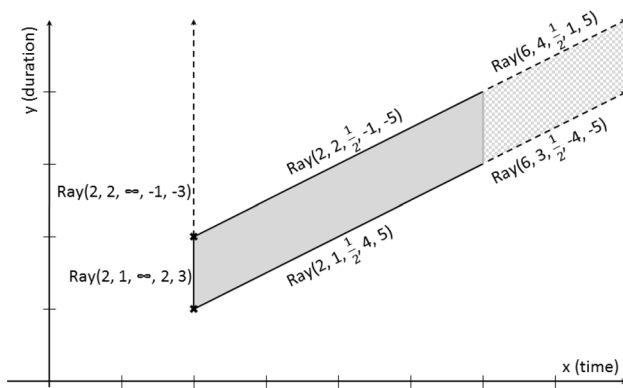

Fig. 2 Polygon stack forming a parallelogram: $(2,1) \to 2$, $(2,2) \to 3 - 1 = 2$, left vertical edge $\to 3$, inner part and right vertical edge $\to 5$, lower edge and its right endpoint $\to 4$, upper edge and its right endpoint $\to 5 - 1 = 4$, dashed lines, dotted and white region $\to 0$

2.2 Operations

To evaluate this representation of a PolygonStack as a set of rays, we need to describe our use cases.

2.2.1 Planning horizon

To simplify some calculations, we will need PolygonStacks which have value 1 outside a bounded region. For this, we introduce the *ambient* value which is a constant $C \in \mathbb{N}_0$ added to all points, denoted by PolygonStack(C, {Rays}), e.g.,

$$P = \text{PolygonStack}(1, \{\text{Ray}(0,0,0,-1,-1)\})$$

denotes the (non-bounded) polygon stack with corresponding function $D \longrightarrow \mathbb{N}_0$

$$\begin{cases} 0 \text{ if } x > 0 \text{ and } y \geq 0 \\ 1 \text{ otherwise.} \end{cases}$$

2.2.2 Addition

As mentioned above, we need to consider multiple constraints, each of which restricts the set of consistent timeline entries. Such a restriction shall be represented by a PolygonStack with value greater than zero for all timeline entries which have a conflict with respect to this constraint. The PolygonStacks of all constraints, therefore, need to be added to specify all conflict free timeline entries as those with value 0. Adding two PolygonStacks requires summing up the ambient values, collecting the rays of the two PolygonStacks and merging rays of same x, y and *slope* to one ray with d_{At} and d_{Above} set to the sum of the respective values.

2.2.3 Evaluation

To evaluate a PolygonStack P on a timeline entry $E = (x,y)$, we need to consider all rays $r = \text{Ray}(x_r, y_r, s_r, d_{\mathrm{At},r}, d_{\mathrm{Above},r})$ of P with

$$s_r \in \mathbb{Q}, x_r < x, y_r + s_r(x - x_r) \leq y$$

or $s_r = \infty, x_r = x, y_r \leq y$

and add the respective d_{At} or d_{Above}. Geometrically, this corresponds to rays whose line lies at or below the timeline entry E.

2.2.4 Scan for value

When asking which timeline entries an activity may be given, we are not interested in the precise set of violated constraints but only in the region where timeline entries are conflict free. We, therefore, provide a function to simplify a PolygonStack such that it only takes values in $\{0,1\}$. More precisely: given a PolygonStack P, we derive a PolygonStack Q with values

$$Q(E) = 1 \iff P(E) > 0,$$
$$Q(E) = 0 \iff P(E) = 0.$$

For this task, we first need to determine all vertical *scan-lines* $x_1 < \cdots < x_n$ where either a ray starts or two rays cross. For all *scan-lines* x_i, we then need to consider all rays with finite slope starting before x_i and all rays with infinite slope starting at x_i. For all open intervals (x_i, x_{i+1}), we need to consider all rays with finite slope starting before x_i. For each ray $r = \text{Ray}(x_r, y_r, s_r, \_, \_)$ let $y(r,x) = x_r + s_r(x - x_r)$ denote the y value of the crossing point of r's line and the *scan-line*
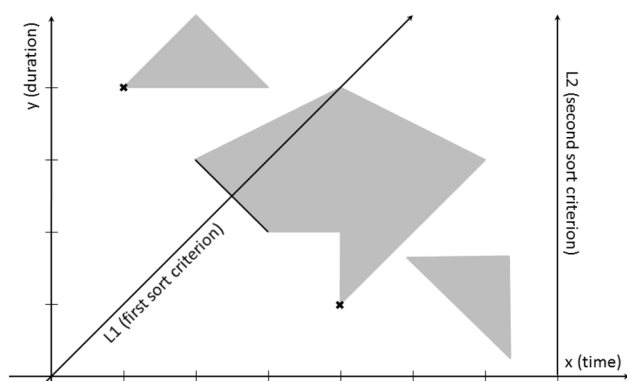
Fig. 3 Sort criterion for elements of PolygonStack; L_1: black marked points are best on P; L_2: (4, 1) is the best among the black marked points

x. Traversing the rays ascending in $y(r, x)$ and s_r, one can derive the regions of values within the considered x-axis interval and build up the resulting PolygonStack Q. Note that multiple rays with equal $y(r, x)$ and s_r can be considered as effectively one ray by summing up their d_{At} and d_{Above}. Rays belonging to the same line in this way occur quite often, because most rays need to be canceled at a later x, e.g., for bounded PolygonStacks. When implementing this operation, one can significantly improve performance by considering the hints provided in [20, ch. 5.1].

2.2.5 Find value

From now on, we consider a PolygonStack P as synonymous with the set of points E with $P(E) = 0$. Also, we emphasize again that our PolygonStacks are assumed to be finite.

Since a PolygonStack represents a set of timeline entries satisfying some constraints, we would like to select a specific timeline entry from within that set by some configurable algorithm. We choose to define an ordering by specifying a directed line L_1. Two points in the plane are compared by comparing their perpendicular projections on L_1. The points which are mapped to the first point on L_1 w.r.t. its direction are considered *best* points. Since this step might result in a set of best points, we require a second directed line L_2, which is not parallel to L_1, yielding a unique result for any non-empty, bounded, closed PolygonStack, see Fig. 3.

In Fig. 3, L_1 corresponds to *earliest end-time* (i.e., smallest sum of *start-time* and *duration*) and L_2 corresponds to *minimum duration*. Note that the best point does not need to exist in case the PolygonStack has excluded edges or vertices or is unbounded. In this case, the algorithm must choose a value within the PolygonStack which is close to the best point of the closure of P. For the following lemma, we restrict, thus, to closed PolygonStacks.

Lemma 1 *For closed PolygonStack s, the best point is either the starting point of a ray or the crossing point of two rays.*

Proof For a point E which is neither part of a *scan-line* nor part of a ray's line, there exists a (2-dimensional) open neighborhood contained in the PolygonStack with the same value, which, when perpendicularly mapped to L_1, is mapped to a (1-dimensional) open neighborhood of the image of E, which means that E is not best w.r.t. L_1. For a point E, which is on a *scan-line* or on a ray's line and which is not a starting point of a ray or a crossing point of two rays, there exists an open neighborhood on the respective line with the same value. If this neighborhood is mapped to a neighborhood of the image of E, E is again not best w.r.t. L_1. If the neighborhood of E is not mapped to a neighborhood of E, it must be mapped to the same value as E, which means that all points of the neighborhood of E are equally good w.r.t. L_1. In this case, L_2 applies and this time the neighborhood must be mapped to a neighborhood of E, because L_2 is not parallel to L_1, and again E is not a best point. $\square$

We, therefore, can restrict the search for the best point to the finite set of starting points and crossing points.

3 Polygon Stack Conversion

3.1 Modeling example

To understand how this data structure shall be applied, we consider an example for which our planning library shall be suitable: an on-ground fire detecting satellite shall perform observations according to on-ground defined target regions. The corresponding on-board procedure might be defined as follows:

1. 10 s before image acquisition, the ACS (Attitude Control System) starts acquiring the required start position.
2. Thereafter, image acquisition is performed. The duration of the image acquisition depends on the target region and, therefore, is variable.
3. Thereafter, image analysis is performed to detect hot spots within the image. The duration of the image analysis is proportional to the duration of the image acquisition, since image analysis is proportional to the amount of data. In this example, we assume that image analysis and image acquisition have same duration.

In our model, we want to define an activity *detectFire*, which starts at image acquisition start and ends at image analysis end. For this activity, we want to formulate the following constraints:

1. during the interval $\left[start - 10sec, start + \frac{1}{2}duration\right]$ the ACS system is allocated, i.e., no other ACS relevant activities may be performed

2. during the interval [*start*, *end*] the memory unit is allocated, i.e., no other memory relevant activities may be performed

With our approach of Sliders and Offsets, we can directly formulate both constraints. In traditional planning systems (see [21]), we only have the possibility to define constraints of types

1. *at start* the condition must hold at the activity's start-time (possibly plus some predefined constant offset),

2. *at end* the condition must hold at the activity's end-time (possibly plus some predefined constant offset),

3. *over all* the condition must hold during the activity's time interval (possibly adapted by predefined constant offsets) or

4. *in ι dur δ* the condition must hold for some subinterval of duration δ within a given interval ι, where the interval bounds of ι may refer to timeline entry start or end plus some constant offset.

Even with the concept of relative ordering ([21], section 5.1) we cannot directly refer to a linear combination of *start-time* and *end-time*, which means that in traditional planning systems we have to define at least two activities: one representing the *ACS activities* and one representing the *memory allocation*, including the *image acquisition* and the *image analysis*. These two activities need to be coupled by the information

> *ACS activities* last 10 seconds and end in the middle of a *memory allocation* start.

Blowing up the model to be able to formulate proper constraints is rather awkward, the main drawback about this solution, however, concerns the algorithm: To begin with, as no constraint may refer to *the middle*, this information must be considered by the algorithm, violating the concept of separation of algorithm and constraint modeling. Above all, however, heuristic reasoning about the valid timeline entries is much easier if all constraints are defined on one activity, because you can intersect the valid timeline entries of each constraint of the activity. If you need to consider multiple coupled activities, you can not consider one after another, unless you implement a complex repair algorithm or a sophisticated preview mechanism.

With the concept of Sliders and Offsets, we provide a simple way to model such linearly coupled constraints. To intersect constraints with different sliders (domain

filtering), we need to introduce the Polygon Stack Conversion, which we describe in the remaining section of this paper.

3.2 Sliders and Offsets

In order to model the above-described example (see 3.1), one may introduce two resources, *acsInUseIndicator* and *memoryInUseIndicator*, both of which are given an initial constant value 0 and an upper bound 1. Each activity which uses ACS (including the activity detectFire) is given a constraint which increases the value of *acsInUseIndicator* by 1 wherever the activity is planned. This way planning two activities in parallel, which both use ACS, would cause a conflict, because the resource *acsInUseIndicator* would have value 2 during times where two activities overlap. Similarly all activities, which read from or write to memory, are given a constraint to increase the *memoryInUseIndicator*, which prevents two such activities from being planned in parallel. Note that the modification profiles of the constraints are not absolute time-based profiles but duration-based profiles, which still need to be mapped to the time axis via the activity's timeline entry (see [8]): only when we know where the activity starts and ends, can we derive the time profile, which is added to the resource profile. As stated in 3.1, the traditional mappings *at start*, *at end* and *over all* are not sufficient to specify the mapping we desire in our example. Instead, we define a *start-reference* and an *end-reference* each of which consists of a *slider* $\in \mathbb{Q}$ and an *offset* $\in \mathbb{Q}$. Each reference (*slider*, *offset*) transforms a timeline entry $E = (start\text{-}time, duration)$ to a time T via an affine transformation

$$T = start\text{-}time + slider \cdot duration + offset$$

In our example, we want to formulate that the constraint shall apply starting 10 s before the timeline entry until the middle of the timeline entry. With our definition, we can achieve this by setting:

$$start\text{-}reference = (0, -10)$$
$$end\text{-}reference = \left(\frac{1}{2}, 0\right).$$

Note that the type of the resource and the effects on it have nothing to do with the concept of Sliders and Offsets. In our example, we chose constraints, which allocate a boolean resource, but we could also add another constraint referring, e.g., to the *state of charge of a battery*, where the effect of planning the activity could be that the resource's profile is reduced by a constant rate, starting and ending at times, specified by one Sliders and Offsets each.

Also note that the concept of Sliders and Offsets may be applied to time dependencies, too, where one slider is

defined for both *predecessor* and *successor* and one common offset specifies the minimum separation in between the two times derived from the activities' timeline entries.

As described in Sect. 3.1, the benefit of introducing Sliders and Offsets is that it simplifies the planning model and the planning algorithm. Unfortunately, this introduces significant complexity within our main use case, finding the set of conflict free timeline entries for a given activity. For this, each constraint must produce a PolygonSet indicating the set of conflicting timeline entries, which now needs to reflect slider and offset. In addition to this, the performance of resource dependency calculations may be improved significantly by cutting off time intervals which are out of scope due to other constraints. This cut-off, however, is not obvious due to the use of sliders.

To solve this challenge, we distinguish in between two different domains, the *Timeline Entry Domain*, which represents sets of timeline entries and the *Constraint Domain*, which represents sets of profile intervals, and we provide a conversion in between them.

3.2.1 Timeline Entry Domain

The Timeline Entry Domain consists of all PolygonSets (i.e., PolygonStacks with values in $\{0, 1\}$), which represent timeline entries. An entry of the form $(x, y)_T$ with $y \geq 0$, therefore, represents a timeline entry with *start-time* $= x$ and *duration* $= y$.

3.2.2 Constraint Domain

The Constraint Domain consists of all PolygonSets, whose values represent the *start-times* and durations of constraint intervals, i.e., an element $(a, b)_C$ of a PolygonSet in Constraint Domain represents the interval $[a, a + b]$ to which the constraint's duration-based profile is mapped to. For our example of an upper resource bound with *start-reference* $ref_s = (f_s, o_s)$ and *end-reference* $ref_e = (f_e, o_e)$, an element $(a, b)_C$ of a PolygonSet in the Constraint Domain represents the *start-time* and *duration* of a timeline entry *after* it has been transformed using the two references. This means, given a timeline entry $E_T = (x, y)_T$, the corresponding constraint interval $E_C = (a, b)_C$ is given by

$$a(x, y) = x + f_s y + o_s, \tag{3}$$

$$b(x, y) = (x + f_e y + o_e) - a$$
$$= (f_e - f_s)y + o_e - o_s. \tag{4}$$

The reverse relation can be easily obtained from (3) and (4) as

$$y(a, b) = \frac{b - o_e + o_s}{f_e - f_s}, \tag{5}$$

$$x(a, b) = a - o_s - f_s y(a, b)$$
$$= a - o_s - f_s \frac{b - o_e + o_s}{f_e - f_s} \tag{6}$$

for $f_e \neq f_s$. Note that in case $f_s = f_e$, the constraint profile interval's duration is constant, no matter how long the timeline entry lasts. This case needs to be handled separately; however, it turns out that it can be easily solved using (3).

3.2.3 Domain filtering

Provided we can convert in between these two domains, we may apply the domain filtering of one constraint (i.e., reducing the set of allowed timeline entries) as follows:

1. *optional* convert the PolygonStack of valid timeline entries of previously considered constraints into a PolygonStack representing the corresponding constraint intervals of the current constraint. This PolygonStack can be used to restrict the resource profile of current constraint's resource to regions where it may be affected by non-conflicting timeline entries
2. determine the PolygonStack of valid constraint intervals for the current constraint, possibly based upon the restricted resource profile
3. convert the PolygonStack of valid constraint intervals for the current constraint into a PolygonStack representing the valid timeline entries according to this constraint
4. intersect the PolygonStack of valid timeline entries of this constraint with the PolygonStack of valid timeline entries of previously considered constraints

When provided with this conversion, the implementation of the constraint may omit the Sliders and Offsets and, therefore, is as simple (or complex) as without them.

3.2.4 Example

As an example, let us consider an upper resource bound with a constant profile 0, where the profile start is included and its end is excluded (extending our original example, this constraint may belong to an activity, which requires ACS to be inactive). The constraint's implementation without slider and offset returns a PolygonStack containing all (a, b) such that the resource's profile remains less than or equal to zero during the time interval $[a, a + b]$. To find the set of timeline entries, which do not violate this constraint, we need

to convert this PolygonStack from Constraint Domain to Timeline Entry Domain.

3.3 Converting PolygonStacks in between Timeline Entry Domain and Constraint Domain

Whereas Sect. 3.2.2 describes how to convert one element of a PolygonSet, the main challenge is to convert rays and, thus, PolygonStacks from one domain to the other. This shall be described in this section.

Note that in case $f_s = f_e$, the duration of the constraint interval is constant for all timeline entries. It turns out that this special case can easily be handled separately, we, therefore, assume in the remaining part of the paper that

$$f_s \neq f_e. \tag{7}$$

Also note that the transformation is affine linear and, thus, maps lines to lines.

Lemma 2 *Let s_T denote the slope of a line in Timeline Entry Domain, then the corresponding line in Constraint Domain has slope s_C with*

$$s_C = \begin{cases} \dfrac{(f_e - f_s)s_T}{1 + f_s s_T} & f_s s_T \neq -1, \tag{8} \\ \infty & f_s s_T = -1. \tag{9} \end{cases}$$

Proof Let

$$L_T(x, y, s_T) = \{(x + \delta, y + \delta s_T) \in (\mathbb{Q} \times \mathbb{Q}) \mid \delta \in \mathbb{Q}\}$$

define a line in the Timeline Entry Domain passing through (x, y) with slope s_T. According to (3) and (4), its image L_C in the Constraint Domain consists of $(\alpha, \beta) \in \mathbb{Q} \times \mathbb{Q}$ with

$$\begin{aligned} \alpha &= x + \delta + f_s(y + \delta s_T) + o_s \\ &= x + f_s y + o_s + (1 + f_s s_T)\delta \quad \text{and} \end{aligned}$$
$$\begin{aligned} \beta &= (f_e - f_s)(y + \delta s_T) + o_e - o_s \\ &= (f_e - f_s)y + o_e - o_s + (f_e - f_s)s_T \delta. \end{aligned}$$

In case $f_s s_T = -1$, we see that α is constant and according to (7) β takes all values in $\mathbb{Q}$, thus L_C is a vertical line and has infinite slope. For $f_s s_T \neq -1$, we obtain for points $(\alpha, \beta) \in L_C$

$$\begin{aligned} \alpha &= a(x, y) + (1 + f_s s_T)\delta \\ &= a(x, y) + \Delta \quad \text{and} \end{aligned}$$
$$\begin{aligned} \beta &= b(x, y) + (1 + f_s s_T)\delta \frac{(f_e - f_s)s_T}{(1 + f_s s_T)} \\ &= b(x, y) + \Delta \frac{(f_e - f_s)s_T}{(1 + f_s s_T)}, \end{aligned}$$

where $\Delta = (1 + f_s s_T)\delta$, yielding the slope (8). □

Lemma 3 *A line with slope s_C in Constraint Domain corresponds to a line with slope s_T in Timeline Entry Domain, where*

$$s_T = \begin{cases} \dfrac{s_C}{f_e - f_s(1 + s_C)} & f_e \neq f_s(1 + s_C) \tag{10} \\ \infty & f_e = f_s(1 + s_C) \tag{11} \end{cases}$$

Proof Let

$$L_C(a, b, s_C) = \{(a + \delta, b + \delta s_C) \in (\mathbb{Q} \times \mathbb{Q}) \mid \delta \in \mathbb{Q}\}$$

denote a line through (a, b) with slope s_C in the Constraint Domain. According to (5) and (6), its image L_T in the Timeline Entry Domain is given by $(\chi, \psi) \in \mathbb{Q} \times \mathbb{Q}$ such that

$$\begin{aligned} \chi &= a + \delta - o_s - f_s \frac{b + \delta s_C - o_e + o_s}{f_e - f_s} \\ &= a + \delta - o_s - f_s \frac{b - o_e + o_s}{f_e - f_s} - \frac{f_s \delta s_C}{f_e - f_s} \\ &= a + \delta - o_s - f_s y(a, b) - \frac{f_s \delta s_C}{f_e - f_s} \\ &= x(a, b) + \delta - \frac{f_s \delta s_C}{f_e - f_s} \\ &= x(a, b) + \delta \left(1 - \frac{f_s s_C}{f_e - f_s}\right) \quad \text{and} \end{aligned}$$
$$\begin{aligned} \psi &= \frac{b + \delta s_C - o_e + o_s}{f_e - f_s} \\ &= \frac{b - o_e + o_s}{f_e - f_s} + \frac{\delta s_C}{f_e - f_s} \\ &= y(a, b) + \delta \frac{s_C}{f_e - f_s}. \end{aligned}$$

If $f_e = f_s(1 + s_C)$ we see that ψ is constant and χ takes all values in $\mathbb{Q}$, thus L_T is a vertical line and has infinite slope. In case $f_e \neq f_s(1 + s_C)$, we obtain a slope

$$\begin{aligned} s_T &= \frac{\delta \frac{s_C}{f_e - f_s}}{\delta \left(1 - \frac{f_s s_C}{f_e - f_s}\right)} \\ &= \frac{s_C}{f_e - f_s(1 + s_C)}. \end{aligned}$$

□

Now that we know how half lines transform under this conversion, we can derive how rays are converted from Timeline Entry Domain to Constraint Domain and back. Notice, however, that it is not enough to map the half line defining the ray via this map since we are actually interested in the subset of the domain D that is defined by this ray. As this area is also bounded by the implicit vertical line above the base point of the ray, it is clear that the image of the area of a ray under such a conversion might need to be described by multiple

rays. This will be shown in various examples in Sect. 3.3.2. Furthermore it is possible that the references are such that the resulting image lies in outside of the domain D, for example *start-time reference* $= (0, 1)$ and *end-time reference* $= (1, 0)$ will result in a constraint interval with negative duration for time intervals of length smaller than 1. Thus a conversion of a PolygonStack P in D in any direction actually means that we map the subset of D described by P to the other domain and then intersect it with D. Therefore, these conversions are not bijections of D and by mapping forth and back we introduce so-called *implicit constraints* as we might remove parts of the PolygonStack. This will also be explained by some examples in Sect. 3.3.2.

3.3.1 Converting rays

The formulas (3–6) and (8–11) show how the half line of a ray is transformed. It remains to consider

1. cutoffs on the left and bottom, since we don't support rays starting at e.g. $x = -\infty$ or rays affecting the region below their half lines,
2. the special cases of *slope* $= \infty$ and
3. whether the region above a ray's half line is mapped to the region above or below the transformed half ray

Regarding 1, recall that our domain D was bounded from below (as durations are non-negative) and from the left (since we consider the whole problem only after a base time X_0). Since both, timeline entries and constraint intervals, can not have negative duration, values below $y = 0$ are considered out of scope, too. To describe the remaining two issues, one has to distinguish various cases from the different relations of *start-* and *end-factors*, i.e.

$$f_s > f_e \quad f_s < f_e \quad f_s = f_e$$

and for the slopes of the considered rays

$$s = \infty \quad s > S \quad s = S \quad s < S,$$

where $S \in \mathbb{Q}$ denotes the *critical slope*, i.e., the one which is mapped to ∞ [see (9), (11)]. For the conversion from Timeline Entry Domain to Constraint Domain, the critical slope is given by

$$S_{T \to C} = -\frac{1}{f_S} \tag{12}$$

and for the conversion from Constraint Domain to Timeline Entry Domain, the critical slope is given by
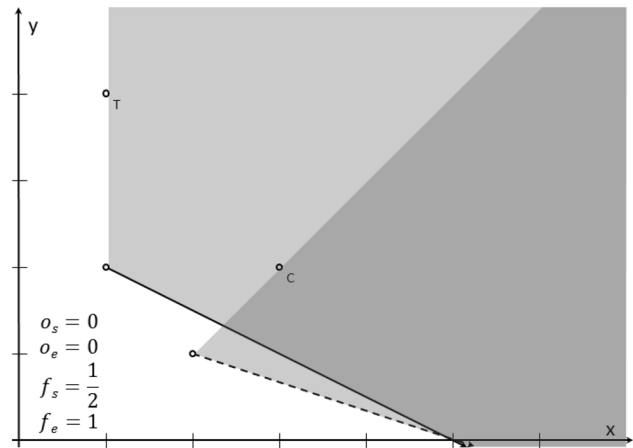
$$S_{C \to T} = \frac{f_e}{f_s} - 1. \tag{13}$$

Fig. 4 Conversion from Timeline Entry Domain (continuous line) to Constraint Domain (dashed line)

As stated in (7), we will not consider the case $f_s = f_e$. However, we still need to investigate $2 \times 4 = 8$ cases for both conversions. In the following, we pick a few examples for demonstrations.

3.3.2 Examples for conversions of rays

Figure 4 shows the conversion *start-reference* $= \left(\frac{1}{2}, 0\right)$, *end-reference* $= (1, 0)$ from Timeline Entry Domain to Constraint Domain, applied to $\text{Ray}\left(1, 2, -\frac{1}{2}, d_{At}, d_{Above}\right)$ which has slope greater than the critical slope. The point $(1, 4)_T$ in Timeline Entry Domain is mapped to $(3, 2)_C$ in Constraint Domain; therefore, the region above the Timeline Entry Domain's ray is mapped to the region within the dashed line and the half line starting at $(2, 1)_C$ and passing through $(3, 2)_C$. Thus, the result of the conversion consists of two rays, the dashed line and the upper edge of the mapped region:

$$\text{Ray}\left(2, 1, -\frac{1}{3}, d_{At}, d_{Above}\right),$$

$$\text{Ray}(2, 1, 1, -d_{At}, -d_{Above}).$$

This is an example of a PolygonStack described by a single ray in Timeline Entry Domain that is mapped to another PolygonStack in Constraint Domain that needs two rays for description.

Figure 5 shows the same conversion but from Constraint Domain to Timeline Entry Domain, applied to $\text{Ray}(2, 1, -\frac{1}{3}, d_{At}, d_{Above})$. The point $(2, 2)_C$ in Constraint Domain is now mapped to $(0, 4)_T$ in Timeline Entry Domain; therefore, the region above the Constraint Domain's ray is mapped to the region above the continuous
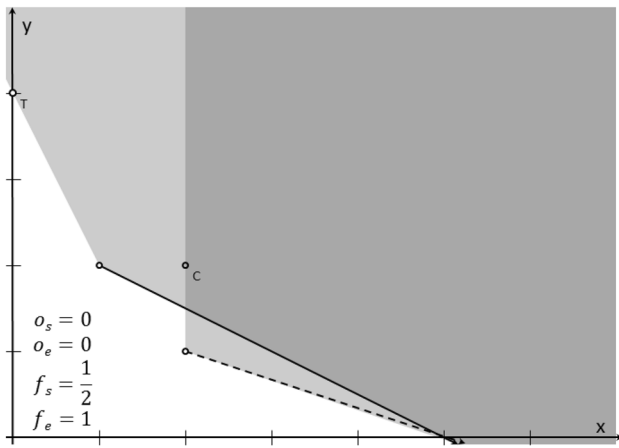
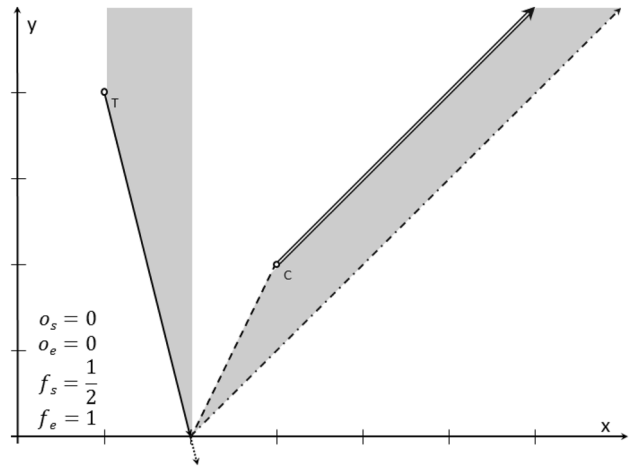**Fig. 5** Conversion from Constraint Domain (dashed line) to Timeline Entry Domain (continuous line)

Fig. 7 Conversion from Timeline Entry Domain to Constraint Domain with *slope < critical slope*
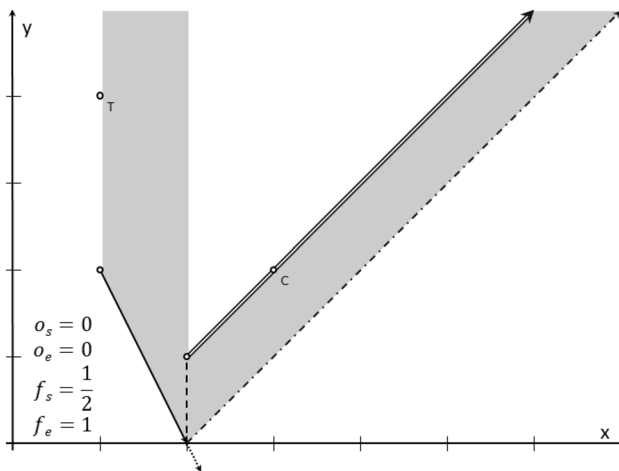

Fig. 6 Conversion from Timeline Entry Domain to Constraint Domain with *slope = critical slope*. The point T is mapped to the point C together with the corresponding lines in the figure

line and the half line starting at $(1, 2)_T$ and passing through $(0, 4)_T$. Unfortunately, we can not specify a ray pointing to the left; therefore, we need to start one ray at X_0 [see (1)] and cancel it at $(1, 2)_T$. The result of the conversion, therefore, consists of three rays:

$$\mathrm{Ray}(X_0, Y_0, -2, 0, d_{\mathrm{Above}}),$$
$$\mathrm{Ray}(1, 2, -2, 0, -d_{\mathrm{Above}}),$$
$$\mathrm{Ray}\left(1, 2, -\frac{1}{2}, d_{\mathrm{At}}, d_{\mathrm{Above}}\right),$$

with $Y_0 = 2 + 2(1 - X_0)$, i.e. the ray passes $(1, 2)_T$.

Another example in Fig. 6 shows the conversion of a ray r with *slope = critical slope*: the converted region

would have to start at $y = -\infty$. To solve this, we introduce a helper-ray $\mathrm{Ray}(2, 0, -2, -d_{\mathrm{At}}, -d_{\mathrm{Above}})$ illustrated by the dotted line. As y can not take values smaller than 0 in the PolygonStack, we know that there must exist further rays on the same line, all of which start at or above $y = 0$, such that all rays of the line sum up to 0 for all $x > 2$. Thus, the helper-rays we introduce for all of these rays on the same line within the PolygonStack sum up to 0, which means that by introducing the helper-rays, we do not modify the PolygonStack. We, therefore, can convert the ray together with its helper-ray. The result consists of four rays

$$\mathrm{Ray}(2, 0, \infty, d_{\mathrm{At}}, d_{\mathrm{At}}),$$
$$\mathrm{Ray}(2, 1, \infty, -d_{\mathrm{At}}, -d_{\mathrm{At}}),$$
$$\mathrm{Ray}(2, 0, 1, d_{\mathrm{Above}}, d_{\mathrm{Above}}) \text{ and}$$
$$\mathrm{Ray}(2, 1, 1, -d_{\mathrm{Above}}, -d_{\mathrm{Above}}).$$

In Fig. 7 one can see the conversion of a ray r with *slope < critical slope*. Similar to the preceding case depicted in Fig. 6, we need to introduce a helper-ray. This time the result of converting the ray and its helper-ray consists of 5 rays

$$\mathrm{Ray}(2, 0, 1, d_{\mathrm{Above}}, d_{\mathrm{Above}}),$$
$$\mathrm{Ray}(2, 0, 2, d_{\mathrm{At}} - d_{\mathrm{Above}}, -d_{\mathrm{Above}}),$$
$$\mathrm{Ray}(3, 2, 2, d_{\mathrm{Above}} - d_{\mathrm{At}}, d_{\mathrm{Above}}),$$
$$\mathrm{Ray}(3, 2, 1, -d_{\mathrm{Above}}, -d_{\mathrm{Above}}),$$
$$\mathrm{Ray}(3, 2, \infty, -d_{\mathrm{At}}, 0),$$

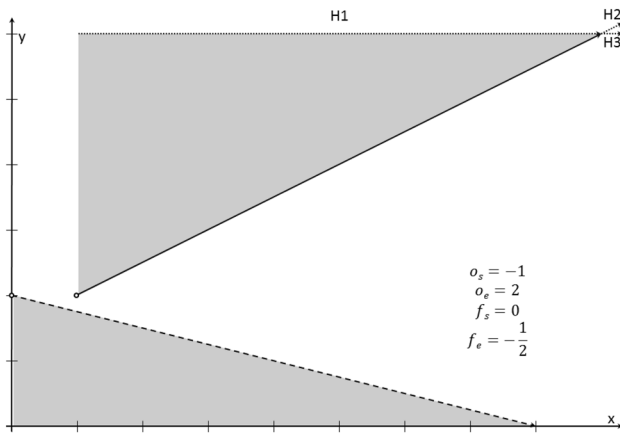where the last one merely corrects the value at point $(3, 2)$, which must evaluate to 0.

Fig. 8 Conversion from Timeline Entry Domain to Constraint Domain with $f_e < f_s$. H_1, H_2 and H_3 are helper-rays, which are required in this case

As Figs. 4, 5, 6 and 7 show, the conversion from Timeline Entry Domain to Constraint Domain distorts and tilts the PolygonStack. In case the slider of the *end-reference* is greater than the slider of the *start-reference*, complexity is introduced mainly due to the fact that rays can not point *to the left* and that they can not specify the value below their half line.

In case the slider of the *end-reference* is smaller than the slider of the *start-reference*, however, the rays are also mirrored, see Fig. 8. This means that a ray $\mathrm{Ray}\left(1, 2, \frac{1}{2}, d_{\mathrm{At}}, d_{\mathrm{Above}}\right)$ with finite slope may transform to a region below a half line. However, in this case there exists a natural bound on the duration of a timeline entry to keep the constraint interval well defined. In our case, no timeline entry may have a duration greater than 6; otherwise, the constraint interval's *end-time* lies before its *start-time*. We consider such a combination of time references as an *implicit duration constraint* and therefore, we can restrict our PolygonStacks to those obeying the upper bound.

We now define a helper-ray H_1 which cancels the values above the upper bound, and—if applicable—two helper-rays H_2 and H_3 canceling $H1$ and the original ray at the crossing point of the upper bound and the to-be-converted ray, see the upper right corner of Fig. 8. Since we know that the PolygonStack has the ambient value above the upper bound, the helper-rays again need to sum up to 0. When mapping the ray and its helper-rays, we obtain the region below and including the dashed line, which we can represent using 4 rays
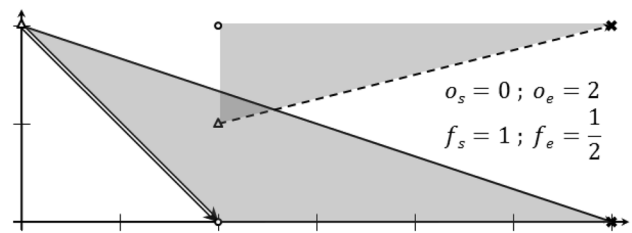

Fig. 9 Conversion from Constraint Domain to Timeline Entry Domain with $f_e < f_s$
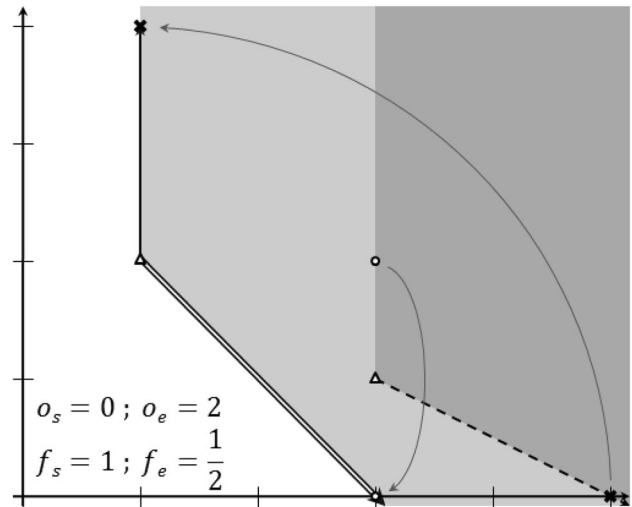

Fig. 10 Conversion from Constraint Domain to Timeline Entry Domain with $f_e < f_s$ for a ray with critical slope

$$\mathrm{Ray}\left(0, 2, -\frac{1}{4}, d_{\mathrm{At}} - d_{\mathrm{Above}}, -d_{\mathrm{Above}}\right),$$
$$\mathrm{Ray}(0, 0, 0, d_{\mathrm{Above}}, d_{\mathrm{Above}}),$$
$$\mathrm{Ray}(8, 0, 0, -d_{\mathrm{Above}}, -d_{\mathrm{Above}}) \text{ and}$$
$$\mathrm{Ray}\left(8, 0, -\frac{1}{4}, d_{\mathrm{Above}} - d_{\mathrm{At}}, d_{\mathrm{Above}}\right).$$

To understand this mapping visually, you have to start with the upper bound helper-ray, which adds $-d_{\mathrm{Above}}$ above its half line. The half line is mapped to the *x*-axis and as it is mirrored, the values at and above the mapped half line must be increased by d_{Above}, which, therefore, starts the region of the mapped PolygonStack. The ray itself is mapped to the dashed line and—as it is mirrored—no longer adds d_{At} and d_{Above} but instead adds $d_{\mathrm{At}} - d_{\mathrm{Above}}$ and $-d_{\mathrm{Above}}$ to set the values on and above the half line.

To understand a conversion from Constraint Domain to Timeline Entry Domain as depicted in Fig. 9, one has to consider the mapping of three points: $(2, 1)_C \rightarrow (0, 2)_T$, $(2, 2)_C \rightarrow (2, 0)_T$ and $(6, 2)_C \rightarrow (6, 0)_T$. Thus, the vertical border of the ray's affected region, i.e., the vertical line above

(2, 1), is mapped to the half line starting at (0, 2) and passing through (2, 0). The dashed line is mapped to the upper border of the mapped region. The ray $\mathrm{Ray}\left(2, 1, \frac{1}{4}, d_{\mathrm{At}}, d_{\mathrm{Above}}\right)_{\mathrm{C}}$, therefore, is mapped to

$\mathrm{Ray}\left(0, 2, -1, 0, d_{\mathrm{Above}}\right)_{\mathrm{T}}$ and

$\mathrm{Ray}\left(0, 2, -\frac{1}{3}, d_{\mathrm{At}} - d_{\mathrm{Above}}, -d_{\mathrm{Above}}\right)_{\mathrm{T}}$.

Note that in this case, we do not have to introduce helper-rays because we can represent the result with two rays.

Figure 10 shows the same conversion as preceding Fig. 9 but this time converting a $\mathrm{Ray}\left(3, 1, -\frac{1}{2}, d_{\mathrm{At}}, d_{\mathrm{Above}}\right)$ which has critical slope. Again, we only need two rays to represent the result of mapping the ray from Constraint Domain to Timeline Entry Domain:

$\mathrm{Ray}(1, 2, \infty, 0, d_{\mathrm{At}})$ and

$\mathrm{Ray}(1, 2, -1, 0, d_{\mathrm{Above}})$.

Note that this time we do not need to add helper-rays either, even though the ray falls below $y = 0$. We only introduced the helper-rays to be able to represent the result as a PolygonStack and as we can represent the result of converting this ray as a PolygonStack of two rays, everything is fine.

4 Performance

This paper shows how the concept of Sliders and Offsets may be handled computationally, which is a prerequisite for usage in a planning modeling language. To see whether this method is useful, we estimate briefly the asymptotic complexity of all involved operations in the following.

4.1 Internal structure

As mentioned in Sect. 1, we want the representation of a PolygonStack as collection of rays to be unique. In the following, we present two possible solutions, both of which assume the following:

> No two rays of the same PolygonStack may be equal in x, y and their slope s simultaneously.

This assumption does not impose any restriction, as two rays

$r_1 = \mathrm{Ray}(x, y, s, d_0^*, d_1^*)$

$r_2 = \mathrm{Ray}(x, y, s, d_0^\dagger, d_1^\dagger)$

may be replaced by a single ray

$r = \mathrm{Ray}\left(x, y, s, d_0^* + d_0^\dagger, d_1^* + d_1^\dagger\right)$.

4.1.1 Ordered set of rays

A simple internal representation of a PolygonStack is a sorted set of rays, where sorting takes place in lexical order of

1. x (time)
2. y (duration)
3. s (slope)

Note that according to the assumption 4.1, the rays of a PolygonStack are strictly sorted by this criterion. This representation is used by [23], a suitable sweep-line algorithm for operations is described in [20].

4.1.2 Lines of rays

The sweep-line algorithm as proposed by [20] needs to consider all crossings of rays, the number of which is of the order $\mathcal{O}(n^2)$, where n is the number of rays. In our use case, we restrict to bounded PolygonSets. Within the underlying bounded PolygonStacks, each ray must be canceled by some ray emanating from a larger x. The ray canceling a given ray must reside on the same line as this ray. We, therefore, group the rays by lines; this way we only need to determine the crossing points of the lines instead of all rays. Sorting by lines introduces a complexity of $\mathcal{O}(n) \cdot C$, where C is the complexity of inserting an element into the collection of rays of the same line.

For the remaining part, we assume choosing a simple balanced tree with complexity of look up and insertion being $C = \mathcal{O}(\log(n))$. The sorting therefore is of complexity $\mathcal{O}(n \cdot \log(n))$. However, the benefit for the calculation of crossing points for bounded PolygonStacks is in the order of $\mathcal{O}((\frac{n}{2})^2) = \mathcal{O}(n^2)$. Although the overall complexity remains $\mathcal{O}(n^2)$, we choose this representation for our estimation of complexity.

We define the precise representation as follows:

1. All rays are grouped by lines, i.e., all rays, which belong to the same line, are stored in the same group.
2. These groups are split into two types that are stored separately, namely

   (a) vertical lines (slope $= \infty$) with rays of one such line being sorted by y and
   (b) non-vertical lines (slope $\neq \infty$) with rays of one such line being sorted by x.

3. Vertical lines are sorted by their x-coordinate.
4. Non-vertical lines are sorted in lexical order of

   (a) the value of the group's line at $x = 0$ and

    (b)   the slope of the group's line.

Note that all rays on the same non-vertical line must have different values for x, as they otherwise violate the assumption from Sect. 4.1. Similarly, the y-values of rays in a vertical line are all mutually distinct.

4.2 Operations

4.2.1 Addition

To add two polygon stacks G and H of size n, we need to

1. identify the set L of lines occurring in both G and H which has complexity $\mathcal{O}(n \cdot \log(n))$, and then
2. merge the groups of rays $g_l \in G$ and $h_l \in H$ which belong to the same line l. Note that merging two rays with the same x, y and *slope* is of order $\mathcal{O}(1)$. In total, we have a complexity $\sum_{l \in L} \sum_{r \in g_l} \log(|h_l|) \leq n \cdot \log(n)$, i.e., $\mathcal{O}(n \cdot \log(n))$.
3. At last, we need to create a new PolygonStack from the resulting lines which has complexity $\mathcal{O}(n)$.

The total complexity for this operation is, therefore, $\mathcal{O}(n \cdot \log(n))$.

4.2.2 Evaluation

Evaluating a polygon stack at a point (x, y) may require to consider each ray in the PolygonStack. The complexity is, therefore, $\mathcal{O}(n)$.

4.2.3 Scan for value

Recall from Sect. 2.2.4 that a *scan-line* is a vertical line containing a ray's start point or a crossing point of two rays. In a general PolygonStack of size n, the number of crossing points is of order $\mathcal{O}(n^2)$. To scan for a value, we need to

1. find all *scan-lines* which has complexity $\mathcal{O}(n^2)$ and then
2. for every *scan-line* s, determine the effect of all rays on s as well as the interval between s and the succeeding *scan-line* and then aggregate the corresponding rays. This has a complexity per *scan-line* of order $\mathcal{O}(n \cdot \log(n))$.

The complexity for this operation is, therefore, $\mathcal{O}(n^2) \cdot \mathcal{O}(n \cdot \log(n)) = \mathcal{O}(n^3 \cdot \log(n))$.

4.2.4 Find value

To find the best value according to some linear optimization problem as described in Sect. 2.2.5, we need to check the

starting points and crossing points of all rays, see Lemma 1. As there are $\mathcal{O}(n^2)$ of these points, the complexity of this operation is $\mathcal{O}(n^2)$.

4.3 Polygon Stack Conversion

As every ray needs to be converted separately, the complexity of the Polygon Stack Conversion is $\mathcal{O}(n)$. This holds for both directions.

4.4 Comparison with resource calculation complexity

Our main use case is to allow resource constraints supporting Sliders and Offsets. We, therefore, compare the complexity of the PolygonStack operations with the complexity of other resource profile operations required by a planning algorithm.

A typical example of a resource operation is adding a modification profile. This may be to update the state-of-charge profile of a satellite's on-board battery when adding an activity to the timeline. In general, this operation requires updating the whole resource profile, beginning at the time where the modification starts. The complexity of updating such a profile is $\mathcal{O}(m)$, where m is the number of profile segments, i.e., intervals where the profile has constant slope and no jumps.

According to the results in Sect. 4.2, the complexity of intersecting the results of different constraints is dominated by the operation *scan for value*, which has to be performed once. The polygon stack calculation, therefore, has complexity $\mathcal{O}(n^3 \cdot \log(n))$, where n denotes the number of rays in the PolygonStack.

The value of n, however, does not correspond to the number of segments m of a resource. Instead, it is determined by comparing a given value b with the values of a resource. Suppose, for example, that a timeline entry may be placed only in such a way, that the resource value beginning with the timeline entry is above a certain bound, e.g., there must remain sufficient energy for all future activities.[1] To represent the set of consistent timeline entries for this query, we do not have to create one ray per segment of the resource profile, because we don't care how much energy is left, as long as there is sufficient energy left. The number of rays n, therefore, does not correspond to m but only to the number of times the profile crosses b. In practice, this value is far less than the number of segments. The $\mathcal{O}(n^3 \cdot \log(n))$ complexity is, therefore, less problematic than one might think.

---

[1] A lost-values logic (piggy-bank) may be used to assure that power supply will not increase the state-of-charge's value above the battery's capacity.

Nevertheless, it remains an important task of the planning engine to apply the constraints in a good order, such that large sections of complex resource profiles may be omitted due to restrictions of less complex constraints.

5 Summary and outlook

The main result of this paper is given by Lemmas 2 and 3. These equations allow converting timeline entry intervals into constraint intervals and back again, which forms the basis of Sliders and Offsets as introduced in Sect. 3.2 within the planning model. Although we heavily rely on Polygon-Stacks when describing why and how the conversion works, the same formulas should be applicable to any representation of sets of timeline entries, although the conversion's implementation will most likely be more complex to implement.

We selected some of the 16 cases one needs to distinguish when dealing with Polygon Stack Conversion. Using these, we have been able to demonstrate how to implement the Polygon Stack Conversion and how to handle all obstacles which occur, mainly due to the non-symmetric representation of PolygonStack.

We also justified that we do not need to fear run-time issues when introducing this kind of representation. However, the theoretical complexity of $\mathcal{O}(n^3 \cdot \log(n))$ clearly indicates where to proceed when improving the run-time behavior: Step 2 in Sect. 4.2.3, where traversing from one interval to the next might re-use the result of the preceding interval.

Another interesting question and topic of future work is to apply the concept of Sliders and Offsets in algorithms based upon Temporal Networks, as e.g. in [2, 9, 17].

References

1. https://github.com/non/spire. Accessed 11 July 2018
2. Allen, J.F.: Maintaining knowledge about temporal intervals. In: Weld, D.S., de Kleer, J. (eds.) Readings in Qualitative Reasoning About Physical Systems. Morgan Kaufmann, pp. 361–372. ISBN: 978- 1-4832-1447-4 (1990). https://doi.org/10.1016/B978-1-4832-1447-4.50033-X; http://www.sciencedirect.com/science/article/pii/B978148321447450033X
3. Barreiro, J., et al.: EUROPA: a platform for ai planning, scheduling, constraint programming, and optimization. In: International Conference on Planning and Scheduling for Space 2012 (2012). http://icaps12.icaps-conference.org/ickeps/ICKEPS2012-EUROPA.pdf
4. Bentley, J.L., Ottmann, T.A.: Algorithms for reporting and counting geometric intersections. In: IEEE Transactions on Computers C-28.9, pp. 643– 647. ISSN: 0018-9340 (1979). https://doi.org/10.1109/TC.1979.1675432
5. Cesta, A., et al.: Mexar2: AI solves mission planner problems. In: IEEE Intelligent Systems 22.4, pp. 12–19. ISSN: 1541-1672 (2007). https://doi.org/10.1109/MIS.2007.75
6. Chien, S., et al.: ASPEN—automating space mission operations using automated planning and scheduling. In: International Conference on Space Operations (SpaceOps 2000). Toulouse, France (2000)
7. Chien, S.A., et al.: A generalized timeline representation, services, and interface for automating space mission operations. In: Proceedings of 12th International Conference on Space Operations. Stockholm, Sweden (2012). https://ai.jpl.nasa.gov/public/papers/chien_spaceops2012_generalized.pdf
8. Planning Modelling Language. https://www.dlr.de/rb/Portaldata/38/Resources/dokumente/GSOC_dokumente/RB-MIB/GSOC_Modelling_Language.pdf. Accessed 11 Sept 2018
9. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artif. Intell. **49**(1–3), 61–95 (1991). https://doi.org/10.1016/0004-3702(91)90006-6
10. Fratini, S., Cesta, A.: The APSI framework: a platform for timeline synthesis. In: 1st Workshops on Planning and Scheduling with Timelines PSTL-12 (2012)
11. Giezeman, G.-J., Wesselink, W.: CGAL chapter 2D-Polygon. https://doc.cgal.org/latest/Polygon/index.html#Chapter_2D_Polygon. Accessed 11 Dec 2018
12. GMV. flexplan. https://www.gmv.com/en/Products/flexplan/. 11 Dec 2018
13. Gutiérrez, L.G., et al.: FlexPlan: deployment of powerful comprehensive Mission Planning Systems (2006). https://doi.org/10.2514/6.2006-5679. Accessed 11 Dec 2018
14. Holwerda, K.: Introduction Boolean algorithm. http://boolean.klaasholwerda.nl/bool.html. Accessed 11 Dec 2018
15. Johnston, M.D., Miller, G.E.: Intelligent scheduling. In: Aarup, M., Zweben, M., Fox, M (eds.) 1st ed. Morgan Kaufmann, San Francisco (1994). ISBN: 9781558602601
16. Johnston, M., Giuliano, M.: MUSE: the multi-user scheduling environment for multi-objective scheduling of space science missions. In: IJCAI-09 Workshop on Artificial Intelligence in Space (2009). http://robotics.estec.esa.int/IWPSS/IWPSS_2011/Papers/Giuliano_Paper.pdf. Accessed 11 Dec 2018
17. Laborie, P.: Resource temporal networks: definition and complexity. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence IJCAI'03, pp. 948–953. Morgan Kaufmann Publishers Inc., Acapulco, Mexico (2003). http://dl.acm.org/citation.cfm?id=1630659.1630796
18. Nibler, R., et al.: PINTA and TimOnWeb—(more than) generic user interfaces for various planning problems. In: IWPSS 2017—10th International Workshop on Planning and Scheduling for Space (2017). https://elib.dlr.de/114095/. Accessed 11 Dec 2018
19. Rabideau, G., et al.: Iterative repair planning for spacecraft operations using the ASPEN system. In: International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS 1999). The Netherlands, Noordwijk (1999). https://trs.jpl.nasa.gov/handle/2014/17411

20. Simonson, L., Suto, G.: Geometry template library for STLlike 2D operations. In: Colorado: GTL Boostcon 2009. Aspen Center for Physics, Aspen, CO 81611 (2009). https://www.boost.org/doc/libs/1_57_0/libs/polygon/doc/GTL_boostcon2009.pdf

21. Smith, D.E., Cushing, W.: The ANML language. In: Poster Session, Proceedings of the 18th International Conference on Automated Planning and Scheduling (2008). https://ti.arc.nasa.gov/m/pub-archive/1423/1423%20(Smith,%20D).pdf

22. STScI. SPIKE. http://www.stsci.edu/institute/software_hardware/spike. Accessed 11 Dec 2018

23. THE BOOST.POLYGON LIBRARY. https://www.boost.org/doc/libs/1_57_0/libs/polygon/doc/index.htm. Accessed 11 July 2018

24. Vatti, B.R.: A generic solution to polygon clipping. In: Communications of the ACM 35.7, pp. 56–63 (1992). ISSN: 0001-0782. https://doi.org/10.1145/129902.129906