Simple Temporal Networks for EVA Timeline Execution

Cameron Pittman, JSC-XI; EOS team

January 2020

1 Introduction

The purpose of this document is to describe what a Simple Temporal Network (STN) is, how to reason about one, and ideas about their application within the context of an extravehicular activity (EVA) timeline. It describes data structures and algorithms for querying an EVA timeline on-the-fly for arbitrary time math.

Our goal is to build software that allows personnel such as flight controllers and intravehicular (IV) operators to ask questions about the temporal relationships between steps in an EVA timeline. While the discussion and examples will draw from EVAs, the techniques described herein come from computer science and are applicable across any timeline so long as it formulates constraints in a compatible way.

We will formulate the problem such that an EVA timeline is a collection of distinct activities with distinct start and end points. In order to have a timeline, the activities must have temporal constraints that dictate the sequential relationships between them. The overall timeline does not need to be strictly linear - the sequential relationships allow for overlapping activities and even activities with temporal constraints that relate to non-sibling (either directly prior or directly after) tasks. Note that the lack of strict linearity does not imply that *any* set of temporal constraints are valid. As will be discussed later, consistent timelines must meet certain criteria to ensure all constraints can be met.

2 Simple Temporal Networks

STNs describe relationships between temporally connected events in a way that enables querying and online updating (in the context of autonomy, "online" is another way of saying "on-the-fly") (see [1] [2] [3] for source material). STNs also enable flexible execution by recalculating consistent timeline calculations as events are updated with as-executed times (not covered here in full).

The walkthrough below draws heavily on MIT Principles of Autonomy and

Decision Making lecture 21 notes (in Confluence), especially in terms of examples.

The following section is divided into three parts: building an STN in section 2.1, uncovering implicit (hidden) constraints in section 2.2, and using the full STN to perform as-executed online scheduling in section 2.4.

2.1 Events and Intervals

We will begin by describing the type of event data applicable to STNs. Given two events, X and Y, the set of relationships we'll consider are in figure 1.

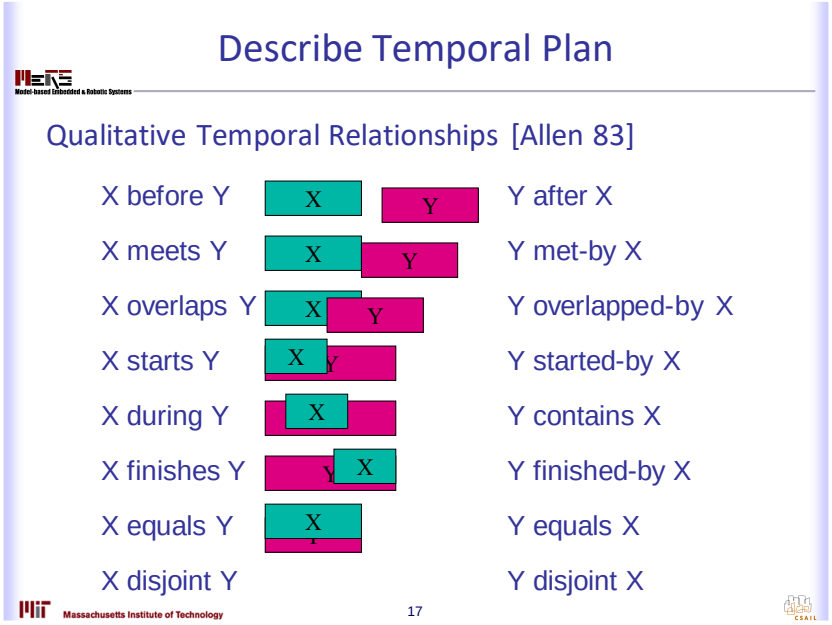


Figure 1: Page 17 in the MIT lecture notes, see also [4]. Note that X disjoint Y refers to two events that cannot overlap.

In more formal notation, X^- represents the start of X and X^+ represents the end of X.

As an example, consider activity C, "Collect Soil Sample", and activity A, "Analyze Soil Sample". C will take between 10 and 30 minutes to complete and A must begin within the same 24-hour period (or 1440 minutes to keep the same units).

$$10 \leq (C^+ - C^-) \leq 30 \tag{1}$$

Equation 1 can be read as saying the difference between the start of C and the end of C is between 10 and 30 minutes.

$$0 \leq (A^- - C^+) \leq 1440 \tag{2}$$

Equation 2 once again says that the difference between the start of A and the end of C is less than one day.

Using this notation, we can mathematically describe the aforementioned general temporal relationships. See figure 2.

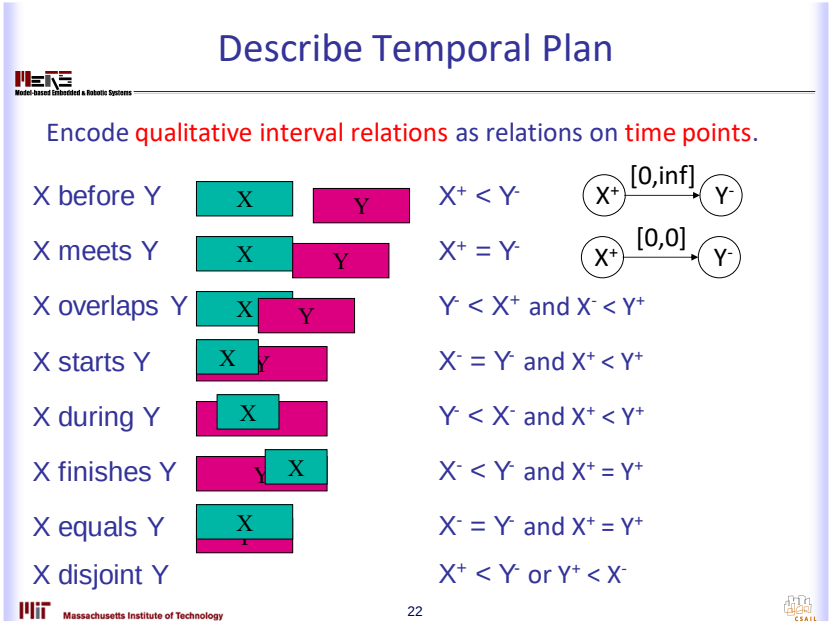


Figure 2: Page 22 in the lecture notes.

Thus, STNs are made of varibles, X_i, that represent points in time, and with binary constraints between them in the form of

$$(X_k - X_i) \in [a_{ik}, b_{ik}] \tag{3}$$

The $\in$ operator means "in" and indicates that $(X_k - X_i)$ falls somewhere within the range of $[a_{ik}, b_{ik}]$. a and b represent the lower and upper bounds respectively between events X_i and X_k. The $[a_{ik}, b_{ik}]$ factor is a time interval, which could be rewritten as $[l(ower)_i, u(pper)_i]$ for simplicity's sake. Given the utility of STNs is found in facilitating arbitrary arithmetic on event intervals, it's worth looking at the basics of arithmetic operations on intervals.

$$
\begin{aligned}
[l_1, u_1] + [l_2, u_2] &= [l_1 + l_2, u_1 + u_2] \\
-[l_1, u_1] &= [-u_1, -l_1] \\
[l_1, u_1] - [l_2, u_2] &= [l_1, u_1] + [-u_2, -l_2] = [l_1 - u_2, u_1 - l_2] \\
[l_1, u_1] \wedge [l_2, u_2] &= [\max(l_1, l_2), \min(u_1, u_2)]
\end{aligned} \tag{4}
$$

where $\wedge$ is the "logical and", or union, operator. We can use a directed graph to make better sense of the relationships between events. Given events X_1, X_2, and X_3, we can demonstrate relationships between them with a graph like figure 3.

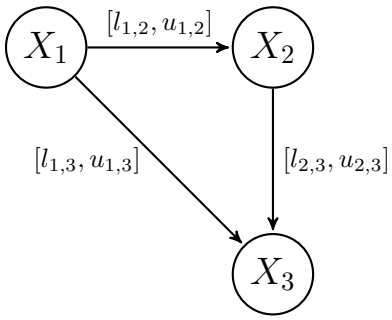


Figure 3: An example STN diagram. These STN diagrams are generally read chronologically from left to right. Note that this style of diagram cannot represent disjoint activities.

Figure 3 shows that X_1 is a requirement for X_2 and X_3, and X_2 is also a requirement for X_3. X_3 has two constraints, which is a situation that already happens in EVAs today. Imagine an EVA wherein the crew is installing a piece of hardware in a multi-step process. This STN represents a scenario where the crew must perform step n and take a picture of the result, X_1, wait for confirmation from ground personnel looking at the image that step n was performed correctly, X_2, before moving on to step $n + 1$, X_3. This format also allows us to specify the lower and upper time boundaries on the durations between activities. It's easy to imagine a situation where the $(X_2 - X_1)$ and $(X_3 - X_1)$ intervals are incompatible. Perhaps ground cannot guarantee that imagery analysis is done in time. Perhaps the timing between steps n and $n + 1$ is too small for analysis due to engineering reasons.

While a person can effectively reason about intervals like these on a small scale, larger timelines introduce too much complexity to reason about on-the-fly. Of course, computers and STNs can make sense of almost arbitrarily complicated timelines, but in order to do so we must expose the implicit constraints.

2.2 Implicit Constraints

Let's take a look at a more complicated STN in figure 4.

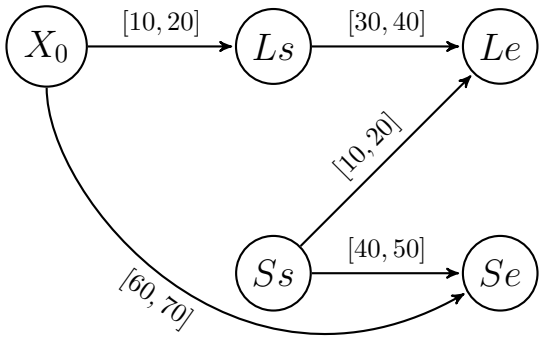


Figure 4: X_0 is the start of the STN. There are two activities, L and S with distinct start and end points, denoted with the suffixes s and e respectively.

Taking a look at figure 4, we can qualitatively observe that this STN represents two activities, L and S, happening in parallel, with the additional constraint between the start of the STN, X_0, and the end of activity Se. Given the constraints here, it's reasonable to ask about the interval between the start times Ls and Ss. And it's also reasonable to ask about interval between the end times, Le and Se. While it should be clear from the STN that there *are* constraints, it's not clear what they are. Let's expose them and run through an example of calculating an implicit constraint. Once we have the implicit constraint, we'll look at how that impacts our flexibility in executing this timeline.

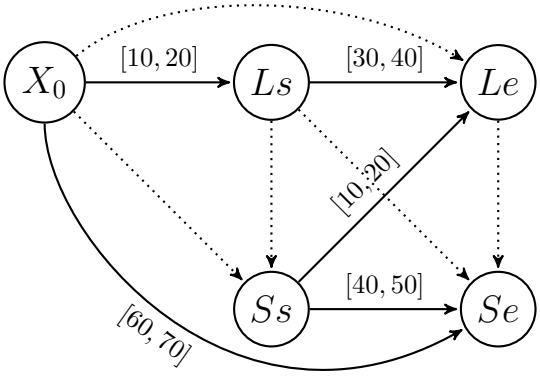


Figure 5: Dotted lines show the additional implicit constraints.

Figure 5 shows constraints that, if exposed, would answer our aforementioned questions about the relationship between L and S. As an example, we'll infer the X_0 to Le constraint by hand. Afterwards, we'll go through the All Pairs Shortest Paths (Floyd-Warshall) algorithm [5], which is what the Maestro STN solver will actually implement to infer implicit constraints.

To start, consider the path between X_0 and Le through Ls.

$$\begin{aligned}(Le - X_0) &\in (Ls - X_0) + (Le - Ls) \\ &\in [10, 20] + [30, 40] \\ &\in [40, 60] \end{aligned} \tag{5}$$

There is a second path between X_0 and Le through Se and Ss.

$$\begin{aligned}(Le - X_0) &\in (Se - X_0) - (Ss - Se) + (Le - Ss) \\ &\in [60, 70] - [40, 50] + [10, 20] \\ &\in [60, 70] + [-50, -40] + [10, 20] \\ &\in [20, 50] \end{aligned} \tag{6}$$

Note the sign change for moving backwards between Se and Ss. We now have two intervals for the same start and end point. We take the union of the paths (see the $\wedge$ operator from 4) from equations 5 and 6 to find the actual implicit constraint.

$$\begin{aligned}(Le - X_0) &\in [40, 60] \wedge [20, 50] \\ &\in [40, 50] \end{aligned}$$

Thus, figure 5 can be updated with the constraint $[40, 50]$ between X_0 and Le.

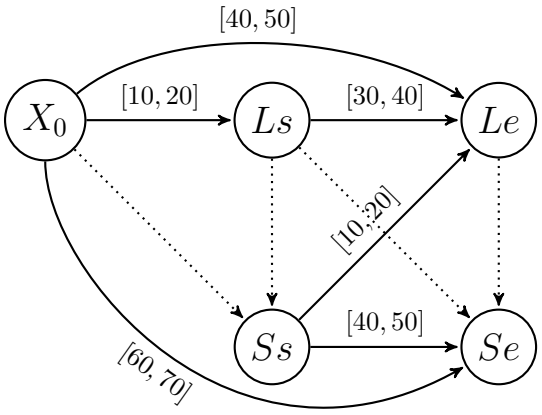


Figure 6: Note the newly inferred constraint between X_0 and Le, which is no longer dotted because we calculated it.

Let's take a look at the execution implications of this new constraint, $(Le - X_0) \in [40, 50]$. Imagine that $(Ls - X_0)$ takes 20 minutes. This satisfies the given constraint $(Ls - X_0) \in [10, 20]$. According to $(Le - Ls)$, we have between 30 and 40 minutes to reach Le. However, the newly inferred constraint, $(Le - X_0) \in [40, 50]$, limits us. In order to calculate the current constraint on $(Le - Ls)$, which we'll denote as $(Le - Ls)'$, we need to union $(Le - X_0)$ with an updated value for Ls.

$$\begin{aligned}(Le - Ls)' &\in [(Le - X_0) - (Ls - X_0)] \wedge (Le - Ls) \\ &\in [[40, 50] - [20, 20]] \wedge [30, 40] \\ &\in [20, 30] \wedge [30, 40] \\ &\in [30, 30] \\ &= 30\end{aligned}$$

The implication of this is that we've blown through our margin by taking 20 minutes to complete $(Ls - X_0)$. Now the only way to satisfy the STN as a whole is to complete $(Le - Ls)$ at the lower bound of its given interval, 30 minutes.

By making the implied constraints visible, we were able to provide online updates about how the execution of one activity impacts the timeline as a whole, even though the timeline isn't strictly linear. What we need now is an algorithm for identifying all of the implied constraints on STNs.

2.3 All Pairs Shortest Paths (APSP) Algorithm

In order to apply APSP, we need to convert our constraint graph into a distance graph. Distance graphs represent STNs by transforming lower interval bounds into negative paths, which is more convenient and efficient, as will be seen in a moment.

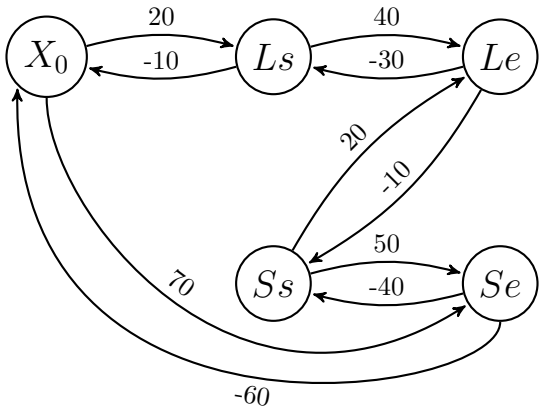


Figure 7: A distance graph representation of figure 4.

In figure 7, the original upper bounds correspond to positive outgoing edges (arrows), and the original lower bounds correspond to negative incoming edges. We can then compare paths going in either direction. Assuming sequential activities X_i and X_j,

$$\begin{aligned} X_j - X_i &\leq \text{upper} \\ X_i - X_j &\leq -\text{lower} \end{aligned} \tag{7}$$

For paths that go between more than two nodes, we simply sum the path lengths. Now let's assume there are additional activities between X_j and X_i. Given path constraints $i_0 = i, i_1 = \cdots, i_k = j$,

$$X_j - X_i \leq \sum_{j=1}^{k} u_{i_{j-1},i_j} \tag{8}$$

The factor u_{i_{j-1},i_j} is the upper bound on the interval between the last node and this one. As we saw when we manually calculated the inferred constraint before, we must union path constraints to find the bounds. However, now we have separated the intervals into separate paths on a distance graph. Rather than taking a union, we simply need to find the shortest path between two

activities to find the tightest upper bound. This amounts to a result that looks like so,

$$X_j - X_i \leq d_{ij}$$

where d_{ij} is the shortest path from X_i to X_j. Thus, we need an algorithm that iterates through all the possible pairs of nodes, uses the distance graph to find all the path distances, and simply takes the smallest one for each pair as the inferred constraint. This is the APSP algorithm.

APSP compares paths between pairs in a distance graph and tries to identify the shortest path between each set of pairs. The easiest way to think about its behavior is by imagining a two-dimensional table. APSP will iterate through the rows and columns, updating the value at each pair with the lowest value it has found so far. For the distance graph from figure 7, the table will be initialized as seen in table 8 below.

		To				
		X_0	Ls	Le	Ss	Se
From	X_0	0	20	∞	∞	70
	Ls	-10	0	40	∞	∞
	Le	∞	-30	0	-10	∞
	Ss	∞	∞	20	0	50
	Se	-60	∞	∞	-40	0

Figure 8: The distance graph in tabular form after it has been initialized with values from figure 7.

All values are first initialized to infinity. The first two for-loops in APSP, lines 2 - 5 of algorithm 1, fill in the actual values from the distance graph. Notice that the distance between any activity and itself is 0.

Algorithm 1 All Pairs Shortest Paths

```
1: Given n nodes in a distance graph
2: for i := 1 to n do
3:     d_ii ← 0                                   {Init. distance to self to 0}
4: for i, j := 1 to n do
5:     d_ij ← w(i, j)        {Init. distances between nodes from distance graph}
6: for k := 1 to n do
7:     for i, j := 1 to n do
8:         d_ij ← min(d_ij, d_ik + d_kj)     {Take the smallest distance seen so far}
```

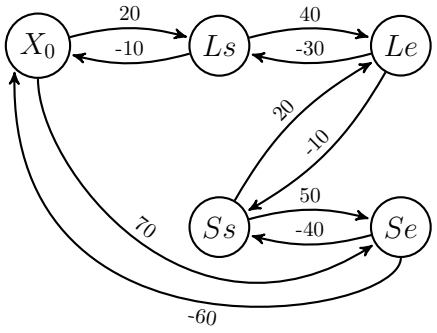
After we gathered the path distances in line 5 of algorithm 1, the APSP algorithm no longer needs to reference the path distances in the distance graph. It is important to recognize that the distances referenced from here on actually come from the current values in the table, not the path values in the distance

graph. Remember, according to line 8 in algorithm 1, we will be performing an update-if-smaller step based solely on the d_{ij}, d_{ik}, and d_{kj} values read from the table.

Next, we iterate through the rows and columns, updating values with the lowest for that pair seen so far. We do so by iterating through intermediate nodes in the distance graph. The for-loop on line 6 has us iterate through all nodes as an intermediary, k. The inner loop on line 7 then iterates through every pair of nodes, i and j in the graph. We update the value by looking at the distance from i through k to j on the table and keeping the value if it is lower than the current d_{ij}, otherwise we discard it.

Let's walk through a few steps in the nested loops of lines 6-8 of APSP. We'll start by looking at $k = Ls$ in the outer loop. (Why not start with $k = X_0$? I'll leave that as an exercise for the reader. Think about it after you go through the walkthrough below.) The first step of the inner loop sets $i = X_0$, which is represented by the first row of the table highlighted in grey. We will iterate through each column by incrementing j. The table below shows the first row updated after iterating j from X_0 to Se in the steps enumerated below. We're including distance graphs in the figures below for reference, but remember, we only need the current distances held in the table for line 8 of APSP.

From \ To	X_0	Ls	Le	Ss	Se
X_0	0	20	~~∞~~ 60	∞	70
Ls	-10	0	40	∞	∞
Le	∞	-30	0	-10	∞
Ss	∞	∞	20	0	50
Se	-60	∞	∞	-40	0



1. In the first step, $k = Ls$, $i = X_0$, $j = X_0$, which is the distance from X_0 through Ls to X_0. Currently we have 0, which is smaller, so no change.
   $(Ls - X_0) + (X_0 - Ls) = 20 + -10 = 10$

2. j moves to Ls. Now we have X_0 through Ls to Ls. This is what we already have, so no change.
   $(Ls - X_0) + (Ls - Ls) = 20 + 0 = 20$

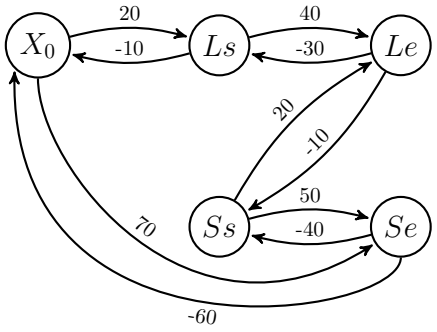
3. Next, j moves to Le. This equates to X_0 through Ls to Le. 60 is smaller than the previous ∞, so it becomes the new value.
   $(Ls - X_0) + (Le - Ls) = 20 + 40 = 60$

4. j moves to Ss, which is not directly reachable from Ls. No change.

5. j ends at Se, which is not directly reachable from Ls. No change.

At this point, j has iterated through all nodes, so now it resets to X_0 and i iterates to Ls. This iteration of i will fill out the second row of the table and represents distances from Ls through Ls. Given that the distance from any node to itself is 0, we won't see any changes through this iteration of i. Let's jump to the next iteration where i is Le.

From \ To	X_0	Ls	Le	Ss	Se
X_0	0	20	60	∞	70
Ls	-10	0	40	∞	∞
Le	~~∞~~ -40	-30	0	-10	∞
Ss	∞	∞	20	0	50
Se	-60	∞	∞	-40	0



1. We start with $k = Ls$, $i = Le$, $j = X_0$. The distance, -40, is smaller than ∞ and becomes the new value.
   $(Ls - Le) + (X_0 - Ls) = -30 + -10 = -40$

2. j moves to Ls. No change.
   $(Ls - Le) + (Ls - Ls) = -30 + 0 = -30$

3. $j = Le$, so we are going from Le through Ls back to Le. The current value of 0 is smaller than 10, so no change.
   $(Ls - Le) + (Le - Ls) = -30 + 40 = 10$

4. j is now Ss. There is no direct route between Ss and Ls, so this yields ∞, which is larger than the current value of -10 and is discarded.

5. $j = Se$. Once again, there is no direct route between Se and Ls, so this will yield ∞ and be discarded.

The process continues until k, i, and j have iterated through all nodes. We finish up with the completed table in figure 9. Remember, the reason we performed APSP is to find the upper and lower bounds for the inferred constraints between all nodes. We iterated through all possible paths between all nodes, keeping only the smallest distances found. Reading the table, rows represent paths that begin at that node. In our distance graph, these were the upper bounds, or latest times, of the interval constraints. Thus, rows represent the upper bounds on intervals between that node and any other. Likewise, columns represent the lower bounds, or earliest times, from that node to any other.

	X_0	Ls	Le	Ss	Se
X_0	0	20	50	30	70
Ls	-10	0	40	20	60
Le	-40	-30	0	-10	30
Ss	-20	-10	20	0	50
Se	-60	-50	-20	-40	0

Figure 9: The distance graph in tabular form after running APSP.

Finally, we have enough information to go back to our original STN and fill in all the inferred constraints. We show that in figure 10.

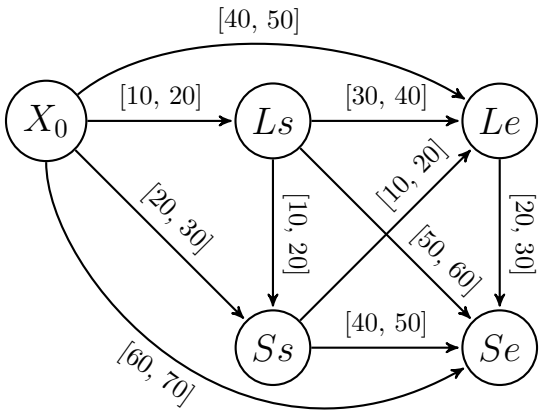


Figure 10: The original STN with all inferred constraints after APSP.

2.3.1 Performance

Given its 3 nested loops, the APSP algorithm is $O(n^3)$[1]. If a timeline has 100 activities, APSP would require 100^3 or 1 million operations to complete. This may sound bad but it is actually fantastic performance in the modern era of computing where even phones can perform billions of operations in a second. With a reasonable amount of attention given to performance, we should be able to run APSP online in a browser.

2.4 Scheduling

Scheduling refers to identifying the intervals when an activity may take place while still satisfying the STN. With respect to EVAs, the process described

[1]This is Big O notation, which is a rough approximation of the number of operations (or memory) required to perform a procedure.

below is analogous to tracking as-performed times and identifying downstream effects. For instance, if an early activity runs long, this algorithm will tell you how much less margin there is for later activities.

I want to point out that so far we have only looked at intervals *between* activities. This algorithm represents a shift in that now we'll look at intervals on the activities themselves, generally called bounds, b. Bounds represent when an activity may occur in relation to the starting point of the STN. We've added bounds to the activities in the STN in figure 11 in bold text assuming that X_0 occurs at $t = 0$.

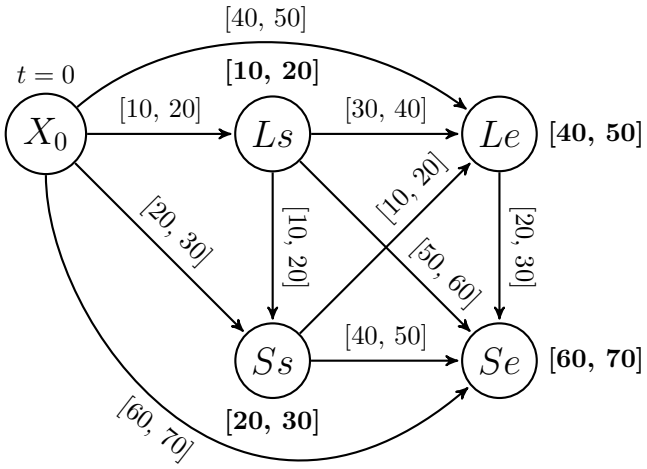


Figure 11: The STN with all activity intervals identified.

Note that setting $t = 0$ on X_0 gives us the nice property that bounds correspond to the interval between X_0 and each activity.

With a full STN with implied constraints, we can perform online updates using a commit and tighten strategy. A commitment is effectively an "as-executed" time within the bounds for an activity. Once we've assigned a commitment, we will propagate changes to later nodes and update (tighten) intervals accordingly. In autonomy parlance, this is called "greedy" scheduling because we can update the bounds on later activities without worrying about their effects on earlier activities. This procedure is described in algorithm 2 below.

Let's walk through an example by committing $(Ls - X_0) = 15$. Our C is then Ls. The unassigned neighbors of C are Le, Ss, and Se. We perform line 7 of algorithm 2 against Le in equation 9. Note that for consistency we'll treat time values as intervals where the lower and upper bounds are the same, eg. $(Ls - X_0) = 15 = [15, 15]$.

Algorithm 2 Greedy Scheduling

```
1: Given an STN with all inferred constraints
2: C ← an unassigned event
3: while C do
4:     begin
5:     time of C ← value within bounds
6:     while X_i in unassigned neighbors of C do
7:         b_{X_i} = b_{X_i} ∧ (time of C + interval C to X_i)
8:     end
```

$$\begin{aligned} b_{Le} &\in [40, 50] \wedge ([15, 15] + [30, 40]) \\ &\in [40, 50] \wedge [45, 55] \\ &\in [45, 50] \end{aligned} \tag{9}$$

Let's do the same for Ss and Se in equations 10 and 11 respectively.

$$\begin{aligned} b_{Ss} &\in [20, 30] \wedge ([15, 15] + [10, 20]) \\ &\in [20, 30] \wedge [25, 35] \\ &\in [25, 30] \end{aligned} \tag{10}$$

$$\begin{aligned} b_{Se} &\in [60, 70] \wedge ([15, 15] + [50, 60]) \\ &\in [60, 70] \wedge [65, 75] \\ &\in [65, 70] \end{aligned} \tag{11}$$

The tightened bounds are shown in bold on the updated STN in figure 12.

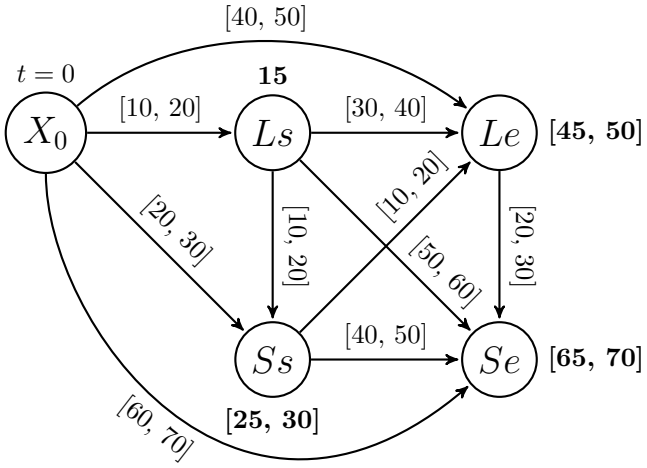


Figure 12: The STN after committing to Ls and tightening the intervals on Le, Ss, and Se.

Let's look at the result in 12. At a high level, we had a 10 minute window for Ls. Ls ate the first five minutes, causing downstream effects of losing five minutes from subsequent bounds.

At this point we are free to continue committing to times and performing greedy scheduling to tighten remaining intervals.

2.4.1 Implications and Properties of Scheduling

We can use the distance graph to determine whether or not the constraints on a timeline are possible to meet, which we can qualify in terms of feasibility. Feasibility refers to the property of being solvable. If an STN is feasible, there is some set of commitments that will satisfy the STN. If it is infeasible, there is no set of commitments that will satisfy it - in other words an infeasible STN is an impossible timeline.

Feasibility can be tested using APSP. The key insight is that if the distance from one activity to any other activity and back to itself is negative, then the timeline is infeasible. This is called a negative cycle. Remember that APSP seeks the smallest distance between activities. Negative cycles indicate a temporally impossible situation where it is somehow faster to complete activity n, activity $n + 1$, and come back to complete activity n again, than it is to just complete activity n once. See figure 13 for an example.

	X_0	Ls	Le	Ss	Se
X_0	-10	20	50	30	70
Ls	-10	0	40	20	60
Le	-40	-30	0	-10	30
Ss	-20	-10	20	0	50
Se	-60	-50	-20	-40	0

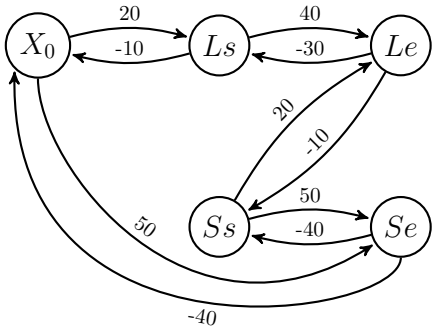


Figure 13: An example of an infeasible STN. The $(Se - X_0)$ interval has been shrunk. Note the negative value on the diagonal in the table at (X_0, X_0). We show that the interval, d, from X_0 through Se and back is negative in equation 12.

$$\begin{aligned} X_0 - Ls &\leq -10 \\ Ls - Le &\leq -30 \\ Le - Ss &\leq 20 \\ Ss - Se &\leq -40 \\ Se - X_0 &\leq 50 \\ d &\leq -10 \end{aligned} \tag{12}$$

3 Conclusion

In this walkthrough, we covered the basics of breaking a timeline into activities, intervals, and bounds. We looked at interval math. We walked through the All Pairs Shortest Paths (Floyd-Warshall) algorithm to uncover implicit constraints between activities. We looked at an example of using an STN to update constraints as activities are executed and as-performed times are collected. And finally we looked at a method for ensuring STNs are feasible.

To dig deeper, check out the resources below. I also recommend looking up "flexible execution" as it extends the ideas here in the context of changing constraints. I also recommend looking up Simple Temporal Networks with Uncertainty for contexts where constraints aren't as well known.

References

[1] R. Dechter, I. Meiri, and J. Pearl, "Temporal Constraint Networks," *Artificial Intelligence*, vol. 49, pp. 61–95, 1991.

[2] T. Vidal and H. Fargier, "Handling Contingency in Temporal Constraint Networks: From Consistency to Controllabilities," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 11, no. 1, pp. 23–45, 1999, ISSN: 13623079. DOI: 10.1080/095281399146607.

[3] P. Morris and N. Muscettola, "Temporal dynamic controllability revisited," *Proceedings of the National Conference on Artificial Intelligence*, vol. 3, pp. 1193–1198, 2005.

[4] J. Allen, "Maintaining knowledge about temporal intervals. communications of ACM, Vol. 26, No. 11, 832–843," vol. 26, no. 11, pp. 832–843, 1983. [Online]. Available: http://scholar.google.com/scholar?q=related:SfE4Y6t68aMJ:scholar.google.com/%7B%5C&%7Dhl=en%7B%5C&%7Dnum=20%7B%5C&%7Das%7B%5C_%7Dsdt=0,5.

[5] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization:Algorithms and complexity*, 6. 1982, vol. 32, pp. 1258–1259. DOI: 10.1109/tassp.1984.1164450.