2018

Evaluating Flexibility Metrics on Simple Temporal Networks with Reinforcement Learning

Hamzah I. Khan
Harvey Mudd College

Evaluating Flexibility Metrics on Simple Temporal Networks with Reinforcement Learning

Hamzah I. Khan

Susan E. Martonosi, Advisor

James C. Boerkoel, Jr., Reader

Department of Mathematics

May, 2018

Abstract

Simple Temporal Networks (STNs) were introduced by Tsamardinos (2002) as a means of describing graphically the temporal constraints for scheduling problems. Since then, many variations on the concept have been used to develop and analyze algorithms for multi-agent robotic scheduling problems. Many of these algorithms for STNs utilize a flexibility metric, which measures the slack remaining in an STN under execution. Various metrics have been proposed by Hunsberger (2002); Wilson et al. (2014); Lloyd et al. (2018). This thesis explores how adequately these metrics convey the desired information by using them to build a reward function in a reinforcement learning problem.

Contents

Abstract **iii**

Acknowledgments **xi**

1 Introduction **1**
- 1.1 Scheduling as a Problem 2

2 Representations of Simple Temporal Networks **5**
- 2.1 Scheduling with Simple Temporal Networks 5
- 2.2 Representations for Simple Temporal Networks 7
- 2.3 Properties of Simple Temporal Networks 11
- 2.4 Conclusion 16

3 Flexibility as a Measure of Slack **17**
- 3.1 Sequential and Concurrent Classes of STNs 17
- 3.2 Desiderata for Flexibility Metrics 19
- 3.3 Flexibility Metrics 20

4 Introduction to Reinforcement Learning **25**
- 4.1 Motivating the Reinforcement Learning Problem 25
- 4.2 Components of Reinforcement Learning Problems 26
- 4.3 Value Estimates & Tabular Monte Carlo Methods 32
- 4.4 Applying Reinforcement Learning to Simple Temporal Networks 34

5 Empirical Evaluation of Flexibility Metrics **37**
- 5.1 Motivation 37
- 5.2 Reinforcement Learning for Scheduling in STNUs 38
- 5.3 Running Through Two Examples 43

5.4 Results 50
5.5 Conclusions 50

Bibliography **51**

List of Figures

1.1 The output of a scheduling algorithm can be visualized as a timeline. In the image above, the boxes represent events that have been scheduled by assigning them times at which they should occur. The brackets represent the constraints given to the scheduler - they dictate the intervals of time (i.e. domains) within which the events must be scheduled. Note that domains for events can overlap. 3

2.1 The simple temporal network from our refueling example can be interpreted geometrically as shown. The red lines on the graph indicate absolute constraints and the black, dotted ones indicate relative constraints. The resulting region describes which values for t_1 and t_2 form valid schedules. The space of valid schedules is the highlighted region in green. Each point in this feasible region represents a valid schedule. 8

2.2 The simple temporal network from our refueling example can be interpreted as a directed distance graph as shown. As in Figure 2.1, absolute constraints are denoted by the color red, and relative constraints are by the color black. Vertices $0, 1,$ and 2 respectively represent the events of the truck arriving, beginning the refueling process, and ending it. The edge from event 0 to event 1 can be interpreted as requiring event 1 to occur at most 5 minutes after event 0 (i.e. $t_1 - t_0 \leq 5$). 9

2.3 The simple temporal network from our refueling example can be interpreted as a compact distance graph as shown. As in Figures 2.1 and 2.2, absolute constraints (i.e. self-loops) are denoted by the color red, and links representing relative constraints colored black. The zero event, the arrival of the truck, is not shown, but vertices 1 and 2 respectively represent the beginning and end of the refueling process. The link from event 1 to event 2 can be interpreted as requiring event 2 to occur between 0 and 5 minutes after event 1 (i.e. $0 \leq t_2 - t_1 \leq 5$). 10
2.4 A comparison of a feasible and infeasible STN. 12
2.5 A comparison of a minimal and nonminimal STN. 13
2.6 A simple temporal network under uncertainty, with two contingent edges between events pairs (1, 2) and (3, 4). There is a constraint that events 2 and 4 must end within 2 time steps of one another. The contingent edges take a uniformly distributed duration, which makes scheduling more difficult. 15

3.1 Two different STNs for the truck delivery example defined in Chapter 3. The values boxed in red are those which differ between the two examples, marking the distinction between concurrent and sequential STNs. 19
3.2 The orange triangle contains a smaller inscribed circle than does the green triangle (both representing feasible regions of arbitrary STNs), implying that it has a lower sphere flexibility 22

4.1 A feedback loop describing the order of events in a reinforcement learning system. The agent selects an action based on its observations of the environment. The action changes the environment in some way. Then, the agent receives some reward signal that indicates how good the new state is (i.e. is it a goal state or a failure state). Image taken from https://en.wikipedia.org/wiki/Reinforcement_learning#/media/File:Reinforcement_learning_diagram.svg . . . 30

5.1 At $t = 0$, the state of the agent $S_0 = (\{1, 3\}, \{0\})$. The agent selects action $A_0 = 3$ from the action space $\mathcal{A}(S_0) = \{\emptyset, 1, 3\}$. 44

5.2 At $t = 1$, the agent gets a reward $R_1 = 0$ since no events were executed at time 0. The state of the agent $S_1 = (\{1\}, \{0, 3\})$. The agent then chooses action $A_1 = \emptyset$ from the action space $\mathcal{A}(S_1) = \{\emptyset, 1\}$. 44
5.3 At $t = 2$, the agent gets a reward $R_2 = 2$ since contingent event 4 was executed in the previous time step. The state of the agent $S_2 = (\{1\}, \{0, 3, 4\})$. The agent then chooses action $A_2 = \emptyset$ from the action space $\mathcal{A}(S_2) = \{\emptyset, 1\}$. 45
5.4 At $t = 3$, the agent gets reward $R_3 = 0$ since no events were completed in $t = 2$. The state of the agent $S_3 = (\{1\}, \{0, 3, 4\})$. The agent then chooses action $A_3 = \emptyset$ from the action space $\mathcal{A}(S_3) = \{\emptyset, 1\}$. 45
5.5 At $t = 4$, the execution fails as a constraint is violated, so the reward is $R_4 = -16$. 46
5.6 At $t = 0$, the state of the agent $S_0 = (\{1, 3\}, \{0\})$. The agent then chooses action $A_0 = 1$ from the action space $\mathcal{A}(S_0) = \{\emptyset, 1, 3\}$. 47
5.7 At $t = 1$, the agent gets a reward $R_1 = 3$ since event 1 was executed at time 0. The state of the agent $S_1 = (\{3\}, \{0, 1\})$. The agent then chooses action $A_1 = \emptyset$ from the action space $\mathcal{A}(S_1) = \{\emptyset, 3\}$. 47
5.8 At $t = 2$, the agent gets a reward $R_2 = 0$ since task 3 has been executed. The state of the agent $S_2 = (\{\}, \{0, 1, 3\}$. The agent then chooses action $A_2 = \emptyset$ from the action space $\mathcal{A}(S_2) = \{\emptyset\}$. 47
5.9 At $t = 3$, the agent gets a reward $R_3 = 2$ since task 1 has been executed. The state of the agent $S_3 = (\{\}, \{0, 1, 3\}$. The agent then chooses action $A_3 = \emptyset$ from the action space $\mathcal{A}(S_3) = \{\emptyset\}$. 48
5.10 At $t = 4$, the agent gets a reward $R_4 = 1$ since contingent event 2 has concluded in time step 3. The state of the agent $S_4 = (\{\}, \{0, 1, 2, 3\}$. The agent then chooses action $A_4 = \emptyset$ from the action space $\mathcal{A}(S_4) = \{\emptyset\}$. 48
5.11 At $t = 5$, the agent gets a reward $R_5 = 0$ since contingent event 4 has concluded in time step 4. The state of the agent $S_4 = (\{\}, \{0, 1, 2, 3, 4\}$. The episode then concludes since the schedule has been successfully executed. 48

Acknowledgments

First and foremost, I'd like to thank God for giving me the opportunities I have received at Harvey Mudd and the energy to pursue them. Secondly, I'd like to thank my parents, Imtiaz and Humaira Khan, for their constant support and love throughout college (and my life).

I'd like to thank my advisor Susan Martonosi for her support of my growth as a researcher through this thesis process. I'd also like to thank my second reader, Jim Boerkoel, for introducing me to this field of study and encouraging my growth as a researcher in robotics. Thank you to my peers in the HEATlab and on the NASA Ames Clinic team, especially Jordan Abrahams, Marina Knittel, Amy Huang, Liam Lloyd, and Brenner Ryan, for their feedback and help throughout the year.

Finally, thank you to Lisette de Pillis and all of my classmates in math thesis for sitting through my presentations and providing feedback.

Chapter 1

Introduction

A disaster strikes! New Exampletown has been hit with a massive earthquake and requires aid immediately in order to search for survivors in the wreckage. As a drone network operator, you have been given the task of using this swarm of unmanned aerial robots to locate as many people as possible. Your central computer relays commands and receives position and timing updates. Each drone collects information vital to these efforts, but due to damaged infrastructure, the large amounts of data can only be shared when two or more drones are within 10 feet of one another or the central computer. In order to circumvent this difficulty, you assign each robot to a set of apartment buildings to explore the relevant areas. To maximize the efficiency of the search, you constrain the times at which each building search can start or end.

This example illustrates a classic constraint-based scheduling problem, where the drone network operator (you) must assign (i.e. schedule) the start and end times for searching each building and set up precisely timed meetings between your drones to maximize saving lives.

In this situation, you might wonder which data transfer strategy would be best. Should you favor large meetups, where one of k drones collects information from the others and brings it to the central location? Or should you instead consider a more natural data transfer strategy in which two robots that happen to be near one another share data at convenient times, where data would propagate through the network back to the central computer? More importantly, which strategy best maintains your drones' ability to satisfy the original schedule in the event of an unexpected delay?

This circumstance illustrates a need for scheduling. This thesis aims to build on research done for using simple temporal networks (STNs) in scheduling problems. It attempts to evaluate flexibility metrics over STNs with reinforcement learning.

In the remainder of Chapter 1, we present an overarching scenario from which we will consider smaller examples in later chapters. In Chapter 2, we review existing literature regarding the definition and representations of simple temporal networks. Chapter 3 introduces prerequisite knowledge about flexibility, a metric that measures slack within simple temporal networks. Chapter 4 briefly discusses reinforcement learning, and Chapter 5 builds on this knowledge with a design of an experiment, as well as future work and next steps for the project. A basic familiarity with graph theory, operations research, and probability is assumed for the content of this work.

1.1 Scheduling as a Problem

For the purposes of this thesis, we will refer to examples relating to OnTime Trucking, whose motto is "We'll deliver it, as long as our scheduling works perfectly!" Consider the case of a single truck driver delivering cars from a supplier in San Francisco to a dealership in Portland, OR. If our driver drove without break at 60 miles per hour, such a trip would take 12 hours, a difficult trip to make in one sitting. In addition, the truck would need to refuel and the driver would need to eat and relieve herself at these stops. Suppose we divided up the trip into segments between San Francisco, gas stations, and Portland. We would then face the question of when, given the distances between stops and the amount of rest our driver might require, we might attempt to plan when our driver should reach each of these stops, and when she should leave. Thinking more abstractly for a moment, we might consider classifying each decision to leave (beginning from the decision in San Francisco) and each decision to stop (ending with our destination in Portland) as **events** indexed from 1 to n. We can then define a **schedule** over a set of events to be an assignment of times $\vec{t} \in \mathbb{R}^n$ at which each event (i.e. stopping for gas, leaving) during the driver's trip should happen. Once we have such a schedule, we can imagine plotting the times assigned to each event onto a timeline, as in Figure 1.1.

Such a schedule could have been produced by a different actor, perhaps another employee, working for the same trucking company. In this case, the employee that designs the schedules for truck drivers might be considered a **scheduler**. The truck company, after signing the contract to deliver these

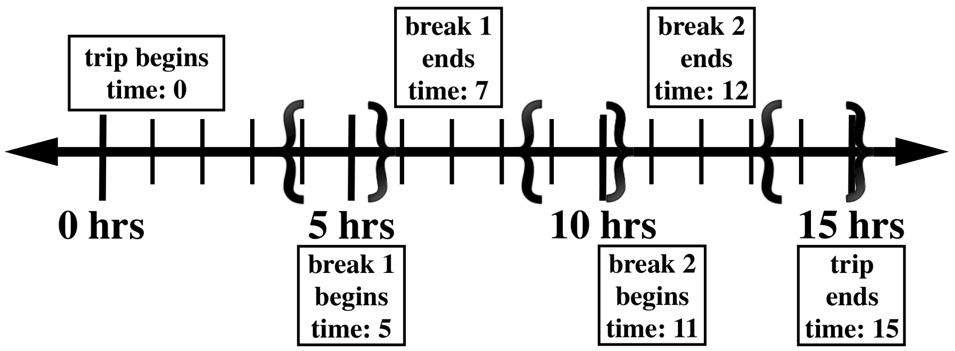


Figure 1.1 The output of a scheduling algorithm can be visualized as a timeline. In the image above, the boxes represent events that have been scheduled by assigning them times at which they should occur. The brackets represent the constraints given to the scheduler - they dictate the intervals of time (i.e. domains) within which the events must be scheduled. Note that domains for events can overlap.

cars, would request a schedule from the scheduler for a trip consisting of these events. Furthermore, the truck company might provide a number of **constraints** on the resulting schedule. It could be that the delivery would need to be made within one day of the driver departing, the driver might need enough chances for rest (e.g. two 1-hour breaks) along the way, or the driver might need to travel on average between 50 and 60 miles per hour on the highway, among others. A schedule is **valid** if the time assigned to each event satisfies all of the temporal constraints. The space of times assignable to each event is called its **domain**. Once the truck company receives a valid schedule, it becomes responsible to execute it by having the truck driver to follow the schedule. The truck company is an example of an **executive** because it is in charge of monitoring the schedule (using drivers as a means to do so).

In a perfect world, the driver would follow this schedule exactly. However, in our uncertain world, she might run into traffic that delays her or she might arrive early. In both of these cases, she might not be able to perfectly follow the schedule. Once she informs her superiors of her situation, the truck company (i.e. the executive) might request a new schedule from the scheduler, who would then reschedule the trip and adapt to the new conditions that the truck driver faces. We can easily imagine expanding this system to allow for scheduling the routes of many truck drivers.

In summary, the executive requests and monitors the execution of a schedule subject to temporal constraints, and the scheduler identifies a schedule that satisfies all of those constraints. When the schedule is executed, but the realized time for an event differs with the scheduled time, the schedule may have been violated. If the realized time has not violated the constraints of the schedule, then a new schedule from the space of remaining possible schedules is required to adjust.

Chapter 2

Representations of Simple Temporal Networks

Before we can discuss multiagent scheduling, we must first consider the existing methods to solve constraint-based scheduling problems. In this chapter, we introduce a data structure that has been used in the scheduling literature to model deterministic, constraint-based scheduling problems. This chapter summarizes work from Dechter et al. (1991); Vidal and Ghallab (1996); Tsamardinos (2002); Wilson et al. (2014); Lund et al. (2017).

Recall that our example trucking company from Chapter 1, OnTime Trucking, requires scheduling for many of its daily tasks. OnTime Trucking has recently struck a deal with SuperFast Gas to prioritize refueling their trucks. As part of the deal, SuperFast Gas has promised that all trucks will be refueled within five minutes of arrival. However, they need help from OnTime Trucking to schedule the refueling process to satisfy this promise.

2.1 Scheduling with Simple Temporal Networks

The refueling process consists of two events once a truck arrives at a station: beginning the fueling process and removing the pump from the truck upon completion. Both of these events must occur within five minutes of the truck arriving. Furthermore, we note that ending the fueling process must occur after the beginning of the fueling process. Through describing the refueling process, we have identified a set of events, each of which must be assigned times to create a schedule and a set of temporal constraints on the times that can be assigned to those events. However, note that there are still many possible schedules that can satisfy these constraints. We can imagine that there exists a structure describing the space of schedules that satisfy all constraints. To this end, Dechter et al. (1991) introduce simple temporal networks to describe the space of valid solutions.

Definition 2.1. *Dechter et al.* (1991) *defines a **simple temporal network** (STN) to be a* 2*-tuple* $\langle T, C\rangle$*, where*

- T *denotes the set of **events*** $0, 1, \ldots, n \in T$ *to be scheduled at times* $t_0, t_1, \ldots, t_n$ *subject to*
- ***temporal difference constraints*** $c_{ij} \in C$*, each of which represent a bound on the elapsed time* $t_j - t_i \leq b_{ij}$ *between events* i *and* j*. Note that* $b_{ij} \in \mathbb{R}$*.*

In addition, we typically describe event 0 as the **zero event** of the temporal network. The zero event always occurs at time $t_0 = 0$, which grounds the schedule against a clock time. For our example, the zero event might correspond to a truck entering the refueling station.

We can see that the set of valid schedules $\{[t_1, \ldots, t_n]^T \in \mathbb{R}^n | t_i \in T, c_{jk} \in C\}$ for refueling can then be described as a simple temporal network represented equivalently as the following constraint satisfaction problem. This example of an STN was first introduced by Wilson et al. (2014), and will be used throughout the remainder of Chapter 2 as a motivating example of an STN.

$$\begin{aligned} T &= \{0, 1, 2\} \\ C &= \{t_1 - t_0 \leq 5, && \text{(refueling begins under 5 minutes after truck enters)} \\ &\quad t_0 - t_1 \leq 0, && \text{(truck enters prior to the start of refueling)} \\ &\quad t_2 - t_0 \leq 5, && \text{(refueling ends under 5 minutes after truck enters)} \\ &\quad t_0 - t_2 \leq 0, && \text{(truck enters prior to the end of refueling)} \\ &\quad t_2 - t_1 \leq 5, && \text{(refueling ends under 5 minutes after starting)} \\ &\quad t_1 - t_2 \leq 0, && \text{(refueling must begin prior to ending)} \end{aligned}$$

From the interpretations of each constraint, we can see that some constraints refer to the clock time (i.e. the zero event at which the truck enters the refueling station). For the remainder of this work, we call these constraints, which are between t_0 and t_i, **absolute temporal constraints**, absolute constraint for short. Absolute constraints explicitly limit the domain of possible times for an event (i.e. "event i must occur between 3pm and 4pm" corresponds to $3 \leq t_i - t_0 \leq 4$). Constraints between other events i and j which do not reference a clock time are then called **relative temporal constraints** because they implictly constrain the domain of possible times for the events in question (i.e. "event j must occur at most 5 minutes after i" corresponds to $t_j - t_i \leq 5$).

2.2 Representations for Simple Temporal Networks

In this section, we review other, equivalent representations of STNs that allow us to draw upon other bodies of mathematical knowledge when analyzing STNs. In total, STNs can be described by four representations. STNs have been represented in the literature geometrically as convex polyhedra (Lloyd et al., 2018) and in graphical form as directed, weighted graphs (Dechter et al., 1991; Wilson et al., 2014). The graphical form can be described more compactly and abstractly as a fourth representation (Lund et al., 2017), which I refer to as the compact graphical representation (Tsamardinos, 2002; Vidal and Ghallab, 1996; Lund et al., 2017).

2.2.1 Constraint Satisfaction Representation

In the refueling example, we represent the STN as a constraint satisfaction problem (CSP). Doing so allows us to draw upon concepts (i.e. feasibility) relating to CSPs. Note, with respect to the CSP representation, that each constraint is a linear inequality between the times assigned to two events.

$$\begin{aligned}
T &= \{0, 1, 2\} \\
C &= \{t_1 - t_0 \leq 5, && \text{(refueling begins under 5 minutes after truck enters)} \\
&\quad t_0 - t_1 \leq 0, && \text{(truck enters prior to the start of refueling)} \\
&\quad t_2 - t_0 \leq 5, && \text{(refueling ends under 5 minutes after truck enters)} \\
&\quad t_0 - t_2 \leq 0, && \text{(truck enters prior to the end of refueling)} \\
&\quad t_2 - t_1 \leq 5, && \text{(refueling ends under 5 minutes after starting)} \\
&\quad t_1 - t_2 \leq 0, && \text{(refueling must begin prior to ending)}
\end{aligned}$$

However, we can also represent STNs in other ways.

2.2.2 Geometric Representation

Since each constraint is a linear inequality between the times assigned to two events, the entire problem can be described as a linear system of inequalities. Lloyd et al. (2018) used this description to visualize the system of linear inequalities geometrically. Consider the region described by our refueling example, as shown in Figure 2.1.

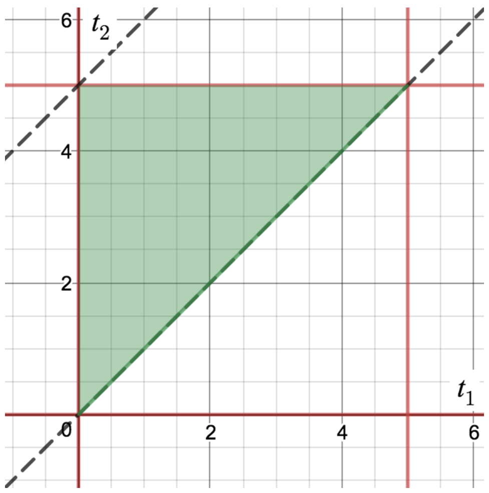


Figure 2.1 The simple temporal network from our refueling example can be interpreted geometrically as shown. The red lines on the graph indicate absolute constraints and the black, dotted ones indicate relative constraints. The resulting region describes which values for t_1 and t_2 form valid schedules. The space of valid schedules is the highlighted region in green. Each point in this feasible region represents a valid schedule.

Lloyd et al. (2018) noted that the shape formed by the intersection of all these linear inequalities forms a convex polygon in two dimensions (i.e. two events) with finite area. Furthermore, this concept generalizes for n events, where the linear inequalities act as hyperplanes that bound an n-dimensional convex polyhedron. Note that the space is convex and any point in this feasible region represents a valid schedule. This representation allows us to leverage results concerning the geometry of linear programming feasible regions.

2.2.3 Distance Graph Representation

Recall that each constraint in a simple temporal network denotes an upper bound on the amount of time that can elapse between two events. Such a description of time can be thought of as a maximum distance between events, and generalized for events that are not adjacent. Using this train of thought, we can represent an STN as a directed graph with edges weighted by distance (Dechter et al., 1991; Wilson et al., 2014).

Given an STN $\langle T, C \rangle$, we then define the distance graph representation as graph $\langle T, E \rangle$ such that each event $i \in T$ is represented by a vertex in the graph. In order to define edges in this graph, consider an edge $e_{ij} \in E$ from event i to event j in the graph. This edge represents the constraint c_{ij}, which takes the form $t_j - t_i \leq b_{ij}$ for some $b_{ij} \in \mathbb{R}$. Since an edge e_{ij} already indicates a relationship between i and j, we can then weight e_{ij} with the distance b_{ij} in order to form our directed graph and preserve all information between the two representations. Figure 2.2 is the distance graph representation of our refueling example.

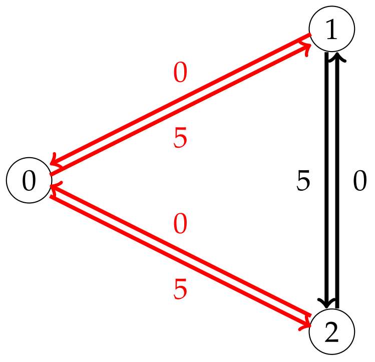


Figure 2.2 The simple temporal network from our refueling example can be interpreted as a directed distance graph as shown. As in Figure 2.1, absolute constraints are denoted by the color red, and relative constraints are by the color black. Vertices 0, 1, and 2 respectively represent the events of the truck arriving, beginning the refueling process, and ending it. The edge from event 0 to event 1 can be interpreted as requiring event 1 to occur at most 5 minutes after event 0 (i.e. $t_1 - t_0 \leq 5$).

This representation allows us to draw on graph theoretical knowledge to sovle STNs.

2.2.4 Compact Graphical Representation

The distance graph representation can be made more concise (Lund et al., 2017). In the distance graph representation of a simple temporal network, each pair of edges $e_{ij}, e_{ji} \in E$ represent upper and lower bounds on the times of the same two events. The constraints formed by e_{ij} and e_{ji} are

$$t_j - t_i \leq b_{ij}$$
$$t_i - t_j \leq b_{ji},$$

which is more succintly described as

$$-b_{ji} \leq t_j - t_i \leq b_{ij}.$$

For this reason, e_{ij} and e_{ji} can also be more compactly represented by a single edge l_{ij}, without loss of generality, from event i to event j. To avoid confusion with edges in the directed graphical representation, we shall refer to the edge l_{ij} as a **link**. The link could then be labelled $[-b_{ji}, b_{ij}]$ for conciseness.

Furthermore, we remove the zero event from the graph. An absolute constraint on event i can then be represented as a link from i to itself, a **self-loop**. In this case, the link would be labelled $[-b_{i0}, b_{0i}]$ and be interpreted identically to the labels on links representing relative edges. Note that these self-loops limit our ability to use graph theoretical concepts directly on this compact graph. However, this graph improves the legibility of an STN.

The STN for the refueling problem is shown in compact graphical form in Figure 2.3.

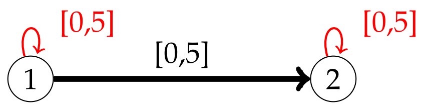


Figure 2.3 The simple temporal network from our refueling example can be interpreted as a compact distance graph as shown. As in Figures 2.1 and 2.2, absolute constraints (i.e. self-loops) are denoted by the color red, and links representing relative constraints colored black. The zero event, the arrival of the truck, is not shown, but vertices 1 and 2 respectively represent the beginning and end of the refueling process. The link from event 1 to event 2 can be interpreted as requiring event 2 to occur between 0 and 5 minutes after event 1 (i.e. $0 \leq t_2 - t_1 \leq 5$).

We note that all of the STN representations described thus far are equivalent: an STN is a set of constraints between the times assigned to events just as much as it is a convex polyhedron in n-dimensional space and a directed distance graph with weighted edges that each describe the upper bounds on the duration between pairs of events. It is important that we are able to analyze a given STN using all of its representations.

2.3 Properties of Simple Temporal Networks

In this section, we describe two important properties of simple temporal networks, referring to our refueling example to motivate the properties.

2.3.1 Consistency

Definition 2.2. *An STN is **consistent**, or **feasible**, if there exists an assignment of times to all events that satisfies all constraints. Identically, a consistent STN contains a valid schedule. If an STN is not consistent, then it is called **inconsistent** or **infeasible**.*

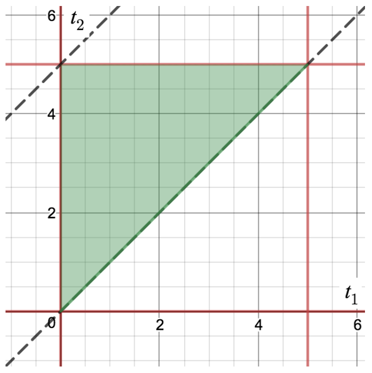
Let S be the STN that describes the refueling scheduling problem. One schedule that satisfies all constraints within S is

$$t_0 = 0, \qquad t_1 = 2.5, \qquad t_2 = 5,$$

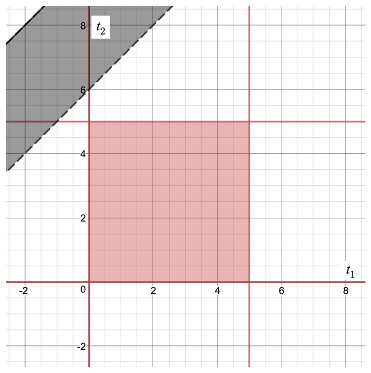
as shown below.

$$\begin{aligned} C = \{t_1 - t_0 &= 2.5 \leq 5, && \text{(refueling begins under 5 minutes after truck enters)} \\ t_0 - t_1 &= -2.5 \leq 0, && \text{(truck enters prior to the start of refueling)} \\ t_2 - t_0 &= 5.0 \leq 5, && \text{(refueling ends under 5 minutes after truck enters)} \\ t_0 - t_2 &= -5.0 \leq 0, && \text{(truck enters prior to the end of refueling)} \\ t_2 - t_1 &= 2.5 \leq 5, && \text{(refueling ends under 5 minutes after starting)} \\ t_1 - t_2 &= -2.5 \leq 0\} && \text{(refueling must begin prior to ending)} \end{aligned}$$

Therefore, since we have found a valid schedule in the space of schedules described by this STN, this STN is consistent. Now, suppose we replace the label on the link between events 1 and 2 with the label $[6, 10]$ to create a different STN S'. Figure 2.4 illustrates the polyhedra representing S and S'. Because there are no points in S', the volume of its polyhedron is 0; S' is



a. Polyhedral representation of STN S: The green shaded area indicates that the polyhedron formed by the STN contains valid schedules. This STN is feasible.



b. Polyhedral representation of STN S': The black shaded area is the space of solutions satisfying only the relative constraints. The red shaded area is the space of solutions satisfying only the absolute constraints. Since neither of them overlap, then the space of solutions satisfying the STN is empty; the polyhedron has no volume.

Figure 2.4 A comparison of a feasible and infeasible STN.

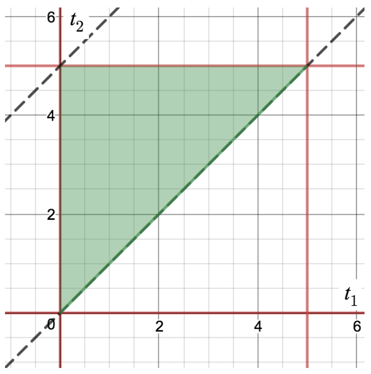
inconsistent and has no valid schedules. Dechter et al. (1991) and Boerkoel et al. (2012), among others, have introduced graph-based algorithms to confirm consistency within STNs.

2.3.2 Minimality

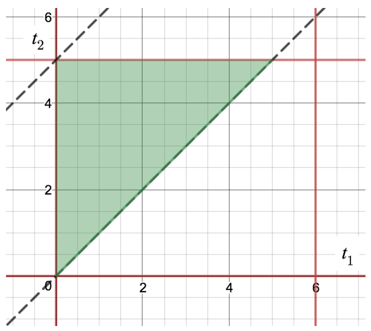
Definition 2.3. *A simple temporal network is **minimal** if every event i can be assigned any time t_i within its domain such that there exists a valid schedule containing t_i. That is, there is no event i for which the selection of some t_i in its domain would force a failed schedule. For a minimal STN, if event i is scheduled, then there exist scheduling assignments for all other events that satisfy all constraints. By contrapositive, if there does not exist a valid schedule containing t_i, then it is not within the valid domain of times for event i in the minimal STN.*

Figure 2.5 illustrates the concept of minimality. The polyhedral representation of STN S shown in Figure 2.5a is minimal: if any constraint boundaries are pushed inward (corresponding to eliminating elements of an event's domain), valid schedules are removed from the polyhedron. By contrast, STN S'' shown in Figure 2.5b is not minimal: the constraint boundary $t_1 = 6$

can be pushed inward without removing valid schedules from the interior of the polyhedron.



a. Polyhedral representation of STN S: Note that no constraint boundary can be "pushed" inward without eliminating some valid schedules from the STN. Therefore, this STN is minimal.



b. Polyhedral representation of STN S'': Note that the constraint boundary $t_1 = 6$ can be "pushed" inward to be $t_1 = 5$ without eliminating any valid schedules from the STN. Since we can perform such an operation, this STN is **not** minimal.

Figure 2.5 A comparison of a minimal and nonminimal STN.

2.3.3 Extensions of Simple Temporal Networks

Simple temporal networks provide a powerful way to describe scheduling problems which are deterministic, ignoring details such as the uncertainty in the real world. In this section, we describe an extension to STNs from the literature that better describes a stochastic model of scheduling.

2.3.4 Representing Uncertainty within Simple Temporal Networks

Activities within the real world are subject to uncertainties that are not modeled well by the formulation of STNs that has been described thus far. Current extensions to STNs include simple temporal networks with uncertainty and probabilistic simple temporal networks (Vidal and Ghallab (1996); Tsamardinos (2002)). Both representations tie uncertainty to the idea of executability, that agents don't necessarily control the duration of every activity (i.e. the amount of time to cross an edge in the distance graph). For example, Nature might (through slippage, localization error, etc.) change the amount of time executing an activity requires. We rely heavily on the compact graphical representation for the following discussion of STNs that model uncertainty.

Consider events $i, j \in T$ constrained such that event i directly precedes event j. If the duration of event i is random, then once event i is executed at time t_i, we cannot directly assign the execution time, t_j, of event j. Thus, the constraint that $-b_{ji} \leq t_j - t_i \leq b_{ij}$ is not directly controllable by the agent executing the schedule. In such a case, we refer to constraint c_{ij} as a **contingent constraint**. Since the realized execution time for event j would then also not be directly controllable, we describe it as a **contingent event**. Constraints for which the duration can be controlled are called **requirement constraints**, and events which are controllable are called **executable**.

We denote the set of

- contingent constraints as C_C,
- contingent events as T_C,
- requirement (i.e. temporal difference) constraints as C_R, and
- executable events T_X,

where the set of events $T = T_C \cup T_X \cup \{0\}$.

Simple Temporal Networks with Uncertainty

A **simple temporal network with uncertainty** (STNU) (Vidal and Ghallab, 1996; Morris et al., 2001) is defined as a tuple $\langle T_X, T_C, C_R, C_C \rangle$ where the set of all events $T = T_C \cup T_X \cup \{0\}$ and the set of all constraints $C = C_R \cup C_C$. Note that for contingent constraints in an STNU, we have one interval between every pair of vertices that describes the range, not necessarily uniformly-distributed, of possible durations for the activity described by the link. Note that we could transform an STNU into an STN by ignoring the distinction between contingent and requirement constraints and events. We show an example STNU in Figure 2.6.

2.3.5 Dispatch Strategies

Recall that feasibility allows us to understand whether an STN describes a space with at least one valid schedule. When dealing with uncertainty, feasibility may not be a strong enough condition for us to guarantee solutions

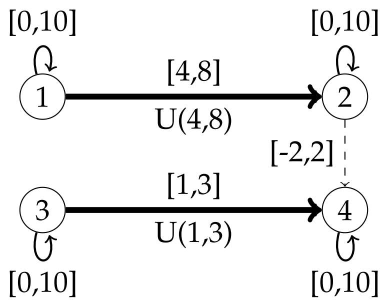


Figure 2.6 A simple temporal network under uncertainty, with two contingent edges between events pairs (1, 2) and (3, 4). There is a constraint that events 2 and 4 must end within 2 time steps of one another. The contingent edges take a uniformly distributed duration, which makes scheduling more difficult.

in STNUs. For this reason, we introduce additional properties that guarantee these properties and help characterize the circumstances under which an STNU is solvable.

Controllability in Uncertain STNs

An STNU is **strongly controllable** if any assignment of values for executable events is guaranteed to be consistent under any realization of uncertainty along contingent edges. The strong controllability of an STNU is checkable beforehand via an algorithm (Lund et al., 2017).

An STNU is **dynamically controllable** if executable events can be assigned values online during execution and is guaranteed to be successful. The dynamic controllability of an STNU is checkable beforehand via an algorithm (Lund et al., 2017).

An executable event i is **live** if the current time is within its bounds. i is **enabled** if it can be executed at the current time without violating any constraints (such as all previous points having been executed).

A contingent event i is **live** if the event preceding its contingent link has been executed. Note that contingent links assign times to their end events.

We assume dynamic controllability for the remainder of this document, which implies that we are able to do online scheduling, which would be required for our reinforcement learning setup described in Chapter 5.

2.4 Conclusion

In this section, we have described a variety of previous work delineating the definitions and representations of simple temporal networks and simple temporal networks under uncertainty and a few of their properties. From this review, and especially through the description of multiple representations, we have expanded the number of tools with which we can analyze STNs and STNUs.

Chapter 3

Flexibility as a Measure of Slack

Thus far, we have introduced simple temporal networks as a data structure that represents a solution space of valid schedules. Many online algorithms for solving STNs and STNUs during execution rely on sequential approaches that cannot account for future events. For any given decision to begin executing an event i at time t_i, it is unknown whether this decision will contribute to the schedule becoming inconsistent at a future time. Thus, we need a tool that quantifies the likelihood that the remaining events in a partially executed STN or STNU still define a consistent scheduling problem. Previous works have used the concept of **flexibility** to quantify this value (Hunsberger, 2002; Wilson et al., 2014; Lloyd et al., 2018). However, each proposed flexibility metric fails to capture some important element of this likelihood. In this chapter, we review two classes of STNs, sequential and concurrent, that were introduced by Wilson et al. (2014) and used by Lloyd et al. (2018) in defining desiderata, or desired properties for flexibility metrics.

3.1 Sequential and Concurrent Classes of STNs

Both Wilson et al. (2014) and Lloyd et al. (2018) use two specific extremal classes of STNs in order to develop intuitions about flexibility. We draw the example STNs in this section from those papers. As motivation, consider OnTime Trucking's service in which its trucks move containers from the dock to desired destinations within the city. Suppose that OnTime Trucking has guaranteed that it delivers within 5 hours of a container being ready for shipment. Suppose that we have a single truck that must deliver 3 packages. In an STN describing the schedule, we might have 3 nonzero events. Furthermore, we might guarantee that each nonzero event occurs within 5 hours of the zero event and that each pair of nonzero events occur within the same five hour timespan. An example of this STN is shown in Figure 3.1a. We can define a class of STNs that generalizes this concurrent structure.

Definition 3.1. *Let $N = \langle T, C\rangle$ be an STN, and let $d, n \in \mathbb{R}$. Then, the class of* ***concurrent*** *STNs $\mathscr{C}$ consist of STNs where for each pair of nonzero events $i, j \in T$, the relative constraint $c_{ij} \in C$ exists and is described as*

$$-d \leq t_j - t_i \leq d$$

and, for each nonzero event i, the absolute constraint $c_{0i} \in C$ exists and is described as

$$0 \leq t_i - t_0 \leq d.$$

We denote N as $\mathscr{C}_{dn}$ if N is a concurrent STN with n events and a maximum duration of d. All events can happen at any time between 0 and d.

Alternatively, the company might instead desire a sequential structure in which the second and third deliveries would occur within five hours of the first and the third within five hours of the second. The absolute constraint would still indicate a maximum five hour duration for the deliveries. An example of this STN is shown in Figure 3.1b. We can define a second class of STNs that generalizes this sequential structure.

Definition 3.2. *Let $M = \langle T, C\rangle$ be an STN, and let $d, n \in \mathbb{R}$. Then, the class of* ***sequential*** *STNs $\mathscr{S}$ consist of STNs where T is ordered such that for each nonzero event $i \in T$, the relative constraint $c_{ij} \in C$ exists and is described as*

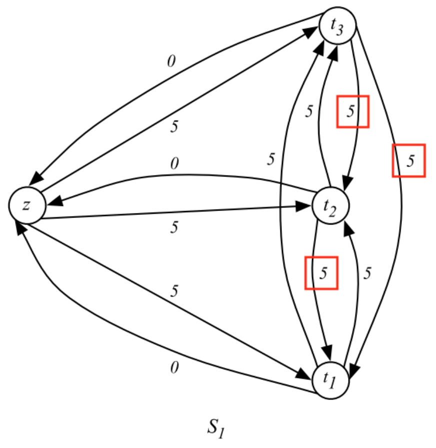
$$0 \leq t_j - t_i \leq d \qquad \text{(where } i < j \in T\text{)}$$

and the absolute constraint $c_{0i} \in C$ exists and is described as

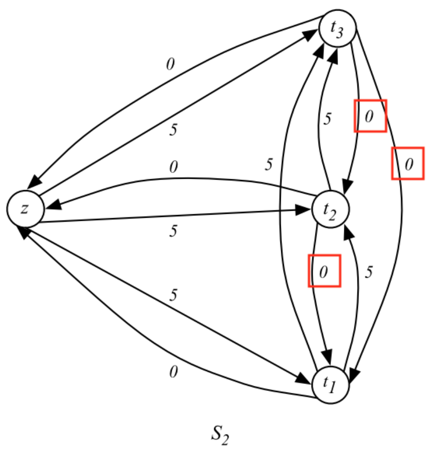
$$0 \leq t_i - t_0 \leq d.$$

We denote M as $\mathscr{S}_{dn}$ if M is a sequential STN with n events and a maximum duration of d.

Note that these two classes do not cumulatively describe the entire set of simple temporal networks, but two significant (albeit small) classes within it. In the next section, we describe how Lloyd et al. (2018) uses these classes to propose the desiderata for flexibility.



a. An example of an STN from the concurrent class of STNs (Wilson et al., 2014).



b. An example of an STN from the sequential class of STNs (Wilson et al., 2014).

Figure 3.1 Two different STNs for the truck delivery example defined in Chapter 3. The values boxed in red are those which differ between the two examples, marking the distinction between concurrent and sequential STNs.

3.2 Desiderata for Flexibility Metrics

Lloyd et al. (2018) proposed four desiderata for flexibility metrics. This section introduces their desiderata, with their context for why each desideratum is necessary to match the geometric inutition we hold for STNs.

3.2.1 Simplicity

Simplicity captures the idea that increasing the number of events in an STN implies that more events have the possibility of failing, which reduces flexibility.

Definition 3.3. (*Lloyd et al.,* 2018) *If S is an STN, and we add an event to create a new STN S', where the new event is independent (no constraints on it exist except the absolute constraint) from all existing events, then*

$$flex(S') \leq flex(S).$$

Additionally, a metric exhibits ***strong simplicity*** *when $flex(S') = flex(S)$ if and only if the new event can be assigned any time in $\mathbb{R}$(i.e. has domain of ∞). Otherwise, we say it exhibits* ***weak simplicity***.

3.2.2 Density

Density captures the idea that as events are added to a sequence, then scheduling must be ever more precise to be successful, which implies a reduction in flexibility.

Definition 3.4. (*Lloyd et al.,* 2018) *For STNs $S_n \in \mathscr{S}$ composed of a set of n sequential events that occur within a fixed, finite interval $[a, b]$, a flexibility metric $flex$ is **dense** if it has the property*

$$\lim_{n\to\infty} flex(\mathscr{S}_n) = 0.$$

3.2.3 Sphericality

Sphericality captures the idea that the flexibility of an STN is limited by the least flexible event within the STN.

Definition 3.5. (*Lloyd et al.,* 2018) *A flexibility metric is **spherical** if and only if for two STNs with the same number of events and whose solution spaces are the same size (in other words, their polyhedra have the same dimension and volume), the STN with the larger inscribed sphere is considered to be more flexible.*

3.2.4 Containment

Containment captures the idea that if every valid schedule for STN S is valid for STN S', then S' is as or more flexible than S.

Definition 3.6. (*Lloyd et al.,* 2018) *Consider two STNs S and S' that have the same number of events and where the polyhedron of S is a proper subset of the polyhedron of S'. A flexibility metric captures **strong containment** if*

$$flex(S') > flex(S),$$

*and it captures **weak containment** if*

$$flex(S') \geq flex(S).$$

3.3 Flexibility Metrics

In this section, we describe four flexibility metrics which will each be evaluated with the system described in Chapter 5 used later to define different reward functions. Each of these metrics build off the previously listed metrics.

3.3.1 The Naïve Flexibility Metric

The naïve flexibility matric simply sums over the range of the feasible intervals of each event in the STN.

$$flex_N(S) = \sum_{i \in T} lst(i) - est(i) \tag{3.1}$$

The metric, however, fails to differentiate between some examples of concurrent and sequential STNs.

3.3.2 The Hunsberger Flexibility Metric

Hunsberger (2002) improved upon naïve flexibility by considering flexibility between pairs of events i, j using the distance matrix, which led to improvements using the metric with regards to concurrent and sequential classes of STNs.

$$flex_H(S) = flex_N(S) + \sum_{i \in T} \sum_{j \in \{k \in T | t_k > t_i\}} (D_S[i,j] + D_S[j,i]) \tag{3.2}$$

However, Wilson et al. (2014) shows that this improvement still fails to capture the dependencies between events.

3.3.3 The Wilson Flexibility Metric

Wilson et al. (2014) propose another metric that is used to temporally decouple events in the STN. By doing so, the metric maximizes the sum of the intervals in the decoupling.

$$flex_W(S) = \sum_{i \in T} t_i^+ - t_i^- \tag{3.3}$$

3.3.4 Sphere Flexibility

Lloyd et al. (2018) introduced sphere flexibility, a metric that satisfies every desideratum and is computationally efficient using a linear program.

Definition 3.7. (*Lloyd et al.,* 2018) *The* ***sphere flexibility*** *of an STN is the nth root of the volume of the inscribed sphere of its polyhedron.*

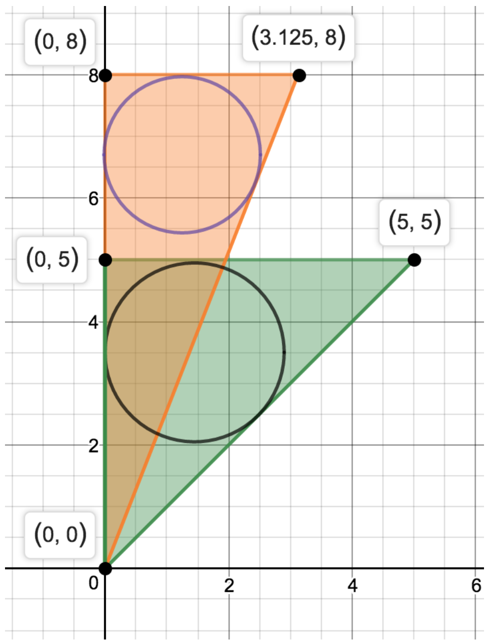


Figure 3.2 The orange triangle contains a smaller inscribed circle than does the green triangle (both representing feasible regions of arbitrary STNs), implying that it has a lower sphere flexibility

.

For example, in Figure 3.2, both triangles have the same area of 12.5 units squared, but the orange one is narrower. Thus, the inscribed circle is smaller (radius 1.26) in the orange region than in the green one (radius 1.45) and has a lower sphere flexibility.

Each of these metrics were analyzed using the desiderata developed in Lloyd et al. (2018). The results of their analysis are shown in Table 3.1

	Naïve	**Huns.**	**Wilson**	**Sphere**
Simplicity	×	×	×	✓*
Density	×	×	×	✓
Sphericality	×	×	×	✓
Containment	✓*	✓	✓*	✓*

Table 3.1 ✓denotes a desideratum that is satisfied by the given flexibility metric, while ×denotes a desideratum that is not satisfied. * indicates that the flexibility metric satisfies a weaker version of the desideratum, as given in Lloyd et al. (2018).

Chapter 4

Introduction to Reinforcement Learning

In this chapter, we introduce the problem of reinforcement learning. We use the classical reinforcement learning problem, the multi-armed bandit problem, to introduce the core concepts of the problem. In addition, we address some challenges and existing workarounds, particularly surrounding large or time-varying problems.

4.1 Motivating the Reinforcement Learning Problem

Consider a young child that loves to play with blocks. This child might know that there are many different types of blocks, and that they can be stacked and placed together in many different ways. If the child wants to make a tower, for example, then he might stack four blocks on top of one another. Alternatively, for a bus, he might place a long block on top of two cubes. Suppose that the child is able to construct k different types of objects with the blocks.

Suppose now that we want the child to construct certain objects. We might request the child to make these objects, but the child is devious and would likely make another object purely to annoy us. Instead, we might consider giving the child a piece of candy each time he makes an object, regardless of whether we want that particular object. Then, we might incentivize him by rewarding him more preferable candy when he makes objects that we want. If we continued this arrangement for a long enough time, the child would slowly learn which configuration to place the blocks to achieve his favorite candy.

It is easy to imagine a child, or even any other biological system, learning to exploit this system of incentives very quickly. Learning from interaction, in fact, underlies every theory of learning and intelligence (Sutton and Barto, 2018) . Reinforcement learning is a field that attempts to use the same system of incentives to teach computers how to identify the correct action to take to maximize some incentive signal. This general problem formulation, that of an agent learning by manipulating its surroundings to reach some goal, is the core of the problem of reinforcement learning.

4.1.1 A Classic Example of Reinforcement Learning: the Multi-Armed Bandit Problem

We can now use the example situation from Section 4.1 to introduce the classic reinforcement learning problem: the multi-armed bandit problem (Sutton and Barto, 2018). In the multi-armed bandit problem, our agent (i.e. the child) can select one of k levers (i.e. place blocks in one of k configurations). After the ith lever is pulled, the agent receives a reward (i.e. candy) indicating how "good" the choice of lever was. The agent's goal is to maximize the reward earned over a long (potentially infinite) number of rounds. However, the reward can be nondeterministic, meaning that the agent is unable to simply try each lever once and then select the best repeatedly. The agent must learn which lever offers the best expected reward. We will describe each of its components in more depth in Section 4.2.

4.2 Components of Reinforcement Learning Problems

The multi-armed bandit problem introduces a number of interesting questions that are relevant to any reinforcement learning problem. In this section, we begin by defining the components of a reinforcement learning problem and the key considerations that need to be made to maximize the success of an agent within this problem. We will see later that these components are contained within a feedback loop that describes how they interact in a reinforcement learning problem (Figure 4.1).

4.2.1 Environment, Agent, and Reward

Environment

Any reinforcement learning problem must be contained within some context, which we call the **environment**. For the child mentioned in Section 4.1, the environment could be defined as the configuration of the blocks. We note here that the agent can not sense nor affect everything in the world, and very little, if anything, beyond the the configuration of the blocks is truly necessary to understand the situation as we have described it. Similarly, for the multi-armed bandit problem, the environment can be described succinctly as the k levers, each of which dispenses some possibly random amount of reward.

The **state** of an environment is a description of the current condition of the environment. The environment must be malleable under action from the agent. An **action** is something that the agent can do to change the state of the environment. If the agent is unable to take actions that change the state of the environment, then the agent will be unable to learn how to manipulate the environment to its advantage. We can then define a **state-action pair** to be a pair consisting of a state and an action the agent could take in that state.

To describe the environment, we may build a model containing relevant information about it (i.e. configuration of blocks). For example, describing the environment of the multi-armed bandit problem may require us to specify a model of the random distribution from which the rewards for each lever are drawn. We might model the reward from the ith lever as being drawn from a normal distribution $\mathcal{N}(\sigma_i, \mu_i)$. This reward would then be returned by the environment to the agent upon each action being taken. We now have a simple description of an environment that enables us to describe the multi-armed bandit problem (Sutton and Barto, 2018).

However, we might wonder if σ_i and μ_i are functions of time, in which case we would call the environment of the multi-armed bandit problem nonstationary because the reward function changes over time. If neither the standard deviation nor the mean change over time, then we would instead call the multi-armed bandit problem stationary. This definition can be generalized to any problem (Sutton and Barto, 2018).

Definition 4.1. *An reinforcement learning environment is* ***stationary*** *if the function describing the reward signal does not change over time. Otherwise, it is called* ***nonstationary****.*

We note that it is generally more difficult to assign credit in a nonstationary reinforcement learning problem than it is for a stationary problem (Sutton and Barto, 2018).

Agent

From our description thus far, we see that the boundary between environment and agent is not necessarily clear. This boundary can be defined in many ways, and we encourage the reader to see Sutton and Barto (2018) for additional information. Consider the child manipulating the blocks. Such a child would not be able to perceive the entire known universe at every moment. It would then be unhelpful to define our environment as the world and everything in it. From this realization, we begin to understand that while the environment is the one being manipulated, the perception of an agent is key a limitation for how an agent may interact with the environment (Sutton and Barto, 2018).

The agent is the part of a reinforcement learning solution that learns to act correctly in different environments - we train it, by designing a reward function for the environment, to learn to reach the desired goal state. Upon interacting with the environment, the agent can collect its rewards and process these in some way to retain experience about how different actions went in different states. It can use this experience (usually as a metric) to identify the best course of action upon returning to the state.

One of the major problems within reinforcement learning is the **exploration-exploitation problem**. The problem can be summarized as follows. When an agent arrives at a state, it has two choices: to *exploit* its previous experience and make a greedy choice based on what its experience suggests is best, or to *explore* actions it has never taken as a means of gaining more experience and identifying whether there exist better alternatives to its currently preferred action. The problem remains unsolved (Sutton and Barto, 2018), but the **ϵ-greedy method** is one way that reinforcement learning agents can train to balance this tradeoff. With the ϵ-greedy method, we select an $0 \leq \epsilon \leq 1$ to be a probability with which the agent selects randomly from among its possible next actions. If $\epsilon = 0$, then the agent uses its initial idea of the best action for every state it reaches and never chooses anything else. At the other extreme, if $\epsilon = 1$, then it always chooses randomly from among the possible actions - it never uses its prior experience and always explores. There exist variations on the method, such as reducing ϵ with the number of actions the agent has taken or the number of times a state has been reached, but the method tends to work well for addressing the exploration-exploitation problem (Sutton and Barto, 2018).

Reward

The core of a reinforcement learning problem is the **reward signal**, which is a function that converts a state-action pair of an environment (described in Section 4.2.1) into a real number that can is returned to the agent (described in Section 4.2.1) acting in the environment. The reward is a measure of how "good" the state-action pair is as a waypoint towards the goal state of the environment.

In some problems, there is a very clear reward signal. For example, in the multi-armed bandit problem, our reward signal is obvious - it is defined to be the reward given after each pull of the lever. However, there exist more complicated problems, such as in a game like Tic Tac Toe, where there is no obvious reward scheme. One possible reward scheme could be +1 for a win, 0 for a tie, and -1 for a loss (Sutton and Barto, 2018). However, even in such a simple game, there exist other reward schemes (i.e. $+1, -0.1, -1$) which may be more conducive to reinforcement learning, or even a reward scheme which rewards different states at different amounts as judged by a human expert. *A good reward function, in the end, must maximize the information from which an agent can identify which interactions with the environment (i.e. actions) made progress towards the goal* (Sutton and Barto, 2018). Identically, we say that a good reward function has a high signal-to-noise ratio, which implies that states that lead to successful solutions consistently return higher rewards with less variability. We also note that reinforcement learning algorithms tend to overfit to the reward function (Sutton and Barto, 2018), which maps a state and an action from that state to a reward. This means that reinforcement learning can often behave unexpectedly (i.e. learn to take unexpected actions) in achieving a desired result because the reward function enables behavior unintended by the designer of the reward scheme.

This example also hints at a deeper challenge in more complicated problems. In the multi-armed bandit problem, every action in the space of possible actions is available in every round. However, in the Tic Tac Toe example mentioned previously, a square can only be selected once, so the number of possible actions decrease by one for each round of the game. We could easily imagine how, given a reward function, an early move that does not inherently signify progress towards the goal is given a low reward. The same action could, however, be crucial for a later move that results in high reward. The problem of identifying the impact of the early move on the success of the agent is called the **credit assignment problem**. We discuss one way to address it in Section 4.2.2. The credit assignment problem is one of the main difficulties in reinforcement learning (Sutton and Barto, 2018).

The Reinforcement Learning Feedback Loop

In summary, a reinforcement learning agent learns by observing its environment and taking an action to change its environment. After each action, the agent receives a new state (i.e. observation) of the environment as well as a reward signal that indicates how "good" the action that led to the new state was, with respect to the objective of reaching the goal state. The process is shown visually in Figure 4.1. In Sections 4.2.2 and 4.3, we describe the specifics of how the agent selects the best action, These components form a feedback loop that describes how they interact in a reinforcement learning problem (Figure 4.1).

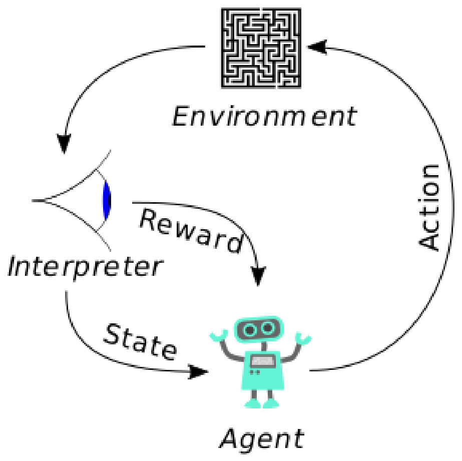


Figure 4.1 A feedback loop describing the order of events in a reinforcement learning system. The agent selects an action based on its observations of the environment. The action changes the environment in some way. Then, the agent receives some reward signal that indicates how good the new state is (i.e. is it a goal state or a failure state). Image taken from https://en.wikipedia.org/wiki/Reinforcement_learning#/media/File:Reinforcement_learning_diagram.svg

4.2.2 Value and Return

In this section, we address the credit assignment problem and introduce the idea of the value of a state.

Suppose that an agent makes a sequence of decisions that result in a large reward. Recall that the credit assignment problem is the question of identifying which decision enabled the agent to reach a state in which it received a large reward. We can begin to address this problem by considering the return, or the total reward over an episode of a problem. One simple way to formally define the return is as follows.

Definition 4.2. (*Sutton and Barto,* 2018) *The* ***return*** G_t *in the* t*th time step is given by the sum of all future rewards* (*where* R_t *is the reward given by action* A_{t-1} *from state* S_{t-1}),

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T,$$

where T *is the final time point in the episode.*

We note that this definition of return allows us to tackle the credit assignment problem by identifying the total reward we can expect as a result of this action. However, we may not want to consider only the long-term benefit in selecting an action. We may, at times, want to balance the long-term return and the next reward. In order to allow this choice, we can simply discount future rewards by generalizing our definition of return.

Definition 4.3. *The* ***discounted return*** (*Sutton and Barto,* 2018) G_t *in the* t*th time step is given by the sum of all future rewards*

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 T_{t+3} + \cdots \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k-1} \\ &= R_{t+1} + \gamma G_{t+1}, \end{aligned}$$

where T *is the final time point in the episode and* $0 \leq \gamma \leq 1$ *is a parameter that indicates how focused an agent should be on maximizing long term reward* (*if* $\gamma = 1$) *as opposed to immediate reward* (*if* $\gamma = 0$).

From now on, we use **return** to refer to the discounted return parameterized by γ. We might now consider using the return from a state s as the value $V(s)$ of that state. Recall that the value of a state is meant to be a metric with which larger values indicate more proximity to the goal state. Now that we have defined the return, we can formally define **value** as follows.

Definition 4.4. (*Sutton and Barto,* 2018) *The **value** $V(s)$ of state s in a reinforcement learning problem can be defined as the expected return given that the agent reaches state S_t at time t:*

$$V(s) = \mathbb{E}\left[G_t \mid S_t = s\right].$$

Using the return to define the value with which we wish to make our action decisions introduces a new problem for the agent. The agent doesn't know the return of a state without progressing through the remainder of the episode and then propagating the result backwards along the states that it visited. We can solve this problem optimally with a dynamic programming approach that relates values of adjacent time steps using the following Bellman's Equation (Sutton and Barto, 2018) for identifying a value function mapping states to real numbers. In order to define our value function, we must first define a policy.

Definition 4.5. *A **policy** $\pi : \mathcal{S} \to \mathcal{A}$ is a function that maps states to actions.*

A reinforcement learning agent is attempting to develop a policy that allows it to optimally solve the problem at hand. We might then consider that an optimal policy π^* for a reinforcement learning problem might rely on having an optimal value function V^* to make decisions with. To find this optimal value function, we can use a system of equations defined by the following Bellman's Equation (Sutton and Barto, 2018).

$$V^*(s) = \max_{a \in \mathcal{A}(s)} \left\{ R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^*(s') \right\}, \tag{4.1}$$

where R is the reward of taking action a from state s to reach state s'. Using dynamic programming techniques, we can directly solve for the optimal value function and use it to identify the optimal policy (Sutton and Barto, 2018). We note, however, that such an approach can be computationally intensive and prohibitively so. In Section 4.3, we explore some methods by which we can estimate the value function to approach an optimal policy through Monte Carlo methods.

4.3 Value Estimates & Tabular Monte Carlo Methods

We can now note that some state or actions spaces are too big to use a dynamic programming approach to solve for the optimal value function.

In this section, we introduce value and policy estimation algorithms that converge, given enough computation, to optimal value and policy estimates. We limit the computational requirements by using tabular Monte Carlo methods.

Monte Carlo methods, generally speaking, are a class of computational methods that use random sampling to find solutions to deterministic problems. With respect to reinforcement learning, Monte Carlo methods often rely on being able to generate many sample episodes quickly (Sutton and Barto, 2018). Through simulation, we can do so for many reinforcement learning problems. Once we generate each sample episode, we can simulate it with a policy π that maps each state s to an action a that maximizes the sum of the reward of the action and the value estimate of the resulting state (Sutton and Barto, 2018). As the value estimate becomes closer to the true value, our policy would constantly select the highest-value action to take.

$$\pi(s) = \max_a \{\text{reward}(s, a) + V(s')\},$$

where s' is the state the system arrives at after choosing action a in state s. Once each episode has completed, we need to update our value estimate for each state S_i that was visited in the simulation. One common method of keeping track of a value estimate while using Monte Carlo methods is through **sample averaging**. When sample averaging, we first select a learning rate α, which can also be defined as a function of the number of episodes of training. We initialize a table V, our value estimates, that map states to some initial value R_0, which could be either a constant or a function on the state space. After the episode, we can calculate the return, and use it to update the value estimates for the next policy with the return from the episode using the following update rule (Sutton and Barto, 2018),

$$V_{\pi'}(S_i) = V_\pi(S_i) + \alpha\left(G_{i+1} - V_\pi(S_{i+1})\right).$$

We call this step **policy evaluation**. We can then update the policy π using the new value estimates (Sutton and Barto, 2018). We call this new policy π'.

$$\pi'(s) = \max_a \{\text{reward}(s, a) + V_{\pi'}\}$$

We call this step **policy improvement**. By using this algorithm, we constantly update our value estimates to be closer, by the law of large numbers (Sutton and Barto, 2018), to the true value. We call this algorithm **iterative policy evaluation**, and it is a common way to iterate simultaneously towards an optimal policy and an optimal value function estimate. We can prove that the algorithm converges, given enough computation, to the optimal values (Sutton and Barto, 2018).

We note that a Monte Carlo approach using randomly generated samples may not explore every possible state, which may not result in the best possible policy. The agent can not exploit what it hasn't explored. Given that many algorithms require initial value estimates, we might consider whether we could alter these estimates to initially bias for exploration, which might accelerate learning. Using optimistic initial estimates biases an agent to explore early in training (Sutton and Barto, 2018). For intuition, we might imagine setting the initial value estimate to be equal for each state, and significantly larger than is realistic. Then, upon simulation of a state, our value for that state will be significantly worse than we suggested our agent should expect, which would result in a large drop in value estimate during the policy evaluation step of the iterative policy evaluation algorithm. Initially, the policy improvement step would select unexplored actions for each state. However, once each state has been explored a few times, then we should see states with high true values losing less of their value estimate than states with low true values. Over time, this will result in a well-explored state space and well-estimated value function and policy.

4.4 Applying Reinforcement Learning to Simple Temporal Networks

In Chapter 5, we use the tools that have been described in this chapter to design and train a reinforcement learning agent that is able to learn an estimate of the value of various states. We can then use this estimate with a policy that maximizes the value at each step of execution to schedule simple temporal networks under uncertainty by learning from the flexibility of the STNUs being scheduled.

In this section, we provide a brief overview of the reinforcement learning agent that will evaluate the flexibility metrics. The core of this idea is designing reward functions as a function of the flexibility metric under test. The exact reward function is shown in Chapter 5.

The environment consists of the STNU that is being executed, as well as the current time, and lists of enabled and completed events. The agent is attempting to build a value estimate of the states in this system that enables it to maximize success in scheduling for the STNU on which the agent is being trained. It does so by taking actions according to its policy, which is based on the maximal-value action available to it. Upon success or failure of each episode, the values of each state along the way will be updated using the SARSA update rule, as described in (Sutton and Barto, 2018). At the end, the agent will be able to use its value estimate to choose which action to take at each state to solve the STNU.

Implementation details for all of these components of the reinforcement learning problems will be given in Chapter 5.

Chapter 5

Empirical Evaluation of Flexibility Metrics

5.1 Motivation

In Chapter 3, we introduced four flexibility metrics: naïve, Hunsberger, Wilson, and spherical flexibility. These metrics were developed sequentially in the literature (Hunsberger, 2002; Wilson et al., 2014; Lloyd et al., 2018). Hunsberger defined naïve flexibility and found inconsistencies in it, as shown in Chapter 3. Instead, he designed his Hunsberger metric, which solved one of these inconsistencies. Wilson flexibility was a response to flaws with the Hunsberger flexibility metric, which still led to some inconsistencies in edge cases, as shown in Lloyd et al. (2018). Sphericality was then developed out of an attempt to rigorously describe properties of flexibility metrics (Hunsberger, 2002; Wilson et al., 2014; Lloyd et al., 2018) and use a polytope to consider the space of schedules. Some of these flexibility metrics were then evaluated through their use in heuristic algorithms (Hunsberger, 2002) in which algorithms selected the next action to maximize each type of flexibiltiy. However, we are unaware of any attempts to directly evaluate each flexibility metric with reinforcement learning. Doing so may enable a more direct comparison between these metrics as to which one is best suited to describe the flexibility of a simple temporal network. In this chapter, we describe an attempt at a direct empirical evaluation of these flexibility metrics.

Prior Work

Lloyd et al. (2018) attempted to evaluate spherical flexibility by running a linear regression to identify the correlation between spherical flexibility and each of the other metrics. They found that spherical flexibility was poorly correlated ($r^2 \leq 0.471$ for all of the metrics). Lloyd et al. (2018) also confirmed, empirically, which metrics satisfied specific desiderata as shown in Figure **??**.

5.2 Reinforcement Learning for Scheduling in STNUs

Recall from Chapter 4 that a good reward function should have a high signal-to-noise ratio; they should quantify, with minimal variance which states lead to the desired result Sutton and Barto (2018). Good reward functions can enable faster training and more accurate results in reinforcement learning agents. To the author's knowledge, there has been no work in using reinforcement learning as a means to gauge the relative signal-to-noise ratios of flexibility metrics for STNs and STNUs nor to use reinforcement learning to identify schedules to satisfy STNUs. In this section, we describe a reinforcement learning agent that learns how to schedule events given a simple temporal network under uncertainty (STNU).

5.2.1 Hypothesizing the Results of the Reinforcement Learning Solution

We expect that the more recent flexibility metrics, and in particular, sphericality, should have the highest signal-to-noise ratio of the metrics under test. It should enable the agents to train the fastest and most successfully if it describes the benefit of a state-action pair in the problem, for getting to a goal state. We note that sphericality satisifies at least a weak form of every desideratum proposed by Lloyd et al. (2018), whereas the naïve, Hunsberger, and Wilson flexibilities all fail at least one of the desiderata (Lloyd et al., 2018).

5.2.2 Structuring the Reinforcement Learning System

In this section, we describe the structure of the reinforcement learning problem - its environment, agent, state space, action space, and reward - that is being trained and evaluated.

Environment

The environment of our reinforcement learning systems consists of the STNU with which events must be scheduled. We model the environment as a simple temporal network under uncertainty that is being executed. We use a timeline-based execution of the STN similar to that which was developed in Knittel et al. (2018). One time step in the environment is shown in Algorithm 1. We assume that each uncertain edge is accurately described by a uniform distribution parameterized by the constraint interval for the edge.

```
Data: Current time $t$
Data: Execution Time Resolution $\Delta t$
Data: Internal STNU state $S$
Data: A valid action to take $a$
Result: An updated STNU state $T$
Result: A float reward $R$
Result: A boolean indicating termination isDone
$pq \leftarrow$ priority queue prioritized by time of completion;
if $a$ is an enabled event then
    Execute $a$ in $S$;
    if $e'$ is an event contingent on $a$ then
        Realize a duration for the edge between them ;
        Add $e'$ to $pq$ with the realized completion time ;
    end
    if there is a constraint between a and event $e'$ then
        Identify when the constraint would fail ;
        Add $e'$ to $pq$ with the time for the requirement to fail ;
    end
end
$t \leftarrow t + \Delta t$;
Update all absolute constraints to begin at time $t$ or later ;
while next event to be completed has time $t$ do
    $e \leftarrow$ next event in priority queue;
    if $e$ is a contingent edge then
        Execute contingent event indicated by $e$;
        Add adjacent contingent and requirement edges to $pq$;
    else
        If events not complete, a constraint has failed;
    end
end
Update list of enabled edges;
Episode is done if all events are executed or a constraint has failed ;
if episode not done then
    $S \leftarrow S \setminus \{$all vertices adjacent to only executed events$\}$;
    minimize $S$ ;
    check whether $S$ is consistent ;
end
Episode is done if all events are executed or a the STN is inconsistent ;
Reward $\leftarrow$ number of newly completed events $\cdot flex(S)$;
if episode has failed to schedule the STNU then
    $R \leftarrow R - n \cdot flex($original STNU$)$
end
```

Algorithm 1: A description of one step in the execution of an STNU with the action chosen by the reinforcement learning agent.

Agent

As described in Section 4.4, the agent is attempting to build a value estimate of the states in this system that enables it to maximize success in scheduling for the STNU on which the agent is being trained. It does so by taking actions according to its policy, which is based on the maximal-value action available to it. The agent is described more be Algorithm 2.

We use $\epsilon-$greed to deal with the exploration-exploitation problem. This method works by first selecting a parameter $0 \leq \epsilon \leq 1$. When the agent is selecting an action, it chooses an action by exploiting the current policy with probability $1 - \epsilon$. Otherwise, it randomly selects among all actions to explore what could happen. If $\epsilon = 0$, then the agent only exploits what it already knows. Similarly, if $\epsilon = 1$, then the agent only explores (randomly taking actions). We choose $\epsilon = 0.1$ for this agent.

```
Data: An STNU, $S$
Result: A reinforcement learning agent with an updated value
        function $Q'$
Initialize a list of visited states;
while the current episode of the STNU is incomplete do
    get all possible actions;
    select which action to take using $\epsilon$-greed;
    call Algorithm 1 with an action to get a reward and new state;
end
check whether the episode succeeded or failed;
for each state visited along the way, use the return to update value
estimate $V$;
```

Algorithm 2: A description of using Monte Carlo methods for training a reinforcement learning agent to schedule simple temporal networks with uncertainty.

We note that not every state will need to be visited. For example, a state depending on many previous states before it can be executed is unlikely to be executed at $t = 0$, so Monte Carlo methods help us narrow down the set of states we might actually need to have value estimates for.

Reward

The third part of any reinforcement learning problem is the reward function. Designing a reward function for any problem is particularly tricky, but crucial for a reinforcement learning agent to train well. Reinforcement learning algorithms tend to overfit to their reward functions, which can lead to reward functions resulting in unexpected behavior from an agent (Mnih et al., 2016). Some of these behaviors may work well and maximize the reward, but behave in surprising ways unanticipated and undesired to the designers of the system.

For this problem, we describe the reward function as follows, where s is a state that contains an STNU S in the environment, a is an action taken by an agent, $flex$ is the flexibility metric under test, n is the number of events completed in the step, and N is the total number of events in the original STNU S_0.

$$R(s,a) = \begin{cases} n \cdot flex(S) & \text{if step does not end in episode failure} \\ n \cdot flex(S) - N \cdot flex(S_0) & \text{else} \end{cases}$$

This function returns a high value when many events end on the same time step without the episode failing. If the episode fails, it penalizes by a large amount when the episode fails to complete successfully.

One example of a possible unanticipated issue in our reinforcement learning agent would occur if the agent ends up in a local maximum in which it gets more reward than any other route it has explored, but is never able to successfully schedule for an STNU.

5.3 Running Through Two Examples

In this section, we run through two example episodes, one failure and one successful run to show how the value function changes over each episode. Recall that the learning rate of a reinforcement learning algorithm is used to identify how much the vaue estimate should change given a new data point from the return of a single run. Let $\alpha = 0.1$ be the learning rate of our agent. Let the flexibility metric f in the following examples be defined by the number of unexecuted states in the given STN (this metric is not helpful for the actual problem, but is used as a simple flexibility function for the following examples).

We also note that the time step resolution $r = 1$ minute.

Let $S_t = (E, C)$ be a state, where t is the time, E is the set of enabled events and C is the set of executed events.

We note that our training run of the example STNU begins, after the zero event is executed, in the state

$$S_0 = (\{1, 3\}, \{0\}).$$

In this example (which is modified from Lund et al. (2017)), we have two robots which must reach a location within some specified time (2 minutes, in this case) of one another. The times for each robot to travel from its initial point to the final location is described by a uniform distribution as shown in Figure 5.1.

5.3.1 A Failure Episode Example

Let Q_i be our state-action value estimate function after the ith training run, where the value estimates are initially the total amount of reward we can get from the STN.

$$Q_0(s, a) = f(S_0) = 4 \qquad (\forall(s, a))$$

Given that the value estimates of every state are equal, then our agent chooses randomly from the set of actions $\{\emptyset, 1, 3\}$. In this example, the agent initially selects action 3 (see Figure 5.1).

The new state $S_1 = (\{1\}, \{0, 3\})$. Since event 3 has been executed in this time step, the reward $R_1 = 3$. Once again, $\max_{a \in \mathcal{A}(S_1)} Q_0(S_1', a)$ does not distinguish between the remaining actions, so the agent randomly selects between starting event 1 and doing nothing. In this example, the agent chooses nothing (see Figure 5.2).

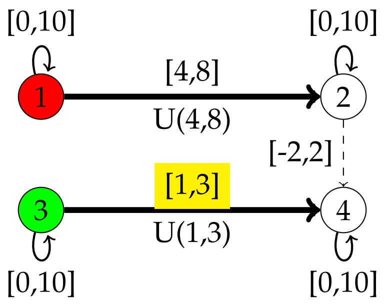


Figure 5.1 At $t = 0$, the state of the agent $S_0 = (\{1, 3\}, \{0\})$. The agent selects action $A_0 = 3$ from the action space $\mathcal{A}(S_0) = \{\emptyset, 1, 3\}$.

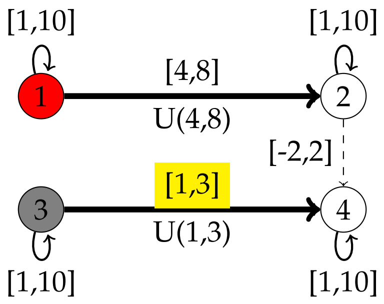


Figure 5.2 At $t = 1$, the agent gets a reward $R_1 = 0$ since no events were executed at time 0. The state of the agent $S_1 = (\{1\}, \{0, 3\})$. The agent then chooses action $A_1 = \emptyset$ from the action space $\mathcal{A}(S_1) = \{\emptyset, 1\}$.

At $t = 2$, the agent receives information that contingent event 4 has been executed. The new state $S_2 = (\{1\}, \{0, 3, 4\})$, so the agent has received the reward $R_2 = 2$ from the previous time step. $\max_{a \in \mathcal{A}(S_2)} Q_0(S_2', a)$ does not distinguish between the remaining actions, so the agent randomly selects between starting event 1 and doing nothing. In this example, the agent chooses to do nothing (see Figure 5.3). We also note that the constraint that requires events 2 and 4 to occur within 2 minutes of one another is active (which is not explicitly represented within the state).

At $t = 3$, the agent finds that no events have terminated, so the agent receives the reward $R_3 = 0$. $\max_{a \in \mathcal{A}(S_3)} Q_0(S_3, a)$ does not distinguish between the remaining actions, so the agent randomly selects between starting event 1 and doing nothing. In this example, the agent chooses to do nothing (see Figure 5.4).

At $t = 4$, the state $S_4 = (\{1\}, \{0, 3, 4\})$. The agent finds that the constraint

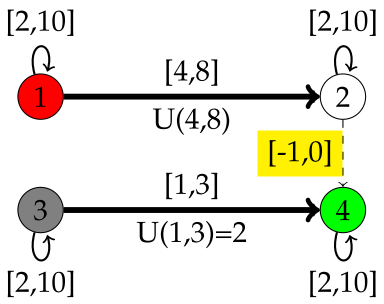


Figure 5.3 At $t = 2$, the agent gets a reward $R_2 = 2$ since contingent event 4 was executed in the previous time step. The state of the agent $S_2 = (\{1\}, \{0, 3, 4\})$. The agent then chooses action $A_2 = \emptyset$ from the action space $\mathcal{A}(S_2) = \{\emptyset, 1\}$.

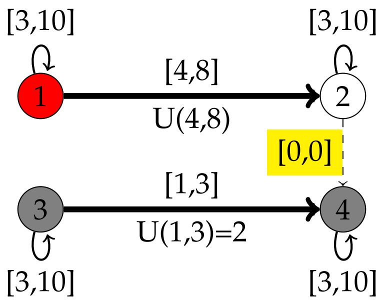


Figure 5.4 At $t = 3$, the agent gets reward $R_3 = 0$ since no events were completed in $t = 2$. The state of the agent $S_3 = (\{1\}, \{0, 3, 4\})$. The agent then chooses action $A_3 = \emptyset$ from the action space $\mathcal{A}(S_3) = \{\emptyset, 1\}$.

between events 2 and 4 failed in the previous time step. Since it is no longer possible to satisfy the constraint between events 2 and 4 and therefore (see Figure 5.5) the agent has identified that the STNU has failed to schedule our problem, the agent receives a large negative reward $R_4 = -nf(S_0) = -16$, where n is the number of events in the original STN. This reward guarantees the return is negative.

Now that we have simulated this episode, we can update the value estimate Q_1. The agent visited the states in the following order, took certain actions, and received the rewards listed afterward.

0. $S_0 : (\{1, 3\}, \{0\}), A_0 = 3 , R_1 = 3$
1. $S_1 : (\{1\}, \{0, 3\}), A_1 = \emptyset, R_2 = 0$

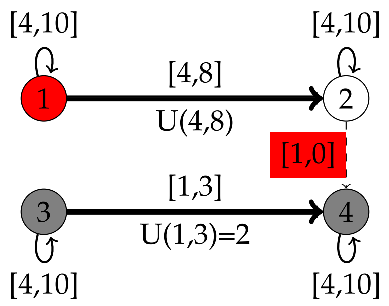


Figure 5.5 At $t = 4$, the execution fails as a constraint is violated, so the reward is $R_4 = -16$.

2. $S_2 : (\{1\}, \{0, 3, 4\})$, $A_2 = \emptyset$, $R_3 = 2$
3. $S_3 : (\{1\}, \{0, 3, 4\})$, $A_3 = \emptyset$, $R_4 = -16$

Since the episode has ended, we can update the value estimates. They are updated according to the following update rule.

$$Q_1(S_t, A_t) = Q_0(S_t, A_t) + \alpha \left[R_{t+1} + Q_0(S_{t+1}, A_{t+1}) - Q_0(S_t, A_t)\right].$$

So, the following state-action pairs from this episode are updated as follows.

$$\begin{aligned}
Q_1(S_3, \emptyset) &= 4 + 0.1 \cdot (-16 + 4 - 4) = 2.6 \\
Q_1(S_2, \emptyset) &= 4 + 0.1 \cdot (2 + 4 - 4) = 4.2 \\
Q_1(S_1, \emptyset) &= 4 + 0.1 \cdot (0 + 4 - 4) = 4.0 \\
Q_1(S_0, 3) &= 4 + 0.1 \cdot (3 + 4 - 4) = 4.3
\end{aligned}$$

We can see that the state-action pairs in which we earned rewards were rated as more valuable, but those in which we lost reward lost value.

5.3.2 A Successful Episode Example

In the following figures (5.6, 5.7, 5.8, 5.9, 5.10, 5.11), we show a successfuly example and the value estimate update from that example.

Now that we have simulated this episode, we can update the value estimate Q_1. The agent visited the states in the following order, took certain actions, and received the rewards listed afterward.

0. $S_0 : (\{1, 3\}, \{0\})$, $A_0 = 1$,$R_1 = 3$

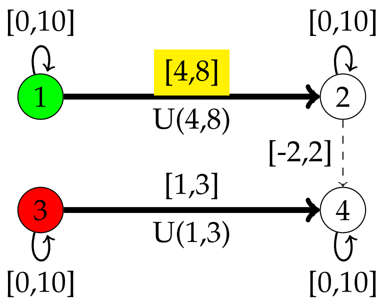


Figure 5.6 At $t = 0$, the state of the agent $S_0 = (\{1, 3\}, \{0\})$. The agent then chooses action $A_0 = 1$ from the action space $\mathcal{A}(S_0) = \{\emptyset, 1, 3\}$.

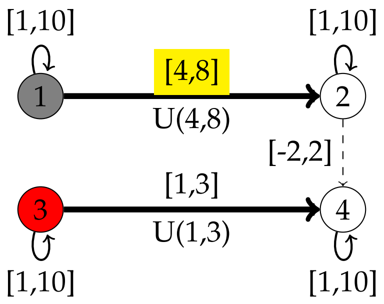


Figure 5.7 At $t = 1$, the agent gets a reward $R_1 = 3$ since event 1 was executed at time 0. The state of the agent $S_1 = (\{3\}, \{0, 1\})$. The agent then chooses action $A_1 = \emptyset$ from the action space $\mathcal{A}(S_1) = \{\emptyset, 3\}$.

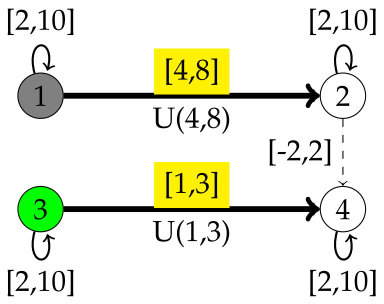


Figure 5.8 At $t = 2$, the agent gets a reward $R_2 = 0$ since task 3 has been executed. The state of the agent $S_2 = (\{\}, \{0, 1, 3\}$. The agent then chooses action $A_2 = \emptyset$ from the action space $\mathcal{A}(S_2) = \{\emptyset\}$.

1. $S_1 : (\{3\}, \{0, 1\}), A_1 = \emptyset, R_2 = 0$

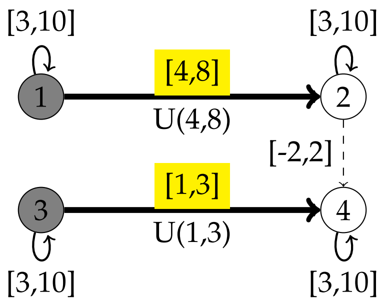


Figure 5.9 At $t = 3$, the agent gets a reward $R_3 = 2$ since task 1 has been executed. The state of the agent $S_3 = (\{\}, \{0, 1, 3\}$. The agent then chooses action $A_3 = \emptyset$ from the action space $\mathcal{A}(S_3) = \{\emptyset\}$.

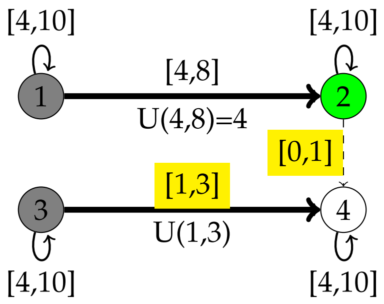


Figure 5.10 At $t = 4$, the agent gets a reward $R_4 = 1$ since contingent event 2 has concluded in time step 3. The state of the agent $S_4 = (\{\}, \{0, 1, 2, 3\}$. The agent then chooses action $A_4 = \emptyset$ from the action space $\mathcal{A}(S_4) = \{\emptyset\}$.

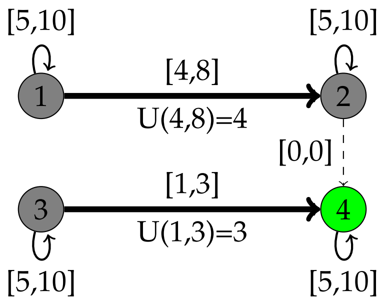


Figure 5.11 At $t = 5$, the agent gets a reward $R_5 = 0$ since contingent event 4 has concluded in time step 4. The state of the agent $S_4 = (\{\}, \{0, 1, 2, 3, 4\}$. The episode then concludes since the schedule has been successfully executed.

2. $S_2 : (\{3\}, \{0, 1, \}), A_2 = 3, R_3 = 2$
3. $S_3 : (\{\}, \{0, 1, 3\}), A_3 = \emptyset, R_4 = 0$
4. $S_4 : (\{\}, \{0, 1, 2, 3\}), A_4 = \emptyset, R_5 = 1$

Since the episode has ended, we can update the value estimates. They are updated according to the following update rule.

$$Q_1(S_t, A_t) = Q_0(S_t, A_t) + \alpha \left[R_{t+1} + Q_0(S_{t+1}, A_{t+1}) - Q_0(S_t, A_t)\right].$$

So, the following state-action pairs from this episode are updated as follows.

$$\begin{aligned}
Q_1(S_4, \emptyset) &= 4 + 0.1 \cdot (1 + 4 - 4) = 4.1 \\
Q_1(S_3, \emptyset) &= 4 + 0.1 \cdot (0 + 4 - 4) = 4.0 \\
Q_1(S_2, 3) &= 4 + 0.1 \cdot (2 + 4 - 4) = 4.2 \\
Q_1(S_1, \emptyset) &= 4 + 0.1 \cdot (0 + 4 - 4) = 4.0 \\
Q_1(S_0, 1) &= 4 + 0.1 \cdot (3 + 4 - 4) = 4.3
\end{aligned}$$

We can see that the state-action pairs in which we earned rewards were rated as more valuable, but those in which we lost reward lost value. We note that the values here are not much higher than the new values for the failed examples. Part of this is due to this example running a single iteration of training. Much more would be necessary to converge to a useful value estimate.

5.4 Results

The proposed reward scheme, in initial testing did not seem to be learning a successful policy. However, more testing is required before any conclusions can be drawn, as well as potential alterations of the reward function. One possibility for the reward function is that it may be too sparse, meaning that the agent doesn't get enough reward to be able to learn a useful value estime. This may happen because it gets no more than n pieces of reward, often less. Shaping the reward signal to provide more consistent reward may help with this problem.

The data for these tests was taken from the dataset generated by Lund et al. (2017), but each contingent probabilistic edge was modified to include a one-sigma interval around the mean of the normal distribution.

5.5 Conclusions

This thesis can not make significant conclusions for lack of data to analyze the performance of the proposed reinforcement learning agent. The next step for this thesis is to train these agents on STNUs.

In order to proceed with the thesis, the author suggests the following steps.

- Collect more data - Collecting more data will allow for a better diagnosis of the issues with the current system.
- Tweak the reward signal - One of the problems with reinforcement learning is reward signal design. It is unclear whether the signal is too sparse to be useful at the moment.
- Extend system to PSTNs - PSTNs have normally distributed contingent edges, and do not use flexibility. It may be interesting to try to learn to solve PSTN scheduling as a means to identify metrics on PSTNs.

Add Sutton and Barto to list

Bibliography

Boerkoel, James C., and Edmund H. Durfee. 2013. Distributed reasoning for Multiagent Simple Temporal Problems. *Journal of Artificial Intelligence Research* 47:95–156. doi:10.1613/jair.3840.

Boerkoel, James C, Leon R Planken, Ronald J Wilcox, and Julie A Shah. 2012. Distributed Algorithms for Incrementally Maintaining Multiagent Simple Temporal Networks. *Proceedings of the Autonomous Robots and Multirobot Systems workshop (at AAMAS-12)* 59(June 2013):256–263.

Brooks, Jeb, Emilia Reed, Alexander Gruver, and James C. Boerkoel. 2015. Robustness in probabilistic temporal planning. In *Proc. of AAAI-15*, 3239–3246.

Dechter, Rina, Itay Meiri, and Judea Pearl. 1991. Temporal constraint networks. *Artificial intelligence* 49(1-3):61–95.

Hunsberger, Luke. 2002. Algorithms for a temporal decoupling problem in multi-agent planning. In *Proc. of AAAI-02*, 468–475.

Knittel, Marina, Liam Lloyd, Grace Diehl, Judy Lin, David Chu, Jeremy Frank, and James C Boerkoel Jr. 2018. Trade-offs Between Communication and Success Rate in Uncertain Multi-Agent Schedules.

Lloyd, Liam, Amy Huang, Mohamed Omar, and James C Boerkoel Jr. 2018. New Perspectives on Flexibility in Simple Temporal Planning. Submitted to ICAPS-18.

Lund, Kyle, Sam Dietrich, Scott Chow, and James C Boerkoel Jr. 2017. Robust Execution of Probabilistic Temporal Plans. *Proceedings of the 31th Conference on Artificial Intelligence (AAAI 2017)* 3597–3604. URL https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14641.

Mnih, V., A. Puigdomènech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *ArXiv e-prints* 1602.01783.

Mnih, Volodymyr, David Silver, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *Neural Information Processing Systems* 1–9. doi:10.1038/nature14236. 1312.5602.

Morris, P., N. Muscettola, and T. Vidal. 2001. Dynamic control of plans with temporal uncertainty. In *Proc. of IJCAI-01*, 494–502.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: MIT Press.

Tsamardinos, Ioannis. 2002. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Methods and Applications of Artificial Intelligence*, 97–108. Springer.

Vidal, T., and M. Ghallab. 1996. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. ECAI-96*, 48–54.

Wilson, Michel, Tomas Klos, Cees Witteveen, and Bob Huisman. 2014. Flexibility and decoupling in Simple Temporal Networks. *Artificial Intelligence* 214:26–44. doi:10.1016/j.artint.2014.05.003.