TUHH
Hamburg University of Technology

Master Thesis

Salvador Fernandez Covarrubias

Incremental Measurement of Model Similarities in Probabilistic Timed Automata Learning

June 24, 2019

supervised by:
Prof. Dr. Sibylle Schupp
Prof. Dr.-Ing. Görschwin Fey
M.Sc. Sascha Lehmann

Hamburg University of Technology (TUHH)
Technische Universität Hamburg
Institute for Software Systems
21073 Hamburg

STS
Software
Technology
Systems

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 24. Juni 2019

Salvador Fernandez Covarrubias

Contents

List of Figures

List of Algorithms

Abstract - Automatic generation of models may lead to the key of understanding and verifying real life systems. Automata learning is a very powerful approach for this manner, that consists of inferring models from observations. However, systems are becoming more complex and consequently difficult to learn and infer in practice. In this paper, we introduce a method of learning and measuring observations of probabilistic timed automata incrementally, by assigning and evaluating costs of the similarities of observations, to model behaviors of non-deterministic systems. In the experiments conducted, we demonstrate how the level of abstraction of an inferred model can drastically vary depending on the different cost parameters applied, and also how this may impact the learning process itself. The utilization of cost functions in automata learning is used as a technique to reduce the complexity of learning systems, and also proposed for other potential areas of application like automata minimization.

1 Introduction

Real-life systems such as embedded controllers and communication protocols are becoming more complex and difficult to understand and verify, mainly due to *non-deterministic* behaviors and large number of components that may interact with each other along processes. Automata learning has been used as a technique to tackle this problem, by inferring models from observations of a system. There exists two principle approaches called *passive and active*. Passive learning mainly consists of analyzing observations without interacting with the system, while active learning interacts with a system to request additional information of observations, if needed [18]. In this project, we decided to apply a passive learning approach to learn non-deterministic systems, as allowing interaction for active learning might become very difficult and sometimes unfeasible in practice, due to lack of resources or documentation of the system to observe. Nevertheless, we do interact with a model that represents the observed system to be learned, with the sole purpose of evaluating the progress of our learning approach. Our idea consists of learning and modelling observations of a system *on-the-fly*, by measuring and evaluating incoming data incrementally. For this, we utilize *Euclidean distances*, *cost functions* and *graph matching* techniques to indicate the progress of learning and to define the structure of our *learned model*.

We begin this paper by reviewing the conceptual foundations of *automata theory*, the software tool *Uppaal*, that was used in this project to observe systems and *graph theory* in Chapter 2. Afterwards, we discuss in Chapter 3 the main topics that inspired this paper and that are also on-going related research topics like *active automata learning*, *automata optimization* and *graph matching*. The main concepts, algorithms and ideas of the proposed *incremental learning* approach are developed an explained in Chapter 4. An overview of the implemented tool and its features is given in Chapter 5. We then show a series of experiments in Chapter 6, which demonstrate the different levels of abstraction that can be obtained by learning automata incrementally with the use of different parameters in our applied cost functions and Euclidean distances. And finally, in Chapter 7, we discuss the feasibility of the incremental learning approach, how it can be extended and applied to other relevant areas of automata learning like *automata minimization*, and we also mention some of the limitations and possible extensions of the approach.

2 Background

2.1 Automata Theory

In this paper, we strictly observe probabilistic timed automata and refer to them as *observed systems*. Therefore, we begin this section by first explaining how time is handled in timed automata, followed by the formal definition of a timed automaton, and at the end we extend the definition to describe a probabilistic timed automaton.

2.1.1 Time

Time in timed automata is managed by units of *clocks*, along with *constraints* and *interpretations* that represent the duration of a simulation of an automaton from the beginning until the end. We review the formal definition of the previous terms and also the definition of *zones* in timed automata, as the terms will be used in the next sections.

Definition 2.1.1. (Clock constraints and clock interpretations) [1] For a set X of clocks, the set $\Phi(X)$ of *clock constraints* φ is defined by the grammar

$$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi \wedge \varphi, \tag{2.1}$$

where x is a clock in X and c is a constant. A *clock interpretation* v for a set X of clocks assigns a real value to each clock; that is, a mapping from X to the set $\mathbb{R}$ of nonnegative real numbers. For $\delta \in \mathbb{R}, v + \delta$ denotes the clock interpretation which maps every clock x to the value $v(x) + \delta$. For $Y \subseteq X, v[Y := 0]$ denotes the clock interpretation for X which assigns 0 to each $x \in Y$, and agrees with v over the rest of the clocks.

Definition 2.1.2. (Zones) [15] Let $Zones(X)$ be the set of zones over X, which are conjunctions of atomic constraints of the form $x \sim c$ for $x \in X, \sim \in \{\leq, =, \geq\}$, and $c \in \mathbb{N}$. The clock valuation v satisfies the zone ζ, written $v \models \zeta$, if and only if ζ resolves true after substituting each clock $x \in X$ with the corresponding clock value from v.

2.1.2 Timed Automata

Definition 2.1.3. (Timed automaton) [1] A timed automaton A is a tuple $< L, L^0, \Sigma, X, I, E >$ where,

- L is a finite set of locations.
- L^0 is the initial location, $L^0 \in L$.
- Σ is a finite set of labels.
- X is a finite set of clocks.
- I is a mapping that labels each location with some clock constraint in $\Phi(X)$.

– $E \subseteq L \times \Sigma \times 2^X \times \Phi(x) \times L$ is a set of switches. A switch $\langle s, a, \varphi, \lambda, s' \rangle$ represents and edge from a location s to location s' on symbol a. φ is a clock constraint over X that specifies when the switch is enabled, and the set $\lambda \subseteq X$ gives the clocks to be reset with this switch.

The semantic of a timed automaton is defined by associating transitions to it. A state is a par (s, v), where s is a location and v a clock interpretation that satisfies the invariant $I(s)$. A state is initial, if s is an initial location and $v(x) = 0$ for all clocks x. We focus on the transitions of locations that allow time to elapse, and the ones that allow a location to switch among other possible locations.

Definition 2.1.4. (Elapse of time) for a state (s, v) and a real-values time increment $\delta \geq 0$, $(s, v \xrightarrow{\delta} (s, v + \delta)$ if for all $0 \leq \delta' \leq \delta, v + \delta'$ satisfies the invariant $I(s)$.

Definition 2.1.5. (Location switch) for a state (s, v) and a switch $\langle s, a, \varphi, \lambda, s' \rangle$ such that v satisfies $\varphi, (s, v) \xrightarrow{a} (s', v[\lambda := 0])$.

2.1.3 Probabilistic Timed Automata

Definition 2.1.6. (Probabilistic timed automaton) [15] A probabilistic timed automaton can be defined as an extended tuple of a timed automaton $< L, L^0, \Sigma, X, I, E, prob >$ where,

$$prob \subseteq L \times Zones(x) \times \Sigma \times Dist(2^X \times L) \text{ is the probabilistic edge relation.}$$

Let $\mathbb{T} \in \{\mathbb{R}, \mathbb{N}\}$ be the time domain of the non-negative real or natural numbers, and a point $v \in \mathbb{T}^{|X|}$ referred as clock valuation. A state of a probabilistic timed automaton is a pair (l, v) where $l \in L$ and $v \in \mathbb{T}^{|X|}$ are such that $v \models I(l)$. In any state (l, v), there is a nondeterministic choice of either (1) making a discrete transition or (2) letting time pass. For case (1), a discrete transition can be made if there exists a distribution $p \in prob(l)$ which is enabled; that is, that a zone g is satisfied by the current clock valuation v. The probability of moving to a location l' and resetting all clocks in X to 0 is given by $p(l', X)$. In case (2), the option is only available if the invariant condition $I(l)$ is satisfied while time elapses.

2.2 Uppaal

UPPAAL is a tool for modeling, simulation and verification of real-time systems. It was designed mainly to check invariant and reachability properties by exploring the state-space of a system, i.e., reachability analysis in terms of symbolic states represented by constraints. [16] We use UPPAAL only to model and perform simulations of probabilistic timed automata, as model-checking techniques are not applied in this project.

2.2.1 Uppaal Models

Models in UPPAAL are based on a stochastic interpretation and extension of the timed automata (TA) formalism. For individual TA components, the stochastic interpretation replaces the non-deterministic choices between multiple enabled transitions by probabilistic choices. Similarly, the non-deterministic choices of time-delays are defined by probability distributions. [9]

The edges of an automaton are normally composed of [16]:

- *Guards*, expressing a condition on the values of clocks and integer variables that must be satisfied for an edge to be taken.
- Synchronization *actions* which are triggered when an edge is taken.
- *Assignments* to variables and or clock resets.

Whereas the locations of an automaton are normally composed of:

- *labels*, used to name and or identify locations uniquely.
- *Invariants*, which are conditions expressing constraints on the clock values that allows a model to let time elapse, by remaining in a specific location.

2.2.2 Uppaal Simulations

UPPAAL provides a query language that allows to visualize the values of expressions along simulated runs, which gives insight on the behavior of the system. The syntax of the queries is as follows: [9]

$$simulate\ N\ [<= bound]\ \{E1, \ldots, Ek\} \tag{2.2}$$

where N is a natural number indicating the number of simulations to be performed, *bound* is the time bound on the simulations, and $\{E1, \ldots, Ek\}$ are the k (state-based) expressions that are to be monitored and visualized.

2.2.3 Example

Let us consider the model from Figure 2.1 as an illustrative example of the systems that will be observed in this paper. The system starts in *init* with the variable x and all clocks t and tg initialized to 0. The initial location *init* is labeled as *urgent*, as it forces the location to take the outgoing edge pointing to the branch point *bp* in the beginning of a simulation. Up to this point, the system has to make a *non-deterministic* choice of either taking the upper path of the model by a chance of (1/4) or the lower path by a chance of (3/4). Thus the functionality of the system is determined by the previous choice. Regardless of which edge is taken, the delays of the transitions are synchronized by the use of the invariants ($t <= 1$), guards ($t >= 1$ && $tg <= 10$), ($tg > 10$) and

clock resets ($t = 0$). For the sake of simplicity, let us assume that the values of the clocks represent seconds. That being said, the overall functionality of the previous system increments the value of variable x (given the *assignment* $x = x + 1$) for ten seconds of a simulation with a probability of (1/4) and/or decrementing the value of variable x by one (given the *assignment* $x = x - 1$) for the same amount of time with a probability of (3/4).

The results of a simulation of the model from Figure 2.1 can be seen in the Figures 2.2 and 2.3. The first figure represents the plotted data of the observed variable x of the system named *Process*. The second figure represents the simulation trace of all 3 runs of the simulation.

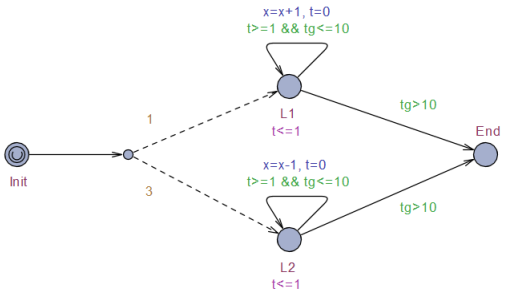

Figure 2.1: Uppaal Model

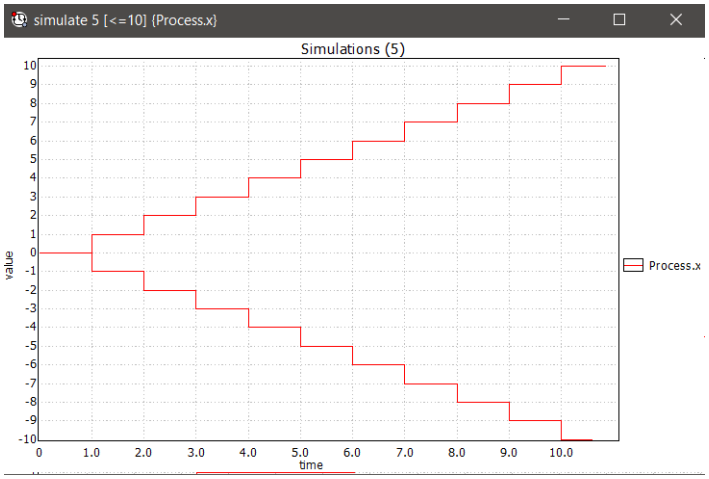

Figure 2.2: Uppaal Simulation Trace Graph

```
#### Simulations (3)
# Process.x #1
0.0 0.0
0.0 0.0
1.0 0.0
1.0 -1.0
2.0 -1.0
2.0 -2.0
3.0 -2.0
3.0 -3.0
4.0 -3.0
4.0 -4.0
5.0 -4.0
5.0 -5.0
6.0 -5.0
6.0 -6.0
7.0 -6.0
7.0 -7.0
8.0 -7.0
8.0 -8.0
9.0 -8.0
9.0 -9.0
10.0 -9.0
10.0 -10.0
10.6081 -10.0
# Process.x #2
0.0 0.0
0.0 0.0
1.0 0.0
1.0 -1.0
2.0 -1.0
2.0 -2.0
3.0 -2.0
3.0 -3.0
4.0 -3.0
4.0 -4.0
5.0 -4.0
5.0 -5.0
6.0 -5.0
6.0 -6.0
7.0 -6.0
7.0 -7.0
8.0 -7.0
8.0 -8.0
9.0 -8.0
9.0 -9.0
10.0 -9.0
10.0 -10.0
10.8384 -10.0
# Process.x #3
0.0 0.0
0.0 0.0
1.0 0.0
1.0 1.0
2.0 1.0
2.0 2.0
3.0 2.0
3.0 3.0
4.0 3.0
4.0 4.0
5.0 4.0
5.0 5.0
6.0 5.0
6.0 6.0
7.0 6.0
7.0 7.0
8.0 7.0
8.0 8.0
9.0 8.0
9.0 9.0
10.0 9.0
10.0 10.0
10.1599 10.0
```

Figure 2.3: Uppaal Simulation Trace Data

2.3 Graph Theory

In the implementation of this project, we represent observations of system simulations as *directed acyclic graphs*, also called *learned models*, by applying variants of techniques like *graph matching* and *graph edit distance* with the incorporation of distance metrics like *Euclidean distance* in an incremental fashion. Therefore we begin by giving the definition of a graph and then an overview of the previous techniques. A more detailed explanation of the mentioned subjects can bee seen in the following two papers of Vincenzo Carletti [5] and Bengoetxea Endika [2]. The first paper mainly focuses on graph edit distance and graph theory, while the other on the graph matching problem. Nevertheless, it is important to mention that the graph matching problem is categorized as NP-Complete.

2.3.1 Graph Definition

A graph is a structure $G = (V, E)$ composed of two finite sets: a node set V (or vertices) that represents objects in a domain and an edge set $E \subseteq V \times V$ which represent the relationships among objects. Each edge $e \in E$ is a couple of nodes $e = (u, u')$ where the nodes connected are called *adjacent*. The set $adj(u) = \{u' \in V | \exists (u, u') \in E\}$ of the nodes adjacent to u is commonly named neighborhood. If the couple (u, u') is ordered, the edge e is said to be directed. A *directed graph* is a graph in which all edges are directed or oriented from one node to another. A directed graph is considered *acyclic* if the directed graph has a topological ordering, such that for every directed edge uv from node u to node v, u comes before v in the ordering.

Graphs can also bring semantic information by the use of labels and attributes. We take advantage of this properties to map probabilistic timed automata along their components as directed acyclic graphs.

Definition 2.3.1. (Labeled Graph) A labeled graph is a tuple $G = (V, E, \mu, v)$ where

- L is a finite alphabet of nodes and edges labels.
- $\mu : V \to L$ is a node labeling function.
- $v : E \to L$ is an edge labeling function.

Definition 2.3.2. (Attributed Graph) Similarly, given that $\mathbb{A}$ is a set of structured information . An attributed graph is a tuple $G = (V, E, \alpha, \gamma)$ where

- $\alpha : V \to \mathbb{A}$ is a node attribute function.
- $\gamma : E \to \mathbb{A}$ is an edge attribute function.

2.3.2 Graph Matching

In pattern recognition, an important problem consists of finding a mapping between the nodes and edges of two graphs, that satisfy a given set of structural constraints. Such problem is commonly known as the *graph matching problem*, whose goal is to find structural correspondence between graphs.

2.3.3 Exact Matching

Definition 2.3.3. (Exact graph matching) Given two graphs, model graph $G_M = (V_M, E_M)$ and data graph $G_D = (V_D, E_D)$, with $|V_M| = |V_D|$, the problem is to find a one-to-one mapping $f : V_D \to V_M$ such that $(u, v) \in E_D$ iff $(f(u), f(v)) \in E_M$. When such a mapping f exists, G_D is said to be isomorphic to G_M.

2.3.4 Inexact Matching

In some real situations the variability of the patterns, the noise in the processes or other causes, may produce deformation in the observed graphs. So two graphs may have similar structures, but with some extra or missing nodes and edges. For these cases, the conditions of exact graph matching are too strict to find out a mapping between two graphs. The most adopted solution is to make the matching process tolerant to deformations by introducing matching costs to penalize structural differences. The closer the structures of two graphs are, the lower is the cost to match them. A well known method to define the matching cost is the *graph edit distance*, which assigns a cost to each operation needed to transform one graph into another one. Please refer to Fig. 2.4 for an example of the graph edit distance technique.

Definition 2.3.4. (Graph edit operations) a graph edit operation is one of the following operations applied on a graph: Node/Edge removal, Node/edge insertion, Node/Edge substitution.

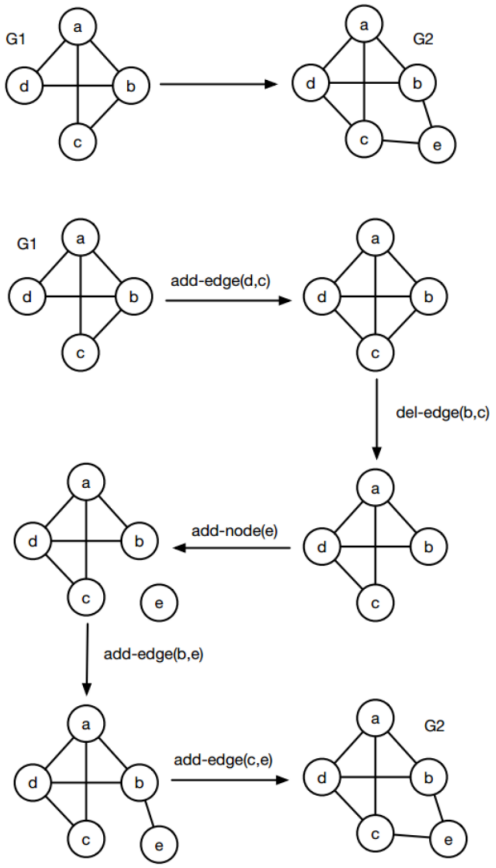

Figure 2.4: Graph edit distance example. Illustration taken from [5]

Definition 2.3.5. (Edit path) an edit path of a graph G is a sequence of elementary operations applied on G. It is possible to assign a cost to each simple operation. Among all possible transformations that involve two graphs $G1$ and $G2$, we are interested in the cheapest one, whose cost is the graph edit distance $d(G1, G2)$ between $G1$ and $G2$.

2.3.5 The Assignment Problem

Graph edit distance compares two graphs with *static structures* that do not change over time. We on the other hand, apply a similar approach incrementally and on the fly, as we attempt to match graphs with *dynamic structures* that change over time. Despite this difference, both approaches share another problem, which is the assignment of nodes from one graph to the other. We tackle this issue by mapping the nodes of graphs with the use of the euclidean distance as a mapping function. Further details are given in Chapter 4

Definition 2.3.6. (Assignment) an assignment is usually represented as a bijective mapping $\varphi : X \to Y$ between two finite sets $X = \{x_i\}_i$ and $Y = \{y_i\}_i$ of size $|X| = |Y| = n$. By assigning the n elements of X to the n elements of Y we get an assignment φ corresponding to a permutation $(\varphi(1), \ldots, \varphi(n))$, where the first element is assigned to $\varphi(1)$, the second to $\varphi(2)$ and so on.

Definition 2.3.7. (Euclidean distance) [19] The euclidean distance or euclidean metric is the ordinary straight-line distance between two points in euclidean space. With this distance, euclidean space becomes a metric space. In Cartesian coordinates, the euclidean distance of between ponint $p = (p_1, p_2, \ldots, p_n)$ and point $q = (q_1, q_2, \ldots, q_n)$ is given by:

$$d(p,q) = d(q,p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \quad (2.3)$$

3 Related Work

In this section we share a rough overview of the principal topics that inspired this paper, and that are also on-going and related research subjects. The main concepts discussed are: *active automata learning*, *incremental minimal construction of deterministic finite automata* and *graph matching*.

Despite the fact that we do not interact directly with models in our incremental learning approach, we do share similar challenges that apply to active automata learning. There is no straightforward pattern that clearly indicates the termination and correctness of learning, as seen in the paper from Howar, Falk and Steffen, Bernhard [13]. In active learning, it is possible to indicate partial correctness, because learning terminates after the equivalence is guaranteed by the execution of equivalence queries. As for termination, the amount of time to learn a system remains to be proved, because the learning process highly depends on queries to the system, which execution may vary in time depending on the received observations of a system. It is to be remarked that in our incremental learning approach we are also not able to determine full correctness and termination, due to the lack of interaction with the system and the types of observation that we evaluate. Nevertheless, there exists papers in which active learning is used with simple practical examples, as seen in paper written by Steffen, Bernhard and Howar, Falk and Merten, Maik [20], where models are learned not only with automata, but with *Mealy machines*.

Incremental learning and construction of automata has been also discussed and applied to improve the efficiency and optimization of automata. Our approach consists on incremental learning and measurement of probabilistic timed automaton, which is strongly related to the approaches of constructing incremental minimal automata from the book of *Optimization of automata* [8]. In this book, incremental construction of minimal finite-state automata is applied in the context of natural language processing by incrementally verifying that letters of a language are recognized by the minimal possible number of states. We do not ensure in our implementation that our *learned models* are minimal, but we are capable of avoiding redundant nodes by verifying the similarity of the nodes with Euclidean distances and cost functions. Minimization of automata is also a very important and related topic, because minimal automata might be easier to learn as automata with redundant nodes. There exists a great variety of algorithms for automata minimization, which are very well described in the paper *Minimization of automata* [4].

Graph matching, as discussed in Chapter 2, is a powerful technique that involves matching nodes and edges of two graphs. We utilize similar techniques related to graph matching like *semantic matching*, which can be seen in more detailed in the paper of *S-Match: an algorithm and an implementation of semantic matching* [12]. In graph semantic matching, the nodes of a graph are identified by calculating the semantic rela-

tions between concepts of nodes and not their labels. This is a very interesting concept that we applied by labeling nodes depending on their functionalities, with the use of Euclidean distances. Graph matching itself is a very wide and complicated topic that has been applied in many areas of the real world. One example can be web search engines, in which nodes and edges of enormous graphs are coupled by relating and categorizing sources or information of graphs by hubs and authorities, explained in paper the paper of *Graph similarity scoring and matching* [21]. Other interesting areas are: image analysis, biometric identification, video analysis, document processing, among others, very well explained in the paper of *Thirty years of graph matching in pattern recognition* [7].

4 Incremental Learning

The learning process of a system begins by analyzing a *trace* from a simulation. A trace is a set of *observations* from which *equations* are derived, by fitting the trend of the data to mathematical functions. Each equation represents a *functionality* with knowledge to be incorporated into a *learned model* as a *node*. We use an *incremental learning* approach, as each node to be incorporated is priorly *measured* or evaluated in terms of cost with respect to the learned model. Simultaneously, the structure of the learned model is matched against the one from the observed system, only to indicate the progress of the learning process.

In this chapter we discuss how incremental learning works by first explaining how the data from observations is fitted to equations in the *Data Fitting* section. Then we describe how the equations are used to learn information in section *Data Learning*. We discuss how learned data is modeled in section *Data Modelling*, and in the last section *Cost Evaluation*, we explain how the use of costs influence the modelling and learning of data.

4.1 Data Fitting

We consider an *observation* as the value of a variable in an exact time, obtained from a simulation trace. By gathering more than one observation and fitting the curve of the data, we are able to derive equations that express the *behavior* of a variable given a period of time. The procedure is implemented by storing the data of a trace using a *buffer* (e.g. 4.1) structure, that gathers observations from a simulation trace, which are later fitted to equations, as seen in algorithm 1. The buffer gathers observations, as long as the trace was not fully traversed (line number 2). The information of the *buffer* was designed as a *Map* in such way that the exact *time* and *value* of an observation can be extracted from it (lines 3-7). Once all observations were extracted from the *buffer*, they are fitted and stored as equations in a list of equations (lines 8-9).

Time	Value
0	**1**
1	**2**
2	**3**
3	4
4	5

Time	Value
0	1
1	**2**
2	**3**
3	**4**
4	5

Time	Value
0	1
1	2
2	**3**
3	**4**
4	**5**

Figure 4.1: Example of buffer storing information of a trace

Algorithm 1 Data-Fitter

```
1: procedure FITDATA(timeStep, buffer, traceData)
2:     while windowSlideLimit == false do
3:         for each entry ∈ buffer do
4:             time ← entry.getKey()
5:             value ← entry.getValue()
6:             observations.add(time, value)
7:         end for
8:         equation ← fitEquation(observations)
9:         equationsList.add(equation)
10:        windowSlideLimit ← slideBuffer(timeStep, traceData)
11:    end while
12: end procedure
```

4.2 Data Learning

Given that each equation trace is organized by time and that the initial node of the learned model *lm* is known. For each equation in an equation trace, we proceed to traverse the learned model to measure the similarity between existing neighbor nodes or *direct successors* and the observed equation. An equation is incorporated to the model depending on the distance that it has among the direct successors (if any), and the knowledge that we obtain from it, as seen in Algorithm 2.

Algorithm 2 Learner

```
1: procedure LEARNER(equationTrace)
2:     parent ← lm.getInitialLocation()
3:     for each equation ∈ equationTrace do
4:         directSuccessors ← lm.getDirectSuccessors(parent)
5:         if directSuccessors.length > 0 then
6:             distances ← measureDistances(directSuccessors,equation)
7:             lastModifiedNode ← incrementalLearning(distances)
8:         else
9:             lastModifiedNode ← addNodeToLearnedModel(equation)
10:        end if
11:        parent ← lastModifiedNode
12:    end for
13: end procedure
```

4.2.1 Measuring Distance

The similarity between direct successors and an observed equation is measured based on Euclidean distance with Algorithm 3. Given that *time steps* represent the time range

in which an equation was observed. For each *neighbor* among the direct successors, we evaluate the fitted function of the observed equation *observed function* and the one from the neighbor *neighbor function* to obtain the two sets of points *observed points* and *neighbor points*, which will later be used to obtain the *Euclidean distance*. It is important to note that the euclidean distance is normalized to a value between 0 and 1 in line 10 of the algorithm, and that we perform the same procedure for every neighbor and store every distance in the list *distance list*.

Algorithm 3 Measure-Distances

```
1:  procedure MEASURE-DISTANCES(directSuccessors, observedEquation)
2:      timeSteps ← observedEquation.timeSteps
3:      observedFunction ← observedEquation.fittedFunction
4:      for each neighbor ∈ directSuccessors do
5:          neighborFunction ← neighbor.fittedFunction
6:          neighborPoints ← evaluatefunction(neighborFunction, timeSteps)
7:          observedPoints ← evaluatefunction(observedFunction, timeSteps)
8:          distance ← getEuclideanDistance(observedPoints, neighborPoints)
9:          distance ← 1/(1 + distance)
10:         distanceList.push(distance)
11:     end for
12:     return distanceList
13: end procedure
```

4.3 Data Modelling

We attempt to merge similar observations to avoid redundant or unnecessary nodes in our learned model, by using the gathered distances from Algorithm 3. Among all distances, we retrieve the node with the closest distance (*closest node*) that satisfies our similarity criteria (*similarity threshold*), with the intention of evaluating the cost of adding a similar observation as a *new node* or as a *merged node*. If an observation does not satisfy the similarity threshold, then the closest node below the threshold (*far node*) is retrieved and added to the learned model as a new location. As depicted in Algorithm 4.

4.4 Cost Evaluation

The gathered knowledge of a system is always resembled by the latest structure and functionality of the learned model. An observed equation represents new knowledge, which we would like to incorporate to our learned model. The way that we evaluate this incorporation is by estimating the cost of the following points: Number of nodes required to express the information in the learned model (either as a new node or a

Algorithm 4 Incremental-Learning. An observed equation is added to the learned model as a new node or as a merged node, based on the closeness that it has among direct successors.
Input: similarity threshold *sim_th*, given by the user

```
1:  procedure INCREMENTAL-LEARNING(distances, observedEquation)
2:      closestNode ← getClosestDistanceAboveThreshold(distances, sim_th)
3:      farNode ← getClosestDistanceBelowThreshold(distances, sim_th)
4:      if closestNode.isNotEmpty() then
5:          nextNode ← addLeastExpensiveChange(closestNode, observedEquation)
6:          return nextNodeToTraverse
7:      end if
8:      if farNode.isNotEmpty() then
9:          nextNode ← addNodeToLearnedModel(farNode, observedEquation)
10:         return nextNode
11:     end if
12: end procedure
```

replacement), plus the error that it may transmit or propagate to other relevant close nodes from the learned model, as seen in Equation 4.1.

$$Cost = nodeCount * nodeCost + propagation \tag{4.1}$$

Based on the previous general cost function, we derive two types of cost functions to decide whether to add or replace nodes: *replacement cost* and *addition cost*, where we always conclude that the function with the lowest cost is the one that determines how an observed equation will be incorporated to the learned model.

For explanation purposes, we will be making the following assumptions: 1) The Uppaal model from Figure 4.2 represents our learned model, 2) We have observed an equation $x = x + 1$ from times *0-10* of a simulation and wish to incorporate it to the model by utilizing cost functions.

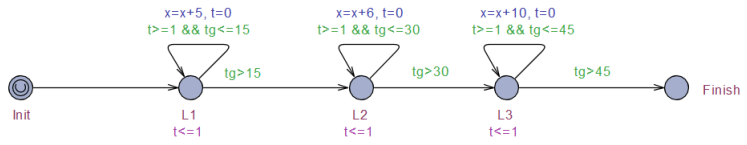

Figure 4.2: Learned model example

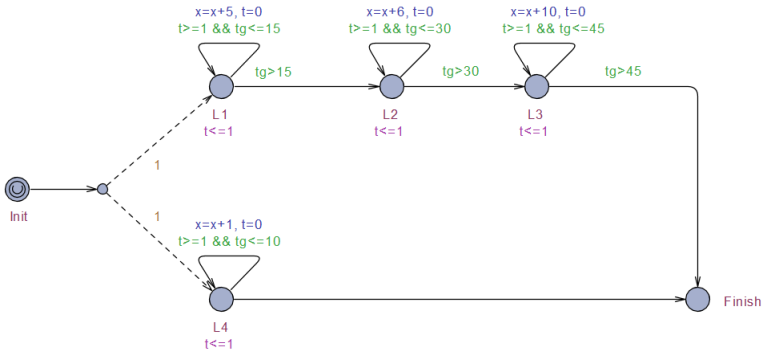

Figure 4.3: Node addition to learned model example

4.4.1 Addition Cost

The addition of an observation consists of adding a node to the learned model without modifying other nodes functionalities and time constraints. The only cost for this change is determined by the *addition cost*, thus the function cost for this case is determined by: $additionCost = nodeCost$.

An example of adding the assumed observation as a new node to the learned model can be seen in Figure 4.3.

4.4.2 Replacement Cost

The replacement of a node in the learned model consists of merging the functionality and time constraints of an observation with the ones from a node, without adding extra nodes. The cost of this change is mainly determined by two types of propagation: functionality propagation ($propagation_f$) and time propagation ($propagation_t$):

$$replacementCost = propagation_f * functionalityCost + propagation_t * timeCost, \quad (4.2)$$

where the first propagation represents how the modification of the functionality of a node can affect the distance of other nodes, while the second represents how modifying time constraints can affect other nodes. We will explain how location $L1$ is merged with the observation in the following subsections.

4.4.3 Time Propagation

The already learned time constraints of $L1$ state that the location should be active for 15 seconds of the simulation and then take the edge to location $L2$. We have observed a functionality with a duration of 10 seconds (5 seconds less than $L1$). The possibilities of merging the time constraints of both the observation and $L1$ are given by the following cases:

- Ignoring: We decide to ignore that we observed 5 seconds of the observation (e.g. times from 10 - 15) and not modify the time constraints of $L1$.
- Updating: We decide to modify the time constraints of $L1$, as depicted in Figure 4.4.
- Splitting: We decide to not modify the time constraints of $L1$ and add extra locations from the constraints that were not satisfied by $L1$. For this, we evaluate the time of the location $L1$ and the time of the observation Obs as sets, and add locations according to the difference of the time sets. An example can be seen in Figure 4.5)

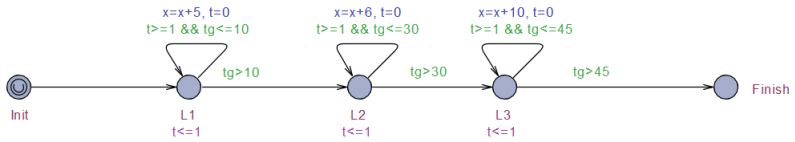

Figure 4.4: Time constraints replacement in learned model example

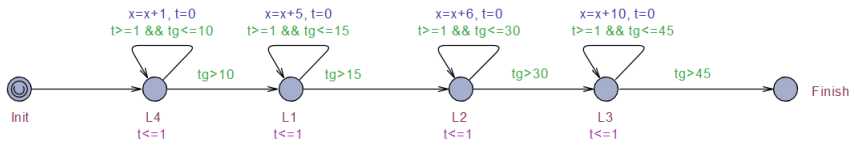

Figure 4.5: Time constraints replacement with extra location in learned model example

For the first two cases, the $propagation_t$ cost is represented by the number of unseen times, which is 5, multiplied by the *time cost*. The only time that the $propagation_t$ is zero, is in the last case when a location is added to keep the time constraints intact. Nevertheless, this last change causes the replacement cost to be raised by adding an additional location required to satisfy the observed time constraints.

4.4.4 Functionality Propagation

Assuming that we want to merge the functionality of location $L1$ ($x = x + 5$) with the observed functionality ($x = x + 1$), we need to obtain a new fitted function. For this, we utilized the same fitting methodology as shown in Algorithm 1. We retrieve two sets of points by evaluating both functions in the same range of time and obtain the new fitted function (e.g. $x = x + 3$). Once the new fitted functionality is replaced in $L1$, the $functionalityPropagation$ is determined by the difference of the Euclidean distance between the new functionality and the functionality of $L1$, along with the one from its

direct successors. The result of the change can be seen in Figure 4.6.

In this particular type of replacement, two aspects are considered:

- Substitution: In this case we consider how close is the Euclidean distance of the new functionality compared to the one that was substituted.
- Propagation: Here we perform the same comparison as in the previous case, but we now compare how the Euclidean distance of each direct neighbor is affected by the substituted location.

For the example in Figure 4.6, we can clearly see that changing the functionality of location *L1* from $x = x + 5$ to $x = x + 3$ has a negative effect, as the new functionality $x = x + 3$ is distanced from $x = x + 5$ and even more distanced from $x = x + 6$. The main goal of this calculation is to consider how changing the functionality of a location can impact not only the location itself, but also others around it.

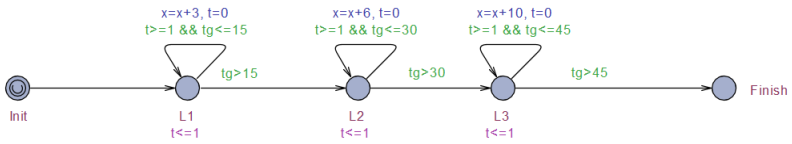

Figure 4.6: Replacement of the functionality of a location in the learned model example

5 Implementation

The implementation of this project was developed as a web application. The *front-end* or *user interface* utilizes the *JavaScript* open source library of *React*. The *back-end* was developed in *JavaScript ES6*, and it utilizes the state management tool *redux* to communicate with the *front-end*.

The web application mainly consists of four dependent *React Components* named: `Data Manager`, `Fitting Manager`, `Learning Manager` and `Graph Matcher`. In this chapter we will explain how the components communicate between each other, and how they are used as routine to perform incremental analysis of system observations.

5.1 Data Manager

As mentioned in Chapter 4, incremental learning begins by obtaining simulation data of an observed system. The `Data Manager`, which is the first section of our tool, is in charge of organizing the data of simulations. A simulation trace file from Uppaal can be imported and also visualized in the tool, as seen in the *Data Plotted* from Figure 5.1. The information from the trace is parsed in this component as an *array* of *Map* objects that contain the points of the data per simulation run. The user can also pick the desired *buffer size* and *time step* value, which is going to be delivered to the `Fitting Manager`, along with the previous information for further analysis.

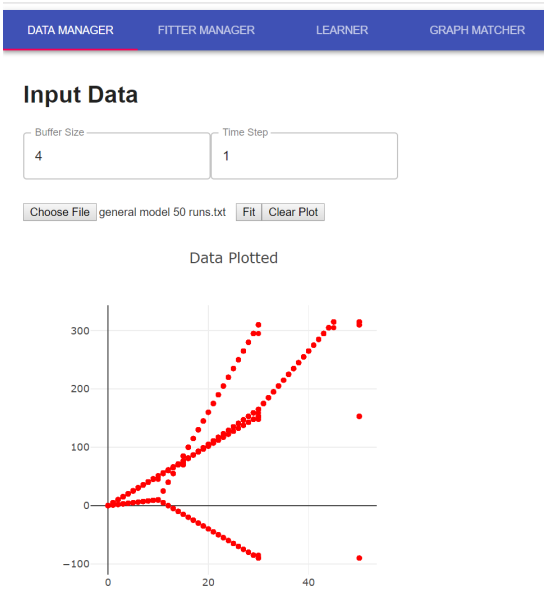

Figure 5.1: Implementation Data Manager

5.2 Fitter Manager

Given an array of maps created by the `Data Manager`, the `Fitter Manager` is in charge of fitting the observations from each map by utilizing Algorithm 1 and the *ml-levenberg-marquardt* library, to transform the array of maps into an array of `Equations` (structure shown in Figure 5.3) which represent the behavior of an observed system, as seen in Figure 5.2

One can see in the *Fitted Functions* text area, the results of fitting the information of each *Simulation run*. For every simulation run, there exists a section that contains the details of each fitted `Equation`. Let us take as an example the first fitted equation from *Simulation 0*. The general information of the equation is written in black and contains the name of the equation (*EQ1*), followed by the function structure ($ax + b$), the type of function that was fitted (*linear*), and the *Start-time* and *End-Time*, that represent the range of time in which the observations were fitted. The details written in gray show the parameters that were fitted by the *ml-levenberg-marquardt* library to evaluate the function, along with error of the calculations (*Regression error*), that reflect the suitability of the fitted function with respect to the observed values.

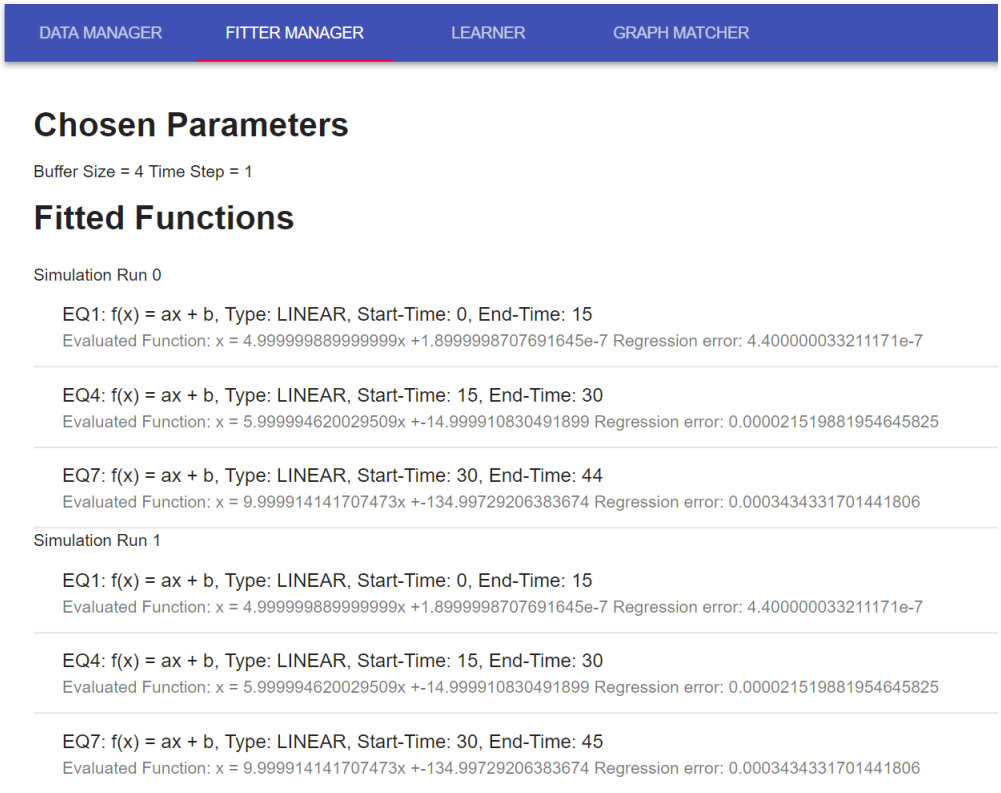

Figure 5.2: Implementation Fitter Manager

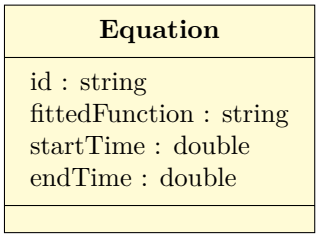

Figure 5.3: Structure of an equation object

5.3 Learning Manager

The `Learning Manager` is responsible to perform the incremental learning approach, as explained in Chapter 4, with the values of the parameters that the user selects. By default, the *Similarity Threshold* is set to 0.5, *Time Cost* to 0.1, *Functionality Cost* to 0.1 and *Location Cost* to 0.5. Due to the fact that normalized Euclidean distance values between the range of zero and one are used in the incremental learning approach, we allowed to change the values of each parameter by this same range. One can also restrict the possible values of the parameters to a set of numbers (e.g. by choosing an option from the check-boxes) and learn observations for each possible combination.

The tool begins the learning process after the user clicks the *Learn* button. There exists two types of learning that the tool may perform: 1) With a model to reference, or 2) Without a model to reference. For the first approach, one can learn observations from systems but without evaluating the learning progress, as there is no possibility to interact with a model that resembles the observed system. Whereas in the second approach, one can import a model that resembles the observed system (e.g. Original model) producing the observations, so that the similarity of the models can also be measured *on-the-fly*. Nevertheless, it is possible for both types of learning to visualize the evolution of the learned model, by selecting the desired model from the *Learned Models* drop-down list. It is also important to notice that the models are constructed as directed acyclic graphs, with the only exception that *self-loops* are allowed, as seen in in Figure 5.4.

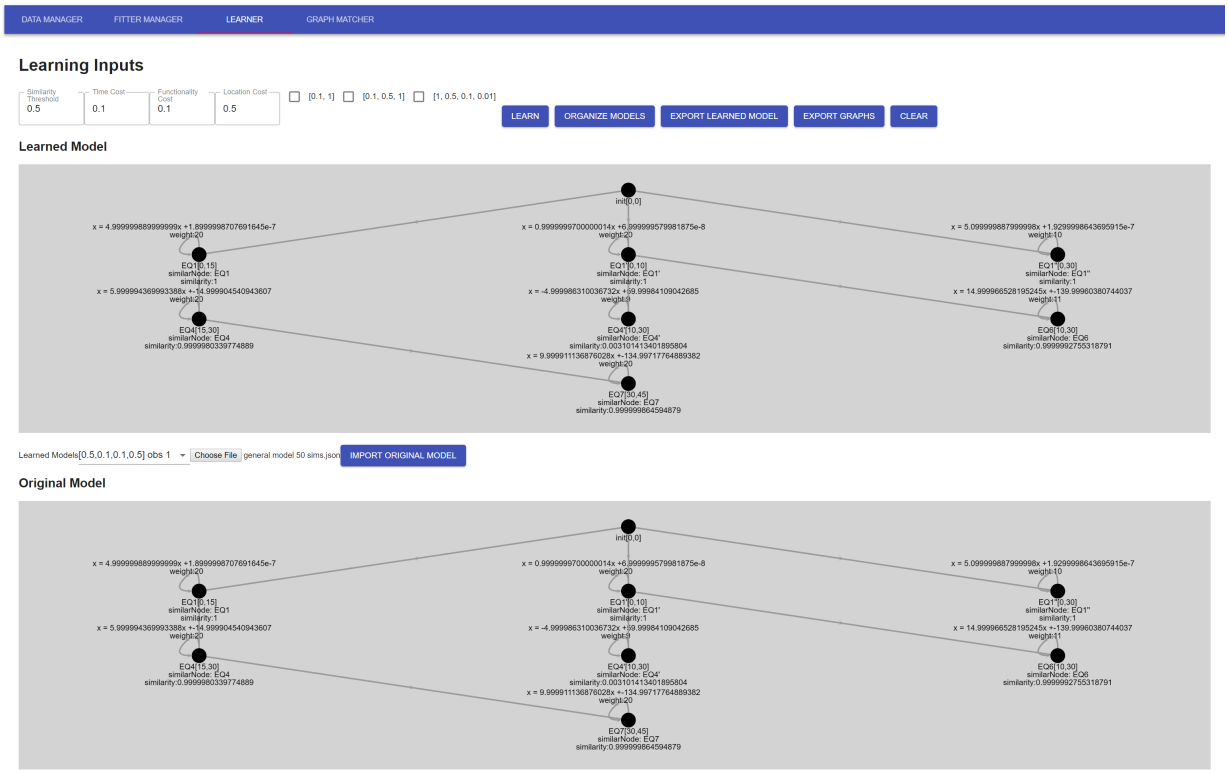

Figure 5.4: Implementation Learning Manager

5.4 Graph Matcher

The `Graph Matcher` component was created for visualizing how the matching of models is performed in incremental learning, whenever there exists an *original model* (e.g. the imported model from `Learning Manager`) that can be compared against the *learned model*. The similarity of the models is calculated by traversing both graphs simultaneously and matching the nodes, disregarding their labels and only by judging how close they are in terms of Euclidean distance. The user interface of the `Graph Matcher`, as seen in Figure 5.5 is essentially the same as the one from the `Learning Manager`, with an additional section called *Matched Model*, where one can see which nodes from the learned model were matched with the ones from the original model. For simplicity and explanation purposes, we matched the two models from Figure 5.4 and show the results in Figure 5.6. As seen in Figure 5.6, the learned model was successfully matched, as it is identical to the original model. The *Graph Matcher* facilitates the interpretation of the comparison by coloring the edges and nodes in green when they were matched, and in red when they were not. Also, the *Matched Model* section includes information of which node from the learned model was matched in the original model. The label *lm* represents the name of the node of the learned model, and the label *om* corresponds to the name of the node from the original model that was matched.

Figure 5.5: Implementation Graph Matcher

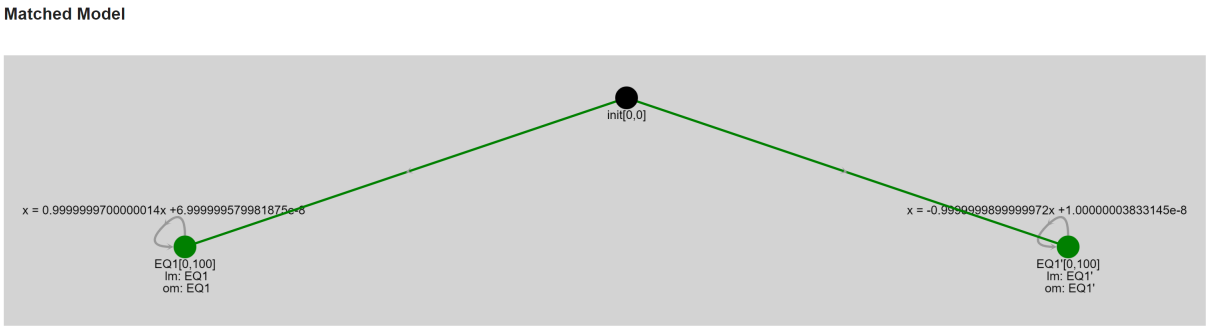

Figure 5.6: Implementation Graph Matcher - Model matched section

6 Experiments and Evaluation

In the following section we will demonstrate the impact that the values of the cost function parameters and the similarity threshold can have in the incremental learning approach. Two sets of experiments were conducted. The first set consists of evaluating the impact of giving priority to one of the four possible parameters that we take into consideration in our approach, which are: *similarity threshold*, *time cost*, *functionality cost* and *location cost*. While the second set consists of finding optimal combinations of parameters that lead to good incremental learning.

6.1 Experiment 1

For the first type of experiment we concentrate on learning one model, depicted in Figure 6.1, and also referred as *original model*. The model is categorized as complex for this project because it involves a mixture of branched and sequential nodes. We focus in this section on evaluating how the incremental learning approach performs when a particular parameter is prioritized. For this, we simply set the prioritized parameter to the value of 1 and the others to the minimum allowed value of 0.1. We do not experiment with values of 0 because that would completely ignore some parameters of our approach, and it would also contradict the nature of this project, as each parameter has a role in the process.

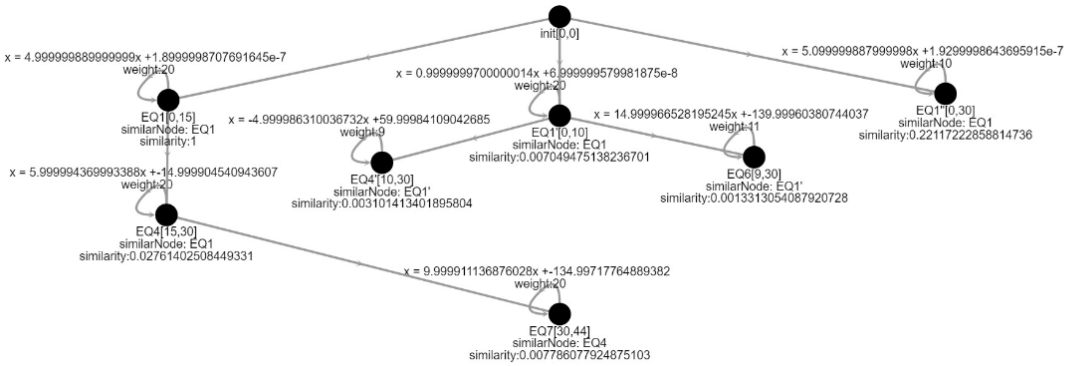

Figure 6.1: Experiment 1 Original Model

6.1.1 Sub-Experiment 1 - Similarity Prioritized

The *Similarity threshold* is responsible for discriminating which learned nodes are close enough to be merged in the learning progress. In this experiment, its value is increased to 1, meaning that only completely identical locations will be considered in the evaluation of merging nodes via the cost function of our approach.

Results

Despite the difference between the original model and the learned model from Figure 6.2, one can notice that the learned model actually contains the structure of the original one, but with additional nodes.

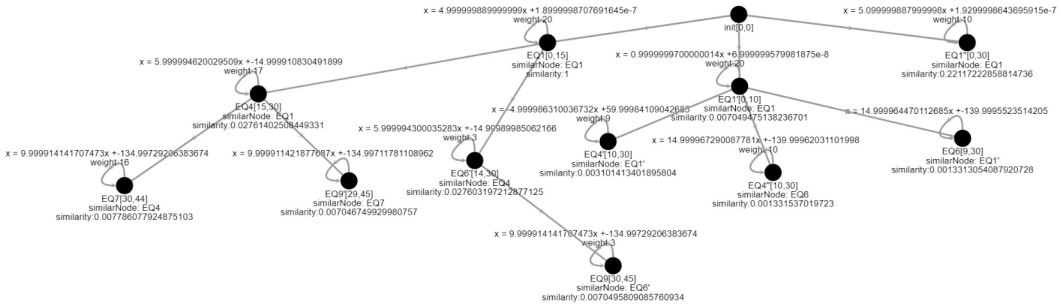

Figure 6.2: Sub-Experiment 1 - Similarity Prioritized - Learned Model

Evaluation

The nature of this experiment can be explained by analyzing two particular learned models. The learned model from Figure 6.3 represents the best performance of the learning approach, as all of the observations until that point were completely identical (e.g. Figure 6.5 in observation 10). It is not until we see a slightly different observation, that the learned model begins to add redundant locations due to the strict criteria of the similarity threshold. One can see that the last two nodes (from left to right) of Figure 6.4 are identical in terms of functionality, but were discovered in slightly different time ranges. Node *EQ4''* was discovered in the time range of 10-30, while the other next to it was discovered in the range of 9-10. This minimal difference in time constraints is detected by the learner and therefore a new node is added, to not loose precision of the observed data. Also being the reason that the replacement cost from Figure 6.7 always stays in 0, because we are only replacing identical locations, according to the similarity threshold of 1. Whenever there is a slight difference in the learned observations, a new node will be added. Therefore, one can see in Figure 6.6 that peaks, represent each node that was added to the learned model. It is also important to notice that from observation 43 until the end of the simulation (e.g. observation 100) the graph similarity stays constant because there are no new observations at any point.

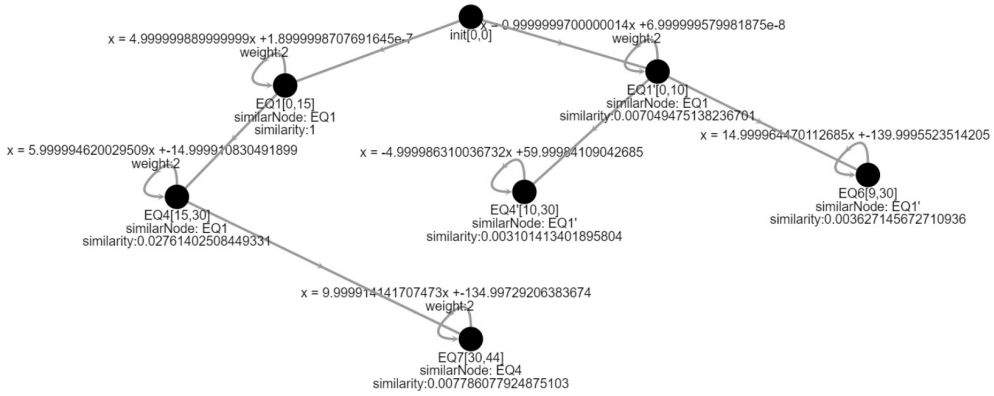

Figure 6.3: Sub-Experiment 1 - Similarity Prioritized - Learned Model - Observation 10

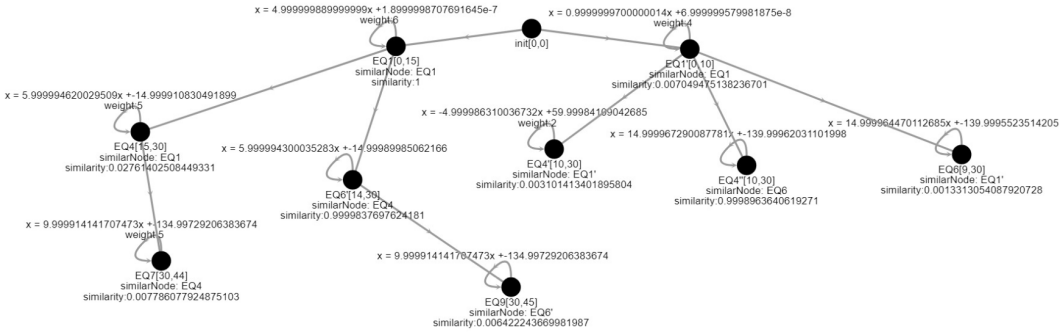

Figure 6.4: Sub-Experiment 1 - Similarity Prioritized - Learned Model - Observation 26

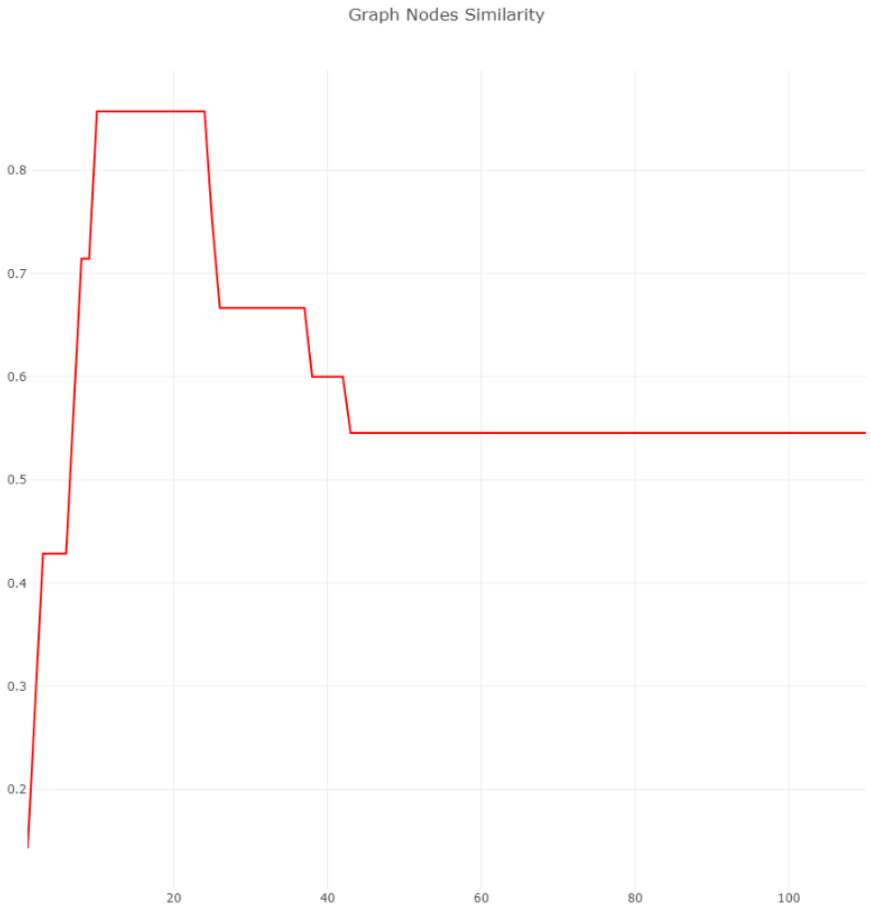

Figure 6.5: Sub-Experiment 1 - Similarity Prioritized - Graph Similarity

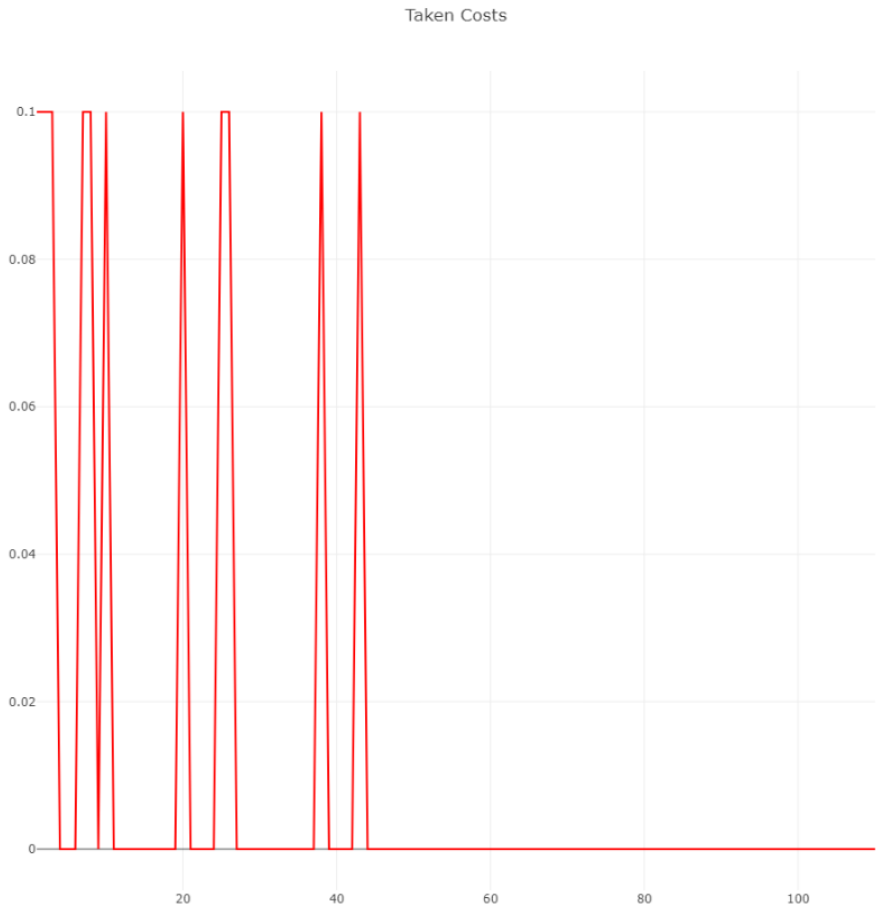

Figure 6.6: Sub-Experiment 1 - Similarity Prioritized - Taken Costs

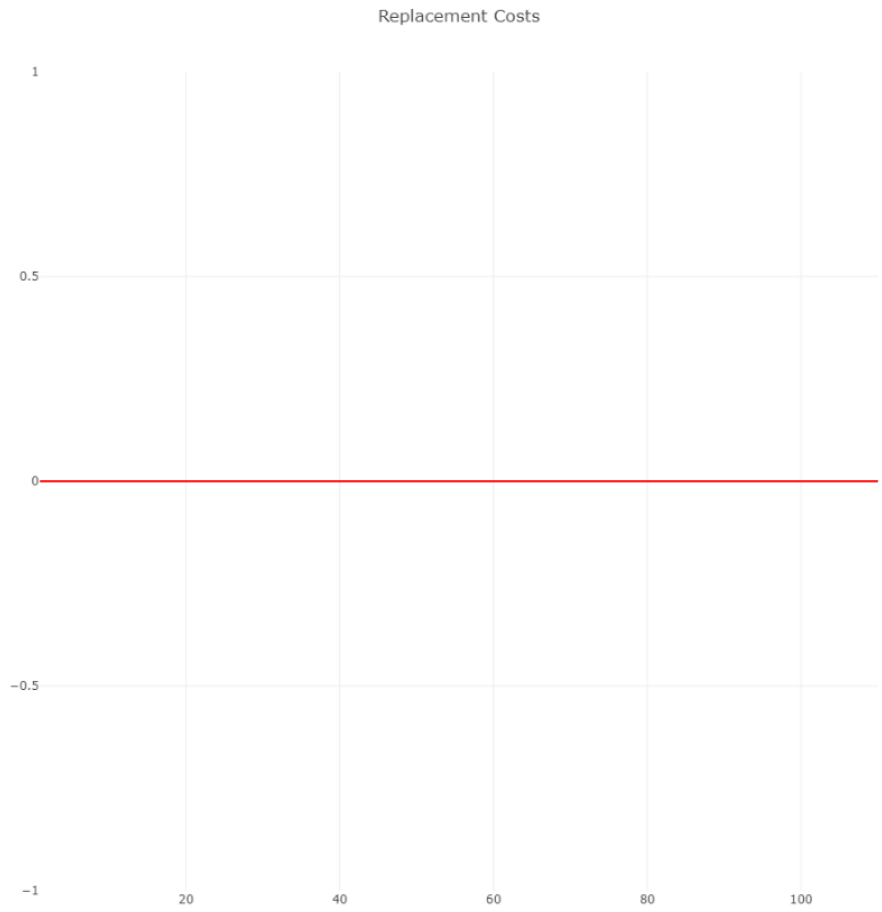

Figure 6.7: Sub-Experiment 1 - Similarity Prioritized - Replacement Costs

6.1.2 Sub-Experiment 2 - Time and Functionality Prioritized

For this experiment the *Time Cost* and *Functionality Cost* will be set to 1 simultaneously. This is because the cost function explained in Chapter 4, calculates the error propagation of nodes based on the difference of time constraints and functionality among other nodes and incoming observations. Therefore, it is not meaningful to ignore one of the two criteria while evaluating observations.

Results

In Figure 6.8 we can see the final learned model. It is very interesting and also meaningful to notice that the structure resembles the one from the previous experiment in Figure 6.2. This is to be expected with the selected parameters, as the nodes must be almost identical when merging. Meaning that minimal differences in terms of functionality or time constraints will have a high cost, and will therefore lead to the addition of new nodes.

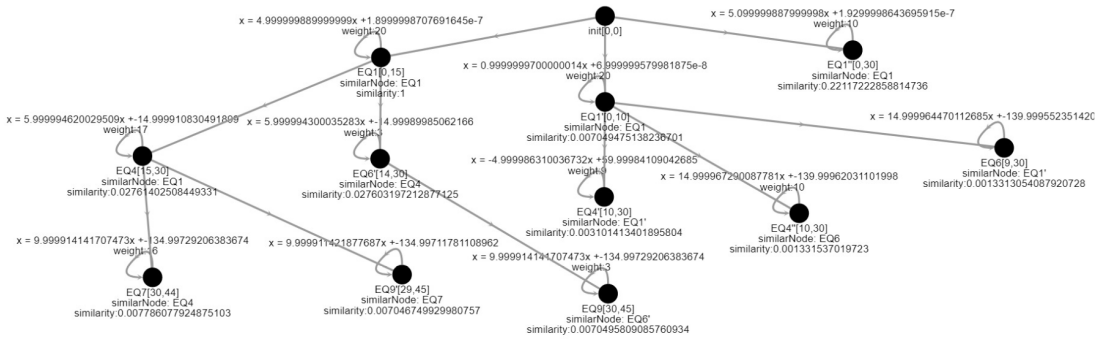

Figure 6.8: Sub-Experiment 2 - Time and Functionality Prioritized - Learned Model

Evaluation

Despite the fact that we obtained the same learned model as in the previous experiment, the observations that caused the evolution of the model were not the same. Let us take a look at the peaks of the replacement costs from Figure 6.13. Each peak represents an added location, as it is always the case that the cost of replacing a location surpasses the cost of adding one, due to the difference of time constraints and functionalities of nodes. We can see in Figure 6.9 that the so far learned model contains an extra location on the bottom-right corner (e.g. node labeled as EQ6), causing the similarity to stay in 0.85 (as seen in Figure 6.11), as the extra node could not be matched in the original model. In this particular case, the functionality cost did not play a role because node *EQ4"* and *EQ6* were identical in terms of functionality and only differed in time constraints by 1 time unit (same case as in the previous sub-experiment), causing the learner to add a new node to retain precision of the observations.

Let us now pay attention to the highest peak from the graph shown in Figure 6.13, as it represents the most expensive change that happened in the simulation. The reason of this peak is strongly related to the low value of the *similarity threshold* (e.g. 0.1). In observation 38, a merge between node *EQ1* and node *EQ1"* (the opposite nodes of the first *depth-level* of the graph) from the model shown in Figure 6.10 was attempted. This happened because the similarity of both nodes is 0.22 (number obtained by Algorithm 3), thus similar enough to be merged according the the similarity threshold value of 0.1. Nevertheless, the merge was unsuccessful because there also existed difference in the time constraints. One node was observed in the range of 0-15 time units and the other one in the range of 0-30 time units. This accumulated a high cost of merging the two nodes because of their different time constraints. For this case, the functionality cost already showed a high cost that would have forced the learner to add a new node, but the difference in time constraints was also taken into account, causing the final cost to be very high. In the end, we could not perfectly learn the observed model mainly because of the strict parameters that were utilized while trying to merge similar nodes. The learned model is still adding redundant nodes of similar observations when they are

not almost identical, as seen in each peak from Figure 6.12.

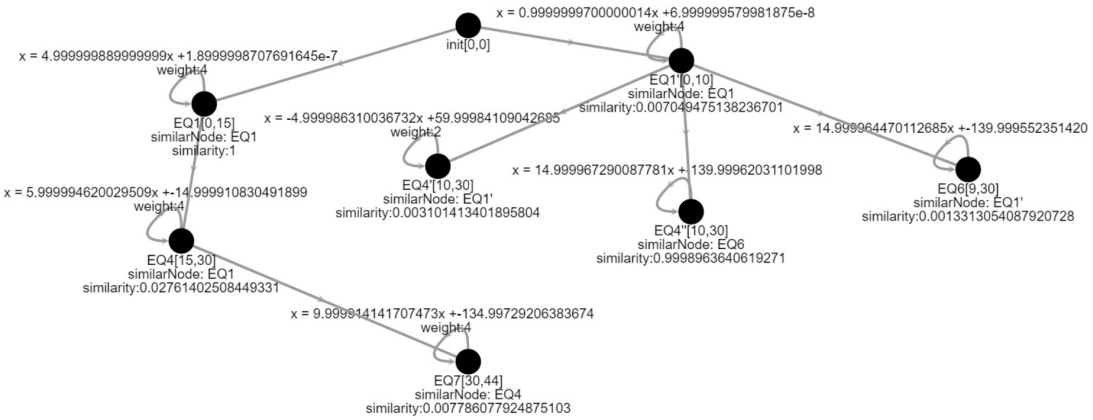

Figure 6.9: Sub-Experiment 2 - Time and Functionality Prioritized - Learned Model - Observation 20

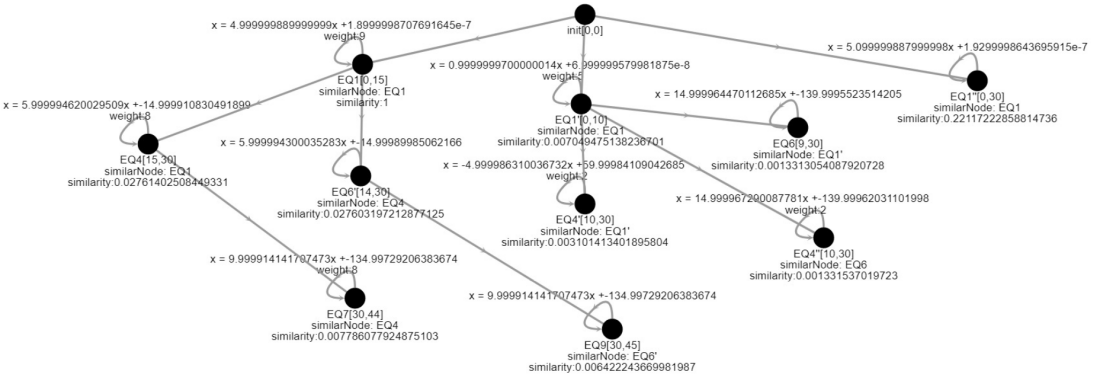

Figure 6.10: Sub-Experiment 2 - Time and Functionality Prioritized - Learned Model - Observation 38

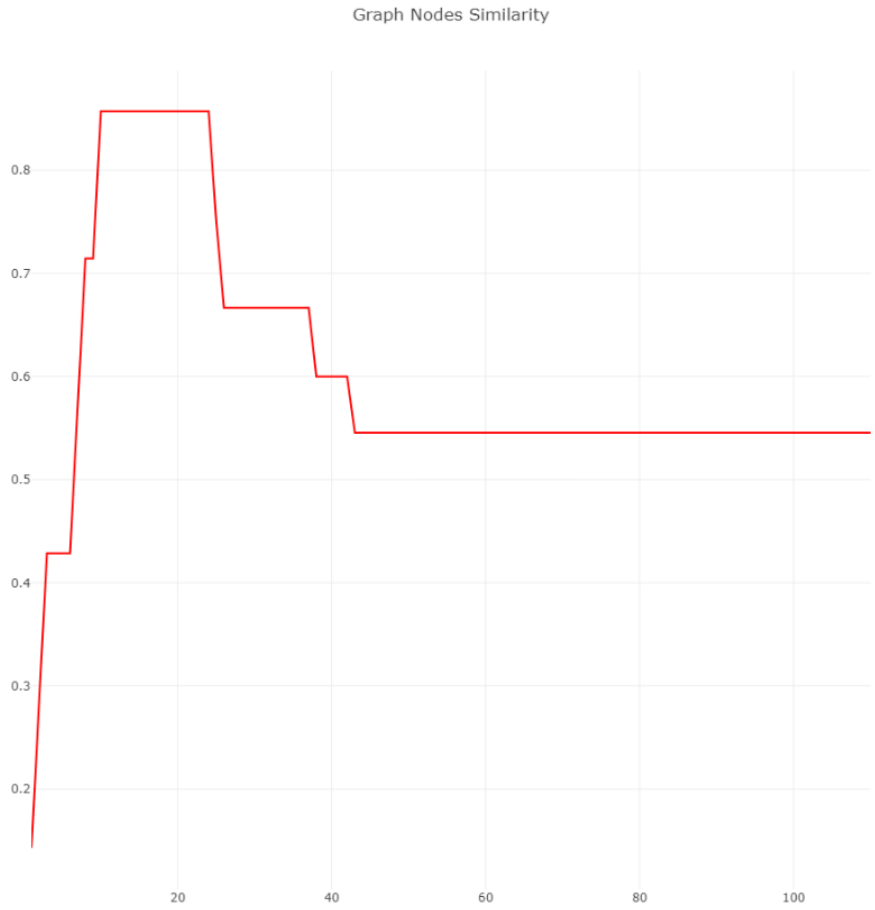

Figure 6.11: Sub-Experiment 2 - Time and Functionality Prioritized - Graph Similarity

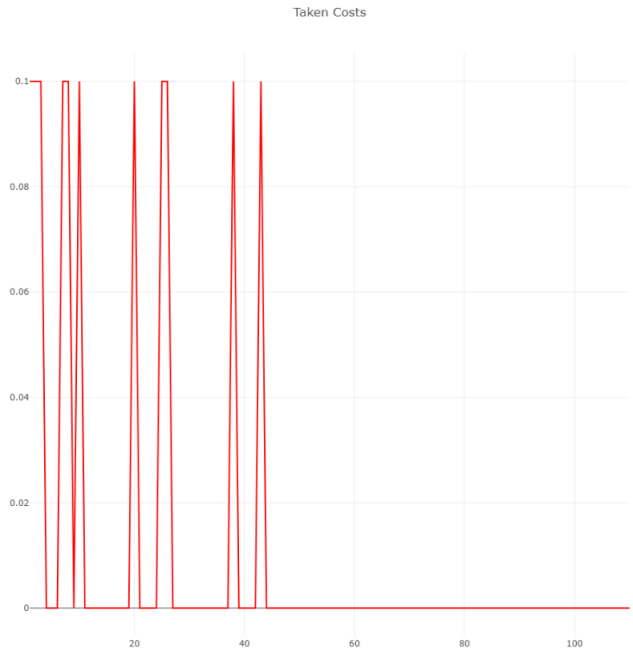

Figure 6.12: Sub-Experiment 2 - Time and Functionality Prioritized - Taken Costs

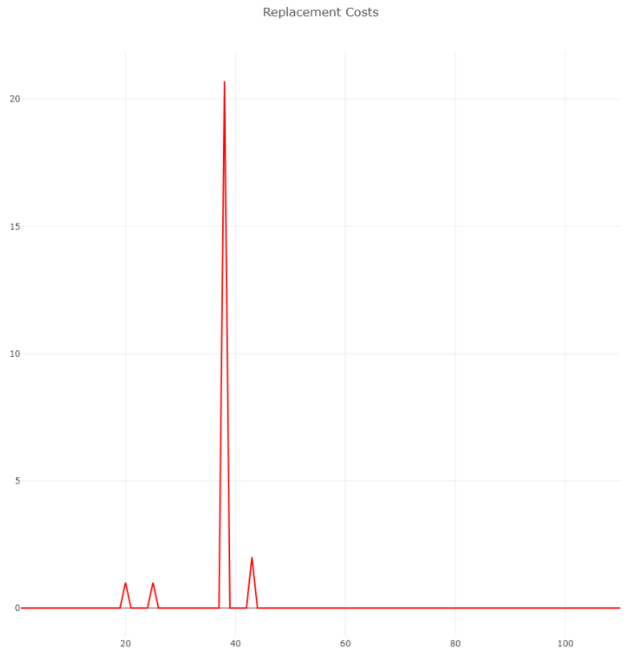

Figure 6.13: Sub-Experiment 2 - Time and Functionality Prioritized - Replacement Costs

6.1.3 Sub-Experiment 3 - Node Addition Prioritized

In this final experiment we set a high cost for adding locations (e.g. 1.0), while the other parameters are set to a low value (e.g. 0.1). Meaning that the learned model will be able to merge slight changes or noise from observations without needing to add an extra node every time that a different observation is learned.

Results

As seen in Figure 6.14, the learned model completely matches the original one. It was the case in this experiment because the learning process gave more priority (or cost) to the addition of new nodes. Which means that nodes that were added in the learned model were very distinguished among others in terms of time constraints and functionality, while similar nodes and observations were merged.

Evaluation

As we know that the original model was successfully learned, we will explain in detail why that was the reason by interpreting two particular observations that made the difference in the learning process. Looking at the first peak of Figure 6.18, the observation number 20 perfectly matched the functionality of node *EQ6* (node located at the medium-right corner of Figure 6.15), with the only difference of 1 time unit from the observed time

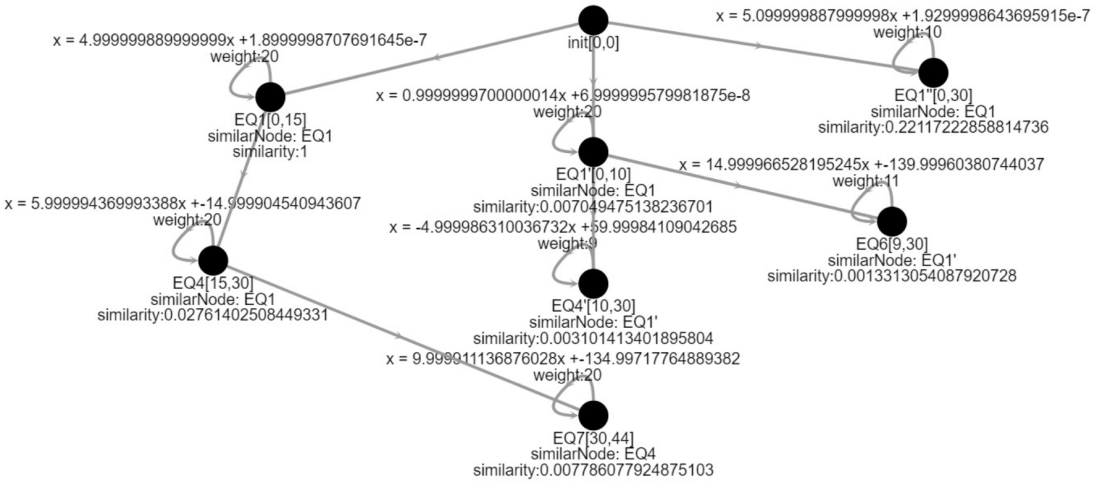

Figure 6.14: Sub-Experiment 3 - Node Addition Prioritized - Learned Model

constraints. In the previous experiments, this difference of time would have caused the learning process to add an extra node to express the observation. But for this experiment it is not the case, as the cost of merging minimal time constraints and functionalities of nodes is very low. Therefore, despite the minimal difference in time constraints, the learned model ignores the unseen time unit to keep a minimal number of nodes, but it also sacrifices some precision of the information learned. Although, if we take a look at the highest peak of the previous graph, one can easily see that the cost of replacing a node is higher than the one of adding a new one. The reason of this high cost is because a new observation was discovered in that point (e.g. the top-right node *EQ1"* from Figure 6.14). This last added node could not be merged with the closest node found (node *EQ1* located at the top-left corner from the same figure) mainly due to their differences in terms of functionality and time constraints. The functionality differed by 0.2 (e.g. Euclidean distance calculated by Algorithm 3) and the number of unseen times were 15 (as in the sub-experiment 2). Given the flexibility of the learned to merge similar nodes, this difference was considered great enough to discard the merging procedure and to add the observation as a new node. At this point, it is important to remark that if the nodes were to be merged, we could have had fewer locations in our learned model and less precision in the data. But this change would have caused the similarity of our models to decrease significantly. Consequently, one can see in Figure 6.17 that the cost of adding a node (e.g. 1.0) was taken instead of the one of merging nodes (e.g. 2.0). And it is indeed from this point the no new significant behavior was learned from the observations, as the similarity between the original model and the learned model remained optimal and constant, as depicted in Figure 6.16

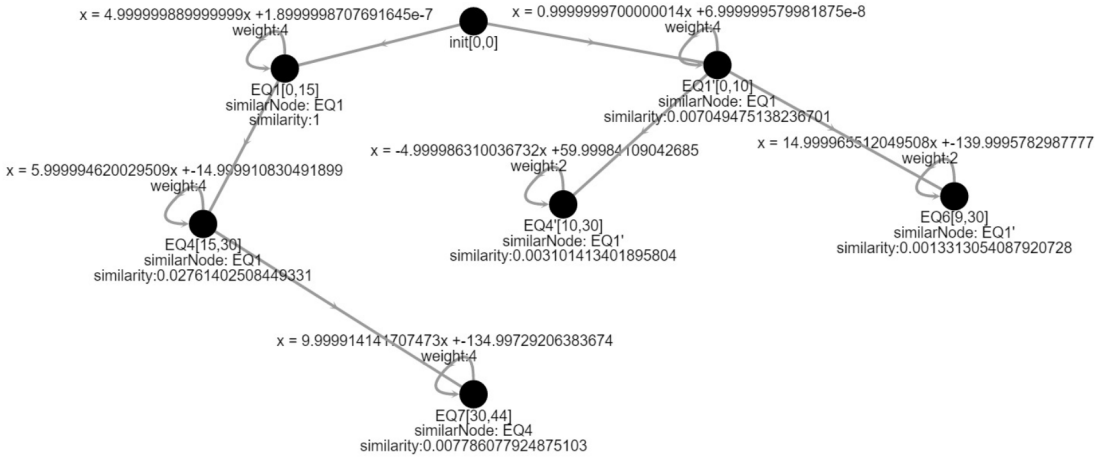

Figure 6.15: Sub-Experiment 3 - Node Addition Prioritized - Learned Model - Observation 20

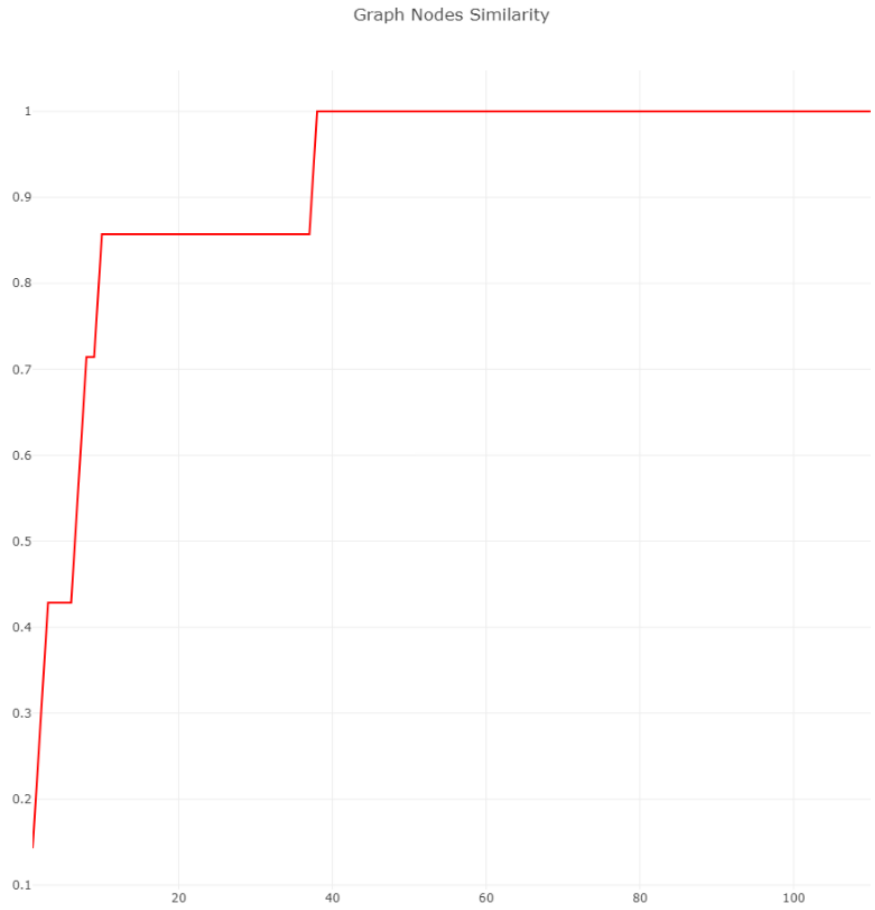

Figure 6.16: Sub-Experiment 3 - Node Addition Prioritized - Graph Similarity

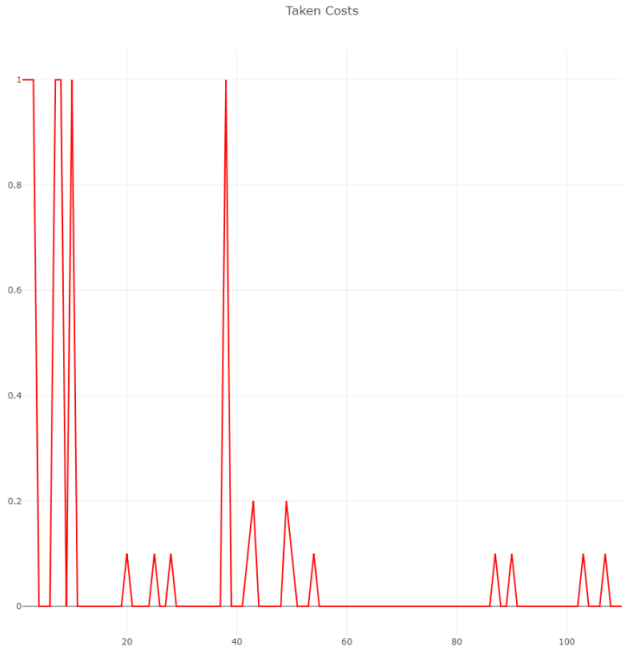

Figure 6.17: Sub-Experiment 3 - Node Addition Prioritized - Taken Costs

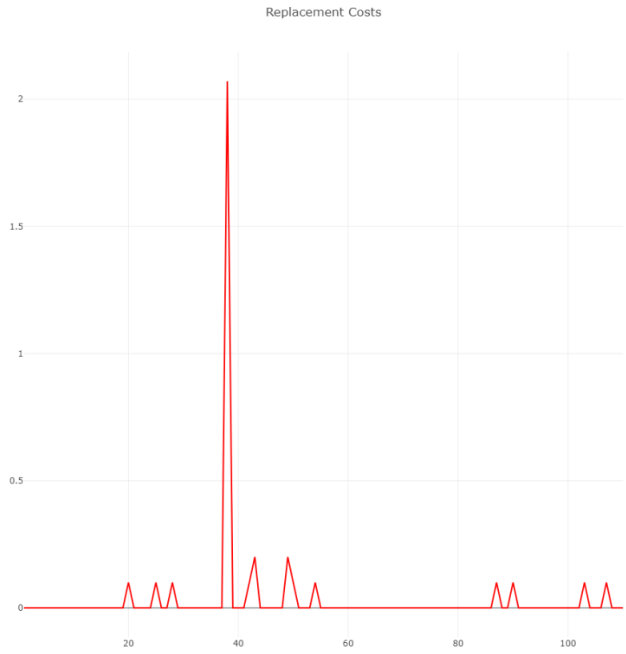

Figure 6.18: Sub-Experiment 3 - Node Addition Prioritized - Replacement Costs

6.2 Experiment 2

After observing how the incremental learning process behaves differently given prioritized parameters, in this section we attempt to find the optimal combination of parameters in which an observed model can be learned. In the previous experiment, we restricted ourselves to one type of combination and to two possible values (e.g. 0.1 and 1) for the *cost parameters* and *similarity threshold*. In this experiment we restrict ourselves to three different values: 0.1, 0.5, and 1, but with the possibility of allowing different combinations of them while learning observations. Therefore, we divided this section into two sub-experiments. In the first sub-experiment we learn a model with combinations of branches and sequential nodes. Whereas in the second sub-experiment we learn two simple and relatively small models. The purpose of this division is to demonstrate how the learning process behaves with different types of models.

6.2.1 Sub-Experiment 1

We used the same model as in Experiment 1 (e.g. Given in Figure 6.1), and learned it once again by using 81 combinations of different parameters for the *cost function* and the *similarity threshold*.

Results

At the end, 63 combinations could match 54% of the original model (learned model in Figure 6.20), 6 out of 81 matched 70% of original model (learned model in Figure 6.21), and only 12 combinations could match it by 100%.

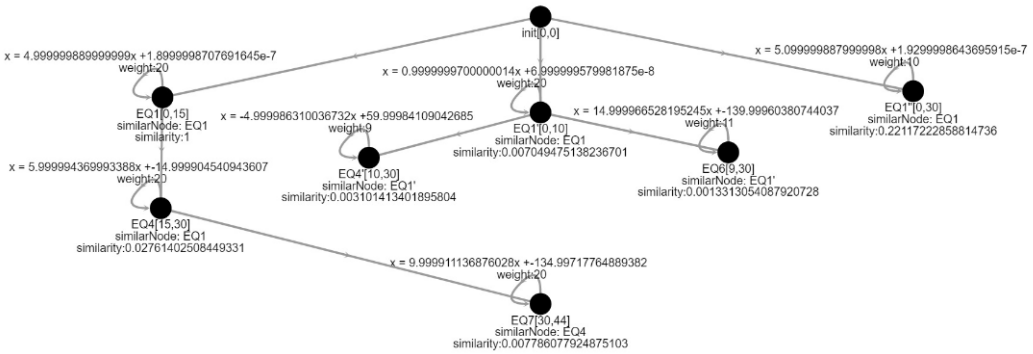

Figure 6.19: Sub-Experiment 1 - Original Model

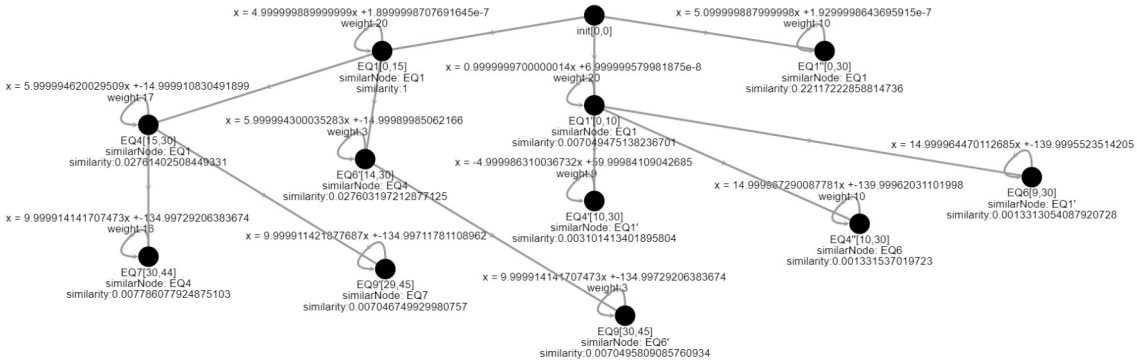

Figure 6.20: Sub-Experiment 1 - 54% of original mode learned

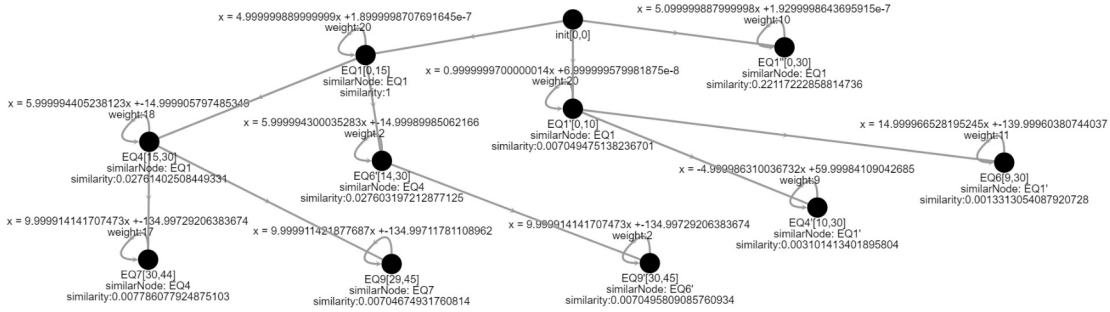

Figure 6.21: Sub-Experiment 1 - 70% of original mode learned

Evaluation

In the following tables we can see exactly which set of combination accomplished to learn the models mentioned previously. Each table will be analyzed separately and the conclusion of this experiment will be drawn until the end of this section.

Let us first take a look when the original model could be learned completely in Table 6.1. We can see that the *similarity threshold* never reached 1.0 and that the time constraints were given low priority at all times. In contrast to the *location cost* which never reached the value of 0.1. It was for almost all of the cases that the model could be perfectly learned, by only giving priority to the functionalities of the observations and the overall number of nodes required to express observations.

In Table 6.2 one can see the combinations when 70 % of the model could be learned. It was the case that raising the priority of the *time cost* had a negative effect on the learning process, despite the fact that the *cost function* and *location cost* remained relatively similar. As seen in Figure 6.21, this combination could not learned the model completely, only because two different time units of unique nodes were observed (nodes *EQ7* and *EQ9* from the same figure).

At last, we will discuss Table 6.3, were the worst combinations can be appreciated. In the majority of the cases, the model was wrongly learned whenever only identical observations were possible to merge (e.g. similarity threshold equal to 1). This is to be expected, as with a high *similarity threshold*, the learned model does not tolerate any noisy or slightly modified observations. The same applies whenever the *time cost* was set to a high value, as some of the observations had different time constraints and identical behaviors.

For this specific experiment, it was enough to ignore the minimal noise of the time constraints of the observations, and to separate the observations based on they're func-

tionalities because they were relatively far between each other.

Similarity Threshold	Time Cost	Functionality Cost	Location Cost	Graph Similarity
0.1	0.1	0.1	0.5	1
0.5	0.1	0.1	0.5	1
0.1	0.1	0.5	0.5	1
0.5	0.1	0.5	0.5	1
0.1	0.1	1	0.5	1
0.5	0.1	1	0.5	1
0.1	0.1	0.1	1	1
0.5	0.1	0.1	1	1
0.1	0.1	0.5	1	1
0.5	0.1	0.5	1	1
0.1	0.1	1	1	1
0.5	0.1	1	1	1

Table 6.1: Sub-Experiment 1 - Combinations that learned the complete model

Similarity Threshold	Time Cost	Functionality Cost	Location Cost	Graph Similarity
0.1	0.5	0.1	1	0.6999942199698811
0.5	0.5	0.1	1	0.6999942199698811
0.1	0.5	0.5	1	0.6999942199698811
0.5	0.5	0.5	1	0.6999942199698811
0.1	0.5	1	1	0.6999942199698811
0.5	0.5	1	1	0.6999942199698811

Table 6.2: Sub-Experiment 1- Combinations that learned 70% of the model

Similarity Threshold	Time Cost	Functionality Cost	Location Cost	Graph Similarity
0.1	0.1	0.1	0.1	0.545447328305253
0.5	0.1	0.1	0.1	0.545447328305253
1	0.1	0.1	0.1	0.545447328305253
0.1	0.5	0.1	0.1	0.545447328305253
0.5	0.5	0.1	0.1	0.545447328305253
1	0.5	0.1	0.1	0.545447328305253
0.1	1	0.1	0.1	0.545447328305253
0.5	1	0.1	0.1	0.545447328305253
1	1	0.1	0.1	0.545447328305253
0.1	0.1	0.5	0.1	0.545447328305253
0.5	0.1	0.5	0.1	0.545447328305253
1	0.1	0.5	0.1	0.545447328305253
0.1	0.5	0.5	0.1	0.545447328305253
0.5	0.5	0.5	0.1	0.545447328305253
1	0.5	0.5	0.1	0.545447328305253
0.1	1	0.5	0.1	0.545447328305253
0.5	1	0.5	0.1	0.545447328305253
1	1	0.5	0.1	0.545447328305253
0.1	0.1	1	0.1	0.545447328305253
0.5	0.1	1	0.1	0.545447328305253
1	0.1	1	0.1	0.545447328305253
0.1	0.5	1	0.1	0.545447328305253
0.5	0.5	1	0.1	0.545447328305253
1	0.5	1	0.1	0.545447328305253
0.1	1	1	0.1	0.545447328305253
0.5	1	1	0.1	0.545447328305253
1	1	1	0.1	0.545447328305253
1	0.1	0.1	0.5	0.545447328305253
0.1	0.5	0.1	0.5	0.545447328305253
0.5	0.5	0.1	0.5	0.545447328305253
1	0.5	0.1	0.5	0.545447328305253
0.1	1	0.1	0.5	0.545447328305253
0.5	1	0.1	0.5	0.545447328305253
1	1	0.1	0.5	0.545447328305253
1	0.1	0.5	0.5	0.545447328305253
0.1	0.5	0.5	0.5	0.545447328305253
0.5	0.5	0.5	0.5	0.545447328305253
1	0.5	0.5	0.5	0.545447328305253
0.1	1	0.5	0.5	0.545447328305253
0.5	1	0.5	0.5	0.545447328305253
1	1	0.5	0.5	0.545447328305253
1	0.1	1	0.5	0.545447328305253
0.1	0.5	1	0.5	0.545447328305253
0.5	0.5	1	0.5	0.545447328305253
1	0.5	1	0.5	0.545447328305253
0.1	1	1	0.5	0.545447328305253
0.5	1	1	0.5	0.545447328305253
1	1	1	0.5	0.545447328305253
1	0.1	0.1	1	0.545447328305253
1	0.5	0.1	1	0.545447328305253
0.1	1	0.1	1	0.545447328305253
0.5	1	0.1	1	0.545447328305253
1	1	0.1	1	0.545447328305253
1	0.1	0.5	1	0.545447328305253
1	0.5	0.5	1	0.545447328305253
0.1	1	0.5	1	0.545447328305253
0.5	1	0.5	1	0.545447328305253
1	1	0.5	1	0.545447328305253
1	0.1	1	1	0.545447328305253
1	0.5	1	1	0.545447328305253
0.1	1	1	1	0.545447328305253
0.5	1	1	1	0.545447328305253
1	1	1	1	0.545447328305253

Table 6.3: Sub-Experiment 1 - Combinations that learned 54% of the model

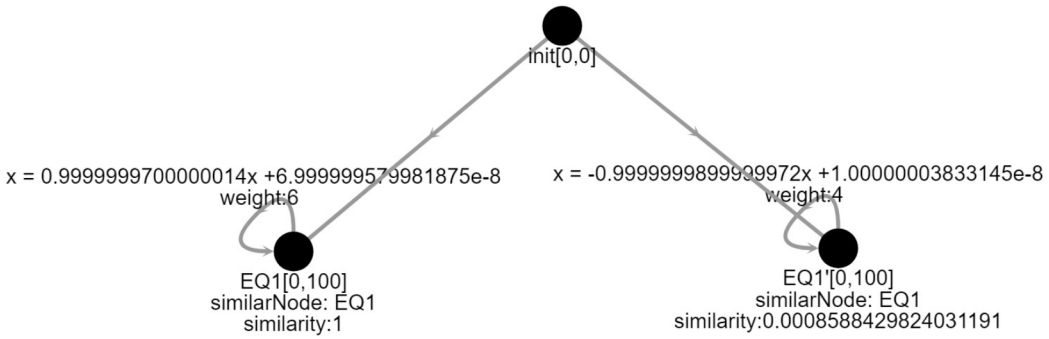

Figure 6.22: Sub-Experiment 2 - Branched Model

6.2.2 Sub-Experiment 2

This experiment was intended to briefly demonstrate how the incremental learning approach works with 2 simple models, under the same conditions as in Sub-Experiment 1. Meaning that we also learn all fo the upcoming models with the same 81 possible combinations of parameters. Due to the simplicity of the tested models, we first describe their structure in sub-section *Model Description* and then we discuss the results in the sub-section *Results*.

Model Description

The first model from Figure 6.22 is a simple branched model with only two *out-going* nodes coming from the same *parent* and no further nodes. While the second model from Figure 6.23 consists of three sequential nodes, where each node can only have one *parent*.

Results

We can see very positive results from both models in Table 6.4 and Table 6.5, as it was always the case that the original models were sucessfully learned. This happens because the nodes from both models are relatively far and can be easily identified by the implementation with the calculation of their normalized Euclidean distances (e.g. similarity value). For example, one can see in Figure 6.22 that the similarity value between *EQ1* and *EQ1'* is of approximately 0.00085. This value is always lower than the similarity threshold, as the last one can only be as low as 0.1. Meaning that the nodes from the previous models can never be merged, due to their difference in functionality. For this particular set of models, the only possibility to merge nodes would be to lower the similarity threshold from 0.1 to a value very close to 0. The only disadvantage of setting parameters like the *similarity threshold* to 0, is that it may cause the learning process to loose significant precision of the observations. It may also confuse the learning

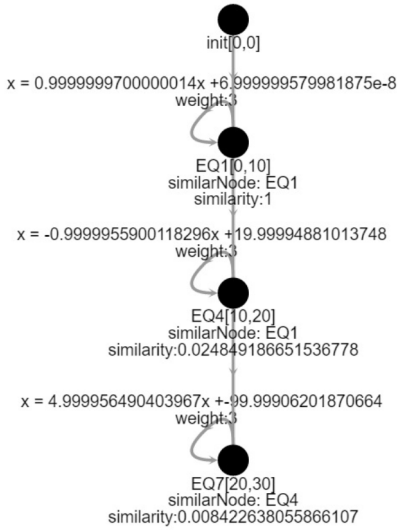

Figure 6.23: Sub-Experiment 2 - Sequential Model

process and indicate that very different nodes should be merged, when in reality they should be treated as different nodes.

Similarity Threshold	Time Cost	Functionality Cost	Location Cost	Graph Similarity
0.1	0.1	0.1	0.1	1
0.5	0.1	0.1	0.1	1
1	0.1	0.1	0.1	1
0.1	0.5	0.1	0.1	1
0.5	0.5	0.1	0.1	1
1	0.5	0.1	0.1	1
0.1	1	0.1	0.1	1
0.5	1	0.1	0.1	1
1	1	0.1	0.1	1
0.1	0.1	0.5	0.1	1
0.5	0.1	0.5	0.1	1
1	0.1	0.5	0.1	1
0.1	0.5	0.5	0.1	1
0.5	0.5	0.5	0.1	1
1	0.5	0.5	0.1	1
0.1	1	0.5	0.1	1
0.5	1	0.5	0.1	1
1	1	0.5	0.1	1
0.1	0.1	1	0.1	1
0.5	0.1	1	0.1	1
1	0.1	1	0.1	1
0.1	0.5	1	0.1	1
0.5	0.5	1	0.1	1
1	0.5	1	0.1	1
0.1	1	1	0.1	1
0.5	1	1	0.1	1
1	1	1	0.1	1
0.1	0.1	0.1	0.5	1
0.5	0.1	0.1	0.5	1
1	0.1	0.1	0.5	1
0.1	0.5	0.1	0.5	1
0.5	0.5	0.1	0.5	1
1	0.5	0.1	0.5	1
0.1	1	0.1	0.5	1
0.5	1	0.1	0.5	1
1	1	0.1	0.5	1
0.1	0.1	0.5	0.5	1
0.5	0.1	0.5	0.5	1
1	0.1	0.5	0.5	1
0.1	0.5	0.5	0.5	1
0.5	0.5	0.5	0.5	1
1	0.5	0.5	0.5	1
0.1	1	0.5	0.5	1
0.5	1	0.5	0.5	1
1	1	0.5	0.5	1
0.1	0.1	1	0.5	1
0.5	0.1	1	0.5	1
1	0.1	1	0.5	1
0.1	0.5	1	0.5	1
0.5	0.5	1	0.5	1
1	0.5	1	0.5	1
0.1	1	1	0.5	1
0.5	1	1	0.5	1
1	1	1	0.5	1
0.1	0.1	0.1	1	1
0.5	0.1	0.1	1	1
1	0.1	0.1	1	1
0.1	0.5	0.1	1	1
0.5	0.5	0.1	1	1
1	0.5	0.1	1	1
0.1	1	0.1	1	1
0.5	1	0.1	1	1
1	1	0.1	1	1
0.1	0.1	0.5	1	1
0.5	0.1	0.5	1	1
1	0.1	0.5	1	1
0.1	0.5	0.5	1	1
0.5	0.5	0.5	1	1
1	0.5	0.5	1	1
0.1	1	0.5	1	1
0.5	1	0.5	1	1
1	1	0.5	1	1
0.1	0.1	1	1	1
0.5	0.1	1	1	1
1	0.1	1	1	1
0.1	0.5	1	1	1
0.5	0.5	1	1	1
1	0.5	1	1	1
0.1	1	1	1	1
0.5	1	1	1	1
1	1	1	1	1

Table 6.4: Sub-Experiment 2 - Branched Model Results

Similarity Threshold	Time Cost	Functionality Cost	Location Cost	Graph Similarity
0.1	0.1	0.1	0.1	1
0.5	0.1	0.1	0.1	1
1	0.1	0.1	0.1	1
0.1	0.5	0.1	0.1	1
0.5	0.5	0.1	0.1	1
1	0.5	0.1	0.1	1
0.1	1	0.1	0.1	1
0.5	1	0.1	0.1	1
1	1	0.1	0.1	1
0.1	0.1	0.5	0.1	1
0.5	0.1	0.5	0.1	1
1	0.1	0.5	0.1	1
0.1	0.5	0.5	0.1	1
0.5	0.5	0.5	0.1	1
1	0.5	0.5	0.1	1
0.1	1	0.5	0.1	1
0.5	1	0.5	0.1	1
1	1	0.5	0.1	1
0.1	0.1	1	0.1	1
0.5	0.1	1	0.1	1
1	0.1	1	0.1	1
0.1	0.5	1	0.1	1
0.5	0.5	1	0.1	1
1	0.5	1	0.1	1
0.1	1	1	0.1	1
0.5	1	1	0.1	1
1	1	1	0.1	1
0.1	0.1	0.1	0.5	1
0.5	0.1	0.1	0.5	1
1	0.1	0.1	0.5	1
0.1	0.5	0.1	0.5	1
0.5	0.5	0.1	0.5	1
1	0.5	0.1	0.5	1
0.1	1	0.1	0.5	1
0.5	1	0.1	0.5	1
1	1	0.1	0.5	1
0.1	0.1	0.5	0.5	1
0.5	0.1	0.5	0.5	1
1	0.1	0.5	0.5	1
0.1	0.5	0.5	0.5	1
0.5	0.5	0.5	0.5	1
1	0.5	0.5	0.5	1
0.1	1	0.5	0.5	1
0.5	1	0.5	0.5	1
1	1	0.5	0.5	1
0.1	0.1	1	0.5	1
0.5	0.1	1	0.5	1
1	0.1	1	0.5	1
0.1	0.5	1	0.5	1
0.5	0.5	1	0.5	1
1	0.5	1	0.5	1
0.1	1	1	0.5	1
0.5	1	1	0.5	1
1	1	1	0.5	1
0.1	0.1	0.1	1	1
0.5	0.1	0.1	1	1
1	0.1	0.1	1	1
0.1	0.5	0.1	1	1
0.5	0.5	0.1	1	1
1	0.5	0.1	1	1
0.1	1	0.1	1	1
0.5	1	0.1	1	1
1	1	0.1	1	1
0.1	0.1	0.5	1	1
0.5	0.1	0.5	1	1
1	0.1	0.5	1	1
0.1	0.5	0.5	1	1
0.5	0.5	0.5	1	1
1	0.5	0.5	1	1
0.1	1	0.5	1	1
0.5	1	0.5	1	1
1	1	0.5	1	1
0.1	0.1	1	1	1
0.5	0.1	1	1	1
1	0.1	1	1	1
0.1	0.5	1	1	1
0.5	0.5	1	1	1
1	0.5	1	1	1
0.1	1	1	1	1
0.5	1	1	1	1
1	1	1	1	1

Table 6.5: Sub-Experiment 2 - Sequential Model Results

7 Conclusion and Future Work

Learning probabilistic timed automata was possible by measuring observations incrementally, with the use of Euclidean distances and cost functions. As demonstrated in the experiments, we can find some optimal combinations for which the incremental learning approach works properly. Unfortunately, the implementation and concept of this approach is not able to detect optimal combinations on the fly. Nevertheless, it would be possible to also adapt parameters of the cost function and the similarity threshold incrementally, as we also measure graph similarity simultaneously. Another remark to take into consideration, is that our approach is able to detect noisy observations. The only disadvantage is that it is impossible to track whether noisy observations truly come from a trace of an automaton, or if the implementation did not receive enough information from the model to learn appropriately, as there is no interaction with the observed automaton like in active learning. Meaning that we do not allow to check properties of the model and are not able to verify which type of traces the observed model is able to emit. Disregarding the lack of communication with the model to be learned and assuming that it provides accurate observations. We can recreate precise models by analyzing and measuring observations in an incremental fashion. Unfortunately, further experiments must be done to prove that we can extend this approach to more complex types of timed automata, like probabilistic timed automata with cyclic behaviors.

One of the future concepts that can be applied to extend this approach is *automata minimization*. An automaton is minimal, when there is no other automaton that accepts the same language with less locations. Working with a fixed number of locations would ensure having a canonical representation of automata, thus reducing complexity and ambiguities while learning and matching models. The incremental learning approach developed only considers the modification of a single close node based on an observation, a cost functions and a similarity threshold. We could extend the concept of incremental learning to create graph reduction routines which would mainly consist of evaluating the cost of eliminating nodes by merging their functionality and time constraints with other similar nodes based on how close they are (e.g. By using different similarity threshold values). As we have seen, replacement of nodes involves the possible propagation of errors. Being that said, we would need to consider two main types of graph reduction routines: *sequential merge* and *parallel merge*. A sequential merge would be done by merging sequential nodes. Specifically, pairs of nodes that do not belong to the same *depth* of the graph, or when one node is *parent* of the other. Whereas in parallel merge, the merge of nodes would be in the same depth of the graph, meaning that nodes might not be directly connected, consequently causing different propagation effects as in a sequential merge. The way that this approach would work, is by performing the reduction routines whenever there exists more locations in the learned model than in the original one. As we can already identify noisy observations and refer to the number of nodes of the original model, this extension is feasible and applicable to the implementation of this project.

Bibliography

[1] Rajeev Alur. Timed automata. In *International Conference on Computer Aided Verification*, pages 8–22. Springer, 1999.

[2] Endika Bengoetxea. *The graph matching problem.* PhD thesis, PhD Thesis, 2002.

[3] Endika Bengoetxea, Pedro Larranaga, Isabelle Bloch, Aymeric Perchant, and Claudia Boeres. Inexact graph matching by means of estimation of distribution algorithms. *Pattern Recognition*, 35(12):2867–2880, 2002.

[4] Jean Berstel, Luc Boasson, Olivier Carton, and Isabelle Fagnot. Minimization of automata. *arXiv preprint arXiv:1010.5318*, 2010.

[5] Vincenzo Carletti. *Exact and Inexact Methods for Graph Similarity in Structural Pattern Recognition PhD thesis of Vincenzo Carletti.* PhD thesis, Université de Caen; Universita degli studi di Salerno, 2016.

[6] Vincenzo Carletti. *Exact and Inexact Methods for Graph Similarity in Structural Pattern Recognition PhD thesis of Vincenzo Carletti.* PhD thesis, Université de Caen; Universita degli studi di Salerno, 2016.

[7] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[8] Jan Daciuk. *Optimization of automata.* Gdańsk University of Technology Publishing House, 2014.

[9] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[10] Javier Esparza. Automata theory: An algorithmic approach (lecture notes), 2012.

[11] Scott Fortin. The graph isomorphism problem. 1996.

[12] Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. S-match: an algorithm and an implementation of semantic matching. In *European semantic web symposium*, pages 61–75. Springer, 2004.

[13] Falk Howar and Bernhard Steffen. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 123–148. Springer, 2018.

[14] Marta Kwiatkowska, Gethin Norman, David Parker, and Jeremy Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29(1):33–78, 2006.

[15] Marta Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282(1):101–150, 2002.

[16] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[17] Viktor Losing, Barbara Hammer, and Heiko Wersing. Incremental on-line learning: A review and comparison of state of the art algorithms. *Neurocomputing*, 275:1261–1274, 2018.

[18] Alexander Maier. Online passive learning of timed automata for cyber-physical production systems. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 60–66. IEEE, 2014.

[19] Jan Ramon, Maurice Bruynooghe, and Wim Van Laer. Distance measures between atoms. In *CompulogNet Area Meeting on Computational Logic and Machine Learning*, pages 35–41, 1998.

[20] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 256–296. Springer, 2011.

[21] Laura A Zager and George C Verghese. Graph similarity scoring and matching. *Applied mathematics letters*.