Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

# Timeline Probabilities

A Final Year Project

written by

**Mohamed Suliman**

under the supervision of **Dr. Arthur White**, and submitted to the
School of Computer Science and Statistics
in partial fulfillment of the requirements for the degree of

BA (Mod) Computer Science

at *Trinity College Dublin, The University of Dublin.*

## Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Mohamed Suliman, May 3, 2021

## Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

Mohamed Suliman, May 3, 2021

## Acknowledgements

I would firstly like to thank my supervisor, Dr. Arthur White, for his continued guidance and support throughout the duration of this project. I would also like to extend my gratitude to Dr. Tim Fernando and Dr. Carl Vogel for their time and their help. I am deeply indebted to them for their insights and expertise in this area. Thanks are also due to my parents for supporting me during my undergraduate career, and to my friends for a memorable four years.

**Abstract**

The 13 temporal relations on time intervals developed by Allen, and the finite temporality approach to temporal knowledge representation and reasoning are used to build an efficient temporal reasoning framework. The value of this framework lies in its universality, any sequence of events and their relationships in time can be encoded and manipulated to uncover new insights about the nature of events and their interdependencies.

Timelines of events are reimagined as stochastic processes, and probabilisitic models are developed to simulate timeline generation. How often timelines are generated by these models point to their probability, the question of how likely is a timeline of events is considered with it being found that one may be much more or less likely than another. A thorough discussion of the model parameters is given, as well as an application of the methods developed on real world events and timelines, taken from the TimeBank corpus of news articles, where temporal information in the texts is annotated.

# Contents


**1 Introduction** **6**
1.1 Contributions . . . . . 7

**2 A Review of Finite Temporality** **9**
2.1 The Foundations of Finite Temporality Strings . . . . . 9
2.1.1 Allen Relations and their Prior Probabilities . . . . . 12
2.1.2 Probabilities Over Interval Names . . . . . 15
2.2 Finite Temporality Strings and their Operations . . . . . 17
2.2.1 Vocabulary Constrained Superposition . . . . . 19
2.3 Conclusion and Discussion of Further Research . . . . . 21

**3 Temporal Reasoning in Python** **22**
3.1 Representing Finite Temporality Strings . . . . . 22
3.1.1 Set Isolation . . . . . 25
3.1.2 Block Compression and its Inverse . . . . . 26
3.1.3 Relation Resolution & Well Formedness . . . . . 28
3.2 Vocabulary Constrained Superposition . . . . . 30
3.3 Performance and Evaluation . . . . . 35
3.4 Calculating Allen Relation Probabilities . . . . . 36
3.5 Conclusion . . . . . 36

**4 Timeline Generation and Likelihood** **38**
4.1 Stochastic Superposition . . . . . 38
4.1.1 Stochastically Generating Well Formed Strings . . . . . 39
4.2 The Unborn Living Dead Construction . . . . . 41
4.2.1 Selection of Transition Probabilities . . . . . 45
4.3 Applications to the TimeBank Corpus . . . . . 49
4.4 Conclusion . . . . . 50

**5 Final Conclusions** **51**
5.1 Applications beyond TimeML . . . . . 51

**A Calculating Probabilities over Interval Names** **52**

**B Approximating $A$-state Transition Probabilities** **53**

# 1 Introduction

This work initially began as a research project, undertaken in the summer preceding the current academic year, to look at the temporal relations that hold in texts taken from the TimeBank corpus of news articles [6]. What resulted was a program that would extract *finite temporality strings*, a method of temporal knowledge representation, and a broad, superficial analysis of the temporal relations that tended to hold between events that occured in the texts. The goal of this thesis, to be considered as a continuation, is to provide a more efficient implementation of the program used to extract finite temporality strings, having been entirely rewritten, as well as a more in depth examination of the *probabilities* of timelines. We look at these finite temporality strings that represent timelines of events and develop a method for assigning probabilities to a way in which events may play out in time.

The structure of this thesis deviates from that which is typical. The relationship between sections is not a linear one, where a review of existing work is followed by a methodology, implementation, and evaluation. Instead each section is related to the general topic of this thesis, being the temporal knowledge representation method that is a *finite temporality string*, and is a self contained piece of writing, however each section makes its own contribution to the idea of timeline probabilities, setting up the next section to continue on.

The three principal sections include a review of the *finite temporality* approach to temporal knowledge representation and reasoning, a detailed explanation of the implementation of an efficient library to represent and manipulate finite temporality strings, and finally the introduction of a novel method of calculating the probability of a timeline of events. Each section is written in a way that facilitates it to be read individually, however reading in the order written attempts to point to the overall goal of the project. It begins, familiarly, with an expositionary literature review, serving as the introduction to the main concepts of finite temporality, these concepts are then implemented in a computationally efficient manner through the use of *generators*. This so far sticks to structure generally accepted. The final section develops a novel method to assign a probability to a finite temporality string, a means of answering the question of how likely a particular *playing out* of events is to happen. The departure from the orthodox thesis outline happens here, where the focus is put on the probabilistic nature of timelines, and how best to simulate them. While the framework that is developed in the antecedent section is used to simulate timelines, it does not necessarily follow, instead it is an offshoot of the topic that is explored, and has yielded interesting results.

The question of timeline probabilities is a strange one, as one would intuitively think that one particular timeline of events is just as likely to occur as

another. A deeper investigation reveals that this is not necessarily the case, and depending on the context, a set of events are much more likely to play out in one way than another. This question is the general version of the question answered in [3], which looks at prior probabilities of individual Allen relations between events. The peculiarity of this question begs further study, and being the main motivation for this thesis, it is a topic upon which we shed some light.

## 1.1 Contributions

This thesis makes the following contributions to temporal knowledge representation and reasoning:

- An efficient temporal reasoning framework in `python` for the creation and manipulation finite temporality strings.
- An extension to the work of Fernando and Vogel [3], which answers the question of probabilities of Allen relations, providing a method to answer the more general question of *what is the probability of a timeline?*

*The clock tick tocks and...*
*Things change, time passes, death happens.*

– Tanya Davis, *Eulogy for You and Me*

# 2 A Review of Finite Temporality

Living things are, to varying degrees, aware of the concept of time. We humans ourselves, are equipped with a deep appreciation of time and its passing, often motivated perhaps by our hyper-awareness of mortality. We can reason temporally and understand the relationships between events as they happen in time. However, the wildebeasts, gazelles, and zebras that roam the plains of Africa, for example, may not necessarily embody such a high regard for the progression of time and the interplay of events within it, yet they still understand that every year, they must participate in the Great Migration, that takes them from the Serengeti of Tanzania to the Masai Mara in Kenya.

Focusing on humans, Brackbill and Fitzgerald [2] have shown that around the age of one month, toddlers have already developed, what we label as, a *sense of time*. By placing infants in a room where a light is periodically turned on and off, Brackbill and Fitzgerald found that when they stopped switching the light on and off, the pupils of the children would still dilate or constrict in anticipation of the change. What is clear is that infants possess, at such a young age, a two-fold understanding of time. Firstly, they show an appreciation of time duration. Secondly, and more importantly, they have an understanding of *intervals* of time, during which they understand that some property of the world around them holds true, and outside of which, it does not.

By representing time as a *collection* of time intervals, *change* occurs when a new event appears or an old one disappears in the next interval *e.g.* the light being on and off. This idea is at the core of the *finite temporality* approach to temporal reasoning. A *collection* of intervals is known as a *finite temporality string*, representing a *timeline* of events, and whose development is subsequently reviewed in detail. In viewing time this way, the question of when do events appear and disappear is naturally raised. We review work that has begun answering this question, and suggest a novel, finite-state method, that attempts to provide an answer by framing the problem differently.

## 2.1 The Foundations of Finite Temporality Strings

The interval-based temporal logic introduced by Allen [1] serves as the foundation for the temporal representation and reasoning methods described in this review.

When we think of the time during which an event occurs, we may intuitively think of it as instantaneous. Taking the example of an event given by Allen to be "we found the letter at twelve noon", we may interpret it as though the letter was found at that midday instance. However, upon closer inspection, this may be broken down further into "looking at the spot where the letter was" and

"realizing it was that singular letter". These sub-events may be decomposed even further into their constituent parts so as to go right down to the sequence of neurons that fire in the observer's brain that lead to the realization that it is in fact *the* letter. Even still this may be broken down further. Allen summarizes this succinctly, saying that

> There seems to be a strong intuition that, given an event, we can always "turn up the magnification" and look at its structure. [1, p. 834]

Thus the notion of time points should not be considered primitive, or the *building block* as it were, but instead intervals of time. Consider a model where we have a fully ordered set of time points, an interval within this set is an ordered pair of points where the first point is less than the second. An important question then is raised: Are these intervals open or closed? The open interval does not include its endpoints whereas the closed interval does. Allen suggests that these intervals should instead be open on the lower end and closed on their upper end. If we try to model the change that occurs say, when a light is switched on, we need intervals to represent times when the light is off and times when it is on. Open intervals imply that there exists a time where the light is neither on nor off, and closed intervals suggest that the light can be both on and off at the same time. The solution of a half-open interval at its upper end allows for both intervals to meet as required and having only one endpoint.

With this framework in mind, Allen goes on to define several relations between intervals. By letting $t$ denote some interval, with its lower point and upper point being $t-$ and $t+$ respectively, then Table 1, taken from [1], gives these relations in terms of their equivalent relations on endpoints.

Table 1: Interval relations defined on endpoints.

| Interval Relation | Equivalent Relations on Endpoints |
|---|---|
| $t$ before $s$ | $t+ < s-$ |
| $t = s$ | $(t- = s-)$ & $(t+ = s+)$ |
| $t$ overlaps $s$ | $(t- < s-)$ & $(t+ > s-)$ & $(t+ < s+)$ |
| $t$ meets $s$ | $t+ = s-$ |
| $t$ during $s$ | $(t- > s-)$ & $(t+ \leq s+)$) or $((t- \geq s-)$ & $(t+ < s+))$ |

These five relations may be broken down further to give the 13 Allen relations on temporal intervals. We break down the *during* relation into *during*, *starts*, and *finishes*, and include the inverses of these relations. Table 2 provides a summary, as well as giving pictorial representations, which are to be read from left to right, and constitute a timeline.

Table 2: The 13 Allen relations.

| Relation | Symbol | Pictoral Representation |
|---|---|---|
| $X$ before $Y$ | $X$ **b** $Y$ | $XXX\ YYY$ |
| $X$ after $Y$ | $X$ **bi** $Y$ | $YYY\ XXX$ |
| $X$ during $Y$ | $X$ **d** $Y$ | $\ \ XXX$<br>$YYYYYY$ |
| $X$ contains $Y$ | $X$ **di** $Y$ | $\ \ YYY$<br>$XXXXXX$ |
| $X$ overlaps $Y$ | $X$ **o** $Y$ | $XXXX$<br>$\ YYYYY$ |
| $X$ overlapped-by $Y$ | $X$ **oi** $Y$ | $YYYYY$<br>$\ XXXXX$ |
| $X$ meets $Y$ | $X$ **m** $Y$ | $XXXYYY$ |
| $X$ met-by $Y$ | $X$ **mi** $Y$ | $YYYXXX$ |
| $X$ starts $Y$ | $X$ **s** $Y$ | $XXX$<br>$YYYYYY$ |
| $X$ started-by $Y$ | $X$ **si** $Y$ | $YYY$<br>$XXXXX$ |
| $X$ finishes $Y$ | $X$ **f** $Y$ | $\ \ XXX$<br>$YYYYY$ |
| $X$ finished-by $Y$ | $X$ **fi** $Y$ | $\ \ YYY$<br>$XXXX$ |
| $X$ equal $Y$ | $X$ **e** $Y$ | $XXX$<br>$YYY$ |

### 2.1.1 Allen Relations and their Prior Probabilities

Taking the scenario where we have that interval $a$ is related to interval $b$ via some relation $R$, and $b$ is related to interval $c$ via another relation $R'$, then how may we reason about the temporal relation between $a$ and $c$? For example, if $a$ is before $b$ and $b$ is before $c$, then we can say with certainty that $a$ must come before $c$. Allen's transitivity table, given in Figure 4 in [1] shows the relations that may hold between intervals $a$ and $c$ given that $a$ and $b$ are related via $R$ and $b$ and $c$ are related via $R'$, for all combinations of $R$ and $R'$. Allen also provides a reasoning algorithm based on constraint propagation as a system for temporal reasoning about events that are not directly related, however this is not reviewed here.

When $a$ is before $b$ and $b$ is after $c$, then we cannot deduce anything about the relation between $a$ and $c$. The context of $b$ does not help here. The intervals $a$ and $c$ could be related by any one of the 13 Allen relations while still respecting the fact that $a$ is before $b$ and $b$ is after $c$.

Can we really say nothing further about the two intervals $a$ and $c$, or in fact any two intervals of time of whose relation we know nothing about? This is the motivation behind Fernando and Vogel [3]. They proceed from the following question:

> Given an Allen relation $R$, what is the probability that $R$ relates intervals $a$ and $a'$, $aRa'$?

This is the same question posed in the introduction, about when events appear and dissapear, albeit reworded. The principle of indifference states that this probability should be 1/13. The fact that some relations occur more often than others in Allen's transitivity table suggest a deeper structure, and that these relations are not simply a matter of indifference. It is this underlying structure that is elucidated in [3].

Following the notation in [3], we define the set $\mathcal{AR}$ to be the set of the 13 names of Allen relations, so we have that

$$\mathcal{AR} = \{\text{b, bi, d, di, o, oi, m, mi, s, si, f, fi, e}\}.$$

Next consider the finite order over $n$ points, $[n]$, to be

$$[n] := \{i \in \mathbb{Z} \mid 1 \leq i \leq n\}.$$

Then we may define an interval $(l, r] := \{i \in [n] \mid l < i \leq r\}$. The question of what is the probability of $aRa'$ becomes the probability that $(l, r]\ R\ (l', r']$. Given that $n \geq 4$, we have that the total number of pairs $l, r$ and $l', r'$ is $\binom{n}{2} \cdot \binom{n}{2}$.

Let each of these possible pairs be hereafter referred to as equiprobable *n-worlds*, and let $\Omega_n$ be the set of these n-worlds. The cardinality of $\Omega_n$ is given by

$$\text{cardinality}(\Omega_n) = \binom{n}{2} \cdot \binom{n}{2}.$$

Similarly, let an *n-world*, given $n \geq 4$, be a function

$$f : \{x, y, x', y'\} \to [n],$$

that maps 4 distinct variables $x, y, x', y'$ to integers in $[n]$ such that

$$f(x) < f(y) \text{ and } f(x') < f(y').$$

We have that

$$(f(x), f(y)] \ b \ (f(x'), f(y')]$$

if $f(x) < f(y) < f(x') < f(y')$ *i.e* this n-world satisfies the *before* relation. Fernando and Vogel go on to say that the probability of $aRa'$ can be written as the proportion of n-worlds in $\Omega_n$ that satisfy the relation $R \in \mathcal{AR}$,

$$p_n(R) := \frac{\text{cardinality}(\{f \in \Omega_n \mid f \text{ satisfies } R\})}{\text{cardinality}(\Omega_n)}. \tag{1}$$

We find that the 13 Allen relations may be categorized into those that are *short*, *medium* and *long*. To understand the reasoning behind this, we may look at the representation of each relation as a string. Table 3 does exactly that, a reproduction of Table 1 in [3].

Table 3: Allen relations and their string representation.

| $(l, r]$ $R$ $(l', r']$ | $\mathfrak{s}_R$ | $R^{-1}$ | $\mathfrak{s}_{R^{-1}}$ |
|---|---|---|---|
| $(l, r]$ b $(l', r']$ | $l \mid r \mid l' \mid r'$ | bi | $l' \mid r' \mid l \mid r$ |
| $(l, r]$ d $(l', r']$ | $l' \mid l \mid r \mid r'$ | di | $l \mid l' \mid r' \mid r$ |
| $(l, r]$ o $(l', r']$ | $l \mid l' \mid r \mid r'$ | oi | $l' \mid l \mid r' \mid r$ |
| $(l, r]$ m $(l', r']$ | $l \mid r, l' \mid r'$ | mi | $l' \mid r', l \mid r$ |
| $(l, r]$ s $(l', r']$ | $l, l' \mid r \mid r'$ | si | $l, l' \mid r' \mid r$ |
| $(l, r]$ f $(l', r']$ | $l' \mid l \mid r, r'$ | fi | $l \mid l' \mid r, r'$ |
| $(l, r]$ e $(l', r']$ | $\boxed{l, l' \mid r, r'}$ | e | |

We see that for example $\boxed{l, l' \mid r, r'}$ represents an n-world where $l = l' < r = r'$, which is the *equals* relation. Looking at Table 3, we see that the first 6

strings are of length 4, the next 6 are of length 3, and the final relation string, *equals*, has a length of 2. Thus these are the 3 categories of relations. The *long* relations are defined as

$$\{R \in \mathcal{AR} \mid \text{length}(\mathfrak{s}_R) = 4\} = \{\text{b, bi, d, di, o, oi}\},$$

the medium relations are

$$\{R \in \mathcal{AR} \mid \text{length}(\mathfrak{s}_R) = 3\} = \{\text{m, mi, s, si, f, fi}\},$$

and finally the short relations, which are

$$\{R \in \mathcal{AR} \mid \text{length}(\mathfrak{s}_R) = 2\} = \{\text{e}\}.$$

In order to calculate $p_n(R)$, we need to know the number of n-worlds that satisfy $R$, given $n \geq 4$.

Fix $n = 4$, then we have $[4] = \{1, 2, 3, 4\}$. For the *equals* relation, there are 6 *4-worlds* that satisfy this relation, namely

$$\begin{gathered}\{(x,1),(x',1),(y,2),(y',2)\}\\ \{(x,1),(x',1),(y,3),(y',3)\}\\ \{(x,1),(x',1),(y,4),(y',4)\}\\ \{(x,2),(x',2),(y,3),(y',3)\}\\ \{(x,2),(x',2),(y,4),(y',4)\}\\ \{(x,3),(x',3),(y,4),(y',4)\}\end{gathered}$$

Fernando and Vogel show that the total number of *n-worlds* satisfying a given relation depends on whether the relation is *long*, *medium*, or *short*. For *long* relations, that number is $\binom{n}{4}$, for *medium* relations, it is $\binom{n}{3}$, and for *short*, there are $\binom{n}{2}$ *n-worlds* that satisfy. Representing the medium relations by $m$, the long relations by $b$, and the short relation by $e$ we have that

$$\frac{p_n(m)}{p_n(e)} = \frac{n-2}{3} \text{ and } \frac{p_n(b)}{p_n(m)} = \frac{n-3}{4},$$

which with

$$1 = \sum_{R \in \mathcal{AR}} p_n(R) = p_n(e) + 6p_n(m) + 6p_n(b),$$

leads to Theorem 2 from [3] which states that for $n \geq 4$ and $R, R' \in \mathcal{AR}$,

$$p_n(R) = p_n(R') \text{ if length}(\mathfrak{s}_R) = \text{ length}(\mathfrak{s}_{R'}),$$

where

$$p_n(e) = \frac{2}{n(n-1)},$$

$$p_n(m) = \frac{2(n-2)}{3n(n-1)},$$

and

$$p_n(b) = \frac{(n-3)(n-2)}{6n(n-1)}.$$

Table 4 gives these probabilities as we increase $n$. In fact we have that as $n \to \infty$, $p_n(R) = 0$ if $R$ is *short* or *medium* and $p_n(R) = \frac{1}{6}$ otherwise, as shown in [3].

Table 4: Some probabilities from Theorem 2.

| $n$ | $p_n(e)$ | $p_n(m)$ | $p_n(b)$ |
|---|---|---|---|
| 4 | 1/6 | 1/9 | 1/36 |
| 5 | 1/10 | 1/10 | 1/20 |
| 6 | 1/15 | 4/45 | 1/15 |
| 8 | 1/28 | 1/14 | 5/56 |

### 2.1.2 Probabilities Over Interval Names

In the previous section, we had summarized Fernando and Vogel's approach to assigning probabilities to Allen relations by treating time as a set of $n$ linearly ordered points. They continue to consider these probabilities by instead construing each $i \in [n]$ to be an interval name, instead of a point, and using the strings given in Table 3 to represent them. Take the following string as an example

| 1,2,4 | 1 | 2,3 | 3 | 4 |
|---|---|---|---|---|

This is a representation of a possible timeline of 4 intervals. We can see that from this string, 2 *overlaps* 3, which would be the string

| 2 | 2,3 | 3 |
|---|---|---|

Fernando and Vogel define two operations on these strings, namely the *X-reduct* $\rho_X(s)$, and the *X-projection* $\pi_X(s)$. The *X-reduct* of a string $s = a_1...a_k$ is given by

$$\rho_X(s) := (a_1 \cap X)...(a_k \cap X).$$

Being the componentwise intersection, the *X-reduct* allows us to narrow our view of the string to those intervals that we are interested in *i.e* those intervals $i \in X$. For example,

$$\rho_{\{2,3\}}(\boxed{1,2,4}\boxed{1}\boxed{2,3}\boxed{3}\boxed{4}) = \boxed{2}\boxed{\ }\boxed{2,3}\boxed{3}\boxed{\ }.$$

Removing the empty boxes, we get the string $\mathfrak{s}_{R/2,3}$ for *overlaps* relation. Note that $\mathfrak{s}_{R/i,j}$ represents the string $\mathfrak{s}_R$ with $l, r$ and $l', r'$ replaced with $i$ and $j$ for $i, j \in [n]$. The operation $\pi_X(s)$ applies the *X-reduct* and removes the empty boxes that occur

$$\pi_{\{2,3\}}(\boxed{1,2,4}\boxed{1}\boxed{2,3}\boxed{3}\boxed{4}) = \boxed{2}\boxed{2,3}\boxed{3}.$$

Let $\mathcal{L}_n$ be the set of strings of non-empty subsets of $[n]$ where each $i \in [n]$ occurs exactly twice.

$$\mathcal{L}_n := \{s \in (2^{[n]} - \{\boxed{\ }\})^+ \mid (\forall i \in [n])\ \pi_i(s) = \boxed{\mathrm{i}}\boxed{\mathrm{i}}\}$$

The string given in the example demonstrating the $X$-projection above would be a member of the set $\mathcal{L}_4$. Fernando and Vogel then go on to say that $p_n(R)$, the proportion of $\mathcal{L}_n$ in which interval 1 is $R$-related to interval 2 is

$$p_n(R) := \frac{\text{cardinality}(\mathcal{L}_n(R))}{\text{cardinality}(\mathcal{L}_n)} \tag{2}$$

where $\mathcal{L}_n(R)$ is the subset

$$\{s \in \mathcal{L}_n \mid s \models 1R2\},$$

of strings whose *{1,2}*-projection corresponds to the string $\mathfrak{s}_{R/1,2}$. Not included here is the procedure given in [3] that is used to calculated the cardinality of $\mathcal{L}_n(R)$ for brevity's sake, however probabilities for the three classes of Allen relations are given in a similar manner to that described previously for $n$ points rather than intervals. Table 3 in [3] gives these probabilities for several values of $n$. We include here in Table 5 the entry for $n = 2$ intervals.

Table 5: The entry for $n = 2$ intervals from Table 3 in Fernando and Vogel.

| n | $p_n(e)$ | $p_n(m)$ | $p_n(b)$ |
|---|---|---|---|
| 2 | 1/13 | 1/13 | 1/13 |
| 3 | 0.031785 | 0.061125 | 0.10024 |

## 2.2 Finite Temporality Strings and their Operations

We have, in prior sections, looked at strings where the endpoints of the intervals within them are given. Taking the example string given previously

| 1,2,4 | 1 | 2,3 | 3 | 4 |,

instead of marking the endpoints of each interval, we may use a *stative* predicate to represent events, and their inclusion in a given interval determines whether they are happening in that interval or not. Let the set $A$ be the set of event names, or *fluents*, as given in Fernando et al. [9], then we have the string $s = \alpha_1...\alpha_n$ where each $\alpha_i$ is a subset of $A$. We read these strings chronologically from left to right with it being undertstood that at position $i$, $1 \leq i \leq n$, every fluent $a \in \alpha_i$ holds. Thus we may translate the above string into the following *finite temporality string* , where where the four interval names that represent the intervals during which some event is occuring are replaced with predicates. We have that $A = \{a, b, c, d\}$ *i.e* we convert the intervals into fluents.

| | $a, b, d$ | $b, c, d$ | $c, d$ | $d$ | |

The empty boxes on either end are for times before these intervals and times after *i.e* times where none of the events $a \in A$ are occuring. These events occur for a finite amount of time, and the empty boxes are there to show that there is indeed a defined start and end for each event in the string. Table 6 gives the 13 Allen relations over the fluents $a$ and $b$ in terms of these finite temporality strings.

Several operations on finite temporality strings are defined in [9], with the most basic of which being the superposition. Given two strings $s$ and $s'$, $s$ & $s'$ is the componentwise union

$$\alpha_1...\alpha_n \ \& \ \alpha'_1...\alpha'_n := (\alpha_1 \cup \alpha'_1)...(\alpha_n \cup \alpha'_n)$$

of each $\alpha_i$ and $\alpha'_i$. For example, we have that

| $a$ | $b$ | & | $c$ | $d$ | = | $a, c$ | $b, d$ |.

Table 6: The 13 Allen relations written as finite temporality strings.

| Relation | Finite Temporality String |
|---|---|
| equal (e) | $\square \mid a, b \mid \square$ |
| before (b) | $\square \mid a \mid \square \mid b \mid \square$ |
| after (bi) | $\square \mid b \mid \square \mid a \mid \square$ |
| during (d) | $\square \mid b \mid a, b \mid b \mid \square$ |
| contains (di) | $\square \mid a \mid a, b \mid a \mid \square$ |
| overlaps (o) | $\square \mid a \mid a, b \mid b \mid \square$ |
| overlapped-by (oi) | $\square \mid b \mid a, b \mid a \mid \square$ |
| meets (m) | $\square \mid a \mid b \mid \square$ |
| met-by (mi) | $\square \mid b \mid a \mid \square$ |
| starts (s) | $\square \mid a, b \mid b \mid \square$ |
| started-by (si) | $\square \mid a, b \mid a \mid \square$ |
| finishes (f) | $\square \mid b \mid a, b \mid \square$ |
| finished-by (fi) | $\square \mid a \mid a, b \mid \square$ |

Note that both strings need to be the same length. The *block compression* operation, and its inverse allow for the length of a string to be changed without altering the temporal information that is encoded. The string $\boxed{\ }\boxed{a}\boxed{a}\boxed{a}\boxed{\ }\boxed{b}\boxed{b}\boxed{\ }$ is temporally equivalent to the string $\boxed{\ }\boxed{a}\boxed{\ }\boxed{b}\boxed{\ }$. The repetition, or *stutter* in these strings does not add any extra information. The duration of each interval is not of importance and is not considered in these strings. What these strings try to encode is the progression of events, therefore if two consecutive subsets $\alpha_i,\ \alpha_{i+1}$ are equal then we can remove one of them, since no *change* happens, coinciding with the Aristotelian view that there is "no time without change". Strings that contain stutter can be made stutter-less by applying the *block compression*, defined as

$$bc(s) := \begin{cases} s & \text{if } length(s) \leq 1 \\ bc(\alpha s') & \text{if } s = \alpha\alpha s' \\ \alpha bc(\alpha' s') & \text{if } s = \alpha\alpha' s' \text{with } \alpha \neq \alpha' \end{cases}$$

The inverse, $bc^{-1}$ generates the infinite language of strings that are said to be *bc-equivalent* to $s$. The ability to change the length of these strings means that we may apply the superposition operation in order to combine two or more strings into one. *Asynchronous superposition* is a method developed in [9] and was further optimized in [8] to become, *vocabulary constrained superposition.*

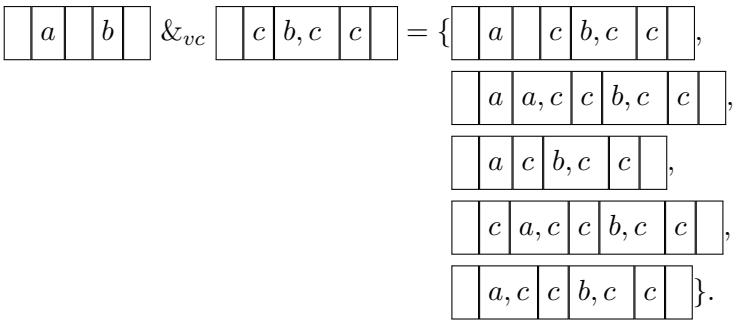
### 2.2.1 Vocabulary Constrained Superposition

Vocabulary constrained superposition is an operation on finite temporality strings that generates temporally consistent timelines. If we let $s_1 = \boxed{\ }\boxed{a}\boxed{\ }$ and $s_2 = \boxed{\ }\boxed{b}\boxed{\ }$, then $s_1\ \&_{vc}\ s_2$ is the set of the 13 strings that correspond to the 13 Allen relations, as given in Table 6. What we are saying here is that we know nothing about intervals $a$ and $b$, only that they are finite. The vocabulary constrained superposition then should result in a set of strings where what we know already about the intervals is maintained. Since we know nothing about $a$ and $b$, all 13 relations could be possible timelines of these events.

Now if we introduce another interval $c$, and we know already that $a$ *before* $b$ and $b$ *before* $c$, we can incorporate these constraints by superposing the strings that represent these relations. Continuing with this example, we have that

$$\boxed{\ }\boxed{a}\boxed{\ }\boxed{b}\boxed{\ }\ \&_{vc}\ \boxed{\ }\boxed{b}\boxed{\ }\boxed{c}\boxed{\ } = \boxed{\ }\boxed{a}\boxed{\ }\boxed{b}\boxed{\ }\boxed{c}\boxed{\ },$$

or if instead we know that $b$ *during* $c$, then we have




By applying block compression to the $\{a, c\}$-reduct of the 5 possible timelines above, we can see that intervals $a$ and $c$ could be related via the relations $b$, $o$, $m$, $d$, and $s$, which corresponds to that entry in Allen's transitivity table for $a\ R\ b$ and $b\ R'\ c$, with $R = b$ and $R' = d$.

Let $voc(s)$ be the union of all subsets $\alpha_i \in s$, namely

$$voc(s) := \bigcup_{i=1}^{n} \alpha_i,$$

where $n$ is the length of the string $s$. The vocabulary constrained superposition of two strings $s$ and $s'$ is defined as

$$s\ \&_{vc}\ s' = s\ \&_{voc(s),voc(s')}\ s'.$$

Let the string $s_1 = \alpha_1 ... \alpha_n$ be rewritten as $s_1 = \alpha s$, where $\alpha$ is the first component $\alpha_1$ of the string $s_1$, and $s = \alpha_2 ... \alpha_n$. Similarly, let $s_2 = \alpha' s'$. If we have two sets $\Sigma$ and $\Sigma'$, then

$$s_1\ \&_{\Sigma,\Sigma'}\ s_2 = \begin{cases} \{(\alpha \cup \alpha')s'' | s'' \in L\}, \text{ if } \Sigma \cap \alpha' \subseteq \alpha \text{ and } \Sigma' \cap \alpha \subseteq \alpha' \\ \varnothing, \text{ otherwise} \end{cases} \tag{3}$$

where

$$L = (\alpha s\ \&_{\Sigma,\Sigma'}\ s') \cup (s\ \&_{\Sigma,\Sigma'}\ \alpha' s') \cup (s\ \&_{\Sigma,\Sigma'}\ s').$$

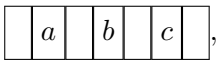
Vocabulary constrained superposition is both commutative and associative [8], which allows us to superpose however many strings together in any order to generate timelines of the events in these strings. If we are interested in the relation between some particular events,

$$bc(\rho_X(s))$$

can be applied to any string that is generated from the superposition to see the relation between the events $a \in X$, and, as Allen puts it, "turn up the magnification".

## 2.3 Conclusion and Discussion of Further Research

Allen's 13 relations are the core of the finite temporality approach to temporal knowledge representation and reasoning. Finite temporality strings allow us to encode temporal information about events and their relationships, and vocabulary constrained superposition lets us reason about events in these strings by combining them into timelines. It is important to consider that the number of possible timelines that are generated from the vocabulary constrained superposition increase exponentially as we add more events. We have seen already that there are 13 possible timelines for 2 events, the 13 Allen relations. With 3 events, the number of timelines is 409, with 4 events we have 23,917, and with 5 events, the number is slightly over 2.2 million. What [9] labels as the "combinatorial explosion" is huge, and by introducing what we may already know about the events into the strings, we are able to reduce the number of possible timelines. For example, the 409 possible timelines of the events $a,\ b,$ and $c$ are reduced to just the one timeline




if we know already that $a\ before\ b$ and $b\ before\ c$. In general, the more information we know about the temporal relationships that hold between intervals, the more we can narrow down the set of their possible timelines.

The probabilities of Allen relations developed in [3], while different when considering time in points or intervals, point to the underlying structure of the relations themselves. Their results are the result of an argument from first principles, and further work may build upon this by coming to probabilities through more empirical means such as simulation, by looking at the divergences that different timelines take after each interval and seeing how often these divergences are taken. The appearance of events and their disappearance changes the state of the *world* of events. This is fertile ground for the application of finite state models such as probabilistic automata. Investigating the behaviour of these models may lead to new insights about the structure of Allen relations.

# 3 Temporal Reasoning in Python

One of the advantages of finite temporality strings is that they can be represented in a computationally efficient manner. The temporal reasoning system described by Allen [1] is based on representing event intervals as nodes and the temporal relations that hold between them as labelled arcs connected two nodes, thereby building a directed graph of temporal knowledge. Representing graphs in code requires complex techniques such as adjacency matrices or adjacency lists. Finite temporality strings can be represented simply by using a primitive string or a basic array data structure that is available in most if not all programming languages, and holds the individual $\alpha_i$ sets of a given finite temporality string. These strings also present the appealing visual of a timeline, where all events and their relationships can be viewed at once.

## 3.1 Representing Finite Temporality Strings

Given a finite temporality string (subsequently referred to simply as a *string*) $s = \alpha_1 ... \alpha_n$ with each $\alpha_i$ being a subset of $A$, the finite fixed alphabet of *fluents*, a primitive python string could be used to represent it, such as `"|a|c|"` or `"|a,b|c,d|"` but these should only be used for displaying strings. Internally some other data structure should be used as raw strings are not easy to manipulate and this will become a problem when implementing operations on these strings. Using an array is a slight improvement, but again, when it comes to manipulating these strings, we would very much like to avoid having to mess around with array indices and mutating these arrays in place.

A better solution is to use a linked list, where each node represents a set in $\mathcal{P}(A)$, the power set of $A$, and contains the set of fluents within it *e.g* `|a,b|c,d|` would be the linked list with two nodes that represent $\boxed{a, b}$ and $\boxed{c, d}$. This allows for easy manipulation, *i.e* superposing, block compression, and generating bc-equivalent strings, and results in readable, *pythonic*[1] code.

We take an object oriented approach, and define the class `FTString`, as that which will represent a single finite temporality string. Its constructor is defined below.

```
1 class FTString:
2     def __init__(self, init=[]):
3         self.head = None
4         self.tail = None
```

[1]Code that is *pythonic* is code that follows the preferred way of doing things in python. There are many ways of course to implement a given program, but doing it the *pythonic* way is often the best.

```
5           self.length = 0
6           try:
7               for a_i in init:
8                   self.add(a_i)
9           except:
10              raise ValueError("Constructor argument is not
                ↪  iterable or does not consist of sets")
11
12      def add(self, fluents):
13          if not isinstance(fluents, set):
14              raise ValueError("Argument must be a set")
15          new = FTNode(fluents)
16          if len(self) == 0:
17              self.head = new
18              self.tail = new
19          else:
20              new.prv = self.tail
21              self.tail.nxt = new
22              self.tail = new
23          self.length += 1
```

In the above Listing, we define the constructor, and another method named `add()`. This class is a linked list, so the constructor initialises three member variables, `head`, `tail`, and `length`. The first two are pointers to the first and last element in the string. These are of type `FTNode`, which is just a python class with three member variables:

`fluents` The set $\alpha$ that is represented by this node

`nxt` A pointer to the next `FTNode` in the list

`prv` A pointer to the previous `FTNode` in the list

```
1   class FTNode:
2       def __init__(self, fluents, nxt=None, prv=None):
3           self.fluents = fluents
4           self.nxt = nxt
5           self.prv = prv
```

The constructor for `FTString` takes an optional iterable of sets, *e.g* a `list`, that would populate the string, otherwise it is initialised as empty. The member function `add()` takes care of this as well as changing the `head` and `tail` pointers

to reflect the newly added `FTNode`. There is also type checking to make sure the input is a python `set`. Each new `FTNode` is simply just appended to the end of the list.

We also implement an iterator for this class so we can use the `for ... in` python syntax and be in keeping with the *pythonic* way of doing things. In python, an iterable is a class which provides an implementation for the `__iter__` built-in function. Typically, this function should return a new instance of an iterator that iterates through each element in the data structure. In this case we return an iterator that iterates through each set $\alpha_i$ in our string $s$. The implementation of `__iter__` is given below and is just one line, returning a new instance of the the `FTStringIterator` class.

```
def __iter__(self):
    return FTStringIterator(self.head)
```

For a class to be considered an iterator it must implement both `__next__` and `__iter__`. As the name suggests, each time `__next__` is called, it returns the next element *i.e.* the next set in the string. When we reach the end of the string, a `StopIteration` exception is raised, signalling the end of the string. Within `__next__`, we keep track of the current `FTNode` the iterator is on, update the pointer to go to the next node, and return the current one. We either return the `FTNode` instance, or just the set `fluents`, depending on the constructor argument `node`. Iterators themselves are iterable, thus we define the `__iter__` method to return the instance of the iterator.

```
class FTStringIterator:
    def __init__(self, string_head, node=False):
        self.head = string_head
        self.index = 0
        self.node = node

    def __next__(self):
        try:
            fluents = self.head.fluents
            f_head = self.head
        except AttributeError:
            raise StopIteration()
        self.index += 1
        self.head = self.head.nxt
        if not self.node: return fluents
        else: return f_head

```

```
18      def __iter__(self):
19          return self
```

To illustrate the iterator's behaviour, the following example is given. We can now create a new string and print out each set within it:

```
s = FTString([{'a'},{'b','c'},{'d'}])
for alpha in s:
    print(alpha)
```

```
{'a'}
{'b', 'c'}
{'d'}
```

### 3.1.1 Set Isolation

Given a string $s$, it is often useful to split it into $s = \alpha\alpha' s'$ or $s = \alpha s'$. The member function `isolate()` takes a single integer parameter and decides how many sets to isolate. For example, `s.isolate(1)` would split the string into $\alpha s'$. This operation is very helpful in the computation of the superposition of two strings, as well as in block compression. This function returns an $n+1$-length tuple of `FTString` objects respectively representing the string split up into $\alpha$, $\alpha'$, and $s'$. The `as_set` argument determines whether to return the separate $\alpha_i$'s as `FTString` instances or as `set` instances.

```
def isolate(self, n, as_set=True):
    if not isinstance(n, int):
        raise ValueError("Trying to isolate non integer number of
        ↪  sets")
    if n > self.length or n < 1:
        raise ValueError("Trying to split string into too many/few
        ↪  values")

    result = []
    s_prime_alphas = []

    for i, alpha in enumerate(self):
        if i < n:
            if as_set: result.append(alpha)
            else: result.append(FTString([alpha]))
        else:
            s_prime_alphas.append(alpha)
    result.append(FTString(s_prime_alphas))
    return tuple(result)
```

The algorithm here is straightforward, we loop through each $\alpha_i$ for $i < n$ and isolate those sets of fluents, and the remaining sets are combined into one string. The result is then returned as a `tuple`. This behaviour is illustrated in the following example, where we isolate the first set, and then the first two sets of the string.

```
s = FTString([{'a'},{'b'},{'c'},{'d'}])
print(s.isolate(1))
print(s.isolate(2))
```

```
({'a'}, FTString([{'b'}, {'c'}, {'d'}]))
({'a'}, {'b'}, FTString([{'c'}, {'d'}]))
```

### 3.1.2 Block Compression and its Inverse

The block compression of a string $s$, $bc(s)$, is defined as the following from Fernando et al. [9]:

$$bc(s) = \begin{cases} s & \text{if } length(s) \leq 1 \\ bc(\alpha s') & \text{if } s = \alpha\alpha s' \\ \alpha bc(\alpha' s') & \text{if } s = \alpha\alpha' s' \text{with } \alpha \neq \alpha' \end{cases} \tag{4}$$

This removes the stutter in strings *e.g* $bc(\boxed{}\boxed{a}\boxed{a}\boxed{a}\boxed{b}\boxed{b}\boxed{}) = \boxed{}\boxed{a}\boxed{b}\boxed{}$. The Listing below gives the python implementation, which is a recursive function since the mathematical definition is itself recursive. When the length of $s$ is less than or equal to 1, we just return a new instance that is the same as $s$ so there are no reference issues with just simply returning the argument. The new instance is created via `s * 1`. The `*` operator has been overloaded to mean repeated concatenation of a string and returns a new object.

```
def __bc(s):
    if len(s) <= 1: return s * 1
    alpha, alpha_p, s_p = s.isolate(2, as_set=False)
    if alpha == alpha_p:
        return FTString.__bc(alpha_p + s_p)
    else:
        return alpha + FTString.__bc(alpha_p + s_p)
```

We overload the unary operator `~` to return its argument block compressed, resulting in concise and readable code.

```
s = FTString([set(),{'a'},{'a'},{'a'},{'b'},{'b'},set()])
print(~s)
```

```
| |a|b| |
```

Inverse block compression is when we introduce stutter to strings. Any string $s$ has an infinite language of *bc-equivalent* strings, and the function `pad(k)` returns the finite set of *bc-equivalent* strings only of length $k$. The implementation of this algorithm is done using the idea that this is just a case of solving the equation

$$\sum_{i=1}^{n} x_i = k$$

where n is the length of the string. The value of each $x_i$ in a given solution represents how many times we repeat the $i$ -th set of fluents in the string. Let us consider an example with the string $s = \boxed{a}\boxed{b}\boxed{c}$, and we want to find all bc-equivalent strings of $s$ of length $k = 6$. One solution of the equation

$$\sum_{i=1}^{3} x_i = 6$$

is $x_0 = 1, x_1 = 1, x_2 = 4$, which would represent the string $\boxed{a}\boxed{b}\boxed{c}\boxed{c}\boxed{c}\boxed{c}$. The following Listing gives the implementation of a recursive function that calculates all $\binom{k-1}{n-1}$ solutions. This is a generator function as given by the `yield` statements. Generator functions are described in greater detail in the next section where they are used to implement vocabulary constrained superposition.

```
1  def inv(n, k):
2      if n <= 0 or k <= 0:
3          yield None
4      if not isinstance(n, int) or not isinstance(k, int):
5          yield None
6      elif n == 1:
7          yield [k]
8      elif n == k:
9          yield [1 for _ in range(n)]
10     else:
11         x1_max = k - (n - 1)
12         for x1 in range(1, x1_max + 1):
13             sol = [x1]
14             for s in inv(n - 1, k - x1):
15                 yield sol + s
```

We have that for any solution $x_1, .., x_n$, the corresponding bc-equivalent string for $s = \alpha_1 ... \alpha_n$ will be $(\alpha_1 * x_1)...(\alpha_n * x_n)$. The following listing gives the member function that generates bc-equivalent strings for a given `FTString` instance.

```
def pad(self, k):
    if k < len(self): raise ValueError("k is smaller than the
    ↪  length of the string")

    for x in inv(len(self), k):
        a_i_strings = map((lambda a_i, x_i : FTString([a_i]) *
        ↪  x_i), self, x)
        yield reduce(add, a_i_strings)
```

The previous example of finding the *bc-equivalent* strings of length 6 for the string `|a|b|c|` is done below in python. We loop through the results of `pad(6)` and print them out below.

```
s = FTString([{'a'},{'b'},{'c'}])
for string in s.pad(6):
    print(string)
```

```
|a|b|c|c|c|c|
|a|b|b|c|c|c|
|a|b|b|b|c|c|
|a|b|b|b|b|c|
|a|a|b|c|c|c|
|a|a|b|b|c|c|
|a|a|b|b|b|c|
|a|a|a|b|c|c|
|a|a|a|b|b|c|
|a|a|a|a|b|c|
```

### 3.1.3 Relation Resolution & Well Formedness

An important property of these strings is that the event intervals have only one beginning and one ending, and equally important is the ability to determine whether a string obeys this property *ie* whether it is *well formed*. That is what `well_formed()` does, taking advantage of the idea given in [9], that applying block compression to a string which has been reduced with $\rho_{\{a\}}(s)$ $\forall a \in A$, produces a string consisting of only $a$ with empty sets on either side, $\boxed{\phantom{a}}\boxed{\phantom{a}}\boxed{a}\boxed{\phantom{a}}\boxed{\phantom{a}}$. The empty sets are there to show that an event does not occur for an infinite period of time *i.e* we can be certain that there existed something before it and

it will end with something existing after it. If an event started and ended more than twice, the result of $bc(\rho_{\{a\}}(s))$ would have the event appearing more than once.

In the Listing below we define three new member functions of the `FTString` class:

`voc` returns the set $A$, the vocabulary of the string

`reduct` Takes argument `X`, of type `set`, and implements $\rho_X(s) = (\alpha_1 \cap X)...(\alpha_n \cap X)$, returning a new `FTString` instance.

`well_formed` returns `True` or `False` representing whether the string is well formed or not.

```
def voc(self):
    v = set()
    for alpha in self: v = v | alpha
    return v

def reduct(self, X):
    if not isinstance(X, set):
        raise ValueError("Argument is not a set")
    s = FTString()
    for alpha in self: s.add(alpha & X)
    return s


def well_formed(self):
    vocab = self.voc()
    for f in vocab:
        reduced = FTString.__bc(self.reduct({f}))
        if reduced != FTString([set(),{f},set()]):
            return False
    return True
```

The *X-reduct* operation $\rho_X(s)$ can also be used to resolve the relation between some subset of events $X$ in the string's vocabulary. Take the string $\boxed{\ }\boxed{a,b,d}\boxed{b,c,d}\boxed{c,d}\boxed{d}\boxed{\ }$ that has vocabulary $\{a,b,c,d\}$. By using $\rho_X(s)$ and then applying the block compression, we can, as Allen puts it, "turn up the magnification".

To resolve the temporal relation between $b$ and $c$ for example, we apply the $\{b,c\}$-reduct to the string

$$bc(\rho_{\{b,c\}}(\boxed{\boxed{}\boxed{a,b,d}\boxed{b,c,d}\boxed{c,d}\boxed{d}\boxed{}})) = \boxed{\boxed{}\boxed{b}\boxed{b,c}\boxed{c}\boxed{}},$$

and we see that $b$ overlaps $c$. The Listing below shows how to do this in code.

```
from ftstring import FTString
s = FTString([set(),{'a','b','d'},{'b','c','d'},{'c','d'},{'d'},set()])
print(~s.reduct({'b','c'}))
```

```
| |b|b, c|c| |
```

### 3.2 Vocabulary Constrained Superposition

Let $voc(s)$ be defined as the union of the components of a string, which is the result of `voc()`. We have that the vocabulary constrained superposition of two strings $s$ and $s'$ as defined in Fernand and Woods [8] to be

$$s \;\&_{vc}\; s' = s \;\&_{voc(s),voc(s')}\; s'.$$

Given string $s_1 = \alpha s$ and $s_2 = \alpha' s'$ and the set $\Sigma$ is $voc(s)$ and $\Sigma'$ is $voc(s')$, then

$$s_1 \;\&_{\Sigma,\Sigma'}\; s_2 = \begin{cases} \{(\alpha \cup \alpha')s'' | s'' \in L\}, \text{if } \Sigma \cap \alpha' \subseteq \alpha \text{ and } \Sigma' \cap \alpha \subseteq \alpha' \\ \varnothing, \text{ otherwise} \end{cases}$$

where

$$L = (\alpha s \;\&_{\Sigma,\Sigma'}\; s') \cup (s \;\&_{\Sigma,\Sigma'}\; \alpha' s') \cup (s \;\&_{\Sigma,\Sigma'}\; s')$$

The implementation of vocabulary constrained superposition is fertile ground for `python`'s generator functions. A generator, much like an iterator, generates each new value lazily, *i.e* as it is needed. Instead of calculating each possible result from the superposition of strings and then returning them in a list, we instead return a generator, which produces the next timeline on demand. In the literature, superposition is written with the & operator, and so we will overload the binary `&` operator. This is done below:

```
def __and__(self, obj):
    if not isinstance(obj, FTString):
        return NotImplemented
    else:
        return SuperpositionGenerator(self, obj)
```

```
6
7  def __rand__(self, obj):
8      if isinstance(obj, SuperpositionGenerator):
9          return SuperpositionGenerator(self, obj)
10     else:
11         return NotImplemented
```

The `SuperpositionGenerator` class encapsulates the generation of the timelines that result from superposition. The constructor is straightforward. We check to make sure that the arguments are indeed only of type `FTString` or `SuperpositionGenerator`. Next we have two static functions , `__L` and `__svc`. These are both generator functions since there is the `yield` keyword. Whatever is on the right hand side of a `yield` expression is returned to the caller, and the next time the function is invoked, the function picks up where it left off until it reaches the next `yield` statement. This is the main feature of generators, and is what allows us to *lazily* generate timelines. We yield the newly generated timeline, and then the next time the function is called, we can calculate the next timeline and yield that, and so on. What is returned by these generator functions are generator objects, which behave much in the same way as iterators. Ramalho [7] explains

> When we invoke `next(...)` on the generator object, execution advances to the next `yield` in the function body, and the `next(...)` call evaluates to the value yielded when the function body is suspended. Finally, when the function body returns, the enclosing generator object raises `StopIteration`, in accordance with the `Iterator` protocol. [7, p. 429]

In `__L`, the function that generates $L$ from the mathematical definition, we include a `yield from` statement, which simply means that the function should yield from the generator expression on the right hand side, and once it is exhausted, continue on with the rest of the statements in the function.

The function `__svc` then implements the algorithm for vocabulary constrained superposition as given in the definition. It is a recursive function, since we call `__L` which in turn calls `__svc` to generate the set $L$. We also implement `__and__` and `__rand__` to allow us to use the `&` operator when superposing instances of `FTString` and `SuperpositionGenerator`. When superposing two `SuperpositionGenerators` this is equivalent to superposing two languages of strings, where every string in one language is superposed with the every string in the other language. This is given in the `__iter__` method, where we check each possible case, and perform the corresponding operation.

```
class SuperpositionGenerator:
    def __init__(self, arg1, arg2):
        if isinstance(arg1, FTString) or isinstance(arg1,
        ↪  SuperpositionGenerator):
            self.arg1 = arg1
        else:
            raise ValueError("Must be an FTString or another
            ↪  SuperpositionGenerator")
        if isinstance(arg2, FTString) or isinstance(arg2,
        ↪  SuperpositionGenerator):
            self.arg2 = arg2
        else:
            raise ValueError("Must be an FTString or another
            ↪  SuperpositionGenerator")


    def __L(a, s, a_p, s_p, sig, sig_p):
        a_s = FTString([a]) + s
        a_p_s_p = FTString([a_p]) + s_p
        yield from SuperpositionGenerator.__svc(a_s, s_p, sig,
        ↪  sig_p)
        yield from SuperpositionGenerator.__svc(s, a_p_s_p, sig,
        ↪  sig_p)
        yield from SuperpositionGenerator.__svc(s, s_p, sig,
        ↪  sig_p)

    def __svc(s1, s2, sig=None, sig_p=None):
        if len(s1) == 0 and len(s2) == 0:  yield FTString()
        elif len(s1) == 0 or  len(s2) == 0: yield None
        else:
            if sig == None: sig = s1.voc()
            if sig_p == None: sig_p = s2.voc()
            a, s = s1.isolate(1)
            a_p, s_p = s2.isolate(1)
            if (sig & a_p <= a) and (sig_p & a <= a_p):
                temp = FTString([a | a_p])
                L = SuperpositionGenerator.__L(a, s, a_p, s_p,
                ↪  sig, sig_p)
                for s_dp in L:
                    if s_dp is not None:
                        yield temp + s_dp
```

```
34              else:
35                  yield None
36
37
38      def __and__(self, other):
39          if isinstance(other, FTString) or isinstance(other,
            ↪  SuperpositionGenerator):
40              return SuperpositionGenerator(self, other)
41
42          else:
43              return NotImplemented
44
45      def __rand__(self, other):
46          if isinstance(other, FTString):
47              return SuperpositionGenerator(self, other)
48          else:
49              return NotImplemented
50
51      def __iter__(self):
52          if isinstance(self.arg1, FTString) and
            ↪  isinstance(self.arg2, FTString):
53              yield from SuperpositionGenerator.__svc(self.arg1,
                ↪  self.arg2)
54
55          elif isinstance(self.arg1, FTString) and
            ↪  isinstance(self.arg2, SuperpositionGenerator):
56              for timeline in self.arg2:
57                  yield from timeline & self.arg1
58
59          elif isinstance(self.arg1, SuperpositionGenerator) and
            ↪  isinstance(self.arg2, FTString):
60              for timeline in self.arg1:
61                  yield from timeline & self.arg2
62
63          else:
64              for timeline1 in self.arg1:
65                  for timeline2 in self.arg2:
66                      yield from timeline1 & timeline2
```

We can now use this class to superpose strings together. The Listing below shows that when we superpose two unconstrained strings *i.e.* a string consisting

of just one fluent, we get the 13 Allen relations as expected.

```
s1 = FTString([set(),{'a'},set()])
s2 = FTString([set(),{'b'},set()])
gen = s1 & s2
for timeline in gen: print(timeline)
```

```
| |b| |a| |
| |b|a, b|a| |
| |b|a, b|b| |
| |b|a, b| |
| |b|a| |
| |a|a, b|a| |
| |a|a, b|b| |
| |a|a, b| |
| |a| |b| |
| |a|b| |
| |a, b|a| |
| |a, b|b| |
| |a, b| |
```

Note that now we have exhausted the generator, so if we try to get the next timeline, it will return nothing.

The next example shows a more complex situation, where we have 3 strings and 4 fluents. When we superpose these strings, we get only one timeline, representing the only way in which these 4 events could have played out in time.

```
s1 = FTString([set(),{'t1'},{'ei1','t1'},{'t1'},set()])
s2 = FTString([set(),{'ei9'},{'t1','ei9'},{'ei9'},set()])
s3 = FTString([set(),{'ei9'},set(),{'ei10'},set()])

for timeline in s1 & s2 & s3:
    print(timeline)
```

```
| |ei9|t1, ei9|ei1, t1, ei9|t1, ei9|ei9| |ei10| |
```

We have that `ei1` happened during `t1`, which happened during `ei9`, which occured before `ei10`. Thus the single timeline produced is the only possible resulting timeline of these 4 events. This example is taken from a document from the TimeBank corpus, [6]. This is a collection of 183 news articles annotated with the TimeML [5] schema. Events that occur in the text are given a unique identifier and the relations that hold between them are annotated using `TLINK` XML tags. By extracting `TLINK` tags from these TimeML documents, we can construct corresponding finite temporality strings, and superpose them together to produce a timeline summary of the text within a document.

### 3.3 Performance and Evaluation

Due to fact that superposition is associative and commutative, we can use brackets to group strings with shared fluents so we can reduce the total number of timelines in any given generator. Take an example with 3 strings, $[\,[a]\,[b]\,]$, $[\,[b]\,[c]\,]$, and $[\,[c]\,[d]\,]$. The first and the last share no fluents, and so we will have 321 possible timelines, but when superposed with the second, the possible timelines are reduced to just 1, $[\,[a]\,[b]\,[c]\,[d]\,]$.

The amount of timelines increases exponentially for a given number of events. This is the main motivation for using generators, as, in any real life application, we will have lots of events and millions of timelines and storing them all in memory is not feasible. Generators allow us to generate timelines one by one and throw them away when we are done. They do not linger in a data structure in memory, and if we want to keep them, the best practice is to store them into a file.

Memory-wise, superposing two strings is the same as superposing five, however depending on the strings, one will generate millions of timelines, and the other may generate only one, and the time taken to run through all these timelines will vary. Table 7 gives 4 example superposition operations where we superpose unconstrained event strings, *i.e* we know nothing beforehand of the temporal relationships that may hold between events. Superposing up to four unconstrained event strings takes very little time, however when we add a fifth event, the time skyrockets since the total number of timelines skyrockets, from 23,917 to slightly over 2.2 million.

Table 7: The time taken and the number of timelines generated for several superposition operations.

| Operation | Time | Timelines |
|---|---|---|
| $[\,[a]\,] \,\&_{vc}\, [\,[b]\,]$ | 0.0016s | 13 |
| $[\,[a]\,] \,\&_{vc}\, [\,[b]\,] \,\&_{vc}\, [\,[c]\,]$ | 0.0516s | 409 |
| $[\,[a]\,] \,\&_{vc}\, [\,[b]\,] \,\&_{vc}\, [\,[c]\,] \,\&_{vc}\, [\,[d]\,]$ | 3.7734s | 23917 |
| $[\,[a]\,] \,\&_{vc}\, [\,[b]\,] \,\&_{vc}\, [\,[c]\,] \,\&_{vc}\, [\,[d]\,] \,\&_{vc}\, [\,[e]\,]$ | 446.2086s | 2244361 |

In general to avoid these long computation times, avoid superposing strings with a lot of events and little connections between them. This uncertainty about what may hold between events is what results in these large numbers of timelines, since we produce every possible result.

### 3.4 Calculating Allen Relation Probabilities

The question of

> Given an Allen relation $R$, what is the probability that $R$ relates intervals $a$ and $a'$, $aRa'$?

that is the motivating question behind [3], and which is answered through analytical means in that paper can also be answered here.

The probability of an Allen relation is the proportion of timelines in the result of the superpostion of $n$ unconstrained event strings where some two intervals $a$ and $b$ are related via that relation. This is done mathematically in [3], and here we can calculate these proportions by using the `SuperpositionGenerator` class to superpose the $n$ event strings, and then count the number of times intervals $a$ and $b$ are related via the relations *equals*, *before*, *after*, *during*, and so on. Appendix A gives the `python` code on how to do exactly this.

We may group relations together into 3 categories: *short (e)*, *medium (m, mi, s, si, f, fi)*, and *long (b, bi, d, di, o, oi)*, as done in [3], since relations that fall into the same category will have the same probability, as described in Section 2. Table 8 summarizes the proportions for short, medium, and long relations for $n = 2, 3$, and 4 unconstrained event strings. Looking closely, these are the exact results from [3] (Table 3) when time is modelled as a collection of intervals. The mathematical procedure for calculating probabilities devised in [3] mirrors that which we do here, since in the end it is just calculating the proportion of the possible outcomes that satisfy a given constraint *i.e* that intervals $a$ and $b$ are temporally related via some relation $R$.

Table 8: Probabilities of short, medium, and long relations as we change the number of events.

| $n$ | $p_n(e)$ | $p_n(m)$ | $p_n(b)$ |
|---|---|---|---|
| 2 | 1/13 | 1/13 | 1/13 |
| 3 | 0.03178 | 0.06112 | 0.10024 |
| 4 | 0.01710 | 0.04921 | 0.11460 |

### 3.5 Conclusion

We have introduced a framework for temporal reasoning in `python` based on the finite temporality approach pioneered by Fernando et al. [9, 8]. The framework allows for the creation and manipulation of finite temporality strings, their combining through an efficient implementation of vocabulary constrained superposition, and the easy resolution of the temporal relations that may hold

between event intervals. In the next section we go on to look at this question of probability more closely, and develop new methods for calcuating timeline probabilities.

# 4 Timeline Generation and Likelihood

The question of

> Given an Allen relation $R$, what is the probability that $R$ relates intervals $a$ and $a'$, $aRa'$?

is considered here in the general sense, *i.e*

> What is the probability of a given *timeline* for some set of events $A$?

The former question is answered in Fernando and Vogel [3], and the latter is the focus of this section, where we attempt to answer it through the lense of finite temporality strings. We have already reproduced the results of [3] by using the temporal reasoning framework introduced in the previous section to generate all possible timelines and then look at the proportions of those possible timelines that satisify a given relation R, with that being understood as the *probability* of a relation. In this section we follow a different approach.

These strings constitute a timeline of events, and we can construe each $\alpha$ interval set of a string as being the result of some random process. Simulation of this random process can be used to generate timelines, and by looking at the proportion of the possible timelines generated for some set of event names $A$, we may come to probabilities of timelines.

## 4.1 Stochastic Superposition

A random process, or a *stochastic* process, is a collection of $n$ random variables $X_0, ..., X_{n-1}$ and it is Markovian if it satisfies the *Markov property*, in that the value of $X_i$ is conditionally dependant only on $X_{i-1}$. The history of the process is irrelevant. Mathematically, it is that

$$P(X_i = x | X_{i-1} = x_{i-1}) = P(X_i = x | X_{i-1} = x_{i-1}, ..., X_0 = x_0).$$

Subsequently we will be considering finite temporality strings as the result of a stochastic process, what we term here as *stochastic superposition*. More precisely, each interval set $\alpha_i$ in a string $s = \alpha_1 ... \alpha_n$ can be reimagined as a random variable, and the string itself as the *collection*.

Strings represent the timeline of some set of events, $A$, or its *vocabulary*. The vocabulary of a string $s$, is the union of its components, and we have that each $\alpha_i$ is a subset of $A$. Thus the *state space* of stochastic superposition *i.e* the possible values for each $X_i$, is simply the power set of $A$, $\mathcal{P}(A)$. An important property of the events in these strings is that they happen only once. They

cannot reoccur since they, philosophically, represent the occurence of a singular event instance. The event $a \in A$ may, for example, represent last Tuesday, and since last Tuesday cannot stop and start again, neither can the event $a$. More formally, if we have a string of length $n$ generated from a stochastic process

$$X_0, ..., X_{i-1}, X_i, ..., X_j, ..., X_{n-1},$$

with each $X_i \in \mathcal{P}(A)$, then $\forall x \in (X_{i-1} - X_i)$ , $x \notin X_j$ , $j > i$, with $-$ representing set difference. We are saying that any event that ends in $X_{i-1}$, cannot reappear in a later interval. What this means is that when we have such a transition where $|X_{i-1} - X_i| > 0$, the number of possible values for the next random variable gets smaller, since we cannot transition to any state where the events that ended in $X_{i-1}$ reappear. Note this language of states and transitions. The stochastic process of *superposition* can be remodelled using finite state methods, an idea we expand greatly upon subsequently.

Taking the string $\boxed{\phantom{a}}\boxed{a}\boxed{\phantom{b}}\boxed{b}\boxed{b,c}\boxed{\phantom{a}}$, as an example we have $X_1 = \{a\}$ and $X_2 = \emptyset$ , and that $X_1 - X_2 = \{a\}$. This tells us that the event represented by $a$ is over, and therefore cannot reappear in any later intervals. This behaviour is obeyed with this example, and the string is said to be *well formed*. In fact, any string $s \in s' \ \&_{vc} \ s''$ will be well formed assuming $s'$ and $s''$ are well formed.

### 4.1.1 Stochastically Generating Well Formed Strings

Each $\alpha_i$ in a finite temporality string is an element from the powerset of the vocabulary. In *stochastic superposition*, each $X_i$ is randomly chosen from its corresponding set of possible values, $S_i$, and how this set evolves is explained below.

Let $S_0$ be the initial *state space* of our stochastic process, which is $\mathcal{P}(A)$. Also, define $O_i$, the set of events that cannot reappear in a later $X_j$, $j > i$, to be

$$O_i = \begin{cases} \emptyset & \text{if } i = 0 \\ O_{i-1} \cup (X_{i-1} - X_i) & \text{if } 0 < i \leq n-1 \end{cases} \tag{5}$$

and observe that $\bigcup_{i=0}^{n-1} O_i = A$ for a well formed string. The value of $X_i$, the random variable, must be chosen from its corresponding state space $S_i$. We know that events cannot reappear once they have ended. We also disallow any stutter in strings, so the state space changes after each step, either removing subsets where an ended event appears or removing the previous $X_i$ in order to prevent stutter. We have that $S_0 = \mathcal{P}(A)$, then let the set $P_i$ represents the set of states such that $\forall j, \ \ i < j < n, \ X_j \notin P_i$. We then have that

$$P_i = \{s \in S_0 \mid \ |s \cap O_i| > 0\} \tag{6}$$

$P_i$ is the set of possible states that a later $X_j$ cannot have. With $S_0 = \mathcal{P}(A)$, we may then define $S_i$, the space of possible values for $X_i$ to be

$$S_i = (S_0 - P_{i-1}) - \{X_{i-1}\} \tag{7}$$

Take the example string $\boxed{\ }\boxed{a}\boxed{a,b}\boxed{a,b,c}\boxed{b,c}\boxed{c}\boxed{\ }$. The initial state space is $S_0 = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\}\}$. Table 9 shows the evolution of the state space after each $X_i$ is randomly chosen from its corresponding state space.

Table 9: State space evolution

| $i$ | $S_i$ | $X_i$ | $O_i$ |
|---|---|---|---|
| 0 | $\{\varnothing, \{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\}\}$ | $\{\varnothing\}$ | $\varnothing$ |
| 1 | $\{\{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\}\}$ | $\{a\}$ | $\varnothing$ |
| 2 | $\{\varnothing, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\}\}$ | $\{a,b\}$ | $\varnothing$ |
| 3 | $\{\varnothing, \{a\}, \{b\}, \{c\}, \{b,c\}, \{a,c\}, \{a,b,c\}\}$ | $\{a,b,c\}$ | $\varnothing$ |
| 4 | $\{\varnothing, \{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}\}$ | $\{b,c\}$ | $\{a\}$ |
| 5 | $\{\varnothing, \{b\}, \{c\}\}$ | $\{c\}$ | $\{a,b\}$ |
| 6 | $\{\varnothing\}$ | $\{\varnothing\}$ | $\{a,b,c\}$ |

We can see that $\forall i, X_i \in S_i$, and that $S_i$ strictly includes only values that, if chosen, result in a well formed string. The exclusion of $\{X_{i-1}\}$ is to prevent stutter. We can also see that $O_i$ keeps track of the events that have already ended.

When $S_i = \varnothing$ *i.e.* step 7 of the preceding example, the string has ended. There no longer exist subsets of $A$ where events that have already ended do not reappear or are not the previous. The sample space is eventually exhausted, and the process cannot continue indefinitely according to the rules defined above. All the possible values have been visited due to the fact that none remain, and since we cannot repeat, there is nothing left to do, thus the string ends.

The selection of $X_i$ from its corresponding state space $S_i$ directly affects what the next state space, from which we select $X_{i+1}$, will be. These depedencies between state spaces are visualised in Figure 1, as a Finite State Acceptor (F.S.A) that accepts well formed strings that concern only two events, $A = \{a, b\}$. The alphabet of this F.S.A is $\mathcal{P}(A)$. Each state represents a possible state space $S_i$ from which $X_i$ may be chosen, and the choice of a particular $X_i$ is what determines the transition from one state space to another. The state $s_0$ in the F.S.A represents the power set, since this is the state space at the beginning,

and $s_4$, the accepting state, represents the empty state space; no transitions are allowed without disobeying the rules of a well formed string. All paths from $s_0$ to $s_4$ constitute a timeline of two events, and the 13 unique paths are the 13 Allen relations.

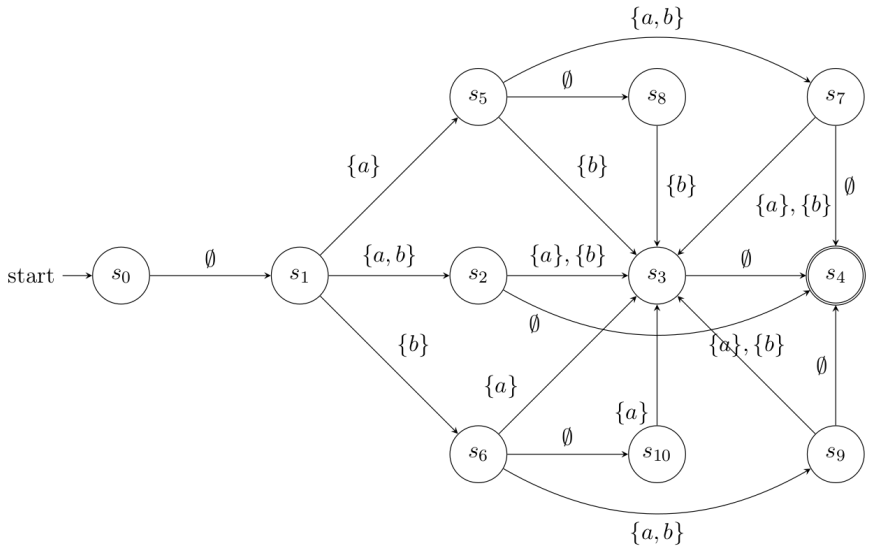



Figure 1: The allowed transitions between state spaces for two events.

To simulate timelines, one simply needs to assign *transition probabilities* to each arc in the F.S.A, and build timelines by keeping track of the $X_i$'s that are chosen. In the subsequent sections, we do exactly this but not before introducing a new construction.

## 4.2 The Unborn Living Dead Construction

Events can be described in one of three ways: they are yet to happen, are currently happening, or have already happened. We can rephrase this by classifying events as being either *unborn*, *living*, or *dead*, forgoing any of the spirituality connotations. Given a finite non-empty set of event names $A$, the triple $(U, L, D)$ partitions $A$ into those events *unborn*, those *living*, and those *dead*. Since events cannot be unborn, living, or dead at the same time, $U$, $L$, and $D$ are disjoint subsets of $A$, and whose union is $A$, since all events must be in one of these three sets. Subsequently we refer to these triples as $A$-states.

Unborn events can be born and made living, and living events can die. Events cannot go straight from unborn to dead, since for something to die, it

must first have lived. Let $\rightsquigarrow$ be the binary relation on $A$-states given by

$$(U, L, D) \rightsquigarrow (U', L', D') \iff U' \subseteq U \text{ and } L \neq L' \text{ and } D \subseteq D' \subseteq D \cup L. \quad (8)$$

We ban stuttering by imposing that $L \neq L'$. $D \subseteq D'$ implies that once an event is dead it is always dead, and $D' \subseteq D \cup L$ says that only those living can die. These rules are necessary in order for a sequence of $A$-states to constitute a temporally coherent timeline *i.e* a well formed string.

An *A-story* is a path from $(A, \emptyset, \emptyset)$ to $(\emptyset, \emptyset, A)$, *i.e* all the events must live and die for a sequence of $A$-states to be considered an *A-story*, a sort of narrative where the characters are these event names, and whose *personalities* and *relationships* are expressed only through their birth and death. An $A$-story is a piece of concise prose bordering on the laconic, however our concern here is not with the medium's expressive potential, rather it is a means to an end, that end being calculating a probability.

Take the most simple example where $A = \{a\}$, the set of a single event name. The only possible $A$-story is then

$$(\{a\}, \emptyset, \emptyset) \rightsquigarrow (\emptyset, \{a\}, \emptyset) \rightsquigarrow (\emptyset, \emptyset, \{a\})$$

where $a$ begins as being unborn, it lives, and finally dies. By concatentating the succesive $L$ sets in each $A$-state, we may build the $A$-story's corresponding *finite temporality string*. The above example would result in the string $\boxed{\phantom{a}}\boxed{a}\boxed{\phantom{a}}$. Now consider Figure 2, which gives a directed graph representation of the allowed transitions between $A$-states for $A = \{a, b\}$. The 13 unique paths along this graph from the starting node to the node where all events are dead correspond to the $A$-stories for the 13 Allen relations.

By assigning probabilities to the transitions, we get a markov chain where the stochastic sequences that result from its traversal begin at the start state (leftmost node) and end at the end state (rightmost node). Figure 2 labels the end state as an accepting state, taking notation from automata theory, and this would be the same as just labelling the transition from the end state to itself as 1, and all other transitions 0. When all events have died, nothing further can happen. There are no more unborn events to be made living, and no living events to kill, and so we end the sequence there. Further study may consider events that can *resurrect*, and look at the types of sequences that result. In this case, the event names would not represent singular event instances like *last Tuesday* but simply *Tuesdays* in general. Here we consider only events that cannot resurrect.

The transition diagram given in Figure 2 can be represented by a $3^{|A|} \times 3^{|A|}$ *right stochastic* matrix $M$, where each of the rows sum to 1. Let $S_A$ be the

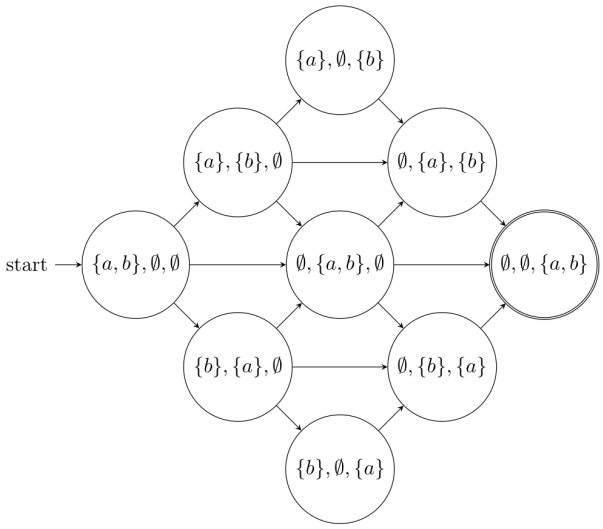



Figure 2: Allowed transitions between $A$-states that constitute $A$-stories.

set of all $A$-states for some vocabulary $A$, then $M_{ij}$ represents the transition probability of going from state $s_i \in S_A$ to $s_j \in S_A$, and $\sum_j M_{ij} = 1$. This matrix will be very sparse, since a small amount of the transitions between states are allowed to give a well formed string, given by the definition. For the diagram given Figure 2 , which we subsequently term a *ULD automaton*, we have that

$$M = \begin{bmatrix} 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

with $S_A$, the set of all $A$-states for $A = \{a, b\}$ being

$$S_A = \{(\{a,b\},\emptyset,\emptyset), (\{a\},\{b\},\emptyset), (\{b\},\{a\},\emptyset), (\{a\},\emptyset,\{b\}), (\emptyset,\{a,b\},\emptyset), (\{b\},\emptyset,\{a\}), (\emptyset,\{a\},\{b\}), (\emptyset,\{b\},\{a\}), (\emptyset,\emptyset,\{a,b\})\}.$$

The matrix M above represents a ULD automaton with uniform transition

probabilities. Figure 3 below gives this automaton with transition probabilities labelled.

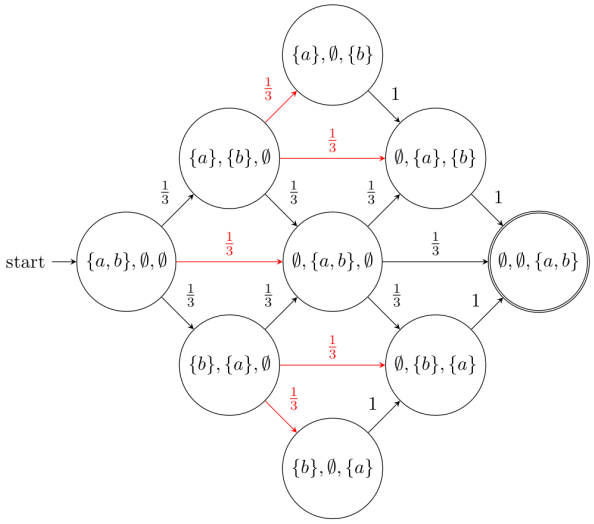



Figure 3: ULD automaton with uniform transition probabilities. Paths that go through a red arc have probability $\frac{1}{9}$, while those that do not have probability $\frac{1}{27}$.

With these transitions labelled, we can produce probabilities of $A$-stories that correspond to probabilities of finite temporality strings, and the Allen relations they represent. Paths from $(\{a,b\},\emptyset,\emptyset)$ to $(\emptyset,\emptyset,\{a,b\})$ that go through a red arc have probability $\frac{1}{9}$, while those that do not have probability $\frac{1}{27}$. Thus we split the 13 Allen relations into two[2] categories:

- 7 with probability $\frac{1}{9}$, namely $b, bi, m, mi, s, si, e$.
- 6 with probability $\frac{1}{27}$, namely $o, oi, f, fi, d, di$.

These probabilities for the 13 Allen relations do not correspond with those arrived at by Fernando and Vogel [3]. Before further discussion, it is important to understand what we mean when we say *probability*. In [3], the probability of an Allen relation is understood to mean the proportion of arrangements of intervals that satisfy it, for some number of intervals, $n$[3]. The probability of an Allen relation here is defined as how likely our ULD automaton is to generate

[2]We get the same probability distribution for *stochastic superposition*, when the probability of choosing any $X_i$ from its corresponding state space is uniform.

[3]Fernando and Vogel also consider these probabilities when time is modelled as a set of points instead of intervals, and these probabilities change as you increase $n$.

the $A$-story that the Allen relation corresponds to. It is not necessary that these two methods coincide because both have parameters that when altered produce different results. In Fernando and Vogel's case, it is the number of intervals (or points) that exist. For us, it is these transition probabilities.

Can we justify the choice in uniform transition probabilities?[4] For some events, it is very unlikely that they occur at the same time, and more likely that they begin and end at different times. Take the example where interval $a$ represents the time during which Alice is alive, and the interval $b$ represents the time during which Bob is alive. It is much more likely that Alice is born before or after Bob's birth, than them being born at the same time. The transitions $(\{a,b\},\emptyset,\emptyset) \rightsquigarrow (\{b\},\{a\},\emptyset)$ and $(\{a,b\},\emptyset,\emptyset) \rightsquigarrow (\{a\},\{b\},\emptyset)$ thus seem much more likely than $(\{a,b\},\emptyset,\emptyset) \rightsquigarrow (\emptyset,\{a,b\},\emptyset)$. If we consider being born at the *same time* means being born in the same month, then the probability is $1/12$. If we consider it to mean the same day, then the probability drops to $1/365$[5]. In general, as we increase the detail, the probability that they are born at the same time drops.

### 4.2.1 Selection of Transition Probabilities

Figure 4 gives the ULD automaton for a single event. The event $a$ is born at a given instant in time with some probability $p$, and dies with probability $q$.

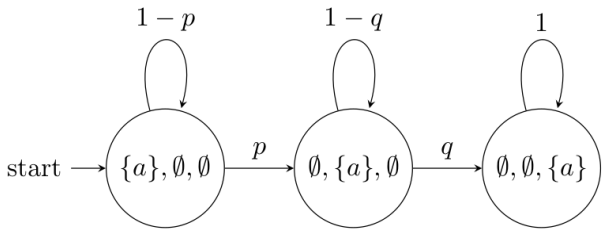



Figure 4: The ULD automaton for a single event $a$.

By running a single ULD automaton for some number of events all in lock-step, and keeping track of when events live and die, we can produce timelines of events. By doing this a large amount times, we can generate approximate distributions and see how likely a single timeline is when we have fixed the *birth* and *death* probabilities, $p$ and $q$. These birth and death probabilities have the effect of altering the probability of transitions between $A$-states. If we assume

[4]We thank Tim Fernando for this suggestion, as well as the initial idea behind the ULD construction.

[5]Disregarding leap years.

that births are unlikely to happen, then having a low birth probability would better model the situation of two people being born. The low probability of birth makes transitions where 2 events are *born* much less likely than a transition where only one event is born. In this way we can fine-tune the automaton to produce timelines that are more in keeping with reality. When we stick with the view that everything is equally likely, *i.e* $p = 0.5,\ q = 0.5$, we get a distribution of timelines that corresponds to assigning uniform transition probabilities. Figure 5 gives the distribution that results after simulating a ULD automaton for 2 events. Clearly visible are the two categories of timelines, those with probability 1/9, and those with probability 1/27, as given previously.

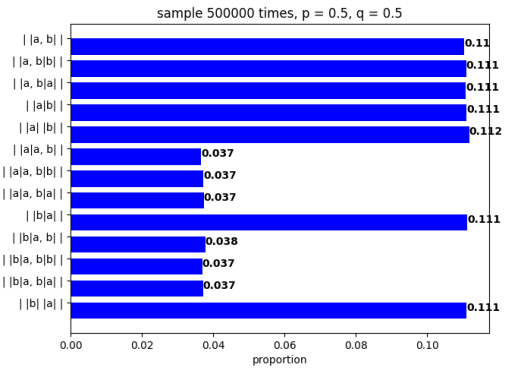



Figure 5: With $p = 0.5,\ q = 0.5$ we have produced a distribution of timelines that corresponds to uniform transitions between nodes in the directed graph representation.

This can be extended to three events. Figure 6 gives the probability distribution for timelines of 3 events, again with uniform transition probabilities.

The tallest bars represent those timelines that are the most likely and are summarized in Table 10. What is clear is that we have similiar *classes* of timelines as with when we have only 2 events. The distribution splits the 409 possible timelines into classes of that have the same probability. With 2 events, there are two classes of timelines, but for 3 events, we can make out from Figure 6 at least 4 such classes. These classes change of course as we alter the birth and death probabilities; timelines that were very likely under one model become highly unlikely in another. When the birth or death probabilities are low, timelines where events are co-occuring are unlikely to happen, and when they are high, the most likely timeline is that all events are born and die at the same time.

Figure 7 gives a more detailed picture. We plot the surface of all possible

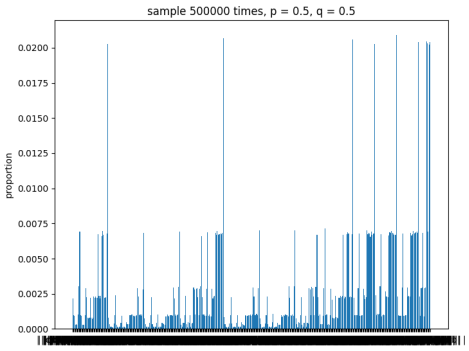



Figure 6: Probability distribution of timelines for 3 events and uniform transition probabilities. Because there are 409 possible timelines, the x -axis labels for each timeline are not shown.

Table 10: 5 of the most likely timelines for 3 events and uniform transition probabilities.

| Timeline | Probability |
|---|---|
| $\{a,b,c\}$ $a$ | 0.02 |
| $b,c$ $a$ | 0.02 |
| $a,c$ $b$ | 0.02 |
| $a,b$ $c$ | 0.02 |
| $a,b,c$ $b$ | 0.02 |

combinations of birth and death probabilites $p$ and $q$, and its height at a given point represents how likely that timeline is to be generated given the $p$ and $q$ values represented by that point. We can see that the probability of certain Allen relations are maximised at a given point. For example, the *equals* relation between two events is most likely when $p$ and $q$ are very close to 1, and the *finishes* relation is most likely when both $p$ and $q$ are around 0.5. The plot for the *equals* relation is given in Figure 8 where a clear slope is seen as the likelihood increases as we increase both $p$ and $q$.

What these surface plots also show is how the probabilities change as we vary $p$ and $q$, and how for some relations they are only highly likely (as given in yellow) for a small number of values for $p$ and $q$. The shapes of these surfaces are of interest, and point to an underlying structure of Allen relations.

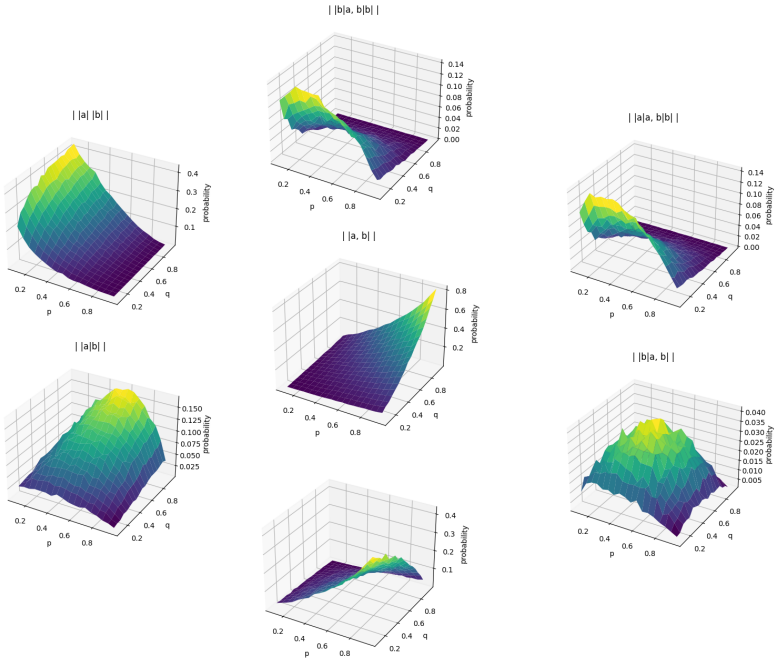



Figure 7: The likeliness of the 7 Allen relations (forgoing their inverses) *b, d, o, m, s, f, e*.

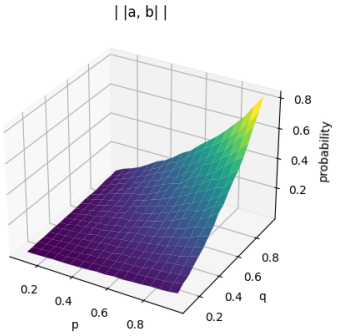



Figure 8: Likelihood plot for the *equals* relation.

We had mentioned earlier the idea of representing the ULD automaton and its transitions between $A$-states as a matrix, which begs the question of what is the relation between these birth and death probabilities, and the actual transition probabilities between $A$-states? Given in Appendix B is a description of how to come to the matrix $M$ that encodes birth and death probabilities of a particular ULD automaton. It is left out of this section since it is not entirely relevant, being an interesting aside into different representations of this albeit strange *unborn*, *living*, *dead* construction. In effect, it works by taking bigram counts of transitions between $A$-states from those strings generated by simulation; this is explained in further detail in the Appendix.

### 4.3 Applications to the TimeBank Corpus

These ideas of timeline probabilities can be applied to real world events taken from the TimeBank corpus [6], a collection of news articles annotated in TimeML [5], a markup language for marking temporal events and relations in text. The `TLINK` tags at the end of the documents give the temporal relations of events as they are stated in the text.

The following example is taken from the `wsj_0555.tml` TimeML file. On extracting some of the `TLINK` tags and superposing their corresponding finite temporality strings together, we get the following possible timeline of events:

| | ei46 | | ei44, ei48 | | t12 | |
|---|---|---|---|---|---|---|

What is the probability of this timeline? That depends on the birth and death probabilities, and Figure 9 shows the probability space of this timeline, for all combinations of $p$ and $q$. We see that it is maximised when $p$ is around 0.2 and $q$ is close to 1. This timeline of events is most likely when events are

unlikely to be born, but once born, the die almost immediately. This is the case with this timeline, as events rarely occur in more than one interval.

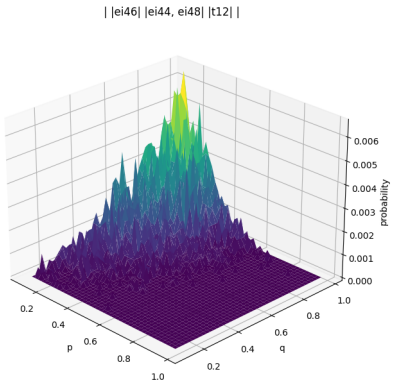



Figure 9: Likelihood plot of a real life timeline.

Altering the birth and death probabilities is a roundabout method to impose constraints on the strings generated by a ULD automaton. Choosing an optimal $p$ and $q$ value maximes the chance that a particular timeline is generated, however it is not certain that it will be generated. If we know already that events do not occur, we can impose a low birth probability however this does not rule out that a timeline with events co-occuring may be generated. What we would like is to ensure some gaurantee that every timeline generated by a ULD automaton will follow some constraint *i.e.* some two events are always related via *before*.

Pachet et al. [4] outline a process to impose constraints on finite length markov chains using a constraint satisfication approach to generate song melodies, and this procedure could be applied here in future work to generate timelines that follow a some predefined order.

## 4.4 Conclusion

This section has introduced and described in detail, a novel method that assigns probabilities to timelines, continuing on from the work of Fernando and Vogel [3]. We formalised the notion of a ULD automaton and showed its behaviour as we change its parameters, and applied the model to real world timelines of events. As an introductory piece, much work exists that can be added including the development of a constraint system to tailor the types of timelines that the ULD automaton generates.

# 5 Final Conclusions

This thesis has introduced finite temporality, implemented a framework for temporal reasoning, and considered the question of how likely are events to play out in a given way. Each piece reads like an individual article, but that is not to say they are disjointed and leave the reader with the fragmented understanding. A clear web is woven that brings the reader through while still maintaining the self-containedness of each section.

Section 1 reviews finite temporality and immediately introduces the idea of how we may extend [3] by simulation. Section 2 goes into detail about how to implement a framework to manipulate and represent finite temporality strings, and Section 3 uses this framework to simulate timelines, and answer the question of timeline probabilites that undercurrents this entire work.

We believe that we have been constructive in answering that question. Novel, finite state methods were developed and parameterized in such a way so that account for the nature of events; how likely they are to happen, and for how long.

However this is still a lot that can be done. Firstly, as described in Section 4, a method of ensuring constraints on the timelines generated by a ULD automaton using the work of Pachet et al. [4]. A consideration of unique birth and death probabilities for individual events is a similiar interesting direction to continue in. Timelines that consider events that do not all have the same birth probability would be better modelled using different probabilities for event births and deaths.

## 5.1 Applications beyond TimeML

Finally, the `python` library that is developed in Section 3 can have many applications beyond TimeML. In the sphere of Natural Language Processing, these strings can be used to produce timeline summaries of text documents, as we have shown with the TimeML examples. However we can use finite temporality strings to summarize any kind of temporal knowledge, serving as a sort-of log file of events. Of course this can be applied to any area in computing *e.g* network traffic logs, user-service interaction logs, etc. Their compact nature also provide an appealing visual representation of the logs they may represent.

# A Calculating Probabilities over Interval Names

Listing 1 below gives a python script that returns the probability of the 13 Allen relations over interval names for $n$ number of intervals. This probability is calculated by finding the proportion of timelines in the superposition of $n$ unconstrained event strings satisfy a given Allen relation.

```
from ftstring import FTString
import argparse as ap
from functools import reduce
from operator import and_
E = FTString([set(),{'a','b'},set()])
B = FTString([set(),{'a'},set(),{'b'},set()])
BI = FTString([set(),{'b'},set(),{'a'},set()])
...
F = FTString([set(),{'b'},{'a','b'},set()])
FI = FTString([set(),{'a'},{'a','b'},set()])
RELATIONS = [E,B,BI,D,DI,O,OI,M,MI,S,SI,F,FI]
NAMES = ['equals', 'before', 'after', 'during', 'contains',
         'overlaps','overlapped-by', 'meets', 'met-by',
         'starts', 'started-by','finishes', 'finished-by']
START = 0x61
def main(args):
    strings = []
    for i in range(args.n):
        strings.append(FTString([set(),{chr(START+i)},set()]))
    for r, n in zip(RELATIONS, NAMES):
        total = 0
        count = 0
        timelines = reduce(and_, strings)
        for t in timelines:
            total += 1
            if ~t.reduct({'a','b'}) == r:
                count += 1
        print('aRb ({0}) : {1}'.format(n, count/total))
```

Listing 1: Using the `ftstring` library to count proportions of Allen Relations

## B Approximating $A$-state Transition Probabilities

The python code given in Listing 2 simulates the ULD process of running a single ULD automaton for events in lockstep. By tracking the events that are living at each step, and building up the corresponding finite temporality string, where the living set $L_i$ at the $i$-th step is the $i$-th interval set $\alpha_i$ in the finite temporality string.

The string distributions given in Section 4 are the result of running the function `uld_process` in Listing 2 500,000 times for a some given $p$ and $q$ values, where the proportion of times a given string was generated with these specific birth and death probabilities is printed in bold black text beside the string's corresponding blue bar.

Of course, every string generated by the ULD automata process follows an $A$-story. By taking the bigram counts of state transitions, we can approximate the value for $P(s_j|s_i)$, the transition probability from state $s_i \in S_A$ to state $s_i \in S_A$. The process for doing so is described in detail subsequently.

Every call to `uld_process` generates a single string, $s$ which corresponds to a given unique $A$-story. The probability $P(s_j|s_i)$ is given by

$$P(s_j|s_i) = \frac{C(s_i, s_j)}{C(s_i)}.$$

We divide the total amount of times the $A$-state $s_j$ is preceded by $s_i$ by the total amount of times state $s_i$ appears. This gives an idea of how *often* a given transition is taken. These probabilities are then *strengthened* or *weakened* by the birth and death probabilites. Certain strings are produced more often or less often by the ULD automaton process, depending on $p$, and $q$. By updating the transition probabilities using every string generated, certain transitions will be strengthened *i.e.* have a larger probability, by the fact that some strings occur more frequently than others, and other transitions will be much less likely. With this in mind, we can recreate the transition matrix $M$ given previously where the entry $M_{ij}$ in the matrix represents $P(s_j|s_i)$.

The ULD process must be simulated many times to get accurate approximations of transition probabilities. Let $M_N$ be the approximate transtion matrix after $N$ runs of the ULD automaton, we have that $M_N$, $N \to \infty$ is the true transition probability matrix. Listing 3 shows the output of a program that calculates $M_N$ and we see as we increase $N$, $M_N$ gets closer to the true transition probabilities.

```python
from ftstring import FTString
import random

def uld_process(A,p,q):
    unborn = A
    living = set()
    dead = set()
    string = FTString()
    string.add(living)
    while dead != A:
        newly_living = set()
        newly_dead = set()
        for u in unborn:
            if random.random() < p:
                newly_living.add(u)
        for l in living:
            if random.random() < q:
                newly_dead.add(l)
        if len(newly_living) == 0 and len(newly_dead) == 0:
            continue
        unborn = unborn - newly_living
        living = living | newly_living
        living = living - newly_dead
        dead = dead | newly_dead
        string.add(living)

    return string
```

Listing 2: `python` code for simulating the ULD process for some set of events `A`. We run the single automaton for each event and use `python`'s random number generator to decide when an event lives and dies. The `ftstring` library described previously is used to build the corresponding finite temporality string.

```
p = 0.5 q = 0.5
N = 100
[[0.    0.34  0.32  0.    0.34  0.    0.    0.    0.   ]
 [0.    0.    0.    0.471 0.176 0.    0.353 0.    0.   ]
 [0.    0.    0.    0.    0.25  0.25  0.    0.5   0.   ]
 [0.    0.    0.    0.    0.    0.    1.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.542 0.25  0.208]
 [0.    0.    0.    0.    0.    0.    0.    1.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    1.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    1.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.   ]]
N = 1000
[[0.    0.33  0.323 0.    0.347 0.    0.    0.    0.   ]
 [0.    0.    0.    0.382 0.321 0.    0.297 0.    0.   ]
 [0.    0.    0.    0.    0.291 0.359 0.    0.35  0.   ]
 [0.    0.    0.    0.    0.    0.    1.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.355 0.313 0.333]
 [0.    0.    0.    0.    0.    0.    0.    1.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    1.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    1.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.   ]]
N = 10000
[[0.    0.328 0.333 0.    0.339 0.    0.    0.    0.   ]
 [0.    0.    0.    0.333 0.328 0.    0.339 0.    0.   ]
 [0.    0.    0.    0.    0.316 0.34  0.    0.344 0.   ]
 [0.    0.    0.    0.    0.    0.    1.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.338 0.328 0.334]
 [0.    0.    0.    0.    0.    0.    0.    1.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    1.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    1.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.   ]]
```

Listing 3: Approximation of $M_N$

## References


[1] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.

[2] Yvonne Brackbill and Hiram E. Fitzgerald. Stereotype temporal conditioning in infants. *Psychophysiology*, 9(6):569–577, 1972.

[3] Tim Fernando. and Carl Vogel. Prior probabilities of allen interval relations over finite orders. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence - Volume 2: NLPinAI,*, pages 952–961. INSTICC, SciTePress, 2019.

[4] Francois Pachet, Pierre Roy, and Gabriele Barbieri. Finite-length markov processes with constraints. pages 635–642, 01 2011.

[5] James Pustejovsky. *ISO-TimeML and the Annotation of Temporal Information*, pages 941–968. Handbook of Linguistic Annotation. Springer Netherlands, 2017.

[6] James Pustejovsky, Patrick Hanks, Roser Saurí, Andrew See, Rob Gaizauskas, Andrea Setzer, Dragomir Radev, Beth Sundheim, David Day, Lisa Ferro, and Marcia Lazo. The timebank corpus. *Proceedings of Corpus Linguistics*, 01 2003.

[7] L. Ramalho. *Fluent Python*. O'Reilly, 2015.

[8] David Woods and Tim Fernando. Improving string processing for temporal relations. In *Proceedings 14th Joint ACL - ISO Workshop on Interoperable Semantic Annotation*, pages 76–86, Santa Fe, New Mexico, USA, August 2018. Association for Computational Linguistics.

[9] David Woods, Tim Fernando, and Carl Vogel. Towards efficient string processing of annotated events. In *Proceedings 13th Joint ISO-ACL Workshop on Interoperable Semantic Annotation (isa-13)*, page 124, 2017.