NTNU
Norwegian University of
Science and Technology

Dynamical Simulation of Causal Sets

Bjarne Ådnanes Bergtun

Master of Science
Submission date: December 2016
Supervisor: Jan Myrheim, IFY

Norwegian University of Science and Technology
Department of Physics

Abstract

This thesis explores two dynamical schemes for the evolution of causal sets by a random Markov chain procedure. In many ways this thesis represents an absolute beginner's introduction to the subject of dynamical, random evolution of causal sets. Rather than relaying on one of the few ready-made solutions, the source code was written from scratch using C++. Consequently, this thesis also functions as a manual and documentation of the custom-made simulation package.

Additionally, the text gives an introduction to causal set theory, providing references to several useful papers for those who would want to study the subject further. In short, causal set theory is the vision that the four dimensional manifold structure of spacetime described by Einstein's general relativity can in actuality be found to arise from a discrete set of points whose only structure is the causal order relation, and that this more fundamental description can bring about the sought after marriage between general relativity and quantum field theory.

The dynamical principles studied in this thesis was found to lead to the creation of three and two layered causal sets, respectively.

Sammendrag

Denne masteroppgaven utforsker to dynamiske systemr for evolusjon av kausale sett ved en tilfeldig Markovkjede-basert prosedyre. På mange måter representer denne avhandlingen en absolutt nybegynners introduksjon til emnet dynamisk, tilfeldig evolusjon av kausale sett. I steded for å støtte seg på en av de få ferdiglagte løsningene som allerede finnes, ble kildekoden skrevet fra bunnen av ved hjelp C++. Følgelig fungere denne avhandlingenogså som en manual og som dokumentasjon for denne skreddersydde simulasjonspakken.

I tillegg gir teksten en grunnleggende innføring i «kausal mengde»-teori, og gir referanser til flere nyttige artikler for den som ønsker å studere dette emnet videre. Kort forklart er «kausal mengde»-teori visjonen om at den firedimensjonale mangfoldighetsbeskrivelsen av romtiden som Einsteins generelle relativitetsteori gir, i virkeligheten oppstår fra en diskret mengde punkter hvis eneste struktur er kausalordningen, samt at denne mer grunnleggende beskrivelsen kan føre til den ettertraktede sammensmeltingen av generell relativitetstoeri og kvantefeltteori.

De to dynamiske prinsippene studert i denne oppgaven ble funnet å føre til kausale sett med henholdsvis tre- og to temporære lag.

Acknowledgments

First and foremost I would like to thank my supervisor, professor Jan Myrheim. He accepted me as his master student, and helped guide me throughout this thesis. His gentle counsel and unfaltering optimism has been of great help to me, both in providing a fresh perspective and clear thought when mine was clouded. I would also like to thank my mom, who helped me proofreading early partial manuscripts—any present spelling mistakes, broken English or cases of unclear language is wholly on me!—and also provided much needed support and encouragements. Finally, I'm especially grateful for one of my dear friends in Oslo, who helped me remember that life is bigger than academic performances and accomplishments alone, and that there is always hope for the living; a second chance might come if one fails the first time around.

Contents

List of Figures

Chapter 1

Introduction

Ever since the problems of 'marrying' Einstein's theory of general relativity with quantum (field) theory became clear, the search for a unifying theory of quantum gravity has been ongoing. Many different approaches have been proposed and explored, but none have, so far, proven irrefutably successful. One of these proposals is causal set theory.

As the name suggests, this approach elevates the *causal structure* of classical spacetime to one of a very few properties with fundamental significance. Most other spacetime qualities, like curvature, dimension and metric, are considered emergent. Among the emergent properties is also *continuity*, spacetime being assumed to be discrete on (roughly) Planckian scales. The manifold nature of general relativity is thus envisioned to arise from a discrete causal *set*, from which this approach takes its name.

1.1 Project overview and motivation

One of the central challenges in the causal set approach is that while it is rather straight-forward to 'extract' a physically meaningful causal set from a continuous Lorentzian manifold, it is difficult to go the opposite way. Attempts at finding fundamental principles which lead to causal sets with 'reasonable' continuum properties have so far proven largely unsuccessful, the only partial exception [23] being the so-called *classical sequential growth models* of Rideout and Sorkin [21]. The exception is only *partial* because although these models leads to spacetimes with *some* desired properties—particularly if one subscribes to a 'bouncing' cosmology where the Universe have undergone several cycles of alternating Big Bangs and Big Crunches [1]—their continuum limit do *not* resemble four dimensional Minkowski spacetime [5].

Originally, this thesis was to be a simulational exploration of the most naïve of approaches to creating a random causal set: Holding the number of elements constant throughout the simulation, one causal relation is randomly added or removed for every simulation step, the probability distribution being uniform over the space of all relations which might be added or removed at

that particular step. Unbeknownst at the time to me and my supervisor, however, the most likely outcome of such a simulation—which turns out to be a negative result—was already known from a mathematical theorem found in 1975 [17], some three years before causal set theory was even introduced!

Unsurprisingly, I thus ended up with a negative result, essentially rediscovering what is known as *the entropy problem of causal set theory*. If that had been all, this thesis could simply had focused on a more thorough analysis of when and how the asymptotic theorem referenced above applies to small causets. However, it turned out that this was recently done by Henson, Rideout, Sorkin and Surya—see [13], a paper which will be referenced many times during this document—and in much more greater detail and depth than what I could possibly hope to achieve in the limited time I had left. Hence, finding another direction of study quickly was of the essence.

When trying to find a more fruitful approach, me and my supervisor stumbled upon some rather unphysical dynamical principles—to be further described in later chapters—which seem to lead to the creation of two-layered sets. These sets could potentially provide a causal set representation of a (limited) spatial spacetime sheet. Unfortunately, due to time running out, the question of whether the generated two-layered sets have a Euclidean continuum limit—as would be required if they are to represent a spatial spacetime sheet—remains unresolved. Their dimension, and whether this quantity is even well-defined, also remains unknown. However, preliminary results appear to suggest negative answers to all of these questions.

1.1.1 A short overview of the rest of this document

Chapter 2 provides a review of the necessary physical theory. The explanation of the produced simulation code is split into two chapters: chapter 3 lays out overarching principles both structural and implementational, while chapter 4 gives a more detailed overview of the project code, listing the different classes and their most important constituents and algorithms, as well as providing some practical usage hints. A review of the most central simulation findings is given in chapter 5, and discussed more closely in chapter 6. The conclusions of this discussion is given in chapter 7, together with some closing remarks and suggestions for further study. Finally, the complete source code is listed in appendix A, together with a handful of practical usage examples.

Chapter 2

Preliminaries

2.1 Causal set theory

Introduced independently by Myrheim [20] and 't Hooft [14] in 1978 and later formalized by Bombelli, Lee, Meyer and Sorkin in 1987 [4], *causal set theory* is one of the lesser-known proposed theories of quantum gravity; its more famous contestants being string theory and (to a lesser extent, at least to the general public) quantum loop gravity. Its central, defining idea is that on a fundamental level, the structure of spacetime is described by only two things: A locally finite set of points (spacetime events), $\mathcal{C}$, and a partial order relation, known as the *causal order-relation*, defined on the points of $\mathcal{C}$.

The causal order-relation, normally represented symbolically by '$\prec$', orders the points of $\mathcal{C}$ according to *causality* (i.e. time), hence the name. An expression like '$a \prec b$' is read as '*a precedes b*', and is equivalent with stating that something situated at a may, at least theoretically, influence something situated at b. The order relation $\prec$ is *partial* because it is entirely possible to have two distinct points $a \neq b$ such that $a \not\prec b$ and $b \not\prec a$ both are true, i.e. two points might be *unordered*. As a simple example, consider any two distinct points on this paper *at the exact same point in time*: No physical signal can travel from one spatial point to another in *no time at all*, and hence these spatially separated points cannot be causally ordered. A scaled up version of this example would be that nothing happening at Alpha Centauri *right now* can have any effect on what's happening *right now* at Earth, since Alpha Centauri is over four light years away, and relativity tells us that no signal can travel faster than the speed of light. Of course, by stating this, one is ignoring the highly non-local effects arising from quantum entanglement. However, these effects act directly on the non-observable wave function and does not lead to any observable signals—and hence no observable *effects*—traveling faster than the speed of light; see [9]. Thus, the conclusion remains unchanged.

As these examples hint at, the causal order-relation $\prec$ might be defined, from the viewpoint of relativity, by the familiar concept of light cones: Assuming $a \neq b$, $a \prec b$ if and only if b is a part of the *filled* future light cone

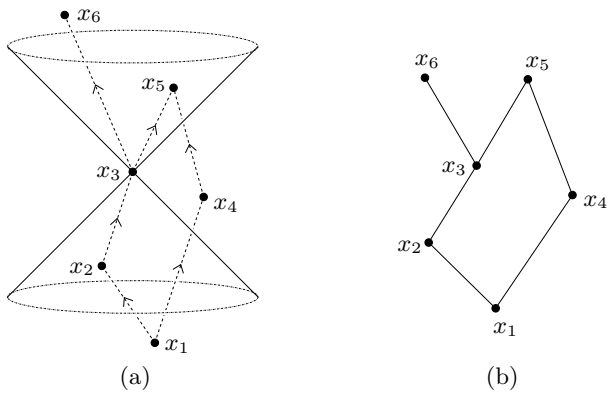


Figure 2.1: A schematic comparison between the spacetime of general relativity (a) and a partially ordered set of causal set theory (b). In (a), the spacetime points x_1 and x_2 is inside the past light cone of x_3, while x_5 and x_6 is inside its future light cone. The final point, x_4, is outside the light cone of x_3, but inside the light cones of x_1 and x_5. The partial order corresponding to this arrangement is illustrated in (b) using what is known as a *Hasse diagram*, where the points in $\mathcal{C}$ is represented by vertices and an upwards line segment from e.g. x_1 to x_2 means that $x_1 \prec x_2$ *and* that $\nexists\, y \in \mathcal{C}\ ; x_1 \prec y \prec x_2$. A crucial difference between (a) and (b) is that the manifold nature of spacetime underlying (a) is completely removed from (b)—geometrical characteristics like dimension and a metric tensor needs to be derived from the causal relations between x_1–x_6 rather than assumed. Consequently, while it might be natural to specify x_1–x_6 by their coordinates in some coordinate system in (a), such a set of coordinates must be *derived* from x_1–x_6 and their causal relations in (b).

of a (see figure 2.1). Alternatively, one might also allow $a = b$, this choice corresponding to whether one takes $\prec$ to be reflexive or not; see section 2.2.

From the vantage point of causal set theory, this identification is entirely backwards: Light cones are defined from the causal order-relation $\prec$, and not the other way around! More specific details on how this might come about will be given in section 2.2, where also a more precise mathematical definition of the order relation $\prec$ and the set $\mathcal{C}$ will be presented, together with some other useful definitions.

A small remark regarding terminology

When the points of $\mathcal{C}$ is labeled spacetime *events*, this should not be understood in a literal (everyday) manner: If, say, $x_a \in \mathcal{C}$, there still might not happen *anything at all* at x_a. Rather, x_a is simply one of the selected few (in the colloquial language often used when comparing infinite sets à la Georg Cantor) spacetime positions where the quantum fields describing matter and the additional forces besides gravity are defined. Thus, speaking *very* roughly, one might view the points of $\mathcal{C}$ as the 'pixels' of reality, provided that one is careful to expel the mental image of a 'regular lattice' that this description

might provoke. Additional complications arise from the fact that $\mathcal{C}$ is assumed to be subject to quantum fluctuations and/or quantum superposition; see section 2.1.2. Furthermore, this simple 'pixel'-image fails to incorporate vector- and tensor fields (and hence also matter!), since spacetime directions necessarily must be an emergent property in causal set theory. It is not known at present how to remedy this issue [23], although some suggestions have been made [16].

2.1.1 Motivations for causal set theory

Now, why would anyone propose that spacetime in actuality arises from this rather simple (from a mathematical point of view) and 'disembodied' description? As an example of this 'disembodiment', consider the fact that there's nothing *in between* the points of $\mathcal{C}$, nor do the points themselves have any (intrinsic) extension in time nor space. In fact, from the perspective of causal set theory, all of the spatial and temporal extension we *do* experience stems from the seemingly *even more* incorporeal causal order-relation: The only reason objects can be separated by (or indeed *occupy*) *time* is because there exists some spacetime points which *can* be causally ordered, while the only reason objects can be separated by (or occupy) *space* is because there are spacetime points which *cannot* be causally ordered. Hence, the causal-order relation is what gives the points of any causal set describing spacetime their volume!

So, what reasons, if any, is there to assume that this description of reality is correct? Why should spacetime be discrete, and why would one suspect that the causal order is more fundamental than, say, the metric?

As it turns out, there are several reasons for seriously considering a discrete spacetime:

First, a fundamental discreteness could potentially remove many of the troublesome infinities of quantum field theory, and provide justification for some of the techniques used to deal with them. While the infinities appearing in the quantum field description of the electromagnetic, strong and week force might be dealt with through the process of renormalization, such a scheme seems to fail for the gravitational force. A discrete spacetime could possibly help to eradicate these issues, and this was one of the main motivations which 't Hooft gave for the introduction of causal set theory in his lectures in 1978 [14]. However, this have not yet been proven, and therefore remains only a conjecture. See pages 14–15 of [23] for a more detailed commentary on this by Sumati Surya.

Second, causal set theory in particular could potentially give some physical *justification* for some of the renormalization techniques used when dealing with other forces besides gravity in quantum field theory. Specifically, causal set theory might provide *some* justification for so-called *dimensional regularization*, whereby one temporarily allow the dimension of spacetime to deviate from exactly 4 (even allowing for fractional values), since geometrical properties like dimension is statistical and emergent in this theory.

Other suggestive evidence might be gathered from the entropy of black holes, which—as have been shown in many different ways [2, 3, 10, 11]—might be identified as proportional to the surface area of the event horizon measured in Planck units. Given that the event horizon is just a (hyper-) surface in empty spacetime, distinguished from its surroundings solely by the fact that any causal relations across is exclusively one-directional—inwards—this result seems to suggest both a fundamental role of the causal order-relation, and that spacetime might be discrete on an approximately Planckian scale. This author was made aware of this argument through a public lecture held by Dowker [7], but it is also reminiscent of an argument made by Rafael D. Sorkin in [22]. An interesting extension of the black hole entropy result, easily applicable to causal set theory, might be found in [15]

Turning now to the question of why one might suspect the causal order-relation to be more fundamental than other spacetime properties, this may be traced back to different results in classical relativity—see [12] and [18] for examples—which shows that the causal structure determines the geometrical (manifold) structure, up to a conformal factor, of any future and past distinguishing spacetime of known dimensionality. In the words of Dowker: The causal order 'is a unifying concept ... [it] *unifies within itself* the other [spacetime] structures, up to a local rescaling of the metric.' [6].

2.1.2 Dynamics

As Dowker so eloquently remarks in her well-written 2013-article 'Introduction to causal sets and their phenomenology' [6], the static causets described above is not believed to be an adequate description of reality in full quantum gravity. In spite of this, the dynamics of causal sets and how they relate to matter is one of the least developed areas of causal set theory. Presently, a sum over histories approach appears to be viewed as the best candidate, but the aforementioned entropy problem—see section 2.3 below—presents difficulties which have not yet been completely overcome.

Thus, finding dynamical principles which leads to physically meaningful causal sets remains a central challenge in the causal set theory programme. The present 'toy example' which is closest to providing a satisfying dynamical scheme is the aforementioned classical sequential growth models of Rideout and Sorkin, where the causal sets are 'grown' by adding new points one by one according to certain statistical rules—see [21] for additional details. However, these models does not have classical spacetime as their continuum limit, and they will not be pursued further in this thesis.

The dynamical schemes explored in this thesis are by no means supposed to be anything more than toy models themselves, as they are essentially classical in their structure. Unfortunately, they did not lead to huge revelations, but maybe they can shed a little bit of light on the form which the correct principles should take. In the worst case scenario, they provide an example of what *not* to do.

2.2 Formal definitions

Unfortunately for the student, causal set theory literature has two competing standard ways of defining causal sets—*causets* for short—based on two different but closely related binary relations: the *irreflexive* causal order-relation $\prec$, and the *reflexive* causal order-relation $\preceq$. Even worse, both standards tends to use the same symbol—that is, '$\prec$' might be reflexive or irreflexive, depending upon convention (if '$\preceq$' is used, the reader can be fairly certain that the reflexive causal order-relation is what's meant). As should be clear by now, this text will employ the notation which is most reminiscent of the way the 'normal' order relation-pair $<$ and $\leq$ is used in mathematics, meaning that $\prec$ will be taken to be irreflexive, while $\preceq$ will be considered reflexive.

The *irreflexive* causal order-relation $\prec$ satisfies the following mathematical rules:

(1) Irreflexivity: $x \not\prec x$.

(2) Transitivity: If $x \prec y$ and $y \prec z$, then $x \prec z$.

(3) Antisymmetry: There's no x and y such that $x \prec y$ and $y \prec x$.

Strictly speaking, antisymmetry follows from irreflexivity and transitivity, but it is included here for comparison with the *reflexive* causal order-relation $\preceq$, which satisfies:

(1) Reflexivity: $x \preceq x$.

(2) Transitivity: If $x \preceq y$ and $y \preceq z$, then $x \preceq z$.

(3) Antisymmetry: If $x \preceq y$ and $y \preceq x$, then $x = y$.

Given these order relations, a causet is simply a set of points $\mathcal{C}$ partially ordered according to $\prec$ (alternatively $\preceq$)—meaning that there might be points $x, y \in \mathcal{C}$ such that $x \not\prec y \not\prec x$—and additionally satisfying the following finiteness-condition:

(4) For any $x, y \in \mathcal{C}$, the cardinality of the set $\{z \,;\, x \prec z \prec y\}$ is finite, i.e. any two points only has a finite number of points lying 'between' them.

Mathematically, $\preceq$ is what's known as a *partial order* ($\prec$ is an example of a *strict* partial order), and $(\mathcal{C}, \preceq)$ might likewise be described as a *locally finite* partially ordered set—or *poset* for short.

In addition to the above axioms, Bombelli, Lee, Meyer and Sorkin introduced the notion of an *embedding*, which clarifies how to know if a specific continuum $(\mathcal{M}, g)$ is a good approximation of a given causet C. An *embedding* is any function $\Phi\colon \mathcal{C} \to (\mathcal{M}, g)$. If the order relation in $\mathcal{C}$ has a one-to-one-correspondence with the causal order induced on $\Phi(\mathcal{C})$, Φ is said to be *order preserving*. If the image of an order preserving embedding Φ, $\Phi(\mathcal{C}) \subset \mathcal{M}$ is a high probability Poisson distribution in $\mathcal{M}$, Φ is known as a *faithful*

embedding [23]. The condition that a specific continuum $(\mathcal{M}, g)$ is a good approximation of a given causet C is then that C can be faithfully embedded into $(\mathcal{M}, g)$.

This leads to one of the fundamental conjectures in causal set theory, still unproven for the general case: *If a causal set $\mathcal{C}$ can be faithfully embedded at the same density into two distinct spacetimes $(\mathcal{M}_1, g_1)$ and $(\mathcal{M}_2, g_2)$, then said spacetimes differ only at the volume scales of the chosen Poisson distribution.*

2.2.1 Some useful terminology

When discussing and comparing different causets in later chapters, some more descriptive terminology will prove useful. Most of the following terminology should be fairly familiar for anyone working in the field, but a few terms useful for discussing clearly layered sets—of which one will see an example of already in section 2.3—are 'home-brewed'.

A *link* is any relation which cannot be deduced by transitivity, i.e. if $x \prec y$ and $\nexists\, z \in \mathcal{C}$ such that $x \prec z \prec y$, the relation $x \prec y$ is known as a *link*. Given this, one might now properly introduce the very useful graphical representation of a causal set known as a *Hasse diagram*, already employed and briefly explained in figure 2.1. A Hasse diagram depicting a given causal set $\mathcal{C}$—where one is using '$\mathcal{C}$' as a shorthand for $(\mathcal{C}, \prec)$, a useful abbreviation which will be employed whenever the chance for confusion is minimal—represents the points of $\mathcal{C}$ by dots, and the links of $\mathcal{C}$ by line segments connecting these dots. The diagrams are 'ordered according to gravity', meaning that if e.g. $x, y \in \mathcal{C}$ is such that $x \prec y$, then the dot representing x is drawn somewhere below the dot representing y.

A *chain* from x to y of *length* n is a set of $n+1$ points $x_0, \ldots, x_n$ such that $x = x_0 \prec x_1 \prec \cdots \prec x_n = y$. Thus, any totally ordered set is a chain. A chain between x and y is called *maximal* if there's no chains of greater length between x and y. It should come as no surprise that maximal chains are the causal set realization of timelike geodesics in Minkowski space.

An *antichain* is any set of points which are not causally related to one another. If two incomparable elements have at least one link in common, the antichain consisting of only these two elements will, in this document, be labeled an *antilink*. Reminiscent of two-point chains, antilinks is assigned a length of 1. Expanding on this, a *connected* antichain is any antichain consisting entirely of interlinked antilinks, its length corresponding to the number of unique antilinks. Two points x and y will be said to be *anticonnected* if there exists at least one connected antichain which contains both of them. Analogous in a certain sense to a maximal chain, a *minimal* connected antichain between x and y will be the *shortest* connected antichain containing x and y.

Returning to more established terminology, the *height* of a causet $\mathcal{C}$ is the length of the longest maximal chain(s) in $\mathcal{C}$. Equally intuitively will any point in $\mathcal{C}$ which has no points preceding it be labeled a *minimal* point, while any point which does not precede any other will be labeled *maximal*.

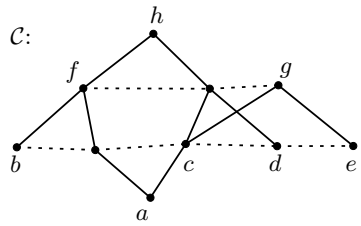


Figure 2.2: A small causet $\mathcal{C}$ illustrating some of the definitions of section 2.2. As can be seen from the figure, $\mathcal{C}$ has *two* longest maximal chains, both connecting a and h, and both having a length of 4. Hence, $\mathcal{C}$ has height 4. The pair b, g is an example of an unconnected antichain, while the roughly horizontal dashed lines show all but one antilinks of $\mathcal{C}$, the one between c and e being suppressed due to readability. Noteworthy are the longest minimal connected antichains—of which there are three: one between b and d, one between b and e, and one between f and g—which illustrate that the shared links connecting the points of a connected antichain need not all be either exclusively 'above' or exclusively 'below' the antichain. The maximal points of $\mathcal{C}$ are h and g, while b, a, d and e are minimal.

Figure 2.2 sums up most of the terminology introduced so far.

Useful sets from general relativity

Given a causal set $\mathcal{C}$ and a point $x \in \mathcal{C}$, the *exclusive future of* x is denoted by

$$T^+(x) \equiv \{z \in \mathcal{C} \,;\, x \prec z\}, \tag{2.1}$$

while the *exclusive past of* x is given by

$$T^-(x) \equiv \{z \in \mathcal{C} \,;\, z \prec x\}. \tag{2.2}$$

Given these definitions, one might now introduce two sets corresponding to the links of x, one for the future links $(x \prec \ldots)$—or 'upwards' links—given by

$$L^+(x) \equiv \{z \in T^+(x) \,;\, \nexists\, y : x \prec y \prec z\}, \tag{2.3}$$

and one for the past links $(\ldots \prec x)$—or 'downwards' links—denoted by

$$L^-(x) \equiv \{z \in T^-(x) \,;\, \nexists\, y : z \prec y \prec x\}. \tag{2.4}$$

Finally, the *Alexandrov interval* $A(x, y)$ is defined as the intersection of the *inclusive future* of x and the inclusive *past* of y:

$$A(x, y) \equiv \{z \in \mathcal{C} \,;\, x \preceq z \preceq y\}. \tag{2.5}$$

2.2.2 The Myrheim–Meyer dimension

In his 1978-preprint [20], Myrheim introduced the so-called *ordering fraction* f, as a dimension estimator for causal sets. In Myrheim's original definition, f

Table 2.1: Values for the ordering fraction f of an Alexandrov set in flat Minkowski spacetime of d dimensions. Copied from Myrheim's preprint from 1978 [20].

d	Ordering fraction, f
1	1
2	1/2
3	8/35
4	1/10

is the fraction of comparable (i.e. ordered) pairs to the total number of pairs in an Alexandrov interval $A(x, y)$. Using the volume–number correspondence which any discrete theory of spacetime must entail, one can then calculate the ordering fraction for flat Minkowski spacetime in different dimensions—see table 2.1 for some reference values, copied from Myrheim's preprint [20]. The resulting values for f might then be compared to the ordering fraction of a causet to get an estimate of its dimension. This procedure was studied in greater detail and generalized by Meyer in 1989—see [19]—and the dimension estimate thus obtained is therefore known today as the *Myrheim–Meyer dimension*.

In this paper, the definition of f will be extended to include *any* causal set $\mathcal{C}$, irrespective of whether said set is an Alexandrov interval or not. Using this definition, f might simply be written as

$$f \equiv \frac{R}{R_{\max}}, \tag{2.6}$$

where R is the number of order relations in the causet under consideration, and $R_{\max}$ is the number of order relations in a *totally ordered* causet with equally many points as the one under consideration. Hence, given a causet of n points, (2.6) becomes

$$f(R, n) \equiv \frac{R}{\binom{n}{2}} = \frac{2R}{n(n-1)}. \tag{2.7}$$

Generally, conventions seem to differ, but this usage is in agreement with the one employed by Henson et al. in [13], a paper which will serve as an important comparison for the first half of this thesis.

2.3 The Kleitman–Rothschild theorem

Presented in a paper by Kleitman and Rothschild in 1975 [17], the Kleitman–Rothschild theorem forms the basis of what has come to be known as 'the entropy problem' of causal set theory: For increasing n, Ω_n—the set of all causal sets with exactly n elements—becomes increasingly dominated by causets which does *not* approximate classical spacetime manifolds in the

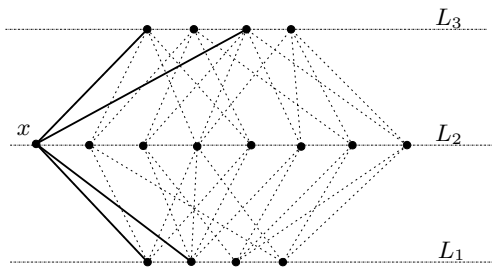


Figure 2.3: A small Kleitman–Rothschild-type set with $n = 16$ points. The middle layer L_2 contains approximately (in this case, *exactly*) twice as many points as the two other layers L_1 and L_3. Additionally, every point in any layer L_i precedes approximately half of the points in the 'above' layer L_{i+1} and/or is preceded by half of the points in the 'below' layer L_{i-1} (depending upon whether said layers exist or not). As an illustration of this last property, the causal relations of x, a point in the middle layer L_2, is highlighted.

slightest! Hence, for increasing n physically meaningful causets become increasingly rare.

Specifically, the Kleitman–Rothschild theorem reveals that Ω_n becomes asymptotically dominated by sets with a natural division into three subsets—or 'layers'—L_1, L_2, L_3, the division being done according to temporal position. The three layers of these 'Kleitman–Rothschild sets' additionally fulfill (see figure 2.3):

(1) The middle layer L_2 contains approximately the same number of elements as the two other layers L_1 and L_3 combined.

(2) The elements in the two top layers L_2 and L_3 are preceded by approximately half of the elements in the layer immediately below.

(3) Similarly, the two bottom layers L_1 and L_2 precede approximately half of the elements in the layer immediately above.

Using these properties, it is easy to find that the ordering fraction of a Kleitman–Rothschild set should tend to approximately 3/8 for large n. However, a closer analysis of the Kleitman–Rothschild theorem performed by Henson et al. in their exploratory paper—of which much more will be said before this thesis is over—revealed that the asymptotic ordering fraction varies between 1/4 and 3/8, averaging to about 1/3, but with 3/8 still being asymptotically favored due to a slowly divergent peak [13].

Chapter 3

Overarching implementation principles

3.1 General design principles

Contrary to what the title of this chapter might suggest, the algorithmic implementation of the project underlying this thesis was far more organic than principle-driven. Thus, while C++ was chosen as the project's programming language due to its speed and flexibility, the actual *implementation* prioritized 'readability' (in a broad sense) and reliability over performance—especially in the early stages of the project. Although this is a natural—and to a certain degree *wanted*—prioritization, it means that a more 'abstracted' code might see significant performance gains over the one used in this project.

An optimization which has already been done, is the removal of 'sanity checks' on function input. This greatly improves runtime at the expense of safety, since this means that the code has to be written such that non-sensible input never occurs.

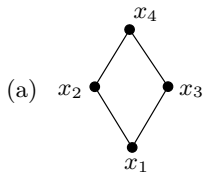
In this chapter the fundamental principles and design goals underlaying the project code will be presented. A more detailed overview of the code can be found in chapter 4, while the raw C++ is included in appendix A.

3.2 Some words about code structure

Before some of the more concrete principles are explained, it will be beneficial to review certain core aspects of how a causet is represented in the project code. Additional details might be found in section 4.1 and in appendix A, section A.1. Incidentally, this provides an example of how readability trumps performance in the chosen implementation, as 'dual bookkeeping' is used extensively.

The relations between points in $\mathcal{C}_n$ is stored in a trinary matrix, specifying whether a point precedes (1), is preceded by (-1), or is incomparable to (0), another point—see figure 3.1a,b. This is itself a superfluous object, since

(a)



(b)

	x_1	x_2	x_3	x_4
x_1	0	1	1	1
x_2	−1	0	0	1
x_3	−1	0	0	1
x_4	−1	−1	−1	0

(c)

	$L^+(x)$	$L^-(x)$
x_1	x_2, x_3	$-$
x_2	x_4	x_1
x_3	x_4	x_1
x_4	$-$	x_2, x_3

Figure 3.1: Different ways of representing the same causal set $(\mathcal{C}_4, \prec)$: Hasse diagram (a), relation matrix (b) and link lists (c). As can be seen from the diagonal, the matrix representation (c) is based on the irreflexive causal order-relation $\prec$ rather than the reflexive $\preceq$. The close relationship between the Hasse representation (a) and the link representation (b) should also be evident.

a binary matrix would be sufficient, analogous to how $b > a$ is redundant information given $a < b$. Additionally, the relations between points are stored in the form of dual lists of links related to the individual points, one for future links [i.e. $L^+(x)$; see (2.3)] and one for past links [i.e. $L^-(x)$; see (2.4)]—see figure 3.1a,c. Again, one list per point would suffice, since the up-links might be inferred from the down links, or vice versa. While this does mean that the relationship data is repeated four times, this solution has some conceptual advantages, promoting the readability of the created code.

These advantages stems from the fact that it differs from one context to another which representation is the most natural to work with, since the two representations focuses on different characteristics of a causal set $(\mathcal{C}_n, \prec)$. The relation matrix exhibits the transitive nature of $\prec$, and provides a quick and easy way to figure out if two points are related, and if so, which is above the other. The link-lists, on the other hand, focuses more on the individual points of $\mathcal{C}_n$, making it easy to see which points are temporarily close to one another. An additional conceptual advantage with the link representation is its close relationship with Hasse diagrams, introduced in section 2.2. Comparing figure 3.1a and -c, one sees that the link representation might be thought of as a systematic recording of the links in a Hasse diagram. Transitioning from one to the other is therefore straight-forward, at least in principle. (Huge, complicated causal sets might prove hard to draw in a clear and transparent manner).

Albeit conceptually useful, this dual bookkeeping is by no means necessary, and one could conceivably make the code run faster by using one representation exclusively. Due to the somewhat cumbersome nature of the link lists, which will be partially revealed in section 3.3, this author suspects that the relation-matrix representation would be best suited for such a solution. In regards to the somewhat milder double bookkeeping involved in listing both 'upwards'

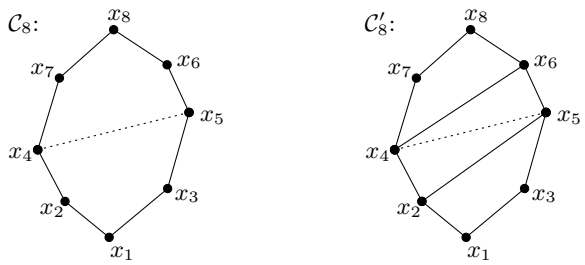
and 'downwards' relations, however, one will see, in chapter 4, that this might actually have some practical benefits, owing to its temporally symmetric structure.

3.3 How to add or remove only one order relation at a time

One of the fundamental design goals of the project code was that one should be able to add or remove only *one* order relation at a time. Given the axiom of transitivity, it is easy to see that the only relations which *might* be added or removed without also changing others, are links. As a simple example, consider the totally ordered causet $\{x, y, z\}$ where $x \prec y \prec z$. If, for example, one tries to remove only the relation $x \prec z$, one must reject the axiom of transitivity to succeed in removing only *one* order relation, since $x \prec y$ and $y \prec z$ otherwise implies $x \prec z$. On the other hand, and possibly somewhat linguistically confusing, the inclusion (or removal) of a single order relation respecting the axiom of transitivity does not necessarily correlate with a similar change in the number of *links*—the *total* number of order relations are increased (or decreased) by 1, but the number of links might increase, decrease or remain unchanged. In the example just considered, the removal of the single relation $y \prec z$—itself a link, as it should be—leaves the number of links unchanged, since now $x \prec y, z$. This linguistic quagmire could be avoided by rejecting the axiom of transitivity, but doing so would mean to consider *multidirected sets* rather than partially ordered sets as providing the fundamental structure of reality. Such a modification of causal set theory have been argued for by at least one author [8], but will not be pursued further here. Hence, for the remainder of this text if one considers the removal or addition of a singular order relation, said order relation will be assumed to be a link.

As hinted at in the end of the previous section, the requirement of being able to change a single order relation at a time, as it turns out, provides a helpful example for familiarizing oneself with some of the differences between the two causet representations used in the project code. In the matrix representation it is completely trivial to *remove* a link, but a little bit of thought is needed to deduce if any given relation actually *is* a link. Similarly, the question of whether one can *add* a causal relation without simultaneously introducing additional relations is also not immediately straight-forward. As for the link representation, it is completely trivial to determine if a relation is a link, but the operation of adding or removing a relation require a little bit of thought.

Section 4.1.4 gives a description of the algorithms used in the project code to implement the principles described below, while section A.1.4 in appendix A gives the raw C++ code itself.



(a) (b)

Figure 3.2: Given the causal set $\mathcal{C}_8$, shown in (a), one cannot introduce the link $x_4 \prec x_5$ without also adding new relations, since $x_4 \prec x_5$ by transitivity also implies $x_2 \prec x_5$, $x_2 \prec x_6$ and $x_4 \prec x_6$. If, on the other hand, the relations $x_2 \prec x_5$, $x_2 \prec x_6$ and $x_4 \prec x_6$ are already present—as in $\mathcal{C}'_8$, shown in (b)—one might freely introduce the link $x_4 \prec x_5$. Doing so leads one to the causal set $\mathcal{C}''_8$, obtained from (a) by replacing the dashed line with a solid line.

3.3.1 Adding or removing one order relation in the Hasse/link representation

A simple example

Reverting for a moment back to the familiar Hasse representation, consider now the causal set $\mathcal{C}_8$, depicted in figure 3.2a. As pointed out in the caption, one cannot add the relation $x_4 \prec x_5$ to $\mathcal{C}_8$ without also introducing other relations. If, on the other hand, these additional relations are already present, as in $\mathcal{C}'_8$, shown in figure 3.2b, one *may* add the relation $x_4 \prec x_5$ to end up with the causal set $\mathcal{C}''_8$, obtainable from figure 3.2a by replacing the dashed line with a solid one.

Looking at this example from the opposite, 'destructive' angle, one sees how, in the Hasse representation, removing the single relation $x_4 \prec x_5$ (and its corresponding link) from $\mathcal{C}''_8$, leads to the introduction of *two new* links in $\mathcal{C}'_8$, namely $x_2 \prec x_5$ and $x_4 \prec x_6$, as shown in figure 3.2b. Consequently, it is not true, in general, that adding or removing one causal relation corresponds to the addition or subtraction of only one link.

The general case

By contemplating this example somewhat more, one realizes that the criterion for when the relation $x \prec y$ can be added to a general causet $\mathcal{C}$, with $x, y \in \mathcal{C}$, without introducing additional relations, is that the exclusive *future* of y must be wholly casually connected to the inclusive future of x, and, likewise, the exclusive *past* of x must be wholly causally connected to the inclusive past of y. Or, using symbols, the link $x \prec y$ might be individually added to $\mathcal{C}$ if and only if

$$T^+(x) \cap T^+(y) = T^+(y) \tag{3.1a}$$

$$T^-(x) \cap T^-(y) = T^-(x), \tag{3.1b}$$

where eg. $T^+(x)$ is the exclusive future of x, and $T^-(x)$ the exclusive past; see (2.1) and (2.2) in section 2.2.

However, due to transitivity, it is enough to compare the future links of y with the exclusive future of x to determine whether (3.1a) is fulfilled. A similar comment applies to (3.1b). Hence, a more 'economical' way of expressing criterion (3.1) is

$$L^+(y) \subseteq T^+(x) \tag{3.2a}$$

$$L^-(x) \subseteq T^-(y), \tag{3.2b}$$

where $L^+(y)$ is the set of points which y links up to, and $L^-(x)$ is the set of points linking up to x; see (2.3) and (2.4) in section 2.2.

Following the foregoing discussion, one realizes that criterion (3.2), or equivalently criterion (3.1), gives a 'checklist' for deciding which links might need to be added to the description of a general causal set $\mathcal{C}$, with $x \prec y \in \mathcal{C}$, if the link $x \prec y$ is removed from $\mathcal{C}$. This 'checklist' becomes even clearer in figure 3.3, which gives a schematic Hasse description of (3.2).

The question of whether a relation can be *removed*, on the other hand, is, as already pointed out, completely trivial in the Hasse/link representation, as this is simply a question of whether said relation is a link.

3.3.2 Adding or removing one order relation in the matrix representation

Now, what does all of this look like in the matrix representation? Going back to the causal sets considered in figure 3.2, one sees that the matrix representation of $\mathcal{C}_8$ becomes

$$\mathcal{C}_8 \mathrel{\widehat{=}} \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \bar{1} & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ \bar{1} & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ \bar{1} & \bar{1} & 0 & 0 & 0 & 0 & 1 & 1 \\ \bar{1} & 0 & \bar{1} & 0 & 0 & 1 & 0 & 1 \\ \bar{1} & 0 & \bar{1} & 0 & \bar{1} & 0 & 0 & 1 \\ \bar{1} & \bar{1} & 0 & \bar{1} & 0 & 0 & 0 & 1 \\ \bar{1} & \bar{1} & \bar{1} & \bar{1} & \bar{1} & \bar{1} & \bar{1} & 0 \end{bmatrix}, \tag{3.3}$$

where the notation $\bar{1} \mathrel{\widehat{=}} -1$ is used in an effort to aid readability and highlight the inherent antisymmetry of the (trinary *and* irreflexive) matrix representation. As stated earlier, the relation $x_4 \prec x_5$ cannot be individually added to $\mathcal{C}_8$; one would have to also include $x_2 \prec x_5$, $x_2 \prec x_6$ and $x_4 \prec x_6$ if $\mathcal{C}_8$ should be kept self-consistent (since $\prec$ is transitive). Providing a contrast to this is

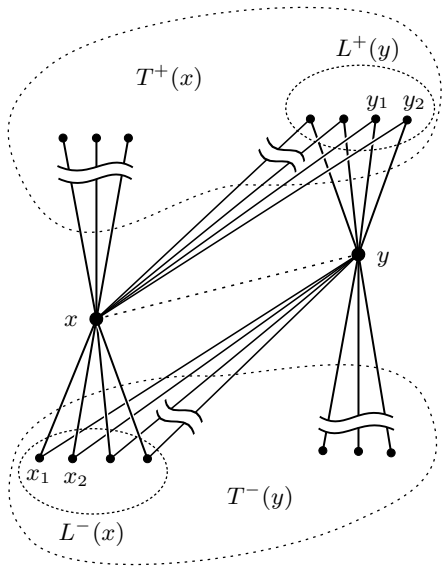


Figure 3.3: A schematic Hasse diagram generalizing the insights gathered from contemplating the examples given in figure 3.2. While $T^+(y) \setminus L^+(y)$ and $T^-(x) \setminus L^-(x)$ are not explicitly shown, it is assumed that $T^+(y) \subseteq T^+(x)$ and $T^-(x) \subseteq T^-(y)$, in accordance with (3.1). The diagram can be interpreted both additively and subtractively: In the additive sense, it shows the conditions under which one might add the link $x \prec y$ to a causal set $\mathcal{C}$, with $x, y \in \mathcal{C}$. Note that *if* $x \prec y$ is added to $\mathcal{C}$, the links $x \prec y_1, y_2$ and $x_1, x_2 \prec y$ will be implied by transitivity, and hence need to be removed from the Hasse diagram (cf. the transition from $\mathcal{C}'$ to $\mathcal{C}''$ detailed in the caption of figure 3.2). Understood from the subtractive point of view, the figure reveals the relations needed to be re-established (in the Hasse diagram representation) if one *removes* the link $x \prec y$ from $\mathcal{C}$. Specifically, the relations needed to be re-instated are the links $x \prec y_1, y_2$ and $x_1, x_2 \prec y$.

$\mathcal{C}'_8$, which has the matrix representation

$$\mathcal{C}'_8 \mathrel{\widehat{=}} \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \bar{1} & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ \bar{1} & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ \bar{1} & \bar{1} & 0 & 0 & 0 & 1 & 1 & 1 \\ \bar{1} & \bar{1} & \bar{1} & 0 & 0 & 1 & 0 & 1 \\ \bar{1} & \bar{1} & \bar{1} & \bar{1} & \bar{1} & 0 & 0 & 1 \\ \bar{1} & \bar{1} & 0 & \bar{1} & 0 & 0 & 0 & 1 \\ \bar{1} & \bar{1} & \bar{1} & \bar{1} & \bar{1} & \bar{1} & \bar{1} & 0 \end{bmatrix} . \tag{3.4}$$

Here, one might indeed add the relation $x_4 \prec x_5$ without introducing other new relations.

How then, does criterion (3.1)/(3.2) translate into the matrix representation, both generally and in this particular example?

First, one notes that (3.1) is the more appropriate version of the criterion to use when looking at things from the matrix perspective, since the exclusive past and/or future of a specific point can be read of directly from that point's corresponding row in the relation matrix (or column, since the matrix representation is antisymmetric). Links, on the other hand, need to be deduced by cross-referencing several rows/columns.

Returning then to the explicit example considered above, one may start by looking at the row corresponding to x_4 in (3.3), which is the fourth row as counted from above, the labels being implicit. It follows from the explanation given in section 3.2 and figure 3.1, that the collection of entries (x_4, z) which have a '1' in them must correspond to the exclusive future of x_4, $T^+(x_4)$, while the collection of entries with '-1' in them correspond to $T^-(x_4)$, the exclusive past of x_4. Using this identification, one sees that (3.1) simply translates into comparing the rows corresponding to x_4 and x_5, checking whether the row of x_4 has a '1' in all the columns that x_5 do, and whether x_5 has a '$\bar{1}$' in all the columns that x_4 do (excess '1'-s in x_4 and '$\bar{1}$'-s in x_5 is of no consequence). Doing this for the matrices in (3.3) and (3.4), one sees that (3.1), with $x = x_4$ and $y = x_5$, is *not* fulfilled for $\mathcal{C}_8$, but is indeed satisfied for $\mathcal{C}'_8$, as it should be.

As to how to recognize the links in (3.3) and (3.4), one sees that the links of any given x can be determined by recursively iterating trough the different elements in $T^(x)$, looking for relations which are not also implied by transitivity: The relation $x_1 \prec x_2$ must be a link because $\nexists\, x \neq x_2 \in T^+(x_1)$ such that $x_1 \prec x \prec x_2$, and so on.

The general case

This procedure generalizes to arbitrary x, y belonging to an arbitrary causal set $\mathcal{C}$ in a straight-forward manner.

One note of caution, however: In (3.3) and (3.4),the matrix rows take the particularly ordered form

$$\begin{bmatrix} T^-(x) & 0 & T^+(x) \end{bmatrix},$$

for any point $x \in \mathcal{C}_8, \mathcal{C}'_8$, the '0' being part of the diagonal; provided that one takes the 'matrix definition' of $T^{\pm}(x)$ to be of length $8 - x$ and x, respectively, effectively ignoring the 'trailing' zeros before/after the diagonal. This is *not* a general future, and arises solely because C_8 and C'_8 are what's known as *naturally labeled*, meaning that $x \prec y \iff x < y$—where the first comparison is to be understood as comparing the *points* $x, y \in \mathcal{C}$, and the second as comparing the (numerical) *labels* $x, y \in \mathbb{Z}$.

Lastly, since the matrix representation is (anti-) symmetric, columns and rows in the foregoing discussion may be interchanged.

3.4 Dynamical principles

The two dynamical principles investigated in this thesis has a few things in common. Firstly, in both of the dynamical systems considered, the *ordering fraction* plays a central role in identifying when the sought after evolution has been brought to fullness. Secondly, and as already hinted at above, both of the systems described here restrict the space of possible dynamical transitions to only include those which does not change more than *one* order relation at a time. (Transitions which would lead to a 'causet' in conflict with the requirements stated in section 2.2 is also, unsurprisingly, prohibited). As can be seen from the results of the previous section, this leads to a reduction in both the number of relations which can be removed—they must be links—and the number of relations which might be added—they must fulfill (3.2), as illustrated in figure 3.3. However, only one of the dynamical schemes utilized in this thesis actually allow relations to be added. This scheme is the first to be presented below.

3.4.1 Naïve dynamics

The dynamical scheme which this document will label as 'naïve'—not because it is inherently inadequate or 'too simple'—it is not—but because it is among the most natural and simplest schemes to devise—was briefly described already in section 1.1: The number of causet points is held constant throughout the simulation, while a single randomly chosen order relation is added or removed (according to whichever applies) at every simulation step. The transitions are chosen using a uniform probability distribution, which assigns an equal weight to any transition—regardless of type—in the space of momentarily available transitions. The elements of this space are continuously updated as relations are added or subtracted. Hence, the probability of whether a relation might be added or removed at any given time depends only on the ratio between the number of relations that might be added or removed at said moment.

It should be relatively easy to see that *any* causet might be obtained by this procedure, but that some are more likely than others. A completely ordered causet is especially unlikely, and easily destroyed if momentarily obtained. The same goes for a completely unordered causet. Here then should be a practical and simple realization of the closing speculations found in Myrheim's 1978-preprint:

> It is interesting to speculate on the possibility that there is a statistical basis for the dimension of space-time [...]. In fact, the linear, or total, ordering relation on a given set is unique, and represents an extreme case when all possible partial orderings are considered. It is therefore a highly improbable situation, in some vague sense. The other extreme, $f = 0$, is an equally unique case, and equally improbable. In a theory of statistical geometry [i.e. *in causal set theory*] one can imagine that there ought to be a 'most

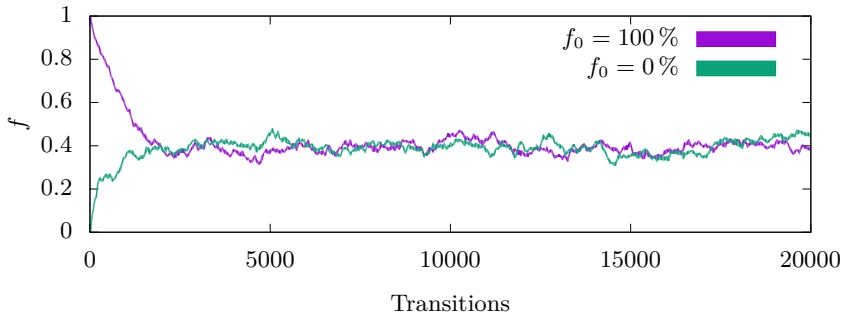


Figure 3.4: Naïve evolution employed on a small causet with 32 points illustrating the initial idea behind the equilibrium criterion. If the naïve dynamical scheme is 'physically well behaved' the initial state of the causet should become more and more unimportant as the simulation time increases. Specifically, a totally ordered causet should, in the end, have the same ordering fraction as a totally unordered causet (barring statistical fluctuations). Thus, by comparing the evolution of these two very special causets starting at opposite extremes, it should—in theory—be possible to directly determine whether an equilibrium exists, and if so, examine its stability. Note that the printed graph shown here is somewhat coarse grained; it shows the maximum and minimum values for every 40 transition cycles. See sections 5.1 and 6.1 for a closer analysis.

> probable value' of f, intermediate between 0 and 1, and a 'most probable dimension' of space-time. [20]

How is this 'most probable causet' to be determined? It is easy to imagine that there should exists some kind of 'equilibrium' where the probability for adding an order relation is approximately equal to the probability of removing one.

As order relations are added and removed, the ordering fraction f changes to reflect this. Hence, if such an equilibrium exists, one should find that the graph of f 'flattens out' once the causet approaches this state of balanced probability. Moreover, if this 'equilibrium state' really is unique (in the statistical, average sense), it should be independent of the initial ordering state.

Thus, one might compare the evolution of two causets with the same number of points—one initially totally ordered, and one initially totally unordered—to determine whether a stable, well-defined equilibrium state exists. This forms the basics for the 'equilibrium criterion' employed for naïve dynamics in this project. See figure 3.4 for a practical illustration. If the two graphs merge into a pair of statistically identical horizontal lines, as seems to be the case in the above figure, then a well-defined equilibrium state exists and the causets will be said to have reached 'thermalisation'.

The dynamical principles used by Henson et al.

The exploratory paper by Henson et al. [13], which examines the onset of the asymptotic behavior described by the Kleitman–Rothschild theorem, has been mentioned a few times earlier in this text. As hinted at in section 1.1, the dynamical principles employed by Henson et al. closely resembles the scheme laid out above. However, their conceptual starting point and overall approach is slightly different, leading to some difference in details.

Unlike the algorithms which will be described later in this thesis, Henson et al. uses a Monte Carlo-type approach, where transitions are chosen from an extended transition space also including elements with zero probability, to evolve their causets. They require that the evolution should be independent of the foregoing history—i.e. it should be a *Markov chain*, a requirement that is satisfied for both of the dynamical systems considered in this thesis—and they employ a uniform probability distribution, just like the above scheme. Moreover, they require that any order state should be repeatably obtainable regardless of initially chosen state—i.e. the Markov chain should be *ergodic*—and that detailed balance should be satisfied, meaning that for any two order states A and B,

$$P_A P_{A\to B} = P_B P_{B\to A}, \tag{3.5}$$

where P_X is the probability of state X and $P_{X\to Y}$ the transition probability from X to Y. Since the probability distribution is uniform, $P_A = P_B$ for any A, B.

As described above, naïve evolution is ergodic. However, since transitions are chosen only from the reduced space of momentarily allowable transitions, it does not generally fulfill detailed balance: Looking back to the simple example considered at the beginning of section 3.3, the initially totally ordered causet $\mathcal{C} = \{x, y, z\}$, with $x \prec y \prec z$, one sees that the probability of transitioning to the reduced state $x \prec y, z$, with $y \not\prec z$, is equal to $1/2$, since no relations can be added to $\mathcal{C}$, but either one of the links $x \prec y$ and $y \prec z$ can be removed. The probability of the *reverse* transition, on the other hand, is $1/4$, since now there is still two links which can be removed—$x \prec y$ and $x \prec z$—but also two links which can be added. Labeling the totally ordered state by C_3 and the reduced state by C_3', one thus has

$$P_{C_3\to C_3'} = \frac{1}{2} \neq P_{C_3'\to C_3} = \frac{1}{4}, \tag{3.6}$$

provided that the points are distinguishable, as is the case for all dynamical systems considered in this thesis. Henson et al. avoids this result by using a Monte Carlo approach, whereby every transition $A \to B$ is deemed equally probable regardless of whether it actually can be executed or not.

These requirements lead Henson et al. to consider two different types of transitions, which they label 'the relation move' and 'the link move'. The relation move is essentially equal to the removal or addition of a single order relation at a time, discussed at length above. The link move, on the other hand, corresponds to the removal or addition of the set of order relations

corresponding to a line segment in the Hasse diagram description of the causet. The addition of a Hasse line segment is subject to the requirement that none of the thereby introduced relations are already present. The two transition types are given equal probability.

As for the equilibrium criterion, Henson et al. seem to have taken a more holistic approach, ultimately deciding that the ordering fraction and the number of minimal elements provides the best indication of when a causet have reached thermalisation.

Due to technical limitations of their chosen implementation, Henson et al. only considers naturally labeled causets.

3.4.2 Destructive dynamics

As already hinted at, the dynamical scheme which this document labels 'destructive' only allows order relations to be removed, never added. The reasoning behind the given name should thus be self-evident. Since relations are only subtracted, this scheme is even simpler than the naïve scheme considered above. However, there is a twist: As points gets completely causally isolated from the rest of the causet, they are removed from the set. Thus, unlike the naïve case, the number of points are not constant throughout the simulation, but rather *decreasing*. Hence, the ordering fraction is *not* constantly decreasing, since the denominator gets smaller as more points are removed. To avoid running into problems with division by zero, the evolution should be halted when only two points remain. Starting the simulation with an initially totally ordered set with more than two points, one should thus end up with a graph of the ordering fraction which both start and end at $f = 1$, and with a global minimum $f = f_{\min} > 0$ somewhere between the two.

This is the motivating idea behind the selection criterion used in destructive dynamics. Rather than letting the terminating two-point causet be the final result, the set corresponding to the global minimum of the ordering fraction is deemed to be the resulting set of the evolution. A practical example is shown in figure 3.5. As this figure reveals, the evolution of the initial causet must be continued until its termination before the global minimum of f can be determined with certainty. Hence, in the practical implementation of this scheme, the destructive evolution is actually executed twice: once to determine the minimum of f, continued until the terminating causet is obtained; and once to obtain the causet corresponding to the minimized ordering fraction.

Given that the minimal causal set cannot be indisputably identified before one have 'dismantled' almost the entire causal set, these dynamics seem rather unphysical. Also, once a minimal causal set have been identified, the dynamical principles changes entirely and abruptly, leaving the causet 'frozen in time'—another unphysical trait. Finally, as the ordering fraction tends to zero, the dimensions of the corresponding spacetime tends to infinity. Hence, minimizing the ordering fraction seems unlikely to result in the familiar four dimensional spacetime of general relativity, or anything closely resembling it.

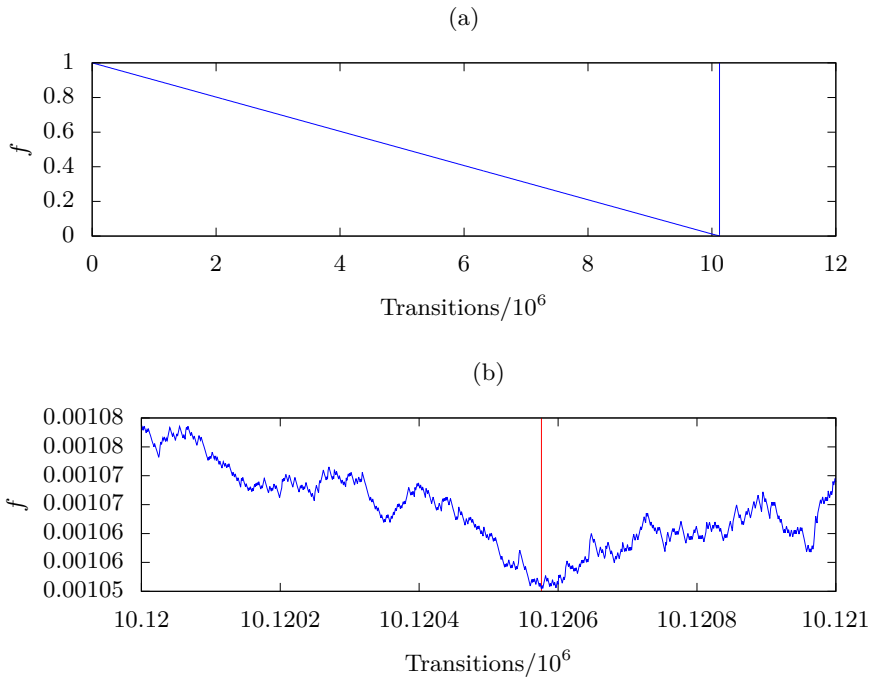


Figure 3.5: An illustration of the selection criterion used in destructive dynamics. Plot (a) shows the overall evolution of a causet with initially 4500 points. The active ordering fraction reaches an absolute minimum after 10 120 576 simulation cycles. This is better illustrated in (b), which shows a close-up of the relevant part of (a). The absolute minimum is marked with a red vertical bar. A closer analysis of the corresponding minimized set can be found in sections 5.2 and 6.2.

As a final remark, one should note that from an implementational perspective there is no need for removing points as they get causally isolated from the rest of the causet if one instead defines an *active* ordering fraction, which only calculates the ordering fraction of the causet given by the points which are 'active', i.e. non-isolated. This is the solution employed in the project code and reflected in the caption of figure 3.5.

Chapter 4

Algorithmic implementation

The goal of this chapter is twofold: to provide an overview of the project code, and to explain the reasoning behind the most central implementations. As for the raw C++ code itself, it might be found in appendix A, together with a few examples of how it might be used.

Considering overarching structure, the project code emerged from an object-oriented programming mindset. Hence, its fundamental structure is a map of nested classes, the 'higher' classes abstracting the more fundamental, 'lower' ones. While this design philosophy might lead to an unnecessary abundance of classes, it makes the abstraction involved in algorithm implementation easy by breaking it down to smaller peaces: provided the fundamental, low-level implementations are well-written, they might safely be ignored when working with more abstract, 'higher-level' objects.

In this particular case, the most basic object was a custom made dynamic storage container named **Narray**—after *N-array*; see below—not to dissimilar to the C++ standard library's **std::vector**. Building upon this, the **Event** class was introduced to describe a single point in a causal set. Additionally, the **Transition** class was defined, serving as an auxiliary structure for describing the space of available transitions. Incorporating all of the former classes, the **World** class was introduced, describing a causal set, its possible evolutions and some basic statistics. Finally, the **Worldview** class was defined for more in-depth analysis of causal sets, although this is more of an *expansion* than a further *abstraction* of the World class.

The first four of these classes are described in greater detail in sections 4.1 and 4.2, which details the different structures used in the code's description of causets and the algorithms by which said causets are evolved, respectively; while the final class is described in section 4.3, which details the analytical tools used to scrutinize the geometric structure of causets.

4.1 Core structures

This section details the central 'building blocks' of the project code, used to represent and store the elements of a causet, and used to edit their order relations on a low level. The implementation of the more high-level dynamical schemes of section 3.4 is given in section 4.2.

4.1.1 The 'Narray' class template

As none of the standard containers (**std::vector**, **std::deque**, **std::list**, etc.) was found entirely appropriate, the dynamical array class template **Narray** was created. Being essentially a tailored implementation of a dynamical array, the template name is taken from 'N-array', i.e. an array with N elements.

The elements of a Narray is stored in a dynamically allocated array created at initialization or after an allocation routine is called. Additionally, the number of active elements (the 'length' of the Narray) is stored as a separate variable, allowing for a very efficient deletion routine: To 'remove' all elements of an array, simply set the active elements to zero; to delete a specific element of a Narray of length N, say element i, with $i < N$, copy element N to i and reduce the length of the Narray with 1. In the project code, to be found in section A.1.1 of appendix A, these procedures are implemented as **erase(int** i**)** and **erase_all()**, respectively. Additionally, a third erase function, for the deletion of a specific element with unknown index is implemented, unsurprisingly named **erase_element(...)**. This member function searches the internal array for the index of the given element and then uses **erase(int** i**)** once the appropriate index is found. However, it only deletes the first instance of the given element that it finds, so in the case of duplicates it needs to be revoked several times. On the other hand, it does not do anything at all if the element is not found, so it is harmless to call it more times than what is needed.

Square brackets are used to access the elements of a Narray, similarly to the normal operation of an ordinary C++ array. As per C++ standard, the index start at zero, but there's no safeguard preventing access to elements with index greater than the current length of the Narray, in accordance with the comment made in section 3.1. There's also no safeguard against negative indexes, or non-integer indexes for that matter. Nonetheless, all the elements of a Narray can easily and safely be iterated through. This is thanks to the member function **length()**, which returns the length of the Narray. Hence, safe iteration can be done by incrementing an **int** variable from zero until the length of the Narray is reached.

A default constructor is also defined, allowing the creation of a Narray of unspecified length. To assign a length to such a Narray, and hence allow elements to be stored in it, the member function **allocate(int** N**)** is provided. This function can also be used to change the allocated length (which is stored as a private variable, just like the length) of a Narray in a semi-safe manner: If the allocated length of a Narray is increased, existing elements remain

intact, but if it is decreased, additional elements are deleted. Either way, the elements of the old internal, dynamical array are copied to a new one as part of the allocation process. Hence, considering speed, it is best to minimize the amount of allocation being done, preferably only allocating Narrays at creation, before they are filled with any elements to copy.

The full implementation of the Narray class template can be found in section A.1.1 of appendix A.

4.1.2 The 'Event' class

As detailed in section 3.2, the project code represents causets in two different ways: a trinary relation matrix—a simple example of which is shown in figure 3.1b—and the point-centered link lists $L^-(x)$ and $L^+(x)$—see figure 3.1c. The **Event** class, being the project code's realization of individual causet points, takes care of the latter, storing the link lists in two separate Narrays.

In keeping with ordinary C++ index standard, the events of a causet with n elements is labeled by integers ranging from 0 to $n-1$. Although this label in a certain sense is a 'global' designator, being dependent upon the specifics of the causal set which the Event object is part of, it is nonetheless stored 'locally' in the Event object itself. A similar commentary applies to the link lists.

To avoid dynamical re-allocation of the link lists as new links are added or removed during runtime, the total number of elements in the causet is stored in a private variable and used to allocate the Narray lists at creation. However, the member function **set_N(int** N**)** is also included, allowing for the re-allocation of the link lists at run-time. This functionality was added to make the conceptual implementation of the geometrical analysis tools easier; cf. section 4.3.1.

At this abstraction level, the adding or removing of links are handled by simple member functions adding or removing the appropriate event identifiers from the internal link lists. A simple iterative check prevents duplicates from being introduced by this process, but otherwise no safeguards are present. Thus nothing prevents a point from being linked to itself, or for being simultaneously both past- and future linked to another point.

Earlier versions of the project code, implementing naïve dynamics, also had two Narrays $L_H^-(x)$ and $L_H^+(x)$ for keeping explicit track of the links which could, according to the dynamical links, *hypothetically* be added to the past- and future link lists $L^-(x)$ and $L^+(x)$, respectively. This made the space of allowed transitions $\mathcal{T}$ directly accessible during any point of runtime, since $\mathcal{T}$ is simply the union of the past- or future link lists and the past or future hypothetical link lists for all the points of the given causet, together with an extra bit of information specifying whether said link should be added or removed. In the present version, the point-specific lists $L_H^-(x)$ and $L_H^+(x)$ are removed, while the explicit $\mathcal{T}$ has been kept, its 'hypothetical' part now constructed by iteratively checking all possible relations to see if they might be lawfully added or not; see the succeeding section for further details.

The full implementation of the Event class is given in section A.1.2 of appendix A.

4.1.3 The 'Transition' class

In order to represent the space of allowed transitions $\mathcal{T}$ the imaginatively named class **Transition** was created. With only three public member variables—two integers x_a and x_b specifying the link $x_a \prec x_b$, and one boolean value corresponding to if the link was to be *added*—this class is almost completely trivial. Apart from two constructors it has no member functions. Nonetheless, this is enough to provide an explicit construction of $\mathcal{T}$ by means of a Transition-filled Narray. This allows the Narray member function **length()** to be used in order to determine the total number of allowed transitions $|\mathcal{T}|$, making the implementation of the different dynamical schemes straightforward; see section 4.2.

4.1.4 The 'World' class

As already mentioned in the introduction to this chapter, the **World** class constitutes the project code's internal representation of a causal set. Additionally, it implements the dynamical principles for their evolution. Hence, it should be no surprise that the World class is more complex than the more fundamental classes considered above. Consequently, this section is divided into several subsections for clarity.

As for the name of the class, it is a slightly whimsical reference to the fact that a causet is assumed to be the fundamental description of reality in causal set theory, at least when ignoring the presence of matter—see [16, 23], earlier referenced in section 2.1.

The full implementation of the World class can be found in section A.1.4 of appendix A.

Member variables

To provide a complete link description of a causal set, the World class saves the individual Event objects constituting the causet in a private Narray. Rather redundantly, the total number of elements in the causet is *also* stored as a private variable, despite the fact that this number is saved in all of the individual Event objects themselves. Again, this was done for ease of implementation.

While the implementation of the link list representation relies heavily on the custom-made Event class, the implementation of the matrix representation uses a simple two dimensional array made from a self-nested array of pointers. Since the matrix is trinary rather than binary—see section 3.2 for details—its basic elements are normal integers rather than boolean values.

As alluded to in section 4.1.3, a Transition-filled Narray is used to represent the space of allowed transitions $\mathcal{T}$. However, since in the case of destructive

dynamics order relations only gets removed, the total number of links must in this case be equal to the total number of possible transitions. Hence, the total number of links is stored as a separate variable. This simplifies the implementation of destructive dynamics. Additionally, the total number of 'active' points—that is, points which are linked to at least one other point—is also stored, together with an iterator list—to be properly explained below—containing the labels of said points.

The possibility of saving a causet to an external file for later examination and backup is also added to the class definition. As a part of facilitating this functionality, a public string variable stores a descriptive identifier. When exporting a causet, a user defined string might be added as a postfix to create a filename. In order to allow for both easy human interpretation and a simple algorithm for loading external causets, the causet is saved twice, using two different file formats. The file meant for humans is saved using a 'txt' extension, while the file meant for later processing by the project code is saved as a custom binary file with the file-extension 'world'.

Constructors and initialization

Three constructors are defined: One copy constructor, one for loading a saved causet from an external file, and one for constructing a new causet with N elements. New causets are initialized to be totally ordered and naturally labeled. The auxiliary member variables, like the total number of links, are initialized to reflect this. Most of this is trivial, but the aforementioned iterator list warrants some additional explanation.

The originating idea behind this list was that given an Event label, say x_1, the label of the 'next' non-detached point should be found at the list-position corresponding to index x_1. Hence, a causet $\mathcal{C}_n$ with n elements—which is labeled using integers from 0 to $n-1$; see above—results in a list of $n-1$ event labels. A final element, having a value greater than $n-1$, could then be added to the end of this list, making it easy to write a condition for when the iteration should stop: When the returned iterator is no longer a valid event label, the iteration should stop. Furthermore, given that an initial causet is naturally labeled, it would be logical to let the iterator list be naturally ordered. As points become completely detached, their entries might be copied to the foregoing entries.

For a practical example, consider the small causet $\{x_0, x_1, x_2\}$, originally ordered such that $x_0 \prec x_1 \prec x_2$. Assuming $x_2 < x_3$, this causet have the corresponding iterator list (x_1, x_2, x_3). If now x_1 gets detached from all its causal relations, the iterator list becomes (x_2, x_2, x_3). Since the second 'x_2'-entry corresponds to the index x_1, this entry will now be skipped. However, this reveals a problem: What if x_0 becomes detached? How would one know where to start?

The answer is obvious: One needs an additional variable keeping track of where the iteration should start. An elegant solution would then be to add this variable to the iterator list, giving it index -1. Since the C++ standard

is to have array indexes start at 0, one might therefore be tempted to define a custom class which allows negative indexes. A much simpler solution, though potentially confusing when writing iteration loops, is to simply redefine the correspondence between Event labels and indexes such that e.g. the label x correspond to the index $x + 1$ rather than x.

In total then, the resulting iterator list has $n + 1$ elements, where the first element gives the 'first' non-detached point in $\mathcal{C}_n$, and where the last element is n. Given a valid Event label i, the next valid label have index $i + 1$. This is the solution implemented in the project code.

In accordance with the natural labeling of initial causets, the iterator list is strictly ordered upon creation. As the causal relations are changed according to the wanted dynamical principles, the procedure **update_il_and_Na()** should be called to ascertain that the iterator list and number of active points (the 'Na' in the function name) stays up to date.

Should the situation call for it, the causet can be re-initialized by calling the member function **order()**. Moreover, in keeping with the requirements of naïve dynamics, the causet can be totally disordered by calling the member function **disorder()**.

Algorithms for adding or removing a single order relation

As mentioned in section 3.3, figure 3.3 provides the blueprint for any algorithms pertaining to the addition or removal of one order relation at a time to or from a causal set.

It should be clear that this is trivial from the point of view of the matrix representation: If one wants to change the relation between two points, say x and y, one simply changes the two corresponding matrix entries—in this case (x, y) and (y, x). Hence, the less trivial part lies in the link representation, and specifically, which additional links *besides* $x \prec y$ that might be affected (the creation or destruction of the direct link $x \prec y$ is trivial, and is handled using the appropriate Event member functions). This is where figure 3.3 enters the picture.

When adding the link $x \prec y$, it is apparent from the figure that one should search through the future links of x, $L^+(x)$, looking for events which are also part of the future links of y, $L^+(y)$. This is handled by a simple, nested for-loop iterating over the elements in $L^+(x)$ and $L^+(y)$ looking for matches, and deleting the relevant events from the appropriate link lists if matches are indeed found (obviously the future links of y should remain unchanged by this). One also sees from the figure that the past links needs to be iterated through in an entirely similar manner, the only difference being that now it is the past link of x that should be left untouched. This algorithm is implemented as the member function **uplink(int** x**, int** y**)** in the project code.

The removal of a single order relation is now simply a matter of following the above algorithm 'in reverse'. After removing the link $x \prec y$, it must be determined which order relations, if any, that then become links. (Criterion

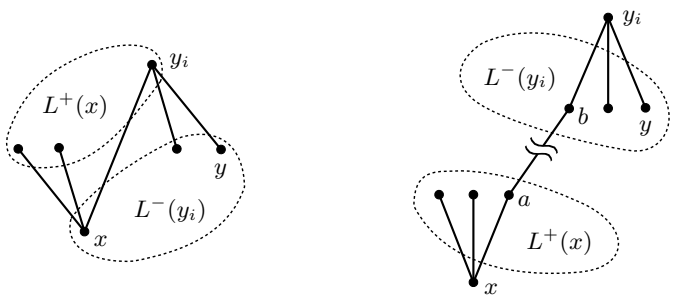


Figure 4.1: If two points, here x and y_i, are causally connected trough a maximal chain, then either (a) they're connected trough a link, in which case $x \in L^-(y_i), y \in L^+(x)$, or (b) they're connected trough intermediate links, in which case there exists at least one point $a \in L^+(x)$, and one point $b \in L^-(y_i)$, such that $a \prec y_i$ and $x \prec b$. Note that (b) allows $a = b$. Comparing these two possibilities, the moral is that given an initially self-consistent causet, its sufficient to look at $L^+(x)$ and $L^-(y_i)$ to determine if x and y_i is connected after the link $x \prec y$, with $y \prec y_i$, has been removed.

(3.2) ensures that, as long as the initial causet is self-consistent, the *only* order relations that *could* be missing after removing the link $x \prec y$, are themselves links). As can be seen from figure 3.3, the relevant relations are the subset of the future links of y which is not already connected to the exclusive future of x through a maximal chain not involving y, as well as the subset of the past links of x which is not likewise connected to the exclusive past of y. In the project code the elements of these subsets are identified through the process of elimination: Iterating through the future links of y, it is checked whether there is a maximal chain connecting x to y_i (where $\{y_i\} = L^+(y)$, in accordance with the notation suggested by figure 3.3). If no such chain is found, the link $x \prec y_i$ needs to be established in the appropriate link lists. The points future linked to x that need to be future linked to y might be identified in an entirely similar manner.

Now, barring consistency errors in the internal representations, the presence of a maximal chain connecting two such points might be determined by a iterative lookup in the relation matrix using the link lists of the appropriate points: By transitivity it is already known that $x \prec y_i$ (continuing the example given above). If there is already a maximal chain enforcing this relation, then either x and y_i are linked—in which case $x \in L^-(y_i)$ and $y_i \in L^+(x)$—or there must be at least one point among the future links of x which precedes y_i, whereby at least one point past linked to y_i must be preceded by x; see figure 4.1. Hence, one might iterate through $L^+(x)$ checking for all $z \in L^+(x)$ if $z = y_i$ or, using the relation matrix, if $z \prec y_i$. Alternatively, one can iterate through $L^-(y_i)$ looking for $z \in L^-(y_i)$ such that $z = x$ or $x \prec z$. If no points in the respective link lists fulfills neither of these criteria, then the link $x \prec y_i$

needs to be established.

The algorithm for checking whether two points are connected via a maximal chain is implemented as the member function **a_precedes_b(int** a**, int** b**)**, while the algorithm for removing the link $x \prec y$ is implemented as **delink_ordered_pair(int** x**, int** y**)**.

In addition to updating the link lists and the relation matrix, the member functions **uplink(int** x**, int** y**)** and **delink_ordered_pair(int** x**, int** y**)** also updates the member variable keeping track of the number of links, but the iterator list and number of non-detached points are, somewhat inconsistently, *not* updated and must instead be updated using the dedicated member function **update_il_and_Na()**.

An algorithm for determining whether a new order relation might be added

The dual representation of a causal set utilized in the project code makes it easy to implement criterion (3.2) from section 3.3 as an algorithm: The link representation reveals the elements of $L^-(x)$ and $L^+(y)$, while the matrix representation gives an accessible realization of $T^+(x)$ and $T^y(y)$, as well as an effective way of determining whether criterion (3.2) is fulfilled, given the elements of $L^-(x)$ and $L^+(y)$. This algorithm is implemented in the private member function **h_link_allowed(int** x_a**, int** x_b**)**, which returns the boolean value **true** if the link $x_a \prec x_b$ can be added to the causet.

Using the link lists to identify the relevant points ensures that one does not (always) have to compare $2n$ matrix elements—assuming a causal set with n elements—while the relation matrix provides a simple look-up to determine the relation between two points, something which could easily become a recursive task if the link representation where to use in isolation. Thus some of the complementary qualities of the dual causet representation used in the project code is revealed.

4.2 Implementation of dynamical principles

This section provides an overview of how the dynamical systems considered in this thesis are implemented. Unless otherwise stated, the member functions referred to herein belongs to the World class. Their raw C++ implementation can therefore be found in section A.1.4 of appendix A.

4.2.1 Naïve dynamics

As described in section 4.1.3, the Narray member function **length()** might be used to determine the total number of allowed transitions $|\mathcal{T}|$. Assuming $|\mathcal{T}| = m$, one might then select a random integer in the closed interval $[0, m-1]$ to serve as a random index singling out the next transition. A trivial World member function can then be used to convert the chosen Transition object into an *actual* transition. After the transition is performed, the ordering

fraction is saved, and the appropriate member variables updated before the algorithm repeats itself.

This, in broad strokes, is the project code's implementation of naïve dynamics. However, a few additional details merit a somewhat closer explanation:

- The random integer is obtained using a 64-bit Mersenne Twister engine providing a random float in the half open interval $[0, 1)$ which is then rescaled and recast to the appropriate closed integer interval. This is accomplished through the use of the C++11 standard library **<random>**, which introduces various random number engines and variable distributions into C++. Consequently, the project code must be compiled using the C++11 standard.

- In keeping with the prescriptions of naïve dynamics, the number of points in the causet is kept constant throughout the entire evolution. This means that all points are surveyed when calculating the ordering fraction of the causet, regardless of whether they are disconnected from the rest of the causet or not. As pointed out in section 3.4.2, this differs from how the ordering fraction is calculated in the case of destructive dynamics. Rather than writing a compromise algorithm which calculates the ordering fraction relative to *any* total number of points, the project code utilizes two different algorithms, one optimized for the naïve case, and one for the destructive case. The member function **f_N()** calculates the ordering fraction relative to the total number of points.

- Due to time-constraints, the code is not parallelized, meaning that it runs on only one CPU-core at a time. Thus, two causets cannot be evolved simultaneously. As a consequence of this, there's no internal way to determine whether thermalisation has occurred or not. (Conceivably, some form of 'averaged relative change' of the ordering fraction could be used as an equilibrium criterion, but as confirmed by the results in chapter 5, such a criterion turns out to be not quite as clear cut as one might expect). Hence, finding the 'correct' number of simulation cycles is a user guided process of trial-and-error. The evolution algorithm therefore contains the number of cycles (or simulation steps) as an argument.

- In order to save disk space, the ordering fraction log is coarse grained, recording only minimum and maximal points for each N transition cycles, where N is specified in a function argument. Additionally, in order to protect against the consequences of unexpected system restarts and crashes, an argument for specifying causet backup intervals was also added. In order to maximize the number of possible cycles, all arguments pertaining to transition cycle numbers use the 64-bit unsigned integer type **uint_fast64_t**.

- As it is written now, the evolution algorithm simply advances the causet it is applied to forwards according to the simple dynamical principles

stated in section 3.4.1. Hence, in order to discover the sought after equilibrium state described in that same section, the algorithm needs to be run twice, once with an initially totally ordered causet and once with an initially totally disordered one. An eventual equilibrium can then be deduced by comparing the two resulting ordering fraction logs. See sections 5.1 and 6.1 for practical examples.

In the project code, naïve evolution dynamics is implemented in the World class member function **evolve_and_save(...)**. In addition to the three arguments already mentioned, a forth argument—incidentally the *first* argument in the actual implementation—provides a seed for the Mersenne Twister engine, enabling simulation runs to be reproduced. For technical reasons this seed should be of the type **uint_fast64_t**, but a normal **int** also works.

4.2.2 Destructive dynamics

Given that the destructive dynamics only *removes* order relations, the total number of allowed transitions $|\mathcal{T}|$ at any given moment is simply equal to the number of links in the destructive scheme. Moreover, since all of the transitions are of the same *kind*, an explicit implementation of $\mathcal{T}$ becomes rather unnecessary—the total number of links is really all that's needed. Assuming a causet $\mathcal{C}_n$ with n elements and m unique links, a random transition might be chosen simply by selecting a random integer between 1 and m, and then removing the correspondingly numbered link, the numbering being done according to the iterator list. To locate the link which is to be removed, one iterates trough the points of $\mathcal{C}_n$, adding up the number of links per point until the chosen number i is reached. The link is then removed, the ordering fraction noted and compared with earlier values, and the appropriate member variables—among them the total number of links—updated. The algorithm then repeats itself.

This is the essence of the project code's implementation of destructive dynamics. A few comments:

- The random integer is obtained in an entirely similar manner as in the preceding section. Hence, an integer seed, of the same type as described above, needs to be provided when the evolution algorithm is run.

- When using destructive dynamics, the ordering fraction should be calculated relative to the total number of so-called 'active' points in the causet. This is implemented in the member function **f()**. An event is labeled 'active' as long as it is linked to at least one other event.

- To avoid running into problems with undefined behavior, the destructive evolution is halted when only two active points remains. Unlike the situation when one is using naïve dynamics, the total number of cycles needed to reach the final stage can therefore quite easily be calculated

given an initially totally ordered causet with n elements. Specifically, the calculation shows that the number of cycles scales like n^2. Even so, a backup functionality—with a corresponding function argument—is built into the algorithm to facilitate the evolution of larger causets, which in this context turns out to be causets with thousands of points; see figure 3.5 and also section 5.2.

- Since the minimal causet cannot be accurately determined before 'almost all' of the ordering relations are removed, and because runtime memory—after all—is limited, the destructive evolution algorithm needs to be run twice, similarly to the naïve dynamics case. However, unlike the naïve case, the goal of these two runs are now subtly different, and hence also their algorithms: The first run is done to determine the exact *number* of simulation cycles needed to reach the minimal causet, while the second is done to obtain *said causet*, enabling further analysis to be carried out (see section 4.3.1. Rather than having one member function which performs both of these algorithms successively, the project code therefore have two implementations of destructive dynamics: The member function **dismantle_and_record(...)** records the evolution of the (active) ordering fraction, and returns the number of cycles needed to reach the causet for whom it is minimized; while the member function **dismantle_and_save(...)** destructively evolves the causet it is applied to a specified amount of steps forwards. This solution makes it easier to resume evolution from a partially 'dismantled' causet in the event of an unexpected system crash.

- For the convenience of the experimenter, an information text file with various useful data is created as part of the evolution algorithm(s). While the evolution is still ongoing, the text file gets updated every time the backup interval is reached, providing a means to see approximately how far the simulation has come. When the evolution is finished, the file gets overwritten with useful simulation data, providing a short summary of the performed evolution and its results.

4.3 Analytical tools

Due to how it is defined, the ordering fraction is of no help when trying to figure out the dimension of a *spatial* cross section of spacetime, which is the continuum that it was hoped had been obtained through use of the destructive dynamical scheme. This forms the backdrop for the creation of the analytical tools described in this section.

4.3.1 The 'Worldview' class

Continuing the whimsical naming convention established by the World class, the **Worldview** class provides tools for closer analysis of the geometrical

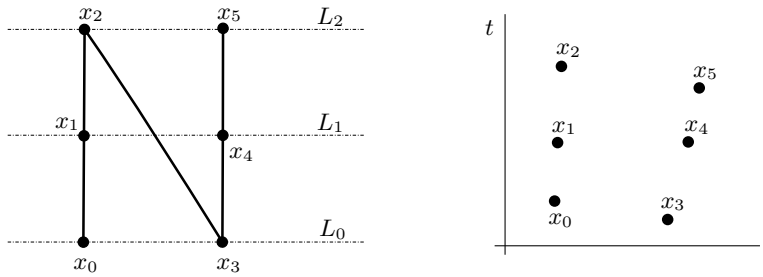


Figure 4.2: To compare the dynamically obtained causets with the Kleitman–Rothschild theorem, the Worldview class divides the causal set points into temporal layers by the length of their longest past chain, as shown in (a). In the continuum limit this corresponds to a specific choice of reference frame. This can be seen from (b), which shows a two dimensional spacetime diagram corresponding to (a).

features of a given causet. As stated in the introduction of this chapter, this class is best understood as an extension of the World class. Technically, the Worldview class is derived from World while also being a **friend** of it. By nature of its goal—and because it is somewhat unfinished and unpolished; the unnecessary parts not completely identified and removed—it is the most haphazard class by far. In terms of memory, it is also the most wasteful, employing a whole plethora of member variables in its implementation.

The full implementation of the Worldview class may be found in section A.2.1 of appendix A.

The basics

Being written first and foremost for use on causets obtained from destructive dynamics, the constructor—which takes a World object as its argument—removes all casually isolated points from the originating causet. As a part of this process, the points are renumbered, removing the need for the inherited iterator list and the variable keeping track of the number of active points. Thus, these variables are simply ignored and left unchanged. Trying to use them inside an implementation could therefore easily lead to unspecified behavior.

A central goal was to be able to compare the dynamically obtained causets with the asymptotically dominating class of causal sets described by the Kleitman–Rothschild theorem. To do this, the resulting causets should be divided into temporal layers according to the length of their longest past chain. Figure 4.2a shows the needed division, while figure 4.2b gives a hint of what this layering signifies in the corresponding continuum limit.

The temporal layers are saved as a private member variable using a Narray. To represent the layers themselves, a Narray filled with integers is used, the integers corresponding to the points which are elements of the given layer.

Hence, the layers are stored in a Narray filled with Narrays filled with integers. Due to the C++ standard of indexes starting at zero, the layers are numbered according to the length of the longest past chain a point must have in order to be an element in said layer. Hence, the elements of the bottom layer L_0 is easily obtained from the link representation by identifying the points which does not have any past links. Successive layers is then constructed by iterating trough the future links of the points in the already obtained layer(s) and adding the linked points to the layer immediately above.

However, since the points of a causet might have several maximal past chains of different lengths, this process is likely to lead to points being added to several layers (consider what the above algorithm does to the point x_2 in figure 4.2a). To remedy this, the layers are then iterated through in descending order, the elements found in higher layers erased from eventual lower layers they are found to also be contained in.

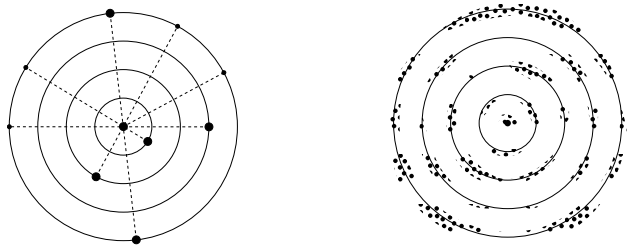
This is the central parts of the algorithm used to initialize the layer Narray in the member function **initialize_temporal_layer()**.

Tools for analyzing the spatial features of a temporally layered causet in greater detail

As already revealed in section 1.1, the causets obtained by use of destructive dynamics are (approximately) two-layered. An analysis of the spatial features of these sets thus becomes very interesting, in order to determine whether they might be provide a causal set description of spatial cross sections of ordinary spacetime. At the core of the methods presented in this section lays two matrices showing the 'spatial' separation—as measured by the length of the minimal connected antichain—between points in the same extremal layer. The reason the minimal and maximal layers are used instead of the temporal layers presented above is simply an accident of history, but it provides for an analysis which to some extent can ignore the presence of a lonely 'lump' in the causal cross section. However, this solution also has a small disadvantage due to how the present code measures spatial separation, but this will be discussed later.

In accordance with how a connected antichain is defined, these two distance matrices have two even more fundamental matrices underlying them, one listing how many past links any pair of points—regardless of their corresponding layers—have in common, and one listing how many future links are in common. These matrices are constructed in a straightforward manner from the link lists.

Using these lists as a starting point, the distance matrices are then constructed in a radial manner: First, the points are sorted into their respective matrices and given zero distance to themselves. The shared links then provide the set of points within the two layers that are connected by a minimal connected antichain of length 1, and these distances are saved to the respective matrix-entries. If some points are found to have no neighbors in their corresponding layer, i.e. have no unit minimal antichains, then these points



(a) (b)

Figure 4.3: The motivating idea for organizing the points in an anticonnected layer into 'shells': Assuming a collection of concentric spheres in ordinary continuous Euclidean geometry, the sphere any given point x belong to might be determined by finding the longest straight line segment between x and the other points in the collection (a). Using that straight spatial lines corresponds to minimal connected antichains in the causal set theory regime, one might thus order the points of an anticonnected layer into concentric 'spherical shells'. Having done so, one might then compare the length distribution of the connected antichains for an average point in the inner 'center' shell to the overall shell distribution to obtain some information as to what degree the obtained shell description is meaningful (b).

can never be reached by a connected antichain, and so they are given an infinite separation from all the points in their respective layer. Having thus established a web of unit minimal antichains, the spatial distance between two points of unknown separation can then be determined by iterating radially outwards from one of them until the other point is found. The theoretical maximum length of a minimal connected antichain in a causal set with n points is $n - 1$. Hence, if one fails to find a minimal connected antichain which is shorter than n connecting two points, said points must be infinitely spatially separated.

In the project code, -1 is used to represent infinite separation, while -3 represents an undetermined distance. For ease of implementation, all points are listed in both distance matrices, and -2 is used to signify that a point does not actually belong in the given layer.

If some points are infinitely separated, the causet can be ordered into causally isolated semi-ordered groups which can then be sorted according to their size. A list of these groups is saved in a member variable with a nested Narray structure reminiscent of the one used for the list of temporal layers.

Finally, by exploiting the connection between minimal connected antichains and straight lines in Euclidean space, one might order the points of an anticonnected layer—i.e. the extremal layers restricted down to a semi-connected group—into 'concentric shells'. Figure 4.3 shows the motivating idea and what this might be used for.

Chapter 5

Simulation results

To avoid an inordinate amount of graphs and figures, some of the simulation results are summarized in words only. The graphs and figures which are included, are thus meant to be representative of different general trends, although a few figures most likely fall short of this ideal, for reasons which will be identified below as said figures are presented.

5.1 Using naïve dynamics

5.1.1 Basic results

Starting of, figure 5.1 reprints the naïve evolution of a causet with 32 points used to illustrate the equilibrium criterion for naïve dynamics in section 3.4.1. In order to facilitate a comparison with the Kleitman–Rothschild theorem (see section 2.3), the mean value of the ordering fraction—disregarding the initial 'transients' before the graphs flatten out—have been computed as an estimate of the equilibrium value of the ordering fraction. Note that the calculation of this average uses the same coarse grained data as the graphs themselves, which only records the extremal values over the given grain duration. Hence, the *true* average value—and especially its standard derivation—might be slightly different from the one calculated from the recorded fraction log, given here.

Corresponding graphs for causets with 64 and 80 points are shown in figures 5.2 and 5.3, respectively. Note that the causet with 80 points does not reach equilibrium during the set simulation time. Although this might be as expected when one compares the jump in thermalisation time from figure 5.1 to 5.2, and considers the—by comparison—modest change in run time between figure 5.2 and 5.3, this is indicative of the results obtained when attempting to simulate larger causets.

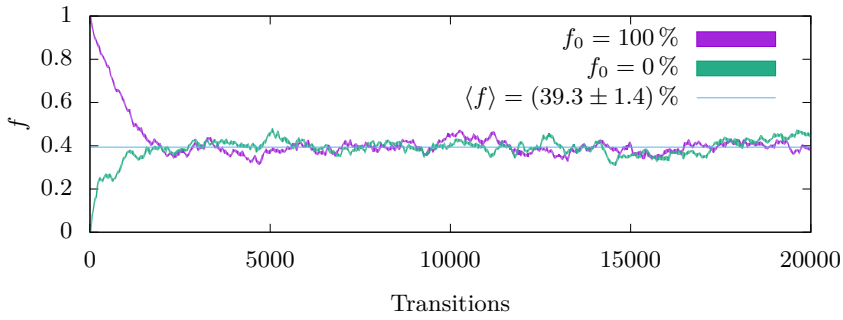


Figure 5.1: A reprint of figure 3.4; this time with an estimated equilibrium value included. The underlying causet(s) has 32 points, and the data is coarse grained using a cycle length of 40 transitions. The estimated equilibrium value $\langle f \rangle = (39.3 \pm 1.4)\,\%$ is the average of the plotted data corresponding to the last 1.8×10^5 transition cycles; the preliminary 2000 cycles before the graphs assimilate are ignored. In the rest of this text, the terminal causet obtained from the initially totally *ordered* causet with 32 points will be labeled $\mathcal{C}^n_{32}$, while the one evolved from the totally *disordered* causet will be labeled $\mathcal{C}'^n_{32}$.

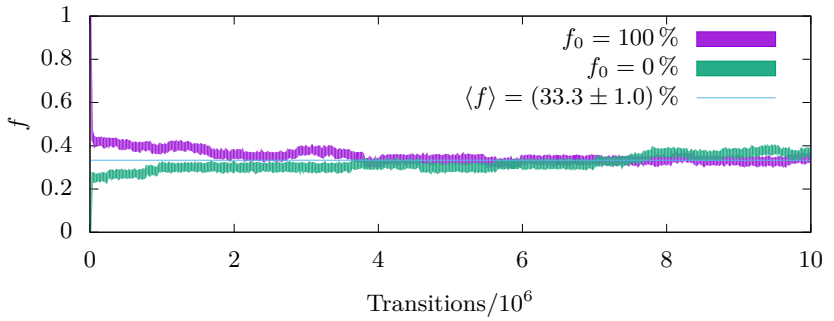


Figure 5.2: Naïve evolution employed on a causet with 64 points. The used coarse graining is 2×10^4 transitions. The estimated equilibrium value $\langle f \rangle = (33.3 \pm 1.0)\,\%$ is the average of the plotted data corresponding to the final 6×10^6 cycles after the graphs 'settles down' around the 4×10^6 mark. In the rest of this paper, the terminal causets obtained from this evolution will be labeled $\mathcal{C}^n_{64}$ and $\mathcal{C}'^n_{64}$, where one is using a similar naming convention as the one described in the caption of figure 5.1.

General remarks

Using an older version of the project code, naïve evolution was attempted employed on causets with 128, 256 and even 512 points. However, as already hinted at, these simulations did not reach equilibrium during the given runtime, even after—in the case of one of the causets with 128 points—over two and

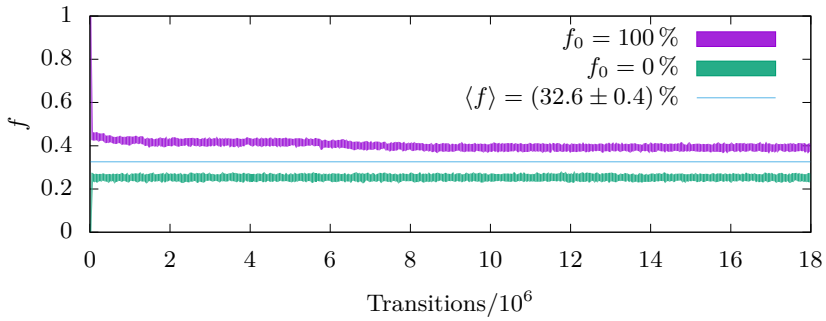


Figure 5.3: Naïve evolution employed on a causet with 80 points, using a coarse graining of 3.6×10^4 transition cycles. The calculation of the estimated equilibrium value $\langle f \rangle = (32.6 \pm 0.4)\,\%$ is done using all of the available data, except for the part corresponding to the first 1.98×10^6 transition cycles. Given that the two graphs doesn't seem set to meet for many millions of cycles yet, the *real* uncertainty in $\langle f \rangle$ is largely unknown. Using a similar naming convention as for the causets in figures 5.1 and 5.2, the terminal causets will be labeled $\mathcal{C}^n_{80}$ and $\mathcal{C}'^n_{80}$.

a half billion transitions! Hence, figure 5.3 is indicative of these results, the larger causets being further from equilibrium by the end of the simulation time than the smaller ones. In addition to depending on the causet size, the transition time was also seen to depend on the complexity of the order relations, and hence also weakly on the ordering fraction; strongly ordered causets transitioned faster than less ordered ones, but lightly ordered causets transitioned faster than the more complex intermediate ones. Hence, when performing a parallel run of one initially totally ordered causet and one initially minimally ordered (see below), the initially totally ordered would finish the first hundred of thousands transitions faster than the other causet, but as the simulation continued on, they would more or less catch up to each other.

5.1.2 A closer look at the resulting causets

In addition to providing an asymptotic value for the average ordering fraction, the Kleitman–Rothschild theorem also details the typical structure of an asymptotic causet, which turns out to be temporally three-layered; see figure 2.3. To address this, figures 5.4, 5.5 and 5.6 are included, giving the temporal layer structure of the six resulting causets obtained after the evolutions shown in figures 5.1, 5.2 and 5.3, respectively. However, these sets is by no means claimed to give representative pictures of typical equilibrium sets, since such a claim would have to be backed by analysis of thousands (or at least hundreds) of similar causets, and such an in-depth analysis have not been performed as part of this thesis. Instead, the characteristics of these causets will be compared to the averages found by Henson et al. [13] in their

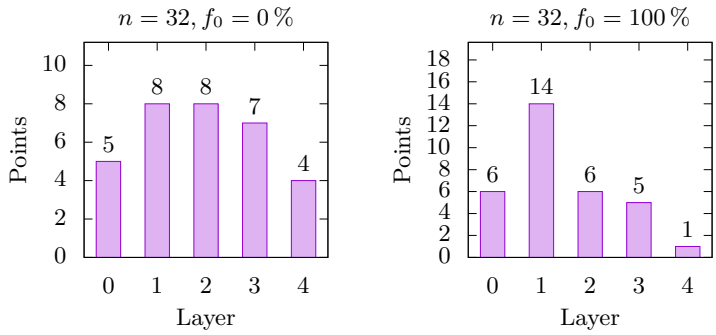


Figure 5.4: The layer structure of $\mathcal{C}'^{n}_{32}$ and $\mathcal{C}^{n}_{32}$, respectively, the two resulting causets after the evolution shown in figure 5.1 has been carried out. The layers are numbered according to the length of the maximal past chains of it is constituent points, which incidentally coincides with the standard index numbering used in C++.

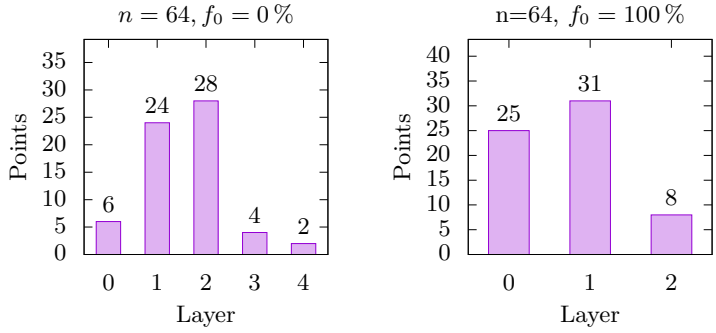


Figure 5.5: The layer structure of respectively $\mathcal{C}'^{n}_{64}$ and $\mathcal{C}^{n}_{64}$, the two causets of figure 5.2 after the evolution has been carried out.

remarkably detailed exploratory work.

Finally, to address the question of whether the apparent equilibriums shown in figures 5.1 and 5.2 are stable, figure 5.7, which shows the naïve evolution of a causet with 64 points continued for over 5 billion cycles, is included. As mentioned in the caption, this figure was created using an older version of the project code, specifically, the same as the one mentioned above when discussing the results for causets with more than 80 points. Unlike the current version of the project code, this older version insisted that the causet should remain an Alexandrov interval at all times, meaning that any causet $\mathcal{C}$ had a fixed starting point $x_0 \in \mathcal{C}$ such that $\forall\, x \neq x_0 \in \mathcal{C} \colon x_0 \prec x$, and a fixed ending point $x_f \in \mathcal{C}$ such that $\forall\, x \neq x_f \in \mathcal{C} \colon x \prec x_f$. Hence, the initial causet cannot be completely disordered, as mentioned but not explained above, and

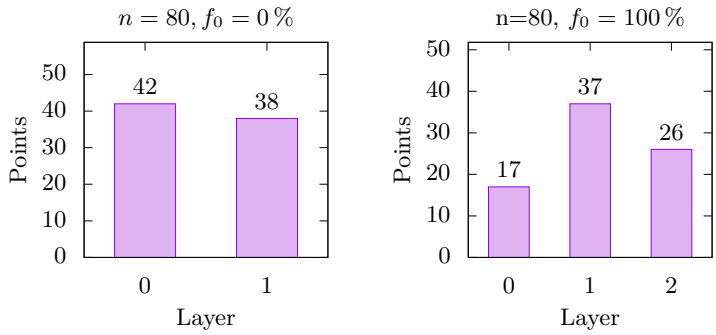


Figure 5.6: The resulting layer structure of respectively $\mathcal{C}'^{n}_{80}$ and $\mathcal{C}^{n}_{80}$, the two causets in figure 5.3 after the evolution has been carried out.

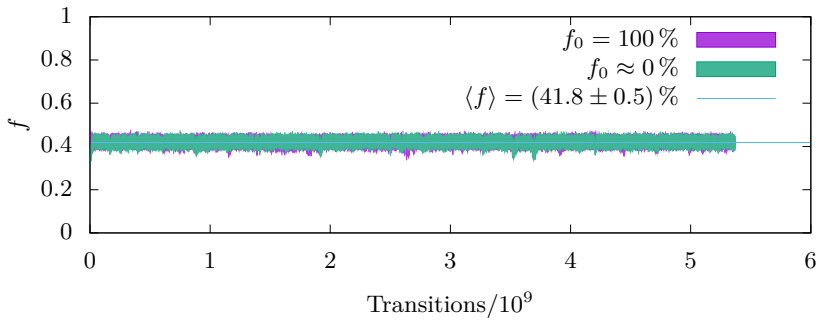


Figure 5.7: Using a slightly older version of the project code, a causet with 64 points was naïvely evolved using a coarse graining of 5×10^9 transition cycles to check the stability of the apparent equilibrium seen in figures corresponding to figures 5.1 and 5.2. Compared to the relative coarse graining of earlier graphs, this graph is quite detailed; extremum values are shown for every 2^{16} cycles. When calculating the estimated equilibrium value $\langle f \rangle = (41.8 \pm 0.5)\,\%$, the first 2^{24} cycles has been ignored. Unlike the code which produced figure 5.2, the older code used here never allowed the causet to cease being an Alexandrov interval. Hence, it should be no surprise that $\langle f \rangle$ as calculated here is higher than the corresponding value in figure 5.2.

one would expect the estimated equilibrium value to be greater than when using the present version of the code. Hence, the estimate of $\langle f \rangle$ shown in figure 5.7 might be reconcilable with that in figure 5.2.

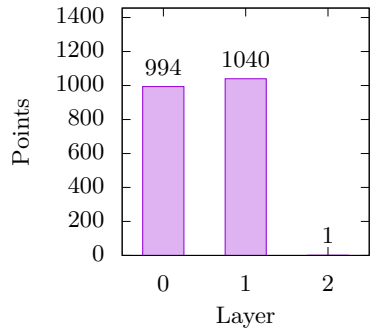


Figure 5.8: The layer structure of $\mathcal{C}^d_{2035}$, the causet obtained after the destructive evolution shown in figure 3.5. While the initial causet had 4500 points, $\mathcal{C}^d_{2035}$, as the name suggests, has 2035.

5.2 Using destructive dynamics

5.2.1 Basic results

Turning now to the simulation results obtained when using destructive dynamics, figure 5.8 shows the layer structure of the largest destructively evolved causet, whose evolution was shown in figure 3.5. Incidentally, this is also the largest causal set the project code has successfully dynamically simulated, having initially as many as 4500 points, and ending up with 2035 still intact when the minimal order relation was reached some days later. As can be seen, the obtained causet, which will from hereon be labeled as $\mathcal{C}^d_{2035}$, is practically two-layered, only one single point having a maximal past chain of length 2. Later results presented below will reveal that this point can safely be ignored, as it turns out to only be linked to one single point. Thus, this point does not represent a significant departure from a two-layered causet, since one might identify it with its linked partner to obtain a two layered causet with 2034 points.

Turning now from the temporal structure of $\mathcal{C}^d_{2035}$ to the spatial one, figures 5.9 and 5.10, reveal the distance, as defined by the length of minimal connected antilinks, between points in the same extremal layer. As pointed out in section 4.3.1, the definition of an extremal layer is subtly different from the layers shown in figure 5.8. Specifically, a *minimal layer* consists solely of points which have no preceding points. Similarly, a *maximal layer* consists exclusively of points which does not precede any other points.

These plots are strongly indicative that $\mathcal{C}^d_{2035}$ does *not* consist of one jointly connected causet, but rather of a disjoint collection of smaller causets; one significantly bigger than the others. The reason the multitude of 'infinitely spatially separated' points are only *suggestive* evidence of a disjoint set, is because the distance tables underlying the shown figures only gives the

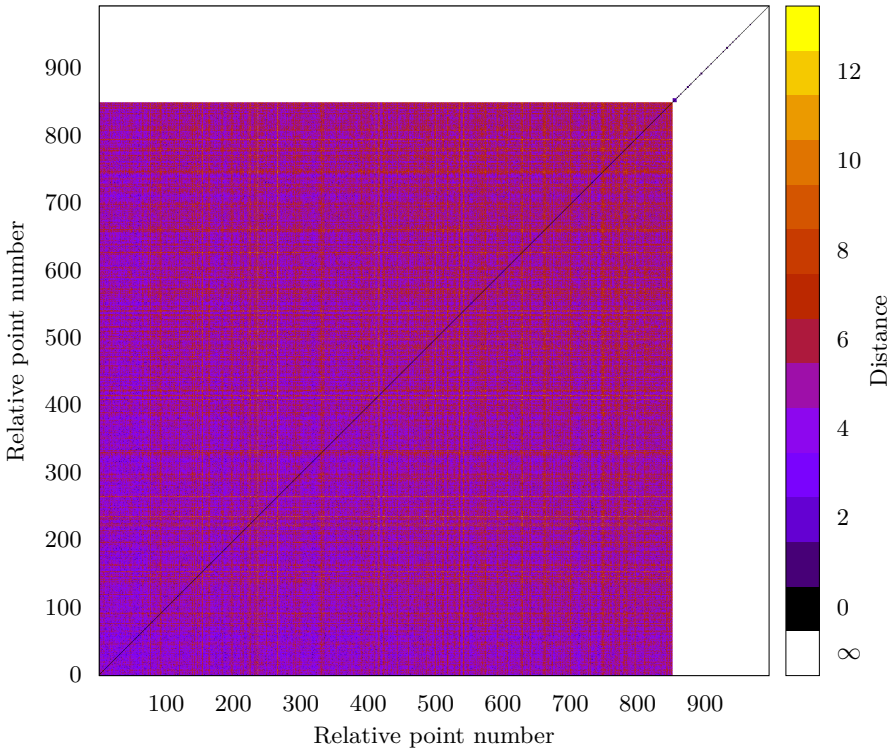


Figure 5.9: A table of the 'spatial' distance between the points in the minimal layer of $\mathcal{C}^d_{2035}$, represented as a heat map. The layer contains a total of 994 points. In this context, the 'spatial distance' between any two points is equal to the length of a minimal connected antilink between them; if no such antilink can be found, the distance is set to ∞. The fact that there are several points which in this sense is 'infinitely' separated is an indication that $\mathcal{C}^d_{2035}$ might not be *one* interconnected causet, but rather a *collection* of smaller disconnected causets.

length of the minimal connected antichains, which is based on common links. Hence, if two points are connected through a set of shared order relations where none of them are shared links, the points will be said to be infinitely spatially separated. Since $\mathcal{C}^d_{2035}$ has three temporal layers, there thus exists the theoretical possibility that all of the points shown to be spatially disconnected in figures 5.9 and 5.10 actually are connected trough the single point occupying the third layer L_2 shown in figure 5.8.

However, by using the order relation matrix together with the distance tables underlying figures 5.9 and 5.10, one might mediate this shortcoming, revealing whether the isolated blocks seen in the figures actually corresponds to causally isolated subsets or not. Doing so, one finds that there is a total of 116 causally isolated subsets in $\mathcal{C}^d_{2035}$, 83 of which only consist of two

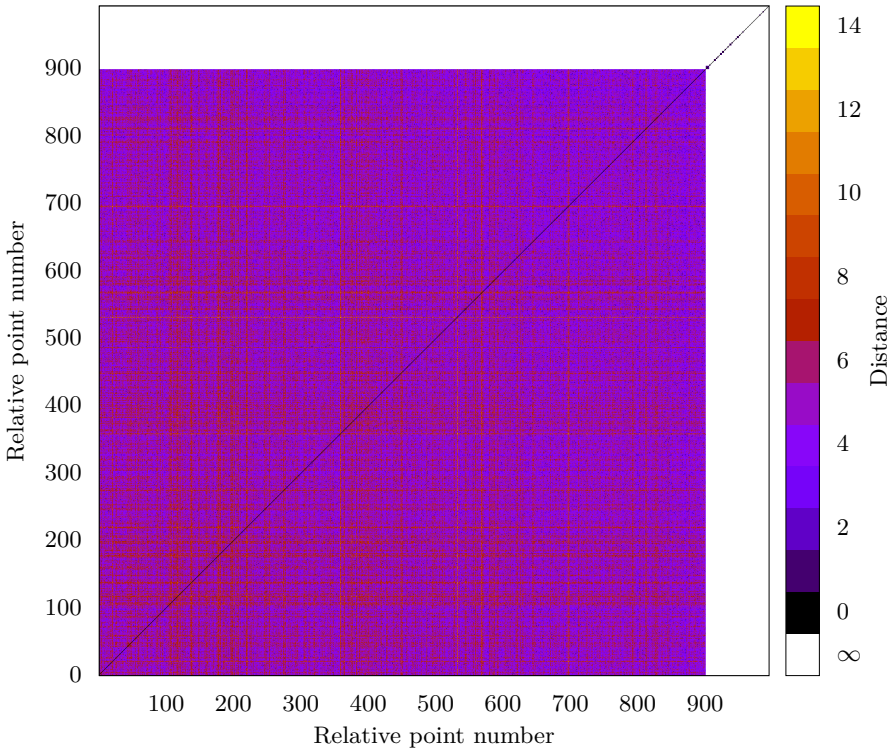


Figure 5.10: A table of the 'spatial' distance between the points in the maximal layer of $\mathcal{C}^d_{2035}$, represented as a heat map. The layer contains 1040 points. Incidentally, this is exactly the same number as in the second layer of figure 5.8, but there's a subtle difference in what is meant by a 'layer' here: In the histograms shown in figures 5.4–5.6 and 5.8, points are assigned a 'layer' according to the length of their longest past chain—'layer 0' contains all points which have no past links, 'layer 1' contains all points with no past chains longer than 1, and so on. The 'maximal layer' shown in this figure, however, is simply the set of points which have no future links; hence, a point which is in 'layer 1' in figure 5.8, will be part of the 'maximal layer' shown in this figure as long as it does not precedes the *one* point in the third layer of figure 5.8! (The 'minimal layer' of figure 5.9 is defined in a similar manner, but here the definitions actually coincide). Again, the presence of infinitely 'spatially' separated points strongly suggests that $\mathcal{C}^d_{2035}$ consists of several smaller disconnected causets.

(totally ordered) points. The composition of the ten largest subsets is listed in table 5.1. The geometry of the largest of these subsets will be further explored in section 5.2.2.

Table 5.1: The ten largest causally isolated subsets of $\mathcal{C}^d_{2035}$ and their composition.

Group	Points in total	... in minimal layer	... in maximal layer
0	1750	850	899
1	11	6	5
2	6	3	3
3	5	2	3
4	5	2	3
5	5	3	2
6	4	2	2
7	4	1	3
8	4	1	3
9	4	2	2

General remarks

Destructively evolving causets of increasing size revealed, unsurprisingly, that larger initial causets results in smaller minimal ordering fractions. Moreover, all of the minimal causets had roughly half of the points of the totally ordered causet that they where made from, but this number was seen to vary quite a bit depending upon the initial seed. However, as only two sets with 4000 or more initial points were sampled, it is unknown whether a 'better' seed would have produced a causet with ~ 3000 points, and if so, how this set would have looked.

Considering the structure of the destructively evolved causets, present results indicate that they indeed seem to be practically or exactly two-layered, a lonely point like the one in figure 5.8 showing up in roughly every other set. The situation shown in table 5.1 is especially typical—I have found no exceptions so far: A clear majority of the causet points are gathered into one relatively large causally connected subset, while the remaining points form a collection of several small subsets consisting only of a handful of points each. In terms of a cross section of a continuous, classical spacetime, the resulting picture seems to somewhat resemble that of a big 'soap bubble' surrounded by several *significantly* smaller ones.

5.2.2 A closer look at the largest simulated subset

Using the analytical tools described in section 4.3.1 on the largest subset of $\mathcal{C}^d_{2035}$, figures 5.11 and 5.12 are obtained. From the first figure one sees that the subset is relatively spatially dense—the great majority of the points is packed together within a 'sphere' 10 length units in diameter, and none are further separated than 14 length units (this last fact can also be inferred from figures 5.9 and 5.10). This might also explain the large standard deviations seen in figure **??**.

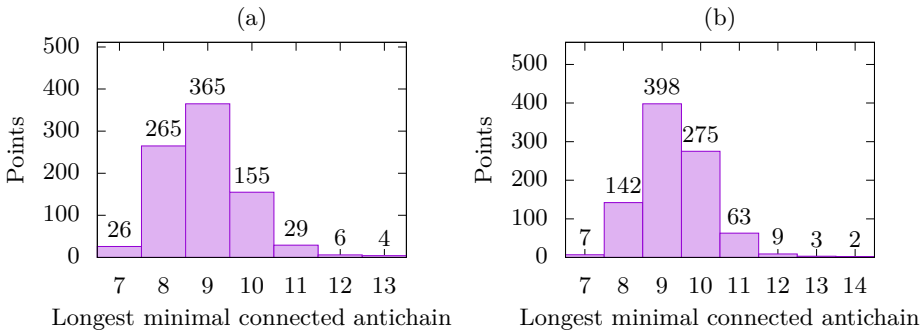


Figure 5.11: The points of the largest causally connected subset of $\mathcal{C}^d_{2035}$ distributed according to the length of their longest minimal connected antichain. In (a), the distribution of the points in the minimal layer is shown, while (b) gives the distribution for the maximal layer. As can be seen using the identification indicated by figure 4.3a, the minimal center shell has 26 points, while the maximal center shell has 7, less than one third of the points in its corresponding shell in the layer below it. Interestingly, the 'diameter' of the minimal layer is less than that of the maximal. The overall shell distribution is also markedly different.

For a d-dimensional sphere in ordinary, continuous Euclidean geometry there exists a natural correlation between the longest distance to another point of the sphere and radial position within the sphere (figure 4.3a). Employing the same identification on the discrete anticonnected points shown in figure 5.11, one sees that the points of the minimal layer are more gathered in the innermost shells than the points of the maximal layer. In fact, the minimal layer has one less shell than the maximal layer.

Figure 5.12a shows the distance distribution of an average point in the center shell in the minimal layer, while figure 5.12b shows the corresponding distribution for an average point in the maximal layer. There is some clear discrepancies between the shell distributions implied by figure 5.12 and those implied by figure 5.11.

It is unknown how these graphs compare to a 'general' causet obtained by use of destructive dynamics. Preliminary results obtained from looking at the average distance distributions in the extremal layers of the causet as a whole, suggest that the statistics shown in figures 5.11 and 5.12 are likely to vary quite a lot from one causet to another.

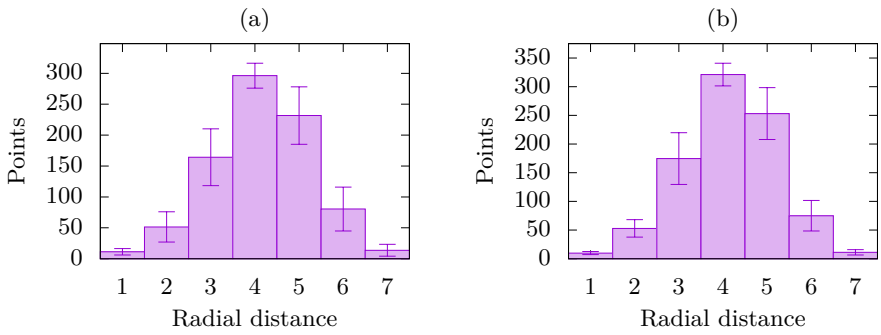


Figure 5.12: Average 'spatial' radial distribution for the points in the two extremal center shells of the largest causally connected subset of $\mathcal{C}^d_{2035}$. Similarly to figure 5.11, (a) shows the distribution for the minimal layer, while (b) shows the same statistics for the maximal layer. The shell distribution seen by an average point in the central shells are markedly 'skewed' compared with the overall 'objective' distributions seen in figure 5.11a,b.

Chapter 6

Analysis

6.1 Naïve dynamics

The results listed in section 5.1 suggests that naïve dynamics is unable to escape the entropy problem revealed by the Kleitman–Rothschild theorem. Although only two of the six analyzed sets have exactly three layers, and only one of them have a layer distribution somewhat resembling that given in the theorem, all the simulations result in an average ordering fraction which is in the ballpark of that suggested by the theorem. Hence, naïve dynamics seem unlikely to result in causets resembling classical four dimensional spacetime. In any case, it is clear that the naïve dynamical scheme quickly becomes impractical when trying to simulate causets of a physically interesting size—even a rather modest sized set with 64 points require millions of simulation cycles to approach something resembling an equilibrium, and a causet only twice as big does not thermalise even after two and a half *billion* cycles!

6.1.1 Comparing with the results of Henson et al.

Despite the slight difference in implementation, the results are also in general agreement with those of Henson et al. [13]. This provides another argument for suspecting that naïve dynamics in general leads to the creation of three-layered Kleitman–Rothschild-type sets, as the main goal of Henson et al. was to find a lower bound of when the asymptotic behavior of the theorem kicks in.

Among other things, Henson et al. find that thermalised causets with close to 30 points have an average height of about 4.5. This is in good agreement with figure 5.4, which shows the layer structure of the two resulting 32-point causets after the evolution shown in figure 5.1. For causets with 64 points Henson et al. finds an average height slightly larger than 3, while causets with 80 or more points have an average height which is very close to 3. This is more or less in agreement with figures 5.5 and 5.6, but while the results of Henson et al. are obtained trough averaging a large sampling of thermalised

causets, figures 5.5 and 5.6 only gives two isolated examples, and in the case of figure 5.6 the causets are not even thermalised!

When it comes to the mean ordering fractions, a small deviation from the results of Henson et al. is seen, but it is most pronounced for the smallest causet. Henson et al. finds a mean order fraction slightly less than 35 % for causets with 32 points, which is significantly lower than the (39.3 ± 1.4) % in figure 5.1. For sets with 64 points, Henson et al. finds an average ordering fraction of approximately 34 %. This is within the standard derivation of the average calculated in figure 5.2. Finally, for causets with 80 points, Henson et al. gets virtually the same average as they did for 64 points. This is slightly above the average found in figure 5.3, but given that the causets of this figure does not reach thermalisation, it is unknown to what degree this estimate reflects the actual average of the thermalised sets.

6.1.2 Stability

The code which produced the evolution shown in figure 5.7 required the causet to always remain an Alexandrov interval. This is different from the present definition of the naïve dynamical scheme, which considers all causets with a set number of points, regardless of type. However, any causet can be transformed into an Alexandrov interval by adding two points: one point which precedes all the other points, and one which is preceded by all the others. Hence, there is no reason to suspect that the present code is not equally stable as the one that produced figure 5.7, and the apparent equilibriums of figures 5.1 and 5.2 are therefore likely to be very robust.

Armed with the combined insight of the Kleitman–Rothschild theorem and figure 3.3, it is also possible to find *theoretical* reasons for why naïve dynamics ought to have a robust three-layered equilibrium for 'sufficiently large' causets. It is also possible to see why this equilibrium is harder to reach for larger sets, effectively branching of into several semi-equilibriums as the number of points increases.

Given an initially totally disordered causet, the situation shown in figure 3.3 gets less and less likely to occur as more and more layers are added to the set. This is provided that the layers are not simply 'stacked' on top of each other, consisting only of a disconnected ensemble of totally ordered subsets of approximately the same hight. However, as order relations are added randomly, this very special configuration is extremely unlikely to occur, at least if the causet is sufficiently large. Figure 3.3 does not prevent the formation of a two-layered causet from an initially ('one-layered') totally disordered set, but once the causet gets sufficiently causally it makes higher layers progressively more unlikely to be created. Thus a sufficiently large causet will get almost completely stuck already at approximately two layers. Reaching a hight of four layers will be practically impossible. This effect can be seen in practice already for 80 points by comparing figures 5.3 and 5.6.

What then about an initially totally *ordered* causet? The principle illustrated in figure 3.3 will make the initial disordering of the causet tend to

happen through growing 'vertical' tears (cf. figure 3.2a,b). This will slowly decrease the height of the causet, but as more and more ordering relations are removed, the likelihood of figure 3.3 working in the opposite direction—adding a relation rather than removing one—will increase. Hence the reduction of layers will not continue in perpetuity. The Kleitman–Rothschild theorem suggests that the eventual balance point should be reached somewhere near an approximately three-layered causet.

Hence both of the evolutions seen in figure 5.3 are explained.

6.2 Destructive dynamics

Destructive dynamics appear to lead to the production of an approximately two-layered causets, but the resulting causets are not a causally connected whole, but rather a collection of several smaller, isolated subsets. Most of the subsets are very small, consisting only of a handful of points, but one of the subsets are significantly greater than the others, consisting of well over half of all the points in the total causet.

Considering how the destructive dynamical scheme is defined, it should be no surprise that the resulting causet is highly disordered. The scheme minimizes the ordering fraction as measured when completely causally isolated points are ignored. Hence, the presence of several small causally isolated subsets is just as one would expect, and the two-layeredness is also, at least on a superficial level, not surprising. Moreover, when starting from an initially totally ordered causet, it is at least somewhat natural to assume that a larger, main subset remains for quite a while as the causet gets disordered and gradually broken up into smaller subsets, given that order relations are only allowed to be removed if they do not change others by transitivity. What the results reveal, then, is the overall point of balance at which the ordering fraction is thus minimized.

Assuming a causet with n points, the removal of an ordering relation which does not causally isolate any points leads to the ordering fraction decreasing by $2/n(n-1)$—see equation (2.7). If, on the other hand, *one* point gets causally isolated, the ordering fraction *increases* with a factor of $1+(2-n/R)/(n-2)$, where R is the number of order relations before the removal happened. This factor *must* be greater than or equal to one, since the fact that points gets removed if they are causally isolated requires $R \geq n/2 \implies 2-n/R \geq 0$. Given this, it is easy to see that if the removal causes *two* points to get causally isolated, the ordering fraction increases even more. This can only happen if there are isolated subsets consisting of only two points, something which is bound to eventually happen as smaller subsets gets isolated from the main causet.

Thus, there should be a point in the evolution where the combined likelihood of an increase in the ordering fraction due to the destruction of isolated two-point fragments or the removal of single nearly-isolated points overcomes the likelihood of continued decrease due to the removal of 'safe' order rela-

tions from the larger subsets. What the results therefore reveals is that this happens at a time where the main subset is still vastly larger than the isolated fragments, but not before all of the subsets have become nearly perfectly two-layered. Moreover, roughly half of the initial points are removed before this tipping point is reached.

While certainly an interesting result by itself, its greater significance seems somewhat moot, especially considering how little geometrical structure seem present in the resulting causet, even when restricted only to the main causally connected subset.

This is revealed from a comparison between figures 5.11 and 5.12. The points appear relatively tightly knit, but the differences between the two different distributions indicate a complicated and distinctly discrete net of anticonnections. The layers appears to have a somewhat well-defined center, but the distributions give no clear indication of dimension or other geometrical features. Additionally, the differences between the two extremal layers are rather pronounced if the causet should be seen as a 'purely spatial' cross section of spacetime.

Chapter 7

Closing remarks

It is well known that hindsight is 20/20. In retrospect, the results obtained by the simulations done as part of this thesis might seem rather intuitive. However, intuition is not always a reliable guide on the path to knowledge, so even if the broader conclusions drawn here might not seem so surprising after the fact, this does not in it self makes the journey worthless or pointless.

Where does the road go from here?

7.1 Possible code improvements

As was pointed out in section 5.2, the definition of anticonnected points used presently does not necessarily correspond to whether or not said points have a shared order relation, something which leads to a somewhat incomplete and unclear relation to the spatial distances present in the continuum limit.

Although this was partly remedied by the use of the relation matrix to provide the division into causally connected subsets, this could be completely remedied by redefining what a connected antichain is—see figure 7.1.

7.2 Regarding the continued hunt for physically sensible dynamics

From the theoretical discussion in section 6.1.2, it follows that the requirement of only changing one order relation at a time should probably be abandoned. However, the results of Henson et al., obtained by partly abandoning this requirement, shows that merely allowing for the *possibility* that several relations be changed simultaneously is not enough to combat the power of figure 3.3. It therefore seems plausible that the changing of only one order relation at a time should be more forbidden than encouraged in order to arrive at a physically meaningful causet.

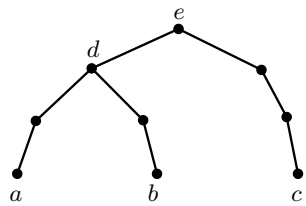


Figure 7.1: Using the definition of connected antichains given in section 2.2, there's no connected antichain between a and b, even though they both precede d. A natural extension would be to say that a and b are anticonnected through a minimal connected antichain of length 2. Likewise, a and c could be said to be anticonnected with a distance of 3. Alternatively, one could take the minimal sum of the maximal chains connecting two causally unrelated points together as the length of the connected antichain between the two points. Then, the 'spatial' distance between a and b would be 4, and the distance between a and c 6.

Bibliography

[1] Maqbool Ahmed and David P. Rideout. "Indications of de Sitter spacetime from classical sequential growth dynamics of causal sets". *Physical Review D* 81.8 (Apr. 2010), p. 083528. DOI: 10.1103/PhysRevD.81.083528.

[2] Jacob D. Bekenstein. "Black Holes and Entropy". *Physical Review D* 7.8 (Apr. 1973), pp. 2333–2346. DOI: 10.1103/PhysRevD.7.2333.

[3] Luca Bombelli, Rabinder K. Koul, Joohan Lee, and Rafael D. Sorkin. "Quantum source of entropy for black holes". *Physical Review D* 34.2 (July 1986), pp. 373–383. DOI: 10.1103/PhysRevLett.34.373.

[4] Luca Bombelli, Joohan Lee, David Meyer, and Rafael D. Sorkin. "Space-Time as a Causal Set". *Physical Review Letters* 59.5 (Aug. 1987), pp. 521–524. DOI: 10.1103/PhysRevLett.59.521.

[5] Graham Brightwell and Nicholas Georgiou. "Continuum limits for classical sequential growth models". *Random Structures & Algorithms* 36.2 (Mar. 2010), pp. 218–250. DOI: 10.1002/rsa.20278.

[6] Fay Dowker. "Introduction to causal sets and their phenomenology". *General Relativity and Gravitation* 45.9 (2012), pp. 1651–1667. DOI: 10.1007/s10714-013-1569-y.

[7] Fay Dowker. *Spacetime Atoms and the Unity of Physics*. Public lecture; video recording available on YouTube. Perimeter Institute for Theoretical Physics, Waterloo, Cananda. Nov. 2, 2011. URL: https://www.youtube.com/watch?v=VhHE86d-Th8 (visited on 10/30/2016).

[8] Benjamin F. Dribus. *On the Axioms of Causal Set Theory*. Dec. 2013. arXiv: 1311.2148v3 [gr-qc].

[9] Phillippe H. Eberhard and Ronald R. Ross. "Quantum field theory cannot provide faster-than-light communication". *Foundations of Physics Letters* 2.2 (Mar. 1989), pp. 127–149. DOI: 10.1007/BF00696109.

[10] Gary W. Gibbons and Stephen W. Hawking. "Action integrals and partition functions in quantum gravity". *Physical Review D* 15.10 (May 1977), pp. 2752–2756. DOI: 10.1103/PhysRevD.15.2752.

[11] Stephen W. Hawking. “Black holes and thermodynamics”. *Physical Review D* 13.2 (Jan. 1976), pp. 191–197. DOI: 10.1103/PhysRevD.13.191.

[12] Stephen W. Hawking, A. R. King, and P. J. McCarthy. “A new topology for curved space-time which incorporates the causal, differential, and conformal structures”. *Jorunal of Mathematical Physics* 17.2 (Feb. 1976), pp. 174–181. DOI: 10.1063/1.522874.

[13] Joe Henson, David P. Rideout, Rafael D. Sorkin, and Sumati Surya. “Onset of the Asymptotic Regime for (Uniformly Random) Finite Orders”. *Experimental Mathematics* (Aug. 2016), pp. 1–14. DOI: 10.1080/10586458.2016.1158134. Currently only published online.

[14] Gerard ’t Hooft. “Quantum Gravity: A fundamental problem and some radical ideas”. *Recent Developments in Gravitation. Cargèse 1978*. Ed. by Maurice Lévy and Stanley Deser. Plenum Press, 1979, pp. 323–345. ISBN: 0306401983.

[15] Ted Jacobson and Renaud Parentani. “Horizon Entropy”. *Foundations of Physics* 33.2 (Feb. 2003), pp. 323–348. DOI: 10.1023/A:1023785123428.

[16] Steven Johnston. “Qunatum Fields on Causal Sets”. PhD thesis. Imperial College London, Sept. 2010. arXiv: 1010.5514 [hep-th].

[17] Daniel J. Kleitman and Bruce L. Rothschild. “Asymptotic enumeration of partial orders on a finite set”. *Transactions of the American Mathematical Society* 205 (Apr. 1975), pp. 205–220. DOI: 10.1090/S0002-9947-1975-0369090-9.

[18] David B. Malament. “The class of continuous timelike curves determines the topology of spacetime”. *Journal of Mathematical Physics* 18.7 (July 1977), pp. 1399–1404. DOI: 10.1063/1.523436.

[19] David A. Meyer. “The Dimension of Causal Sets”. PhD thesis. Massachusetts Institute of Technology, 1989. URL: http://hdl.handle.net/1721.1/14328.

[20] Jan Myrheim. *Statistical geometry*. Preprint CERN-TH-2538. Geneva: CERN, Aug. 1978. URL: https://cds.cern.ch/record/293594.

[21] David P. Rideout and Rafael D. Sorkin. “Classical sequential growth dynamics for causal sets”. *Physical Review D* 61.2 (Dec. 1999), p. 024002. DOI: 10.1103/PhysRevD.61.024002.

[22] Rafael D. Sorkin. “Forks in the road, on the way to quantum gravity”. *International Journal of Theoretical Physics* 36.12 (1997), pp. 2759–2781.

[23] Sumati Surya. “Directions in Causal Set Quantum Gravity”. *Recent Research in Quantum Gravity*. Ed. by Arundhati Dasgupta. Nova Science Publishers, Inc, 2011, pp. 7–40. ISBN: 9781619423862.

Appendix A

C++ code used

The attached code utilizes parts of the C++11 standard library, and will most likely not compile if older C++ standards are used.

A.1 Core structures

A.1.1 The 'Narray' class template

Since Narray is a class template, all implementations are in the header file. Hence there is no accompanying source file.

Header file

```
#ifndef NARRAY_H_INCLUDED
#define NARRAY_H_INCLUDED

#include <utility>

template<class T>
class Narray{
private:
    T* theArray;
    int L; // The length of the array, i.e. the part of it which has
        data.
    int L_max; // The allocated, maximum length of the array.
public:
    Narray();
    Narray(const int N);
    Narray(const int N, const int length);
    Narray(const Narray<T>& other);
    void initialize();
    ~Narray();
    bool is_valid() const;
    void allocate(const int length);
    int length() const;
    int allocated_length() const;
    void push_back(T element);
```

```
    void erase(const int i); // Erases element i.
    void erase_element(T element); // Erases the given element if
        possible.
    bool contains_element(T element); // Checks if the given element
        already exists.
    void erase_all(); // Sets L=0; doesn't actually delete stuff.
    Narray<T>& operator=(Narray<T> rhs);
    T& operator[](const int i);
};


/* Implementations. */

template<class T>
Narray<T>::Narray(){
    initialize();
}

template<class T>
Narray<T>::Narray(const int N){
    L = 0;
    if(N<1){
        theArray = 0;
        L_max = 0;
    }else{
        theArray = new T[N]{};
        L_max = N;
    }
}

template<class T>
Narray<T>::Narray(const int N, const int length){
    L = length;
    theArray = new T[N]{};
    L_max = N;
}

template<class T>
Narray<T>::Narray(const Narray<T>& other){
    if(other.is_valid()){
        L = other.L;
        L_max = other.L_max;
        theArray = new T[L_max]{};
        for(int i=0; i<L; ++i){
            theArray[i] = other.theArray[i];
        }
    }else{
        initialize();
    }
}

template<class T>
void Narray<T>::initialize(){
    theArray = 0;
    L_max = 0;
    L = 0;
}
```

```
template<class T>
Narray<T>::~Narray(){
    delete [] theArray;
    theArray = 0;
    L = 0;
    L_max = 0;
}

template<class T>
bool Narray<T>::is_valid() const{
    return L_max!=0 && theArray!=0;
}

template<class T>
void Narray<T>::allocate(const int length){
    Narray<T> temp(*this);
    delete [] theArray;
    theArray = 0;
    L_max = length;
    theArray = new T[L_max]{};
    int i = 0;
    while(i<temp.L && i<L_max){
        theArray[i] = temp.theArray[i];
        ++i;
    }
    L = i;
}

template<class T>
int Narray<T>::length() const{
    return L;
}

template<class T>
int Narray<T>::allocated_length() const{
    return L_max;
}

template<class T>
void Narray<T>::push_back(T element){
    theArray[L] = element;
    ++L;
}

template<class T>
void Narray<T>::erase(const int i){
    theArray[i] = theArray[L-1];
    --L;
}

/* Note: The member-function "erase_element" only deletes the
   first instance of the given element, and must therefore be
   revoked several times in order to deal with duplicates.
*/
template<class T>
void Narray<T>::erase_element(T element){
```

```
    int i = 0;
    while(i<L){
        if(theArray[i]==element){
            erase(i);
            break;
        }else{
            ++i;
        }
    }
}

template<class T>
bool Narray<T>::contains_element(T element){
    int i = 0;
    while(i<L){
        if(theArray[i]==element){
            return true;
        }else{
            ++i;
        }
    }
    return false; // If this point is reached, the element is not in
        Narray.
}

template<class T>
void Narray<T>::erase_all(){
    L = 0;
}

template<class T>
Narray<T>& Narray<T>::operator=(Narray<T> rhs){
    std::swap(L, rhs.L);
    std::swap(L_max, rhs.L_max);
    std::swap(theArray, rhs.theArray);
    return *this;
}

template<class T>
T& Narray<T>::operator[](const int i){
    return theArray[i];
}

#endif // NARRAY_H_INCLUDED
```

A.1.2 The 'Event' class

Header file

```
#ifndef EVENT_H_INCLUDED
#define EVENT_H_INCLUDED

#include <iostream>
#include "narray.h"
```

```
class Event{
private:
    int N; // Total # of events (in the World). Must be >0.
public:
    int id; // Identifier. Must be >=0 and <N, in accordance with
        array-indexes.
    Narray<int> up_links; // Realized "upwards" links. Initially
        empty.
    Narray<int> down_links; // Realized "downwards" links. Initially
        empty.
    /* End of public variables. */
    Event();
    Event(const int n, const int N_max); // n = ID, N_max = # events
        in total.
    Event(const Event& other);
    void set_N(int new_N);
    bool up_linked(const int id_b);
    bool down_linked(const int id_b);
    void up_link(const int id_b);
    void down_link(const int id_b);
    Event& operator=(Event rhs);
    friend std::ostream& operator <<(std::ostream& os, Event& a);
};

#endif // EVENT_H_INCLUDED
```

Source file

```
#include "event.h"
#include "narray.h"
#include <iostream>
#include <utility>

using std::cout;
using std::endl;
using std::ostream;
using std::swap;

Event::Event(){
    N = 0;
    id = 0;
    up_links.initialize();
    down_links.initialize();
}

Event::Event(const int n, const int N_max){
    N = N_max;
    id = n;
    up_links.allocate(N);
    down_links.allocate(N);
}

Event::Event(const Event& other){
    N = other.N;
    id = other.id;
    up_links = other.up_links;
```

```
    down_links = other.down_links;
}

void Event::set_N(int new_N){
    N = new_N;
    up_links.allocate(N);
    down_links.allocate(N);
}

/* Note: The member functions "up_linked" and "down_linked" only
   returns TRUE if the corresponding lists for a actually contains
   b. Hence, they can not be used to check whether a and b ought
   to be linked (e.g. a.up_linked(b) cannot be used to check
   whether there's any point c such that a<c<b).
*/
bool Event::up_linked(const int id_b){ // Implies a < b.
    for(int i=0; i<up_links.length(); ++i){
        if(up_links[i] == id_b){
            return true;
        }
    }
    return false;
}

bool Event::down_linked(const int id_b){ // Implies b < a.
    for(int i=0; i<down_links.length(); ++i){
        if(down_links[i] == id_b){
            return true;
        }
    }
    return false;
}

void Event::up_link(const int id_b){
    bool in_a = false;
    for(int i=0; i<up_links.length(); ++i){
        if(up_links[i]==id_b){
            in_a = true;
            break;
        }
    }
    if(!in_a){
        up_links.push_back(id_b);
    }
}

void Event::down_link(const int id_b){
    bool in_a = false;
    for(int i=0; i<down_links.length(); ++i){
        if(down_links[i]==id_b){
            in_a = true;
            break;
        }
    }
    if(!in_a){
        down_links.push_back(id_b);
    }
```

```
}

Event& Event::operator=(Event rhs){
    swap(N,rhs.N);
    swap(id,rhs.id);
    swap(up_links,rhs.up_links);
    swap(down_links,rhs.down_links);
    return *this;
}

ostream& operator <<(ostream& os, Event& a){
    os << a.id;
    os << "\t|";
    for (int i=0; i<a.up_links.length(); ++i){
        os << a.up_links[i] << " ";
    }
    os << "\t|";
    for (int i=0; i<a.down_links.length(); ++i){
        os << a.down_links[i] << " ";
    }
    return os;
}
```

A.1.3 The 'Transition' class

Header file

```
#ifndef TRANSITION_H_INCLUDED
#define TRANSITION_H_INCLUDED

class Transition{
public:
    bool order;
    int id_a;
    int id_b;
    Transition();
    Transition(const bool add_link, const int a_id, const int b_id);
};

#endif // TRANSITION_H_INCLUDED
```

Source file

```
#include "transition.h"

Transition::Transition():order(false), id_a(0), id_b(0){
    // Intentionally left empty.
}

Transition::Transition(const bool add_link, const int a_id, const int
     b_id){
    order = add_link;
    id_a = a_id;
```

```
    id_b = b_id;
}
```

A.1.4 The 'World' class

Header file

```
#ifndef WORLD_H_INCLUDED
#define WORLD_H_INCLUDED

#include <iostream>
#include <string>
#include "narray.h"
#include "event.h"
#include "transition.h"

class World{
    friend class Worldview;
private:
    Narray<Event> the_World;
    int N; // Total # of events (in the World). Must be >0.
    int N_active; // active N used by dismantle functions and f().
    int N_links; // Total # of (up/down) links.
    Narray<Transition> possible_transitions;
    Narray<int> iterator_list; // iteration list used by the
        dismantle-functions and f(). Note that i = iterator_list[i+1]
         always gives next value of i.
    int** order_table; // Table over the orderings between a and b.
    /* Utility functions of different kinds. */
    void create_missing_links(const int id_a); // Links all events in
         a's lists to a.
    void delink_ordered_pair(const int id_a, const int id_b); //
        Assumes a<b. Does NOT update il and Na!
    void uplink(const int id_a, const int id_b); // Makes a<b. Does
        NOT update il and Na!
    void transition(Transition T); // Does NOT update
        possible_transitions!
    bool h_link_allowed(const int id_a, const int id_b); // Checks if
         a<b is allowed, given a!=b, a!<b, b!<a.
    void update_il_and_Na(); // Updates N_active and iterator_list.
    int count_up_links();
public:
    std::string id;
    World(const int sum_events); // sum_events = total # of events.
        Must be >0.
    World(std::string filename); // file-type should NOT be included!
    World(World& other);
    ~World();
    void order(); // Initializes causet to f=100%.
    void disorder(); // Initializes causet to f=0%.
    bool a_precedes_b(const int id_a, const int id_b);
    bool a_precedes_b_req(const int id_a, const int id_b);
    bool b_precedes_a_req(const int id_a, const int id_b);
    void evolve_and_save(const uint_fast64_t seed, const
        uint_fast64_t steps, const uint_fast64_t step_length, const
```

```
        uint_fast64_t save_interval); // Uses naive dynamics to
        evolve the causet.
    uint_fast32_t dismantle_and_record(const uint_fast64_t seed,
        const uint_fast32_t save_interval); // Removes links until no
         events are left. Returns the # of steps needed to reach
        f_min.
    void dismantle_and_save(const uint_fast64_t seed, const
        uint_fast32_t save_step, const uint_fast32_t save_interval);
    void set_id(std::string unique_id);
    void update_possible_transitions();
    double f(); // Returns the ordering fraction f relative N_active.
    double f_N(); // Returns the ordering fraction f relative N.
    void print_order_table();
    void save_order_table(const std::string post_id); // file-type
        should NOT be included!
    void save_to_file(const std::string post_id); // file-type should
         NOT be included!
    friend std::ostream& operator <<(std::ostream& os, World& w);
};

#endif // WORLD_H_INCLUDED
```

Source file

```
#include "world.h"
#include "transition.h"
#include "event.h"
#include "narray.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <random>
#include <cmath>
#include <stdlib.h>

using std::cout;
using std::endl;
using std::ostream;
using std::ofstream;
using std::ifstream;
using std::stringstream;
using std::string;
using std::mt19937_64;
using std::uniform_real_distribution;
using std::abs;

World::World(const int sum_events){
    N = sum_events;
    N_active = N;
    the_World.allocate(N);
    iterator_list.allocate(N+1);
    possible_transitions.allocate(N*N); // More than ever needed.
    for (int i=0; i<N; ++i){
        the_World.push_back(Event(i, N));
        iterator_list.push_back(i);
```

```
        if(i>0){
            the_World[i-1].up_links.push_back(i);
            the_World[i].down_links.push_back(i-1);
        }
    }
    iterator_list.push_back(N);
    order_table = new int*[N];
    order_table[0] = new int[N*N];
    for(int i=1; i<N; i++){
        order_table[i] = order_table[i-1] + N;
    }
    for(int i=0; i<N; ++i){
        for(int j=0; j<N; ++j){
            if(j>i){
                order_table[i][j] = 1;
            }else if(i==j){
                order_table[i][j] = 0;
            }else{
                order_table[i][j] = -1;
            }
        }
    }
    set_id("");
    N_links = count_up_links();
    update_possible_transitions();
}

World::World(string filename){
    id = filename;
    filename += ".world";
    string input = "";
    ifstream file (filename.c_str());
    if(file.is_open()){
        getline(file,input);
        N = atoi(input.c_str());
        N_active = N;
        the_World.allocate(N);
        iterator_list.allocate(N+1);
        possible_transitions.allocate(N*N); // More than ever needed.
        for(int i=0; i<N; ++i){
            the_World.push_back(Event(i, N));
            iterator_list.push_back(i);
        }
        iterator_list.push_back(N);
        int l = 0;
        int id_b = -1;
        for(int id_a=0; id_a<N; ++id_a){
            getline(file,input);
            l = atoi(input.c_str());
            for(int j=0; j<l; ++j){
                getline(file,input);
                id_b = atoi(input.c_str());
                the_World[id_a].up_links.push_back(id_b);
                the_World[id_b].down_links.push_back(id_a);
            }
        }
        file.close();
```

```
        order_table = new int*[N];
        order_table[0] = new int[N*N];
        for(int i=1; i<N; i++){
            order_table[i] = order_table[i-1] + N;
        }
        for(int i=0; i<N; ++i){
            for(int j=0; j<N; ++j){
                if(a_precedes_b_req(i,j)){
                    order_table[i][j]=1;
                }else if(b_precedes_a_req(i,j)){
                    order_table[i][j]=-1;
                }else{
                    order_table[i][j]=0;
                }
            }
        }
        update_il_and_Na();
        update_possible_transitions();
        N_links = count_up_links();
    }else{
        cout << "ERROR: Couldn't open " << filename << endl;
    }
}

World::World(World& other){
    N = other.N;
    iterator_list.allocate(N+1);
    possible_transitions.allocate(N*N);
    id = other.id;
    the_World = other.the_World;
    order_table = new int*[N];
    order_table[0] = new int[N*N];
    for(int i=1; i<N; i++){
        order_table[i] = order_table[i-1] + N;
    }
    for(int i=0; i<N; ++i){
        iterator_list.push_back(i);
        for(int j=0; j<N; ++j){
            if(a_precedes_b_req(i,j)){
                order_table[i][j]=1;
            }else if(b_precedes_a_req(i,j)){
                order_table[i][j]=-1;
            }else{
                order_table[i][j]=0;
            }
        }
    }
    iterator_list.push_back(N);
    update_il_and_Na();
    update_possible_transitions();
    N_links = count_up_links();
}

World::~World(){
    delete [] order_table[0];
    order_table[0] = 0;
    delete [] order_table;
```

```
    order_table = 0;
}

void World::order(){
    iterator_list.erase_all();
    N_active = N;
    for(int i=0; i<N; ++i){
        the_World[i].up_links.erase_all();
        the_World[i].down_links.erase_all();
        iterator_list.push_back(i);
        if(i>0){
            the_World[i-1].up_links.push_back(i);
            the_World[i].down_links.push_back(i-1);
        }
        for(int j=0; j<N; ++j){
            if(j>i){
                order_table[i][j] = 1;
            }else if(i==j){
                order_table[i][j] = 0;
            }else{
                order_table[i][j] = -1;
            }
        }
    }
    iterator_list.push_back(N);
    N_links = count_up_links();
}

void World::disorder(){
    for(int i=0; i<N; ++i){
        the_World[i].up_links.erase_all();
        the_World[i].down_links.erase_all();
        for(int j=0; j<N; ++j){
            order_table[i][j]=0;
        }
    }
    update_possible_transitions();
    update_il_and_Na();
    N_links = 0;
}

bool World::h_link_allowed(const int id_a, const int id_b){
    Narray<int>& a_d = the_World[id_a].down_links;
    Narray<int>& b_u = the_World[id_b].up_links;
    for(int j=0; j<a_d.length(); ++j){
        if(order_table[a_d[j]][id_b]!=1){
            return false;
        }
    }
    for(int j=0; j<b_u.length(); ++j){
        if(order_table[id_a][b_u[j]]!=1){
            return false;
        }
    }
    return true; // If all tests above are passed, this is the
        conclusion.
}
```

```
void World::create_missing_links(const int id_a){
    Narray<int> *a = &the_World[id_a].up_links;
    for(int i=0; i<(*a).length(); ++i){
        Event& c = the_World[(*a)[i]];
        if(!c.down_linked(id_a)){
            c.down_link(id_a);
        }
    }
    a = &the_World[id_a].down_links;
    for(int i=0; i<(*a).length(); ++i){
        Event& c = the_World[(*a)[i]];
        if(!c.up_linked(id_a)){
            c.up_link(id_a);
            ++N_links;
        }
    }
}

/* Note: The following member-function is NOT a simple table-
   lookup! It's purpose is to determine whether there's a chain
   of links connecting a and b, specially if the order_table
   already states that a<b (Thus, it might be used to "fill in"
   "missing links").
*/
bool World::a_precedes_b(const int id_a, const int id_b){
    if(id_a==id_b){
        // Intentionally left empty. This is simply a time-saver.
    }else{
        Narray<int>& a = the_World[id_a].up_links;
        Narray<int>& b = the_World[id_b].down_links;
        for(int i=0; i<a.length(); ++i){
            for(int j=0; j<b.length(); ++j){
                if(a[i]==id_b || b[j]==id_a){
                    return true;
                }
                else{
                    if(order_table[a[i]][id_b]==1){
                        return true;
                    }else if(order_table[id_a][b[j]]==1){
                        return true;
                    }
                }
            }
        }
    }
    return false;
}

/* Note: The following member-function checks whether a<b using
   the up-links lists, and can thus be used to check part of the
   consistency of the order_table.
*/
bool World::a_precedes_b_req(const int id_a, const int id_b){
    bool precedes = false;
    if(id_a == id_b){
        // Intentionally left empty. This is simply a time-saver.
```

```
    }else{
        Narray<int>& a = the_World[id_a].up_links;
        for(int i=0; i<a.length(); ++i){
            if(a[i] == id_b){
                return true;
            }else{
                precedes = a_precedes_b_req(a[i], id_b);
                if(precedes){
                    return true;
                }
            }
        }
    }
    return precedes;
}

/* Note: The following member-function checks whether a>b using
   the down-links lists, and can thus be used to check part of
   the consistency of the order_table.
*/
bool World::b_precedes_a_req(const int id_a, const int id_b){
    bool succeed = false;
    if(id_a == id_b){
        // Intentionally left empty. This is simply a time-saver.
    }else{
        Narray<int>& a = the_World[id_a].down_links;
        for(int i=0; i<a.length(); ++i){
            if(a[i] == id_b){
                return true;
            }else{
                succeed = b_precedes_a_req(a[i], id_b);
                if(succeed){
                    return true;
                }
            }
        }
    }
    return succeed;
}

/* Note: The following member-function assumes a<b.
*/
void World::delink_ordered_pair(const int id_a, const int id_b){
    Event& a = the_World[id_a];
    Event& b = the_World[id_b];
    // Start actual code. Above just for readability.
    order_table[id_a][id_b] = 0;
    order_table[id_b][id_a] = 0;
    a.up_links.erase_element(id_b);
    --N_links;
    for(int i=0; i<b.up_links.length(); ++i){
        if(!a_precedes_b(id_a, b.up_links[i])){
            a.up_link(b.up_links[i]);
            the_World[b.up_links[i]].down_link(id_a);
            ++N_links;
        }
    }
```

```
    b.down_links.erase_element(id_a);
    for(int i=0; i<a.down_links.length(); ++i){
        if(!a_precedes_b(a.down_links[i], id_b)){
            b.down_link(a.down_links[i]);
            the_World[a.down_links[i]].up_link(id_b);
            ++N_links;
        }
    }
}

void World::uplink(const int id_a, const int id_b){
    Event& a = the_World[id_a];
    Event& b = the_World[id_b];
    // Start actual code. Above just for readability.
    a.up_link(id_b);
    b.down_link(id_a);
    order_table[id_a][id_b] = 1;
    order_table[id_b][id_a] = -1;
    ++N_links;
    // Deletion of illegal links upwards from a or downwards from b.
    for(int i=0; i<b.up_links.length(); ++i){
        for(int j=0; j<a.up_links.length(); ++j){
            if(a.up_links[j]==b.up_links[i]){
                Event& c = the_World[a.up_links[j]];
                c.down_links.erase_element(id_a);
                a.up_links.erase(j);
                --N_links;
            }
        }
    }
    for(int i=0; i<a.down_links.length(); ++i){
        for(int j=0; j<b.down_links.length(); ++j){
            if(b.down_links[j]==a.down_links[i]){
                Event& c = the_World[b.down_links[j]];
                c.up_links.erase_element(id_b);
                b.down_links.erase(j);
                --N_links;
            }
        }
    }
}

void World::evolve_and_save(const uint_fast64_t seed, const
    uint_fast64_t steps, const uint_fast64_t step_length, const
    uint_fast64_t save_interval){
    mt19937_64 gen(seed); // Random number generator.
    uniform_real_distribution<double> rand_double(0, 1);
    stringstream post_id;
    post_id << ".f";
    int f_value = 100*f_N();
    if(f_value < 10){
        post_id << "00";
    }else if(f_value < 100){
        post_id << "0";
    }
    post_id << f_value << "(seed" << seed << ")c" << steps << "(" <<
        step_length << ")";
```

```
    string datafile_name = id + post_id.str() + ".txt";
    string savefile_post_id = post_id.str();
    double f_min = f_N();
    double f_max = f_min;
    Narray<double> temp(step_length);
    ofstream file (datafile_name.c_str());
    if(file.is_open()){
        file << 0 << "\t" << f_min << "\t" << f_max << endl;
        for(uint_fast64_t i=0; i<steps; ++i){
            int j = rand_double(gen) * possible_transitions.length();
            transition(possible_transitions[j]);
            update_possible_transitions();
            temp.push_back(f_N());
            if(i%step_length==step_length-1){
                f_min = temp[0];
                f_max = f_min;
                for(int j=1; j<temp.length(); ++j){
                    if(temp[j]<f_min){
                        f_min = temp[j];
                    }else if(temp[j]>f_max){
                        f_max = temp[j];
                    }
                }
                file << i+1 << "\t" << f_min << "\t" << f_max << endl
                    ;
                temp.erase_all();
            }
            if(i%save_interval==save_interval-1){
                save_to_file(savefile_post_id);
            }
        }
        file.close();
        save_to_file(savefile_post_id);
    }else{
        cout << "ERROR: Couldn't open " << datafile_name << endl;
    }
}

uint_fast32_t World::dismantle_and_record(const uint_fast64_t seed,
    const uint_fast32_t save_interval){
    mt19937_64 gen(seed); // Random number generator.
    uniform_real_distribution<double> rand_double(0, 1);
    stringstream post_idstrm;
    post_idstrm << ".dismantled(seed" << seed << ")";
    string info_file_name = id + post_idstrm.str() + ".info";
    string datafile_name = id + post_idstrm.str() + ".txt";
    string savefile_post_id = post_idstrm.str();
    string saveinfofile_name = id + post_idstrm.str() + ".saveinfo";
    uint_fast32_t i = 0; // Counter.
    double f_temp = f();
    double f_min = f_temp;
    uint_fast32_t i_min = 0;
    int N_f_min = N_active;
    ofstream file (datafile_name.c_str());
    if(file.is_open()){
        file << 0 << "\t" << f_temp << "\t" << N_active << endl;
        while(N_active>2){
```

```
            int n = rand_double(gen) * N_links + 1;
            int j = 0;
            int id_a = -1;
            while(n>j){
                id_a = iterator_list[id_a+1];
                j += the_World[id_a].up_links.length();
            }
            int k = 0;
            j -= the_World[id_a].up_links.length(); // Re-sets j.
            ++j; // Synchronizes j with n.
            while(n>j){
                ++k;
                ++j;
            }
            int id_b = the_World[id_a].up_links[k];
            delink_ordered_pair(id_a,id_b);
            update_il_and_Na();
            ++i;
            f_temp = f();
            if(f_temp < f_min){
                f_min = f_temp;
                i_min = i;
                N_f_min = N_active;
            }
            file << i << "\t" << f_temp << "\t" << N_active << endl;
            if(i%save_interval==0){
                save_to_file(savefile_post_id);
                ofstream info_file (saveinfofile_name.c_str());
                if(info_file.is_open()){
                    info_file << "Runs:\t" << i+1 << endl;
                    info_file.close();
                }
            }
        }
        file.close();
    }else{
        cout << "ERROR: Couldn't open " << datafile_name << endl;
    }
    file.open(info_file_name.c_str());
    if(file.is_open()){
        file << "Seed:\t\t\t" << seed << endl;
        file << "Total steps:\t\t" << i << endl;
        file << "-------------------------------------" << endl;
        file << "f_min:\t\t\t" << f_min << endl;
        file << "Step at which f=f_min:\t" << i_min << endl;
        file << "Active points at f_min:\t" << N_f_min << endl;
        file.close();
    }else{
        cout << "ERROR: Couldn't open " << info_file_name << endl;
    }
    return i_min;
}

void World::dismantle_and_save(const uint_fast64_t seed, const
    uint_fast32_t save_step, const uint_fast32_t save_interval){
    mt19937_64 gen(seed); // Random number generator.
    uniform_real_distribution<double> rand_double(0, 1);
```

```
    stringstream post_idstrm;
    post_idstrm << ".dismantled(seed" << seed << ")run" << save_step;
    string savefile_post_id = post_idstrm.str();
    string saveinfofile_name = id + post_idstrm.str() + ".saveinfo";
    uint_fast32_t i=0;
    while(i<save_step){
        int n = rand_double(gen) * N_links + 1;
        int j = 0;
        int id_a = -1;
        while(n>j){
            id_a = iterator_list[id_a+1];
            j += the_World[id_a].up_links.length();
        }
        int k = 0;
        j -= the_World[id_a].up_links.length(); // Re-sets j.
        ++j; // Synchronizes j with n.
        while(n>j){
            ++k;
            ++j;
        }
        int id_b = the_World[id_a].up_links[k];
        delink_ordered_pair(id_a,id_b);
        update_il_and_Na();
        if(i%save_interval==0){
            save_to_file(savefile_post_id);
            ofstream info_file (saveinfofile_name.c_str());
            if(info_file.is_open()){
                info_file << "Runs:\t" << i+1 << endl;
                info_file.close();
            }
        }
        ++i;
    }
    save_to_file(savefile_post_id);
    ofstream info_file (saveinfofile_name.c_str());
    if(info_file.is_open()){
        info_file << "f_min:\t\t" << f() << endl;
        info_file << "Runs:\t\t" << i << endl;
        info_file << "Active points:\t" << N_active << endl;
        info_file << "-------------------------------------" << endl;
        info_file << "Minimum f reached." << endl;
        info_file.close();
    }
}

void World::set_id(string unique_id){
    stringstream tempid;
    tempid << N;
    tempid << unique_id;
    id = tempid.str();
}

void World::transition(Transition T){
    if(T.order){
        uplink(T.id_a, T.id_b);
    }else{
        delink_ordered_pair(T.id_a, T.id_b);
```

```
    }
}

void World::update_possible_transitions(){
    possible_transitions.erase_all();
    for(int i=0; i<N; ++i){
        Event& a = the_World[i];
        // Update the list of possible transitions.
        for(int j=0; j<a.up_links.length(); ++j){
            Transition temp(false, i, a.up_links[j]);
            possible_transitions.push_back(temp);
        }
        for(int j=0; j<N; ++j){
            if(i!=j && order_table[i][j]==0){
                if(h_link_allowed(i,j)){
                    Transition temp(true, i, j);
                    possible_transitions.push_back(temp);
                }
            }
        }
    }
}

void World::update_il_and_Na(){
    N_active = 0;
    for(int i = 0; i<N; ++i){
        int j = i;
        while(i<N && the_World[i].up_links.length()==0 && the_World[i
            ].down_links.length()==0){
            ++i;
        }
        iterator_list[j] = i;
        if(i<N){
            ++N_active;
        }
    }
}

double World::f(){
    double x = 0.0;
    for(int i=iterator_list[0]; i<N;){
        for(int j=iterator_list[0]; j<N;){
            if(order_table[i][j]==1){
                ++x;
            }
            j = iterator_list[j+1];
        }
        i = iterator_list[i+1];
    }
    double y = N_active*(N_active-1.0)/2.0;
    return x/y;
}

double World::f_N(){
    double x = 0.0;
    for(int i=0; i<N; ++i){
        for(int j=0; j<N; ++j){
```

```
            if(order_table[i][j]==1){
                ++x;
            }
        }
    }
    double y = N*(N-1.0)/2.0;
    return x/y;
}

int World::count_up_links(){
    int total_links = 0;
    for(int i=0; i<N; ++i){
        total_links += the_World[i].up_links.length();
    }
    return total_links;
}

void World::print_order_table(){
    for(int i=0; i<N; ++i){
        cout << i << "\t|";
        for(int j=0; j<N; ++j){
            if(j!=0){
                cout << " ";
            }
            if(order_table[i][j]!=-1){
                cout << " ";
            }
            cout << order_table[i][j];
        }
        cout << "|" << endl;
    }
}

void World::save_order_table(const string post_id){
    string filename = id + post_id + ".txt";
    ofstream file (filename.c_str());
    if(file.is_open()){
        for(int i=0; i<N; ++i){
            file << i << "\t|";
            for(int j=0; j<N; ++j){
                if(j!=0){
                    file << " ";
                }
                if(order_table[i][j]!=-1){
                    file << " ";
                }
                file << order_table[i][j];
            }
            file << "|" << endl;
        }
        file.close();
    }else{
        cout << "ERROR: Couldn't open " << filename << endl;
    }
}

void World::save_to_file(const string post_id){
```

```
    string filename1 = id + post_id + ".world.txt"; // Redundant file
        . Formatted for human eyes.
    string filename2 = id + post_id + ".world";
    ofstream file1 (filename1.c_str());
    ofstream file2 (filename2.c_str());
    if(file1.is_open()){
        if(file2.is_open()){
            file2 << N << endl;
            for(int i=0; i<N; ++i){
                Narray<int>& i_up_links = the_World[i].up_links;
                file1 << i << "\t|";
                file2 << i_up_links.length() << endl;
                for(int j=0; j<i_up_links.length(); ++j){
                    file1 << i_up_links[j];
                    file2 << i_up_links[j] << endl;
                    if(j<i_up_links.length()-1){
                        file1 << ", ";
                    }
                }
                file1 << endl;
            }
            file2.close();
        }else{
            cout << "ERROR: Couldn't open " << filename2 << endl;
        }
        file1.close();
    }else{
        cout << "ERROR: Couldn't open " << filename1 << endl;
    }
}

ostream& operator <<(ostream& os, World& w){
    for(int i=0; i<w.N; i++){
        os << w.the_World[i] << endl;
    }
    return os;
}
```

A.2 Analytical tools

A.2.1 The 'Worldview' class

Header file

```
#ifndef WORLDVIEW_H_INCLUDED
#define WORLDVIEW_H_INCLUDED

#include "narray.h"
#include "world.h"
#include <string>

class Worldview: public World{
private:
    Narray<Narray<int>> temporal_layer; // A list of the id-s of
        events belonging to temporal layer n.
```

```
    Narray<Narray<int>> group; // A list of the id-s of events
        belonging to group n.
    Narray<int> minimal_layer;
    Narray<int> maximal_layer;
    int** shared_up_links;
    int** shared_down_links;
    int** distance_table_min_layer;
    int** distance_table_max_layer;
    /* Utility functions of different kinds. */
    void initialize_groups();
    void initialize_temporal_layer();
public:
    Worldview(World& basis);
    ~Worldview();
    void save_info_file(const std::string post_id);
    void save_layers(const std::string post_id); // Saves the
        elements of the temporal layers to a text file.
    void save_layer_plot(const std::string post_id, const
        uint_fast64_t seed); // Uses a random spatial coordinate.
    void save_shared_links_plots(const std::string post_id);
    void save_distance_tables(const std::string post_id);
    void save_distance_histograms(const std::string post_id);
    void save_distance_histograms(const int group_id, const std::
        string post_id); // Provides histograms per shell.
    void save_layer_histogram(const std::string post_id);
    void save_max_distance_histograms(const int group_id, const std::
        string post_id); // Histogram of longest minimal connected
        antichains.
};

#endif // WORLDVIEW_H_INCLUDED
```

Source file

```
#include "worldview.h"
#include "narray.h"
#include "world.h"
#include <cmath> // sqrt.
#include <fstream> // ofstream, ifstream.
#include <iostream> // cout, endl.
#include <random> // mt19937_64, uniform_real_distribution.
#include <string>
#include <sstream> // stringstream.

using std::cout;
using std::endl;
using std::mt19937_64;
using std::ofstream;
using std::string;
using std::stringstream;
using std::uniform_real_distribution;

/*  When creating a Worldview from a World, any isolated events
    get removed. As a part of this process, the remaining events
    are renamed so that they can easily be iterated through with-
    out having to use the cumbersome iterator list and N_active
```

```
    defined in World. Hence, these variables are completely
    ignored by the constructor below.
*/
Worldview::Worldview(World& basis): World(basis.N_active){
    id = basis.id;
    temporal_layer.allocate(N);
    minimal_layer.allocate(N);
    maximal_layer.allocate(N);
    int j = 0; // Counter. Could also be used for consistency-
        checking.
    int old_N = basis.N;
    Narray<int> old_ids(old_N); // List used to temporarily link "old
        " and "new" event ID-s.
    shared_down_links = new int*[N];
    shared_up_links = new int*[N];
    distance_table_min_layer = new int*[N];
    distance_table_max_layer = new int*[N];
    shared_down_links[0] = new int[N*N];
    shared_up_links[0] = new int[N*N];
    distance_table_min_layer[0] = new int[N*N];
    distance_table_max_layer[0] = new int[N*N];
    for(int i=1; i<N; i++){
        shared_down_links[i] = shared_down_links[i-1] + N;
        shared_up_links[i] = shared_up_links[i-1] + N;
        distance_table_min_layer[i] = distance_table_min_layer[i-1] +
             N;
        distance_table_max_layer[i] = distance_table_max_layer[i-1] +
             N;
    }
    for(int i=0; i<old_N; ++i){
        old_ids.push_back(-1); // Temporary placeholders.
    }
    for(int i=basis.iterator_list[0]; i<old_N;){
        Event temp(basis.the_World[i]);
        temp.set_N(N);
        the_World[j] = temp;
        old_ids[i] = j;
        ++j;
        i = basis.iterator_list[i+1];
    }
    for(int i=0; i<N; ++i){ // Renames all link-IDs, and identifies
        top- and bottom-layer events.
        the_World[i].id = i;
        for(int j=0; j<the_World[i].down_links.length(); ++j){ // Re-
            use of j.
            the_World[i].down_links[j] = old_ids[the_World[i].
                down_links[j]];
        }
        if(the_World[i].down_links.length()==0){
            minimal_layer.push_back(i);
        }
        for(int j=0; j<the_World[i].up_links.length(); ++j){ // Re-
            use of j.
            the_World[i].up_links[j] = old_ids[the_World[i].up_links[
                j]];
        }
        if(the_World[i].up_links.length()==0){
```

```
            maximal_layer.push_back(i);
        }
    }
    for(int i=0; i<N; ++i){
        for(j=0; j<N; ++j){ // Re-use of j.
            if(a_precedes_b_req(i,j)){
                order_table[i][j]=1;
            }else if(b_precedes_a_req(i,j)){
                order_table[i][j]=-1;
            }else{
                order_table[i][j]=0;
            }
            // Update shared-links-tables.
            int sum_down = 0;
            int sum_up = 0;
            if(i!=j){
                for(int k=0; k<the_World[i].down_links.length(); ++k)
                    {
                    int id_a = the_World[i].down_links[k];
                    if(the_World[j].down_links.contains_element(id_a)
                        ){
                        ++sum_down;
                    }
                }
                for(int k=0; k<the_World[i].up_links.length(); ++k){
                    int id_a = the_World[i].up_links[k];
                    if(the_World[j].up_links.contains_element(id_a)){
                        ++sum_up;
                    }
                }
            }
            shared_down_links[i][j] = sum_down;
            shared_up_links[i][j] = sum_up;
            // Initialize distance-tables. -1=\inf, -2=should not be
                included, -3=undetermined.
            if(i!=j){
                if(the_World[i].down_links.length()==0 && the_World[j
                    ].down_links.length()==0){
                    distance_table_max_layer[i][j] = -2;
                    if(shared_up_links[i][j]>0){
                        distance_table_min_layer[i][j] = 1;
                    }else{
                        distance_table_min_layer[i][j] = -3;
                    }
                }else if(the_World[i].up_links.length()==0 &&
                    the_World[j].up_links.length()==0){
                    distance_table_min_layer[i][j] = -2;
                    if(shared_down_links[i][j]>0){
                        distance_table_max_layer[i][j] = 1;
                    }else{
                        distance_table_max_layer[i][j] = -3;
                    }
                }else{
                    distance_table_min_layer[i][j] = -2;
                    distance_table_max_layer[i][j] = -2;
                }
            }else{
```

```
                if(the_World[i].down_links.length()==0 && the_World[j
                    ].down_links.length()==0){
                    distance_table_min_layer[i][j] = 0;
                    distance_table_max_layer[i][j] = -2;
                }else if(the_World[i].up_links.length()==0 &&
                    the_World[j].up_links.length()==0){
                    distance_table_min_layer[i][j] = -2;
                    distance_table_max_layer[i][j] = 0;
                }else{
                    distance_table_min_layer[i][j] = -2;
                    distance_table_max_layer[i][j] = -2;
                }
            }
        }
    }
    // Update distance-tables.
    for(int i=0; i<N; ++i){
        int neighbors_min = 0;
        int neighbors_max = 0;
        for(j=0; j<N; ++j){ // Re-use of j.
            if(distance_table_min_layer[i][j]==1){
                ++neighbors_min;
            }else if(distance_table_max_layer[i][j]==1){
                ++neighbors_max;
            }
        }
        if(neighbors_min==0){
            for(j=0; j<N; ++j){ // Re-use of j.
                if(distance_table_min_layer[i][j]==-3){
                    distance_table_min_layer[i][j] = -1;
                }
                if(distance_table_min_layer[j][i]==-3){
                    distance_table_min_layer[j][i] = -1;
                }
            }
        }
        if(neighbors_max==0){
            for(j=0; j<N; ++j){ // Re-use of j.
                if(distance_table_max_layer[i][j]==-3){
                    distance_table_max_layer[i][j] = -1;
                }
                if(distance_table_max_layer[j][i]==-3){
                    distance_table_max_layer[j][i] = -1;
                }
            }
        }
    }
    bool finished = false;
    int r = 1; // Distance.
    while(!finished){
        for(int i=0; i<N; ++i){
            for(j=0; j<N; ++j){ // Re-use of j.
                if(distance_table_min_layer[i][j]==r){
                    for(int k=0; k<N; ++k){
                        if(distance_table_min_layer[j][k]==1){
                            if(distance_table_min_layer[i][k]==-3){
                                distance_table_min_layer[i][k] = r+1;
```

```
                            }
                        }
                    }
                }else if(distance_table_max_layer[i][j]==r){
                    for(int k=0; k<N; ++k){
                        if(distance_table_max_layer[j][k]==1){
                            if(distance_table_max_layer[i][k]==-3){
                                distance_table_max_layer[i][k] = r+1;
                            }
                        }
                    }
                }
            }
        }
        // Check if tables are finished.
        finished = true;
        for(int i=0; i<N; ++i){
            for(j=0; j<N; ++j){ // Re-use of j.
                if(distance_table_min_layer[i][j]==-3){
                    if(r<N){
                        finished = false;
                        goto check_done;
                    }else{
                        distance_table_min_layer[i][j] = -1;
                    }
                }else if(distance_table_max_layer[i][j]==-3){
                    if(r<N){
                        finished = false;
                        goto check_done;
                    }else{
                        distance_table_max_layer[i][j] = -1;
                    }
                }
            }
        }
        check_done:
        ++r;
    }
    initialize_groups();
    initialize_temporal_layer();
}

Worldview::~Worldview(){
    delete [] shared_up_links[0];
    shared_up_links[0] = 0;
    delete [] shared_up_links;
    shared_up_links = 0;
    delete [] shared_down_links[0];
    shared_down_links[0] = 0;
    delete [] shared_down_links;
    shared_down_links = 0;
    delete [] distance_table_max_layer[0];
    distance_table_max_layer[0] = 0;
    delete [] distance_table_max_layer;
    distance_table_max_layer = 0;
    delete [] distance_table_min_layer[0];
    distance_table_min_layer[0] = 0;
```

```
    delete [] distance_table_min_layer;
    distance_table_min_layer = 0;
}

void Worldview::initialize_groups(){
    Narray<int> group_list(N);
    for(int i=0; i<N; ++i){
        group_list.push_back(-1);
    }
    int N_groups = 0;
    for(int i=0; i<N; ++i){
        if(group_list[i]==-1){
            ++N_groups;
            if(distance_table_min_layer[i][i]==0){
                for(int j=0; j<N; ++j){
                    if(distance_table_min_layer[i][j]>=0){ // This
                        will also assign a group_list to event i.
                        group_list[j] = N_groups-1;
                    }
                }
                for(int j=0; j<N; ++j){
                    for(int k=0; k<N; ++k){
                        if(group_list[k]==group_list[i]){
                            if(a_precedes_b(k,j)){
                                group_list[j] = N_groups-1;
                            }
                        }
                    }
                }
            }else if(distance_table_max_layer[i][i]==0){
                for(int j=0; j<N; ++j){
                    if(distance_table_max_layer[i][j]>=0){ // This
                        will also assign a group to event i.
                        group_list[j] = N_groups-1;
                    }
                }
                for(int j=0; j<N; ++j){
                    for(int k=0; k<N; ++k){
                        if(group_list[k]==group_list[i]){
                            if(a_precedes_b(j,k)){
                                group_list[j] = N_groups-1;
                            }
                        }
                    }
                }
            }else{ // There might be events which are neither in the
                top ~ nor bottom layer!
                group_list[i] = N_groups-1;
                for(int j=0; j<N; ++j){
                    if(a_precedes_b(j,i)){
                        group_list[j] = N_groups-1;
                    }else if(a_precedes_b(i,j)){
                        group_list[j] = N_groups-1;
                    }
                }
                for(int j=0; j<N; ++j){
                    for(int k=0; k<N; ++k){
```

```
                        if(group_list[k]==group_list[i]){
                            if(a_precedes_b(j,k)){
                                group_list[j] = N_groups-1;
                            }else if(a_precedes_b(k,j)){
                                group_list[j] = N_groups-1;
                            }
                        }
                    }
                }
            }
        }
    }
    group.allocate(N_groups);
    Narray<int> temp(N);
    for(int n=0; n<N_groups; ++n){
        group.push_back(temp);
        for(int i=0; i<N; ++i){
            if(group_list[i]==n){
                group[n].push_back(i);
            }
        }
    }
    // Sort the groups according to size using insertion sort.
    for(int i=1; i<N_groups; ++i){
        temp = group[i];
        int j = i-1;
        while(temp.length()>group[j].length() && j>0){
            --j;
        }
        for(int k=i-1; k>j; --k){
            group[k+1] = group[k];
        }
        if(temp.length()>group[j].length()){
            group[j+1] = group[j];
            group[j]=temp;
        }else{
            group[j+1]=temp;
        }
    }
}

void Worldview::initialize_temporal_layer(){
    Narray<int> temp(N);
    // Establish the bottom layer, equal to the minimal layer.
    temporal_layer.push_back(temp);
    for(int i=0; i<N; ++i){
        if(the_World[i].down_links.length()==0){
            temporal_layer[0].push_back(i);
        }
    }
    // Attach remaining points, and move them up to their correct
        layer.
    for(int i=1; i<N; ++i){
        temporal_layer.push_back(temp);
        for(int j=0; j<temporal_layer[i-1].length(); ++j){ // Re-use
            of j.
            int id_a = temporal_layer[i-1][j];
```

```
            for(int k=0; k<the_World[id_a].up_links.length(); ++k){
                int id_b = the_World[id_a].up_links[k];
                if(!temporal_layer[i].contains_element(id_b)){
                    temporal_layer[i].push_back(id_b);
                }
            }
        }
    }
    // Erase misplaced, duplicated id-s, and empty layers.
    for(int i=N-1; i>0; --i){
        if(temporal_layer[i].length()!=0){
            for(int j=0; j<temporal_layer[i].length(); ++j){ // Re-
                use of j.
                int id_a = temporal_layer[i][j];
                for(int k=i-1; k>-1; --k){
                    temporal_layer[k].erase_element(id_a); // Deletes
                         id_a iff temporal_layer[k] contains it.
                }
            }
        }else{
            temporal_layer.erase(i); // Deletes all empty layers (
                which MUST be at the top).
        }
    }
}

void Worldview::save_info_file(const string post_id){
    string filename = id + post_id + ".info";
    ofstream file (filename.c_str());
    if(file.is_open()){
        int N_b = minimal_layer.length();
        int N_t = maximal_layer.length();
        file << "Bottom layer points [N_b]:\t" << N_b << endl;
        file << "Top layer points [N_t]:\t\t" << N_t << endl;
        file << "Total points [N]:\t\t" << N << endl;
        file << endl << "Ratio [N_b/N_t]:\t\t" << (1.0*N_b)/(1.0*N_t)
             << endl;
        file << "Ratio [N_b/N]:\t\t\t" << (1.0*N_b)/(1.0*N) << endl;
        file << "Ratio [N_t/N]:\t\t\t" << (1.0*N_t)/(1.0*N) << endl;
        file << endl << endl << "------------------------------";
        file << "------------------------------" << endl;
        file << "Group\t\t#total\t\t#bottom\t\t#top" << endl;
        file << "------------------------------";
        file << "------------------------------" << endl;
        for(int i=0; i<group.length(); ++i){
            file << i << "\t\t" << group[i].length() << "\t\t";
            N_b = 0; // Re-use.
            N_t = 0; // Re-use.
            for(int j=0; j<group[i].length(); ++j){
                int id_a = group[i][j];
                if(distance_table_min_layer[id_a][id_a]==0){
                    ++N_b;
                }else if(distance_table_max_layer[id_a][id_a]==0){
                    ++N_t;
                }
            }
            file << N_b << "\t\t" << N_t << endl;
```

```
        }
        file << "------------------------------";
        file << "------------------------------" << endl;
        file.close();
    }else{
        cout << "ERROR: Couldn't open " << filename << endl;
    }
}

void Worldview::save_layers(const string post_id){
    string filename = id + post_id + ".txt";
    ofstream file (filename.c_str());
    if(file.is_open()){
        file << "Layer\tPoints" << endl;
        file << "------------------------------";
        file << "------------------------------" << endl;
        for(int i=0; i<temporal_layer.length(); ++i){
            file << i << "\t";
            int j = 0;
            for(;j<temporal_layer[i].length()-1; ++j){
                file << temporal_layer[i][j] << ", ";
            }
            file << temporal_layer[i][j] << endl;
        }
        file << "------------------------------";
        file << "------------------------------" << endl;
        file.close();
    }else{
        cout << "ERROR: Couldn't open " << filename << endl;
    }
}

void Worldview::save_layer_plot(const string post_id, const
    uint_fast64_t seed){
    string filename = id + post_id + ".plot";
    ofstream file (filename.c_str());
    if(file.is_open()){
        mt19937_64 gen(seed); // Random number generator.
        uniform_real_distribution<double> rand_double(0, 1);
        for(int i=0; i<N; ++i){
            int t_i = 0;
            for(int j=0; j<temporal_layer.length(); ++j){
                if(temporal_layer[j].contains_element(i)){
                    t_i = j;
                    break;
                }
            }
            double x_i = 0.5*N*(2*rand_double(gen) - 1);
            file << x_i << "\t" << t_i << endl;
        }
        file.close();
    }else{
        cout << "ERROR: Couldn't open " << filename << endl;
    }
}

void Worldview::save_shared_links_plots(const string post_id){
```

```
    string filename_ul = id + post_id + "_up.plot";
    string filename_dl = id + post_id + "_down.plot";
    string infofilename = id + post_id + ".info";
    int total_ul = 0;
    int total_dl = 0;
    double N_ul = N - maximal_layer.length();
    double N_dl = N - minimal_layer.length();
    ofstream file_ul (filename_ul.c_str());
    if(file_ul.is_open()){
        ofstream file_dl (filename_dl.c_str());
        if(file_dl.is_open()){
            for(int i=0; i<N; ++i){
                for(int j=0; j<N; ++j){
                    if(distance_table_max_layer[i][j]==-2){
                        int sum = shared_up_links[i][j];
                        total_ul += sum;
                        file_ul << i << "\t" << j << "\t" << sum <<
                            endl;
                    }else if(distance_table_min_layer[i][j]==-2){
                        int sum = shared_down_links[i][j];
                        total_dl += sum;
                        file_dl << i << "\t" << j << "\t" << sum <<
                            endl;
                    }
                }
            }
            file_dl.close();
        }else{
            cout << "ERROR: Couldn't open " << filename_dl << endl;
        }
        file_ul.close();
        file_ul.open(infofilename.c_str()); // Re-use.
        if(file_ul.is_open()){
            double mean_ul_1 = 1.0*total_ul/(N_ul*N_ul-N_ul);
            double mean_ul_2 = 1.0*total_ul/N_ul;
            double mean_dl_1 = 1.0*total_dl/(N_dl*N_dl-N_dl);
            double mean_dl_2 = 1.0*total_dl/N_dl;
            file_ul << "Average shared up-links between two bottom
                layer-points:\t" << mean_ul_1 << endl;
            file_ul << "Average shared down-links between two top
                layer-points:\t\t" << mean_dl_1 << endl << endl;
            file_ul << "Average neighbors per bottom layer-point:\t\t
                \t" << mean_ul_2 << endl;
            file_ul << "Average neighbors per top layer-point:\t\t\t\
                t" << mean_dl_2 << endl;
            file_ul.close();
        }else{
            cout << "ERROR: Couldn't open " << infofilename << endl;
        }
    }else{
        cout << "ERROR: Couldn't open " << filename_ul << endl;
    }
}

void Worldview::save_distance_tables(const string post_id){
    string filename_b = id + post_id + "_minimal_layer.plot";
    string filename_t = id + post_id + "_maximal_layer.plot";
```

```
    ofstream file_b (filename_b.c_str());
    int N_groups = group.length();
    if(file_b.is_open()){
        ofstream file_t (filename_t.c_str());
        if(file_t.is_open()){
            int i_b = 1; // i_b,j_b and i_t,j_t are used to rename
                the points of the plots in order to avoid problems
                with "missing points".
            int i_t = 1;
            for(int i=0; i<N_groups; ++i){
                for(int j=0; j<group[i].length(); ++j){
                    int id_a = group[i][j];
                    int j_b = 1;
                    int j_t = 1;
                    for(int k=0; k<N_groups; ++k){
                        for(int l=0; l<group[k].length(); ++l){
                            int id_b = group[k][l];
                            if(distance_table_min_layer[id_a][id_b
                                ]!=-2){
                                file_b << i_b << "\t" << j_b << "\t"
                                    << distance_table_min_layer[id_a
                                    ][id_b] << endl;
                                ++j_b;
                            }else if(distance_table_max_layer[id_a][
                                id_b]!=-2){
                                file_t << i_t << "\t" << j_t << "\t"
                                    << distance_table_max_layer[id_a
                                    ][id_b] << endl;
                                ++j_t;
                            }
                        }
                    }
                    if(j_b>1){
                        ++i_b;
                    }
                    if(j_t>1){
                        ++i_t;
                    }
                }
            }
            file_t.close();
        }else{
            cout << "ERROR: Couldn't open " << filename_t << endl;
        }
        file_b.close();
    }else{
        cout << "ERROR: Couldn't open " << filename_b << endl;
    }
}

void Worldview::save_distance_histograms(const string post_id){
    string filename_b = id + post_id + "_minimal_layer.plot";
    string filename_t = id + post_id + "_maximal_layer.plot";
    ofstream file_b (filename_b.c_str());
    if(file_b.is_open()){
        ofstream file_t (filename_t.c_str());
        if(file_t.is_open()){
```

```
int N_b = minimal_layer.length();
int N_t = maximal_layer.length();
Narray<int> sum_b (N_b);
Narray<int> sum_t (N_t);
for(int r=1; r<N; ++r){
    sum_b.erase_all();
    sum_t.erase_all();
    int total_sum_b = 0;
    int total_sum_t = 0;
    for(int i=0; i<N; ++i){
        int temp_b = 0;
        int temp_t = 0;
        for(int j=0; j<N; ++j){
            if(distance_table_min_layer[i][j]==r){
                ++total_sum_b;
                ++temp_b;
            }else if(distance_table_max_layer[i][j]==r){
                ++total_sum_t;
                ++temp_t;
            }
        }
        if(temp_b>0){
            sum_b.push_back(temp_b);
        }
        if(temp_t>0){
            sum_t.push_back(temp_t);
        }
    }
    if(total_sum_b>0){
        long double mean = 1.0*total_sum_b/N_b;
        long double sigma = 0.0;
        int i=0;
        for(; i<sum_b.length(); ++i){
            long double temp = 1.0*sum_b[i] - mean;
            sigma += temp*temp;
        }
        for(; i<N_b; ++i){
            sigma += mean*mean;
        }
        sigma = sqrt(sigma/N_b); // Makes sigma equal to
            the standard derivation.
        long double mean_min = mean - sigma;
        long double mean_max = mean + sigma;
        if(mean_min<0){ // mean_min<0 makes no physical
            sense.
            mean_min = 0;
        }
        file_b << r << "\t" << mean << "\t" << mean_min
            << "\t" << mean_max << endl;
    }
    if(total_sum_t>0){
        long double mean = 1.0*total_sum_t/N_t;
        long double sigma = 0.0;
        int i=0;
        for(; i<sum_t.length(); ++i){
            long double temp = 1.0*sum_t[i] - mean;
            sigma += temp*temp;
```

```
                    }
                    for(; i<N_t; ++i){
                        sigma += mean*mean;
                    }
                    sigma = sqrt(sigma/N_t); // Makes sigma equal to
                        the standard derivation.
                    long double mean_min = mean - sigma;
                    long double mean_max = mean + sigma;
                    if(mean_min<0){ // mean_min<0 makes no physical
                        sense.
                        mean_min = 0;
                    }
                    file_t << r << "\t" << mean << "\t" << mean_min
                        << "\t" << mean_max << endl;
                }
            }
            file_t.close();
        }else{
            cout << "ERROR: Couldn't open " << filename_t << endl;
        }
        file_b.close();
    }else{
        cout << "ERROR: Couldn't open " << filename_b << endl;
    }
}

void Worldview::save_distance_histograms(const int group_id, const
    string post_id){
    int N_b = 0;
    int N_t = 0;
    for(int i=0; i<group[group_id].length(); ++i){
        int id_a = group[group_id][i];
        if(distance_table_min_layer[id_a][id_a]==0){
            ++N_b;
        }else if(distance_table_max_layer[id_a][id_a]==0){
            ++N_t;
        }
    }
    Narray<Narray<int>> shell_b(N_b);
    Narray<Narray<int>> shell_t(N_t);
    int i_shell_b = 0;
    int i_shell_t = 0;
    Narray<int> temp_b(N_b);
    Narray<int> temp_t(N_t);
    shell_b.push_back(temp_b);
    shell_t.push_back(temp_t);
    for(int r=0; r<N; ++r){
        int sum_b = 0;
        int sum_t = 0;
        for(int i=0; i<N; ++i){
            if(group[group_id].contains_element(i)){
                int r_max = 0;
                if(distance_table_min_layer[i][i]==0){
                    for(int j=0; j<N; ++j){
                        if(distance_table_min_layer[i][j]>r_max){
                            r_max = distance_table_min_layer[i][j];
                        }
```

```
                    }
                    if(r_max==r){
                        shell_b[i_shell_b].push_back(i);
                        ++sum_b;
                    }
                }else if(distance_table_max_layer[i][i]==0){
                    for(int j=0; j<N; ++j){
                        if(distance_table_max_layer[i][j]>r_max){
                            r_max = distance_table_max_layer[i][j];
                        }
                    }
                    if(r_max==r){
                        shell_t[i_shell_t].push_back(i);
                        ++sum_t;
                    }
                }
            }
        }
        if(sum_b>0){
            shell_b.push_back(temp_b);
            ++i_shell_b;
        }
        if(sum_t>0){
            shell_t.push_back(temp_t);
            ++i_shell_t;
        }
    }
    shell_b.erase(shell_b.length()-1); // Deletes the last element,
        which is empty.
    shell_t.erase(shell_t.length()-1); // Deletes the last element,
        which is empty.
    for(int i=0; i<shell_b.length(); ++i){
        stringstream addendum;
        addendum << "(Group" << group_id << "Shell" << i << ")";
        string filename = id + addendum.str() + post_id + "
            _minimal_layer.plot";
        ofstream file (filename.c_str());
        if(file.is_open()){
            Narray<int> sum(N);
            N_b = shell_b[i].length(); // Re-use;
            for(int r=1; r<N; ++r){
                sum.erase_all();
                int total_sum = 0;
                for(int j=0; j<N; ++j){
                    if(shell_b[i].contains_element(j)){
                        int temp = 0;
                        for(int k=0; k<N; ++k){
                            if(distance_table_min_layer[j][k]==r){
                                ++total_sum;
                                ++temp;
                            }
                        }
                        if(temp>0){
                            sum.push_back(temp);
                        }
                    }
                }
```

```
                if(total_sum>0){
                    long double mean = 1.0*total_sum/N_b;
                    long double sigma = 0.0;
                    int j=0;
                    for(; j<sum.length(); ++j){
                        long double temp = 1.0*sum[j] - mean;
                        sigma += temp*temp;
                    }
                    for(; j<N_b; ++j){
                        sigma += mean*mean;
                    }
                    sigma = sqrt(sigma/N_b); // Makes sigma equal to
                        the standard derivation.
                    long double mean_min = mean - sigma;
                    long double mean_max = mean + sigma;
                    if(mean_min<0){ // mean_min<0 makes no physical
                        sense.
                        mean_min = 0;
                    }
                    file << r << "\t" << mean << "\t" << mean_min <<
                        "\t" << mean_max << endl;
                }
            }
            file.close();
        }else{
            cout << "ERROR: Couldn't open " << filename << endl;
        }
    }
    for(int i=0; i<shell_t.length(); ++i){
        stringstream addendum;
        addendum << "(Group" << group_id << "Shell" << i << ")";
        string filename = id + addendum.str() + post_id + "
            _maximal_layer.plot";
        ofstream file (filename.c_str());
        if(file.is_open()){
            Narray<int> sum(N);
            N_t = shell_t[i].length(); // Re-use;
            for(int r=1; r<N; ++r){
                sum.erase_all();
                int total_sum = 0;
                for(int j=0; j<N; ++j){
                    if(shell_t[i].contains_element(j)){
                        int temp = 0;
                        for(int k=0; k<N; ++k){
                            if(distance_table_max_layer[j][k]==r){
                                ++total_sum;
                                ++temp;
                            }
                        }
                        if(temp>0){
                            sum.push_back(temp);
                        }
                    }
                }
                if(total_sum>0){
                    long double mean = 1.0*total_sum/N_t;
                    long double sigma = 0.0;
```

```
                    int j=0;
                    for(; j<sum.length(); ++j){
                        long double temp = 1.0*sum[j] - mean;
                        sigma += temp*temp;
                    }
                    for(; j<N_t; ++j){
                        sigma += mean*mean;
                    }
                    sigma = sqrt(sigma/N_t); // Makes sigma equal to
                        the standard derivation.
                    long double mean_min = mean - sigma;
                    long double mean_max = mean + sigma;
                    if(mean_min<0){ // mean_min<0 makes no physical
                        sense.
                        mean_min = 0;
                    }
                    file << r << "\t" << mean << "\t" << mean_min <<
                        "\t" << mean_max << endl;
                }
            }
            file.close();
        }else{
            cout << "ERROR: Couldn't open " << filename << endl;
        }
    }
}

void Worldview::save_layer_histogram(const string post_id){
    string filename = id + post_id + ".plot";
    ofstream file (filename.c_str());
    if(file.is_open()){
        for(int i=0; i<temporal_layer.length(); ++i){
            file << i << "\t" << temporal_layer[i].length() << endl;
        }
        file.close();
    }else{
        cout << "ERROR: Couldn't open " << filename << endl;
    }
}

void Worldview::save_max_distance_histograms(const int group_id,
    const string post_id){
    stringstream addendum;
    addendum << "(Group" << group_id << ")";
    string filename_b = id + addendum.str() + post_id + "
        _minimal_layer.plot";
    string filename_t = id + addendum.str() + post_id + "
        _maximal_layer.plot";
    ofstream file_b (filename_b.c_str());
    if(file_b.is_open()){
        ofstream file_t (filename_t.c_str());
        if(file_t.is_open()){
            for(int r=0; r<N; ++r){
                int sum_b = 0;
                int sum_t = 0;
                for(int i=0; i<N; ++i){
                    if(group[group_id].contains_element(i)){
```

```
                        int r_max = 0;
                        if(distance_table_min_layer[i][i]==0){
                            for(int j=0; j<N; ++j){
                                if(distance_table_min_layer[i][j]>
                                    r_max){
                                    r_max = distance_table_min_layer[
                                        i][j];
                                }
                            }
                            if(r_max==r){
                                ++sum_b;
                            }
                        }else if(distance_table_max_layer[i][i]==0){
                            for(int j=0; j<N; ++j){
                                if(distance_table_max_layer[i][j]>
                                    r_max){
                                    r_max = distance_table_max_layer[
                                        i][j];
                                }
                            }
                            if(r_max==r){
                                ++sum_t;
                            }
                        }
                    }
                }
                if(sum_b>0){
                    file_b << r << "\t" << sum_b << endl;
                }
                if(sum_t>0){
                    file_t << r << "\t" << sum_t << endl;
                }
            }
            file_t.close();
        }else{
            cout << "ERROR: Couldn't open " << filename_t << endl;
        }
        file_b.close();
    }else{
        cout << "ERROR: Couldn't open " << filename_b << endl;
    }
}
```

A.3 Some usage examples

The following main file creates the log files necessary to recreate figure 5.2, and in doing so also saves the two causets which are closer examined in figure 5.5:

```
#include "narray.h"
#include "event.h"
#include "transition.h"
#include "world.h"
#include "worldview.h"

int main(){
```

```
    int N = 64;
    uint_fast64_t steps = 10000000;
    uint_fast64_t step_l = 20000;
    uint_fast64_t save_int = 1000000;
    uint_fast64_t seed = 90073;
    World theMatrix(N);
    theMatrix.evolve_and_save(seed,steps,step_l,save_int);
    theMatrix.disorder();
    theMatrix.set_id("");
    theMatrix.evolve_and_save(seed,steps,step_l,save_int);
    return 0;
}
```