Protocol Synthesis in the Dynamic Gossip Problem

Leo Poulson

April 24, 2019

Contents

1 Introduction

Planning is the task of computing a set of actions to take some state within a model to a successful state. In such a model we have a set of *actors*, who will perform these actions that we search for.

Planning, as put forward in Ghallab, Nau, and Traverso 2004, covers a wide range of applications; from making a robot enter a room to synchronising multiple computers working on the same document. In this project we concern ourselves with a special case of planning; namely *epistemic planning*. In this we begin with a state embodying the knowledge states for a set of agents. We then want to compute a set of actions the agents may perform in order to take the knowledge state of the agents to a successful one.

Epistemic planning is a relatively young area of research, first put forward in Bolander and Birkegaard 2011. It rests upon work done in Dynamic Epistemic Logic, which is a logic used to reason about knowledge and how it is changed by actions. Despite epistemic planning being proved to be undecidable in a lot of cases [1], interest in the area has been maintained since its birth (Aucher, Maubert, and Pinchinat 2014), leading to the work that this project uses.

The gossip problem is a problem regarding peer-to-peer information sharing. A set of agents start out with some *secret information*, and their goal is to transfer this information across the network, such that every other agent finds out their secret. We call an agent who knows the secret of every other agent to be an *expert*; we want to reach a state where every agent is an expert. When two agents communicate they tell each other all of the secrets they know; hence the slightly frivolous but very apt title of the *gossip* problem.

The gossip problem was first put forward in Tijdeman 1971, and was such that each agent began with the phone number of every other agent. We can hence visualise the set of agents as nodes and the fact of agent a knowing the phone number of b as a directed edge, and so such a formulation would be a complete graph as in Figure 1a.

(a) *An example of a gossip graph in the classical formulation.*

(b) *An example of an incomplete gossip graph.*

Figure 1: *Some exemplar gossip graphs.*

Recently a variation of this problem has been studied, entitled the *dynamic* gossip problem(van Ditmarsch et al. 2018, van Ditmarsch et al. 2016). This is a popular research topic,

[1] It is undecidable for models where the number of agents is greater than 1 (Aucher and Bolander 2013), and also in single-agent models where the accessibility relation is not an equivalence relation. It is however decidable for models where the event models are propositional (Yu, Wen, and Liu 2013).

with applications in the study of epidemics and information discovery (Haeupler et al. 2016, **Find some more refs**). In these scenarios our network may model some peer to peer network where our agents are computers, where the goal is to find the IP addresses of all of the other computers in the network; or a social network, where our nodes are people who want to connect with every other user in their circle of friends.

In this, we begin with each agent knowing some subset of the phone numbers of every other agent. Hence a starting configuration of the dynamic gossip problem may look like Figure 1b. When two agents converse on the phone, they also exchange all of the phone numbers that they know as well as all of the secrets. As such, the graph's edges increase in number as phone calls occur.

1.1 Existing Software

As research work is done into a topic, it is only natural that software is developed to support it. The three related pieces of software are van Eijck 2014a, Gattinger 2018c and Gattinger 2018a. These three tools are all *model checkers* for dynamic epistemic logic. A model checker is a program that, given a model $\mathcal{M}$ and a formula ϕ, tells the user whether or not $\mathcal{M} \models \phi$. Given the setting, they specifically answer the question "given an initial state and a set of events, does the state reached by the occurence of these events at the initial state satisfy some property?".

We can see model checking as the *dual* of planning; in planning we want to find such a sequence of events, whilst model checking verifies that these events do indeed bring the system to a successful state. Gattinger 2018a is also capable of planning, however in a naïve way; it enumerates the set of all possible calls and then checks if any of these are successful. As we see later, this is a very inefficient method for planning.

Despite this, there does not exist any software to solve the epistemic planning problem as envisioned in the literature; indeed, the only software to solve any epistemic planning problem is restricted to just gossip models.

1.2 Contributions

At a high level, our aim in this project is to put forward a process to solve the planning problem for epistemic models and propositional event models. This is motivated by the dynamic gossip problem; despite the literature on the topic there is little work done on the topic of planning for it. Dynamic Gossip can be modelled perfectly by epistemic models and propositional event models, and hence is a very fitting case study for the tool we will develop. Although our algorithm is designed to work for any models, it is hoped that the work done in this thesis can shed light on how a specialised process could be designed for the dynamic gossip problem.

Our process uses the algorithm outlined in Aucher, Maubert, and Pinchinat 2014. This paper provides an outline of an algorithm which could be used, however much work remains to be done in order to make this outline into an implementable algorithm.

As a testament to the functionality of the algorithm, we give an implementation of the process in Haskell. There is currently no existing algorithm or software that performs planning for epistemic models.

This process will entail use of automata, a method discussed in the literature however yet to be formalised to the point at which it can be implemented, nor implemented. The literature

surrounding the task spans from epistemic logic to game theory, and as such is far outside the reach of the standard degree content. In this thesis we hope to present all of the related literature in a simple manner, so that the reader may understand all of the concepts used.

This project puts forward several challenges. The first challenge is compiling the assorted literature on the topic. As we come to see later, the literature stretches very far and wide; the collection, comprehension and recollection of this literature is a substantial piece of work in itself.

The second is the act of designing the algorithm that we will use to solve the planning problem. Whilst the algorithm is roughly laid out in Aucher, Maubert, and Pinchinat 2014 and Maubert and Pinchinat 2014, this is but a sketch. There still remains a lot of work to be done on it in order to make it an easy to understand and implement process. The development of this forms our second challenge.

The third is the implementation in Haskell of the designed process. Due to Haskell's syntactic similarity to the mathematical notation we use to formalise the problem[2] this is easier than it would be in some languages, however this still has its difficulties.

1.3 Aims

Concretely, our aims are as follows:

- Research and survey the literature on the topic of planning and the gossip problem, with an aim to combining them.
- Review the extant software on model checking and planning for epistemic models, with an aim to drawing comparions between the existing tools and the one developed.
- Design an algorithm to solve the planning problem for epistemic models and propositional event models.
- Provide an implementation of the developed algorithm in Haskell.
- Design and implement a testing system, to check the correctness of the program and also aid with profiling.
- Perform a study of performance in terms of time and space, and explain any strengths or shortcomings in comparison to existing tools.

[2]This similarity is discussed again later, and is one of the reasons Haskell was chosen for the task.

2 Background

2.1 Dynamic Epistemic Logic

2.1.1 Epistemic Logic

Epistemic Logic is the logical language that we use to reason about knowledge. It is a modal logic; this is a **type / class** of logic that supplements the language of propositional logic[3] with some operator $\boxdot$. Modal logic can be used to model the passage of time, knowledge, obligation or any other modality. For example, $\boxdot\phi$ in temporal logic can be read as "at all points in the future, ϕ holds"; in deontic (obligation) logic, we can read $\boxdot\phi$ as "it is morally necessary that ϕ holds". We see that the addition of the $\boxdot$ operator provides us with much more expressivity than the standard language of propositional logic.

In this thesis, we concern ourselves with *epistemic* logic; here we interpret $\boxdot\phi$ as "it is known that ϕ holds". However, we change the operator slightly; we index it with the name of an agent, and we write it as $K_i\phi$[4] Hence, we interpret $K_i\phi$ as "Agent i knows that ϕ is true.".

The essential reference on epistemic logic is Fagin et al. 1995, and it is from here that most of the information in this section comes.

If we have some set of agents A and some set of propositions Λ, then we define the language of epistemic logic over this set of propositions, $\mathcal{L}(\Lambda)$, with the following BNF;

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid K_i\phi$$

where $p \in \Lambda$ and $i \in A$. We can also define the dual to K, in the classic way as $\widehat{K}_i\phi ::= \neg K_i \neg\phi$. We read this as "agent i considers it possible that ϕ is true". We can give our epistemic logic a semantics through use of *Kripke models*.

We refer to $\mathcal{L}(\Lambda)$ without the knowledge operator K_i as $\mathcal{L}_P(\Lambda)$.

A Kripke model $\mathcal{M}$ over a set of agents I and a set of propositions Λ is a triple (W, R, V), where W is a **(maybe)** finite set of worlds, R is a set of binary relations over W indexed by an agent, such that $R_i \subseteq W \times W$, and $V : W \to \mathcal{P}(\Lambda)$ is a valuation function that associates to every world in W some set of propositions that are true at it.

For epistemic logic, we think of R_i as being the set of pairs of worlds that an agent i cannot distinguish between, and thus considers possible. In our semantics, we use this relation to define knowledge in terms of possibility; agent i knows that something is true if it is true at all of the worlds that i cannot distinguish between.

When giving a semantics to a formula on a Kripke model, we need to use a *pointed Kriple model*. This is just a pair $(\mathcal{M}, w)$ where w is a world of $\mathcal{M}$. Then we read $(\mathcal{M}, w) \models \phi$ as "w satisfies ϕ". We define the evaluation as follows:

[3] By which we mean the language defined by the grammar $\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi$, as well as the abbreviations $\bot, \vee, \to$ as is classic.

[4] Where the K stands for **K**nows.

$(\mathcal{M}, w) \models \top$
$(\mathcal{M}, w) \models p$ iff $p \in V(w)$
$(\mathcal{M}, w) \models \neg\phi$ iff $(\mathcal{M}, w) \not\models \phi$
$(\mathcal{M}, w) \models \phi \wedge \psi$ iff $(\mathcal{M}, w) \models \phi$ and $(\mathcal{M}, w) \models \psi$
$(\mathcal{M}, w) \models K_i\phi$ iff for all v such that $(w, v) \in R_i, (\mathcal{M}, v) \models \phi$

We need to give some properties to our knowledge operator in order to better understand it. We make all of our relations R_i equivalence relations. This means three things;

- R_i is *reflexive*; for all $w \in W$, $(w, w) \in R_i$.
- R_i is *symmetric*; for all $w, v \in W, (w, v) \in R_i$ iff $(v, w) \in R_i$.
- R_i is *transitive*; for all $w, v, u \in W$ if $(w, v) \in R_i$ and $(v, u) \in R_i$, then $(w, u) \in R_i$.

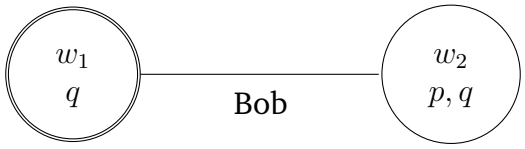
This is done in order to convey that agent i considers world v possible from world w if in both w and v agent i has the same information; that is, they are indistinguishable to the agent.

It is identical to say that our relations R_i are equivalence relations, as it is to say that our model is an S5 model. This is defined as a model in which the modal operator K obeys the following axioms:

- K: $K(\phi \to \psi) \to (K\phi \to K\psi)$
- T: $K\phi \to \phi$
- 5: $\widehat{K}\phi \to K\widehat{K}\phi$

These axioms hold independent of which agent's knowledge we are reasoning about.

Let's now look at an example S5 Kripke model.



In this epistemic model, we have two agents, Alice and Bob, and two worlds, w_1 and w_2. Alice can distinguish between w_1 and w_2, and as such has no lines on the diagram. However Bob cannot, hence the line connecting w_1 and w_2. At w_1 the proposition q is true, and at w_2 propositions p and q are true. Our "actual" world is w_1, and as such it is double ringed.

To show some examples, let's evaluate some formulae on this model.

- $(\mathcal{M}, w_1) \models q$ as at w_1, q is true.
- $(\mathcal{M}, w_1) \models K_{\text{Alice}}q$. To check that this is true, let's consider all of the worlds Alice considers possible from w_1. The only world is w_1 itself, and we know from above that $(\mathcal{M}, w_1) \models q$. Hence $K_{\text{Alice}}q$ holds here.
- $(\mathcal{M}, w_1) \models \neg K_{\text{Bob}}(p \wedge q)$. Now let's consider all the worlds that Bob considers possible from w_1. This is both w_1 and w_2. But we see that $(\mathcal{M}, w_1) \not\models (p \wedge q)$, since $(\mathcal{M}, w_1) \not\models p$.

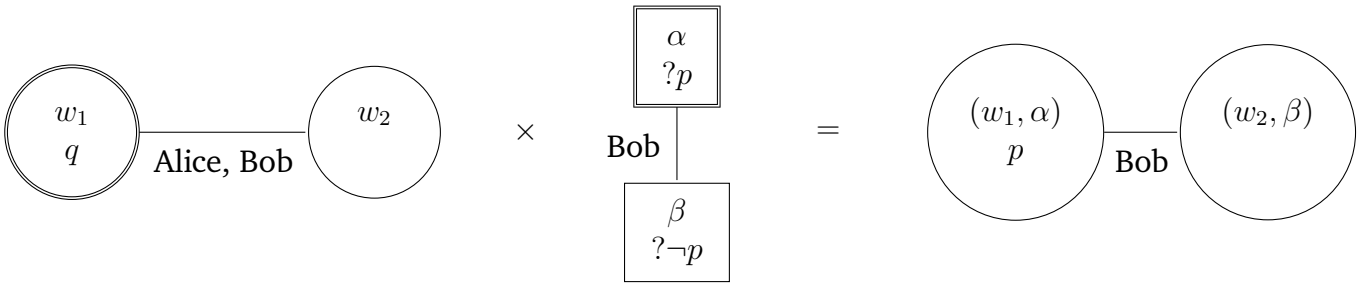


Figure 2

2.1.2 Event Models

Public announcement logic was the first development in epistemic logic to support information change in epistemic logic, and is described in Plaza 2007. However, in public annoucement logic, we may only model truthful, public communications; but we all know that there are many other types of communication and events after which some information has changed. An agent may communicate with another agent in a private phone call, or may be lying to them; furthermore, we may have some *physical* action, like a coin flip occuring, after which some knowledge has changed.

To model these more complicated events, we use *event models*. These treat events in a very similar way to how Kripke models treat worlds; we think of a set of possible events that can occur, and encode the events that an agent can tell apart. We give the modern definition of event models, as in Gattinger 2018b, and Aucher, Maubert, and Pinchinat 2014, however these were first defined in **Cite 1998 Baltag paper**

Formally, an event model $\mathcal{E}$ is a tuple $(E, Q, \mathsf{pre}, \mathsf{post})$. E is a finite set of events; Q is a set of relations Q_i for each $i \in Ag$, such that $Q_i \subseteq E \times E$. As before, we make all relations Q equivalence relations. $\mathsf{pre} : E \to \mathcal{L}(\Lambda)$ is the *precondition function*; given an event $e \in E$, it returns to us a formula that must be true in order for e to occur. $\mathsf{post} : E \times \Lambda \to \mathcal{L}(\Lambda)$ is the *postcondition function*; given an event $e \in E$ and a proposition $p \in \Lambda$, it returns to us some formula that had to be true at the prior state s in order for p to be true after event e occurs at s. We will later give examples for these two functions.

Updating a Kripke model $\mathcal{M}$ with an event model $\mathcal{E}$ is gives us another Kripke model $\mathcal{M} \times \mathcal{E} ::= (W', R', V')$, where:

$$
\begin{aligned}
W' &::= \{(w, e) \in W \times E \mid (\mathcal{M}, w) \models \mathsf{pre}(e)\} \\
R'_i &::= \{((w, e), (v, f)) \mid (w, v) \in R_i, (e, f) \in Q_i\} \\
V'(w, e) &::= \{p \in \Lambda \mid (\mathcal{M}, w) \models \mathsf{post}(e, p)\}
\end{aligned}
$$

We see that it is the postcondition function that allows for factual change; that is, the update of what is true at a state given the occurence of some event.

As above, we now show some example event models, and show the effects of updating an epistemic model with an event model.

In this example, consider two agents, again Alice and Bob. They are waiting in the same room to receive a letter about Alice's entrance onto a PhD program. Suddenly the postman arrives, and Bob sees Alice pick up and open a letter with the University's logo on the front. Alice reads the letter and learns that she gets the position, but she does not tell Bob the result. Bob sees that Alice now knows whether she got it, but he does not know if she did get it or not.

Here, p stands for the proposition "Alice gets the position". This initial situation is represented by model $\mathcal{M}$. Both Alice and Bob cannot distinguish between the worlds where p is true and where p is not true.

Then the event model $\mathcal{E}$ represents the event of Alice reading the letter from the unviersity. α represents the event of Alice getting the position and β that she doesn't - and hence the preconditions are p and $\neg p$ respectively; that is, $\mathsf{pre}(\alpha) = p$ and $\mathsf{pre}(beta) = \neg p$. Bob cannot distinguish between either of these events, but Alice can.

Then finally the event model $\mathcal{M} \times \mathcal{E}$ represents the situation in $\mathcal{M}$ after the event $\mathcal{E}$ has occurred. We can see the epistemic model $\mathcal{M}$, event model $\mathcal{E}$ and updated model $\mathcal{M} \times \mathcal{E}$ in Figure 2.

As a second example, consider that Alice and Bob are in a room. There's a coin on the table with the heads face up - we set this to be represented by the proposition p. Alice and Bob can both see the coin face up. This situation is modelled by event model $\mathcal{M}$.

Then Bob picks up the coin and flips it in the air. Bob sees the result of this coin flip, but does not show Alice. We can see the event model and epistemic model, and their update in Figure **??**.

Our epistemic model has just one state - that is, the state where the coin is on the table face up. Both Alice and Bob can see the coin, and as such they both know that p holds. Then our event model has two events; α, in which the coin lands face-up, and β, in which the coin lands face-down. Bob can distinguish between these two events as he can see the result of the coin toss, but Alice cannot, as she does not see it. In the former p is set to true, and in the latter p is set to false. Hence in the epistemic model $(\mathcal{M} \times \mathcal{E})$ we have two states; that in which the coin landed face-up and that in which the coin landed face-down.

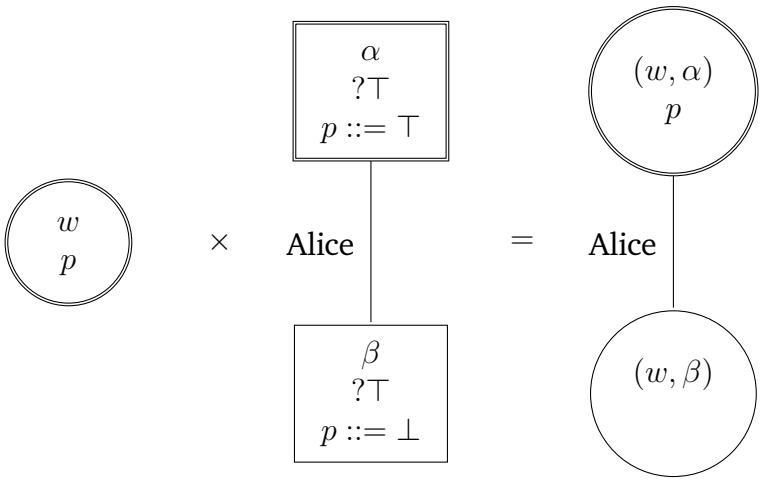


Figure 3: *An event model and epistemic model for the coin toss example, and the updated event model.*

2.2 The Gossip Problem

Gossip is a procedure for spreading secrets around a group of agents, where the agents are commonly displayed as nodes in a graph and the ability of one agent to contact another displayed as an edge between two nodes. Gossip was first put forward in Tijdeman 1971, where the network is a complete graph; all agents can contact one another. One question here was to find how many calls are needed for every agent to learn the secret of every other agent. We will henceforth describe an agent knowing the secret of every other agent as this agent being an *expert*. It was quickly proved (A. Hajnal and Szemere 1972, Baker and Shostak 1972) that this number, for a network where the number of agents n is greater than 4, is $2n - 4$.

2.2.1 Dynamic Gossip

Dynamic Gossip is a variant of the classical gossip problem, in which we start off with an incomplete graph, representing the fact that the agents have only the phone numbers of some of the other agents. In this case, when two agents talk on the phone, they also exchange all of the phone numbers that they know as well as the secrets that they know.

2.2.2 Protocols

The gossip problem, as we have mentioned so far, can rely on some central scheduler to tell the agents what call to make, and lets them know when to stop. However in distributed computing, methods that do not require this central authority are desirable. In such a situation, the agents need to have some form of rules to follow to decide how to behave and who to call next. This is the motivation for a *gossip protocol*, which are short conditions that must be fulfilled in order for an agent to make a specific call. These protocols were first proposed in R. Apt, Grossi, and Hoek 2016, Attamah et al. 2014, and some exemplar ones are:

- ANY - If x knows the phone number of y, x may call y.
- LNS - If x knows the phone number of y and x does not know the secret of y, then x may call y.

In this thesis, we do not study the distributed gossip problem; rather, a version with a central authority that surveys the network topology and then decides which agent is allowed to make a call, and also who they will call. However, these protocols do still have use for us. ANY allows for infinite call sequences - for example, one where agent a just repeatedly calls b, whereas all of the call sequences induced by LNS are of finite length. In the long run this does not really matter; both ANY and LNS have runtime expected execution length in $O(n \log n)$ (van Ditmarsch et al. 2018), yet in our implementation tests performed with LNS took considerably less time than ANY. For this pragmatic reason LNS is often used in this thesis.

It should also be noted that there are certain classes of graphs for which LNS cannot induce a successful call sequence, yet ANY can (van Ditmarsch et al. 2018). However, this thesis does not investigate this topic, and as such this will no longer be mentioned.

2.2.3 Formalisation

We now go on to formalise some of the ideas mentioned so far in this section. The definitions of gossip graphs are classic and can be found in all of the related literature (e.g. van Ditmarsch et al. 2018, Gattinger 2018b)

We formally denote a gossip graph $\mathcal{G}$ with a triple (A, N, S). A is a finite set of agents in the graph; $N \subseteq A \times A$ is a set of ordered pairs of agents such that $(u, v) \in N$ (or Nuv) iff u knows the phone number of v. $S \subseteq A \times A$ is a set of ordered pairs of agnets such that $(u, v) \in S$ (or Suv) iff u knows the *secret* of v.

(a) *An exemplar gossip graph.* **(b)** *A Kripke model $\mathcal{G}$ for a Gossip State*

Figure 4

We also have that for all gossip graphs, for any agent a, $(a, a) \in N$ and $(a, a) \in S$, expressing that any agent always knows their own phone number and secret.

We can use the notation above to formally describe a gossip graph. For instance, the gossip graph that we can see in Figure 4a can be expressed as the triple $\mathcal{G} = (\{a, b, c\}, \{(a, a), (a, b), (b, a), (b, b), (b, c), (c, c)\}, \{(a, a), (b, b), (c, c)\})$. However given that we have the proprety described above, we can represent this as the more terse $\mathcal{G} = (\{a, b, c\}, \{(a, b), (b, a), (b, c)\}, \{\})$

One might ask how gossip fits into the picture we drew above of Kripke models and event models. We can see a gossip state as a very boring Kripke model drawn in Figure 4b, where we have just one state, which is the one that we are in. We can then create an event model that enumerates all of the calls in this example, which can be seen in Figure 5[5].

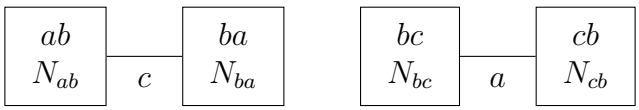


Figure 5: *An Event model $\mathcal{E}_\mathcal{G}$ for the calls associated with Figure 4a.*

In Figure 6 we can see the result of updating the Kripke model $\mathcal{G}$ with the event model $\mathcal{E}_\mathcal{G}$. We see that the call cb does not have a corresponding state; its precondition was not satisfied by any of the states in $\mathcal{G}$. Also note that the states (w, ab) and (w, ba) are indistinguishable to c; this is because c cannot distinguish between calls ab and ba as c is not included in it. Also note

[5]We omit the two calls ac and ca to make the example shorter, and also because their preconditions are not satisfied by the only state in the Kripke model and as such will have no effect.

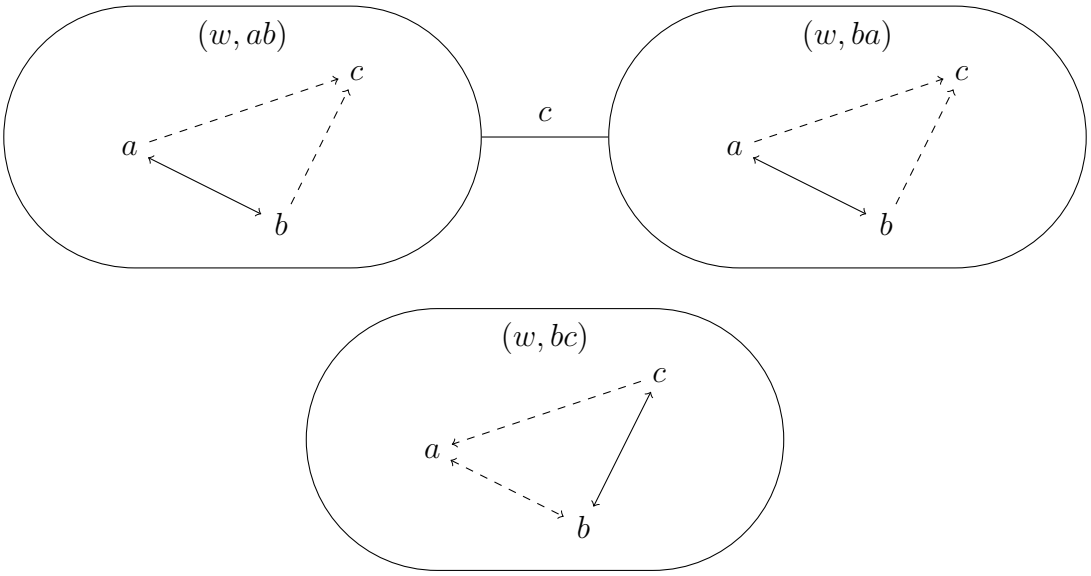


Figure 6: *The Kripke model $\mathcal{G} \times \mathcal{E}_{\mathcal{G}}$.*

that the two states (w, ab) and (w, ba) have the exact same graph representing them[6]; in the future we will not make this distinction. In the implementation a state is represented *just by the propositions that are true at it*; in our case, this is the graph.

2.3 Planning

Automated planning is the process of computing which set of events must occur to take a system from some given initial state to some successful state. Such a system is defined as a triple $\Sigma = (S, A, \gamma)$, where:

- S is some set of states;
- A is some set of *actions*;
- $\gamma : S \times A \hookrightarrow S$ is a state-transition function. It is partial; for $s \in S, a \in A$, either $\gamma(s, a) \in S$ or it is undefined.

Then an instance of the planning problem is a triple (Σ, s_i, S_g), where;

- Σ is a planning system;
- $s_i \in S$ is some initial state;
- $S_g \subseteq S$ is some set of goal states.

A *solution* to the planning problem is some ordered set of actions $a_0, a_1, \ldots, a_n$ such that $\gamma(\gamma(\ldots \gamma(s_0, a_0) \ldots, a_{n-1}), a_n) \in S_g$.

Note that we call a planning problem *multi-agent* if the number of agents in the system is greater than 1, and *single-agent* if the number of agents is equal to 1.

[6]We revisit this idea of gossip graphs with three agents being "boring" again later. This "boring"-ness comes from the fact that calls ab and ba are essentially the same thing.

2.3.1 Epistemic Planning

The dynamic epistemic logic community has recently been investigating a special case of planning, namely *epistemic* planning (Bolander and Birkegaard 2011, Aucher and Bolander 2013). This is planning where we may have some initial epistemic state (e.g. agent a knows ϕ), some actions that update these epistemic states, and some accepting epistemic state (e.g. agent b knows ψ). It is not hard to see how this can be applied to the gossip problem.

We give a formal definition of the epistemic planning problem that slightly differs from convention, however with this definition and those from the literature are equivalent. Then the epistemic planning problem is as follows; given a pointed epistemic model $(\mathcal{M}, w)$ and an event model $\mathcal{E}$, and some goal formula ϕ, find some set of events $e_1, e_2, \ldots, e_n$ such that $(\mathcal{M}, w) \otimes (\mathcal{E}, e_1) \otimes \ldots \otimes (\mathcal{E}, e_n) \models \phi$. The *propositional epistemic planning problem* is the restriction of this to event models with pre- and post-condition functions whose codomains are in the propositional fragment of $\mathcal{L}$; that is, they do not contain the modal operator K. These event models are what we call propositional event models.

It is proven in Bolander and Birkegaard 2011 that the multi-agent epistemic planning problem is undecidable for an aribtrary system. However, placing certain restrictions on the system used yields decidability (Yu, Wen, and Liu 2013, Aucher, Maubert, and Pinchinat 2014). One particularly important result is that the propositional epistemic planning problem is decidable; indeed, it is this result that this thesis relies on.

In this thesis, we focus on solving the epistemic planning problem by means of the automata constructions described in Aucher, Maubert, and Pinchinat 2014. These methods, and the resulting piece of software associated with this thesis, are unique in that they will work for any epistemic model and (propositional) event model. This unfortunately means that we cannot plan for situations like muddy children or drinking logicians (van Eijck 2014a) as they require epistemic pre- or post-conditions.

2.4 Existing Tools

We now give an overview of existing software in the realm of model checking or planning for epistemic systems.

2.4.1 DEMO-S5

DEMO-S5 (van Eijck 2014a) is a model checker for epistemic models, limited to $S5$ models. It is also limited in that it only supports events that are either public announcements (e.g. an agent telling every other agent some statement) or publicly observable factual change (e.g. the flip of a coin, where the result is seen by every agent).

It performs the task of model checking by taking an epistemic model and a formula representing either the announcement or the fact that changed, and updating the model with the event represented by the formula. The definition of the update used is very similar to the update of an epistemic model with an event model as defined in Section 2.1.2. The given events are applied, and at this point we check if we're in a successful state or not.

Unfortunately we cannot use this to model check a gossip problem, as events in the gossip problem are neither of these; they are private one-to-one communications. Furthermore, the software is incapable of planning; simply checking. Despite this, the software was incredibly useful in our project in order to aid understanding of epistemic and event models. Indeed, the formalisation of epistemic and event models demonstrated here was used in the code associated with this thesis.

2.4.2 Gossip

Gattinger 2018a is another model checker, however now for strictly the gossip problem. This program has a significant advantage over the other two pieces of software; namely that it is capable of planning. It does this by generating all of the possible sequences of calls on the graph and then checking which of these are successful (i.e., after execution of all of the calls the graph is in some successful state) and which are not. It also has further capabilities for studying protocols deeply, however these are not relevant to our thesis.

Although the methods used within this software are very different to the ones used in our implementation, this system proved to be very useful for testing purposes, due to its simplicity of use. The results returned by our software during testing were verified using this system. Further elaboration on this will be given in the evaluation section of this thesis.

One limitation of this system is that is may only handle gossip states, and not generic epistemic models. Also limiting is that it enumerates all possible call sequences from the initial state; we are only interested in successful ones. This means that we have the possibility of lots of wasted computation if we need to validate lots of unsuccessful paths before we get to a successful one.

2.4.3 SMCDEL

SMCDEL is the most sophisticated of the three pieces of software. A technical report is given in Gattinger 2018c, whilst a lot of the underlying theory is in Gattinger 2018b. The main difference is that it uses *symbolic model checking*, a method put forward in Burch et al. 1992. This gives a much more efficient implementation **why**?, as can be seen in the Chapter 4 of Gattinger 2018b.

This software is capable of efficiently model checking all classes of epistemic models, and also has capacity for planning; however this side of the program is not well-displayed, and it is unclear precisely how to use it. Furthermore, due to a reasonably unpleasant interface we choose not to use this software for benchmarking and instead use the simpler Gattinger 2018a.

2.5 Uniform Strategies

In this thesis, we use a lot of ideas from Aucher, Maubert, and Pinchinat 2014. This paper puts forward a method for epistemic planning, through use of automata representing states of the model and events in the event model as the states and transitions respectively. This then references a process in Maubert and Pinchinat 2014 for the creation of an automata which has a set of the other indistinguishable states baked into the states of the automata. This is a very useful thing for our context, as it means that we can evaluate formulas of the form $K_a\phi$ *positionally*; that is, we need only to look at the state to be able to evaluate the formula. This is

in contrast to the alternative, which is to check our current state then compute the possible call sequences that are indistinguishable from our current one, and then go and check there too, and so on.

The process in Maubert and Pinchinat 2014 is very verbose and is beyond the scope of this thesis; however, in order to illustrate the fact that our algorithm is a special case of the process, we give a brief introduction to the process and its theory. We slightly change some of the definitions to lend itself better to our scenario. The setting is game theory; hence the frequent references to plays and strategies.

2.5.1 Game Arenas

A game with *imperfect information* is a game in which the players have some uncertainty concerning hte current configuration of the game (for example Poker; a player does not know which cards are in the other players' hands). A player in such a game cannot plan to play differently in a situation she cannot distinguish; hence we have to define her strategy *uniformly* across situations she cannot distinguish.

We consider two-player turn-based games played on a some graph, where the vertices are labelled with propositions. These propositions hold some useful information about the state; it could be what people know about it, or some fact about the environment the player is in and so on. This set of propositions will be referred to as AP.

Then a game arena is a structure $\mathcal{G} = (V, E, v_I, l)$ where $V = V_1 \uplus V_2$ [7] is a set of positions split between those of Player 1 (V_1) and those of Player 2 (V_2). $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$ is the set of edges, $v_I \in V$ is the initial position and $l : V \to \mathcal{P}(AP)$ is a valuation function, which maps each position to the finite set of propositions that are true at it.

2.5.2 Transducers

We give a quick recap of transducers, as they are used heavily in the next two sections. A transducer is like a nondeterministic finite automaton with two tapes; an *input* tape and an *output* tape. The transducer reads an input finite wood on its input and writes a finite word on its output tape. Hence a transducer defines a binary relation. Relations recognised by a transducer are called *regular* or *rational* relations.

A transducer is a 6-tuple $T = (Q, \Sigma, \Gamma, I, F, \Delta)$, where Q is a finite set of states, Σ is the input alphabet, Γ is the output alphabet, $I \subseteq Q$ is a finite set of initial states, $F \subseteq Q$ is a finite set of final states, and $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the transition function.

$(q, a, b, q') \in \Delta$ means that the transducer can move from state q to state q' by reading a from the input tape and writing b to the output tape. We will use this notation throughout, however this is equivalent to saying that $\Delta(q, a) = (b, q')$. Note that transducers are, in this thesis, nondeterministic; hence if $\Delta = \{(q, e, f, p), (q, e, g, r)\}$ then $\Delta(q, e) = \{(f, p), (g, r)\}$.

Then we can define the extended transition relation Δ^* as the smallest relation satisfying:

- for all $q \in Q$, $(q, \epsilon, \epsilon, q) \in \Delta^*$
- if $(q, w, w', q') \in \Delta^*$ and $(q', a, b, q'') \in \Delta$, then $(q, w.a, w'.b, q'') \in \Delta^*$

[7] We use $\uplus$ here to indicate that this is a partition

We can see Δ^* as the transitive closure of Δ. We abbreviate $(q, w, w', q') \in \Delta^*$ to $q \rightsquigarrow_{w|w'} q'$ in the rest of this section.

Finally, we say that the relation recognised by some transducer T is

$$[T] ::= \left\{(w, w') \mid w \in \Sigma^*, w' \in \Gamma^*, \exists q_F \in F, \exists q_i \in I, q_i \rightsquigarrow_{w|w'} q_F\right\}$$

2.5.3 The Powerset Arena

In games with imperfect information, the information set of a player after a certain play is the set of positions she may possibly be in, consistent with what she has observed[8]. We can define a similar notion in our setting, and we show that this is sufficient to build a powerset arena in which formulas of the form $K_a\phi$, where ϕ is propositional, can be evaluated positionally.

For an arena $\mathcal{G}$ and a transducer T over finite sequences of events $e_i \in E$, we construct this automata $\widehat{\mathcal{G}}$ in which we can evaluate formulas of the form $K_i\phi$...

Our new information set after a move occurs cannot be computed knowing only the previous information set and the new position; we need to simulate the nondeterministic execution of T, taking as input the sequence of positions played and writing as output the related plays. Precisely, we need two things; the set of states the transducer may be in after reading the sequence of positions played so far, and for each of these states the set of possible last positions written on the output tape. We need only remember the last letter and not the whole tape because the information set we aim to compute is just the set of the last positions of related plays.

Then our positions are of the form $(v, S, Last)$, where $v \in V$, $S \subseteq Q$, where Q is the set of states in the transducer and S is the set of possible current states of T, and $Last : S -> \mathcal{P}(V)$[9] associates to a state $q \in S$ the set of the possible last positions on the output tape fo T if the current state is q. The transitions in this arena follow those in $\mathcal{G}$, except we now maintain the additional information about the configuration of the transducer.

To construct the initial position $\widehat{v_I} = (v_I, S_I, Last_I) \in \widehat{\mathcal{G}}$, we need to simulate the execution of T starting from its initial state and reading v_I. To do this, we introduce an artificial position $\widehat{v_{-1}} = (v_{-1}, S_{-1}, Last_{-1})$, where $v_{-1} \notin V$ is some fresh position, $S_{-1} = \{q_i\}$ where q_i is the initial state of the transducer because before starting the transducer is in its initial state, and $Last_{-1}(q_i) = \emptyset$ because we have nothing written on the output tape.

We now come onto defining $\mathcal{G}$. Let $\mathcal{G} = (V, E, v_I, l)$ be an arena and $T = (Q, V, q_i, Q_F, \Delta)$ be an FST. Then we define the arena $\widehat{\mathcal{G}} = (\widehat{V}, \widehat{E}, \widehat{v_I}, \widehat{l})$ as:

- $\widehat{V} = V \times \mathcal{P}(Q) \times (Q \to \mathcal{P}(V))$
- $(u, S, Last)\widehat{\rightarrow}(v, S', Last')$ if
  - $u = v_{-1}$ and $v = v_I$, or $u \to v$,
  - $S' = \left\{q' \mid \exists q \in S, \exists \lambda' \in V^*, q \rightsquigarrow_{v|\lambda'} q'\right\}$ and

[8] For instance in a game of poker with one deck of cards, a player may have no idea what the other players have in their hands except that they know they do not have the cards the player has in their own hands, nor the cards face-up on the table.

[9] It may be more helpful to think of this as something a bit more tangible than a function, e.g. a list of tuples $(S, [P])$

  - $Last'(q') = \{v' \mid \exists q \in S, \exists \lambda' \in V^*, q \rightsquigarrow_{v|\lambda' \circ v'} q',$ or $q \rightsquigarrow_{v|\epsilon} q'$ and $v' \in Last(q)\}$
- $\widehat{v_I}$ is the only $\widehat{v} \in \widehat{V}$ such that $\widehat{v_{-1}} \widehat{\rightarrow} \widehat{v}$.
- $\widehat{l}(\widehat{v}) = l(v)$ if $\widehat{v} = (v, S, Last)$.

The definition of transitons is quite complicated, and as such we give a high-level explanation of it. Regarding the definition of the transitions, the first point means that our transitions in $\widehat{E}$ follow those in E, except for the transition leaving $\widehat{v_{-1}}$, which we use to define $\widehat{v_I}$.

The second point expresses that when we move from u to v in $\mathcal{G}$, we give v as input to the transducer. Then the set of states that we can be in, that is, S', is the set of states that can be reached from some previous possible state in $q \in S$ by reading v on the input tape and outputting some sequence λ'.

Finally, the third point expresses that if some position v' is at the end of the output tape after the transducer reads v and reaches q, then this is either because while reading v the last letter it wrote is v', or it wrote nothing and v' was already at the end of the output tape before reading v.

Finally, we say that $\widehat{v_I}$ is the only successor of $\widehat{v_{-1}}$, and that the valuation of a position in the powerset arena is the valuation of the underlying position in the original arena.

2.5.4 Lifting Transducers

We can keep on repeating this process, creating a power-powerset arena $\widehat{\widehat{G}}$. However we first need to lift our transducer T where $[T] \subseteq Plays_* \times Plays_*$ to a transducer $\widehat{T}$ where $[\widehat{T}] \subseteq \widehat{Plays_*} \times \widehat{Plays_*}$.

We write $T\downarrow$ for the transducer that computers the bijective function f that maps a play $\widehat{p} \in \widehat{Plays_*}$ to the underlying play $p \in Player_*$, and $T\uparrow$ for the deterministic transducer that computes f^{-1}. Both can be easily constructed from our powerset arena $\widehat{G}$.

Then the lift of a transducer T is $\widehat{T} = T\downarrow \circ T \circ T\uparrow$.

Once we have the transducer $\widehat{T}$, we can repeat the process in Section **??** with $\widehat{G}$ to get $\widehat{\widehat{G}}$, and so on ad infinitum. $\widehat{\widehat{G}}$ would allow us to positionally check formulae of the form $K_i K_j \phi$.

3 Algorithm

3.1 Introduction

In this section, we formalise the algorithm that we use to solve the propostional epistemic planning problem. We see this as a meaningful contribution of the thesis; there is no existing, clear implementation of this in the literature.

We first give a rough explanation of the process, before getting into the details of the process later in this chapter.

As input, we receive an epistemic model $\mathcal{M}$, an event model $\mathcal{E}$, and some formula $\phi \in \mathcal{L}(\Lambda)$, for some set of propositions Λ. First, we construct an automata called $\mathcal{ME}^*$, by simulating the repeated application of $\mathcal{E}$ on $\mathcal{M}$. This lets us easily traverse the space of possible states we could be in, after applying the events in $\mathcal{E}$ to the model $\mathcal{M}$.

For the next step, we need to consider the shape of the formula ϕ. If it does not include a knowledge modality K_i then we proceed to the next stage. If it does, then we need to perform the process detailed in Section 3.3, where the transducer $\mathcal{T}_i$ used is the transducer relating the events that are indistinguishable to agent i. This is a powerful procedure that brings the information regarding the states that are indistinguishable to the agent i into the state we are currently in. Then, in order to check if a formula of the form $K_i\phi$ is true, we just need to check that ϕ holds at all of the states that are indistinguishable from our current one. We can now do this in a very straightforward manner, given that the information regarding the states we could be in is now baked-into the state.

We repeat the process in Section 3.3 each time we encounter a K modality [10]; in cases where we have a conjunction or negation where one of the formulas included is a knowledge-based formula, then we perform intersection and complement of the resulting automata. This is a very pleasing result and is a great display of why finite-state automata are so well suited to this area.

Finally, once we have finished building automata we perform breadth-first search on the resulting machine. At this point, we have an automata in which formulae of the same shape [11] as ϕ can be evaluated positionally. Then we set the states which satisfy ϕ to be the successful states. We can now perform breadth-first search on the resulting automata to find the shortest path to a successful state.

With this all in mind, we now move into the implementation details of the system.

3.2 $\mathcal{ME}^*$

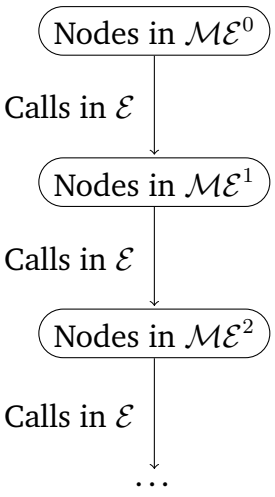
Put forward in Aucher, Maubert, and Pinchinat 2014, $\mathcal{ME}^*$ is a structure we use to represent *any* epistemic model and event model pair. We slightly adjust the definition given in the literature to a version much more suited to our situation.

[10] Note that in a situation where we have a formula $K_aK_a\phi$ we do not need to build another powerset automata as in an S5 model, $KK \dots KK = K$ (Garson 2018). However, for a formula $K_aK_b\phi$ we do; this is because we can see K_i and K_j as being different modalities for different agents i and j.

[11] By shape, we mean that the propositions in the formula may be anything in $\mathcal{L}_P(\Lambda)$, as long as the knowledge modalities, conjunctions and negations remain in place

It represents the application of some event model $\mathcal{E}$ onto an epistemic model $\mathcal{M}$ repeatedly. Consider that $\mathcal{ME}^0$ represents $\mathcal{M}$, $\mathcal{ME}^1$ would be $\mathcal{M} \times \mathcal{E}$, and $\mathcal{ME}^2$ represents $\mathcal{M} \times \mathcal{E} \times \mathcal{E}$, and so on and so forth. Then we define $\mathcal{ME}^*$ as

We can picture $\mathcal{ME}^*$ as having layers, where each node in a layer $\mathcal{ME}^n$ has taken exactly n events to arrive there.



$\mathcal{ME}^*$ consists of an automaton $A_{\mathcal{D}}$ and a set of transducers $\{\mathcal{T}_i \mid i \in A\}$. $A_{\mathcal{D}}$ represents the states in our epistemic model $\mathcal{M}$ and the transitions between these states due to the events in $\mathcal{E}$. The * of $\mathcal{ME}^*$ refers to the fact that the event model has been applied to the original model maximally; it contains $\mathcal{M} \cup (\mathcal{M} \times \mathcal{E}) \cup (\mathcal{M} \times \mathcal{E} \times \mathcal{E}) \cup \ldots$. $\mathcal{T}_i$ represents the events that are indistinguishable to agent i.

The roles of these structures will become more clear once we see how they are constructed.

Let $\mathcal{M} = (W, R, V)$ be an epistemic model and let $\mathcal{E} = (E, Q, \mathsf{pre}, \mathsf{post})$ be an event model.

Then define the automaton $A_{\mathcal{D}} = (\Sigma, Q, \delta, q_i, F)$, where $\Sigma = W \cup E, F = \{q_v \mid v \subseteq \Lambda\}$ and $Q = F \uplus q_i$. We define δ, the transition function, as follows:

$$\forall w \in W, \forall e \in E,$$

$$\delta(q_0, w) = q_{v(w)} \qquad \delta(q_0, e) = \bot$$

$$\delta(q_v, w) = \bot \qquad \delta(q_v, e) = \begin{cases} q_{v'}, \text{where } v' = \{p \mid v \models \mathsf{post}(e, p)\} & \text{if } v \models \mathsf{pre}(e) \\ \bot & \text{otherwise.} \end{cases}$$

In order to construct the set of transducers $\mathcal{T}$, consider the one state transducer $\mathcal{T}_i = (\Sigma, Q', \Delta_i, q, F')$ where $Q' = \{q\}$, $F' = \{q\}$ and $\Delta_i = \{(q, w, w', q \mid (w, w') \in R)\} \cup \{(q, e, e', q \mid (e, e') \in Q)\}$
TODO: Consider moving this to the earlier section?.

3.2.1 $\mathcal{ME}^*$ in context

We adapt the definition slightly for our context. In our context, when we are starting to move through the automata we do not need to be able to dynamically pick the world; rather, we fix it in the code as the set of initial states of the epistemic model. Hence we l

3.3 Powerset Adapted

The powerset construction given in **??** is a very broad definition, designed for use in game arenas. We need a simpler version of this for our context, as finite automata are simpler than game arenas and as such we do not need as much information. We first show how the special transducer in this section is built. We will then define our version of the powerset construction, and then explain where the differences arise. This process puts the powerset construction in **??** into context.

3.3.1 Transducer Composition

Essential to this stage of the algorithm is the construction of a transducer to relate a state with the states considered possible after some event happens by some agent. **add example**.

This requires two components, the first of which is the transducer defined in **??**, which relates pairs of events that are indistinguishable to an agent. This can be constructed with ease from the event model, as discussed earlier.

The second component is the identity transducer over the automata $\mathcal{ME}^*$. This can be computed in another very straightforward manner. Given $\mathcal{ME}^* = (\Sigma, Q, \delta, q_0, F)$, we can construct the identity transducer $\mathcal{T}_{\text{id}} = (\Sigma, Q, \Delta, q_0, F)$ where $(q, e, e', q') \in \Delta$ iff $(q, e, q') \in \delta$ and $e = e'$.

The next step is to compose the two. We use a slightly different method than those put forward in the literature (Argueta and Chiang 2018), given that one of our transducers has only one state; hence, it would not make sense to need to move on from the state in the first transducer, as is classic. For a single-state transducer $\mathcal{T}_1 = (\Sigma, \{q\}, \Delta_1, q, \{q\})$ and a transducer $\mathcal{T}_2 = (\Sigma, Q, \Delta_2, q_0, F)$, we define the composition $\mathcal{T} = (\Sigma, Q, \Delta, q_0, F)$ where

$$\Delta(s, e) = \{(e', s') \mid (e', q) \in \Delta_1(q, e), (e', s') \in \Delta_2(s, e')\}$$

Hence, if we give as input to Δ an input pair (q, e), where q is a state and e an event, it returns a set of pairs $\{(e', q')\}$, where e' is some event indistinguishable from e and q' is the state we reach through the occurence of e' at state q. This transducer is the one used in the definition of the transition relation in Section 3.3.2. This makes the transitions in the Powerset automata much more simple to implement.

3.3.2 Powerset

Given an automata $\mathcal{ME}^* = (\Sigma, Q, \delta, q_i, F)$ and a transducer $\mathcal{T}_i = (\Sigma, Q, \Delta, q_0, F_{\mathcal{T}})$, we construct a powerset automata $\widehat{\mathcal{ME}^*} = (\widehat{\Sigma}, \widehat{Q}, \widehat{\delta}, \widehat{q}_i, \widehat{F})$. In this automata, our states $\widehat{q} \in \widehat{Q}$ are of the form (v, S), where $S \subseteq Q$. This is because, in our case, when we need to evaluate a formula of the

form $K_i\phi$, we just need to know the set of states indistinguishable to agent i at this moment. This is exactly what the set S is.

- $\widehat{Q} = Q \times \mathcal{P}(Q)$
- $\widehat{\Sigma} = \Sigma$
- $\widehat{\delta}((v, S), \sigma) = (v', S')$ where
  - $v' = \delta(v, \sigma)$
  - $S = \{q' \mid \exists q \in S, \exists \sigma' \in \Sigma, (\sigma', q') \in \Delta(q, \sigma)\}$
- $\widehat{q_0}$ is the state (q_0, S) where S is the set of all worlds indistinguishable from q_0 in $\mathcal{ME}^*$.
- $\widehat{F} = \{(v, S) \mid v \in F\}$

We see that this version is much simpler than the definition given in Section **??**. This is because we don't need to keep track of the set of potential final states of the transducer's output tape; we only have to keep track of the other possible states of the transducer. The set of states in the transducer is exactly the same as the set of states in our $\mathcal{ME}^*$ automata, so these transducer states correspond directly to the states that we *could* be in in our $\mathcal{ME}^*$ automata.

This is easy to see from the definition of transitions in $\widehat{\mathcal{ME}^*}$. S' becomes the set of all states q' that can be reached by making a transducer transition with the input event σ from some state $q \in S$. If we recall the definition of our transducer from Section 3.3.1, a transition $\Delta(q, \sigma)$ gives us a set of pairs σ', q' where σ' is some event indistinguishable to agent i, and q' is the state we reach from making that transition from state q.

We now give an example execution of this process. Consider the following gossip graph;

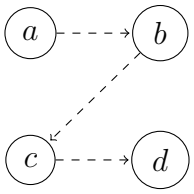


Figure 7: *An example gossip graph*

Let us take this gossip graph to be our example. We want to get to a state where the formula $K_a(Sbc)$ is true; that is, we want to get to a state where agent a knows that agent b knows the secret of agent c. One call sequence that will guarantee this is $bc; ab$; b calls c to discover c's secret, and then a calls b. At this point a realises that b knows the secret of c, and hence the formula $K_a(Sbc)$ is satisfied.

This is of course a very high-level description, and is (unfortunately) not hwo the program will reason about it. Let us step through one application of the event model $\mathcal{E}$ on this graph. In this case, our event model $\mathcal{E}$ is the set of all possible calls; however only three are permitted. This is displayed in Figure 8

The above diagram shows the three possible states that we could be in afer making one call from the initial state displayed in 7. The red line indicates that agent a cannot distinguish between these two states, as they were both reached by the execution of a call that agent a was

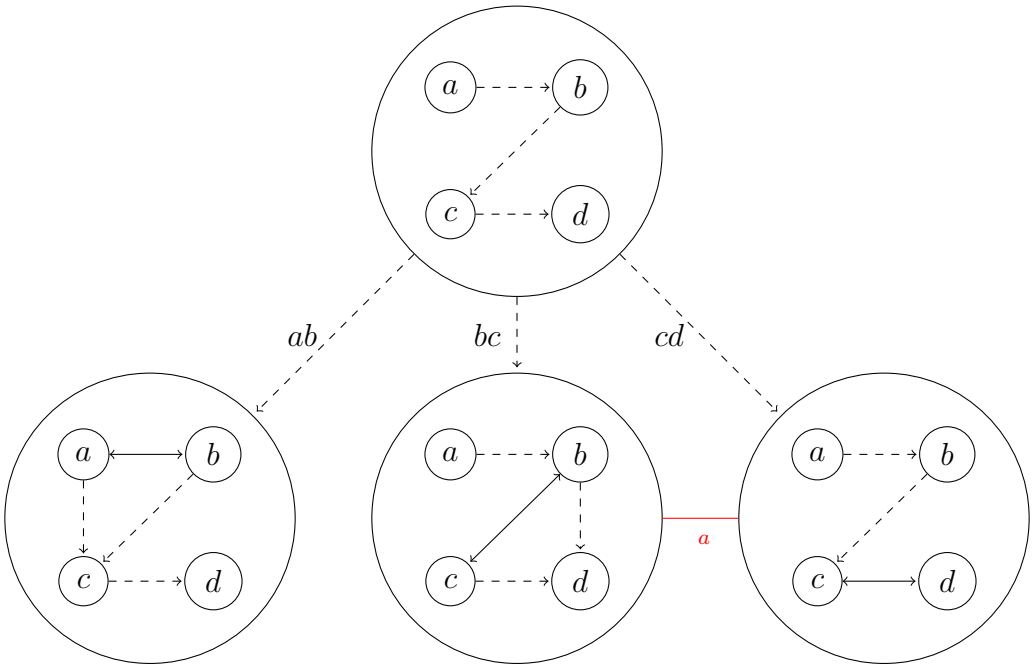


Figure 8: *The gossip graph in 7, updated with $\mathcal{E}$ once.*

not a member of. Note that in the algorithm this connection does not exist; it's here simply for labelling. We can see the three states on the lower row as a Kripke model, where the three are worlds labelled by their valuation, and the two connected by the red line are indistinguishable to a.

Now say that we wanted to evaluate the formula $\neg S_{ba}$ on the bottom-left world in Figure 8. This is very straightforward; we just need to check if b does indeed know a's secret; here it doesn't, and so the world satisfies $\neg S_{ba}$. But what if we want to evaluate the formula $K_a \neg S_{ba}$? We need to visit each of the states indistinguishable from our state and see if $\neg S_{ba}$ holds there. This is currently impossible; we have no idea whether agent a cannot distinguish between this world and any others, and certainly no way of reaching those worlds if they are. This is where the powerset construction comes into play; we pull the indistinguishable states into the states themselves. Hence, Figure 8 becomes Figure 9.

The difference between Figures 8 and 9 now is that the former has all of the states indistinguishable to itself inside the state. This means that we can evaluate the formula $K_a \neg S_{ba}$ at a state by simply inspecting the states indistinguishable from the one we're looking at to check a formula of such form. This is now an extremely straightforward task!

An interesting result that arose during implementation and testing of this algorithm was that the "actual" state - that is, the v of the (v, S) pairs that make up states in the powerset automata - have absolutely no bearing on the progression of the system. I had originally expected that the actual state would be treated as a higher-class item than those it was indistinguishable from, however this was not the case. I suppose this lives up to the nature of the indistinguishability relation; these states are truly all equal.

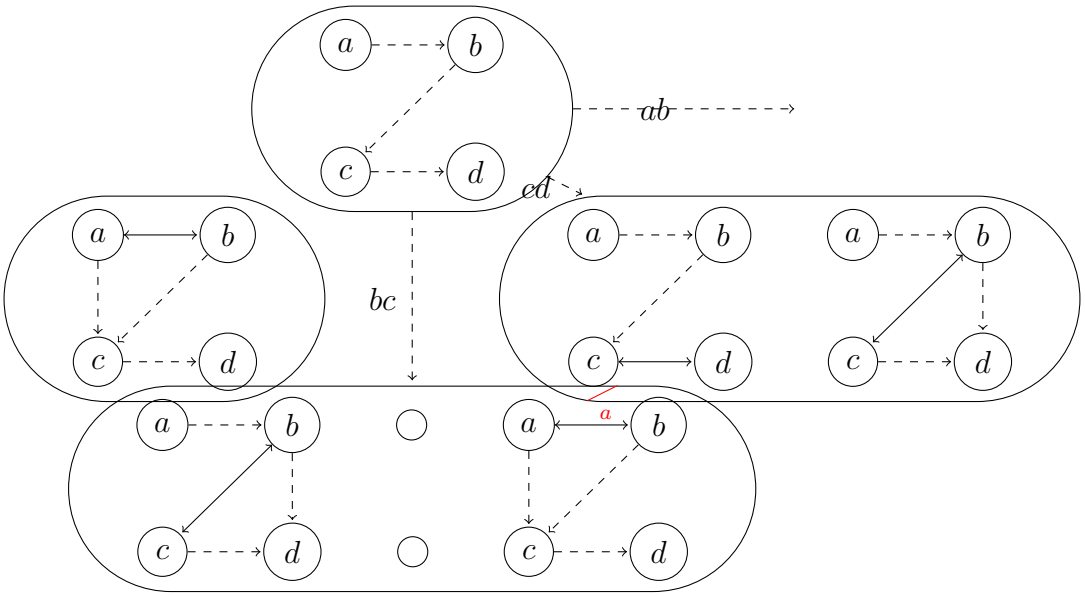


Figure 9: *The automata in Figure 8, now with the indistinguishable states in the states.*

3.4 Combining Automata

When we encounter a formula of the form $\phi \wedge \psi$, where $\phi, \psi \in \mathcal{L}_P(\Lambda)$, we may simply construct an automata to solve this formula. However this is less straightforward when we encounter a formula of the formula $K_a\phi \wedge K_b\psi$ [12]; in this case, we want to produce powerset automata for $K_a\phi$ and $K_b\phi$ and then join them back up somehow.

Luckily, intersection for finite automata is well-defined. We give the definition here. For two automata $A = (\Sigma, Q_A, \delta_A, q_{0A}, F_A)$, $B = (\Sigma, Q_B, \delta_B, q_{0B}, F_B)$, we define their intersection $A \cap B = (\Sigma, Q, \delta, q_0, F)$ where $Q = Q_A \times Q_B$, $F = F_A \times F_B$, $q_0 = (q_{0A}, q_{0B})$ and $\delta(q_A, q_B) = (\delta_A(q_A), \delta_B(q_B))$.

This definition expands in the simple way when we want to take the conjunction of n automata.

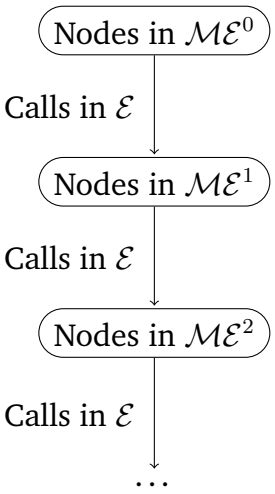
We have a similar story for negation; given a formula $\neg K_a\phi$, to create an automata to solve this we build the automata to solve $K_a\phi$ and then take the complement of this. This consists of simply changing the set of accepting states. Given automata $A = (\Sigma, Q, \delta, q_0, F)$, we define $A_\neg = (\Sigma, Q, \delta, q_0, F_\neg)$ where $F_\neg = Q \setminus F$.

To complete our little family of set operations, we should define disjunction too. Taking the union of automata is, algorithmically, less straightforward than the intersection; hence we just exploit de Morgan's law and use the identity $A \cup B = (A^C \cap B^C)^C$. We recognise that when we come to implementation this may be more costly than some alternative implementation, but for now this is of little importance.

[12] This occurs very frequently; in this gossip problem, we often want to reach a state where every agent knows that everyone is an expert.

3.5 Searching

Then given our solving automata, we want to traverse it to find paths that takes us from the initial state to some successful state. Recall the structure of the graph $\mathcal{ME}^*$ from Section 3.2;



After any kind of powerset, intersection or complement operation, it will still follow the same shape; we have layers, and as they get deeper more events get applied to them. One big benefit of this topology is that the resultant graph is *acyclic*; there is no way we can return to an earlier state.

This is one of the reasons that a breadth-first traversal of the resultant graph is most suitable for this context.

4 Implementation

We now come to covering the implementation of the algorithm developed in the previous chapter. We will first give an overview of the structure of the software and then a summary of all design decisions taken throughout.

For the implementation of our algorithm, we chose to use Haskell. Haskell is a lazy functional programming language, which gives us several benefits for this particular task that other languages lack.

The first one is the syntax; its functional style lends itself very well to mathematical definitions. Its list comprehension and pattern matching let us write code that very closely mimics the original notation, thus erasing a lot of the difficulty of converting a mathematical definition to code once we have implemented it.

Another benefit is its laziness. In our implementation we often use automata with a ridiculously large state space; however (thankfully) Haskell's laziness means that we never have to enumerate these states. We only need to access them when we need them. This saves a lot of processing time, as well as greatly reduces the space the program takes up.

Its strong type system and typeclasses support us and give us safety guarantees, preventing us from writing erroneous code that in another language may only be detected at runtime. A simple example is that in our implementation we cannot represent a formula that is not well-formed; a more sophisticated one is the ability to make sure that an epistemic model and an event model span the same set of atomic propositions and events. Typeclasses help us make code that is very reusable but also safe; if we wanted to use a new language of atomic propositions for an existing model, we would only need to give an instance of the `Prop` typeclass for our new language and hey presto.

The classification of functions as first-class objects also comes in very helpful. In the implementation we make the transition function of an FSM a function $\delta : Q \times \Sigma \to Q$. Often we want to "lift" the states Q to some other data type, e.g. $P(Q)$. We can very easily unwrap the datatype P to access the underlying Q and then use the previous δ function; this would be much more tricky in a language other than Haskell, however in Haskell it is a very pleasant thing to do.

Haskell offers a simple way to divide each file into a module and import (or not) a module into another. We give a brief overview of each module in the system, and highlight notable uses of any of the above language features in this process. We also highlight any notable design choices made.

4.1 `Model.hs`

This section is arguably the foundation of the system. Here we implement the structures defined in Section 2.1; we have the language of our formulae, Kripke models, Event models and updates thereof.

In Figure 10, we can see the implementation of both Kripke models and event models in Haskell. `states` and `events` respectively represent the states and events of the Kripke and Event models, and `eprel` and `evrel` respectively represent the indistinguishability relations between them. Note that for ease of implementation we use *partitions* (van Eijck 2014b) of

sets rather than sets of pairs. This is done because we regularly want to just access the set of states we cannot distinguish an item between, and the use of partitions makes this a much simpler task.

```
data EpistM st prp = Mo {
  states :: [st],
  agents :: [Agent],
  val :: [(st, [Form prp])],
  eprel :: [(Agent, [[st]])],
  actual :: [st],
  allProps :: [prp]
}
```

(a) *The Kripke model datatype*

```
data EventModel ev prp = EvMo {
  events :: [ev],
  evrel :: [(Agent, [[st]])],
  pre :: ev -> Form prp,
  post :: (ev, prp) -> Form prp
}
```

(b) *The Event model datatype*

Figure 10

The user then defines their own pre and post -condition functions. One deviation from the mathematical notation is the addition of the set of agents; this is purely a pragmatic thing, as it makes certain operations more straightforward later on.

```
update :: (Eq ev, Prop p) => EpistM (State ev) p -> EventModel ev p -> EpistM (State
update epm evm = Mo states' (agents epm) val' rels' (actual epm) (allProps epm)
  where
    states' = [stateUpdate s ev | s <- states epm, ev <- events evm, satisfies (epm,
    rels' = [(ag, newRel ag) | ag <- agents epm]
    newRel agent = filterRel states' [liftA2 stateUpdate ss es |
                                      ss <- fromMaybe [] (lookup agent $ eprel epm),
                                      es <- fromMaybe [] (lookup agent $ evrel evm)]
    val' = [(s, ps s) | s <- states']
    ps s = [P p | p <- props, satisfies (epm, trimLast s) (post evm (lastEv s, p))]
    props = allProps epm
```

Figure 11: *The $\mathcal{M} \times \mathcal{E}$ function*

Now we see the definition for the update of a Kripke model with an event model. Whilst the agents, set of propositions and actual world remain the same, we update the states, epistemic relations and valuation function in a way very similar to the definition in Section **??**. This is a display of the merits of Haskell; our implementation stays very close to the specification, allowing for simple visual verification that the code that we've written matches the specification.

4.2 `ME.hs`

We now come onto the module that handles construction of the automata $\mathcal{ME}^*$. This is a slightly more interesting case.

In this module we make use of one of the many language extensions Haskell offers us, namely `MultiParamTypeClasses`.

```
class (Ord st, Prop p) => EvalState st p where
evalState :: Form p -> st -> Bool
```

This lets us ensure in our functions that the propositions we're using can be evaluated at the states we want them to be evaluated at. This prevents us from trying to evaluate formula of a certain language on a world it is incompatible with.

4.3 `FSM.hs` and `FST.hs`

These two modules hold our finite state machines. Given how much of the algorithm relies on state machines, it was very important in development to have a reliable structure for state machines. Furthermore, we needed to do some non-standard operations on the machines we implemented (namely, the composition mentioned in Section 3.3.1). To do this easily we need low-level access to the states and a good understanding of the library.

The two libraries that seemed most suitable were Camilleri 2015 and Kmett 2018. However, in both libraries, there seemed to be far too much to take in; it felt like the time spent trying to understand how to use the library (particularly in the latter) would far outweigh any potential benefit gained from using them. Furthermore it was particularly unclear how we would go about implementing our single-state composition as mentioned above.

Fortunately, implementation of these finite-state machines is incredibly straightforward. We can see in Figure 12 that the datatypes nearly identically mimic the tuple definitions in Section 2.5.2.

```
data FSM ch st = FSM {
  alphabet :: [ch],
  states :: [st],
  transition :: (st, ch) -> Maybe st,
  initial :: [st],
  accepting :: st -> Bool
 }
```

(a) *The FSM datatype*

```
data FST ch st = FST {
  alphabet :: [ch],
  states :: [st],
  bitransition :: (st, ch) -> [(ch, st)],
  initial :: [st],
  accepting :: st -> Bool
}
```

(b) *The FST datatype*

Figure 12

Our FSM's transition returns a `Maybe st` to encode that it is *nondeterministic*; meaning that it is not guaranteed to return a state on a transition. This comes in useful in practise when we have as input to δ some pair (w, e) where e is not permitted at w. In this case, $\delta(w, e) = \texttt{Nothing}$. An example of this is a call ij in the gossip problem where i does not know j's phone number.

Similarly, the FST transition returns a list of (e, q) pairs. This again encapsulates the nondeterminism of the FST. This is useful in the case of Section 3.3.1, where we want to return from $\Delta(q, e)$ the set of all pairs (e', q') where e' is some event indistinguishable from e and q' is the state reached by making the transition $\delta(q, e')$.

4.4 `Powerset.hs`

This module contains all of the functions pertaining to the powerset construction, as specified in Section 3.3.

As before, we keep definitions very close to the mathematical specification. We introduce a new datatype, `PState`, which holds our states in the powerset. `PVar` is the "atomic" constructor, and it is here that the formulae are actually evaluated. `PCon` represents an actual state and the set of states indistinguishable from this state; such states arise when we want to evaluate a modal formula. Finally the `PList` constructor is used for conjunctions; we pass through all of the states in the list simultaneously.

```
data PState st = PList [PState st]
               | PCon (PState st)
                      [PState st]
               | PVar st
```

Figure 13: *The `PState` datatype.*

The reason we do this is so that we can create our automata recursively. Consider the function in Figure 14[13]. This is the function we use to create the automata we traverse to find successful sequences of events; it pattern matches on the formula given in the first argument to decide the "shape" of the returned automata.

Let us first consider the bottom case. We receive some formula, e.g. $S_{ab} \wedge S_{bc}$. Then we build an automata $\mathcal{ME}^*$ whose successful states are those which satisfy the formula $S_{ab} \wedge S_{bc}$, through the procedure in Section 4.2.

If it receives a formula $K_a\phi$, then we call the `buildPSA` function, which performs the powerset creation process as detailed earlier. Note that it does so with the automata that solves the formula ϕ, invariant of the form of ϕ - it could be an atomic proposition or another modality, or a conjunction of modalities.

```
cSA :: Form p -> EpistM (State ev) p -> EventModel ev p
       -> FSM (Character ev) (PState (QState p))
cSA form@(K agent phi) ep ev
  = buildPSA
             (cSA phi ep ev)
             (buildComposedSS agent ep ev (cSA phi ep ev))

cSA (Not phi) ep ev = complement $ cSA phi ep ev tfilter

cSA (And phis) ep ev = case includesK (And phis) of
  True  -> toPList $ intersection $ map (\phi -> cSA phi ep ev) phis
  False -> makeP $ buildMEStar (And phis) ep ev

cSA phi ep ev = makeP $ buildMEStar phi ep ev
```

Figure 14: *The `createSolvingAutomata` function.*

[13] For clarity and brevity, we omit some of the supporting function calls, and also omit some of the cases.

5 Evaluation

5.1 Plan for Testing

Throughout development we used the Haskell library `HUnit`. This provides a wealth of combinators that we can use to tersely write unit tests for our functions. This was used in a standard manner; before implementing a function we would write tests for it, and then implement the function, ensuring it passes all tests and as such functions correctly.

However, more interesting is our plan for functional testing. This is the part of testing in which we check that the system *correctly functions*, and will tell us if what we have implemented is correct or not.

Although our system is capable of planning for any epistemic model and propositional event model, we chose to test solely on gossip models. This is for several reasons:

- Gossip models cover every part of the system (epistemic models, $\mathcal{ME}^*$, the powerset construction) and as such give us maximal test coverage;
- Testing with a certain class of models tells us that all of the code works; the algorithm does not depend on what the particular model is. Hence if we know it is correct for a certian class of models, it will also be correct for any other class of models;
- We have an existing model checker speciailised for the Gossip problem that we can use (Gattinger 2018a);
- It is the class of models best understood by myself; hence if an error arises it should be quite easy to understand.

This in mind, we chose to use Gattinger 2018a to verify our results. It was chosen over the other two tools (Gattinger 2018c, van Eijck 2014a) due to its ease of accessibility; we can provide it with just a list of lists encoding who knows who's phone number, and it will produce an epistemic model for us. This is very easily produced from the epistemic models in our software. This is a sharp contrast to Gattinger 2018c, which uses a text file based input - output interface which would have added significant complication to the testing process. We could have very easily used van Eijck 2014a, given the similarity betweeen the Kripke models in our code and in van Eijck 2014a, however Gattinger 2018a was eventually chosen due to the conversion between the two representations of a Gossip graph being so straightforward. The last thing we wanted to do was create some kind of bug within the testing software!

5.2 Testing Frameworks

The next decision to be made was how to actually go through with the testing process. Haskell offers the seminal property-based library `QuickCheck`, which, when given a function `a` $\rightarrow$ `Bool` and a way to generate arbitrary values of type `a`, will test the function on these arbitrary values and ensure that the function returns $True$. All that we need to provide in order to generate these arbitrary values is an instance of the `Arbitrary` typeclass. We can see an example of this in Figure 15.

Once we have this, we perform a breadth-first search on the automata generated by this epistemic model and the target formula in order to find either a path that takes us to a successful

```
instance Arbitrary (EpistM StateC GosProp) where
  arbitrary = standardEpistModel agents <$> (sublistOf $ allNumbers agents)
```

Figure 15: *An `Arbitrary` instance for an epistemic model.*

state, or a response telling us that no such path exists. Whatever the outcome, we then send this result and the original model to a function that converts our model to a model in the style of Gattinger 2018a and uses the model checker there to verify our answer.

`QuickCheck` was working well, however I quickly wanted to be able to check for every possible gossip graph. QuickCheck will sometimes repeat instances [14]; hence to get every single possible gossip graph I simply created a list with every single graph in and performed the same process of using the model checker from Gattinger 2018a on each graph.

The creation of gossip graphs is very straightforward; we just produce some set of propositions $N = \{N_{ij}\}$ where $\forall N_{ij} \in N, i \neq j$. We only look for cases where the agents start out not knowing the phone number of any agent apart from each other - we call these *phonebook* graphs. There are a few reasons for this;

- These cases are the only interesting ones. Any gossip graph where some agent already knows the secret of another will either have the same call sequence or a shorter one than the same graph, where no agent knows the secret of any other.
- These cases are the most faithful to the applications of the gossip problem in reality.
- On a pragmatic note, it is more straightforward to convert phonebook graphs to the representation in Gattinger 2018a than non-phonebook graphs.

5.3 Profiling

One of the project aims was to perform a space and time analysis of our implemented system. We have a number of reasons to do this;

- Firstly, it lets us give a quantitative comparison of our program's efficacy compared to the existing tools.
- It also lets us find the parts of our code to try and optimise. It is said that a program spends 90% of its time in 10% of its code; a profiler helps us find this 10%.
- It also lets us find the weaknesses of our program in comparison to other tools, and as such find the weaknesses of the algorithm put forward.

In order to do this we used the profiling tool built into GHC. This gives us a detailed cost-centre breakdown of in which functions the majority of the time is spent, as well as the space used by these functions. We can see an example in Figure 16.

In the following section, we will give an analysis of the time and space used for several sizes of gossip graph and several different winning formulae, and compare them against the

[14] To verify this, simply open up your nearest Haskell REPL, import `QuickCheck`, and run the line of code `verboseCheck ((λ s -> s == s) :: Bool -> Bool)`

```
       Mon Apr 15 14:23 2019 Time and Allocation Profiling Report  (Final)

          Main +RTS -p -RTS

       total time  =        1.46 secs   (1457 ticks @ 1000 us, 1 processor)
       total alloc = 1,036,538,824 bytes  (excludes profiling overheads)

COST CENTRE            MODULE                  %time %alloc

models                 ME                       20.7    3.3
compare                Model                    14.3    0.0
compare                ME                        9.7   22.0
meTrans                ME                        9.2   11.5
enqueue                BFSM                      5.2   10.8
```

Figure 16: *An example cost centre output from GHC.*

corresponding results in Gattinger 2018a. We use Gattinger 2018a as it is the only tool capable of planning.

Note that the way that our tool and Gattinger 2018a plan are quite different. In our tool we return just one successful sequence of events; in Gattinger 2018a all possible sequences are returned. It is unclear how to use the latter tool to return just one; hence in the interests of fairness we just use the function that returns all the possible paths.

5.3.1 Profiling Framework

In our profiling we will generate random graphs as in Section 5.2, use them in both tools and then take averages. This is because certain graphs will take much less time than others [15], and we want to give the most accurate statistics.

We then perform the planning process on the generated gossip graph. As mentioned earlier we run for some amount of graphs and then take the average.

5.3.2 Profiling Results

When profiling, we generated gossip graphs of size 4 (by which we mean graphs with 4 agents in). This is not an arbitrary number; 4 is the smallest "interesting" gossip graph, meaning the smallest graph where a call occuring, not including some arbitrary agent a, has interesting other options; it could be from b to c, or b to d, or c to d [16]. Furthermore, graphs of size greater than 4 take an impractically long time - it is in the interests of time to use graphs of size 4.

[15] Consider a graph where no one knows anyone elses phone number; both tools will very quickly decide that there is no successful path through this. A graph where there is a successful path will take much longer to compute the path, in comparison.

[16] Compare this to a gossip graph of size 3; a call not involving some agent a can only be from either b to c or c to b.

When testing, the variable that we varied is the complexity of target formula. This might seem like an odd choice, but it is well-motivated;

- In Aucher, Maubert, and Pinchinat 2014, it is postulated that the runtime and space complexity of the program is k-exponential, where k is the maximal depth of the nesting of our K modalities. We want to investigate this claim.
- This will give us a handle to compare the methodologies in the two tools, and spot the differences in how knowledge is handled.

We tabulate the results here. We abbreviate the statement "all agents are experts" to the propositional variable E.

	E	K_aE	K_bK_aE
Our tool	0.00917s, 7.36Mb	0.0150s, 10.15Mb	0.0239s, 14.12Mb
Other tool	0.1449s, 251Mb	0.724s, 1.25Gb	3.849s, 6.425Gb

Table 1: *Profiling Results between our tool and Gattinger 2018a*

We can clearly see that our tool is much more time and space efficient than the existing tool. However this is a quite straightforward victory; the other tool enumerates *all* of the possible paths through a gossip graph (for a graph of size 4, there thousands of these), and then checks the success of each one. Compare this to our tool, which simply find the first, shortest, successful path it can take to reach some successful state.

Hence to get a more accurate comparison, we slightly modify the code of the existing tool to exit once it finds a single successful path. This means that it closer mimics our tool, and as such will let us get a better performance comparison between the two.

	E	K_aE	K_bK_aE	$K_cK_bK_aE$
Our tool	0.00917s, 7.36Mb	0.0150s, 10.15Mb	0.0239s, 14.12Mb	0.0965s, 65.094Mb
Other tool	0.0006s, 0.694Mb	0.0015s, 2.006Mb	0.0042s, 4.6104Mb	0.0074s, 11.077Mb

Table 2: *Profiling Results between our tool and Gattinger 2018a, with modification*

We see that the other tool now outperforms ours, in both space and time efficiency.

5.3.3 Conclusions from Profiling

Through looking at the cost centres given to us by GHC, we can see where our program spends the bulk of the time computing. In our tool, the culprit is `models`, which can be seen in Figure 17. Unfortunately there's no more we can do to speed up this function [17], suggesting that the problem lies in the amount of times we call it.

```
models :: Prop p => Set.Set p -> Form p ->
models _  Top          = True
models ps (Not form)   = not $ models ps for
models ps (P form)     = Set.member form ps
models ps (Or forms)   = any (models ps) for
models ps (And forms)  = all (models ps) for
```

Figure 17: *The `models` function*

[17] Short of using a HashSet, which gives us quicker lookup times.

The function is one of the essential parts of the program; it lets us move from state to state.

The issue seems to arise from the final part of the definition of transitions in $\mathcal{ME}^*$. Recall that this is defined as:

$$\delta(q_v, e) = q_{v'}, \text{where } v' = \{p \in \Lambda \mid v \models \mathsf{post}(e, p)\} \text{ if } \models \mathsf{pre}(e) \tag{1}$$

where, given a world where a set of propositions v is true and an event e, we go through each proposition p in the set of propositions for the current model and check if the set of propositions v *model* the postcondition of the given proposition p and the event e. This occurs every time we wish to make a transition between two sets in our automata; clearly, this piece of code is being called a lot. However, this does not seem to be a particularly bad thing in itself; rather the opposite, it seems essential that this piece of code would be called a lot.

To understand the differences, we mainly need to consider the algorithmic differences. The other tool first enumerates the possible set of call sequences in a depth-first manner, after which it finds the first successful sequence of these calls. It does this latter task like a conventional model checker, and as such can use more powerful model checking techniques. In contrast, our tool performs a slightly different task; it builds a structure with which it can find paths through the gossip graph, and then finds the shortest.

The point at which this difference is really shown is in the generation of indistinguishable states. In our tool we progressively keep track of the states indistinguishable from our current one, and update each of these states with each transition, leading to a lot of time spent in the `models` function. In contrast, the other tool takes the string of calls to be checked against and generates all of the other strings of calls indistinguishable to it, and then checks if these model the target formula. The latter technique is much more computationally efficient than ours; the generation of these indistinguishable call sequences is much more straightforward and requires much less heavy lifting than maintaining this set of indistinguishable states.

One might ask why we don't copy this method of simply maintaining the set of strings of indistinguishable events which could have happened in the states of our automaton. This is a fine idea, however a problem arises when we come to considering the successful states. In the other tool it is known that the sequence of calls is "final", in the sense of a model checker - we want to know if this is successful or it isn't. But in ours we only stop either when we can make no more calls, or we are in a successful state. As such, we need to know if we are in a successful state *every time we enter one*, and the only way we can do this is by performing each string of indistinguishable calls on the initial state and evaluating the target formula at the resultant states; it is clear that this will be less efficient.

Another method we spend a lot of time on is the BFSM method `enqueue`. In this we take the existing queue and append onto it the set of states that are next to be visited. This function uses a lot of memory and time as it has two memory-costly functions inside it; namely `filter` and `(++)`. This was an interesting function to investigate and try and optimise. We only need to perform two actions on our queue:

1. Read an item from the front

2. Append items onto the back

In Haskell, list concatenation has runtime $O(n)$, where n is the length of the list being appended to[18]. This is far from ideal - however the `Data.Seq` package offers us constant-time access to the first and last element of a sequence. This seems like it would be perfect for our situation, however **for some reason currently beyond me** the time nearly doubles and the space used balloons.

Another thing we could have used to reduce computation is the monotonicity of the propositions true at a state. At a state in the gossip protocol, if some set v of propositions are true and we make some call e, then the set of propositions at the state we move to, v', is guaranteed to be a superset of v. Consider what this means; there is no phone call that can be made that will make anyone know any *less*. Hence we could change Equation 1 to

$$\delta(q_v, e) = q_{v'}, \text{where } v' = \{p \in \Lambda \setminus v \mid v \models \mathsf{post}(e, p)\} \cup v \text{ if } \models \mathsf{pre}(e) \tag{2}$$

This is because we know that the propositions in v will be true at the state $\delta(q_v, e)$; hence we do not need to check again whether or not they will be true.

However I fear this would be an irresponsible change to make to our code. Remember that we do not want our system to solely plan for the Gossip problem; rather we want it to be capable of doing so for *any* epistemic model and propositional event model. We have an example of a model where this occurs in Figure **??**; although this is quite a toy example, it does indeed show that such cases are possible. In the interests of portability then, it would be unwise to make such a change.

5.4 Negatives of Design

[18] I.e., in the command `xs ++ ys`, n is the length of `xs`.

Bibliography

A. Hajnal, E. C. Milner and E. Szemere (1972). "A cure for the telephone disease". In: *Canadian Math Bulletin*.

Argueta, Arturo and David Chiang (2018). "Composing Finite State Transducers on GPUs". In: *Association for Computational Linguistics* 58.

Attamah, M et al. (2014). "Knowledge and gossip". In: *Frontiers in Artificial Intelligence and Applications* 263, pp. 21–26. DOI: 10.3233/978-1-61499-419-0-21.

Aucher, Guillaume and Thomas Bolander (2013). "Undecidability in Epistemic Planning". In: *IJCAI - International Joint Conference in Artificial Intelligence*.

Aucher, Guillaume, Bastien Maubert, and Sophie Pinchinat (2014). "Automata Techniques for Epistemic Protocol Synthesis". In: *Proceedings of the 2nd International Workshop on Strategic Reasoning*.

Baker, Brenda and Robert Shostak (1972). "Gossips and Telephones". In: *Discrete Mathematics*.

Bolander, Thomas and Mikkel Birkegaard (2011). "Epistemic planning for single- and multi-agent systems". In: *Journal of Applied Non-Classical Logics*.

Burch, J. R. et al. (1992). "Symbolic Model Checking: 10^{20} states and beyond". In: *Information and Computation*.

Camilleri, John J. (2015). *fst*. Online; accessed 15-April-2019. URL: https://hackage.haskell.org/package/fst.

Fagin, Ronald et al. (1995). *Reasoning about Knowledge*. MIT Press.

Garson, James (2018). *Modal Logic*. The Stanford Encyclopedia of Philosophy. Online; Accessed 11/04/19. URL: https://plato.stanford.edu/archives/fall2018/entries/logic-modal/.

Gattinger, Malvin (2018a). *Explicit Epistemic Model Checking for Dynamic Gossip*. Online; accessed 7-April-2019. URL: https://github.com/m4lvin/gossip.

— (2018b). "New Directions in Model Checking Dynamic Epistemic Logic". PhD thesis. Institute for Logic, Language and Computation, Universiteit van Amsterdam.

— (2018c). *SMCDEL — An Implementation of Symbolic Model Checking for Dynamic Epistemic Logic with Binary Decision Diagrams*. Tech. rep. Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Ghallab, Malik, Dana Nau, and Paolo Traverso (2004). *Automated Planning and Acting*. Cambridge University Press.

Haeupler, Bernhard et al. (2016). "Discovery through Gossip". In: *Random Structures and Algorithms*.

Kmett, Edward (2018). *machines*. Online; accessed 15-April-2019. URL: https://hackage.haskell.org/package/machines.

Maubert, Bastien and Sophie Pinchinat (2014). "A General Notion of Uniform Strategies". In: *International Game Theory Review*.

Plaza, Jan (2007). "Logics of Public Communication". In: *Synthese*.

R. Apt, Krzysztof, Davide Grossi, and Wiebe Hoek (2016). "Epistemic Protocols for Distributed Gossiping". In: *Electronic Proceedings in Theoretical Computer Science* 215, pp. 51–66. DOI: 10.4204/EPTCS.215.5.

Tijdeman, R. (1971). "On a telephone problem". In: *Nieuw Archief voor Wiskunde (3)* 19, pp. 188–192.

van Ditmarsch, Hans et al. (2016). “Epistemic Protocols for Dynamic Gossip”. In: *Journal of Applied Logic,*

— (2018). “Dynamic Gossip”. In: *Bulletin of the Iranian Mathematical Society*.

van Eijck, Jan (2014a). *DEMO-S5*. Tech. rep. CWI Amsterdam.

— (2014b). *EREL*. Tech. rep. CWI Amsterdam.

Yu, Quan, Ximing Wen, and Yongmei Liu (2013). “Multi-Agent Epistemic Explanatory Diagnosis via Reasoning about Actions”. In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*.