A SYSTEM FOR PLAN RECOGNITION IN DISCRETE AND CONTINUOUS DOMAINS

ALISTAIR SCHEUHAMMER

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
JANUARY 2022

Abstract

For my thesis I seek to implement a programming framework which can be used to model and solve plan recognition problems. My primary goal for this system is for it to be able to easily handle continuous probability spaces as well as discrete ones. My framework is based primarily on the probabilistic situation calculus developed by Belle and Levesque, and is an extension of a programming language developed by Levesque called Ergo. The system I have built allows one to specify complex domains and dynamic models at a high-level and is written in a language which is user-friendly and easy to understand. It has strong formal foundations, can be used to compare multiple different plan recognition methods, and makes it easier to perform plan recognition in tandem with other forms of reasoning, such as threat assessment, reasoning about action, and planning to respond to the actions performed by the observed agent.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Overview

Plan/goal recognition is a sub-field of Artificial Intelligence focussed on identifying an observed AI agent's intention, based on the behaviour observed so far. Plan recognition has many useful applications, including computer games and security. Plan recognition problems (and AI problems in general) can be modelled in both continuous and discrete spaces, but as it stands, most of the existing work in the field of artificial intelligence is focussed primarily on discrete spaces. The system I have developed seeks to provide a framework in which plan recognition problems can be easily modelled and solved, regardless of whether or not the problem features a discrete or a continuous space. This framework allows one to specify complex domains and dynamic models at a high-level and in a user-friendly, easy-to-understand language. It has strong formal foundations and can be used to compare the performance of multiple different methods of plan recognition. It also makes it easy to perform plan recognition in tandem with other forms of reasoning, including threat assessment, reasoning about action, and making plans to respond to the actions performed by the observed agent. For example, one could design a system representing a parking lot where each car is an AI agent, with one car being the "primary agent"; in such a situation, my system could be used to model observing the behaviour of other cars as a plan recognition problem, and the system's beliefs regarding the other cars' intentions could then fuel the primary car's planner, using those predicted intentions to plan its own path through the lot.

1.2 Contributions

The contributions of this thesis are as follows:

1. A framework for modelling both plan and goal recognition problems in the probabilistic situation calculus developed by Hector J. Levesque and Vaishak Belle [3, 4] which can handle both continuous and discrete probability spaces with equal ease. This framework views plan/goal recognition as a form of Bayesian belief update, and can specify complex stochastic domains at a high level through

a user-friendly language. An observed agent's plan library can be specified in this framework using a high-level non-deterministic programming language.

2. An implementation of said framework built on top of Ergo, a variant of the agent programming language Golog. This implementation can be used to perform plan/goal recognition, and uses Ergo to perform approximate Bayesian inference through Monte Carlo sampling.

3. Various experimental problems and benchmarks featuring both discrete and continuous distributions as a sort of showcase of what kinds of problems can be modelled and solved in my system, including one in which the observed agent is a customer in a jewelry store and the system is tasked with predicting whether or not they are there to steal something, buy something, or just browsing the store; one in which the observed agent is a robot moving through an environment and the system is tasked with predicting which of several potential goal destinations is the robot's actual target destination; and one in which the observed agent is a car approaching an intersection and the system is tasked with predicting both where they are planning on going after passing through the intersection and whether they plan on speeding up, slowing down, or continuing at a constant speed to get there.

4. Various statistical evaluations of the system, primarily for the purposes of measuring how different parameters affect the system's running time and accuracy, whilst also comparing the system's performance to existing work.

Ergo, the language my system is largely built on top of, was developed by Hector J. Levesque [22] and is based heavily on the probabilistic situation calculus developed by him and Vaishak Belle [3, 4]. I have received a significant amount of assistance in building my system from my supervisors, Petros Faloutsos and Yves Lespérance. Yves in particular has been very directly involved in the creation of many of the sample problems I have modelled in my system, and has also developed some new execution modes for Ergo which my system heavily relies on. Some of the strategies I have used to solve sample plan recognition problems were also partially inspired by work done by Gal A. Kaminka, Mor Vered, and Noah Agmon [19].

1.3 Outline

Chapter 2 features some necessary background information clarifying just what the field of plan recognition is, the distinction between continuous and discrete spaces, and how that distinction relates to my thesis. The chapter then also features a discussion of existing work in the field and how my work expands on that existing work. Chapter 3 features a discussion of the syntax and semantics of Ergo, the language upon which my system is built. It is divided into the three sections; the first discusses the semantics of what's called a "Basic Action Theory" (a formalism for representing dynamic domains that Ergo is built around), the second discusses the semantics of Ergo programs themselves, and the third discusses the syntax of Ergo programs. Chapter 4 goes into detail on the syntax and semantics of my own system, Ergo4PPR, explaining how some of the relevant functions behave by way of example. Chapter 5 discusses various sample problems that I have already modelled and solved in Ergo4PPR, then provides statistical data I gathered on the performance of the system on these problems, primarily in terms of runtime and accuracy. I also replicate an algorithm developed by Kaminka et al and run a statistical evaluation on it similar to their own for the sake of comparison. Finally, chapter 6 summarizes some final thoughts on the system and discusses various ways my work could hypothetically be expanded upon in the future.

2 Plan Recognition

Plan Recognition is a sub-field of Artificial Intelligence wherein the focus is primarily on predicting the intended actions of an AI agent, rather than planning out a sequence of actions for an AI agent to follow. Keyhole plan recognition is a specific type of plan recognition wherein the system is observing an agent which is unaware that it is being observed, as if the agent is being observed from another room through a keyhole; as a result, the information gleaned from said agent's behaviour is non-interactive and frequently incomplete. Every plan recognition problem discussed in this paper is technically a keyhole plan recognition problem. Goal recognition, also known as intention recognition, is a related field and a special case; the primary distinction between plan recognition and goal recognition is that goal recognition is specifically focussed on predicting the agent's goal, while plan recognition is instead more generally focussed on predicting the entirety of the observed agent's intended plan. For example, if the agent in question were a car trying to find a space in a parking lot, goal recognition would be focussed on predicting which parking space the car will ultimately park in, while plan recognition would instead concern itself with predicting the car's entire path through the parking lot including the final destination. For the sake of simplicity, in this paper the term "plan recognition" will be used as a blanket term including the field of goal recognition.

The input to a plan recognition problem consists of a) a set of possible plans or goals that the observed may follow/try to achieve, or alternatively a prior probability distribution over the set of possible plans/goals; b) some beliefs about the initial state of the world (including information which is known to be true with 100% certainty); c) an outline of every action the observed agent can perform, including information about pre-conditions that must be met for that action to be performed and the possible outcomes of performing that action, with those outcomes potentially including how that action affects the probability distributions representing the system's current beliefs; and d) a sequence of observations (which may or not be noisy) representing the specific sequence of actions performed by the observed agent. Using this information, the system's goal is to determine which of the possible plans/goals the observed agent is most likely following/aiming for; this output would constitute the posterior probability distribution over the plans/goals if the system were formulated as a Bayesian inference problem, and would typically be provided and updated after every observed action is processed by the system. For example, going off the parking lot example described above, the input might be a) either a list of every parking space in the lot (potentially along with the initial likelihood of that space being the observed car's desired space), or a list of every path the observed car might take through the lot (again, potentially along with an initial likelihood for each path); b) the location of every car and parking space currently in the lot, including their location and information about which parking spaces are already occupied and which aren't; c) information about what actions the observed car may perform as they manoeuvre through the lot (driving forward, honking the horn, turning, etc.) and their effects on the world; and d) the sequence of actions performed by the observed car as it tries to find a parking space, while the output might be a probability value for each parking space representing the likelihood of that parking space being the one the observed car is trying to get to, or the same for a set of paths the observed car may be intending to follow.

Plan Recognition is a very active research area [29, 16] and has many applications. These include human robot interaction [30], cognitive assistance [10], computer games [6], and security [18]. In many plan recognition problems, noise and uncertainty play a major role. Most obviously, there is uncertainty in regards to the plans/goals the observed agent has chosen to pursue, but there can also be noise in the observations themselves, such as due to imperfect motors on the observed agent's part (i.e. say the agent intends to move 5 feet forward but winds up moving 5 and a half feet instead by mistake) or due to inaccurate sensors on the observer's part that can't measure observations with 100% precision.

2.1 Related Work

2.1.1 Early Work

Much of the early work in the field of plan recognition was not probabilistic in nature, instead being focussed on determining an agent's goal or plan without estimating the likelihood of the hypothesis [21, 20, 1]. However, over time, probabilistic approaches became more and more common. Some of these include [9], who argued that plan recognition was an abductive reasoning problem and could be solved using Bayesian probabilistic inference, and [8], who showed that plan recognition could also be solved using Hidden Markov Models (HMM, often used in signal processing). These days, much of the work has shifted to probabilistic models, such as [13], who presented an algorithm for solving plan recognition problems based on a model of plan execution, and [12], who defined what it means for an agent to be in the process of performing an action. One particularly notable development was that of [26, 27], who developed a "plan-recognition-as-planning" approach. The basic idea of this is that, for every goal an agent could possibly have, a pre-existing AI planner is used to generate an optimal plan taking the observed agent to that goal. Then, as the agent begins performing actions, the sequence of actions it has performed so far is compared to the sequences corresponding to these optimal plans, and probability values are generated for each goal based on the degree to which the agent's actions matches each plan, with the most likely goal being the one for which the agent's actions comes closest to matching those of that goal's optimal plan.

2.1.2 Plan Recognition in Continuous Domains

Today, most of the existing work in the field focusses specifically on plan recognition in discrete domains, where every aspect of the world (including the probability distribution functions in particular) is given a discrete representation. However, many application areas (such as robotics and autonomous vehicles) involve either continuous spaces or spaces with both discrete and continuous elements. In a continuous space, aspects of the world are instead represented with an uncountably infinite range of values. Mathematically, probability distribution functions in discrete domains are summed over a range of possible values to produce the probability that a random variable falls within that range; by contrast, continuous domains use integration over probability density functions to accomplish the same thing. This can be challenging to represent perfectly via a computer program, and often some level of discretization (i.e. converting a continuous model to a discrete one) must be used. At present, though, very little work exists in the field of plan recognition in continuous spaces compared to discrete spaces.

Most of the notable pre-existing work that does exist in the field of plan recognition in continuous spaces was conducted by Gal Kaminka, Mor Vered, and Noa Agmon [19, 31] who developed a method for plan recognition (including goal recognition) in continuous spaces which involves comparing the actions an agent has executed to plans generated using pre-computed off-the-shelf planners. This is in turn based heavily off of the "plan-recognition-as-planning" approach of Ramírez and Geffner [26, 27]. While Ramírez and Geffner's method was designed specifically for systems using discrete probability distributions, Kaminka et al. worked to adapt it to allow for continuous distributions as well. Another result of Kaminka et al.'s work on plan recognition in continuous spaces is the discovery that any plan recognition problem in a continuous space can be sufficiently represented by an equivalent problem in a discrete space so long as the discretization level is sufficiently small [19]. Therefore, one can feel free to represent the continuous probability space of any plan recognition problem as a discrete space without worrying about information being lost in the discretization process, provided their system can adapt its discretization level to the specific problem it has been given. The primary factor distinguishing my work from that of Kaminka et al. is that my work is focussed on developing a general system in which plan recognition problems can be easily modelled and solved using continuous and discrete probability spaces, whereas the work done by Kaminka et al. is focussed more on developing specific methods for solving plan recognition problems in continuous spaces. One could easily use my system to model a plan recognition problem and then solve it using the methods developed by Kaminka et al., or one could solve it using another method entirely if they so chose.

2.1.3 Modelling Uncertainty and Change in the Situation Calculus

Another thing worth noting is much of the existing work in the field of plan recognition has focussed more on algorithms than it has representation; this is the case with the work done by Kaminka et al. Many simplifying assumptions are also typically made about domains and plan structure. As an example, it is often assumed that events in the world happen sequentially, i.e. no two events happen concurrently. In the more general field of reasoning about action, meanwhile, a lot more work has been done on crafting logical frameworks for representing and reasoning about dynamic domains. Much of this work is set in the situation calculus [25, 28] and addresses issues such as solving the frame problem, concurrency, sensing, and more [28]. High-level agent programming languages such as Golog [23] and ConGolog [14] have been developed (and will be discussed more thoroughly in the next section), and the situation calculus has been extended to take into account noisy sensors and actuators and to support reasoning about uncertainty using Bayesian probabilistic models. This framework supports both discrete domains [2] and hybrid discrete-and-continuous domains [5], is based on first-order logic, and can handle infinite domains. It also supports complex noisy action models with effects that depend on context.

The system I have built is very heavily based on pre-existing planning systems built by Hector J. Levesque and Vaishak Belle. While the work done by Belle and Levesque is not specifically focussed on the field of plan recognition, it nonetheless serves as solid base for my work due to their work being heavily focussed on modelling uncertainty in the situation calculus with both discrete and continuous probabilistic spaces. All of the languages developed by Belle and Levesque were built on top of Racket Scheme. The first language they developed was called Prego [3]. Prego modelled uncertainty using fluents (effectively variables which represent values that can change over time) and actions which can affect the values of fluents. More specifically, any given fluent would be given an initial, either discrete or continuous, set of possible values according to some probability distribution (note that it can be such that a given fluent simply has a 100% chance of being given some specific value). Every action would then affect the probability distributions of some number of fluents, as appropriate. Belief updates in Prego were computed using something called "logical regression"; given a situation-suppressed expression e (which could possibly be a belief expression) and an action sequence σ, an equivalent expression e' can be obtained such that the value of e' in the initial situation is equal to value of e in the situation resulting from executing σ in the initial situation.

2.1.4 Agent Programming in the Situation Calculus

Another important language in the field of plan recognition is Golog. Golog was an agent programming language developed by Hector J. Levesque et al. in the late 1990s [23], originally implemented in Prolog. Golog was built to model a dynamic world which an AI agent could interact with, and the main point of interest distinguishing Golog from similar languages was that it allowed agent behaviour to be described using programs at a higher-level of abstraction than what existed previously. These programs contained high-level actions whose pre-conditions and effects were specified by an action theory, and the Golog interpreter could reason about the effects of these actions to find successful executions of non-deterministic programs. ConGolog [14], meanwhile, is an extension of Golog which incorporates concurrency, allowing multiple different actions to occur simultaneously. Another extension of Golog called DTGolog exists [7], which, when given a non-deterministic program, determines the optimal method of executing that program.

Golog was an instrumental component in the next language Belle and Levesque worked on, known as Allegro [4]. By including the agent programming functionality of Golog wherein the program's tests are evaluated against the agent's beliefs, Allegro extended Prego with the ability to use its uncertainty models in the programming of actual, physical, robots. Allegro also improved on how Prego computed belief updates by using progression and Monte Carlo sampling (in which a number of samples are generated representing different possible world states, with each sample having a different value for all probabilistic elements and a weighting representing the liklihood of that sample being the true world state). By calculating belief updates in this manner, Allegro could keep track of an agent's current beliefs in real-time. In a sense, Prego ran offline, producing probability values for belief statements given a hypothetical world state, while Allegro ran online, being able to determine the current probability of any given belief statement based on which specific actions have occurred so far.

The language my system is primarily built on is known as Ergo. Ergo is an implementation of Golog in Scheme by Hector J. Levesque alone [22]. Though Ergo pre-dates Prego and Allegro, much of the functionality for computing uncertainty featured in Prego and Allegro was added into Ergo at a later date. Ergo features the probabilistic aspects of Prego and Allegro as well as Allegro's ability to use Golog-like programs (in this case called "Ergo Programs") to describe agent behaviour, producing a robust and easy-to-understand tool for modelling AI environments with noise and uncertainty regarding an agent's actions and beliefs. Ergo also provides the ability to directly update sample weights as if they were fluents; if a sample's weight is set to 0, then the world-state described by that sample is officially considered impossible. If every sample is given a weight of zero, the system shuts down entirely. Employing this system allows users to define the likelihood with which any scenario occurs with ease.

2.1.5 Plan Recognition in the Situation Calculus

Some prior work on Plan Recognition exists which incorporates the situation calculus. [12] presents a formalization wherein plans are represented as Golog programs which feature two additional constructs: σ, which can match any sequence of actions, and $\alpha_1 - \alpha_2$, which only matches to executions of plan α_1 which do not match with an execution of plan α_2. The second of those two constructs is quite powerful due to its ability to allow one to specify plans both in regards to what can happen and in regards to what must not happen. This allows programs in the plan library to effectively serve as "plan recognition templates" (rather than serving as actual plans an agent would follow), which cannot be done in most plan recognition frameworks. This can be useful for monitoring applications in which one wants to both represent policies and also detect violations of those policies.

[12]'s approach is reformulated and extended by [17] in two main ways. The first is that their formulation supports hierarchical plans by making note in the plan recognition hypothesis when a sub-plan or sub-procedure started and finished its execution, which helps to clarify the structure of the observed agent's behaviour and aids in predicting its future actions. The second is that their formulation incorporates the transition semantics of ConGolog and supports incremental plan recognition, wherein the set of hypotheses is updated after each new action is observed.

[12]'s approach is also given a probabilistic extension in [11] which assigns probabilities to plan recognition hypotheses; this approach only handles discrete probability distributions, however.

3 Syntax and Semantics of Ergo

As my system is an extension of Ergo, the mathematical foundations upon which is built are the same as those of Ergo, using the same basic syntax and semantics. Ergo is founded on top of the situation calculus [25, 28] and its probabilistic extensions [2, 5]. The semantics of the Ergo programs are based on Golog [23] and ConGolog [14].

3.1 BAT Semantics

The situation calculus is a dialect of predicate logic that features the means to represent many different types of "sorts" (i.e. categories), including actions, situations, and objects. *Situations* represent possible world histories, with the initial situation being represented by the constant S_0. The functional symbol *do* is used to represent sequences of actions, with $do(a, s)$ representing the situations which results from performing action a in situation s. *Fluents*, meanwhile, are functions which vary in value depending on the situation, and can represented as a function taking a situation as an argument (for example, $\textit{robot-pos}(s)$). There is a finite number of fluents, $f_1, \ldots, f_k$, and their values may range over any set, including the set of real numbers. The formulas appearing in the initial belief state and action declaration of Ergo are situation calculus formulas ϕ, sans the situation argument of the fluents involved (this is called a *situation suppressed formula*). $\phi[s]$ denotes the same formula ϕ with the situation argument restored by some situation s.

Dynamic domains are represented using Basic Action Theories (BAT) [5] using four special predicates: *Poss* (which indicates whether or not an action is possible in a given situation), p (which indicates the probability that doing some action a in some situation s will yield some situation s'), l (which indicates the probability that some action a will occur in some situation s), and *alt* (which represents possible alternate actions for noisy actions). To go into more a detail, a BAT consists of the following:

- An *initial state theory* $\mathcal{D}_0$ which specifies the probability distribution over the initial set of possible world states using an axiom of the form:

$$p(s, S_0) \equiv INIT(f_1, \dots, f_n)[s]$$

- *action precondition axioms* which specify wether an action is possible in a given situations:

$$Poss(a, s) \equiv APA_a(\vec{x})[s]$$

- *likelihood and alternate actions axioms* which describe the outcomes of executing noisy actions:

$$\begin{aligned} &l(a(\vec{x}), s) = LH_a(\vec{x})[s] \\ &alt(a, u) = a' \end{aligned}$$

- *successor state axioms* (SSA) which specify how actions alter the values of fluents (note that this includes Reiter's solution to the frame problem):

$$f(do(a, s)) = SSA_f(a)[s]$$

To clarify the definition of the alternate action axiom, a is some noisy action, u is the term replacing the action's noisy argument, and a' is the action the agent knows was executed as a result. For example, say we have the action $nfwd(x, y)$ representing a mobile agent attempting to move forward, where x is the distance the agent intends to move while y is the distance the agent actually moves. $alt(nfwd(x, y), z) = nfwd(x, z)$ specifies that if $nfwd(2, 2.79)$ occurs, all the agent actually knows is that $nfwd(2, z)$ occurred for some unknown value of z (and similarly for other values of x and y) – or, put another way, the agent only knows that it intended to move forward 2 units; it does not know precisely how many units it actually travelled.

A special SSA for the p fluent exists which specifies the value of p is generated from a given pair of situations and an action, namely by multiplying the p-value of the first situation by the likelihood of the given action occurring:

$$\begin{aligned} &p(s', do(a, s)) = u \equiv \\ &\quad \exists s''[s' = do(a, s'') \wedge Poss(a, s'') \wedge \\ &\qquad u = p(s'', s) \times l(a, s'')] \\ &\quad \vee \neg\exists s''[s' = do(a, s'') \wedge Poss(a, s'')] \wedge u = 0 \end{aligned}$$

Given a BAT as described above, the *degree of belief* in a situation suppressed formula ϕ in situation s is calculated as follows:

$$Bel(\phi, s) \overset{\text{def}}{=} \frac{1}{\gamma} \int_{f_1,\dots,f_n} \int_{u_1,\dots,u_k} Density(\phi, s^*)$$

The normalization factor γ is the value of the numerator after a) replacing ϕ with $True$ and b) setting $s^* = do([alt(a_1, u_1), \dots, alt(a_i, u_k)], S_0)$ (if $s = do([a_1, \dots, a_k], S_0)$). The full definition of the *Density* function has been omitted here, but roughly what it does is apply the successor state axiom for p to any situation term produced by a noisy action (i.e. the various possible outputs of $alt(a, u)$, for some action a). For a given situation-suppressed formula ϕ, *Density* will either output the p-value of its situation argument (if ϕ holds) or 0 (if ϕ does not hold). Ultimately, the degree of belief formula integrates over the possible initial values of the fluents and the possible outcomes of noisy actions. For discrete distribution, the integral in this formula will be replaced by a sum.

3.2 Program Semantics

Ergo uses an implementation of ConGolog [14] to construct Ergo programs which can be used to control the behaviour of one or more agents. The following is a list of constructs present in ConGolog:

$$\delta ::= nil \mid \alpha \mid \varphi? \mid \delta_1;\delta_2 \mid \delta_1|\delta_2 \mid \pi x.\delta \mid \delta^* \mid \delta_1\|\delta_2$$

Each of the above terms is described as follows:

1. nil - the empty program, which does nothing and is terminated by default.

2. α - an atomic action, possibly with parameters.

3. $\varphi?$ - a situation-suppressed formula whose truth-value is being tested (with $\varphi[s]$ denoting the formula obtained upon restoring the situation argument s as before).

4. $\delta_1;\delta_2$ - A set of two programs executed in sequence.

5. $\delta_1|\delta_2$ - A nondeterministic choice between two programs, only one of which is executed.

6. $\pi x.\delta$ - A program executed with the variable x bound to some value chosen nondeterministically.

7. δ^* - A program executed zero or more times, with the number chosen nondeterministically.

8. $\delta_1\|\delta_2$ - Two program executed concurrently by way of interleaving.

More complex programs, such as those featuring conditionals or loops, can be defined in terms of the above constructs. The semantics of ConGolog are specified in terms of single-step transitions using two predicates:

1. $Trans(\delta, s, \delta', s')$ - A predicate which holds is if one step of the program δ in situation s yields s' as the new situation and δ' as the remaining program.

2. $Final(\delta, s)$ - A predicate which holds if δ is capable of legally terminating in situation s.

The definitions of $Trans$ and $Final$ used here are as they are used in [15], which differs from [14] in that the term $\varphi?$ does not yield a transition (i.e. $Trans(\varphi?, s, \delta', s')$ only holds if $\delta' = \varphi?$ and $s' = s$), but nonetheless satisfies $Final$ when it is also satisfied. Combining $Trans$ and $Final$ produces the predicate $Do(\delta, s, s')$, which holds as long as executing the program δ in situation s produces s' as a legal terminating situation. This predicate is defined as $Do(\delta, s, s') \doteq \exists \delta'. Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$ (note that $Trans^*$ is the reflexive transitive closure of $Trans$).

Finally, it is also useful to introduce the idea of a ConGolog program being *situation-determined* (SD). This means that for any possible sequence of transitions produced by the program, the remaining program can be uniquely determined using the resulting situation. This is represented by the predicate $SituationDetermined(\delta, s)$. Formally,

$$SituationDetermined(\delta, s) \doteq \forall s', \delta', \delta''. Trans^*(\delta, s, \delta', s') \wedge Trans^*(\delta, s, \delta'', s') \supset \delta' = \delta''$$

To give an example, the program $a; (b \mid c)$ is SD in situation S_0 (i.e. the initial situation) as there is a unique remaining program $(b \mid c)$ in $do(a, S_0)$. However, the program $(a; b) \mid (a; c)$ is not SD in the same situation, since there are multiple possible remaining programs after executing $do(a, S_0)$ (i.e. either b or c).

3.3 Syntax of Ergo Programs

Next let's discuss the specific syntax Ergo uses in the construction of programs, which as mentioned is built directly on top of the pre-existing semantics of ConGolog. Every Ergo program either ultimately succeeds or fails depending on the circumstances, and there are a wide variety of different expressions that can be used in an Ergo program. The full list of available expressions are as follows [22]:

1. :nil - Corresponds to *nil* above, and always succeeds.

2. :fail - Like :nil, except that it always fails.

3. (:test *fexpr*) - Corresponds to $\varphi?$, succeeding or failing based on the truth value of *fexpr*.

4. (:act *action*) - Corresponds to α. Fails only if the action has a prerequisite that is not satisfied when this action is reached, and succeeds otherwise.

5. (:begin *pgm1 ... pgmn*) - Corresponds to $\delta_1; \delta_2$, executing each Ergo program in sequence. Succeeds if every program succeeds and fails otherwise.

6. (:choose *pgm1 ... pgmn*) - Corresponds to $\delta_1 | \delta_2$, nodeterministically choosing a single program and executing it. Succeeds if the chosen program succeeds and fails otherwise.

7. (:if *fexpr pgm1 pgm2*) - Executes *pgm1* if *fexpr* evaluates to $True$ and *pgm2* otherwise. Succeeds if the chosen program succeeds and fails otherwise.

8. (:when *fexpr pgm1 ... pgmn*) - Behaves the same as (:if *fexpr* (:begin *pgm1 ... pgmn*) :nil)

9. (:unless *fexpr pgm1 ... pgmn*) - Behaves the same as (:when (not *fexpr*) *pgm1 ... pgmn*)

10. (:until *fexpr pgm1 ... pgmn*) - Repeatedly executes (:begin *pgm1 ... pgmn*) until *fexpr* becomes true. Succeeds and fails under the same circumstances as :begin.

11. (:while *fexpr pgm1 ... pgmn*) - Behaves the same as (:until (not *fexpr*)*pgm1 ... pgmn*)

12. (:star *pgm1 ... pgmn*) - Corresponds to δ^*, except that it executes a sequence of programs zero or more times instead of a single program. Succeeds and fails under the same circumstances as :begin.

13. (:for-all *var list pgm1 ... pgmn*) - Repeatedly executes (:begin *pgm1 ... pgmn*) once for every value in the list *list*. Each run will set the value of the variable *var* to the current value from *list*. Succeeds and fails under the same circumstances as :begin.

14. (:for-some *var list pgm1 ... pgmn*) - Corresponds to $\pi x.\delta$, choosing the value of *var* nondeterministically from the values of *list*. Succeeds and fails under the same circumstances as :begin.

15. (:conc *pgm1 ... pgmn*) - Corresponds to $\delta_1 \| \delta_2$, concurrently executing all the programs in the given list. Single steps are nondeterministically interleaved. Succeeds and fails under the same circumstances as :begin.

16. (:atomic *pgm1 ... pgmn*) - Behaves the same as (:begin *pgm1 ... pgmn*), except without interleaving concurrent programs if any exist. Succeeds and fails under the same circumstances as :begin.

17. (:monitor *pgm1 pgm2 ... pgmn*) - Execute *pgm1* before every step of *pgm2* and do the same for all subsequent programs. Succeeds and fails under the same circumstances as :begin.

18. (:<< *fexpr1 ... fexprn*) - Evaluate all the expressions, then succeed.

19. (:>> *fexpr1 ... fexprn*) - Evaluate all the expressions, then fail.

20. (:let ((*var1 fexpr1*) ... (*varn fexprn*) *pgm1 ... pgmn*) - Behaves the same as (:begin *pgm1 ... pgmn*), except in an environment where every *vari* is set to the value of *fexpri*. Succeeds and fails under the same circumstances as :begin.

21. (:wait) - Pause until an exogenous action occurs, then succeed.

22. (:search *pgm1 ... pgmn*) - Behaves the same as (:begin *pgm1 ... pgmn*), except it guards against failure by looking ahead for instances of nondeterminism and ensuring that the choice made leads to successful termination.

In order to execute an Ergo program, the Ergo command ergo-do must be called. The syntax of ergo-do is (ergo-do *mode pgm*), where *pgm* is the program being executed and *mode* is the execution mode, typically either 'online (meaning the program interacts with an external source by sending it endogenous actions, by receiving exogenous actions from it, or both) or 'offline (meaning the program does not interact with an external source).

Any other Ergo syntax relevant to my system will be discussed in the following section, specifically in regards to how it is is relevant to my system.

4 Ergo4PPR

The system I have been working on with my supervisors is an extension of Ergo tentatively entitled Ergo4PPR. The main factor by which systems built in Ergo4PPR differ from those built in base Ergo is that they are more focussed specifically on plan recognition, while base Ergo is more focussed on planning in general (as well as high-level program execution). A system built in Ergo is typically meant to represent a single AI agent, with fluents representing information known to the agent and the agent's behaviour being described by an Ergo program as outlined above. Ergo4PPR, however, provides a simple means by which one can describe an entire world containing one or more agents, with fluents being global information. A major factor in the ability to do this is a new execution mode for Ergo programs which can be used to update a program by sending it a single action, whereupon the remaining program after that action's execution will be returned. It is assumed that all programs are situation-determined, as described above. By using this execution mode, Ergo programs representing an agent's behaviour can be to stored in fluents which can then be updated accordingly with each action taken by the observed agent (or agents as the case may be). It is also possible to have multiple different subprograms each containing a different possible "plan" a given agent may follow. Ergo4PPR also generally provides a convenient, easy-to-use framework for describing tools which the system may use to observe each agent's actions, particularly in regards to the ability to introduce noise and uncertainty both into the actions themselves and into the observations of those actions.

4.1 Implementation Details

One of the primary benefits of Ergo and Ergo4PPR is in its ability to use Monte Carlo sampling to compute belief updates. Essentially, the system works by generating some number of samples, each representing a possible world-state, with a weighting representing the probability of that sample being the "true" world state. The degree of belief that a fluent had a given value was ultimately calculated as the weighted average number of samples wherein that fluent had that value. The number of samples to use is something which the programmer could set whilst designing the system, and the more samples there were, the more likely it was that the system's degree of belief in the value of each fluent would be close to the correct value. Calculating belief updates in this manner makes it easy to efficiently maintain an estimate of the belief state in real-time as observations are acquired.

The main extensions that Ergo4PPR makes to Ergo are the introduction of two new execution modes for Ergo programs. By default, Ergo has two main execution modes, `online` and `offline`. In `online` execution, the system receives exogenous actions from an external source (such as a sensor) which it then uses to update the world state, sending the endogenous actions out to effectors. `offline` execution, meanwhile, means that the program is executed internally without interacting with the environment. The two new execution modes are essentially extensions of the paradigms used in the `online` and `offline` modes. The first new execution mode is the `onlineSynchronized` mode, which repeatedly alternates between processing exogenous actions and progressing through the Ergo program, in a way which removes the need for complex synchronization between the two. The second new execution mode is the `offlineStepMatch` mode, which takes an action as an argument and tries to execute it on the Ergo program, returning either a pair containing that action and the remaining program after its execution (in the case where the action can be executed by the program), or an empty list (in the case where the action cannot be executed by the program). If more than one possible remaining program exists, it returns the first it finds; if the program is situation-determined, though, there won't be more than one remaining program. Situation-determined programs like these are used to model the possible behaviours of the observed agent.

4.2 Syntax

In the Ergo4PPR system a domain is specified using a BAT, as described above. A sample of some of the syntax used by Ergo is summarized in Table 4.1. As previously mentioned, a fluent is essentially a variable whose value can change over time.

To illustrate how this works specifically in the context of Ergo4PPR, let us examine a simple example involving targets and an agent which shoots at those targets. The agent is capable of changing its aiming angle and firing at a target; there are multiple targets of varying sizes and point values, with smaller targets being worth more but being harder to hit. In the specific example described in this paper, there are three targets: a target located at $30°$ of size $10°$ and of value 30, a target located at $60°$ of size $30°$ and of value 10, and a target located at $90°$ of size $20°$ and of value 20. Note that target sizes are measured in degrees, centred on their position value; i.e. the first target spans a range from $25°$ to $35°$. This system's initial state is specified using fluents, as follows:

Table 4.1: Sample Ergo Syntax

Syntax	Definition
(define-states ((i *num*)) *fluent1 val1* ... *fluentn valn*)	Defines the initial value of *fluent1* to be *val1* and similarly for the other fluents. Also sets the number of samples used for Monte Carlo sampling to *num*.
(define-action *action* #:*args fluent1 val1* ... *fluentn valn*)	Defines an exogenous action called *action* which updates the value of *fluent1* to be *val1* and similarly for the other fluents. Optional arguments such as pre-conditions are specified using *args*.
(GAUSSIAN-GEN *mu sigma*)	Represents a gaussian random variable of mean *mu* and standard deviation *sigma*.
(GAUSSIAN *val mu sigma*)	Returns the probability of a gaussian random variable of mean *mu* and standard deviation *sigma* having the value *val*.
(UNIFORM-GEN *low high*)	Represents a uniform random variable whose value ranges from *low* to *high*.
(UNIFORM *val low high*)	Returns the probability of a uniform random variable whose value ranges from *low* to *high* having the value *va1*.
(DISCRETE-GEN *v1 p1* ... *vn pn*)	Represents a discrete random variable whose value is *v1* with probability *p1*, *v2* with probability *p2*, etc. The sum *p1* + ... + *pn* must add to 1.
(DISCRETE *val v1 p1* ... *vn pn*)	Returns the probability of a discrete random variable whose value is *v1* with probability *p1*, *v2* with probability *p2*, etc. having the value *val*. Again, the sum *p1* + *pn* must add to 1.
(UNIFORM-DISCRETE-GEN *n*)	Represents a random variable whose value is any of the integers from 1 to *n*, with each value having a 1/*n* probability of being that variable's value.
(UNIFORM-DISCRETE *val n*)	Returns the probability of a random variable whose value is any of the integers from 1 to *n* (with each value having a 1/*n* probability of being that variable's value) having the value *val*.
(BINARY-GEN *p*)	Represents a random variable whose value is #*t* with probability *p* and #*f* with probability 1 - *p*.
(BINARY *val p*)	Returns the probability of a random variable whose value is #*t* with probability *p* and #*f* with probability 1 - *p* having the value *val*.
(ergo-do #:mode *mode program*)	Executes the Ergo program *program* in the execution mode *mode*.

```
(define-states ((i 1000000))
  decisionPlan 'notDecided
  decisionTarget 'notDecided
  aim 0.0
  someObservedAim 'notSet
  targetHit (vector #f #f #f)
  halted #f
  exoProg (:begin
    (:act choosePlan!)
    (:if (eq? decisionPlan 'greedy)
       (greedy-plan)
       (:if (eq? decisionPlan 'safe)
          (safe-plan)
          (optimal-plan)))
    (:choose (:for-some t '(0 1 2)
       (:act (obsHit! t)))
    (:act obsNoHit!))
    (:act halt!))
  )
```

In the above example, there are six fluents – `decisionPlan`, representing the agent's chosen plan, `decisionTarget,` representing the agent's chosen target; `aim`, representing the angle at which the agent is aiming; `someObservedAim`, representing the angle the system observes; `targetHit`, a vector of boolean values representing whether or not each target has been hit by the agent; `halted`, which indicates whether or not my system has finished its execution; and `exoProg`, which outlines which behaviours are possible for the observed robot (and which also incorporates the system's own sensing capabilities). These behaviours are specified using Ergo programs. In the above example, `exoProg` specifies the agent's behaviour as follows: first, the agent chooses a plan according to some probability distribution, either a "greedy" plan where it aims and shoots at the most valuable target, a "safe" plan where it aims and shoots at the largest target, or an "optimal" plan where it aims and shoots at the target with the largest value × size. Due to the way the three targets are set up in this specific example, each plan will directly correspond to a different specific target. After deciding which target to aim at, the agent executes the plan; afterwards, the system observes which target is hit if any, and then halts.

The three plans are themselves also specified using Ergo programs. The "greedy" plan is defined as follows:

```
(define (greedy-plan)
  (:begin
   (:act chooseTarget!)
   (:act setAim!)
   (:choose :nil (:act setAim!))
   (:starDFS (:act (obsAim! someObservedAim)))
   (:act shoot!)
   )
)
```

First, the agent chooses a target (determined by the adopted plan), then aims at the target either once or twice (to increase its chances of aiming at the correct target). Next, the system obtains zero or more noisy observations of the agent's aim (`:starDFS` is a function defined in the target shooting program itself which recursively executes its argument zero or more times), and then finally the agent shoots. The "safe" and "optimal" plans are the same except that in the former the agent aims one to three times and in the latter it aims only once.

In addition to initializing all the fluents as described above, I have also set the system to use 1000000 states when doing Monte Carlo Sampling under *define-states* above. Since the process by which fluents are assigned their initial values is deterministic, all states will have the same fluent values initially. Each state implicitly has a weight which is initialized to 1/1000000. This weight is represented using a special fluent (appropriately called `weight`) which does not need to be outlined under "define-states". As the system progresses and more and more noisy actions/observations are processed and observed, the states will be updated by sampling the probability distributions associated with those actions. Using `weight`, certain states deemed "impossible" following a sequence of observations can also be ruled out by setting the value of `weight` in those states to zero.

There are four types of actions that can impact the system: accurate actions performed by the agent, noisy actions performed by the agent, accurate sensing actions performed by the system, and noisy sensing actions performed by the system. Actions performed by the agent are exactly as they sound: actions the observed agent performs itself. If these are accurate, then the observed agent does exactly what it intends to do (for example, the `setAim` action causes the agent to aim exactly at its chosen target); meanwhile, if these are noisy, then the observed agent may be off from its intentions by some degree (for example, the `nSetAim` action causes the agent to try and aim at the centre of its chosen target within some margin of error). Meanwhile, sensing actions are ones my system performs to gather information about the world. If these are accurate, then the system can be confident that the information it retrieves is exactly as it appears (for example, the `obsHit` action indicates that a certain target was hit with 100% certainty); if these are noisy, then the system cannot be completely confident that the information it retrieves is exact (for example, the angle returned by `obsAim` indicates that the observed agent is likely aiming within a certain range around the returned angle, but it is not a given that it is aiming exactly at the returned angle). To give an example of the syntax of an action, let us consider the `choosePlan!` action, an internal decision action performed by the observed agent. When this action occurs, the system detects that a decision has been made but is unable observe the result of the decision. `choosePlan!` is defined as follows:

```
(define-action choosePlan! #:sequential? #t
  decisionPlan (DISCRETE-GEN 'greedy (/ 1.0 3.0)
    'safe (/ 1.0 3.0) 'optimal (/1.0 3.0))
  exoProg (let ((res (ergo-do #:mode
    'offlineStepMatch #:matchAct 'choosePlan!
     exoProg))) (if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )
```

This action has no pre-conditions. When this action occurs, the system first sets the `decisionPlan` fluent to one of three values, according to a uniform distribution: greedy, safe or optimal. Then, the `exoProg` fluent is updated; the value used to update this fluent is a fair bit more complicated, but to put it simply, `exoProg` is updated by calling ergo-do in `offlineStepMatch mode`, passing in the action `choosePlan!` to be processed. If the action being called on the system doesn't match up with the next action as specified by `exoProg`, then `exoProg` is set to `:fail`; otherwise, it is set to the remaining program. By using the `offlineStepMatch` mode, the program can easily be advanced by a single step for each action. Finally, the weight of the current sample is set to 0 if `exoProg` has been set to `:fail`, remaining unchanged otherwise. Updating the weight in this manner is necessary so as to account for the possibility of a given sample assuming the agent has chosen a specific plan, only to observe an action which is incompatible with that plan; for example, if `decisionPlan` were set to `optimal`, only for multiple `setAim!` actions to be observed. The `chooseTarget!` action is defined similarly (though it behaves deterministically), with the system setting the `decisionTarget` fluent to a target determined by the chosen plan.

The `setAim!` action is a deterministic, non-noisy action performed by the agent which sets the value of the `aim` fluent to the centre of the selected target. If it is defined as follows:

```
(define-action setAim! #:sequential? #t
  aim (vector-ref targetPosition decisionTarget)
  exoProg (let
              ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'setAim! exoProg
                 )))
              (if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )
```

Note that the system does not directly observe what the agent sets its aim to; it just knows the logic by which the agent operates (i.e. that it always sets its aim to the centre of its chosen target with 100% accuracy). Next we have the `nSetAim!` action, which the noisy counterpart of `setAim!`:

```
(define-action nSetAim! #:sequential? #t
  aim (GAUSSIAN-GEN (vector-ref targetPosition decisionTarget) 5.0)
  exoProg (let
                  ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'nSetAim!
                     exoProg)))
                  (if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )
```

Both the `setAim!` and `nSetAim!` actions produce the same results, except that `nSetAim!` specifies that the `aim` fluent should be set to `(GAUSSIAN-GEN decisionTargetPosition sd)`; rather, it should be set according to a Gaussian probability distribution whose mean is the centre of the selected target (with a standard deviation of `sd`) instead of being set to the exact centre.

Another action worth examining in more detail is the `obsAim!` action, a noise sensing action in which the system gathers information about the observed agent's current aim:

```
(define-action (obsAim! a)  #:sequential? #t
  someObservedAim a
  exoProg (let ((res (ergo-do
    #:mode 'offlineStepMatch
    #:matchAct (list 'obsAim! a)
     exoProg)))(if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0
    (* weight (GAUSSIAN (remainder a 360.0)
      aim 10.0)))
  )
```

Here, the `someObservedAim` fluent is set to the observed angle `a` to ensure that the agent program can execute the action `(obsAim!  someObservedAim)`, given any real angle as the argument. Next, `exoProg` is updated as normal. Finally, the weight of the sample is multiplied by the probability with which the system would observe the agent aiming at an angle of `a` (wrapping around from 360° if necessary), according to a Gaussian distribution where the current value of the `aim` fluent is the mean value, while the standard deviation is 10°.

In order to run the system, `(ergo-do #:mode 'onlineSynchronized observeUpdtLoop)` is called; `observeUpdtLoop` is an Ergo program which repeatedly processes observed actions/observations, updates the system's beliefs as needed, the displays the results of some queries about the system's current beliefs. This runs continuously until the agent halts. This is run in `onlineSynchronized` mode, allowing the system to easily alternate between processing actions and updating the agent program and displaying the results of belief queries accordingly.

Ergo also provides functionality which allows the user the define input and output interfaces through which actions and observations can be sent and received over TCP-IP. The method typically used in my benchmarks/case studies was to simply list the actions in a plaintext file which the input interface would then read from, outputting the results of processing those actions to the standard output.

Information about the system's beliefs can be retrieved using one of two commands. "(sample-mean *number*)" returns the weighted average value across every sample of *number* (provided that *number* evaluates to some numerical value). Meanwhile, "(belief *boolean*)" returns the proportion of the samples in which the boolean statement *boolean* is true. The system can return information in this way whenever appropriate. For example, using my above example, "(sample-mean aim)" could be used to return the average value of the observed agent's aim, while "(belief (eq? decisionTarget 0))" could be used to return the probability/degree of belief that the decisionTarget fluent currently has a value of zero.

Finally, a sample of some the system's output during a run of the target shooting example is shown below. First, after processing the observations `choosePlan!` and `chooseTarget!`, the system outputs the following:

```
(belief (eq? decisionPlan greedy)) returns 0.331045
(belief (eq? decisionPlan safe)) returns 0.32939
(belief (eq? decisionPlan optimal)) returns 0.339565
(belief (eq? decisionTarget 0)) returns 0.331045
(belief (eq? decisionTarget 1)) returns 0.32939
(belief (eq? decisionTarget 2)) returns 0.339565
```

At this stage, each plan and target is equally likely. Next, the system observes two executions of the `setAim!` action. Following the first `setAim!` action, the system is equally likely to be aiming at any of the three targets; however the second execution produces the following beliefs instead:

```
(belief (eq? decisionTarget 0)) returns 0.5012529620628828
(belief (eq? decisionTarget 1)) returns 0.4987470379371172
(belief (eq? decisionTarget 2)) returns 0.0
(belief (aiming at target 0)) returns 0.5012529620628828
(belief (aiming at target 1)) returns 0.4987470379371172
(belief (aiming at target 2)) returns 0.0
```

Since the agent never executes the `setAim!` action twice in the optimal plan, the target chosen by that plan is no longer considered a possible target, and as a result only the other two targets are considered possible, each with a roughly equal probability. Next, the system processes the observation `(obsAim! 39.0)`, producing the following:

```
(belief (aiming at target 0)) returns 0.706824678506833
(belief (aiming at target 1)) returns 0.2931753214919648
(belief (aiming at target 2)) returns 0.0
```

Now the system believes it to be far more likely that the agent is aiming at target 0 than target 1. Finally, the observations `shoot!` and `(obsHit! 1)` are processed, producing:

```
(belief (aiming at target 0)) returns 0.0
(belief (aiming at target 1)) returns 1.0
```

```
(belief (aiming at target 2)) returns 0.0
```

At this stage, the system now knows with 100% certainty that target 1 has been hit (and, in turn, that target is the one the observed agent was aiming at), thereby concluding the system's execution.

The entirety of this example program's code can be found in the appendix.

5 Case Studies

Various plan recognition problems have already been modelled and solved in the Ergo4PPR System. These examples were crafted to demonstrate the power, expressiveness, and utility of the system. Each of these systems work as intended, demonstrating the ease with which the system can be used to represent plan recognition problems in either continuous or discrete spaces. The complete code for all three of these examples can be found in the appendix; note that some examples features multiple code files and have been separated accordingly.

5.1 Examples

Each of the following examples was written as a sort of proof-of-concept to show the variety of different kinds of plan recognition problems that can be modelled and solved in my system. There are three such examples; one exclusively featuring discrete probability distributions, one exclusively featuring continuous probability distributions, and one featuring both. In each case, the probability values generated by the system were as expected.

5.1.1 Example 1 - Jewelry Store

The first example I developed is perhaps the simplest one, featuring an entirely discrete probability space. In this example, the agent being observed is an individual within a jewelry store. There are three main goals that this agent may have: steal (in which the agent steals an item from the store), browse (in which the agent examines an item from the store, but puts it back before leaving), and buy (in which the agent makes a purchase). Each of these three possible behaviours are quite similar, but have subtle differences between them which the system uses to try and predict which goal the observed agent has in mind, and as each of the observed agent's actions are observed, the beliefs are updated accordingly. Additionally, the observed agent also has a choice of which of three pieces of jewelry to interact with – a bracelet, a necklace, and a ring. The chances of which item the agent would interact with are different depending on whether or not the agent is planning on stealing the item, buying the item, or simply examining the item, and as such which item the agent chooses also has a impact on the likelihood of each plan. The agent will also look around the room some number of times; specifically, the agent is guaranteed to look around the room at least once no matter what, has a low chance of looking around a second time when browsing or buying and item, and has a guaranteed chance of looking around a second time when planning on stealing an item.

5.1.2 Example 2 - Robot with Multiple Goals

The second example problem I built is a somewhat more complicated one primarily featuring continuous distributions. This problem is also notable for being somewhat of a template from which other, even more detailed problems may be built. In this problem, the observed agent is a mobile robot trying to manoeuvre its way through an environment featuring various obstacles, ultimately hoping to reach one of four destinations. The objective of the system is to use information about the robot's path to try and predict which destination the robot is heading towards. Note that the robot cannot execute any actions other than choosing a destination and turning and moving throughout the environment. Various different methods of calculating the probability of each destination being the true destination were considered and tested. The first and most complicated method was to discretize the environment and then use the A* algorithm [32] to calculate the optimal path to each destination – note that this is a very similar strategy for plan recognition as the one derived by [31], though this is largely a coincidence. At each time step, the robot's current path would be compared to these optimal paths and for each destination a probability value would be generated as the (normalized) degree to which the robot's path matches that destination's optimal path. The optimal paths would also be recalculated if the robot's path deviated too far from any of them. The second and third methods used to calculate the probability for each goal, meanwhile, were significantly simpler, but also less effective. The second method calculated the distance between the robot and each goal both before and after every move action. The probabilities were then calculated as the (again, normalized) degree of difference between the robot's initial distance from the goal and its distance after moving, with the goal whose distance decreased the most being seen as the most likely goal. Finally, the last method used to calculate the probabilities measured the probability for any given goal as a weighted sum of a) the degree to which the robot's angle matched the angle necessary for the robot to be looking directly at that goal, and b) how close the robot was to that goal. Note that for all three of the above-described methods, the prior probabilities for each destination also factored into the new probability calculated at each time step, so if the system were convinced the robot was going after, say, destination 1 for most of the system's run, the robot suddenly making a movement which most strongly corresponds to destination 2 wouldn't necessarily cause the system to abandon its prior belief that destination 1 was the robot's target destination. Exactly how quickly the system would shift its beliefs regarding the robot's chosen destination varies depending on a number of factors, such as how long the robot had been moving towards one destination before switching, the locations of each possible destination, and more.

After testing each of the three methods described above, it was determined that the A* method correctly predicted the robot's actual chosen goal the fastest, the maximum distance difference method ultimately made the correct prediction but took longer to do so than the A* method, and the orientation/distance method was almost completely ineffective at predicting the targeted goal. However, the A* method also had a much longer execution time than any of the other methods. Each individual execution of the A* algorithm on its own took a non-insignificant amount of time to compute, and since it not only had to be computed once for each goal, but also possibly re-computed every time to robot strays too far from each of the generated optimal paths, a full run of the system using the A* method was quite slow compared to the other two methods.

As mentioned, this example problem can be expanded into many other, more specific problems. A wide variety of different environments can be used, along with a wide variety of different kinds of cost functions for different kinds of terrain. While the specific example I used featured exactly four goals, it would be relatively easy to write an example with an arbitrarily large number of goals. One specific application that could be generated using my code as a base would be a parking lot scenario, with the possible goals being the various parking spaces and the observed agent being a vehicle which is trying to park in one of them. This problem could also be expanded to include additional functionality, such as by adding multiple robots each with their own goal, or by adding the possibility that the observed agent's true goal is not any of the known possible goals.

As a final aside, note that when writing the code for this example I used a scheme implementation of the A* algorithm I found online, with some minor alterations made by myself to suit my purposes [24].

5.1.3 Example 3 - Intersection

The final example problem I addressed was a relatively simple one which nonetheless featured both continuous and discrete probability distributions and a fairly complicated program describing the agent's behaviour.

Here, the observed agent is a car approaching a 4-way intersection. Another car, controlled by the system, is also approaching the intersection at the same time, and the two cars will crash if they continue at the same pace at which they are currently moving. See Figure 5.1 for a diagram showing the layout of the intersection and the locations of every relevant party.

The observed agent, in this case, has three possible goal destinations *and* three possible strategies it may follow to get to those destinations. Specifically the three goal destinations are for the observed agent to turn right, turn left, and to travel straight ahead, while the three strategies are for the observed agent to speed up so as to pass through the intersection before the user car does, to slow down so as to allow the user car to pass through the intersection first, and to continue at the same speed at which it is already travelling (representing the scenario wherein the driver of the other car has not yet noticed the impending collision). Each goal and strategy has its own probability of occurring, and the goal of the system is to predict both the other car's goal and the other car's strategy (a total of nine different possible behaviours) so that the user can decide how to best respond to the situation (effectively, this is predicting both the goal and part of the plan to achieve it). There are various factors about the world that the system can observe to better determine which goal/strategy is the most likely, such as the possible presence of a pedestrian crossing the road, the possible presence of a stop sign, and the possible presence of traffic lights and their state. Each has a different probability of occurring in each of the nine possible scenarios and therefore learning this information about the world will impact the probability distributions across the nine scenarios. Meanwhile, the other car also has to make a choice of whether or not it wants to be the right lane or the left lane. Both turns can occur in either lane, but left turns are more likely in the left lane and vice versa, so the other car's choice of lane impacts the probability distribution over the three goal destinations. As an aside, note that for the sake of simplicity, the system does not currently model the need to slow down to make a turn. Finally, outside of changing lanes, the observed car also has the ability to either accelerate or decelerate as it approaches the intersection. Each of three strategies has an acceleration value it expects to see (for example the "continue" strategy expects an acceleration of 0.0), so the acceleration that the system observes will impact the probability distributions for the three strategies. For the sake of simplicity, there is no noise in the observed agent's actions or my observations, though this is one way in which this example could theoretically be expanded.

To help illustrate how this example works, here is a sample of the code used to define the observed agent's behaviour:

```
(:begin
```

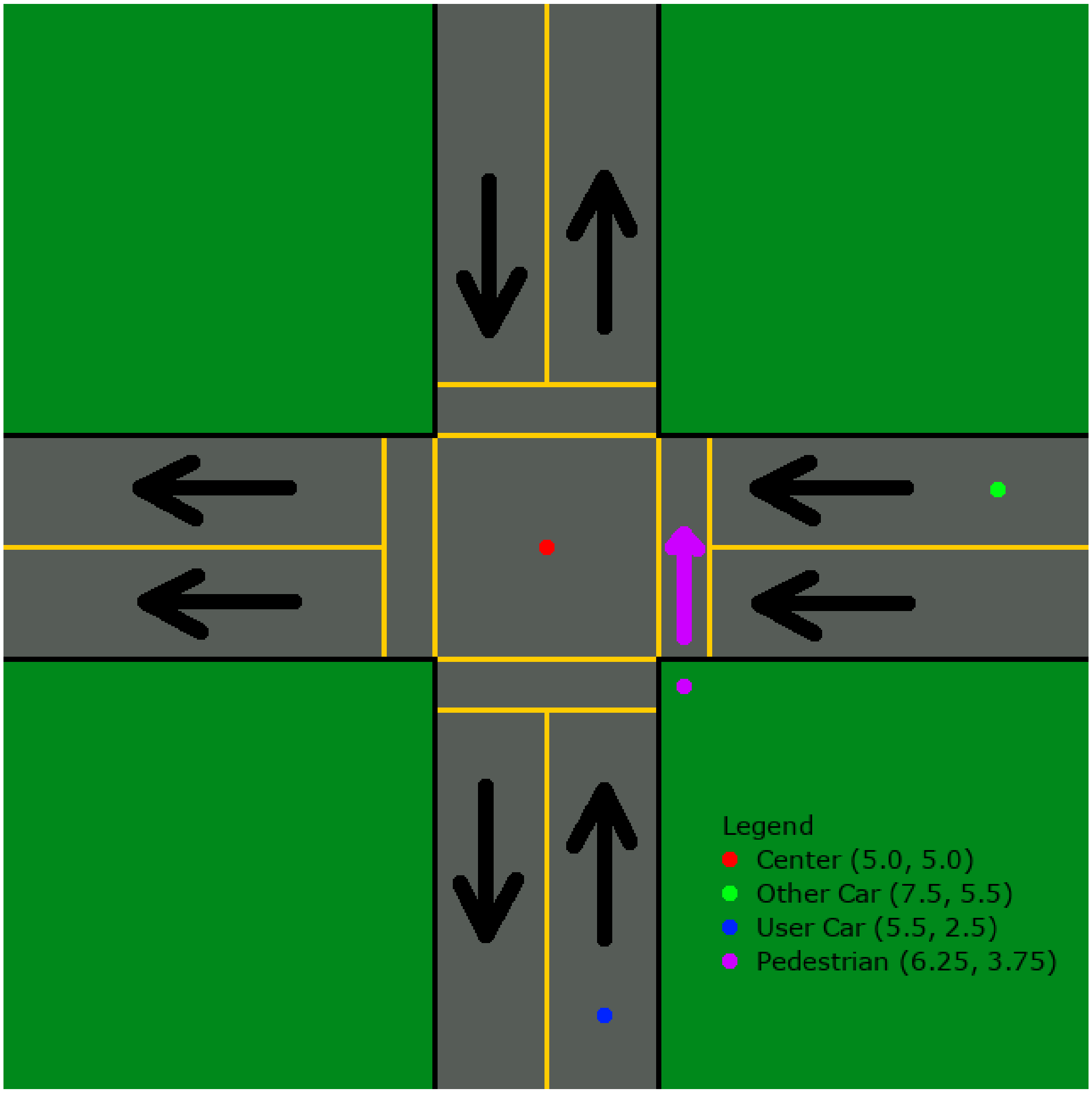


Figure 5.1: Diagram of the Intersection Example

```
(:act obsChooseGoal!) ;; Will the other car be turning left, right, or
    travelling straight?
(:act obsChooseTactic!) ;; Will the other car be speeding up, slowing down,
     or continuing at the speed limit?
(:act (obsPedestrian! pedestrianAnswer)) ;; Is there a pedestrian?
(:act (obsStopSign! stopSignAnswer)) ;; Is there a stop sign?
(:act (obsTrafficLight! trafficLightAnswer)) ;; Is there a traffic light?
(:when (eq? trafficLightAnswer 'yes)
   (:act (obsTrafficLightColour! trafficLightColourAnswer))) ;; What is the
        traffic light's colour?
(:act (obsAccelerate! accelAmount))
(:act obsChooseLane!) ;; Will the car be changing lanes?
(:act (obsAccelerate! accelAmount))
(:if (eq? decisionLane 'left)
   (:begin
      (:act obsLeftTurnSignal!)
      (:act (obsShift! shiftAmount))
      (:act obsLeftTurnSignal!))
   (:act (obsShift! shiftAmount)))
(:until (<= (car otherCar-pos) 6.0) ;; Repeatedly accelerate until the
    intersection is reached.
   (:act (obsAccelerate! accelAmount)))
(:if (eq? decisionGoal 'turnRight)
   (:act obsRightTurnSignal!)
   (:when (eq? decisionGoal 'turnLeft)
      (:act obsLeftTurnSignal!)))
(:if (eq? decisionGoal 'turnRight) ;; Turn until the intersection is passed
   (:until (>= (cdr otherCar-pos) 6.0)
      (:act obsTurn!))
   (if (eq? decisionGoal 'turnLeft)
      (:until (<= (cdr otherCar-pos) 4.0)
         (:act obsTurn!))
      (:until (<= (car otherCar-pos) 4.0)
         (:act obsTurn!)))))
```

5.2 Experimental Evaluation

In order to test the behaviour of my system, I chose to measure how my system's running time scales as various parameters change. I also measured how accurate my system's predictions were and how that accuracy varied as the number of states changed. I also adapted some of the experiments conducted by Kaminka et al. so as to compare how my system performs to theirs.

5.2.1 Computational Performance

5.2.1.1 Testing Protocol

In order to test the running time of my system, I used a modified version of the Robot with Multiple Goals example which features randomly-generated environments. These environments are generated according to some simple rules. First, it should be noted that each environment is represented as a continuous 2D space with a square-shaped border of length 10, with the lower-left corner of the world being at (0, 0) (though the system doesn't limit the robot to only exploring within those borders). The discretization done by the A* star algorithm in the calculation of the optimal path makes each cell a size of 0.2, resulting in a total of 2500 cells, with any cell more than half-containing an obstacle being considered an "obstacle" cell. The obstacles themselves are generated as follows. The obstacles are represented as circles, and are given a randomly-chosen centre point along with a randomly-chosen radius. The centre points are constrained to be within the 10x10 border (though the rest of the circle can extend outside of it), while the radii are constrained to be no longer than 2.5. Once the obstacles are generated, the system automatically generates some number of goals. Goals are represented as being single points in 2D space, and like obstacles centres, are constrained to be within the 10x10 box. They also cannot overlap with obstacles. After generating the randomized environment, the system would then automatically generate an action sequence using one of three simulated robots – a robot which only computes random actions (up to a maximum of 100), a robot which follows the optimal path to its chosen goal (as calculated by the A* algorithm), and a robot which follows the optimal path with some degree of noise in its movements (specifically, the robot was set to always turn at an angle within a 20° range, centred on the angle the optimal robot would follow). Theoretically, further robots could derived from the optimal robot with noise simply by changing the amount of noise present in the robot's turn angle. After an action sequence is generated, the system uses that action sequence in its testing, trying to predict what the robot's chosen goal is based on the movements generated by the simulated robot. For the sake of simplicity, I have also chosen to run my first set of tests without re-computing the optimal path, as re-computing the path has a major affect on the running time of the system. Later tests did re-compute the path, however, to see just how significantly doing so impacts the running time.

The parameters I chose to vary were the number of goals NG, the number of obstacles NO, and the number of states NS. How the running time scales with some of these parameters can already be shown theoretically using the following (roughly-described) formula:

$$TR = m * NG * E + u * NS + NG * b * NS$$

TR is the total running time of the system. The full formula is split into three sections: pre-processing, belief query, and belief update. The pre-processing running time is the time taken by the system to compute optimal paths using the A* algorithm (and also to generate the environment, though that had a negligible impact on the overall time), the belief update running time is the time taken by the system to update the values of each fluent after the system receives a new action, and the belief query running time is the time taken by the system to display its current beliefs in the values of the fluents. In the pre-processing section, m is the number of times the path is calculated per goal and E is the number of edges in the graph used by the A* algorithm to calculate the optimal path. For the first handful of tests, $m = 1$ (and note that this parameter cannot be directly controlled when path re-calculation is active). Meanwhile, E and NG are constants (E could theoretically not be a constant if I chose to vary the size of the environment in my tests). Ultimately, this portion of the formula is according to the running time of the A* algorithm, which is linear in the number of edges and vertices in the graph being used provided that the heuristic function h satisfies the condition that $|h(x) - h^*(x)| = O(log\ h^*(x))$ [32]. The heuristic function I use satisfies this condition. It is as follows:

$$h(x,y) = \begin{cases} |x - GX| * \sqrt{2} + ||x - GX| - |y - GY|| & |x - GX| < |y - GY| \\ |y - GY| * \sqrt{2} + ||x - GX| - |y - GY|| & |x - GX| \geq |y - GY| \end{cases}$$

Where GX and GY are the x and y coordinates of the goal node. Note that the A* calculations are done independently of any Monte Carlo sampling – in other words, the A* algorithm is only run once per goal (and once per re-calculation), not once per state, and thus the running time of any work done per state can be separated from the running time of the A* algorithm in the running time formula. Next, in the belief update section, u is the amount of work done to update a single state. Finally, in the belief query section, b is the amount of work done per state and per goal; NG has to be used again here since every belief query outputs a probability value for every goal. From the formula, it can be seen that the running time of the system should increase linearly as the number of goals increases and as the number of states increases. However, there is no clearly-defined formula describing how NO would impact the running time of the system. Intuitively, it seems likely that more obstacles should increase the running time of the system, as more obstacles will result in more complicated generated paths. Thus, the hypothesis that I hope to prove is that the running time of the system will increase linearly as the number of obstacles increases. I also hope to confirm the above-shown relationship between the running time and the number of goals/number of states.

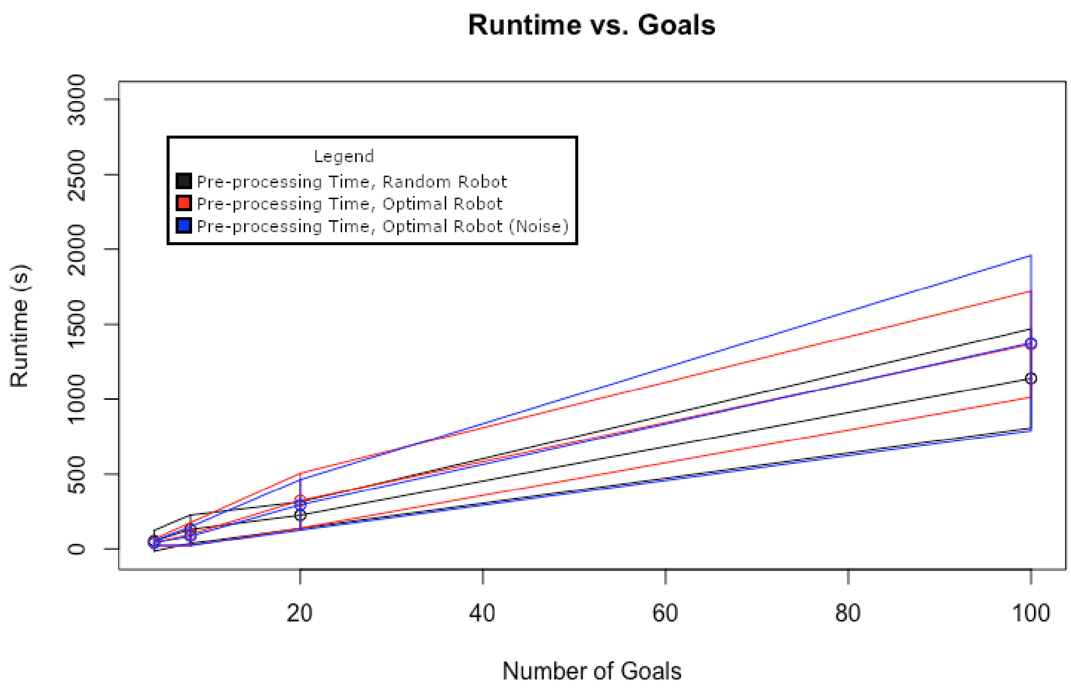
As a brief aside, note the each robot got it its own distinct set of randomly-generated environments; also note this had no impact on how each robot's performance varied with each parameter.

I ultimately did twenty runs of the system for each of the three robots and for each value of each parameter being tested, calculating the average running time across those twenty runs. I chose to do twenty because, through experimentation, I discovered that it was roughly after twenty runs that the average running time levelled out; additional tests had little impact on the average running time. I did four sets of tests on NG, using 4, 8, 10 and 20 goals, and I did the same for NO. For NS, I did tests with 100 states, 1000 states, 10000 states, 100000 and 1000000 states. The default values used for each parameter when it was not being tested was 4 goals, 4 obstacles, and 1000000 states. Finally, in addition to including the average pre-processing, belief update, and belief query times, I also included the total average running time of each set of twenty tests (which includes the pre-processing time in addition to the total belief update and belief query times).

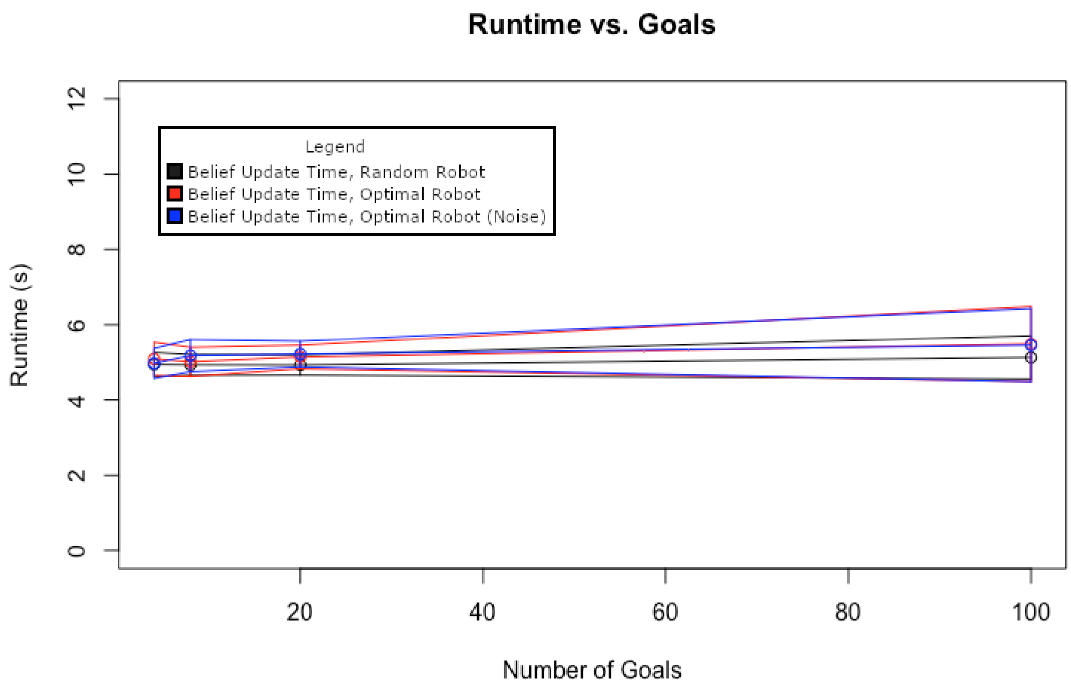
5.2.1.2 Results

The results of my testing are summarized in Tables 5.1-5.9 and Figures 5.2-5.7. "Avg PreProcessing Time" refers to the average amount of time taken to perform pre-processing (per test), "Avg Update Time" refers to the average amount of time taken to perform belief updates (per update), "Avg Query Time" refers to the average amount of time taken to perform belief queries (per query), and "Total Avg Time" refers to the overall average amount of time taken by the system to perform a single test. Note again that the total number of belief updates and queries depends on the number of observed actions, which cannot be easily controlled; overall, there should be just as many belief updates as there are belief state queries. In addition to calculating the average time, I have also calculated the standard deviation for each collection of tests run under a given robot and a given number of goals/obstacles/states. In the graphs, the standard deviation is noted using a bar with three lines; the middle line is the mean, and the top and bottom lines are the mean plus-or-minus the standard deviation. The standard deviation values given in the tables have been rounded to the nearest integer number.

Ultimately, the trends seen in the results are largely as expected. Increasing the number of goals in particular increases the pre-processing, belief query, and total times at a roughly linear rate (the average belief update time is unaffected since belief updates do not do work for every goal). Increasing the number

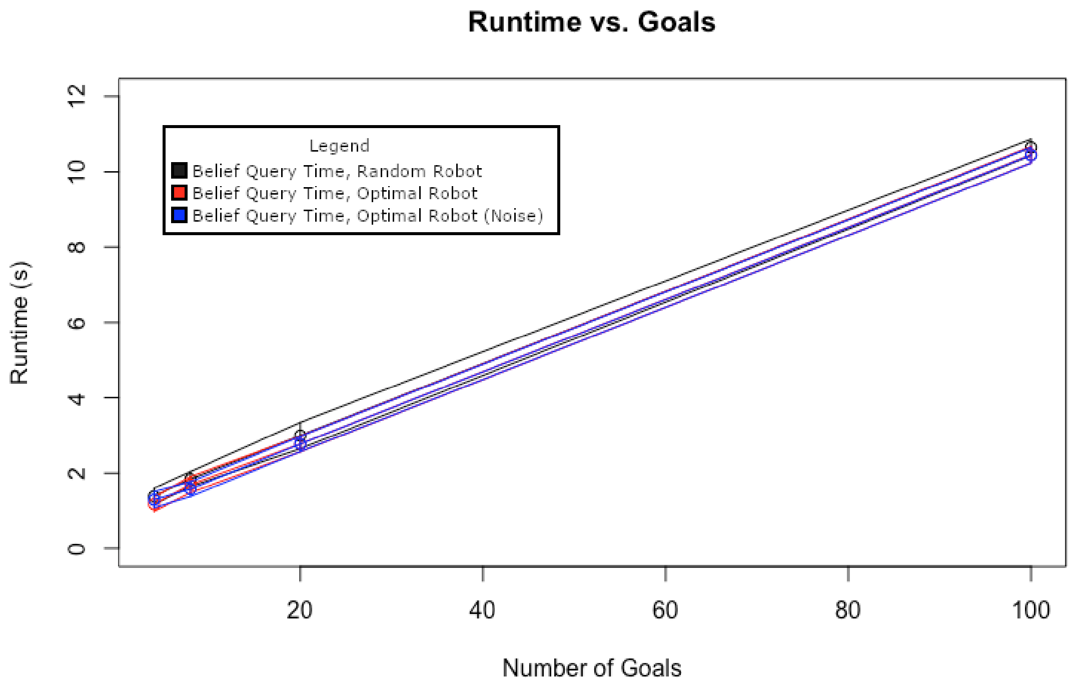


(a) Effect of Number of Goals on Preprocess Running Time

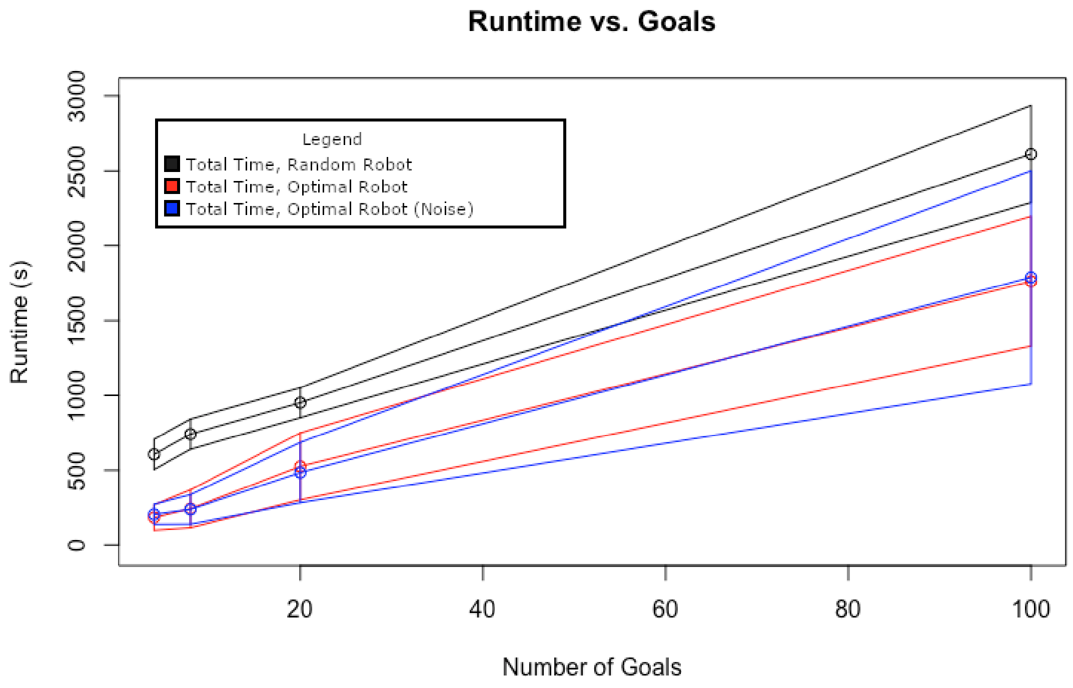


(b) Effect of Number of Goals on Belief Update Running Time

Figure 5.2: Effects of Number of Goals, Part 1

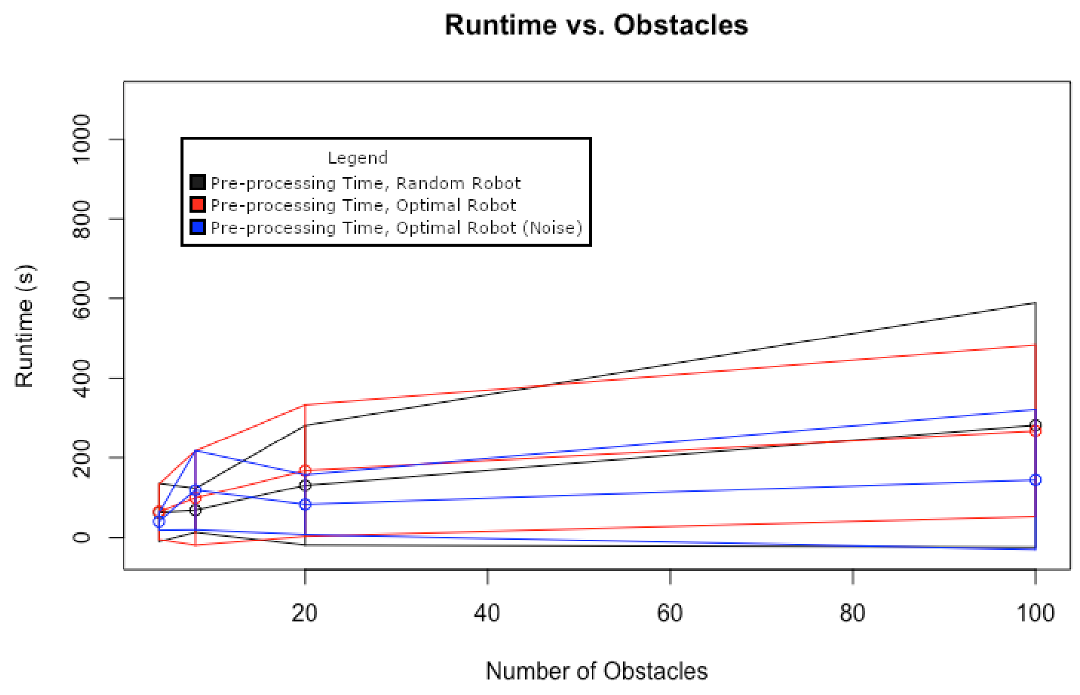


(a) Effect of Number of Goals on Belief Query Running Time

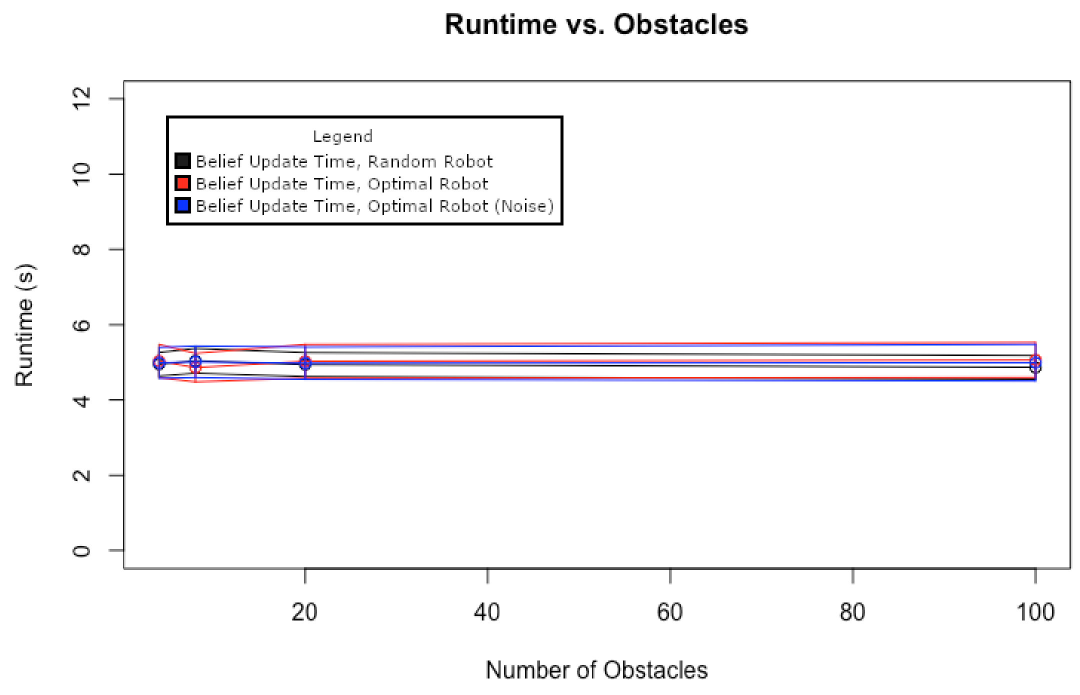


(b) Effect of Number of Goals on Total Running Time

Figure 5.3: Effects of Number of Goals, Part 2

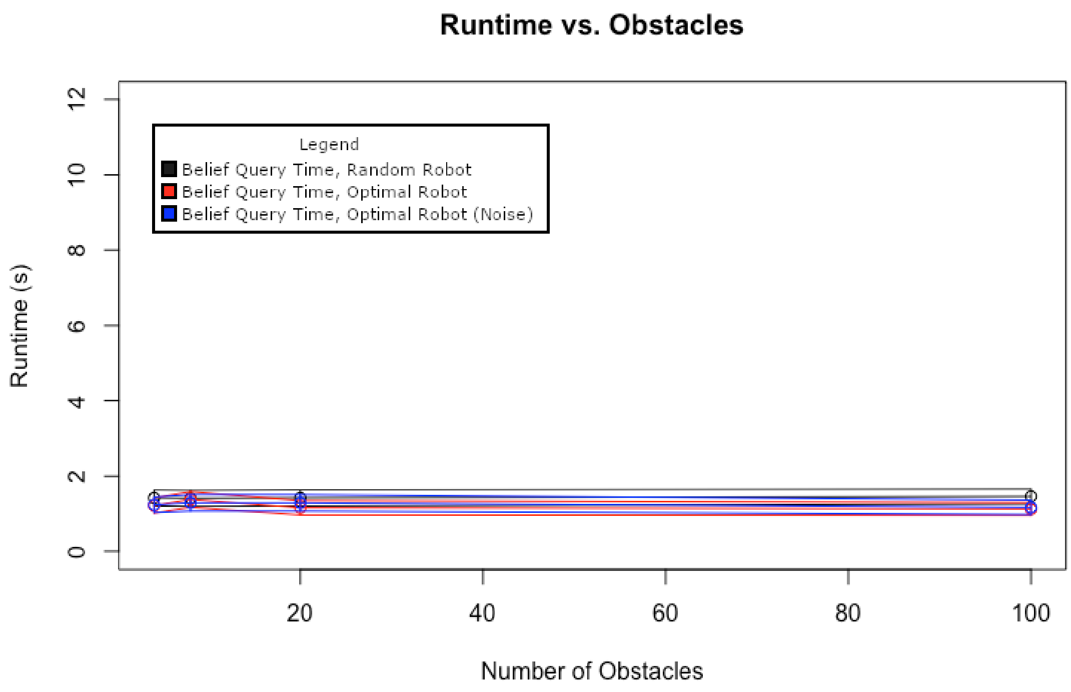


(a) Effect of Number of Obstacles on Preprocess Running Time

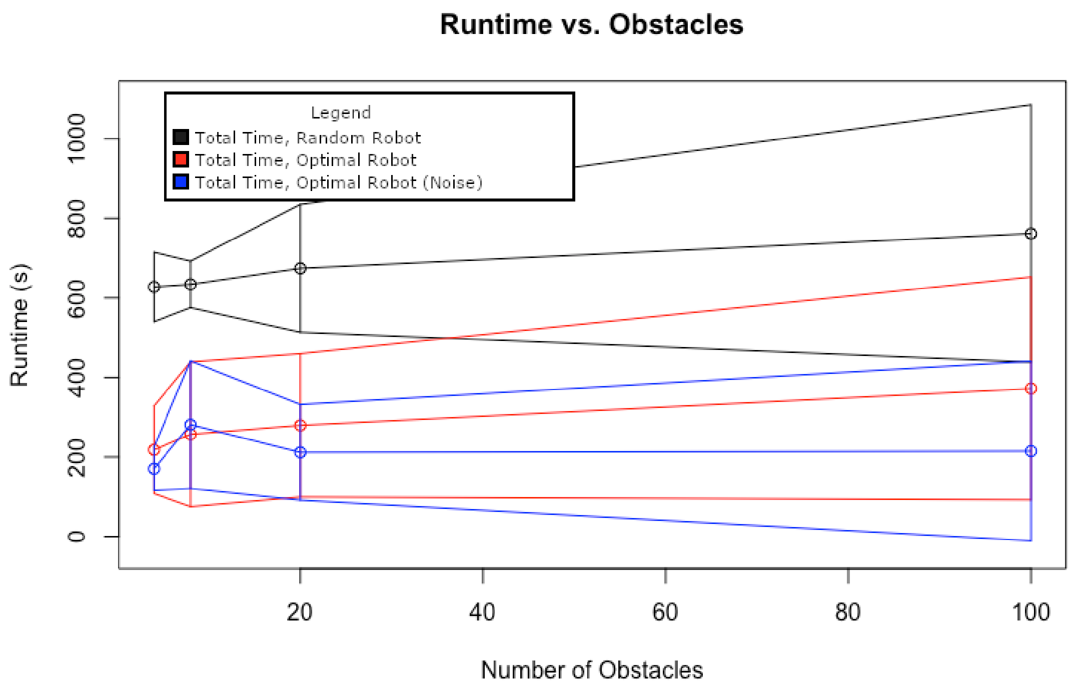


(b) Effect of Number of Obstacles on Belief Update Running Time

Figure 5.4: Effects of Number of Obstacles, Part 1

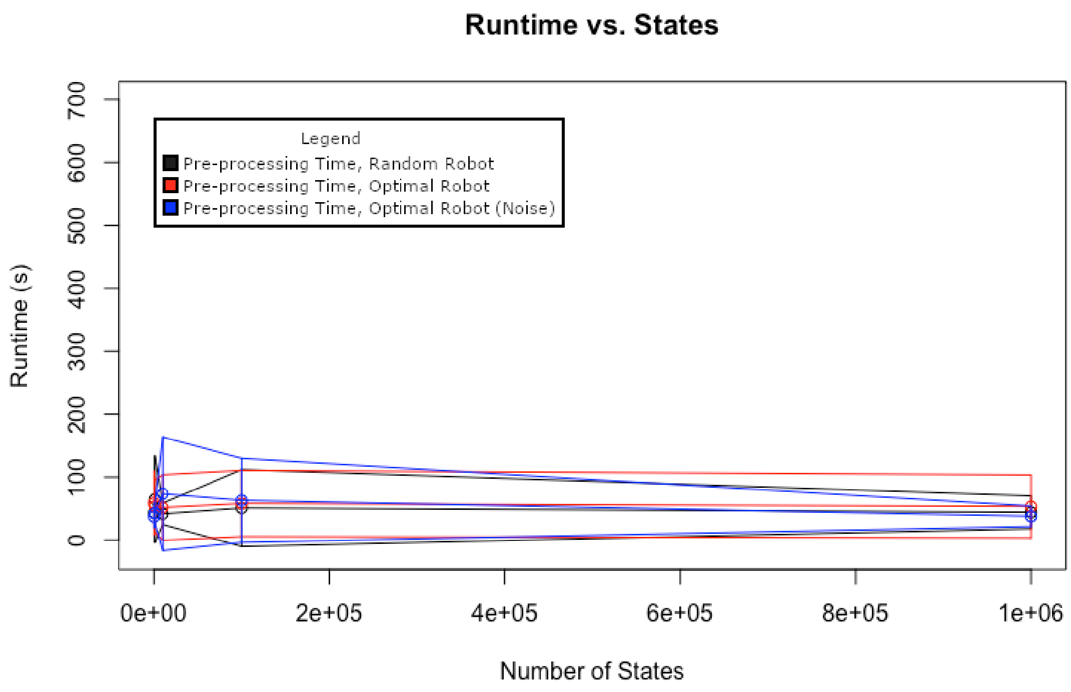


(a) Effect of Number of Obstacles on Belief Query Running Time

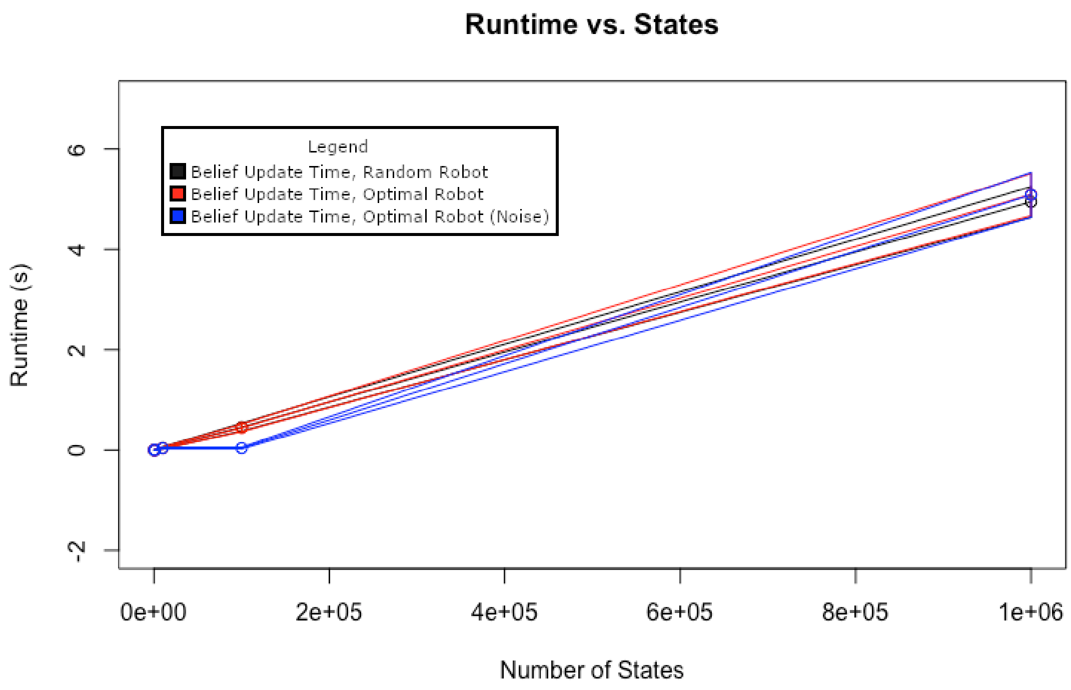


(b) Effect of Number of Obstacles on Total Running Time

Figure 5.5: Effects of Number of Obstacles, Part 2

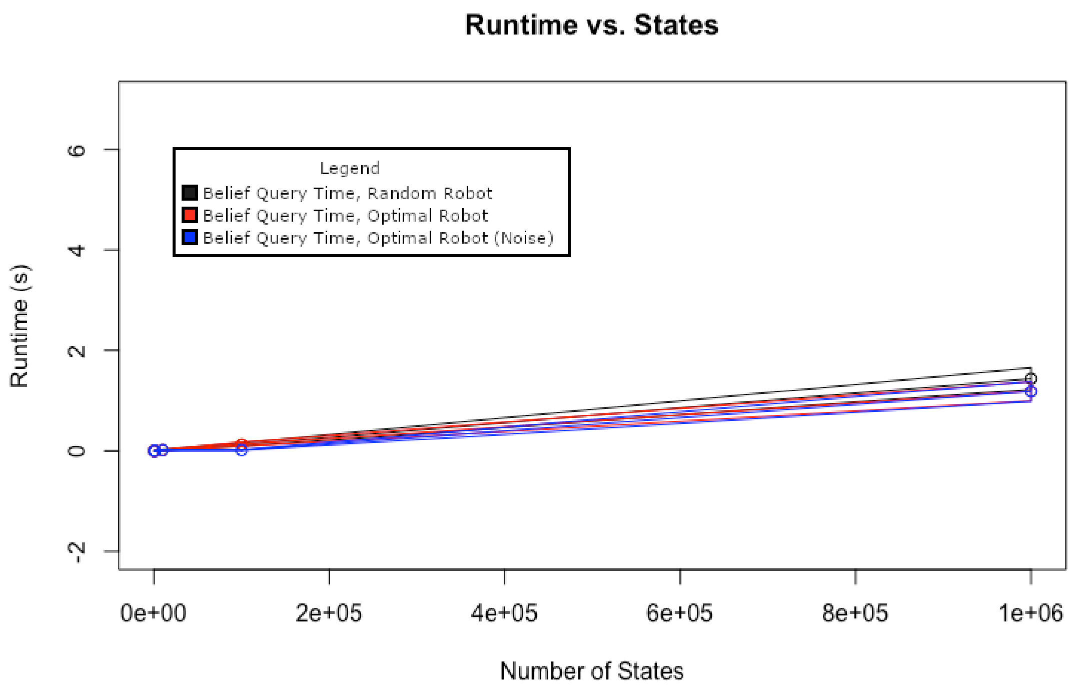


(a) Effect of Number of States on Preprocess Running Time

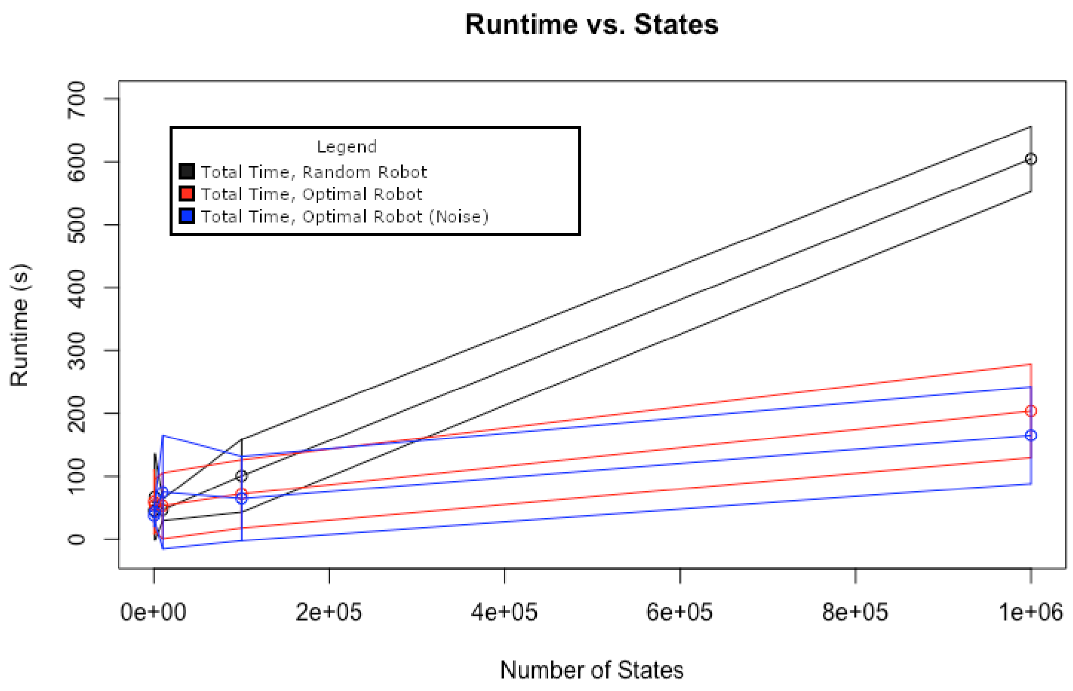


(b) Effect of Number of States on Belief Update Running Time

Figure 5.6: Effects of Number of States, Part 1



(a) Effect of Number of States on Belief Query Running Time



(b) Effect of Number of States on Total Running Time

Figure 5.7: Effects of Number of States, Part 2

Table 5.1: Effect of Goals on Running Time (Random Robot)

# Goals	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
4 Goals	$54906ms \pm 70284ms$	$4945ms \pm 317ms$	$1384ms \pm 215ms$	$605933ms \pm 101275ms$
8 Goals	$131656ms \pm 95104ms$	$4935ms \pm 276ms$	$1842ms \pm 198ms$	$740984ms \pm 1000780ms$
20 Goals	$225682ms \pm 88111ms$	$4931ms \pm 266ms$	$2995ms \pm 349ms$	$950939ms \pm 100036ms$
100 Goals	$1139149ms \pm 324212ms$	$5129ms \pm 574ms$	$10647ms \pm 216ms$	$2611157ms \pm 324212ms$

Table 5.2: Effect of Goals on Running Time (Optimal Robot)

# Goals	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
4 Goals	$42995ms \pm 23597ms$	$5094ms \pm 436ms$	$1171ms \pm 190ms$	$184518ms \pm 84480ms$
8 Goals	$98873ms \pm 772525ms$	$5014ms \pm 384ms$	$1694ms \pm 214ms$	$243151ms \pm 126379ms$
20 Goals	$322528ms \pm 182989ms$	$5142ms \pm 312ms$	$2772ms \pm 207ms$	$525547ms \pm 223196ms$
100 Goals	$1365873ms \pm 354183ms$	$5487ms \pm 998ms$	$10452ms \pm 218ms$	$1762076ms \pm 433389ms$

Table 5.3: Effect of Goals on Running Time (Optimal Robot (± 10 Degrees Noise))

# Goals	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
4 Goals	$39454ms \pm 15979ms$	$4973ms \pm 396ms$	$1292ms \pm 214ms$	$205855ms \pm 66731ms$
8 Goals	$86673ms \pm 62684ms$	$5179ms \pm 425ms$	$1574ms \pm 197ms$	$238870ms \pm 98507ms$
20 Goals	$294631ms \pm 167953ms$	$5221ms \pm 341ms$	$2771ms \pm 202ms$	$484439ms \pm 201342ms$
100 Goals	$1374230ms \pm 584950ms$	$5455ms \pm 970ms$	$10439ms \pm 208ms$	$1788383ms \pm 711617ms$

Table 5.4: Effect of Obstacles on Running Time (Random Robot)

# Obstacles	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
4 Obstacles	$62935ms \pm 73173ms$	$4950ms \pm 309ms$	$1417ms \pm 214ms$	$627510ms \pm 87173ms$
8 Obstacles	$68351ms \pm 55162ms$	$5040ms \pm 326ms$	$1407ms \pm 214ms$	$633902ms \pm 58469ms$
20 Obstacles	$130782ms \pm 150388ms$	$4941ms \pm 314ms$	$1418ms \pm 216ms$	$674244ms \pm 160428ms$
100 Obstacles	$282077ms \pm 307378ms$	$4863ms \pm 313ms$	$1458ms \pm 201ms$	$761602ms \pm 323504ms$

Table 5.5: Effect of Obstacles on Running Time (Optimal Robot)

# Obstacles	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
4 Obstacles	$64689ms \pm 69847ms$	$5034ms \pm 437ms$	$1216ms \pm 199ms$	$218131ms \pm 110123ms$
8 Obstacles	$98708ms \pm 118673ms$	$4862ms \pm 376ms$	$1375ms \pm 207ms$	$256966ms \pm 181959ms$
20 Obstacles	$167943ms \pm 163768ms$	$5028ms \pm 444ms$	$1156ms \pm 189ms$	$279502ms \pm 180089ms$
100 Obstacles	$267739ms \pm 215294ms$	$5063ms \pm 461ms$	$1121ms \pm 167ms$	$372140ms \pm 280426ms$

Table 5.6: Effect of Obstacles on Running Time (Optimal Robot (± 10 Degrees Noise))

# Obstacles	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
4 Obstacles	$40276ms \pm 21804ms$	$4987ms \pm 407ms$	$1235ms \pm 204ms$	$170642ms \pm 54667ms$
8 Obstacles	$118886ms \pm 99237ms$	$5014ms \pm 413ms$	$1287ms \pm 217ms$	$281167ms \pm 160340ms$
20 Obstacles	$82592ms \pm 75015ms$	$4978ms \pm 425ms$	$1286ms \pm 216ms$	$211689ms \pm 120351ms$
100 Obstacles	$144933ms \pm 176264ms$	$4990ms \pm 484ms$	$1165ms \pm 191ms$	$214744ms \pm 225747ms$

Table 5.7: Effect of States on Running Time (Random Robot)

# Obstacles	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
100 States	$43476ms \pm 11417ms$	$0ms \pm 0ms$	$0ms \pm 1ms$	$345514ms \pm 11393ms$
1000 States	$64927ms \pm 68784ms$	$0ms \pm 1ms$	$1ms \pm 2ms$	$66995ms \pm 68672ms$
10000 States	$41628ms \pm 16807ms$	$43ms \pm 20ms$	$16ms \pm 13ms$	$46586ms \pm 16968ms$
100000 States	$50941ms \pm 61011ms$	$459ms \pm 78ms$	$129ms \pm 36ms$	$100660ms \pm 57776ms$
1000000 States	$43983ms \pm 26077ms$	$4947ms \pm 303ms$	$1443ms \pm 216ms$	$604468ms \pm 51743ms$

Table 5.8: Effect of States on Running Time (Optimal Robot)

# States	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
100 States	$58893ms \pm 50809ms$	$0ms \pm 0ms$	$0ms \pm 0ms$	$59497ms \pm 50889ms$
1000 States	$53840ms \pm 41668ms$	$0ms \pm 1ms$	$1ms \pm 2ms$	$54386ms \pm 41587ms$
10000 States	$51693ms \pm 52108ms$	$39ms \pm 18ms$	$15ms \pm 8ms$	$53002ms \pm 52247ms$
100000 States	$58076ms \pm 52608ms$	$444ms \pm 75ms$	$133ms \pm 43ms$	$71976ms \pm 54101ms$
1000000 States	$53131ms \pm 50165ms$	$5093ms \pm 422ms$	$1190ms \pm 192ms$	$203878ms \pm 74396ms$

Table 5.9: Effect of States on Running Time (Optimal Robot (± 10 Degrees Noise))

# States	Avg Pre-Processing Time	Avg Update Time	Avg Query Time	Total Avg Time
100 States	$36992ms \pm 17376ms$	$0ms \pm 0ms$	$0ms \pm 0ms$	$37507ms \pm 17444ms$
1000 States	$42708ms \pm 15516ms$	$0ms \pm 1ms$	$1ms \pm 0ms$	$43232ms \pm 15558ms$
10000 States	$73566ms \pm 90213ms$	$38ms \pm 15ms$	$16ms \pm 10ms$	$74803ms \pm 90383ms$
100000 States	$63074ms \pm 66721ms$	$39ms \pm 16ms$	$15ms \pm 6ms$	$64478ms \pm 66981ms$
1000000 States	$37549ms \pm 16080ms$	$5087ms \pm 447ms$	$1187ms \pm 196ms$	$165167ms \pm 77074ms$

of obstacles, meanwhile, also increased the pre-processing time at a roughly linear rate, albeit not to the same degree as with the goals. This is understandable, as while changing the number of obstacles has less of a direct impact on the running time, it does nonetheless produce more complex paths while requiring more time to generate the environment. Notably, altering the number of obstacles had no impact on the average belief update and belief query time. The belief query time was also consistently lower than the belief update time for small numbers of goals, most likely because the process of simply querying the values of the fluents is less complex than updating them. It's worth noting how the belief update and belief query times were only affected by the number of states and the number of goals, and even then the number of goals only affected the belief update time in a minor way. Furthermore, the average time for both belief updates and belief queries remained consistently low, maxing out at roughly 12 seconds per belief query for 100 goals and 5 seconds per belief update for 1 million states. These running times are not terribly high, but may need to be improved if the system is to be run in real-time (though the large running time is partly down to the complexity of the problem). It's also worth noting that the different agent behaviours had very little impact on the results; the only notable difference is that the random robot took a significantly larger amount of total time than the other two due to it taking significantly longer for it to reach its target. In most cases the difference in total running time between the random robot and the other two robots was constant, but its worth noting that that this difference actually scaled in the case of states, being almost nonexistent for small quantities of states and rather large for large quantities of states; this is most likely a combination of the belief query and belief update times scaling directly as the number of states scales (note that the longer it takes for the robot to reach its target destination, the more belief queries and belief updates occur) and also the fact that I tested much larger quantities of states compared to goals and obstacles (100-1000000 states vs. 4-100 goals/obstacles).

Regarding the standard deviation, it is admittedly the case that the standard deviation is relatively high when calculating the average pre-processing time. Examining the output of each test more carefully revealed that the pre-processing time was abnormally high in a few specific cases. What seems to be the most likely explanation for this is that it occurred during the generation of the random environment. The random environment generator was built to discard and regenerate an obstacle if it overlapped with a goal, since the idea was that obstacles were impassable – perhaps there were a few runs wherein this wound up happening more often than usual. The fact that the standard deviation is highest in the tests wherein I increased the number of obstacles (and thereby increased the frequency with which a generated obstacle would have to be discarded), and second-highest in the tests wherein I increased the number of goals (thereby increasing the chances of a generated obstacle overlapping a goal) heavily supports this theory.

5.2.2 Accuracy Tests

For my accuracy experiments, I used the target shooting example defined in Chapter 4 and fed it a simple action sequence wherein the observed agent chooses a plan and a target, aims at its chosen target, and fires. Prior to the agent firing a shot, I made it so that my system would observe it aiming at a $45°$ angle. I also made it such that the action used to represent the agent aiming its shot was noisy as I felt that would produce more interesting results. I then measured the system's beliefs in: a) which target the agent was intending to aim at and b) which target the agent was actually aiming at (which might not be the same target given the use of noisy aiming) after the agent fires its shot but before said shot actually hits a target.

5.2.2.1 Testing Protocol

After measuring my systems beliefs, I calculated what those beliefs *should* be at that point in time, and compared these "correct" values to my system's outputs to see if there were any discrepancies. Since the system is a simple one where all the probability distributions involved are known quantities, calculating these correct beliefs was a relatively simple process, and comparing this pre-calculated correct beliefs to the system's actual beliefs is more valuable than comparing how quickly the system can correctly predict the observed agent's plan because that is largely dependant on the quality of the sensors used to gather data and on the quality of the model used to formulate these predictions, which my system cannot directly control (essentially, the onus is on the person using my system to ensure that the model/sensors they are using are

Table 5.10: Degree of Belief that the Agent is Intending to Aim at Target 1

# States	Correct Belief	My System's Belief	Difference
1 State	37.75%	$35\% \pm 14.53\%$	-2.75%
10 States	37.75%	$44.10\% \pm 6.222\%$	$+6.346\%$
100 State	37.75%	$39.53\% \pm 0.3943\%$	$+1.774\%$
1000 States	37.75%	$38.61\% \pm 0.7857\%$	$+0.8537\%$
10000 States	37.75%	$37.58\% \pm 0.2810\%$	-0.1747%
100000 States	37.75%	$37.64\% \pm 0.008301\%$	-0.1176%
1000000 States	37.75%	$37.74\% \pm 0.01331\%$	-0.009855%

Table 5.11: Degree of Belief that the Agent is Intending to Aim at Target 2

# States	Correct Belief	My System's Belief	Difference
1 State	62.24%	$20\% \pm 40\%$	-42.25%
10 States	62.24%	$55.88\% \pm 18.60\%$	-6.364%
100 State	62.24%	$60.45\% \pm 9.161\%$	-1.800%
1000 States	62.24%	$61.36\% \pm 2.180\%$	-0.8844%
10000 States	62.24%	$62.39\% \pm 0.5956\%$	$+0.1466\%$
100000 States	62.24%	$62.34\% \pm 0.2208\%$	$+0.08932\%$
1000000 States	62.24%	$62.23\% \pm 0.06504\%$	-0.01835%

Table 5.12: Degree of Belief that the Agent is Intending to Aim at Target 3

# States	Correct Belief	My System's Belief	Difference
1 State	$2.619e-7\%$	$45\% \pm 49.75\%$	$+45.00\%$
10 States	$2.619e-7\%$	$0.01889\% \pm 0.02163\%$	$+0.01889\%$
100 State	$2.619e-7\%$	$0.02524\% \pm 0.01307\%$	$+0.02524\%$
1000 States	$2.619e-7\%$	$0.03073\% \pm 0.007416\%$	$+0.03073\%$
10000 States	$2.619e-7\%$	$0.02808\% \pm 0.001853\%$	$+0.02808\%$
100000 States	$2.619e-7\%$	$0.02825\% \pm 0.0005711\%$	$+0.02825\%$
1000000 States	$2.619e-7\%$	$0.02821\% \pm 0.0001742\%$	$+0.02821\%$

Table 5.13: Degree of Belief that the Agent is Aiming at Target 1

# States	Correct Belief	My System's Belief	Difference
1 State	24.01%	$25\% \pm 43.30\%$	$+0.9934\%$
10 States	24.01%	$26.95\% \pm 17.41\%$	$+2.940\%$
100 State	24.01%	$25.49\% \pm 6.158\%$	$+1.480\%$
1000 States	24.01%	$24.35\% \pm 1.627\%$	$+0.3480\%$
10000 States	24.01%	$23.91\% \pm 0.3910\%$	-0.09274%
100000 States	24.01%	$23.97\% \pm 0.1704\%$	-0.03433%
1000000 States	24.01%	$24.00\% \pm 0.04686\%$	-0.01042%

Table 5.14: Degree of Belief that the Agent is Aiming at Target 2

# States	Correct Belief	My System's Belief	Difference
1 State	64.35%	$20\% \pm 40\%$	-44.35%
10 States	64.35%	$60.40\% \pm 18.92\%$	-3.943%
100 State	64.35%	$62.84\% \pm 8.703\%$	-1.511%
1000 States	64.35%	$63.70\% \pm 2.043\%$	-0.6483%
10000 States	64.35%	$64.53\% \pm 0.5341\%$	-0.1841%
100000 States	64.35%	$64.41\% \pm 0.2327\%$	$+0.05932\%$
1000000 States	64.35%	$64.32\% \pm 0.06104\%$	0.02420%

Table 5.15: Degree of Belief that the Agent is Aiming at Target 3

# States	Correct Belief	My System's Belief	Difference
1 State	0.01659%	$45\% \pm 49.75\%$	+44.98%
10 States	0.01659%	$0.01889\% \pm 0.02163\%$	+0.002300%
100 State	0.01659%	$0.01571\% \pm 0.006210\%$	−0.0008755%
1000 States	0.01659%	$0.01676\% \pm 0.001576\%$	+0.0001686%
10000 States	0.01659%	$0.01644\% \pm 0.0006767\%$	−0.0001450%
100000 States	0.01659%	$0.01657\% \pm 0.0002113\%$	$-1.868e-5\%$
1000000 States	0.01659%	$0.01659\% \pm 4.139e-5\%$	$+5.063e-6\%$

ones which can produce accurate predictions). So instead, I measured on system's accuracy by determining how many states are necessary for Monte Carlo Sampling to produce the "correct" predictions for a given model/set of sensors. The way these "correct" predictions were calculated was as follows:

To calculate the likelihood that the observed agent was intending to aim at a given target (given the 45° observation), I first calculated the probability of the system observing the agent aiming at a 45° angle in the scenario where the agent was intending to aim at the first target (call this $P(45^\circ\ Observed \mid Intending\ to\ aim\ at\ target\ 1))$ by integrating over the product of the gaussian random variable representing the probability of the agent's aim being some value given that they were intending to aim at the first target and the gaussian random variable representing the probability of observing the agent aiming at a 45° angle given the agent's actual aim (with the agent's actual aim being the variable being integrated over in a range spanning the entire field). I then did the same for the other two targets, then calculated the general probability of observing an aim of 45° as $P(45^\circ\ Observed \mid Intending\ to\ aim\ at\ target\ 1) * 1/3 + P(45^\circ\ Observed \mid Intending\ to\ aim\ at\ target\ 2) * 1/3 + P(45^\circ\ Observed \mid Intending\ to\ aim\ at\ target\ 3) * 1/3$ (since the target the agent intends to aim at is directly determined by which plan its following and all three plans are equally likely); this gives us $P(45^\circ\ Observed)$. Finally, I calculated the likelihood that the observed agent was intending to aim at the first target as $(P(45^\circ\ Observed \mid Intending\ to\ aim\ at\ target\ 1)\ /\ P(45^\circ\ Observed)) * 1/3)$ as per Bayesian rules, then did the same for the other two targets. To calculate the likelihood of the agent actually aiming at a given target given the 45° observation, I next calculated the probability of both the system observing the agent aiming at a 45° angle and the agent aiming at the first target in the scenario where the agent was intending to aim at the first target (call this $P(45^\circ\ Observed\ \&\ Aiming\ at\ target\ 1 \mid Intending\ to\ aim\ at\ target\ 1))$ by integrating over the same product as before, but this time only over the range of angles at which the agent would be aiming at target 1. After performing the same calculation for the scenarios wherein the agent was intending to aim at targets 2 and 3, I calculated the general probability of the system observing a 45° angle while the agent was aiming at target 1 as $P(45° \textit{ Observed} \ \& \textit{ Aiming at target } 1 \mid \textit{Intending to aim at target } 1) * 1/3 + P(45° \textit{ Observed} \ \& \textit{ Aiming at target } 1 \mid \textit{Intending to aim at target } 2) * 1/3 + P(45° \textit{ Observed} \ \& \textit{ Aiming at target } 1 \mid \textit{Intending to aim at target } 3) * 1/3$; this gives us $P(45° \textit{ Observed} \ \& \textit{ Aiming at target } 1)$. Finally, I calculated the probability of the agent actually aiming at target 1 given the 45° observation as $P(45° \textit{ Observed} \ \& \textit{ Aiming at target } 1) \ / \ P(45° \textit{ Observed})$, then repeated the whole process for the other two targets.

Once again, I have included the standard deviation in my results. I measured my results using 1, 10, 100, 1000, 10000, 100000, and 1000000 states. Note that all of my results have been rounded to the nearest integer.

5.2.2.2 Results

The results I obtained are summarized in Tables 5.10-5.15. As before, I averaged the results over 20 runs of the system for each number of states. To keep the tables uncluttered, I split each query being evaluated. As you can see from the tables, it didn't take many states for my system to begin reporting percentages that were quite close to the correct values; 1000 states was consistently enough to produce results within less than 1% of the correct result, and the standard deviation of my data steadily decreased with time. While it is true that the number of samples needed to produce accurate data depends on the specific scenario, the fact that I was able to produce accurate results at such a significantly smaller number of sample states than I used in many of my other examples suggests that I can safely reduce the number of states used in modelling plan recognition problems in my system without significantly sacrificing accuracy, which will have benefits in regards to the running time of the system.

5.2.3 Additional Tests

As mentioned, I also adapted and ran some of the tests conducted by Kaminka et al. so as to better compare the results of my work to theirs. Specifically, I adapted the goal recognition algorithm they developed into my system, and ran it under similar conditions. Their algorithm, again as mentioned, was coincidentally very similar to the one I used in the example featuring the robot with multiple goals, though the method they used to compare the robot's path to the optimal path for a given potential goal was different from mine; while I calculated the "ideal" orientation for the robot at each time step (i.e. the orientation which would have the robot pointing directly at the next node in the optimal path) and then compared it to the robot's actual orientation, Kaminka et al. estimated the robot's complete path to the goal (using both the robot's current path and an optimal path from the robot's current position), then compared the length of this estimated path to the length of the optimal path from robot's original starting position. Additionally, Kaminka et al.'s method featured various heuristic strategies which could be used to alter the base algorithm to improve performance. The base version of their algorithm recomputed the optimal path for each goal at every time step, but they also formulated a version using a heuristic strategy called "RECOMPUTE" which would only recompute the path under certain conditions (specifically, whenever the goal with the highest probability of being the actual target goal is not the one whose path the robot is in closest proximity to); otherwise, the system instead estimated the complete path to each goal by removing a portion of the current estimated path (up to whichever node in the path would be the robot's next target, were it following the path exactly) and then connecting the path the robot has followed so far to this "shortened" version of the optimal path. The other heuristic strategy they formulated, called "PRUNE", was to prune certain potential goals (i.e. removed them from consideration) whenever they were deemed no longer likely to be the true goal, specifically by checking if the angle between the robot's current trajectory and the hypothetical trajectory which would lead it directly to the optimal path to be greater than some threshold angle whose value would be determined by how strict or lenient one wishes the pruning process to be. When evaluating their algorithm, they specifically used a threshold of $120°$. Provided that the actual target goal never gets pruned by mistake, pruning goals in this manner would ideally improve the efficiency of the algorithm, as goals that have been pruned would never need to have their optimal paths recomputed [31].

5.2.3.1 Test Domain.

For these tests, I once again used a randomly-generated environment. However, for these tests, I used one single randomly-generated environment across every run of the system, rather than changing it every time. This environment featured 11 possible goals and four obstacles.

5.2.3.2 Testing Protocol.

The experiment analysis I conducted on this version of the robot with multiple goals was similar to the analysis conducted by Kaminka et al, though I didn't gather statistics in quite the same way as they did, admittedly due to a misunderstanding on my part (i.e. I was trying to gather statistics in the same way they did, but I misunderstood their explanation of how they gathered statistics). This should not have significantly affected the trends, however. For each run of the system, I used one of the 11 possible goals as the starting point, with the actual target goal being randomly chosen from among the remaining 10 possible goals. I then did two runs of the system using each possible starting point (with a different random target goal for each run), and then generated four values, each averaged across the 22 runs of the system: the time to end (i.e. the number of actions remaining once the system accurately ranks the correct goal as the most likely goal and never changes that prediction again, as a percent of the total number of actions), the number of times the system correctly ranked the actual target goal as the most likely goal (again, as a percent of the total number of actions), the amount of time spent planning, and the number of planner calls. I then generated these averages across 5 different approaches to the algorithm based on different combinations of "RECOMPUTE" and "PRUNE": one in which the optimal paths to each goal are recomputed every time the robot moves and no goals are pruned, one in which the paths were only recomputed as in "RECOMPUTE", one in which goals were pruned as in "PRUNE", one in which both the "RECOMPUTE" and the "PRUNE" heuristic strategies were used, and one in which the paths were never recomputed under any circumstances (and no pruning was done). For the sake of simplicity, let's call these approaches "Baseline", "Recompute", "Prune", "Both", and "No Recompute" (which are the terms Kaminka et al. used), respectively.

Examining the data I got after running one batch of tests, I found that the data did not seem to be converging. In order to check this, I ran statistics under the same conditions a second time, then averaged the results from both the first set of statistics and the second. I then repeated this procedure multiple times until the data appeared to converge; this specifically happened after running statistics nine times, for a total of 198 samples for each approach (11 starting points * 2 runs to a random target goal * 9 total sets of statistics).

For clarity's sake, Kaminka et al.'s statistics were gather similarly, except rather than only doing two runs per starting point each with a randomly chosen target goal, they instead did two runs for each starting point AND potential target goal, resulting in a total of 220 runs per approach (11 starting points * 10 potential target goals * 2 runs per each starting point and target goal). They also did not need to run their experiments additional times to get the data to converge. So to summarize, the key difference between how they gathered statistics and how I gathered statistics were a) the number of data points per approach (220 in their case, 198 in mine), and b) the fact that Kaminka et al.'s data points very strictly consisted of two runs for every possible starting point-target goal pair, whereas mine consisted 18 runs for every starting point with a random target goal for each run (meaning that some starting point-target goal pairs may have had more than two runs while others may have had less).

Table 5.16: Comparison Test Results

	Planning Time	Planner Calls	Time to End	Correct Ranking
Baseline	3065789.93939	289.141414141	47.4033372358%	54.9834013582%
Recompute	2167598.87374	207.525252525	40.6477307982%	53.5540098302%
Prune	1844673.07576	178.621212121	49.1695010491%	57.5980983843%
Both	1275578.73737	123.752525253	42.2374053338%	55.5128665457%
No Recompute	136932.363636	10.0	22.6588122726%	37.1053543378%

5.2.3.3 Results.

The results I generated were largely as expected; they can be found in Table 5.16. Interestingly, the results don't quite match up with those of Kaminka et al., but regardless they are largely as expected. In terms of planning time and planner calls, the "Recompute" and "Prune" approaches both reduced the number of planner calls and, in turn, the planning time. Why this happened is obvious in the case of "Recompute"; in the case of "Prune" this happened because there is no need to continue computing paths for goals that have been pruned. Between the two, "Prune" outperformed "Recompute", since the paths still needed to be recomputed fairly often under the "Recompute" approach, whereas pruning a goal meant that goal's path never needed to recomputed again under any circumstances. Naturally, combining both strategies under the "Both" approach produces an even smaller number of calls. The "No Recompute" approach, meanwhile, had the best running time out of all the approaches, since it only ever computed a path once for each goal. Finally, the "Baseline" approach had the worst performance in terms of both planning time and number of planning calls, as expected.

In regards to accuracy (in terms of time to end and the frequency of correct rankings), the "Baseline" approach outperformed the "Recompute" approach; this is the most notable deviation my results had compared to those of Kaminka et al. However, I do not think this is surprising, as it is only natural that you get more accurate results by actually computing the optimal path from the robot's current position than you would making an educated guess as to what that path is. Thus, the "Recompute" approach sacrifices a small degree of accuracy in favour of improving the running time of the system. The "Prune" approach, meanwhile, was more accurate than "Baseline", which makes sense since removing a goal from the proceedings makes all the remaining goals (including the actual target goal) slightly more likely, assuming the goal being removed isn't the actual target goal. "Both", meanwhile, was slightly more accurate than "Recompute" due to the pruning being done and slightly less accurate than "Prune" due to only recomputing the path under certain conditions, and was also roughly on par with "Baseline", having a worse average time to end but a slightly better average frequency of correct rankings. Finally, “No Recompute” was naturally by far the least accurate of all the approaches.

6 Conclusion and Future Work

I have discussed in detail the framework I have developed for modelling plan and goal recognition problems in Belle and Levesque's situation calculus [3, 4], particularly in regards to the equal ease with which it can handle discrete and continuous probability spaces. I have also heavily discussed the Ergo-based implementation of said framework that I have developed. I have outlined various experimental problems and benchmarks I have used to demonstrate the variety of problems that can be modelled and solved in this framework, and also outlined various statistical evaluations I used evaluate the system's performance in terms of both running time and accuracy. Overall, the versatility of the framework has been clearly established, as has the ease with which it can be used.

There are definitely more ways my system could be expanded in the future, however. One task I was working on which I never quite completed involved projection, i.e. getting my system to display the likelihood of any given action being the next action to be performed by the observed agent, or the likelihood of a given sequence of actions being the next sequence of actions to be performed. For the most part, this is actually a fairly simple task, were it not for one roadblock I have not yet overcome. The method I used to calculate these predictions was to simply process the given action sequence the same way as I would normally (using another new Ergo execution mode called offlineStepColl that was later replaced by offlineStepMatch; rather than returning a list containing the remaining program and the action that was processed, offlineStepColl instead returns a list containing all possible first actions and the remaining program), except without actually updating the exoProg fluent (which is necessary since the likelihood of a future action may be affected by the current program state); while this works for the most part, there is one issue in that fluents are not updated directly in the program, and thus aren't updated when processing an action sequence. This causes trouble when the program expects certain fluents to have certain values that they would have by that point under normal conditions, but wouldn't have when "looking ahead". Being able to process queries about the likelihood future actions would be a very valuable feature for the system, and would thus be an ideal task to look into completing in the future.

There are also numerous ways the robot with multiple goals experiment could be expanded on. These include:

- Giving the goals a prior probability distribution. This would be a very simple matter of changing the initial distribution based on the problem specification.
- Allowing for a wider variety of different kinds of terrain, with different costs for traversing them. Making this change would be a bit more complex, but ultimately still fairly simple. First, a new obstacle function with a wider range of possible return values (one for each type of terrain) would have to be created. Second, the A* algorithm's cost function would itself have to be updated with the new costs for each type of terrain.
- Allowing the system to deal with a robot whose movement speed does not match the discretization procedure used for the A* version of the system. This could be dealt with by updating the discretization procedure so that the number of cells always matches up the robot's speed (i.e. such that the robot always travel exactly one cell every time step), but depending on the exact speed of the robot, this may not be feasible. Alternatively, the same discretization procedure as before could be used, but rather than always retrieving the next node in the optimal path, the system could instead "jump ahead" based on the speed of the robot – i.e. if the robot's speed is such that the robot would cover roughly 5 cells per time step, the system would always jump ahead five nodes when retrieving the next node in the optimal path.
- Related to the above, the system could be re-built to no longer assume that the robot travels with constant speed in the first place. In order to accommodate for this, the system could simply keep track of how the robot's speed changes over time and use the current movement speed to determine how many nodes to jump over when determining the next target node (i.e. with the number of nodes being skipped changing as the robot's speed changes). For the version of the system using maximum distance differences, the system would be largely the same, except the maximum and minimum values used to convert the differences to probability values would be constantly updated to account for the changes in the robot's speed. Meanwhile, for the version of the system using the robot's orientation and position, the changing speed could be used to potentially allow the system to refine its probability calculations. This could be done by also using the robot's acceleration as a component of the overall calculation.

- Allowing for the possibility that the robot's true goal is not any of the known potential goals. This would be a fairly complicated change; in order to allow for such a possibility, not only would the system have to determine an adequate initial probability that the goal is not any of the known potential goals, but there would also have to be a way to update this value accordingly. The best way to accomplish this would most likely be to find some means of determining that the robot's actions are not conducive to any of the known goals and updating the probability values when this happens, but actually quantifying which actions should trigger a scenario such as this is tricky. One possible idea would be to, in samples where the chosen goal is ”unknown” (i.e. the system decides that the robot's intended goal is not one of the known ones), determine what the returned probability value would would be for each known goal; if each value is low, the weight of this sample should be made higher; alternatively, if one or more of these values are high, the weight of this sample should be made lower. This would ensure that sample which chose ”unknown” as the robot's intended goal would only have high weights when the robot's actions do not seem conducive to any of the known goals.

- Building the system to allow for the possibility of moving obstacles; this would also be very tricky to accomplish. The maximum distance difference and orientation/position methods would be unaffected, but the only way to ensure accuracy with the A* method would be to constantly re-compute the path as the obstacles move about the environment. This would obviously be extremely inefficient, so alternative means of updating the optimal path calculation would have to be developed.

- Allowing for additional uncertainty in regards to the system's observations. This could be interesting and would allow for modelling of a system which cannot determine the robot's location/orientation with 100% accuracy. Building the system in this way would be a simple matter of using additional probability distributions to represent information such as the robot's position or observed turn angles.

- Allowing for multiple robots. Ultimately, the simplest means of having multiple robots in the system would be to simply execute the system for each robot separately. However, this would potentially cause problems in environments with narrow entrances that both robots pass through simultaneously. Each additional robot could be treated as a moving obstacle relative to the others, but this would run into the efficiency problems outlined above. More complex strategies could also be developed under the assumption that the robots have the ability to communicate with each other, with the system attempting to predict how the robots would behave in response to this communication to avoid running into each other. Other factors which may further complicate things are whether or not the robots are all controlled by a single agent and whether or not there's any uncertainty in the other robots/agent's knowledge.

- A 3-Dimensional version of the system could be examined. This would be a fairly simple extension to deal with, as all the calculations done by the system could be pretty easily extended to three dimensions. The trickiest matter would be adapting the A* algorithm, particularly in a way which represents the system with sufficient granularity without drastically increasing the running time of the system.

- Finally, it might also be interesting to investigate other methods for calculating the optimal path to the goal. As has been mentioned, there are many different means of planning out a path from a start position to a goal. It might be interesting to re-run this system with different path planning algorithms to try and see which algorithm produces the best results.

Bibliography

[1] Dorit Avrahami-Zilberbrand and Gal A. Kaminka. Fast and complete symbolic plan recognition. In *IJCAI*, pages 653–658. Professional Book Center, 2005.

[2] Fahiem Bacchus, Joseph Y. Halpern, and Hector J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence*, 111(1-2):171–208, 1999.

[3] Vaishak Belle and Hector J. Levesque. PREGO: an action language for belief-based cognitive robotics in continuous domains. In *AAAI*, pages 989–995. AAAI Press, 2014.

[4] Vaishak Belle and Hector J. Levesque. ALLEGRO: belief-based programming in stochastic dynamical domains. In *IJCAI*, pages 2762–2769. AAAI Press, 2015.

[5] Vaishak Belle and Hector J. Levesque. Reasoning about discrete and continuous noisy sensors and effectors in dynamical systems. *Artificial Intelligence*, 262:189–221, 2018.

[6] Francis Bisson, Froduald Kabanza, Abder Rezak Benaskeur, and Hengameh Irandoust. Provoking opponents to facilitate the recognition of their intentions. In *AAAI*, 2011.

[7] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI/IAAI*, pages 355–362, 2000.

[8] Hung Hai Bui, Svetha Venkatesh, and Geoff A. W. West. Policy recognition in the abstract hidden markov model. *J. Artif. Intell. Res. (JAIR)*, 17:451–499, 2002.

[9] Eugene Charniak and Robert P. Goldman. A Bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[10] Yi Chu, Young Chol Song, Richard Levinson, and Henry A. Kautz. Interactive activity recognition and prompting to assist people with cognitive disabilities. *JAISE*, 4(5):443–459, 2012.

[11] Robert Demolombe and Ana Mara Otermin Fernandez. Intention recognition in the situation calculus and probability theory frameworks. In *Computational Logic in Multi Agent Systems*, pages 358–372, London, 2005.

[12] Robert Demolombe and Erwan Hamon. What does it mean that an agent is performing a typical procedure? a formal definition in the situation calculus. In *AAMAS*, pages 905–911, 2002.

[13] Christopher W. Geib and Robert P. Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132, 2009.

[14] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[15] Giuseppe De Giacomo, Yves Lespérance, and Adrian R. Pearce. Situation calculus based programs for representing and reasoning about game structures. In *KR*, 2010.

[16] Robert P. Goldman, Christopher W. Geib, Henry A. Kautz, and Tamim Asfour. Plan recognition (dagstuhl seminar 11141). *Dagstuhl Reports*, 1(4):1–22, 2011.

[17] Alexandra Goultiaeva and Yves Lespérance. Incremental plan recognition in an agent programming framework. In *PAIR*, 2007.

[18] Peter Jarvis, Teresa F. Lunt, and Karen L. Myers. Identifying terrorist activity with ai plan recognition technology. *AI Magazine*, 26(3):73–81, 2005.

[19] Gal A. Kaminka, Mor Vered, and Noa Agmon. Plan recognition in continuous domains. In *AAAI*, pages 6202–6210. AAAI Press, 2018.

[20] Henry A. Kautz. A formal theory of plan recognition. Technical report, Dept. of Computer Science, University of Rochester, May 1987.

[21] Henry A. Kautz and James F. Allen. Generalized plan recognition. In *AAAI*, pages 32–37. Morgan Kaufmann, 1986.

[22] Hector J. Levesque. Manuscript on programming for cognitive robotics, 2018. Unpublished Manuscript.

[23] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

[24] Jay McCarthy, April 2013.

[25] John McCarthy and Patrick J. Hayes. Some Philosophical Problems From the StandPoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

[26] Miquel Ramírez and Hector Geffner. Plan recognition as planning. In *IJCAI*, pages 1778–1783, 2009.

[27] Miquel Ramírez and Hector Geffner. Probabilistic plan recognition using off-the-shelf classical planners. In *AAAI*. AAAI Press, 2010.

[28] Ray Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[29] Gita Sukthankar, Christopher Geib, Hung Hai Bui, David V. Pynadath, and Robert P. Goldman. Plan, activity, and intent recognition: theory and practice. *Burlington: Elsevier Science*, 2014.

[30] Karim A. Tahboub. Intelligent human-machine interaction based on dynamic Bayesian networks probabilistic intention recognition. *Journal of Intelligent and Robotic Systems*, 45(1):31–52, 2006.

[31] Mor Vered and Gal A. Kaminka. Heuristic online goal recognition in continuous domains. In *IJCAI*, pages 4447–4454. ijcai.org, 2017.

[32] Wen Zeng and Richard L. Church. Finding shortest paths on real road networks: the case for a*. *International Journal of Geographical Information Science*, 23:531–543, 2009.

A Appendices

A.1 Complete Code of the Target Shooting Example

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; author: YL, with minor alterations by Alistair Scheuhammer
;; date: 24 Nov 2019
;;
;; This is an Ergo implementation of a simple plan recognition example
;; involving discrete and continuous distributions.  The observed agent
;; selects a target, aims at it, and shoots.  Initially the target is
;; unknown, but after making noisy observations of the aim, we get more
;; accurate knowledge of the selected target.
;;
;;  Here, ERGO is used in onlineSynchronized mode where:
;;     - actions/beliefs query results generated by the program
;;       are simply printed
;;     - exogenous actions are read from a file targetShootObs3.txt
;;
;; To run:
;; In a terminal enter "racket -l ergoExt -i -f targetShootwExoProgV6.ergo.scm"
;;               and then run the program by entering "(main)"
;;
;; The file targetShootObs1.txt (in the same directory) should contain
;; the following:
;;      chooseTarget!
;;      setAim! or nSetAim!
;;      (obsAim! <angle>) e.g. (obsAim! 75.0) zero or more times
;;      shoot!
;;      (obsHit! <target>) or obsNoHit!
;;      halt!
;;
;; In the terminal, Ergo displays the updated beliefs, which evolve as expected.
;;
;; After (obsAim! 75.0) is entered, the agent thinks that it is equally
;; likely that target 1 and 2 are aimed at, and target 0 is extremely unlikely.
;; After (obsHit! 2) is entered, the agent is certain that target 2 was
;; aimed at and hit.
;;
;; If one enters nSetAim! instead of setAim!, the degree of belief that target 0
;; was aimed at after (obsAim! 75.0) is greater than in the previous case
;; because the aiming is inaccurate.
;;

;; Auxiliary definitions including non-fluents

(define (remainder n m)
  (- n (* (floor (/ n m)) m)))

(define nTargets 3)

(define targetPosition (vector 30.0 55.0 90.0))

(define targetSize (vector 10.0 30.0 20.0))

(define targetValue (vector 30.0 10.0 20.0))

;; Auxiliary definitions

(define (aimedAtTarget)
  (let loop ((t 0))
    (if (>= t nTargets) #f
        (if (and (>= aim (- (vector-ref targetPosition t)
                            (/ (vector-ref targetSize t) 2)))
                 (<= aim (+ (vector-ref targetPosition t)
                              (/ (vector-ref targetSize t) 2))))
            t
            (loop (+ t 1))))))

(define (max-vec-elem-index vec)
  (let loop ((i 1) (maxIndex 0))
    (if (>= i (vector-length vec))
        maxIndex
        (if (> (vector-ref vec i) (vector-ref vec maxIndex))
            (loop (+ 1 i) i)
            (loop (+ 1 i) maxIndex)))))

;; we use an alternative version of non-deterministic iteration
;; defined as follows

(define (:starDFS prog)
  (:choose :nil (:begin prog (:starDFS prog))))


;; Plans

(define (greedy-plan)
  (:begin
   (:act chooseTarget!)
   (:act nSetAim!)
   (:choose :nil (:act nSetAim!))
   (:starDFS (:act (obsAim! someObservedAim)))
   (:act shoot!)
   )
  )

(define (safe-plan)
  (:begin
   (:act chooseTarget!)
   (:act nSetAim!)
   (:choose :nil (:act nSetAim!))
   (:choose :nil (:act nSetAim!))
   (:starDFS (:act (obsAim! someObservedAim)))
   (:act shoot!)
   )
  )

(define (optimal-plan)
  (:begin
   (:act chooseTarget!)
   (:act nSetAim!)
   (:starDFS (:act (obsAim! someObservedAim)))
   (:act shoot!)
   )
  )


;; States and actions

(define-states ((i 1000000))
  decisionPlan 'notDecided
  decisionTarget 'notDecided
  aim 0.0
  someObservedAim 'notSet
  targetHit (vector #f #f #f)
  halted #f
  exoProg (:begin (:act choosePlan!)
                  (:if (eq? decisionPlan 'greedy) (greedy-plan)
                       (:if (eq? decisionPlan 'safe) (safe-plan)
                              (optimal-plan)))
                   (:choose (:for-some t '(0 1 2) (:act (obsHit! t)))
                            (:act obsNoHit!))
                   (:act halt!)
                   )
  )

(define-action halt! #:sequential? #t
  halted #t
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'halt! exoProg))) (if
       res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )

;; accurate decision action
(define-action choosePlan! #:sequential? #t
  decisionPlan (DISCRETE-GEN 'greedy 0.33 'safe 0.33 'optimal 0.34)
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'choosePlan! exoProg)
      )) (if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )

;; accurate decision action
(define-action chooseTarget! #:sequential? #t
  decisionTarget  (if (eq? decisionPlan 'greedy)
                      (max-vec-elem-index targetValue)
                      (if (eq? decisionPlan 'safe)
                           (max-vec-elem-index targetSize)
                           (max-vec-elem-index (vector-map * targetValue targetSize)))
                              )
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'chooseTarget!
      exoProg))) (if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )

;; accurate unobservable actuation action
(define-action setAim! #:sequential? #t
  aim (vector-ref targetPosition decisionTarget)
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'setAim! exoProg))) (
      if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )

;; noisy unobservable actuation action
(define-action nSetAim! #:sequential? #t
  aim (GAUSSIAN-GEN (vector-ref targetPosition decisionTarget) 5.0)
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'nSetAim! exoProg)))
      (if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )

;; noisy continuous sensor
(define-action (obsAim! a)  #:sequential? #t
  ;; #:prereq (and (>= a 0.0) (< a 360.0))
  someObservedAim a ;; fluent someObservedAim is set to argument angle a
                    ;; ensuring the exoProg can run to match the action
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch
                              #:matchAct (list 'obsAim! a) exoProg)))
             (if res (cadr res) :fail))
  weight (if (equal? exoProg :fail)
             0.0
             (* weight (GAUSSIAN (remainder a 360.0) aim 10.0)))
  )
```

```
;; accurate unobservable actuation action
(define-action shoot! #:sequential? #t
  targetHit (let ((aat (aimedAtTarget)))
              (if (not aat) targetHit
                  (vector-set targetHit aat #t)))
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'shoot! exoProg))) (
      if res (cadr res) :fail))
  weight (if (equal? exoProg :fail) 0.0 weight)
  )

;; accurate discrete sensor
(define-action (obsHit! t) #:sequential? #t
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch
                              #:matchAct (list 'obsHit! t) exoProg)))
                (if res (cadr res) :fail))
  weight (if (and (vector-ref targetHit t)
                  (not (equal? exoProg :fail)))
             weight
             0.0)
  )

;; accurate discrete sensor
(define-action obsNoHit! #:sequential? #t
  exoProg (let ((res (ergo-do #:mode 'offlineStepMatch #:matchAct 'obsNoHit! exoProg)))
       (if res (cadr res) :fail))
  weight (if (and (not (or-map (lambda (i) (vector-ref targetHit i))
                              (iota nTargets)))
                  (not (equal? exoProg :fail)))
             weight
             0.0
             )
  )

(define (displayBeliefs)
    (printf "(beliefl (eq? decisionPlan notDecided)) returns ~s~n" (belief (eq?
        decisionPlan 'notDecided)))
  (printf "(belief (eq? decisionPlan greedy)) returns ~s~n" (belief (eq? decisionPlan
      'greedy)))
  (printf "(belief (eq? decisionPlan safe)) returns ~s~n" (belief (eq? decisionPlan '
      safe)))
  (printf "(belief (eq? decisionPlan optimal)) returns ~s~n" (belief (eq? decisionPlan
       'optimal)))
  (printf "(beliefl (eq? decisionTarget notDecided)) returns ~s~n" (belief (eq?
      decisionTarget 'notDecided)))
  (printf "(belief (eq? decisionTarget 0)) returns ~s~n" (belief (eq? decisionTarget
      0)))
  (printf "(belief (eq? decisionTarget 1)) returns ~s~n" (belief (eq? decisionTarget
      1)))
  (printf "(belief (eq? decisionTarget 2)) returns ~s~n" (belief (eq? decisionTarget
      2)))

  (printf "(belief (aiming at target 0)) returns ~s~n"
           (belief (and (>= aim (- (vector-ref targetPosition 0)
                              (/ (vector-ref targetSize 0) 2)))
                 (<= aim (+ (vector-ref targetPosition 0)
                              (/ (vector-ref targetSize 0) 2)))))
           )

  (printf "(belief (aiming at target 1)) returns ~s~n"
           (belief (and (>= aim (- (vector-ref targetPosition 1)
                              (/ (vector-ref targetSize 1) 2)))
                 (<= aim (+ (vector-ref targetPosition 1)
                              (/ (vector-ref targetSize 1) 2)))))
           )
  (printf  "(belief (aiming at target 2)) returns ~s~n"
           (belief (and (>= aim (- (vector-ref targetPosition 2)
                                (/ (vector-ref targetSize 2) 2)))
                  (<= aim (+ (vector-ref targetPosition 2)
                                (/ (vector-ref targetSize 2) 2)))))
           )

  (printf  "(belief (vector-ref targetHit 0)) returns ~s~n"
           (belief (vector-ref targetHit 0))
           )

  (printf  "(belief (vector-ref targetHit 1)) returns ~s~n"
           (belief (vector-ref targetHit 1))
           )

  (printf  "(belief (vector-ref targetHit 2)) returns ~s~n"
           (belief (vector-ref targetHit 2))
           )
  )

(define buStartTime 0)

(define observeUpdtLoop
  (:begin
   (:>>  (let ((start  (current-milliseconds)))
                (displayBeliefs)
                (printf "Elapsed time for belief queries and display ~a ms\n"
                       (- (current-milliseconds) start))))
   (:while (< (belief (eq? halted #t)) 0.9)
           (:>> (set! buStartTime (current-milliseconds)))
           (:test #t)
           (:>> (printf "Elapsed time for belief update ~a ms\n"
                       (- (current-milliseconds) buStartTime)))
           (:>>  (let ((start  (current-milliseconds)))
                (displayBeliefs)
                (printf "Elapsed time for belief queries and display ~a ms\n"
                       (- (current-milliseconds) start))))

   )
  )
)

(define-interface 'out write-endogenous)

(define-interface 'in
  (let ((iport (open-input-file "BatchActions.txt")))
    (displayln "Opening file BatchActions.txt to receive exogenous actions!")
    (lambda () (let ((exog (read iport)))
                  exog))))

(define (main) (let ((runStartTime (current-milliseconds)))
                    (ergo-do #:mode 'onlineSynchronized observeUpdtLoop)
                    (printf "Total time for run ~a ms\n"
                          (- (current-milliseconds) runStartTime))
                  ))
```

A.2 Complete Code of the Jewelry Store Example

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; author: Alistair Scheuhammer
;;
;; This is an Ergo implementation of a very simple plan recognition
;; example. The observed agent is an individual who has entered a
;; jewelry store. Their intentions are not yet known, and it is the goal
;; of the systems to predict those intentions: specifically, whether
;; they are there to steal something, to buy something, or simply
;; to browse the store's selection. Stealing is the least common
;; choice, while browsing and buying are equally likely The store
;; offers three items for sale: a ring, a necklace, and a bracelet. A
;; simple Ergo program models the observed agent's behaviour.
;; First, they choose which of the three possible goals (steal,
;; browse, or buy) they are intending to pursue. Then, they enter the
;; shop and look around for a moment. Then, they approach the
;; counter and choose which of the three items of jewelry they are
;; planning on targeting; they are more likely to pick the ring than
;; the necklace or the bracelet if their goal is to steal the object,
;; but otherwise all three objects have an equal probability of being
;; selected. Next, they decide whether or not to look around again;
;; they have a fairly low chance of doing so when browsing or
;; buying, but are guaranteed to do so when stealing. Finally, if the
;; agent intends to buy the object, they will pay for it and leave. If
;; they intend to browse, they will set the object down and then
;; leave. If they intend to steal the object, they will leave without
;; doing anything else.
;;
;;  Here, ERGO is used with TCP in a basic way similar to that
;;  in the standard example reactive-elevator-tcp1.scm:
;;     - actions generated by the program are simply printed
;;     - exogenous actions arrive over TCP port 8678
;;
;; To run:
;; In terminal 1 enter "racket -l ergoExt -i -f jewelrystoreProgramFinal.scm"
;;              and then run the program by entering "(main)"
;;
;; In terminal 2 enter "telnet localhost 8345"
;;        and then enter the exogenous actions/observations at the prompt
;;              chooseGoal!
;;              lookAround!
;;              (take! ring)
;;              etc.
;;
;; In terminal 1, Ergo displays the updated beliefs, which evolve as expected.
;;
;; To stop, kill the racket process by entering ^C in terminal 1.

(define-states ((i 1000000))
   goalChoice (DISCRETE-GEN '1 0.2 '2 0.4 '3 0.4)
   decisionGoal 'notDecided
   objChoice (DISCRETE-GEN '1 1.0) ;; Doesn't get set until the observed agent decides
      which potential goal to pursue
   decisionObj 'notDecided
   lookChoice (DISCRETE-GEN '1 1.0) ;; Doesn't get set until the observed agent decides
      which potential goal to pursue
   decisionLook 'notDecided
   exoProg
      (:begin
         (:act chooseGoal!)
         (:act enterShop!)
         (:act lookAround!)
         (:act approachCounter!)
         (:act chooseObj!)
         (:for-some o (list 'ring 'necklace 'bracelet)
            (:test (eq? o decisionObj)) ;; Fails if o doesn't match with the object the
                observed agent chose
            (:act (take! o))
            (:act chooseLook!)
            (:if (eq? decisionLook 'yes)
               (:begin
                  (:act lookAround!)
                  (:if (eq? decisionGoal 'steal)
                     (:begin
                        (:act leaveShop!))
                     (:if (eq? decisionGoal 'browse)
                        (:begin
                           (:act (putDown! o))
                           (:act leaveShop!))
                        (:if (eq? decisionGoal 'buy)
                           (:begin
                              (:act (pay! o))
                              (:act leaveShop!))
                           (:fail)))))
               (:if (eq? decisionLook 'no)
                  (:begin
                     (:act wait!)
                     (:if (eq? decisionGoal 'steal)
                        (:begin
                           (:act leaveShop!))
                        (:if (eq? decisionGoal 'browse)
                           (:begin
                              (:act (putDown! o))
                              (:act leaveShop!))
                           (:if (eq? decisionGoal 'buy)
                              (:begin
                                 (:act (pay! o))
                                 (:act leaveShop!))
                              (:fail)))))
                  (:fail))))))

(define-action chooseGoal! #:sequential? #t
   decisionGoal (if (eq? goalChoice '1)
                   'steal
                   (if (eq? goalChoice '2)
                      'browse
                      (if (eq? goalChoice '3)
                         'buy
                         decisionGoal)))

   objChoice (if (eq? decisionGoal 'steal)
                (DISCRETE-GEN '1 0.5 '2 0.25 '3 0.25)
                (if (or (eq? decisionGoal 'browse) (eq? decisionGoal 'buy))
                   (DISCRETE-GEN '1 0.33 '2 0.335 '3 0.335)
                   objChoice))

   lookChoice (if (or (eq? decisionGoal 'browse) (eq? decisionGoal 'buy))
                (DISCRETE-GEN '1 0.2 '2 0.8)
                lookChoice)

   exoProg (let* ((act 'chooseGoal!)
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (equal? exoProg :fail)
             0.0
             weight))

(define-action chooseObj! #:sequential? #t
   decisionObj (if (eq? objChoice '1)
                  'ring
                  (if (eq? objChoice '2)
                     'necklace
                     (if (eq? objChoice '3)
                        'bracelet
                        decisionObj)))

   exoProg (let* ((act 'chooseObj!)
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (equal? exoProg :fail)
             0.0
             weight))

(define-action chooseLook! #:sequential? #t
   decisionLook (if (eq? lookChoice '1)
                   'yes
                   (if (eq? lookChoice '2)
                      'no
                      decisionLook))

   exoProg (let* ((act 'chooseLook!)
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (equal? exoProg :fail)
             0.0
             weight))

(define-action enterShop! #:sequential? #t
   exoProg (let* ((act 'enterShop!)
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (eq? exoProg :fail)
          0.0
          weight))

(define-action lookAround! #:sequential? #t
   exoProg (let* ((act 'lookAround!)
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (eq? exoProg :fail)
          0.0
          weight))

(define-action approachCounter! #:sequential? #t
   exoProg (let* ((act 'approachCounter!)
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (eq? exoProg :fail)
          0.0
          weight))

(define-action (take! o) #:sequential? #t
   exoProg (let* ((act (list 'take! o))
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (eq? exoProg :fail)
          0.0
          weight))

(define-action leaveShop! #:sequential? #t
   exoProg (let* ((act 'leaveShop!)
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (eq? exoProg :fail)
          0.0
          weight))

(define-action wait! #:sequential? #t
   exoProg (let* ((act 'wait!)
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (eq? exoProg :fail)
          0.0
          weight))

(define-action (putDown! o) #:sequential? #t
   exoProg (let* ((act (list 'putDown! o))
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (eq? exoProg :fail)
          0.0
          weight))

(define-action (pay! o) #:sequential? #t
   exoProg (let* ((act (list 'pay! o))
            (possConfigs (ergo-do #:mode 'offlineStepColl exoProg))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (eq? exoProg :fail)
          0.0
          weight))

(define (displayBeliefs)
  (printf "(belief (eq? decisionGoal 'notDecided)) returns ~s~n" (belief (eq?
      decisionGoal 'notDecided)))
  (printf "(belief (eq? decisionGoal 'steal)) returns ~s~n" (belief (eq? decisionGoal
      'steal)))
  (printf "(belief (eq? decisionGoal 'browse)) returns ~s~n" (belief (eq? decisionGoal
       'browse)))
  (printf "(belief (eq? decisionGoal 'buy)) returns ~s~n" (belief (eq? decisionGoal '
      buy)))
  (printf "(belief (eq? decisionObj 'notDecided)) returns ~s~n" (belief (eq?
      decisionObj 'notDecided)))
  (printf  "(belief (eq? decisionObj 'ring)) returns ~s~n" (belief (eq? decisionObj '
      ring)))
  (printf  "(belief (eq? decisionObj 'necklace)) returns ~s~n" (belief (eq? decisionObj
       'necklace)))
  (printf  "(belief (eq? decisionObj 'bracelet)) returns ~s~n" (belief (eq? decisionObj
       'bracelet)))
  (printf  "(belief (eq? decisionLook 'notDecided)) returns ~s~n" (belief (eq?
      decisionLook 'notDecided)))
  (printf  "(belief (eq? decisionLook 'yes)) returns ~s~n" (belief (eq? decisionLook '
      yes)))
  (printf  "(belief (eq? decisionLook 'no)) returns ~s~n" (belief (eq? decisionLook 'no
      )))
)

(define observeUpdtLoop
  (:while #t
          (:>>  (displayBeliefs))
          (:wait)
          (:>> (printf  "Exogenous Action Received~n"))))

(define-interface 'out write-endogenous)

(define-interface 'in
  (let ((ports (open-tcp-server 8345)))
    (displayln "Ready to receive exogenous actions!" (cadr ports))
    (lambda () (display "Act: " (cadr ports)) (read (car ports)))))

(define (main) (ergo-do #:mode 'online observeUpdtLoop))
```

A.3 Complete Code of the Robot with Multiple Goals Example

A.3.1 Utils.scm

```
(define (remainder n m) ;; Returns the the difference between m and the nearest
    multiple of n thats less than m
  (- n (* (floor (/ n m)) m)))

(define (approx-eq? x y err) ;; Returns true if x is between y - err and y + err,
    inclusive
   (and (<= x (+ y err)) (>= x (- y err))))

(define-syntax incf ;; Increases the numerical value of a variable
  (syntax-rules ()
    ((_ x)   (begin (set! x (+ x 1)) x))
    ((_ x n) (begin (set! x (+ x n)) x))))

(define-syntax decf ;; Decreases the numerical value of a variable
  (syntax-rules ()
    ((_ x)   (incf x -1))
    ((_ x n) (incf x (- n)))))

(define-syntax resetf ;; Resets the numerical value of a variable to 0
  (syntax-rules ()
    ((_ x)   (incf x (- x)))))
```

A.3.2 RandomEnvironmentGen.scm

```
((require dyoo-while-loop)

(include "Utils.scm")

(define numGoals 4)
(define numObstacles 100)

(define startPos (cons 5.0 2.0))

(define genCoord (cons 0.0 0.0)) ;; Used to temporarily store a generated coordinate
(define genRadius 0.0) ;; Used to temporarily store a generated radius

(define (overlapsObstacle? coord obstacleCoord obstacleRadius) ;; Returns true if the
    given coordinate is inside the given obstacle
   (let
      ((xDiff (abs (- (car coord) (car obstacleCoord))))
      (yDiff (abs (- (cdr coord) (cdr obstacleCoord)))))
         (let
            ((dist (sqrt (+ (expt xDiff 2) (expt yDiff 2)))))
               (or (< dist obstacleRadius) (approx-eq? dist obstacleRadius 0.05)))))

(define (overlapsObstacles? coord) ;; Returns true if the given coordinate overlaps any
     obstacle
   (for/or ([i (in-range numObstacles)])
      (overlapsObstacle? coord (vector-ref obstaclePositions i) (vector-ref
          obstacleRadii i))))

(define (overlapsGoals? coord) ;; Returns true if the given coordinate is within a 0.1
    x0.1 box centred on the goal coordinate for any goal
   (for/or ([i (in-range numGoals)])
      (and (approx-eq? (car coord) (car (vector-ref goalPositions i)) 0.05) (approx-eq?
           (car coord) (car (vector-ref goalPositions i)) 0.05))))
```

```
(define (overlapsStart? coord) ;; Returns true if the given coordinate is within a 0.1
    x0.1 box centred on the start coordinate
   (and (approx-eq? (car coord) (car startPos) 0.2) (approx-eq? (car coord) (car
       startPos) 0.2)))

(define goalPositions
   (make-vector numGoals (cons 0.0 0.0)))

(define obstaclePositions
   (make-vector numObstacles (cons 0.0 0.0)))

(define obstacleRadii
   (make-vector numObstacles 0.0))

(define (genEnvironment)
   (begin
      (for ([i (in-range numObstacles)])
         (begin
            (set! genCoord (cons (* (random) 10.0) (* (random) 10.0)))
            (set! genRadius (* (random) 2.5))
            (while (overlapsObstacle? startPos genCoord genRadius)
               (begin
                  (set! genCoord (cons (* (random) 10.0) (* (random) 10.0)))
                  (set! genRadius (* (random) 2.5))))
            (vector-set! obstaclePositions i genCoord)
            (vector-set! obstacleRadii i genRadius)))
      (for ([i (in-range numGoals)])
         (begin
            (set! genCoord (cons (* (random) 10.0) (* (random) 10.0)))
            (while (or (overlapsObstacles? genCoord) (overlapsStart? genCoord))
               (set! genCoord (cons (* (random) 10.0) (* (random) 10.0))))
            (vector-set! goalPositions i genCoord)))))

(define (envFunc x y) ;; A function which returns 1 f the coordinate defined by the x
    and y arguments overlaps any obstacle and 0 otherwise
   (if (overlapsObstacles? (cons x y))
      1
      0))
```

A.3.3 AStarFinal.scm

```
(require rackunit
         math/matrix
         racket/unit
         racket/match
         racket/list
         data/heap
         2htdp/image
         racket/runtime-path
         racket/include
         dyoo-while-loop)

;; This is an implementation of the A* algorithm by Jay McCarthy, found online at https
    ://raw.githubusercontent.com/jeapostrophe/jeapostrophe.github.com/source/posts
    /2013-04-15-astar.rkt and edited slightly by myself (Alistair Scheuhammer)

(define-signature graph^
  (node? edge? node-edges edge-src edge-cost edge-dest))

(define (make-map N func)
  (build-matrix N N func))

(struct map-node (M x y) #:transparent)
(struct map-edge (src dx dy dest))
```

```
(define-unit map@
  (import) (export graph^)

  (define node? map-node?)
  (define edge? map-edge?)
  (define edge-src map-edge-src)
  (define edge-dest map-edge-dest)

  (define (edge-cost e)
    (match-define (map-edge (map-node sM sx sy) _ _ (map-node dM dx dy)) e)
    (let
       ((baseCost (match (matrix-ref dM dx dy)
                    [0   1]
                    [1   1000])))
       (if (not (or (= sx dx) (= sy dy)))
          (* baseCost (sqrt 2))
          baseCost)))
  (define (node-edges n)
    (match-define (map-node M x y) n)
    (append*
     (for*/list ([dx (in-list '(1 0 -1))]
                 [dy (in-list '(1 0 -1))]
                 #:when
                 (not (and (zero? dx) (zero? dy))))
       (cond
         [(and (<= 0 (+ dx x) (sub1 (matrix-num-cols M)))
               (<= 0 (+ dy y) (sub1 (matrix-num-rows M))))
          (define dest (map-node M (+ dx x) (+ dy y)))
          (list (map-edge n dx dy dest))]
         [else
          empty])))))

(define (A* graph@ initial node-cost)
  (define-values/invoke-unit graph@ (import) (export graph^))
  (define count 0)
  (define node->best-path (make-hash))
  (define node->best-path-cost (make-hash))

  (hash-set! node->best-path      initial empty)
  (hash-set! node->best-path-cost initial 0)
  (define (node-total-estimate-cost n)
    (+ (node-cost n) (hash-ref node->best-path-cost n)))
  (define (node-cmp x y)
    (<= (node-total-estimate-cost x)
        (node-total-estimate-cost y)))
  (define open-set (make-heap node-cmp))
  (heap-add! open-set initial)

  (begin0
    (let/ec esc
      (while (not (= (heap-count open-set) 0))
        (define x (heap-min open-set))
        (heap-remove! open-set x)
        (set! count (add1 count))
        (define h-x (node-cost x))
        (define path-x (hash-ref node->best-path x))

        (when (zero? h-x)
          (esc (reverse path-x)))

        (define g-x (hash-ref node->best-path-cost x))
        (for ([x->y (in-list (node-edges x))])
          (define y (edge-dest x->y))
          (define new-g-y (+ g-x (edge-cost x->y)))
          (define old-g-y
            (hash-ref node->best-path-cost y +inf.0))
          (when (< new-g-y old-g-y)
            (hash-set! node->best-path-cost y new-g-y)
            (hash-set! node->best-path y (cons x->y path-x))
            (heap-add! open-set y))))
      #f)

    (printf "visited ~a nodes\n" count)))

(define ((make-node-cost GX GY) n)
  (match-define (map-node M x y) n)
  (let
      ((xDist (abs (- x GX)))
      (yDist (abs (- y GY))))
          (if (< xDist yDist)
              (+ (* xDist (sqrt 2)) (abs (- xDist yDist)))
              (+ (* yDist (sqrt 2)) (abs (- xDist yDist))))))
(define N 51)
(define (my-M func)
  (make-map N func))
(define (my-path sx sy gx gy func)
  (let
     ((result
        (time
         (A* map@
            (map-node (my-M func) sx sy)
            (make-node-cost gx gy)))))
     (printf "path is ~a long\n" (length result))
     result))
```

A.3.4 Discretize.scm

```
(define (discretize5 num)
   (exact-round (* num 5)))

(define (discretize5-inv num)
   (/ num 5))

(define (discretize5-biFunc func) ;; Converts a function to one whose output is equal
    to 5 times the original function, rounded to the nearest integer
   (? (x y) (func (discretize5-inv x) (discretize5-inv y))))
```

A.3.5 OptimalPathProbability.scm

```
(require racket/include)

(include "AStarFinal.scm")
(include "Discretize.scm")
(include "Utils.scm")

(define (safe-list-ref lst pos) ;; Retrieves an item stored in a list using pos an
    index into that list, guarding against an invalid index argument by either
    retrieving the first element in the list (if pos is too small) or the last element
    in the list (if pos is too large)
   (if (>= pos (length lst))
      (list-ref lst (sub1 (length lst)))
      (if (< pos 0)
         (list-ref lst 0)
         (list-ref lst pos))))
```

```
(define (pathGoal start-pos goal-pos func) ;; Generates a sequence of edges
    representing the path from star-pos to goal-pos using the A* algorithm
   (let
      ((sx (discretize5 (car start-pos)))
      (sy (discretize5 (cdr start-pos)))
      (gx (discretize5 (car goal-pos)))
      (gy (discretize5 (cdr goal-pos)))
      (obsFunc (discretize5-biFunc func)))
         (my-path sx sy gx gy obsFunc)))

(define index 0) ;; Used to keep track of how many steps along each goal path we've
    taken

(define path-cache (make-hash)) ;; Hash set storing all the generated goal paths

(define (set-path key start-pos goal-pos func) ;; Stores a generated goal path in the
    hash set
   (hash-set! path-cache key (pathGoal start-pos goal-pos func)))

(define (get-cur-node-src choice) ;; Gets the last visited node in the goal path
    specified by choice
   (cons (/ (map-node-x (map-edge-src (safe-list-ref (hash-ref! path-cache choice null)
        index))) 5.0) (/ (map-node-y (map-edge-src (safe-list-ref (hash-ref! path-cache
        choice null) index))) 5.0)))

(define (get-cur-node-dest choice) ;; Gets the next node in the goal path specified by
    choice
   (cons (/ (map-node-x (map-edge-dest (safe-list-ref (hash-ref! path-cache choice null
       ) index))) 5.0) (/ (map-node-y (map-edge-dest (safe-list-ref (hash-ref! path-
       cache choice null) index))) 5.0)))

(define (get-full-path choice) ;; Returns the full sequence of nodes for the goal path
    specified by choice
   (map (lambda (x) (cons (/ (map-node-x (map-edge-dest x)) 5) (/ (map-node-y (map-edge
       -dest x)) 5))) (hash-ref! path-cache choice null)))

(define (print-path choice) ;; Prints the full sequence of nodes for the goal path
    specified by choice
   (printf "Path: ~s" (cons (/ (map-node-x (map-edge-src (safe-list-ref (hash-ref! path
       -cache choice null) 0))) 5) (/ (map-node-y (map-edge-src (safe-list-ref (hash-
       ref! path-cache choice null) 0))) 5)))
   (printf "~s~n" (get-full-path choice)))

(define (get-distribution choice pos angle) ;; Returns the degree to which angle
    matches the orientation one standing at pos would need to have in order to be
    directly facing the next node in the goal path specified by choice
   (let
      ((intermediateGoal-pos (get-cur-node-dest choice)))
      (let
         ((idealAngle (remainder (+ 360.0 (radians->degrees (atan (- (cdr
             intermediateGoal-pos) (cdr pos)) (- (car intermediateGoal-pos) (car pos)))
             )) 360.0)))
         (GAUSSIAN angle idealAngle 40.0))))
```

A.3.6 MaximumDistanceProbability.scm

```
(require racket/include)

(include "Utils.scm")

(define max
   100.0)

(define (min)
   (- max))

(define (set-new-max val)
   (set! max val))

(define (a)
   (/ (- 1.0 0.0) (- max (min))))

(define (b)
   (- 1.0 (* (a) max)))

(define (normalize value)
   (+ (* (a) value) (b)))

(define old-pos ;; The previous position of the agent
   (cons -1.0 -1.0))

(define (set-old-pos val)
   (set! old-pos val))

(define difference-hash
   (make-hash))

(define (set-difference key diff)
   (hash-set! difference-hash key diff))

(define (get-difference pos goal-pos) ;; Returns the difference between the agent's
    current distance from goal-pos and the same distance before the agent's most recent
     move action
   (let
      ((oldXDiff (abs (- (car goal-pos) (car old-pos))))
      (oldYDiff (abs (- (cdr goal-pos) (cdr old-pos))))
      (newXDiff (abs (- (car goal-pos) (car pos))))
      (newYDiff (abs (- (cdr goal-pos) (cdr pos)))))
      (let
         ((oldLength (sqrt (+ (expt oldXDiff 2) (expt oldYDiff 2))))
         (newLength (sqrt (+ (expt newXDiff 2) (expt newYDiff 2)))))
         (- oldLength newLength))))

(define (get-distribution pos choice goals) ;; Sets a normalized difference (i.e.
    normalized to be a percentage based on where that difference falls in the range
    from min to max) for every goal based on the observed agent's current and previous
    positions, then returns the difference for the goal specified by choice
   (begin
      (define count 1)
      (for ([goal goals])
         (let
            ((difference (get-difference pos goal)))
            (set-difference count (normalize difference)))
         (set! count (add1 count)))
      (hash-ref difference-hash choice)))
```

A.3.7 OrientationDistanceProbability.scm

```
(require racket/include)

(include "Utils.scm")

(define (get-distribution pos angle goal-pos) ;; Returns 80\% times the degree to which
     pos matches goal-pos plus 20\% times the degree to which angle matches the
    orientation one at pos would need to be directly facing goal-pos
   (let
      ((xdiff (- (car goal-pos) (car pos)))
      (ydiff (- (cdr goal-pos) (cdr pos))))
      (let ((distanceFromGoal (sqrt (+ (expt xdiff 2) (expt ydiff 2)))))
         (let ((probPosition (GAUSSIAN distanceFromGoal 0.0 2.5)))
            (if (and (< xdiff 0.1) (< ydiff 0.1))
               probPosition
               (let ((idealAngle (remainder (+ 360.0 (radians->degrees (atan ydiff
                   xdiff))) 360.0)))
                  (let ((probDirection (GAUSSIAN angle idealAngle 20.0)))
                     (+ (* probDirection 0.8) (* probPosition 0.2)))))))))
```

A.3.8 robotOptimal.scm

```
(require racket/include
         racket/format)

(define curPos (cons 5.0 2.0))
(define direction 90.0)
(define speed 0.2)

(define (runRobot path goalPos filename) ;; Manoeuvres the robot through path and
    outputs the sequence of actions corresponding to this movement to filename
   (let ((outFile (open-output-file filename #:exists 'replace)))
      (displayln "(chooseGoal!)" outFile)
      (for/list ([intermediateGoal-pos path])
         (let ((idealAngle (remainder (+ 360.0 (radians->degrees (atan (- (cdr
             intermediateGoal-pos) (cdr curPos)) (- (car intermediateGoal-pos) (car
             curPos))))) 360.0))) ;; The orientation the robot needs to face to be
             facing directly at the next target node
            (let ((turnAngle (remainder (- idealAngle direction) 360.0))) ;; How much
                the robot needs to turn by to oriented at idealAngle
               (let ((action (string-append "(turnAndMove! " (~a turnAngle) ")")))
                  (displayln action outFile))
               (set! direction (remainder (+ direction turnAngle) 360.0))
               (let
                  ((x-speed (* speed (cos (degrees->radians direction))))
                  (y-speed (* speed (sin (degrees->radians direction)))))
                     (set! curPos (cons (+ x-speed (car curPos)) (+ y-speed (cdr curPos
                        ))))))))
      (flush-output outFile)))
```

A.3.9 robotOptimalNoise.scm

```
(require racket/include
         racket/format)

(define curPos (cons 5.0 2.0))
(define direction 90.0)
(define speed 0.2)
(define noise 10.0) ;; The amount of noise in the robot's turn actions

(define (runRobot path goalPos filename) ;; Manoeuvres the robot through path and
    outputs the sequence of actions corresponding to this movement to filename
   (let ((outFile (open-output-file filename #:exists 'replace)))
      (displayln "chooseGoal!" outFile)
      (for/list ([intermediateGoal-pos path])
         (let ((idealAngle (remainder (+ 360.0 (radians->degrees (atan (- (cdr
             intermediateGoal-pos) (cdr curPos)) (- (car intermediateGoal-pos) (car
             curPos))))) 360.0))) ;; The orientation the robot needs to face to be
             facing directly at the next target node
            (let ((turnAngle (+ (remainder (- idealAngle direction) 360.0) (- (* (
                random) noise 2) noise))))
               (let ((action (string-append "(turnAndMove! " (~a turnAngle) ")")))
                  (displayln action outFile))
               (set! direction (remainder (+ direction turnAngle) 360.0))
               (let
                  ((x-speed (* speed (cos (degrees->radians direction))))
                  (y-speed (* speed (sin (degrees->radians direction)))))
                     (set! curPos (cons (+ x-speed (car curPos)) (+ y-speed (cdr curPos
                        ))))))))
      (flush-output outFile)))
```

A.3.10 robotRandom.scm

```
(require racket/include
         racket/format)

(define curPos (cons 5.0 2.0))
(define direction 90.0)
(define speed 1.0)

(define (onGoal goalPos)
   (and (approx-eq? (car curPos) (car goalPos) 0.1) (approx-eq? (cdr curPos) (cdr
      goalPos) 0.1)))

(define (runRobot path goalPos filename) ;; Manoeuvres the robot through the
    environment until it reaches goalPos or until it has performed 100 move actions (
    with path only being included in the arguments for consistency with the other robot
     types); outputs the sequence of actions corresponding to this movement to filename
     throughout this process
   (let ((outFile (open-output-file filename #:exists 'replace)))
      (displayln "(chooseGoal!)" outFile)
      (for ([i (in-range 100)] #:break (onGoal goalPos))
         (let ((angle (* (random) 360.0)))
            (let ((action (string-append "(turnAndMove! " (~a angle) ")")))
               (displayln action outFile))
            (set! direction (remainder (+ direction angle) 360.0))
            (let
                ((x-speed (* speed (cos (degrees->radians direction))))
                (y-speed (* speed (sin (degrees->radians direction)))))
                   (set! curPos (cons (+ x-speed (car curPos)) (+ y-speed (cdr curPos))
                      )))))
   (flush-output outFile)))
```

A.3.11 Main Program

```
(require racket/include)

(include "RandomEnvironmentGen.scm")
(include "OptimalPathProbability.scm") ;; Or MaximumDistanceProbability.scm, or
    OrientationDistanceProbability.scm
(include "robotOptimal.scm") ;; Or robotOptimalNoise.scm, or robotRandom.scm
(include "Utils.scm")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; author: Alistair Scheuhammer
;;
;; This is an Ergo implementation of a plan recognition
;; example featuring a mobile robot and a randomly
;; generated environment. The randomly generated
;; environment is defined in "RandomEnvironmentGen.scm"
;; Throughout the environment are various randomly-placed
;; points serving as potential target locations for the
;; robot; as the robot explores the environment, the
;; system's job is to analyze the robot's movements
;; and make predictions regarding how likely each
;; potential target goal is to be the robot's actual target
;; goal. The environment also features various obstacles
;; for the robot to avoid. These obstacles are represented
;; as circles of varying the sizes and are also placed
;; randomly throughout the environment. Note: All
;; angles are measured in degrees with 0 degrees
;; representing facing exactly to the right.
;;
;; There are three different methods used for calculating
;; the probability of a given goal being the actual target
;; goal. The first uses the A* algorithm to calculate the
;; optimal path to each destination and compares the
;; robot's path to each of these optimal paths, with the
;; potential goal who's optimal path most closely
;; matches the actual path being treated as the most
;; likely goal. The second method calculates the
;; difference between the robot's distance from each
;; potential goal before and after every move action, with
;; the potential goal with the smallest difference being
;; seen as the most likely goal. Finally, the last method
;; measures the probability for any given goal as a
;; weighted sum of a) the degree to which the robot's
;; angle matches the angle necessary for the robot to be
;; looking directly at that goal, and b) how close the robot
;; is to that goal.  For all three of the above-described
;; methods, the prior probabilities for each destination are
;; also factored into the new probability calculated at each
;; time step, so if the system were convinced the robot
;; was going after, say, destination 1 for most of the
;; system's run, the robot suddenly making a movement
;; which most strongly corresponds to destination 2
;; wouldn't necessarily cause the system to abandon its
;; prior belief that destination 1 was the robot's target'
;; destination. Which of the three methods is used by the
;; depends on which of three external files is included in
;; the header: OptimalPathProbability.scm,
;; MaximumDistanceProbability.scm, or
;; OrientationDistanceProbability.scm.
;;
;; There are also three different types of robots used in
;; this example: a robot which always follows the optimal
;; path to its desired destination, a robot who follows an
;; near-optimal path but with some degree of noise in
;; every move action, and a robot that follows a
;; completely random path. Once again, which robot is
;; used is determined by which external file is included
;; in the header: robotOptimal.scm, robotOptimalNoise.scm,
;; or robotRandom.scm.
;;
;; In this example, the process is entirely automated;
;; running the program will cause it to select a random
;; potential goal to be the actual target goal, after which
;; it will simulate the robot's movement through the
;; environment, generating a sequence of actions which
;; it will then output to a text file. It will then use this text
;; file as input, reading each action in turn and using it
;; to update its beliefs.
;;
;; To run:
;; In a terminal enter "racket -l ergoExt -i -f robot-
    GoalsObstaclesRandomEnvironmentAutomated.scm"
;;                and then run the program by entering "(main)"
;;
;; In the terminal, Ergo displays the updated beliefs, which evolve as expected
    automatically.
```

```
;;
;; To stop, kill the racket process by entering ^C in the terminal. The program will
    automatically
;; end once the observed robot's entire action sequence has been processed.

(define robot-pos-init startPos)
(define robot-speed 0.2)
(define chosenGoal 0)

(define-states ((i 1000000))
   robot-pos robot-pos-init
   robot-direction 90.0
   decisionGoal 'notDecided
   decisionGoal-pos (cons -1.0 -1.0)
   turnAngle 0.0
   robot-prog
      (:begin
         (:act chooseGoal!)
         (:while #t
            (:act (turnAndMove! turnAngle)))))

(define-action chooseGoal! #:sequential? #t
   decisionGoal (UNIFORM-DISCRETE-GEN numGoals) ;; Each of the potential target goals
       is equally likely to be chosen

   decisionGoal-pos (vector-ref goalPositions (- decisionGoal 1))

   robot-prog (let* ((act 'chooseGoal!)
            (possConfigs (ergo-do #:mode 'offlineStepColl robot-prog))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (equal? robot-prog :fail)
             0.0
             weight))

(define-action (turnAndMove! angle) #:sequential? #t ;; Turn by the given the angle,
    then move one step
   turnAngle angle

   robot-direction (remainder (+ robot-direction turnAngle) 360.0)

   robot-pos (let
                 ((x-speed (* robot-speed (cos (degrees->radians robot-direction))))
                 (y-speed (* robot-speed (sin (degrees->radians robot-direction)))))
                    (cons (+ x-speed (car robot-pos)) (+ y-speed (cdr robot-pos))))

   robot-prog (let* ((act (list 'turnAndMove! angle))
            (possConfigs (ergo-do #:mode 'offlineStepColl robot-prog))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   weight (if (equal? robot-prog :fail)
            0.0
            (* weight (get-distribution decisionGoal robot-pos (remainder (+ robot-
                direction angle) 360.0)))))

(define (displayVals)
  (printf "(sample-mean robot-pos.X) returns ~s~n" (sample-mean (car robot-pos)))
  (printf "(sample-mean robot-pos.Y) returns ~s~n" (sample-mean (cdr robot-pos)))
  (printf "(sample-mean robot-speed) returns ~s~n" (sample-mean robot-speed))
  (printf "(sample-mean robot-direction) returns ~s~n" (sample-mean robot-direction))
  (printf "(sample-mean turnAngle) returns ~s~n" (sample-mean turnAngle))
  (printf "(belief (eq? decisionGoal 'notDecided)) returns ~s~n" (belief (eq?
      decisionGoal 'notDecided)))
  (printf  (goalBeliefsString numGoals)))

(define (goalBeliefsString n) ;; Recursive function to generate a string containing
    information about how likely the system thinks each potential goal is to be the
    true target goal
   (if (> n 1)
      (string-append (goalBeliefsString (- n 1)) (format "(belief (eq? decisionGoal ~s)
          ) returns ~s~n" n (belief (eq? decisionGoal n))))
      (format "(belief (eq? decisionGoal ~s)) returns ~s~n" n (belief (eq? decisionGoal
           n)))))

(define (checkUpdateNeeded) ;; Checks to see if the optimal path to each goal needs to
    be recomputed,
   (let
      ((posX (sample-mean (car robot-pos)))
      (posY (sample-mean (cdr robot-pos))))
      (when (for/and ([i (in-range numGoals)]) ;; Recompute the optimal path to each
          goal (starting from the robot's current position) if the robot's current
          position deviates significantly from all of the existing optimal paths.
            (or (not (approx-eq? posX (car (get-cur-node-dest (+ i 1))) 1.0)) (not (
               approx-eq? posY (cdr (get-cur-node-dest (+ i 1))) 1.0))))
         (begin
            (for ([i (in-range numGoals)])
               (set-path (+ i 1) (cons posX posY) (vector-ref goalPositions i) envFunc)
                  )
            (resetf index)))))

(define buStartTime 0) ;; The time at which the system begins updating its beliefs

(define observeUpdtLoop
  (:while #t
          (:>>  (let ((start (current-milliseconds)))
             (displayVals)
             (printf "Elapsed time for belief queries and display ~a ms~n" (- (current-
                milliseconds) start))))
          (:>> (set! buStartTime (current-milliseconds)))
          (:test #t) ;; Process the next input action
          (:>> (printf "Exogenous Action Received~n"))
          (:>> (printf "Elapsed time for belief update ~a ms~n" (- (current-
              milliseconds) buStartTime)))
          (:>> (incf index)))) ;; Increment the index used to determine what the next
              node to travel to in each optimal path is
          (:>>  (checkUpdateNeeded))))

(define-interface 'out write-endogenous)

(define-interface 'in
  (let ((iport (open-input-file "batchActions.txt")))
    (displayln "Opening file batchActions.txt to receive exogenous actions!")
    (lambda () (let ((exog (read iport)))
                 (printf "<<< Exogenous act: ~a\n" exog)
                 exog))))

(define (main)
   (genEnvironment)
   (printf  "Goal Positions: ~s~n" goalPositions)
   (printf  "Obstacle Positions: ~s~n" obstaclePositions)
   (printf  "Obstacle Radii ~s~n" obstacleRadii)
   (let ((preProcessingStartTime (current-milliseconds)))
      (for ([i (in-range numGoals)])
         (set-path (+ i 1) robot-pos-init (vector-ref goalPositions i) envFunc))
      (printf "Total time for pre-processing ~a ms\n" (- (current-milliseconds)
          preProcessingStartTime)))
```

```
(let ((goalChoice (exact-round (+ (* (random) 3.0) 1))))
   (set! chosenGoal goalChoice)
   (printf "Chosen goal: ~s~n" goalChoice)
   (runRobot (get-full-path goalChoice) (vector-ref goalPositions (- goalChoice 1))
       "batchActions3.txt"))
(let ((runStartTime (current-milliseconds)))
   (ergo-do #:mode 'onlineSynchronized observeUpdtLoop)
   (printf "Total time for run ~a ms\n" (- (current-milliseconds) runStartTime))))
```

A.4 Complete Code of the Intersection Example

A.4.1 Utils.scm

```
(define (remainder n m) ;; Returns the the difference between m and the nearest
    multiple of n thats less than m
  (- n (* (floor (/ n m)) m)))

(define (approx-eq? x y err) ;; Returns true if x is between y - err and y + err,
    inclusive
   (and (<= x (+ y err)) (>= x (- y err))))

(define-syntax incf ;; Increases the numerical value of a variable
  (syntax-rules ()
    ((_ x)   (begin (set! x (+ x 1)) x))
    ((_ x n) (begin (set! x (+ x n)) x))))

(define-syntax decf ;; Decreases the numerical value of a variable
  (syntax-rules ()
    ((_ x)   (incf x -1))
    ((_ x n) (incf x (- n)))))

(define-syntax resetf ;; Resets the numerical value of a variable to 0
  (syntax-rules ()
    ((_ x)   (incf x (- x)))))
```

A.4.2 Splines.scm

```
(require racket/include)

(include "Utils.scm")

;; This is a simple utility program for handling splines and calculating motion along a
     curved trajectory

(define y0 0)
(define y1 0)
(define s0 0)
(define s1 0)

(define table (make-hash))

(define (distance cp1 cp2)
   (let
      ((xdist (abs (- (car cp2) (car cp1))))
      (ydist (abs (- (cdr cp2) (cdr cp1)))))
         (sqrt (+ (expt xdist 2) (expt ydist 2)))))

(define (spline-func-x t)
   (let
      ((sum1 (* (expt t 3) (+ (* -2 (- (car y1) (car y0))) (car s0) (car s1))))
      (sum2 (* (expt t 2) (- (* 3 (- (car y1) (car y0))) (* 2 (car s0)) (car s1))))
      (sum3 (* t (car s0)))
      (sum4 (car y0)))
         (+ sum1 sum2 sum3 sum4)))

(define (spline-func-y t)
   (let
      ((sum1 (* (expt t 3) (+ (* -2 (- (cdr y1) (cdr y0))) (cdr s0) (cdr s1))))
      (sum2 (* (expt t 2) (- (* 3 (- (cdr y1) (cdr y0))) (* 2 (cdr s0)) (cdr s1))))
      (sum3 (* t (cdr s0)))
      (sum4 (cdr y0)))
```

```
        (+ sum1 sum2 sum3 sum4)))

(define (spline-func t)
   (cons (spline-func-x t) (spline-func-y t)))

(define (interpolate val)
   (let
      ((keys (hash-keys table)))
         (begin
            (define closest (list-ref keys 0))
            (for ([i (in-range 1 (length keys))])
               (when (> (abs (- val closest)) (abs (- val (list-ref keys i))))
                  (set! closest (list-ref keys i))))
            closest)))

(define (initialize-spline a-y0 a-y1 a-s0 a-s1)
   (begin
      (set! y0 a-y0)
      (set! y1 a-y1)
      (set! s0 a-s0)
      (set! s1 a-s1)
      (define distSum 0)
      (for ([i (in-range 5)])
         (hash-set! table distSum (/ i 4))
         (incf distSum (distance (spline-func (/ i 4)) (spline-func (/ (+ i 1) 4)))))))

(define (get-point arclength)
   (let
      ((t (hash-ref table (interpolate arclength) 0)))
         (spline-func t)))
```

A.4.3 Main Program

```
(require racket/include)

(include "Splines.scm")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; author: Alistair Scheuhammer
;;
;; This is an Ergo implementation of a very simple plan recognition
;; example. Two cars are at an intersection, arriving at approximately the
;; same time. One car represents the user (i.e. "we" are this car), while
;; the other car represents an observed agent. The goal of the system is
;; to use the other car's behaviour, as well as known information about
;; the world, to try and predict which of three courses of action the other
;; car will undertake: "speedUp" (i.e. the other car sees us and speeds up
;; to get through the intersection before we do), "slowDown" (i.e. the
;; other car see us and slows down to let us through first), and "continue"
;; (i.e. the other car does not see us and obliviously continues through
;; the intersection without changing its speed); let's call this the other
;; car's "tactic". Additionally, we would also like to predict whether or
;; not the other car plans on turning left, turning right, or travelling
;; straight through the intersection; let's call this the other car's
;; "goal". Taking a behaviour as being the combination of a goal and a
;; tactic, there are a total of 9 possible behaviours.
;;
;; The system starts with an initial position, velocity, and acceleration
;; for both vehicles, which get updated as time goes on. The system
;; provides no means for directly updating the "user" car's acceleration,
;; as that is not the point of this system; this system is intended to
;; make predictions about the other car's intentions based on their
;; behaviour, and as such it only retains the information about the "user"
;; car necessary for it to make its predictions. Directly updating the
;; "user" car's motion beyond those changes which arise naturally from
;; its initial acceleration and velocity would be the task of an external
;; tool.
;;
;; There are various other factors which serve to complicate the system.
;; Each of these factors affect our degree of belief regarding the other
;; car's behaviour. Three aspects of the world which influence this
;; degree of belief (each being positioned so as to be in the way of the
;; observed car) include the presence of a stop sign, the presence of a
;; traffic light, and the presence of a pedestrian. If there is a stop
;; sign, we are more confident the other car will try to slow down. If
;; there is a traffic light, the car is more likely to slow down on red
;; or amber lights and more likely to speed up or continue on green lights
;; (with amber lights also having a greater chance of resulting in the
;; car speeding up or continuing than red ones). In general, there is
;; always a 50/50 chance of their being a traffic light. Finally, the car
;; is also more likely to slow down if there if a pedestrian; note that
;; the pedestrian themselves is only modelled to travel forward at a
;; constant speed if they exist. As for the other car's decision as to
;; whether they are going to turn (and in which direction if they do),
;; there are two lanes, and right and left turns are only possible in the
;; right and left lanes, respectively. So, if we see the other car move
;; into the left lane, we know they aren't turning right.
;;
;; The other car's behaviour is modelled as an Ergo program consisting of
;; multiple steps. First, the other car chooses one of the nine
;; aforementioned sets of behaviours. Next, we observe other information
;; about the world (i.e. the presence of traffic lights, stop signs, and
;; pedestrians), and update our beliefs accordingly by multiplying the
;; weight of each sample by some appropriate probability value. The other
;; car then begins accelerating. The chosen behaviour affects the velocity
;; the system expects to see (naturally, if "speedUp" is the chosen
;; behaviour, then system expects the other car to speed up, and similarly
;; for the other two behaviours). The weight of each sample is updated by
;; how closely the observed behaviour matches the expected behaviour for
;; that sample's chosen goal. Next the other car chooses which lane it
;; wants to be in (either staying in the right lane as it is initially or
;; moving into the left lane). If the car chooses to move, it activates its
;; turn signal and shifts accordingly. There is some noise in this shifting
;; in that there is an amount the system expects to see, but the system will
;; accept any amount; again, the sample wights are updated based on how
;; closely the car's shifting matches the expected amount. After the car
;; shifts lanes, it continues accelerating as described above under it
;; reaches the edge of the intersection, at which point it travels in the
;; direction of its chosen destination. In the event that the other car
;; chose to turn, splines are used to control the turning motion of the car.
;;
;; As this is a preliminary version of the system, a number of assumptions
;; are made to simplify the system. First, we assume that we can perfectly
;; observe the other car's behaviour; i.e. if the other car accelerates by
;; exactly 3 m/s^2, then we observe an acceleration of exactly 3 m/s^2.
;; Second, we assume that the vehicles are outside the intersection as long
;; as they are a meter or more away from the centre of the intersection. We
;; also heavily simplify how the other car's ideal velocity is determined
;; using a constant "speed limit" parameter. If the other car is speeding up,
;; its ideal velocity will be equal to the speed limit plus 2; if it is
;; slowing down, its ideal velocity will be equal to the speed limit minus 2;
;; if it is continuing, its ideal velocity will be equal to the speed limit
;; exactly. Finally, we also assume that each behaviour is equally likely
;; initially.
;;
;; Note that the "user" car is considered to be moving along the positive y
;; direction while the other car is moving along the negative x direction.
;; In particularly note that due the other car's motion already being
;; negative, it is negative acceleration that is necessary to produce
;; "speedUp" behaviour. It is also worth noting that the only actions the
;; other car undertakes which cause the world to update are the "accelerate",
;; "shift", and "turn" actions, and these actions will cause the user car
;; and the pedestrian to travel forward at a constant speed. If the other
;; car does not change its initial behaviour, it will collide with both the
;; pedestrian and the user car. Finally, also note that by default the
;; centre of the intersection is assumed to be at coordinate (5, 5).
;;
;;  Here, ERGO is used with TCP in a basic way similar to that
;;  in the standard example reactive-elevator-tcp1.scm:
;;     - actions generated by the program are simply printed
;;     - exogenous actions arrive over TCP port 8678
;;
;; To run:
;; In terminal 1 enter "racket -l ergoExt -i -f intersection4.scm"
;;                and then run the program by entering "(main)"
;;
;; In terminal 2 enter "telnet localhost 8678"
;;       and then enter the exogenous actions/observations at the prompt
;;                 obsChooseGoal!
;;                 (obsPedestrian! yes)
;;                 (obsAccelerate! 2.0)
;;
;; In terminal 1, Ergo displays the updated beliefs, which evolve as expected.
;;
;; To stop, kill the racket process by entering ^C in terminal 1.

(define (solve-quadratic-equation-first a b c) ;; Returns one solution to the quadratic
     equation modelled by a, b, and c (specifically, the one produced by adding the
    square root).
  (define disc (sqrt (- (* b b)
                        (* 4.0 a c))))
  (/ (+ (- b) disc)
     (* 2.0 a)))

(define (solve-quadratic-equation-second a b c) ;; Returns one solution to the
    quadratic equation modelled by a, b, and c (specifically, the one produced by
    subtracting the square root).
  (define disc (sqrt (- (* b b)
                        (* 4.0 a c))))
  (/ (- (- b) disc)
     (* 2.0 a)))

(define (solve-quadratic-equation a b c) ;; Returns the non-negative solution to a
    quadratic equation modelled by a, b, and c.
   (if (eq? a 0.0) ;; If a is 0, return the solution to the equation 0 = c + bx
      (/ (- c) b)
      (let
         ((posResult (solve-quadratic-equation-first a b c)))
         (if (< posResult 0)
            (solve-quadratic-equation-second a b c)
            posResult))))

;; Initial values for the position, velocity, the acceleration of both cars, and speed
    limits
(define otherCar-pos-init (cons 9.5 5.5))
(define otherCar-vel-init (cons -1.0 0.0))
(define otherCar-acc-init (cons 0.0 0.0))
(define ourCar-pos-init (cons 5.5 0.5))
(define ourCar-vel-init (cons 0.0 1.0))
(define ourCar-acc-init (cons 0.0 0.0))
(define pedestrian-pos-init (cons 6.25 3.75))
```

```
(define pedestrian-vel-init (cons 0.0 0.25))
(define pedestrian-acc-init (cons 0.0 0.0))
(define speedLimit 1.0)

(define (ideal-acc cur-pos cur-vel goal-pos time)
   (/ (- cur-pos (car other-pos)) time)) ;; Use the equation p = p0 + vt to solve for
       the ideal velocity

(define-states ((i 1000000))
   otherCar-pos otherCar-pos-init
   otherCar-vel otherCar-vel-init
   otherCar-acc otherCar-acc-init
   ourCar-pos ourCar-pos-init
   ourCar-vel ourCar-vel-init
   ourCar-acc ourCar-acc-init
   pedestrian-pos pedestrian-pos-init
   pedestrian-vel pedestrian-vel-init
   pedestrian-acc pedestrian-acc-init
   accelAmount 0.0 ;; The value used to update the other car's acceleration
   shiftAmount 0.0 ;; The value used to update the other car's y-position when changing
        lanes
   decisionGoal 'notDecided
   decisionTactic 'notDecided
   pedestrianAnswer 'unknown
   stopSignAnswer 'unknown
   trafficLightAnswer 'unknown
   trafficLightColourAnswer 'unknown
   decisionLane 'notDecided
   lTurnSignal 'off
   rTurnSignal 'off
   turning 'false
   arclength 0.0
   otherCar-prog
      (:begin
         (:act obsChooseGoal!) ;; Will the other car be turning left, right, or
             travelling straight?
         (:act obsChooseTactic!) ;; Will the other car be speeding up, slowing down, or
              continuing at the speed limit?
         (:act (obsPedestrian! pedestrianAnswer)) ;; Is there a pedestrian?
         (:act (obsStopSign! stopSignAnswer)) ;; Is there a stop sign?
         (:act (obsTrafficLight! trafficLightAnswer)) ;; Is there a traffic light?
         (:when (eq? trafficLightAnswer 'yes)
            (:act (obsTrafficLightColour! trafficLightColourAnswer))) ;; What is the
                traffic light's colour?
         (:act (obsAccelerate! accelAmount))
         (:act obsChooseLane!) ;; Will the car be changing lanes?
         (:act (obsAccelerate! accelAmount))
         (:if (eq? decisionLane 'left)
            (:begin
               (:act obsLeftTurnSignal!)
               (:act (obsShift! shiftAmount))
               (:act obsLeftTurnSignal!))
            (:act (obsShift! shiftAmount)))
         (:until (<= (car otherCar-pos) 6.0) ;; Repeatedly accelerate until the
             intersection is reached.
            (:act (obsAccelerate! accelAmount)))
         (:if (eq? decisionGoal 'turnRight)
            (:act obsRightTurnSignal!)
            (:when (eq? decisionGoal 'turnLeft)
               (:act obsLeftTurnSignal!)))
         (:if (eq? decisionGoal 'turnRight) ;; Turn until the intersection is passed
            (:until (>= (cdr otherCar-pos) 6.0)
               (:act obsTurn!))
            (if (eq? decisionGoal 'turnLeft)
            (:until (<= (cdr otherCar-pos) 4.0)
               (:act obsTurn!))
            (:until (<= (car otherCar-pos) 4.0)
               (:act obsTurn!))))))

(define-action obsChooseGoal! #:sequential? #t
   decisionGoal (DISCRETE-GEN 'forward 0.33 'turnRight 0.335 'turnLeft 0.335) ;; All
       three behaviours are assumed to be equally likely initially

   otherCar-prog (let* ((act 'obsChooseGoal!)
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail)
            0.0
            weight))

(define-action obsChooseTactic! #:sequential? #t
   decisionTactic (DISCRETE-GEN 'speedUp 0.33 'continue 0.335 'slowDown 0.335) ;; All
       three behaviours are assumed to be equally likely initially

   otherCar-prog (let* ((act 'obsChooseTactic!)
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail)
            0.0
            weight))

(define-action (obsPedestrian! answer) #:sequential? #t
   pedestrianAnswer answer

   otherCar-prog (let* ((act (list 'obsPedestrian! answer))
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail) ;; Update sample weights based on the chosen
        tactic and whether or not a pedestrian was observed.
            0.0
            (if (eq? decisionTactic 'speedUp)
               (* weight (DISCRETE answer 'yes 0.2 'no 0.8))
                  (if (eq? decisionTactic 'slowDown)
                     (* weight (DISCRETE answer 'yes 0.7 'no 0.3))
                        (if (eq? decisionTactic 'continue)
                           (* weight (DISCRETE answer 'yes 0.1 'no 0.9))
                           weight)))))

(define-action (obsStopSign! answer) #:sequential? #t
   stopSignAnswer answer

   otherCar-prog (let* ((act (list 'obsStopSign! answer))
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail) ;; Update sample weights based on the chosen
        tactic and whether or not a stop sign was observed.
            0.0
            (if (eq? decisionTactic 'speedUp)
               (* weight (DISCRETE answer 'yes 0.05 'no 0.95))
                  (if (eq? decisionTactic 'slowDown)
                  (* weight (DISCRETE answer 'yes 0.9 'no 0.1))
                     (if (eq? decisionTactic 'continue)
                        (* weight (DISCRETE answer 'yes 0.05 'no 0.95))
                        weight)))))

(define-action (obsTrafficLight! answer) #:sequential? #t
   trafficLightAnswer answer

   otherCar-prog (let* ((act (list 'obsTrafficLight! answer))
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail)
            0.0
            (* weight (DISCRETE answer 'yes 0.5 'no 0.5)))) ;; There is always a 50-50
                 chance of there being a traffic light

(define-action (obsTrafficLightColour! answer) #:sequential? #t
   trafficLightColourAnswer answer

   otherCar-prog (let* ((act (list 'obsTrafficLightColour! answer))
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail) ;; Update sample weights based on the chosen
        tactic and what colour was observed from the traffic light.
            0.0
            (if (eq? decisionTactic 'speedUp)
               (* weight (DISCRETE answer 'green 0.75 'red 0.05 'amber 0.2))
                  (if (eq? decisionTactic 'slowDown)
                     (* weight (DISCRETE answer 'green 0.05 'red 0.5 'amber 0.45))
                        (if (eq? decisionTactic 'continue)
                           (* weight (DISCRETE answer 'green 0.85 'red 0.05 'amber
                               0.1))
                           weight)))))

(define-action (obsAccelerate! amount) #:sequential? #t
   accelAmount amount

   otherCar-prog (let* ((act (list 'obsAccelerate! amount))
           (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
           (config (assoc act possConfigs)))
           (if config (cadr config) :fail))

   otherCar-acc (cons (+ (car otherCar-acc) accelAmount) (cdr otherCar-acc)) ;;
       Increase the other car's acceleration by accelAmount

   otherCar-vel (cons (+ (car otherCar-vel) (car otherCar-acc)) (+ (cdr otherCar-vel) (
       cdr otherCar-acc)))

   otherCar-pos (cons (+ (car otherCar-pos) (car otherCar-vel)) (+ (cdr otherCar-pos) (
       cdr otherCar-vel)))

   ourCar-vel (cons (+ (car ourCar-vel) (car ourCar-acc)) (+ (cdr ourCar-vel) (cdr
       ourCar-acc)))

   ourCar-pos (cons (+ (car ourCar-pos) (car ourCar-vel)) (+ (cdr ourCar-pos) (cdr
       ourCar-vel)))

   pedestrian-vel (cons (+ (car pedestrian-vel) (car pedestrian-acc)) (+ (cdr
       pedestrian-vel) (cdr pedestrian-acc)))
   pedestrian-pos (cons (+ (car pedestrian-pos) (car pedestrian-vel)) (+ (cdr
       pedestrian-pos) (cdr pedestrian-vel)))

   weight (if (equal? otherCar-prog :fail) ;; Update the sample weight using a Gaussian
        distribution centred on the expected velocity for that sample's chosen goal
            0.0
            (if (eq? decisionTactic 'speedUp)
               (* weight (GAUSSIAN (car otherCar-vel) (- (+ speedLimit 2.0)) 0.5)) ;;
                   Taking the negative of the speed limit formula due to the car
                   travelling in the negative direction.
               (if (eq? decisionTactic 'continue)
                  (* weight (GAUSSIAN (car otherCar-vel) (- speedLimit) 0.5))
                  (if (eq? decisionTactic 'slowDown)
                     (* weight (GAUSSIAN (car otherCar-vel) (- (- speedLimit 2.0))
                         0.5))
                     weight)))))

(define-action obsChooseLane! #:sequential? #t
  decisionLane (if (eq? decisionGoal 'forward) ;; Each lane has a different probability
       depending on the chosen goal
                 (DISCRETE-GEN 'left 0.6 'right 0.4)
                 (if (eq? decisionGoal 'turnLeft)
                    (DISCRETE-GEN 'left 0.95 'right 0.05)
                    (if (eq? decisionGoal 'turnRight)
                       (DISCRETE-GEN 'left 0.05 'right 0.95)
                       decisionLane)))

   otherCar-prog (let* ((act 'obsChooseLane!)
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail)
            0.0
            weight))

(define-action obsLeftTurnSignal! #:sequential? #t ;; Switch the left turn signal
    between the "on" and "off" positions
   lTurnSignal (if (eq? lTurnSignal 'on)
                 'off
                 'on)

   otherCar-prog (let* ((act 'obsLeftTurnSignal!)
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail)
            0.0
            weight))

(define-action obsRightTurnSignal! #:sequential? #t ;; Switch the right turn signal
    between the "on" and "off" positions
   rTurnSignal (if (eq? rTurnSignal 'on)
                 'off
                 'on)

   otherCar-prog (let* ((act 'obsRightTurnSignal!)
         (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
         (config (assoc act possConfigs)))
         (if config (cadr config) :fail))

   weight (if (equal? otherCar-prog :fail)
            0.0
```

```
            weight))

(define-action (obsShift! amount) #:sequential? #t
   shiftAmount amount

   otherCar-prog (let* ((act (list 'obsShift! amount))
            (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
            (config (assoc act possConfigs)))
            (if config (cadr config) :fail))

   otherCar-vel (cons (+ (car otherCar-vel) (car otherCar-acc)) (+ (cdr otherCar-vel) (
       cdr otherCar-acc)))

   otherCar-pos (cons (+ (car otherCar-pos) (car otherCar-vel)) (+ (cdr otherCar-pos) (
       cdr otherCar-vel) amount))

   ourCar-vel (cons (+ (car ourCar-vel) (car ourCar-acc)) (+ (cdr ourCar-vel) (cdr
       ourCar-acc)))

   ourCar-pos (cons (+ (car ourCar-pos) (car ourCar-vel)) (+ (cdr ourCar-pos) (cdr
       ourCar-vel)))

   pedestrian-vel (cons (+ (car pedestrian-vel) (car pedestrian-acc)) (+ (cdr
       pedestrian-vel) (cdr pedestrian-acc)))

   pedestrian-pos (cons (+ (car pedestrian-pos) (car pedestrian-vel)) (+ (cdr
       pedestrian-pos) (cdr pedestrian-vel)))

   weight (if (equal? otherCar-prog :fail) ;; Update the sample weight using a Gaussian
        distribution centred on the expected position of the other car, based on the
       chosen lane
            0.0
            (if (eq? decisionLane 'left)
               (* weight (GAUSSIAN (cdr otherCar-pos) 4.5 0.125))
               (if (eq? decisionLane 'right)
                  (* weight (GAUSSIAN (cdr otherCar-pos) 5.5 0.125))
                  weight))))

(define-action obsTurn! #:sequential? #t ;; Use splines to transition the other car
    through the intersection
   otherCar-prog (let* ((act 'obsTurn!)
        (possConfigs (ergo-do #:mode 'offlineStepColl otherCar-prog))
        (config (assoc act possConfigs)))
        (if config (cadr config) :fail))

   otherCar-vel (cons (+ (car otherCar-vel) (car otherCar-acc)) (+ (cdr otherCar-vel) (
       cdr otherCar-acc))) ; Note that since the simulation ends once the car has
       finished turning, we don't need to concern ourselves with updating the direction
        of the other car's velocity/acceleration

   arclength (+ arclength (abs (car otherCar-vel))) ;; Travel along the spline at a
       constant speed

   otherCar-pos (if (not (eq? weight 0.0))
                  (if (eq? decisionGoal 'turnRight)
                     (begin
                        (when (eq? turning 'false)
                           (initialize-spline otherCar-pos (cons 5.5 6.0) (cons -1.0
                               0.0) (cons 0.0 1.0))) ;; Initialize the start and end
                               points of the spline for turning right
                        (get-point arclength)) ;; Travel along the spline
                     (if (eq? decisionGoal 'turnLeft)
                        (begin
                           (when (eq? turning 'false)
                               (initialize-spline otherCar-pos (cons 4.5 4.0) (cons
                                   -1.0 0.0) (cons 0.0 -1.0))) ;; Initialize the start
                                   and end points of the spline for turning left
                           (get-point arclength)) ;; Travel along the spline
                        (cons (+ (car otherCar-pos) (car otherCar-vel)) (+ (cdr
                            otherCar-pos) (cdr otherCar-vel))))) ;; If the car isn't
                            turning, simply have it move forward at its current speed
                  otherCar-pos)

   turning 'true ;; Used to prevent the spline from being initialized at every time
       step

   ourCar-vel (cons (+ (car ourCar-vel) (car ourCar-acc)) (+ (cdr ourCar-vel) (cdr
       ourCar-acc)))

   ourCar-pos (cons (+ (car ourCar-pos) (car ourCar-vel)) (+ (cdr ourCar-pos) (cdr
       ourCar-vel)))

   pedestrian-vel (cons (+ (car pedestrian-vel) (car pedestrian-acc)) (+ (cdr
       pedestrian-vel) (cdr pedestrian-acc)))

   pedestrian-pos (cons (+ (car pedestrian-pos) (car pedestrian-vel)) (+ (cdr
       pedestrian-pos) (cdr pedestrian-vel)))

   weight (if (equal? otherCar-prog :fail)
            0.0
            weight))

(define (displayVals)
  (printf "(sample-mean otherCar-pos.X) returns ~s~n" (sample-mean (car otherCar-pos))
      )
  (printf "(sample-mean otherCar-pos.Y) returns ~s~n" (sample-mean (cdr otherCar-pos))
      )
  (printf "(sample-mean otherCar-vel.X) returns ~s~n" (sample-mean (car otherCar-vel))
      )
  (printf "(sample-mean otherCar-vel.Y) returns ~s~n" (sample-mean (cdr otherCar-vel))
      )
  (printf "(sample-mean otherCar-acc.X) returns ~s~n" (sample-mean (car otherCar-acc))
      )
  (printf "(sample-mean otherCar-acc.Y) returns ~s~n" (sample-mean (cdr otherCar-acc))
      )
  (printf "(sample-mean ourCar-pos.X) returns ~s~n" (sample-mean (car ourCar-pos)))
  (printf "(sample-mean ourCar-pos.Y) returns ~s~n" (sample-mean (cdr ourCar-pos)))
  (printf "(sample-mean ourCar-vel.X) returns ~s~n" (sample-mean (car ourCar-vel)))
  (printf "(sample-mean ourCar-vel.Y) returns ~s~n" (sample-mean (cdr ourCar-vel)))
  (printf "(sample-mean ourCar-acc.X) returns ~s~n" (sample-mean (car ourCar-acc)))
  (printf "(sample-mean ourCar-acc.Y) returns ~s~n" (sample-mean (cdr ourCar-acc)))
  (printf "(sample-mean pedestrian-pos.X) returns ~s~n" (sample-mean (car pedestrian-
      pos)))
  (printf "(sample-mean pedestrian-pos.Y) returns ~s~n" (sample-mean (cdr pedestrian-
      pos)))
  (printf "(sample-mean pedestrian-vel.X) returns ~s~n" (sample-mean (car pedestrian-
      vel)))
  (printf "(sample-mean pedestrian-vel.Y) returns ~s~n" (sample-mean (cdr pedestrian-
      vel)))
  (printf "(sample-mean pedestrian-acc.X) returns ~s~n" (sample-mean (car pedestrian-
      acc)))
  (printf "(sample-mean pedestrian-acc.Y) returns ~s~n" (sample-mean (cdr pedestrian-
      acc)))
  (printf "(belief (eq? decisionGoal 'notDecided)) returns ~s~n" (belief (eq?
      decisionGoal 'notDecided)))
  (printf "(belief (eq? decisionGoal 'turnRight)) returns ~s~n" (belief (eq?
      decisionGoal 'turnRight)))
  (printf "(belief (eq? decisionGoal 'turnLeft)) returns ~s~n" (belief (eq?
      decisionGoal 'turnLeft)))
  (printf "(belief (eq? decisionGoal 'forward)) returns ~s~n" (belief (eq?
      decisionGoal 'forward)))
  (printf "(belief (eq? decisionTactic 'notDecided)) returns ~s~n" (belief (eq?
      decisionTactic 'notDecided)))
  (printf "(belief (eq? decisionTactic 'speedUp)) returns ~s~n" (belief (eq?
      decisionTactic 'speedUp)))
  (printf "(belief (eq? decisionTactic 'continue)) returns ~s~n" (belief (eq?
      decisionTactic 'continue)))
  (printf "(belief (eq? decisionTactic 'slowDown)) returns ~s~n" (belief (eq?
      decisionTactic 'slowDown)))
  (printf "(belief (eq? pedestrianAnswer 'unknown)) returns ~s~n" (belief (eq?
      pedestrianAnswer 'unknown)))
  (printf "(belief (eq? pedestrianAnswer 'yes)) returns ~s~n" (belief (eq?
      pedestrianAnswer 'yes)))
  (printf "(belief (eq? pedestrianAnswer 'no)) returns ~s~n" (belief (eq?
      pedestrianAnswer 'no)))
  (printf "(belief (eq? stopSignAnswer 'unknown)) returns ~s~n" (belief (eq?
      stopSignAnswer 'unknown)))
  (printf "(belief (eq? stopSignAnswer 'yes)) returns ~s~n" (belief (eq?
      stopSignAnswer 'yes)))
  (printf "(belief (eq? stopSignAnswer 'no)) returns ~s~n" (belief (eq? stopSignAnswer
       'no)))
  (printf "(belief (eq? trafficLightAnswer 'unknown)) returns ~s~n" (belief (eq?
      trafficLightAnswer 'unknown)))
  (printf "(belief (eq? trafficLightAnswer 'yes)) returns ~s~n" (belief (eq?
      trafficLightAnswer 'yes)))
  (printf "(belief (eq? trafficLightAnswer 'no)) returns ~s~n" (belief (eq?
      trafficLightAnswer 'no)))
  (printf "(belief (eq? trafficLightColourAnswer 'unknown)) returns ~s~n" (belief (eq?
       trafficLightColourAnswer 'unknown)))
  (printf "(belief (eq? trafficLightColourAnswer 'red)) returns ~s~n" (belief (eq?
      trafficLightColourAnswer 'red)))
  (printf "(belief (eq? trafficLightColourAnswer 'green)) returns ~s~n" (belief (eq?
      trafficLightColourAnswer 'green)))
  (printf "(belief (eq? trafficLightColourAnswer 'amber)) returns ~s~n" (belief (eq?
      trafficLightColourAnswer 'amber)))
  (printf "(belief (eq? decisionLane 'notDecided)) returns ~s~n" (belief (eq?
      decisionLane 'notDecided)))
  (printf "(belief (eq? decisionLane 'left)) returns ~s~n" (belief (eq? decisionLane '
      left)))
  (printf "(belief (eq? decisionLane 'right)) returns ~s~n" (belief (eq? decisionLane
      'right))))

(define observeUpdtLoop
  (:while #t
        (:>> (displayVals))
        (:wait)
        (:>> (printf "Exogenous Action Received~n"))))

(define-interface 'out write-endogenous)

(define-interface 'in
  (let ((ports (open-tcp-server 8678)))
    (displayln "Ready to receive exogenous actions!" (cadr ports))
    (lambda () (display "Act: " (cadr ports)) (read (car ports)))))

(define (main)
   (ergo-do #:mode 'online observeUpdtLoop))
```