Simulation-Based Autonomous Systems in Discrete and Continuous Domains

Lenz Belzner

Dissertation an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität München

München, den 31.03.2016

1. Gutachter: Prof. Dr. Martin Wirsing
2. Gutachter: Prof. Alberto Lluch Lafuente, PhD
Tag der mündlichen Prüfung: 23.05.2016

Eidesstattliche Versicherung
(Siehe Promotionsordnung vom 12.07.11, §8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Belzner, Lenz .....
Name, Vorname

München, den 31.03.2016

.....
Unterschrift Doktorand/in

Abstract

Highly dynamic, probabilistic and potentially only partially known domains render classical techniques for system development infeasible. Enabling autonomous adaptation by providing decision making and learning capabilities yields systems with the abilities that are necessary to deal with these challenges. The key to system autonomy is to drop exact specification of runtime behavior in favor of a whole *space* of solutions. This space can be explored by the system at runtime according to its current situation, and potential traces in this space can be evaluated w.r.t. the system's goals. Concrete behavior is then compiled based on the results of search and evaluation.

This thesis studies the use of simulation-based Monte Carlo methods for decision making and learning that enable efficient and adequate system autonomy in discrete and continuous domains.

Zusammenfassung

Hochdynamische, probabilistische und möglicherweise nur teilweise bekannte Domänen erschweren die Anwendung klassischer Ansätze der Systementwicklung. Automatisierte Entscheidungs– und Lernprozesse ermöglichen autonome Anpassung und unterstützen resultierende Systeme dabei, mit diesen Herausforderungen umzugehen. Der Schlüssel zur Systemautonomie ist dabei das Ersetzen einer exakten Spezifikation des Laufzeitverhaltens durch einen ganzen Lösungs*raum*. Dieser kann von einem System zur Laufzeit unter Einbeziehen seiner aktuellen Situation durchsucht, und potentielle Abläufe in Bezug auf die Systemziele bewertet werden. Konkretes Verhalten ist dann ein Resultat von Suche und Bewertung.

Diese Arbeit untersucht auf Simulation basierende Monte-Carlo-Methoden für Entscheidungs– und Lernprozesse, um effiziente und adäquate Systemautonomie in diskreten und kontinuierlichen Domänen zu ermöglichen.

Acknowledgements

I thank my supervisor Martin Wirsing for making this work possible. Thank you for the freedom that encouraged exploration, for providing invaluable guidance and for your enthusiasm and interest in our joint work – one can say we are a great "collective adaptive system".

I thank Alberto Lluch Lafuente for his patience and interest, and for asking about my progress just at the right moments. Thank you for your gentle and encouraging smile at the various occasions we met.

I thank my coworkers Alexander Neitz, Rolf Hennicker and Martin Wirsing for the fruitful discussions on the topics of this thesis and beyond. Thank you for all the things I learned from and with you. I thank Matthias Hölzl for invaluable help and inspiration, and for providing a starting point.

I thank my colleagues. You made working worthwhile (campfire beneath Bavarian skies, or a lively discussion on the train about the meaning of adaptation – audience included, you name it), and I am thankful for your inspiration and support.

I thank my family for their love. I thank my friends for their music. And I thank my sweetheart for always listening – even when pretending not to. Thank you for being with me on this mysterious and wonderful journey.

Contents

Chapter 1

Introduction

I'd far rather be happy than right any day.

Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

1.1 Motivation

Modern software systems are required to deal with highly complex application domains [WHKM15]. Change and uncertainty play a central role both when specifying a system and also at runtime. Application environments typically exhibit a large amount of dynamics that are not directly induced by the designed system. Rather, the system itself represents only a part of the whole domain. The interplay of system activity as a reaction to environmental development and the influence of this activity on the course of events yields enormous challenges to building systems for these domains. Figure 1.1 illustrates this mutual influence of system behavior and domain dynamics.

In the face of these challenges, this thesis is driven by a central question.

How to build systems that autonomously and adequately react to change?

Highly dynamic, probabilistic and potentially only partially known domains render classical techniques for system development infeasible. Enabling autonomous adaptation by providing decision making and learning capabilities yields systems with the abilities that are necessary to deal with these challenges.

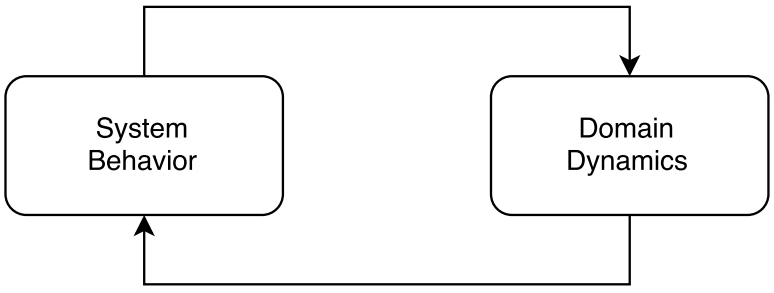


Figure 1.1: *Mutual influence of system behavior and domain dynamics.*

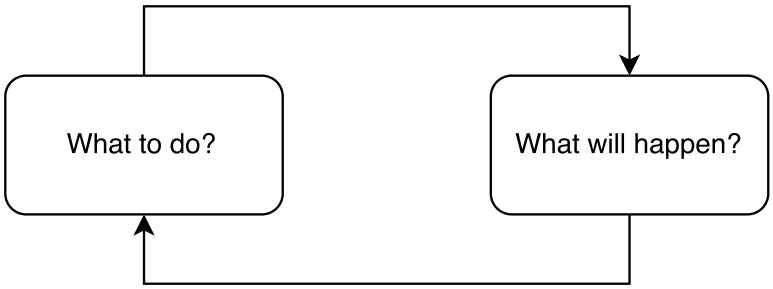


Figure 1.2: *Questions for autonomous systems w.r.t. behavior and domain dynamics.*

1.1.1 System Autonomy

The key to system autonomy is to define a whole *space* of behavioral solutions rather than a single one for the system to be executed at runtime. This space can be explored by the system at runtime according to its current situation, and potential traces in this space can be evaluated w.r.t. the system's goals. Concrete behavior is then compiled based on the results of search and evaluation.

In the face of uncertainty and change, there are two distinct problems to be solved in order to enable system autonomy.

1. A first question for a system that is provided with autonomy in the sense of a solution space is:

   What to do?

   We call the corresponding problem of answering this question the *decision problem*. For further information on decision making and automated planning, see e.g. [Bel57b, Bel57a, SB98, GNT04, WvO12]. The decision problem is further discussed in section 1.1.2.

2. Decision making is based on knowledge about the domain. This yields the second question to be answered by an autonomous system:

   What will happen?

   We call the corresponding problem of answering this question the *learning problem*. For further information on machine learning and automated learning of predictive models from data see e.g. [Mac03, WF05, Bis06, MCM13]. The learning problem is further discussed in section 1.1.3.

Figure 1.2 shows the two questions in the context of system behavior and domain dynamics (cf. Figure 1.1). We will outline the two corresponding problems in the following.

1.1.2 The Decision Problem

We will now discuss and formalize the problem of decision making in more detail. Let $\mathcal{S}$ denote the state space of a system, and let $\mathcal{A}$ denote its action space.

$\mathcal{S}$ typically denotes the belief state of a system. A system's belief state does not necessarily represent the real situation of the system. It rather represents a state that

the system considers to be true due to its past and current observations. For example, given a basic set of system configurations $\mathcal{W}$, a belief state in $\mathcal{S}$ may be the result of an observed history of configurations $\mathcal{W}^*$.

In general, actions in $\mathcal{A}$ are not restricted to represent single atomic actions. They may also represent action sequences, non-deterministic actions (i.e. action sets) or probability distributions over actions. That is, given a set of atomic actions $\mathcal{B}$, the set $\mathcal{A}$ could be a sequence of atomic actions $\mathcal{B}^*$, non-deterministic sets of actions $2^{\mathcal{B}}$ or probability distributions over actions $P(\mathcal{B})$.

Let $\mathcal{P}$ denote a system's behavioral policy space. A policy (also called strategy in some domains) is a function (or, more generally, a distribution, cf. Chapter 2) that maps states of the environment to system actions, thus defining a system's behavior.

$$\mathcal{P} : \mathcal{S} \to \mathcal{A} \tag{1.1}$$

Let $\mathcal{M}$ denote the space of models (i.e. knowledge) about domain dynamics. The decision problem is then equivalent to building a policy based on the model.

$$\mathcal{M} \to \mathcal{P} \tag{1.2}$$

This in itself is a very general formulation: For example, a system that acts randomly or one that does nothing at all would then provide a solution to this problem. Typically however, we will require our system to fulfill particular *goals*. Behavioral decisions should then exploit knowledge about the domain and system interaction capabilities to complete the task in a satisfactory manner. Let $\mathcal{M}$ and $\mathcal{P}$ be domain models and system policies as above; and let $\mathcal{G}$ denote system goals. Then, goal-driven decision making can be abstractly formulated as follows.

$$\mathcal{M} \times \mathcal{G} \to \mathcal{P} \tag{1.3}$$

Note that this implies that the same domain (i.e. $\mathcal{S}$ and $\mathcal{A}$) and the same knowledge about it (i.e. $\mathcal{M}$) potentially yield different system behavior (i.e. policies in $\mathcal{P}$) depending on the system goals $\mathcal{G}$. This in turn means that changing the system goals at runtime should result in accordingly adapted system behavior, given a sensible mechanism is used to solve the decision problem. This property of adaptation should also hold if all other components of the system (i.e. perceptions in $\mathcal{S}$, system interaction in $\mathcal{A}$, and knowledge about domain dynamics in $\mathcal{M}$) remain unchanged.

1.1.3 The Learning Problem

As has been stated above, in order to solve the decision problem a system has to be provided with some model about domain dynamics in $\mathcal{M}$ in order to compile appropriate behavior w.r.t. its goals in $\mathcal{G}$ at runtime. While this model can be designed manually and may incorporate expert knowledge about the domain, it is also possible to equip the system in question with the ability to observe the dynamics of its environment, and to learn a model from these observations on its own.

A system can observe the dynamics of its environment and the effects of its interactions by comparing its situation before and after execution of an action. More formally, let $\mathcal{S}$ and $\mathcal{A}$ be state and action spaces as before. Then, an observation of domain dynamics in $\mathcal{O}$ can be formulated as follows.

$$\mathcal{O} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S} \tag{1.4}$$

Such an observation triple consists of a state perceived by the system before execution of a particular action, the action itself, and the state that is perceived after execution of the action. Note that this formulation also allows to observe dynamics that happen without interaction of system by including a no-operation to the system's actions. In a sense this means that, as the environment proceeds in its state on itself, not performing an action is not possible.

Based on such observations about the environment and the system's interaction possibilities, the learning task is then to compile a set of observations into a model in $\mathcal{M}$ that can be used to solve the decision problem.

$$2^{\mathcal{O}} \rightarrow \mathcal{M} \tag{1.5}$$

In cases, the model may also be learned incrementally. This means that new observations are interpreted in context of an already existing model, yielding a new model that incorporates information about the new observations.

$$2^{\mathcal{O}} \times \mathcal{M} \rightarrow \mathcal{M} \tag{1.6}$$

Learning is typically done by finding statistical structure in the observed information, and by exploiting this structure in a way that allows to compress the observations. In an information theoretic sense, learning identifies conditions about states in $\mathcal{S}$ that allow predictions about observations which exhibit reduced entropy (or variance, respectively) in comparison the unconditionally observed data.

1.1.4 Interplay of Learning and Planning

There is a subtle mutual influence between the decision and the learning problem: Deciding on a policy depends on a model, which is the result of learning. On the other hand, learning is based on the observations made by the system, which in turn depend on the actions that a system executes. Figure 1.3 illustrates this mutual interplay.

1.2 Problem & Approach

While the decision and learning problems can be formulated abstractly as done in the previous Section, solving them in practice is a very involved task. The main challenge here is the combination of domain complexity and limited resources that are available for solving the decision and learning tasks.

Due to uncertainty and inherent non-determinism, typical application domains have very large or even infinite state and action spaces. Branching factors – i.e. the number of different potential outcomes of domain progression – are very large or even infinite for the same reasons. Exhaustive search and reasoning in the resulting space of behavioral possibilities will typically not yield a sensible result in reasonable time, or within reasonable computational cost. Also, behavioral reasoning has to be efficient in order to react in time to changing situations. In fact, if a system reasons about its optimal course of action for too long, the problem at hand may have changed before the optimal solution to the initial problem has been produced.

We will thus refine the initially stated question to be answered in this thesis by the notion of *efficiency* to reflect the additional challenges posed to system autonomy in very large, non-deterministic domains.

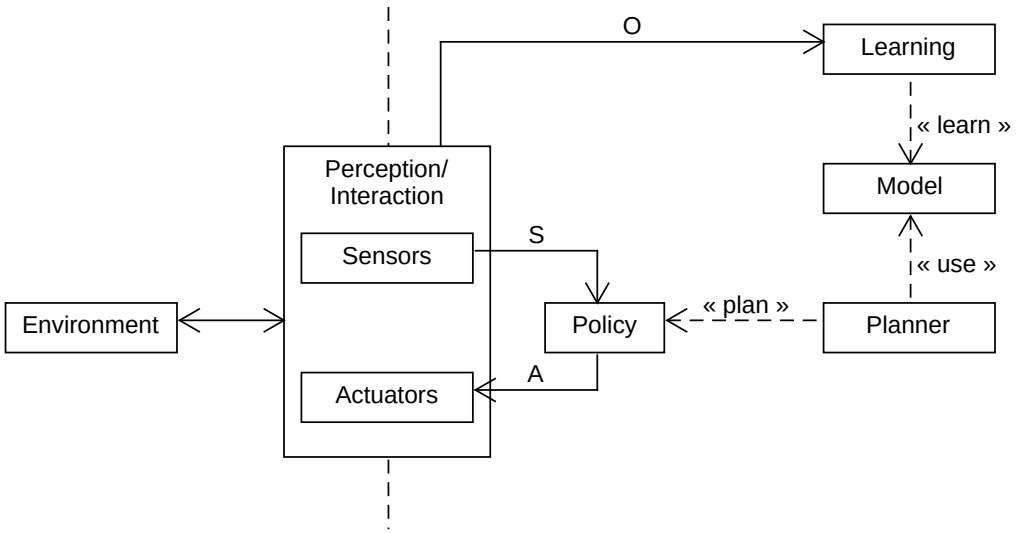


Figure 1.3: *Interplay of planning and learning based on observations about the dynamics of the environment.*

Problem Statement

How to build systems that autonomously, adequately, and efficiently react to change?

This thesis considers *approximation* as a key factor to successfully answer this question. The idea is that in cases where an *optimal* solution is not tractably computable, or where optimality changes too fast to be realized, it is valuable and often sufficient to generate a solution that is *as good as possible* given constrained computational resources.

To this end, we will consider three techniques to cope with the complexity of domains and the corresponding size of solution space:

1. We consider *online planning*, which parallelizes system execution and deliberation [KDMW12, KH13, Wei14]. The key idea is that executing an action reduces any uncertainty about its outcome. This in turn yields a valuable reduction of the search space. Additionally, online planning continuously incorporates new information about the potentially changing environment into the decision making process. We will further discuss online planning in section 1.2.1.

2. We consider *probabilistic representations* for policies and models in order to deal efficiently with the complexity of modern application domains as outlined above [Jay03, Dur10, Lee12]. That is, we explicitly incorporate information about uncertainty, and will use statistical methods to update (and, hopefully, reduce) this uncertainty based on information that is available to a system at runtime. By resorting to statistical representations and techniques, many problems that are unfeasible to solve optimally through exhaustive search become efficiently tractable. I.e., we are trading exactness and optimality for efficient approximation and solution quality that is based on available information and computational

resources. We will further discuss planning under uncertainty and corresponding probabilistic representations in section 1.2.2.

3. We consider simulation-based *Monte Carlo sampling* and *importance sampling* for efficiently evaluating particular behavioral choices of a system [KW08, Dia09, RK11, RK13, Ham13]. We do so by sampling potential consequences of behavior execution based on a simulation of the application domain. Importance sampling is an approach for concentrating simulation and evaluation effort to potential high-value regions of the search space. We will further discuss planning with Monte Carlo methods in section 1.2.3 and the application of importance sampling for planning in section 1.2.4.

1.2.1 Online Planning

In application domains that permanently change, it is necessary for an autonomous system to continuously adapt their behavior as a reaction to this change. *Online planning* is an approach to effectively deal with this setting, where efficiently finding behavioral solutions to permanently changing problems is a crucial factor for successful system execution. Here, planning and action execution are iteratively repeated in a parallel fashion. As in previous statements of the decision problem, the system's behavioral policy is optimized according to knowledge about the domain. However, in the online planning approach, the system performs this optimization step also based on information about the current state. This ensures that computational effort is focused on (a) problems that are currently to be solved and (b) potential future progress that is likely to happen, as the close future can typically be predicted more reliably from domain knowledge that events that could occur far in the future.

Another effect of parallelizing action and deliberation is that by executing actions, the uncertainty about their potential outcome is reduced by observing the action's effect. Given the observation is free of noise (an assumption that we make in this thesis), the uncertainty about the action's effect is even completely removed. This in turn yields a much smaller search space to be investigated and evaluated by an autonomous system in order to solve the current decision problem.

We reformulate the decision problem (cf. Equation 1.3) for the online planning setting to incorporate information about the current situation of the autonomous system.

$$\mathcal{S} \times \mathcal{M} \times \mathcal{G} \to \mathcal{P} \tag{1.7}$$

1.2.2 Planning under Uncertainty

We model our knowledge (and correspondingly, our uncertainty) about the application domain and the potential solution space by defining probability distributions for domain models in $\mathcal{M}$ and for behavioral policies $\mathcal{P}$.

1.2.2.1 Probabilistic Domain Models

To be more precise, the knowledge about domain dynamics is modeled as a probability distribution over states that is conditional w.r.t. a given state and a given action. Note that we represent probability distributions with a capital P, not to be confused with the calligraphic $\mathcal{P}$ used to denote policies.

$$\mathcal{M} \subseteq P(\mathcal{S}|\mathcal{S} \times \mathcal{A}) \tag{1.8}$$

This probability distribution encodes our uncertainty about probabilistic domain dynamics: It defines the distribution of potential successor states that are reached when executing a particular action in a particular state.

In general, this probability distribution is not required to be known explicitly. That is, it suffices to require a *simulation* of the application domain: If we settle on a particular sequence of actions, we are able to generate corresponding sequences of states from the distribution. The concrete probabilities (in numbers) associated with the observed transitions are not required to be specified explicitly. Typically, it suffices to approximate the distribution based on observations sampled from it as needed.

For many domains, such simulations are readily available or can be build from data. This also yields an interesting approach for solving the learning problem, as many modern machine learning techniques allow to build simulations from data based on statistical inference.

1.2.2.2 Probabilistic Behavioral Policies

As for the dynamics of domain progression, we encode our uncertainty about behavioral policies that are to be generated as a solution to the decision problem as probability distributions. More precisely, a policy is a conditional distribution of actions, depending on the system's current situation (i.e. state). Again, note that probability distributions are represented by a capital P, in contrast to policies that are denoted by the calligraphic $\mathcal{P}$.

$$\mathcal{P} \subseteq P(\mathcal{A}|\mathcal{S}) \tag{1.9}$$

1.2.2.3 Probabilistic Decision Making

A potential solution to the decision problem is to statistically refine an give probabilistic policy based on knowledge about the domain. In fact, we can extend online planning (cf. Equation 1.7) as follows to reflect the additional argument of a prior policy to be refined.

$$\mathcal{S} \times \mathcal{M} \times \mathcal{G} \times \mathcal{P} \to \mathcal{P} \tag{1.10}$$

In order to solve the decision problem in this setting, the key idea is to gather information about the current policy by probabilistically exploring the model of domain dynamics. This information can then in turn be used to adapt the policy in order to increase the quality of system behavior w.r.t. specified system goals.

Reward Functions In order to evaluate the quality of a particular system policy, and in order to make different behavioral choices qualitatively comparable, we define a function that maps states to real values. The more beneficial or valuable a state, the higher the value we assign to it should be. Correspondingly, unwanted or hazardous states should be provided with mappings to low values.

We can use this function to encode the system goals. For example, if particular properties of the state should be established and maintained by an autonomous system, we can provide some positive value to states that exhibit the corresponding property. The following formalizes the idea of providing a value to states for encoding system goals in terms of a so called *reward function* in R. Note that this reward function can be used to formalize system goals in $\mathcal{G}$.

$$R = \mathcal{S} \to \mathbb{R} \cup \{+\infty, -\infty\} \tag{1.11}$$

Execution Traces Given a reward function in R, the value we assign to a particular behavior is defined as the accumulated reward gathered while executing a particular policy in $\mathcal{P}$. I.e. the value of a single execution run of a policy is the sum of rewards assigned to the states that were visited in this run.

In this context, *system execution* means to perform the following steps iteratively.

1. Observe the current system state.
2. Sample an action from the policy w.r.t. current system state.
3. Execute the sampled action.

As system policies as well as domain dynamics are probabilistic, sequences of states encountered by system execution are as well probabilistically distributed. We denote the corresponding distribution by $\mathcal{X}$. The distribution depends conditionally on the initial state in $\mathcal{S}$ of execution and on the policy in $\mathcal{P}$ to be executed.

$$\mathcal{X} \subseteq P(\mathcal{S}^* | \mathcal{S} \times \mathcal{P}) \tag{1.12}$$

Policy Value We are interested in finding a policy that maximizes the *expected* accumulated reward. That is, we can usually not give guarantees about what will happen when following a certain behavioral policy, but we can evaluate a policy about its performance w.r.t. system goals accounting for the probabilistic, uncertain nature of application domain dynamics. We do so by defining as the value of a policy the expected sum of gathered reward from following this policy. As both policy and state progression are probabilistically distributed, also the expected gathered reward of a system executing a particular policy in a particular domain is a distribution. We denote this distribution of policy value by $\mathcal{V}$. Note that, as $\mathcal{X}$ is parameterized by policies in $\mathcal{P}$, also the distribution $\mathcal{V}$ is parameterized by a policy, together with a particular reward function in R.

$$\mathcal{V} \subseteq P(\mathbb{R} | \mathcal{X} \times R) \tag{1.13}$$

The goal of probabilistic decision making in a given current state $s \in \mathcal{S}$ is then to find a policy $p \in \mathcal{P}$ that maximizes the expected value of the corresponding policy value distribution, given some reward function $r \in R$. Note again that s and p are the conditioning parameters of the execution distribution in $\mathcal{X}$, which is in turn a conditioning parameter of the value distribution $\mathcal{V}$.

$$\underset{p \in \mathcal{P}}{\operatorname{argmax}} \, \mathbb{E}\left[\mathcal{V}(\cdot | s, p, r)\right] \tag{1.14}$$

Figure 1.4 informally sketches the relationship of behavioral policy choice and corresponding policy value distribution. The vertical axis represents potential behavioral choices of a planning agent. Note that, while being shown in one dimension only, the space of policies is typically of higher dimensionality. The vertical axis represents the distribution of corresponding policy values in $\mathcal{V}$ for a fixed reward function in $\mathcal{R}$. The shaded areas sketch the distribution, indicating different densities by color darkness.

Figure 1.5 informally illustrates the changing of policy values over time in dynamic domains. This change occurs due to (a) system interaction with the environment or (b) domain dynamics that take place independently of system interaction. Both types of change typically influence the value distributions of behavioral choices. Note that this implies that actions executed by a system at a given point in time influence the value of future decisions.

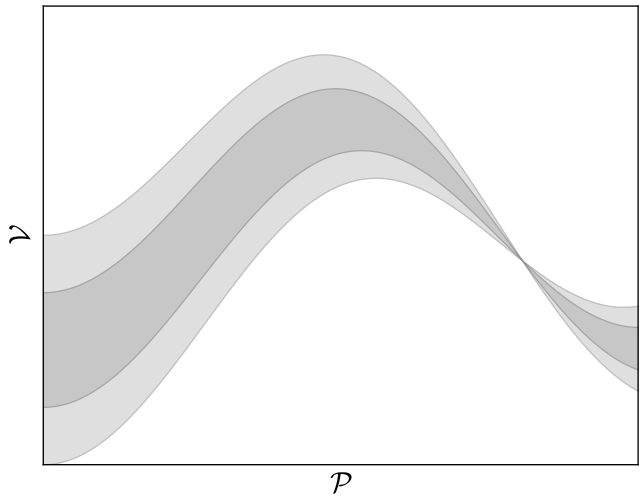


Figure 1.4: *Sketching an example value distribution in $\mathcal{V}$ w.r.t. parametrization by a behavioral policy in $\mathcal{P}$ at a particular time, given a fixed reward function (i.e. system goals). Estimating the value distribution solves the goal-based decision problem.*

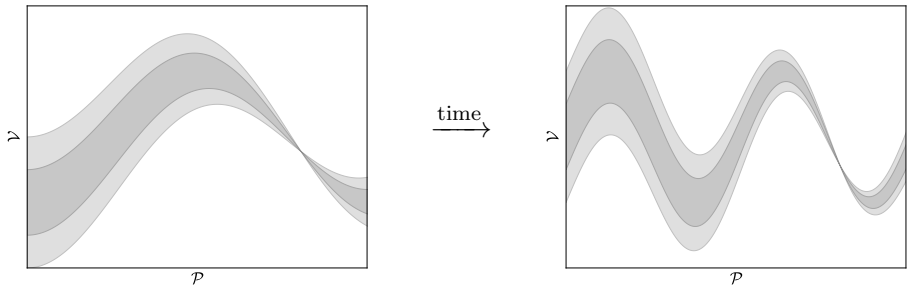


Figure 1.5: *Sketching change of the value distribution in $\mathcal{V}$ w.r.t. parametrization by a behavioral policy in $\mathcal{P}$ over time, given a fixed reward function. Note that typically the value distributions at a particular time t depend on future value distributions.*

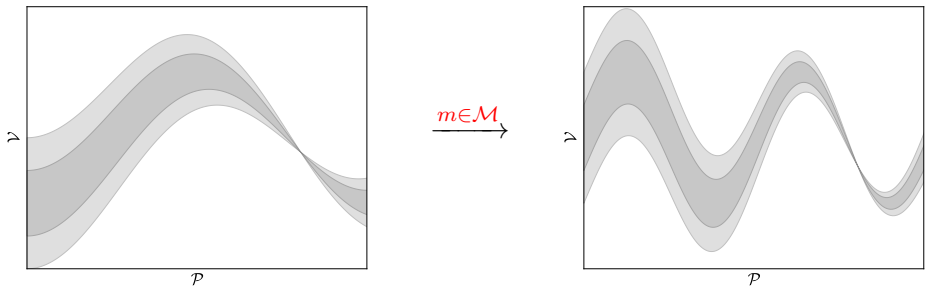


Figure 1.6: *Monte Carlo planning exploits a domain model $m \in \mathcal{M}$ to estimate the expected change of the value distribution in $\mathcal{V}$ over time.*

1.2.3 Monte Carlo Planning

The space of potential system policies and the consequences of their execution grows very fast with the dimensions of $\mathcal{S}$ and $\mathcal{A}$. In many domains, infinitely many potential system traces exist. Also, as the environment changes without interaction with system, assessment of current policy choices has to be efficient. I.e. decisions have to be made based on information about the current environment before the situation changes so much that this information becomes obsolete for decision making. Therefore, we resort to a statistical approach to evaluate the quality of particular behavioral choices: Monte Carlo sampling.

The key idea of using Monte Carlo sampling for planning is to exploit a simulation of the application domain to approximate a target value distribution w.r.t. behavioral choices efficiently. The probabilistic exploration of the solution space is realized by computing potential execution traces of the system in a simulation of the application domain. The simulated execution traces are distributed depending on the current state in $\mathcal{S}$ and the system policy in $\mathcal{P}$, but also on the model of domain dynamics in $\mathcal{M}$. We denote the resulting distribution of simulated sequences of states w.r.t. to a given policy, a given state and a given model by $\mathcal{X}_{\mathcal{M}}$.

$$\mathcal{X}_{\mathcal{M}} \subseteq P(\mathcal{S}^* | \mathcal{S} \times \mathcal{P} \times \mathcal{M}) \tag{1.15}$$

By analyzing elements sampled from $\mathcal{X}_{\mathcal{M}}$ for a given model and a given policy, we are able to measure how well the current policy satisfies the system goals based on information gathered from simulating many execution traces. This enables us to build an estimate of the real policy distributions in $\mathcal{V}$. We denote the corresponding estimated distribution of expected policy value by $\hat{\mathcal{V}}$.

$$\hat{\mathcal{V}} \subseteq P(\mathbb{R} | \mathcal{X}_{\mathcal{M}} \times R) \tag{1.16}$$

Figure 1.6 illustrates the idea of using a model in $\mathcal{M}$ to estimate future distributions of policy values. Here, the effect of passing time on the values of particular behavioral choices is determined by querying the available model of the domain dynamics. The left graph sketches the distribution at a given point in time, and a model $m \in \mathcal{M}$ is used to estimate a future distribution, which is sketched in the right graph.

In this setting, the task of Monte Carlo planning in a given current state $s \in \mathcal{S}$ can be stated as finding a policy $p \in \mathcal{P}$ that maximizes the expected value for a given model of domain dynamics $m \in \mathcal{M}$ and a given reward function $r \in \mathcal{R}$.

$$\underset{p \in \mathcal{P}}{\operatorname{argmax}} \, \mathbb{E}\left(\hat{\mathcal{V}}(\cdot | s, p, m, r)\right) \tag{1.17}$$

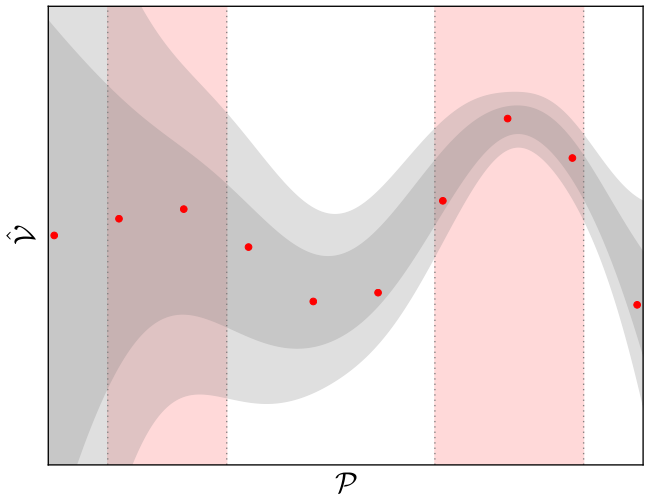


Figure 1.7: *Initial, equally distributed sampling from $\mathcal{P}$ yields an estimate in $\hat{\mathcal{V}}$ from correspondingly sampled policy values, shown as red dots. This enables identification of potential high-value regions of interest in $\mathcal{P}$, which are sketched as red shaded areas.*

1.2.4 Importance Sampling

As mentioned above, our domains of interest dynamically change their state over time. This in turn yields a change of the corresponding planning task, as the current state is a parameter of the distribution to be optimized. Therefore, we are interested in finding the policy with maximal expected value as efficiently as possible w.r.t. available information to be able react to changing situations as good as possible and in time.

As we are interested in finding the maxima, and not necessarily to estimate the whole value distribution, we apply *importance sampling* to concentrate estimation effort in regions that are expected provide high values w.r.t. previous samples from the policy space. Importance sampling works by iteratively performing the following steps.

1. Sample from a proposal distribution of interest over the solution space.

2. Evaluate the sampled solutions w.r.t. the target distribution.

3. Shift the proposal distribution to more likely generate solutions of high interest.

The initial proposal distribution should cover most or all of the solution space equally. In our setting, this means that initially, all potential behavioral choices should be equally considered for sampling. The interest of a solution or a region of solutions is defined by the expected value of the corresponding polices, which we determine by sampling a value of executing this policy (or policies from the region, respectively) through simulation. We then use the sampled values of policies to find high-value regions in the policy space. In the subsequent sampling step, we more likely sample and evaluate policies from the currently estimated high-value regions. By iterating this procedure, we concentrate our estimation effort in promising areas of the policy space, yielding an efficient solution to the Monte Carlo planning problem.

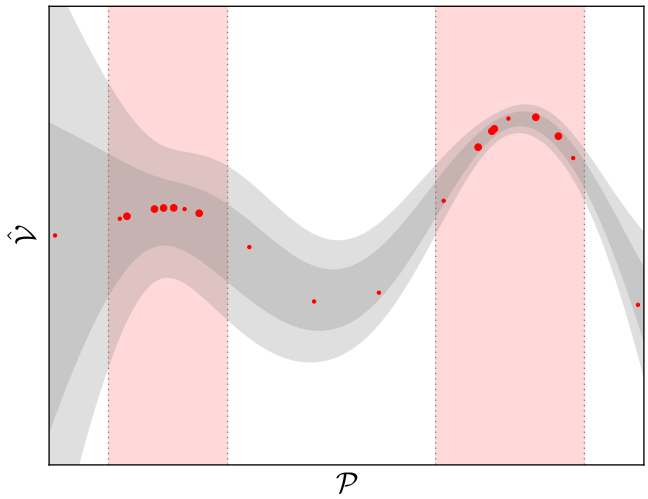


Figure 1.8: *Resampling from the high-value regions of interest in $\mathcal{P}$ yields a refined estimate in $\hat{\mathcal{V}}$ from correspondingly sampled policy values, shown as red dots. Samples from the previous iterations are shown as smaller dots. Resampling from high-value regions is likely to produce samples that yield a high value as well, thus resulting in a better estimate of high-value regions, indicated by smaller vertical extension of the estimated value distribution in the regions of interest.*

1.2.4.1 Properties of Simulation-Based Autonomous Systems

We summarize the main properties of the simulation-based approach to autonomous systems.

1. Simulation-based autonomy renders large state spaces with high branching factors tractable by resorting to statistical approaches rather than using exhaustive search and explicit symbolic reasoning. Using a simulation as for reasoning allows to sample information about potential future system traces statistically. Sampling is a feasible approach for estimating very complex probability distributions.

2. A black-box model of the domain is sufficient (see e.g. [BPW+12]). That is, a simulation of the domain suffices for applying the simulation-based approach to system autonomy, even without explicit knowledge about the internal dynamics of the simulation. These black-box simulations can be used and integrated straightforwardly to enable autonomous adaptation for systems operating in these domains. Simulations can straightforwardly be implemented in widespread general purpose languages, rendering simulation-based techniques widely applicable.

3. Simulations of domain dynamics can also be learned from runtime observations by statistical classification or regression algorithms (see Chapter 4). This allows predefined simulation models to be refined at system runtime according to the actual domain dynamics.

4. A simulation-based approach to engineering system autonomy is simple and general. Without providing empirical evidence, we argue that simplicity and generality render the simulation-based approach to system autonomy understandable, easily communicated and thus widely adoptable and applicable.

5. Simulation-based system autonomy is scalable. The quality of solutions determined by online planning algorithms typically increases with the resources (e.g. computation time, parallelization, etc.) it is given to run simulations in order to aggregate information that is used for goal-based decision making. In cases, this property can even be proved mathematically. The corresponding algorithms are said to be *anytime optimal* (see e.g. [SV10]).

The approximative nature of the simulation-based approach to system autonomy yields the following restrictions.

1. Results are approximate and based on empirical evidence rather than rigorous symbolic deduction. This means that results are only valid up to some level of statistical confidence. However, due to the inherent uncertainty of modern application domains, we consider it reasonable, if not necessary, to deal with this uncertainty explicitly and to exploit it in the reasoning process.

2. Simulation is bounded by a finite horizon. In dynamically changing domains resources for planning are limited, as pursuing particular goals in these domains require action regularly. Therefore, it seems to be a valid approach to put most deliberation effort to solve problems that are temporally local (i.e. in the near future), and that are probable to happen.

3. Simulation-based optimization is tractable and efficient for a limited number of features. Scaling algorithms to high-dimensional or sparse optima feature spaces is currently subject to active research (see e.g. [ANW12, BB12, GMT14]).

1.3 Example Domains

This thesis provides an approach to simulation based system autonomy that is applicable to both discrete and continuous problem domains. Nevertheless, particular instantiations of the approach vary depending on the representation of the problem domain. For illustration of the ideas developed in this thesis, we introduce two problem domains, one discrete and the other continuous. In both cases, we settle with a search and rescue scenario where an agent has to find some target victims in an accident site.

1.3.1 Discrete Example Domain

Figure 1.9 shows a class diagram of the discrete example search and rescue scenario. A number of arbitrarily connected positions defines the domain's topology. At some positions there is an ambulance (pos.safe = true). Positions may be on fire, except those that host an ambulance, i.e. class *Position* has the following invariant: pos.safe implies not(pos.fire) for all pos $\in$ *Position*. Fires ignite or cease probabilistically depending on the number of fires at connected neighbor positions. A position may host any number of robots and victims. A robot can carry a number of victims that is bounded by its capacity. A carried victim does not have a position. A robot has five types of actions available.

1. Do nothing.

2. Move to a neighbor position that is not on fire.

3. Extinguish a fire at a neighbor position.

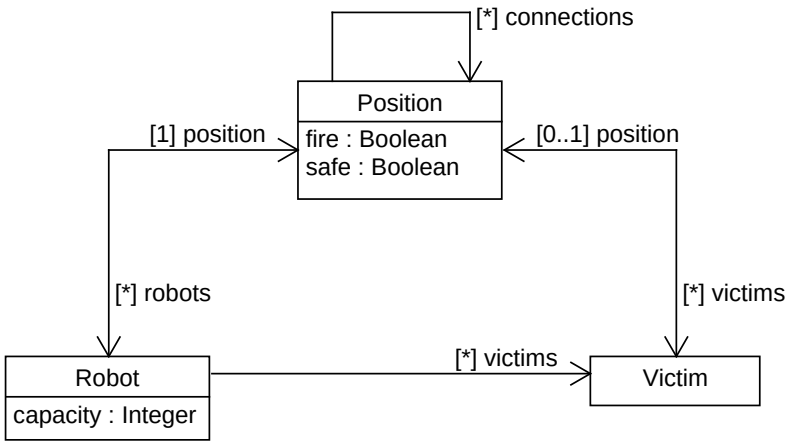


Figure 1.9: *Class diagram of the example domain.*

4. Pick up a victim at the robot's position if capacity is left.

5. Drop a carried victim at the robot's position.

All actions have unit duration. Each action may fail with a certain probability, resulting in no effect. Note that the number of actions available to a robot in a particular situation may vary due to various possible instantiations of action parameters (such as the particular victim that is picked up or the concrete target position of movement).

Figure 1.10 shows an example situation in the scenario. Positions and their corresponding connections are shown as graph. There are fires at certain positions, and stylized persons depict victims at their locations (i.e. their positions). Also shown are safe positions (indicated by ambulances), and the robot agent.

1.3.2 Continuous Example Domain

Figure 1.11 depicts our continuous search and rescue scenario. The circle bottom left represents our agent. Dark rectangular areas are static obstacles, and small boxes are victims to be collected by the planning agent. The agent is provided with unit reward on collecting a victim. Victims move with Gaußian random motion (i.e. their velocity and rotation are randomly changed based on a normal distribution). Note that this yields a highly fluctuating value function of the state space – a plan that was good a second ago could be a bad idea to realize a second later. This means that information aggregation from simulations should be as efficient as possible to be able to react to these changes in real time.

An agent can perform an action by first rotating for a second and then moving forward for the same amount of time. Rotation rate and movement speed are action parameters to be optimized by the planner in order to collect the victims as fast as possible. The agent is provided with a simulation of the environment as described above. Note that this simulation is an abstraction of the real environment. This means that reality and simulation may differ in their dynamics, even if performing the exactly same set of actions. Also, the simulation is not informed about the movement model of the victims.

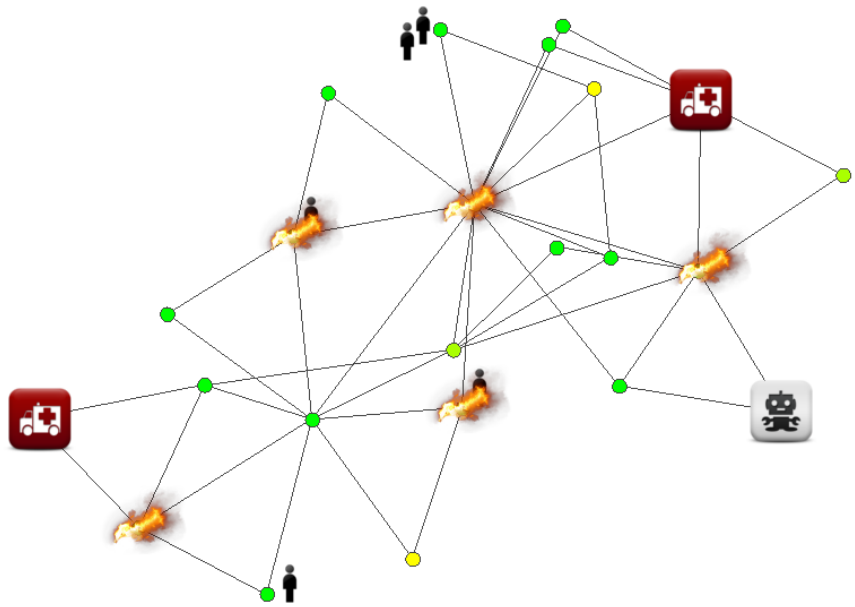


Figure 1.10: *A situation in the discrete example domain. Positions and their corresponding connections are shown as graph. There are fires at certain positions, and stylized persons depict victims at their locations (i.e. their positions). Also shown are safe positions (indicated by ambulances), and the robot agent.*

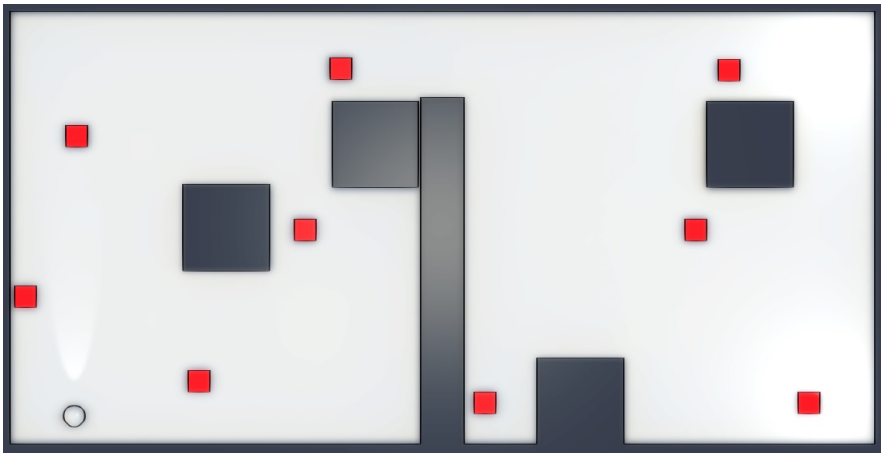


Figure 1.11: *A situation in the continuous example domain. Grey areas are static obstacles. Small boxes represent victims with Gaußian motion. The agent is the circle bottom left.*

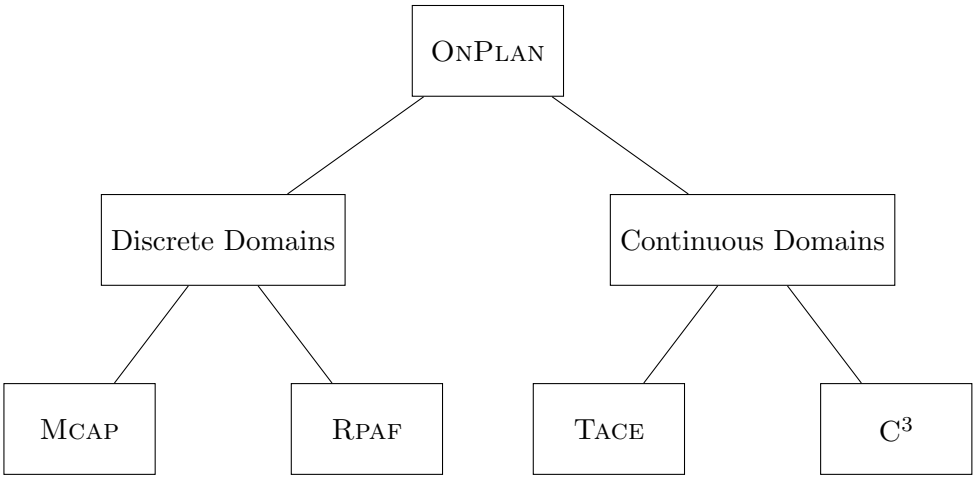


Figure 1.12: *Overview of contributions.*

1.4 Contributions

This Section outlines the technical contributions of the thesis, and discusses the author's individual contributions w.r.t. content that has been previously published. Figure 1.12 provides an overview of the contributions. The OnPlan framework provides a top-level framework for engineering simulation-based autonomous systems based on importance sampling at runtime. Based on this framework, we distinguish particular contributions for discrete and continuous state, action and time domains.

For discrete domains, the thesis comprises the following contributions:

1. Monte Carlo Action Programming (Mcap) provides a non-deterministic procedural programming language to define the potential behavioral space based on expert knowledge.

2. Relational Probabilistic Action Forests (Rpaf) are a machine learning approach to generating simulations about domains dynamics from runtime observations.

For continuous domains, the thesis comprises the following contributions:

1. Time-Adaptive Cross Entropy Planning (Tace) extends the existing online planner Ceolp [WL13, Wei14] by incorporating temporal information into the planning process.

2. Continuous Time Cross Entropy Control (C^3) is an approach that parallelizes execution and deliberation for online planning real-time systems.

The particular contributions are briefly described in the following sections.

1.4.1 A Framework for Simulation-Based Autonomy

The thesis proposes the OnPlan framework for modeling autonomous systems operating in domains with large probabilistic state spaces and high branching factors. The framework defines components for acting and deliberation, and specifies their interactions. It comprises a mathematical specification of requirements for autonomous systems. We discuss the role of such a specification in the context of simulation-based online planning. We also consider two instantiations of the framework: Monte Carlo

Tree Search for discrete domains, and Cross Entropy Open Loop Planning for continuous state and action spaces. The framework's ability to provide system autonomy is illustrated empirically on a robotic rescue example.

Previous Publication and Author's Individual Contribution The OnPlan framework has been previously published in [BHW15] by the author together with Rolf Hennicker and Martin Wirsing. The author contributed idea, concept, structure and writing of the paper, as well as implementation and experimental evaluation. The co-authors engaged in discussion and helped the author with the formalization of the OnPlan framework.

1.4.2 Monte Carlo Action Programming

The thesis proposes Monte Carlo Action Programming (Mcap), a programming language framework for autonomous systems that act in large probabilistic state spaces with high branching factors. The language is used to constrain the behavioral solution space of an online planning agent. It comprises formal syntax and semantics of a nondeterministic action programming language. The language is interpreted stochastically via Monte Carlo Tree Search. Effectiveness of the approach is shown empirically.

Previous Publication and Author's Individual Contribution Monte Carlo Action Programming has not been previously published. It is based on ideas on decision-theoretic action programming formulated in the framework of rewriting logic that have been previously published in [Bel13] and [Bel14]. The author has developed and published these ideas on his own. Monte Carlo Action Programming extends the previously published ideas by performing program interpretation stochastically via Monte Carlo Tree Search.

1.4.3 Relational Probabilistic Action Forests

The thesis proposes *Relational Probabilistic Action Forests* (Rpaf), an approach for learning probabilistic predictive action models for relational data with decision forests. Performance of online planning correlates with predictive quality of the used model. A predictive domain model manually specified at design time may yield poor predictive quality at runtime, either due to specification errors or due to unexpected change. Learning a predictive model from runtime observations allows to identify and recover from predictive inaccuracy due to erroneous specification or unexpected change. Relational Probabilistic Action Forests enable generalization over discrete relational training data, yielding fast learning rates by exploitation of relational structure. The effectiveness of the approach is demonstrated empirically.

Previous Publication and Author's Individual Contribution Relational Probabilistic Action Forests have been previously published in [BNar] by the author together with Alexander Neitz. The author of this thesis contributed the idea for the approach, structure and writing of the published paper, and the formalization of the approach. Work on conceptual details, implementation and experimental evaluation were realized by both authors. Alexander Neitz studied preliminary ideas on the approach in his Bachelor thesis, which was supervised by the author of the thesis on hand.

1.4.4 Time-Adaptive Cross Entropy Planning

The thesis proposes *Time-Adaptive Cross Entropy Planning* (TACE) to increase flexibility of online planning agents in continuous state, action and time domains with infinite state-action spaces and branching factors. TACE reduces simulation effort of planning with cross entropy optimization by maintaining and adapting a probability distribution over the optimal planning horizon. This allows to identify temporally local problems in a global context, and to subsequently concentrate on the solution of the local problem. We show the effectiveness of TACE by comparing it empirically to a state-of-the-art online planner for continuous domains.

Previous Publication and Author's Individual Contribution Time-Adaptive Cross Entropy Planning has been previously published by the author in [Belar]. The contribution is completely work of his own.

1.4.5 Continuous Time Cross Entropy Control

The thesis proposes *Continuous Time Cross Entropy Control* (C^3), an online planning approach for continuous real-time planning. The key idea is to maintain particular points in time around which action parameters are optimized, and to move these points according to time that passes in the course of system execution. C^3 enables real parallelization of deliberation and action in real-time domains, and proposes to exploit timing as a first class property of real-time planning systems. The approach is instantiated in the context of cross-entropy planning (while being applicable to other stochastic optimization techniques) and its effectiveness is shown empirically.

Previous Publication and Author's Individual Contribution Continuous Time Cross Entropy Control has not been previously published. It is extends the idea of maintaining and optimizing action durations from the TACE approach to continuous flows of time.

1.5 Outline

Chapter 2 introduces the ONPLAN framework for simulation-based online planning. We discuss a mathematical formalization of the framework, and its realization in terms of a component model. We illustrate the effectiveness of the approach with two instantiations of the framework: Monte Carlo Tree Search in discrete domains, and Cross Entropy Open Loop Planning in continuous domains.

In Chapter 3, Monte Carlo Action Programming is proposed as a method to reduce the solution space for autonomous systems by incorporating expert domain knowledge into the reasoning process. This knowledge is specified in terms of a procedural nondeterministic action programming language that is interpreted stochastically by Monte Carlo Tree Search.

In Chapter 4, we discuss Relational Probabilistic Action Forests as a technique to learn simulations from observations of domain dynamics in discrete domains. Relational Probabilistic Action Forests use decision forests for generalizing from concrete observations, and expose additional generalization capabilities by exploiting relational structure of observations.

Chapter 5 proposes Time-Adaptive Cross Entropy Planning as a simulation-based online planning approach for solving the decision problem in continuous real-time domains. By incorporating time and in particular timing as features to be optimized besides other action parameters, more flexible solutions can be generated by autonomous systems. Based on the idea of timing, we also discuss Continuous Time Cross Entropy Control in this Chapter.

Chapter 6 concludes this thesis with a discussion of limitations and possibilities of simulation-based autonomous systems and outlines future lines of research in the field.

Chapter 2

The OnPlan Framework

Modern application domains such as machine-aided robotic rescue operations require software systems to cope with uncertainty and rapid and continuous change at runtime. The complexity of application domains renders it impossible to deterministically and completely specify the knowledge about domain dynamics at design time. Instead, high-level descriptions such as probabilistic predictive models are provided to the system that give an approximate definition of chances and risks inherent to the domain that are relevant for the task at hand.

Also, in contrast to classical application domains, in many cases there are numerous different ways for a system to achieve its task. Additionally, the environment may rapidly change at runtime, so that completely deterministic behavioral specifications are likely to fail. Thus, providing a system with the ability to compile a sensible course of actions at runtime from a high-level description of its interaction capabilities is a necessary requirement to cope with uncertainty and change.

One approach to deal with this kind of uncertain and changing environments is *online planning*. It enables system autonomy in large (or even infinite) state spaces with high branching factors by interleaving planning and system action execution (see e.g. [KDMW12, KH13, Wei14]). In many domains, action and reaction are required very often, if not permanently. Resources such as planning time and computational power are often limited. In such domains, online planning replaces the requirement of absolute optimality of actions with the idea that in many situations it is sufficient and more sensible to conclude as much as possible from currently available information within the given restricted resources. One particular way to perform this form of rapid deliberation is based on simulation: The system is provided with a generative model of its environment. This enables it to evaluate potential consequences of its actions by generating execution traces from the generative model. The key idea to scale this approach is to use information from past simulations to guide the future ones to directions of the search space that seem both likely to happen and valuable to reach.

In this Chapter we propose the OnPlan framework for modeling autonomous systems operating in domains with large or infinite probabilistic state spaces and high branching factors.[1] The remainder of the Chapter is outlined as follows. In Section 2.1 we introduce the OnPlan framework for online planning, define components for

[1] This Chapter is based on a previous publication of the OnPlan framework by the author together with Rolf Hennicker and Martin Wirsing [BHW15]. The author contributed idea, concept, structure and writing of the previously published paper, as well as implementation and experimental evaluation. The co-authors engaged in discussion and helped the author with the formalization of the OnPlan framework.

acting and deliberation, and specify their interactions. We then extend this framework to simulation-based online planning. In Sections 3.1.1 and 2.3 we discuss two instantiations of the framework: Monte Carlo Tree Search for discrete domains (Section 3.1.1), and Cross Entropy Open Loop Planning for continuous state and action spaces (Section 2.3). We illustrate each with empirical evaluations on a robotic rescue example. Section 5.6 concludes the Chapter and outlines potential lines of further research in the field.

2.1 A Framework for Simulation-Based Online Planning

In this Section we propose the OnPlan framework for modeling autonomous systems based on online planning. We introduce the basic concept in Section 2.1.1. In Section 2.1.2, we will refine the basic framework to systems that achieve autonomy performing rapidly repeated simulations to decide on their course of action.

2.1.1 Online Planning

Planning is typically formulated as a search task, where search is performed on sequences of actions. The continuously growing scale of application domains both in terms of state and action spaces requires techniques that are able to (a) reduce the search space effectively and (b) compile as much useful information as possible from the search given constrained resources. Classical techniques for planning have been exhaustively searching the search space. In modern application scenarios, the number of possible execution traces is too large (potentially even infinite) to get exhaustively searched within a reasonable amount of time or computational resources.

The key idea of online planning is to perform planning and execution of an action iteratively at runtime. This effectively reduces the search space: A transition that has been executed in reality does not have to be searched or evaluated by the planner any more. Online planning aims at effectively gathering information about the *next* action that the system should execute, exploiting the available resources such as deliberation time and capabilities as much as possible. Algorithm 1 captures this idea informally. In the following, we will introduce the OnPlan framework that formalizes the idea of online planning.

Algorithm 1 Online Planning (Informally)

```
1: while true do
2:     observe state
3:     plan action
4:     execute action
5: end while
```

2.1.1.1 Framework Specification

The OnPlan framework is based on the following requirements specification.

1. A set $\mathcal{S}_{\text{real}}$ which represents states of real environments. While this is a part of the mathematical formulation of the problem domain, it is not represented by a software artifact in the framework.

2. A set *Agent* that represents deliberating and acting entities.

3. Representations of the agent's observable state space $\mathcal{S}$ and the agent's action space $\mathcal{A}$. The observable state space $\mathcal{S}$ represents information about the environment $\mathcal{S}_{\text{real}}$ that is relevant for an agent and its planning process. It is in fact an abstraction of the environment.

4. A function $observe : Agent \times \mathcal{S}_{\text{real}} \to \mathcal{S}$ that specifies how an agent perceives the current state of its environment. This function defines the abstraction and aggregation of information available to an agent in its real environment to an abstract representation of currently relevant information. In some sense, the function *observe* comprises the monitor and analyze phases of the MAPE-K framework for autonomous computing [Kep03, KC03].

5. A function $actionRequired : Agent \times \mathcal{S} \to Bool$ that is used to define triggering of action execution by an agent. A typical example is to require execution of an action after a certain amount of time has passed since the last executed action.

6. For each action in $\mathcal{A}$, we require a specification of how to execute it in the real domain. To this end, the framework specification comprises a function $execute : \mathcal{A} \times \mathcal{S}_{\text{real}} \to \mathcal{S}_{\text{real}}$. This function defines the real (e.g. physical) execution of an agent's action.

7. We define a set *RewardFunction* of reward functions of the form $R : \mathcal{S} \to \mathbb{R} \cup \{+\infty, -\infty\}$. A reward function is an encoding of the system goals. States that are valuable should be mapped to high values by this function. States that should be avoided or even are hazardous should provide low values.

8. We define a set *Strategy* of strategies (also called policies, cf. Chapter 1). Each strategy is a probability distribution $P_{\text{act}}(\mathcal{A}|\mathcal{S})$ of actions over states. In the following, we will often omit the signature and simply write P_{act} for $P_{\text{act}}(\mathcal{A}|\mathcal{S})$. It defines the probability that an agent executes a particular action in a given state. If an agent $a \in Agent$ in state $s_{\text{current}} \in \mathcal{S}$ is required to act (i.e. when $actionRequired(a, s_{\text{current}})$ returns true) then the action that is executed is sampled from the distribution: $a \sim P_{\text{act}}(\cdot|s_{\text{current}})$, where $P_{\text{act}}(\cdot|s_{\text{current}})$ denotes the probability distribution of actions in state s_{current} and $\sim$ denotes sampling from this distribution. Sampling can be seen as non-deterministic choice proportional to a distribution.

9. A set *Planner* of planning entities. Planning is defined by a function $plan : Planner \times \mathcal{S} \times RewardFunction \times Strategy \to Strategy$. A planning entity refines its strategy P_{act} w.r.t. its currently observed abstract state and a reward function to maximize the expected cumulative future reward. It is usually defined as the sum of rewards gathered when following a strategy.

2.1.1.2 Framework Model

Figure 2.1 shows a class diagram for the OnPlan framework derived from the mathematical specification. It comprises classes for the main components *Agent* and *Planner*. States and actions are also represented by a class each: states $s \in \mathcal{S}$ are represented by objects of class *State*, actions $a \in \mathcal{A}$ by objects of class *Action*. Probability distributions of actions over states (defining potential agent strategies) are modeled by the

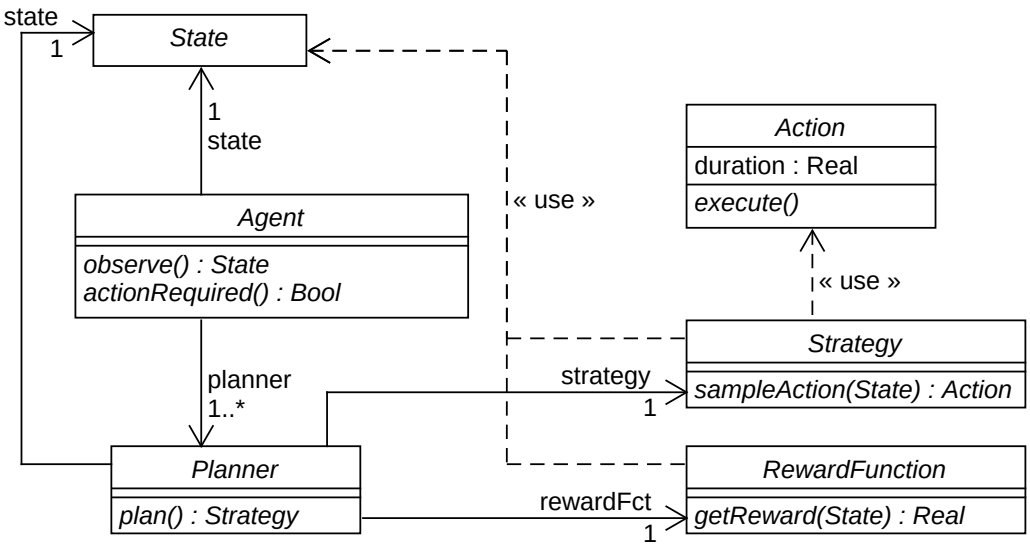


Figure 2.1: *Basic components of* OnPlan

class *Strategy*. Reward functions are represented by objects of class *RewardFunction*. All classes are abstract and must be implemented in a concrete online planning system.

Note that OnPlan supports multiple instances of agents to operate in the same domain. While inter-agent communication is not explicitly expressed in the framework, coordination of agents can be realized by emergent system behavior: As agents interact with the environment, the corresponding changes will be observed by other agents and incorporated into their planning processes due to the online planning approach.

2.1.1.3 Component Behavior

Given the specification and the component model, we are able to define two main behavioral algorithms for *Agent* and *Planner* that are executed in parallel: *Agent* ‖ *Planner*. I.e., this design decouples information aggregation and execution (performed by the agent) from the deliberation process (performed by the planner).

Algorithms 2 and 3 show the behavior of the respective components. We assume that all references shown in the class diagram have been initialized. Both behaviors are infinitely looping. An agent observes the (real) environment, encodes the observation to its (abstract) state and passes the state to its corresponding planning component, as long as no action is required (Algorithm 2, lines 2–5). When an action is required – e.g. due to passing of a certain time frame or occurrence of a particular situation/event – the agent queries the planner's current strategy for an action to execute (line 6). Finally, the action proposed by the strategy is executed (line 7) and the loop repeats.

The behavior of the planning component (Algorithm 3) repeatedly calls a particular planning algorithm that refines the strategy w.r.t. the current state and the specified reward function. We will define a particular class of planning algorithms in more detail in Section 2.1.2.

Algorithm 2 Agent Component Behavior

Require: Local variable action : *Action*

```
1: while true do
2:     while !actionRequired() do
3:         state ← observe()                          ▷ observe environment
4:         planner.state ← state                      ▷ inform planner
5:     end while
6:     action ← planner.strategy.sampleAction(state)  ▷ sample from strategy
7:     action.execute()                               ▷ execute sampled action
8: end while
```

Algorithm 3 Planner Component Behavior

```
1: while true do
2:     strategy ← plan()
3: end while
```

2.1.1.4 Framework Plug Points

The OnPlan framework provides the following plug points derived from the mathematical specification. They are represented by abstract operations such that domain specific details have to be implemented by any instantiation.

1. The operation *Agent::observe() : State*. This operation is highly dependent on the sensory information available and is therefore implemented in a framework instantiation.

2. The operation *Agent::actionRequired() : Bool*. The events and conditions that require an agent to act are highly depending on the application domain. The timing of action execution may even be an optimization problem for itself. The state parameter of the mathematical definition is implicitly given by the reference of an agent to its state.

3. The operation *Action::execute()*. Action execution is also highly dependent on technical infrastructure and physical capabilities of an agent.

4. The operation *RewardFunction::getReward(State) : Real*. Any concrete implementation of this operation models a particular reward function.

5. The operation *Strategy::sampleAction(State)* : *Action* should realize sampling of actions from the strategy w.r.t. to a given state. It depends on the used kind of strategy, which may be discrete or continuous, unconditional or conditional, and may even be a complex combination of many independent distributions.

6. Any implementation of the operation *Planner::plan()* should realize a concrete algorithm used for planning. Note that the arguments of the function *plan* from the mathematical specification are modeled as references from the *Planner* class to the classes *State*, *RewardFunction* and *Strategy*. We will discuss a particular class of simulation-based online planners in the following Section 2.1.2.

2.1.2 Simulation-Based Online Planning

We now turn our focus on a specific way to perform online planning: *simulation based online planning*, which makes use of a simulation of the domain. It is used by the planner to gather information about potential system episodes (i.e. execution traces). Simulation provides information about probability and value of the different state space regions, thus guiding system behavior execution. After simulating its possible choices and behavioral alternatives, the agent executes an action (in reality) that performed well in simulation. The process of planning using information from the simulation and action execution is iteratively repeated at runtime, thus realizing online planning.

A simple simulation based online planner would generate a number of randomly chosen episodes and average the information about the obtained reward. However, as it is valuable to generate as much information as possible with given resources, it is a good idea to guide the simulation process to high value regions of the search space. Using variance reduction techniques such as importance sampling, this guidance can be realized using information from previously generated episodes [Has70, RK11, RK13].

2.1.2.1 Framework Specification

In addition to the specification from Section 2.1.1, we extend the OnPlan framework requirements to support simulation-based online planning.

1. For simulation based planning, actions $a \in \mathcal{A}$ require a duration parameter. If no such parameter is specified explicitly, the framework assumes a duration of one for the action. We define a function $d : \mathcal{A} \to \mathbb{R}$ that returns the duration of an action. Any suitable time metric may be used (e.g. ms, sec, etc.).

2. OnPlan requires a set *Simulation* of simulations of the environment. Each simulation is a probability distribution of the form $P_{\text{sim}}(\mathcal{S}|\mathcal{S} \times \mathcal{A})$. It takes the current state and the action to be executed as input, and returns a potential successor state according to the transition probability. Simulating the execution of an action $a \in \mathcal{A}$ in a state $s \in \mathcal{S}$ yields a successor state $s' \in \mathcal{S}$. Simulation is performed by sampling from the distribution P_{sim}: $s' \sim P_{\text{sim}}(\cdot|(s, a))$, where $P_{\text{sim}}(\cdot|(s, a))$ denotes the probability distribution of successor states when executing action a in state s and $\sim$ denotes sampling from this distribution. Note that the instantiations of the framework we discuss in Sections 3.1.1 and 2.3 work with a fixed simulation of the environment. It does not change in the course of system execution, in contrast to the strategy.

3. We require a set $\mathit{SimPlanner} \subseteq \mathit{Planner}$ of simulation based planners.

4. Any simulation based planner defines a number $e_{\max} \in \mathbb{N}^+$ of episodes generated for each refinement step of its strategy.

5. Any simulation based planner defines a maximum planning horizon $h_{\max} \in \mathbb{N}^+$ that provides an upper bound to its simulation depth. A low planning horizon results in fast but shallow planning – long term effects of actions are not taken into account when making a decision. The planning horizon lends itself to be dynamically adapted, providing flexibility by allowing to choose between fast and shallow or more time consuming, but deep planning taking into account long term consequences of actions.

6. Any simulation based planner defines a discount factor $\gamma \in [0;1]$. This factor defines how much a planner prefers immediate rewards over long term ones when refining a strategy. The lower the discount factor, the more likely the planner will build a strategy that obtains reward as fast as possible, even if this means an overall degradation of payoff in the long run. See Algorithm 5 for details on discounting.

7. We define a set $\mathcal{E} \subseteq (\mathcal{S} \times \mathcal{A})^*$ of episodes to capture simulated system execution traces. We also define a set $\mathcal{E}_w \subseteq \mathcal{E} \times \mathbb{R}$ of episodes weighted by the discounted sum of rewards gathered in an execution trace. The weight of an episode is defined as its cumulative discounted reward, which is given by the recursive function $R_{\mathcal{E}} : \mathcal{E} \to \mathbb{R}$ as shown in Equation 2.1. Let $s \in \mathcal{S}$, $a \in \mathcal{A}$, $e, e' \in \mathcal{E}$ where $e = (s, a) :: e'$ (with :: denoting list concatenation), and let $R : \mathcal{S} \to \mathbb{R}$ be a reward function.

$$\begin{aligned} R_{\mathcal{E}}(\text{nil}) &= 0 \\ R_{\mathcal{E}}(e) &= R(s) + \gamma^{d(a)} R_{\mathcal{E}}(e') \end{aligned} \tag{2.1}$$

   An element of $\mathcal{E}_w$ is then uniquely defined by $(e, R_{\mathcal{E}}(e))$.

8. In the OnPlan framework, the simulation-based planner uses the simulation P_{sim} to generate a number of episodes. The resulting episodes are weighted according to rewards gathered in each episode, w.r.t. the given reward function of the planner. Simulation is driven by the current strategy P_{act}. The following function reflects this process.

$$\mathit{generateEpisode} : \mathit{SimPlanner} \times \mathit{Simulation} \times \mathit{Strategy} \times \mathit{RewardFunction} \to \mathcal{E}_w$$

9. Importance sampling in high value regions of the search space is realized by using the resulting weighted episodes to refine the strategy such that its expected return (see Section 2.1.1) is maximized. The goal is to incrementally increase the expected reward when acting according to the strategy by gathering information from simulation episodes in an efficient way. This updating of the strategy is modeled by the following function.

$$\mathit{updateStrategy} : \mathit{SimPlanner} \times 2^{\mathcal{E}_w} \times \mathit{Strategy} \to \mathit{Strategy}$$

2.1.2.2 Framework Model

Using mathematically justified approaches for strategy refinement provides a solution to the notorious exploration-exploitation tradeoff (see e.g. [AMS09]): While learning (or planning), an agent has to decide whether it should exploit knowledge about high-value regions of the state space, or whether it should use its resources to explore previously unknown regions to potentially discover even better options. We will discuss two instances of OnPlan that provide principled and mathematically founded methods that deal with the question where to put simulation effort in Sections 3.1.1 and 2.3.

Figure 2.2 shows the components of the OnPlan framework for simulation-based online planning. It comprises the components of the basic OnPlan framework (Section 2.1.1), and additionally defines a specialization *SimPlanner* of the *Planner* class, and a class *Simulation* that models simulations of the form P_{sim}. The parameters $e_{\max}, h_{\max}$ and γ are modeled as attributes of the *SimPlanner* class. We further assume a class

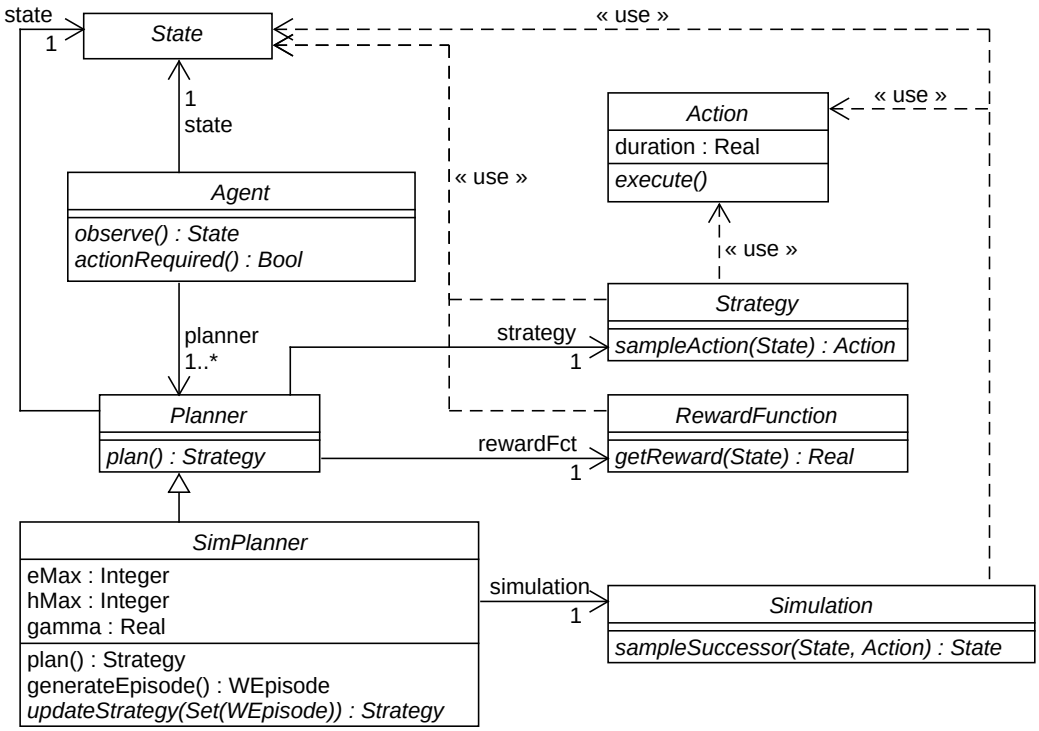


Figure 2.2: *Components of the* OnPlan *framework*

WEpisode that models weighted episodes. As this is a pure data container, it is omitted in the class diagram shown in Figure 2.2.

The *SimPlanner* class also provides two concrete operations. The operation *SimPlanner::plan() : Strategy* realizes the corresponding abstract operation of the *Planner* class and is a template method for simulation based planning (see Algorithm 4). Episodes are modeled by a type *Episode*, weighted episodes by a type *WEpisode* respectively. The function *generateEpisode* is realized by the concrete operation *generateEpisode() : WEpisode* of the *SimPlanner* class and used by the *plan* operation. The function *updateStrategy* from the mathematical specification is realized as abstract operation *updateStrategy(Set(WEpisode))* in the class *SimPlanner*.

2.1.2.3 Simulation-Based Planning

SimPlanner realizes the *plan* operation by using a simulation to refine its associated strategy. We formalize the algorithm of the *plan* operation in the following. Algorithm 4 shows the simulation-based planning procedure. The algorithm generates a set of episodes weighted by rewards (lines 2 – 5). This set is then used to refine the strategy (line 6). The concrete method to update the strategy remains unspecified by OnPlan.

Algorithm 5 shows the generation of a weighted episode. After initialization (lines 2 – 5), an episode is built by repeating the following steps for $h_{\max}$ times.

1. Sample an action $a \in \mathcal{A}$ from the current strategy w.r.t. the current simulation state $s \in \mathcal{S}$, i.e. $a \sim P_{\text{act}}(s)$ (line 7).

2. Store the current simulation state and selected action in the episode (line 8).

Algorithm 4 Simulation-based planning

Require: Local variable E_w : *Set(WEpisode)*

```
1: procedure PLAN
2:     E_w ← ∅
3:     for 0 ... eMax do
4:         E_w ← E_w ∪ generateEpisode()
5:     end for
6:     return updateStrategy(E_w)
7: end procedure
```

3. Simulate the execution of a. I.e., use the action a previously sampled from the strategy to progress the current simulation state s, i.e. $s \sim P_{\text{sim}}(s, a)$ (line 9).

4. Add the duration of a to the current episode time $t \in \mathbb{R}$. This is used for time-based discounting of rewards gathered in an episode (line 10).

5. Compute the reward of the resulting successor state discounted w.r.t. the current episode time t and the specified discount factor γ, and add it to the reward aggregation (line 11).

After simulation of h_{max} steps, the episode is returned weighted by the aggregated reward (line 13).

Algorithm 5 Generating weighted episodes

Require: Local variables s : *State*, r, t : *Real*, e : *Episode*, a : *Action*

```
1: procedure GENERATEEPISODE
2:     s ← state
3:     r ← rewardFct.getReward(s)
4:     t ← 0
5:     e ← nil
6:     for 0 ... hMax do
7:         a ← strategy.sampleAction(s)
8:         e ← e::(s, a)
9:         s ← simulation.sampleSuccessor(s, a)
10:        t ← t + a.duration
11:        r ← r + gamma^t · rewardFct.getReward(s)
12:    end for
13:    return (e, r)
14: end procedure
```

2.1.2.4 Framework Plug Points

In addition to the plug points given by the basic framework (see Section 2.1.1), the framework extension for simulation-based online planning provides the following plug points.

1. The operation *Simulation::sampleSuccessor(State, Action)* : *State*. This operation is the interface for any implementation of a simulation P_{sim}. The concrete design of this implementation is left to the designer of an instance of the framework. Both simulations for discrete and continuous state and action spaces can

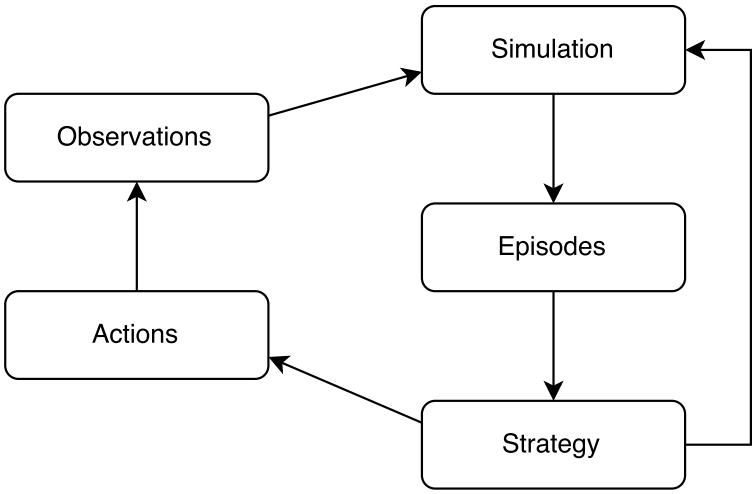


Figure 2.3: *Mutual influence of* ONPLAN *concepts*

instantiate ONPLAN. Note that, as P_{sim} may be learned from runtime observations of domain dynamics, this operation may be intentionally underspecified even by an instantiated system. Also note that the implementation of this operation does not necessarily have to implement the real domain dynamics. As simulation based planning typically relies on statistical estimation, any delta of simulation and reality just decreases estimation quality. While this also usually decreases planning effectiveness, it does not necessarily break planning completely. Thus, our framework provides a robust mechanism to deal with potentially imprecise or even erroneous specifications of P_{sim}.

2. The operation *SimPlanner::updateStrategy(Set(WEpisode)) : Strategy*. In principle, any kind of stochastic optimization technique can be used here. Examples include Monte Carlo estimation (see e.g. [RK11]) or genetic algorithms. We will discuss two effective instances of this operation in the following: Monte Carlo Tree Search for discrete domains in Section 3.1.1, and Cross Entropy Open Loop Planning for domains with continuous state-action spaces in Section 2.3.

Figure 2.3 shows an informal, high-level summary of ONPLAN concepts and their mutual influence. Observations result in the starting state of the simulations. Simulations are driven by the current strategy and yield episodes. The (weighted) episodes are used to update the strategy. The strategy yields actions to be executed. Executed actions influence observations made by an agent.

In the following Sections, we will discuss two state-of-the-art instances of the ONPLAN framework for simulation-based online planning introduced in Section 2.1. In Section 3.1.1, we will illustrate Monte Carlo Tree Search (MCTS) [BPW+12] and its variant UCT [KS06] as an instantiation of ONPLAN in discrete domains. in Section 2.3, we will discuss Cross Entropy Open Loop Planning (CEOLP) [Wei14, WL13] as an instance of ONPLAN for simulation based online planning in continuous domains with infinite state-actions spaces and branching factors.

2.2 Framework Instantiation in Discrete Domains

In this Section we discuss Monte Carlo Tree Search (MCTS) as an instantiation of the OnPlan framework in discrete domains.

2.2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) provided a framework for the first discrete planning approaches to achieve human master-level performance in playing the game Go autonomously [GKS+12]. MCTS algorithms are applied to a vast field of application domains, including state-of-the-art reinforcement learning and planning approaches in discrete domains [KH13, BPW+12, SSM13].

MCTS builds a search tree incrementally. Nodes in the tree represent states and action choices, and in each node information about the number of episodes an its expected payoff is stored. MCTS iteratively chooses a path from the root to leaf according to these statistics. When reaching a leaf, it simulates a potential episode until search depth is reached. A new node is added to the tree as a child of the leaf, and the statistics of all nodes that were traversed in this episode are updated according to the simulation result.

Figure 2.4 illustrates an iteration of MCTS. Each iteration consists of the following steps.

1. Nodes are selected w.r.t. node statistics until a leaf is reached (Figure 2.4a).

2. When a leaf is reached, simulation is performed and the aggregated reward is observed (Figure 2.4b).

3. A new node is added per simulation, and node statistics of the path selected in step (a) are updated according to simulation result (Figure 2.4c).

Steps (1) to (3) are repeated iteratively, yielding a tree that is skewed towards high value regions of the state space. This guides simulation effort towards currently promising search areas.

2.2.2 UCT

UCT (upper confidence bounds applied to trees) is an instantiation of MCTS that uses a particular mechanism for action selection in tree nodes based on regret minimization [KS06]. UCT treats action choices in states as multi-armed bandit problems. Simulation effort is distributed according to the principle of *optimism in the face of uncertainty* [BC12]: Areas of the search space that have shown promising value in past iterations are more likely to be explored in future ones. UCT uses the mathematically motivated upper confidence bound for regret minimization UCB1 [ACBF02] to formalize this intuition. The algorithm stores the following statistics in each node.

1. $\bar{x}_a$ is the average accumulated reward in past episodes that contained the tuple (s, a), where s is the state represented by the current node.

2. n_s is the number of episodes that passed the current state $s \in \mathcal{S}$.

3. n_a is the corresponding statistic for each action a that can be executed in s.

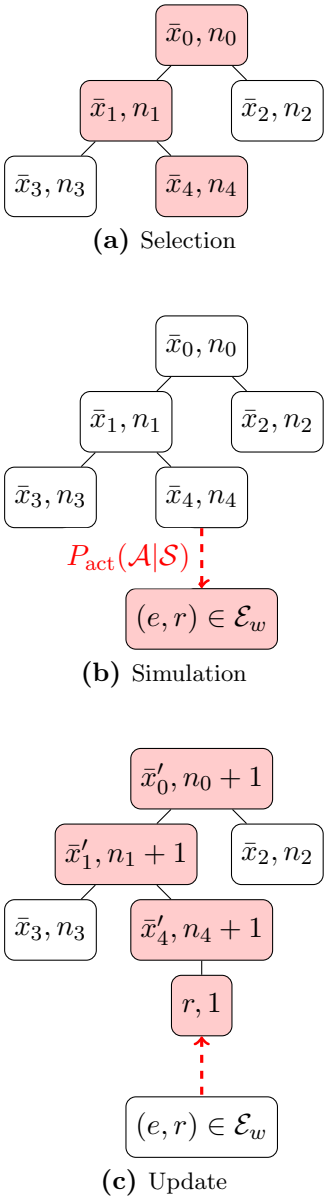


Figure 2.4: *Illustration of Monte Carlo Tree Search.* $(e, r) \in \mathcal{E}_w$ *is a weighted episode as generated by Algorithm 5. Nodes' mean values can be updated incrementally (see e.g. [GS11]):* $\bar{x}'_i = \bar{x}_i + \frac{r - \bar{x}_i}{n_i + 1}$.

Equation 2.2 shows the selection rule for actions in UCT based on node statistics. Here, $c \in \mathbb{R}$ is a constant argument that defines the weight of exploration (second term) against exploitation (first term). The equation provides a formalization of the exploration-exploitation tradeoff – the higher the previously observed reward of a child node, the higher the corresponding UCT score. However, the more often a particular child node is chosen by the search algorithm, the smaller the second term becomes. At the same time, the second term increases for all other child nodes. Thus, child nodes that have not been visited for some time become more and more likely to be included into future search episodes.

$$UCT(s,a) = \bar{x}_a + 2c\sqrt{\frac{2\ln n_s}{n_a}} \tag{2.2}$$

2.2.3 Framework Instantiation

Monte Carlo Tree Search instantiates the OnPlan framework for simulation-based online planning based on the following considerations.

1. *Strategy::sampleAction(State) : Action* is instantiated by the action selection mechanism used in MCTS. As MCTS is a framework itself, the particular choice is left underspecified. Examples of action selection mechanisms include uniform selection (all actions are chosen equally often), ϵ-greedy selection (the action with best average payoff is selected, with an ϵ probability to chose a random action) or selection according to UCT (see 2.2). Note that also probabilistic action selection strategies can be used, providing support for mixed strategies in a game-theoretic sense. Simulation outside the tree is performed according to an initial strategy. Typically, this is a uniformly random action selection. However, given expert knowledge can also be integrated here to yield potentially more valuable episodes with a higher probability.

2. *SimPlanner::updateStrategy(Set(WEpisode)) : Strategy* adds the new node to the tree and updates all node statistics w.r.t. the simulated episode weighted by accumulated reward. Note that a single episode suffices to perform an update. Different mechanisms for updating can be used. One example is averaging rewards as described above. Another option is to set nodes' values to the maximum values of their child nodes, yielding a Monte Carlo Bellman update of the partial state value function induced by the search tree [KH13].

3. While multiple simulations may be performed from a node when leaving the tree, typically the update (adding a node and updating all traversed nodes' statistics) is performed after each iteration. Thus, when using mcts for simulation-based planning, the number of episodes per strategy update $e_{\max}$ is usually set to 1.

4. The remaining plug-points – *execute* of class *Action*, *getReward* of class *RewardFunction* and *sampleSuccessor* of class *Simulation* – have to be instantiated individually for each domain and/or system use case.

2.2.4 Empirical Results

We implemented an instantiation of OnPlan with UCT in an example search-and-rescue scenario to show its ability to generate autonomous goal-driven behavior and its robustness w.r.t. unexpected events and changes of system goals at runtime.

2.2.4.1 Example Domain

We used the sample scenario described in the introduction for evaluation of the approach (see Section 1.3.1 in Chapter 1). We shortly recap the scenario here for the ease of reading.

A number of arbitrarily connected positions defines the domains topology. At some positions there is an ambulance (pos.safe = true). Positions may be on fire, except those that host an ambulance, i.e. class *Position* has the following invariant: pos.safe implies not(pos.fire) for all pos ∈ *Position*. Fires ignite or cease probabilistically depending on the number of fires at connected neighbor positions. A position may host any number of robots and victims. A robot can carry a number of victims that is bounded by its capacity. A carried victim does not have a position. A robot has five types of actions available.

1. Do nothing.
2. Move to a neighbor position that is not on fire.
3. Extinguish a fire at a neighbor position.
4. Pick up a victim at the robot's position if capacity is left.
5. Drop a carried victim at the robot's position.

All actions have unit duration. Each action may fail with a certain probability, resulting in no effect. Note that the number of actions available to a robot in a particular situation may vary due to various possible instantiations of action parameters (such as the particular victim that is picked up or the concrete target position of movement).

2.2.4.2 Experimental Setup

In all experiments, we generated randomly connected topologies with 20 positions and a connectivity of 30%, resulting in 6 to 7 connections per position on average. We randomly chose 3 safe positions, and 10 that were initially on fire. 10 victims were randomly distributed on the non-safe positions. We placed a single robot agent at a random starting position. All positions were reachable from the start. Robot capacity was set to 2. The robot's actions could fail with a probability of up to 5%, chosen uniformly distributed for each run. One run consisted of 80 actions executed by the agent. Results for all experiments have been measured with the statistical model checker MultiVeStA [SV13]. In all experiments, we set the maximum planning depth $h_{\max} = 20$. The discount factor was set to $\gamma = 0.9$. As MCTS was used for planning, we set $e_{\max} = 1$: The tree representing $P_{\text{act}}(\mathcal{A}|\mathcal{S})$ is updated after every episode. UCT's exploratory constant was set to $c = 20$ in all experiments.

In the following experiments, we let the agent deliberate for 0.2 seconds. That is, *actionRequired()* returned true once every 0.2 seconds; i.e. each action was planned for 0.2 seconds, incorporating information from past planning steps.

As long as not stated otherwise, we provided a reward of 100 to the planning agent for each victim that was located at a safe position. Let $I : \mathit{Bool} \to \{0, 1\}$ be an indicator function that yields 1 if the argument is defined and true and 0, otherwise. Let $\mathit{victims} : \mathcal{S} \to 2^{\mathit{Victim}}$ be the set of all victims present in a given state. Then, for any state $s \in \mathcal{S}$ the reward function was defined as follows.

$$R(s) = 100 \cdot \sum_{v \in \mathit{victims}(s)} I(v.\mathit{position.safe}) \tag{2.3}$$

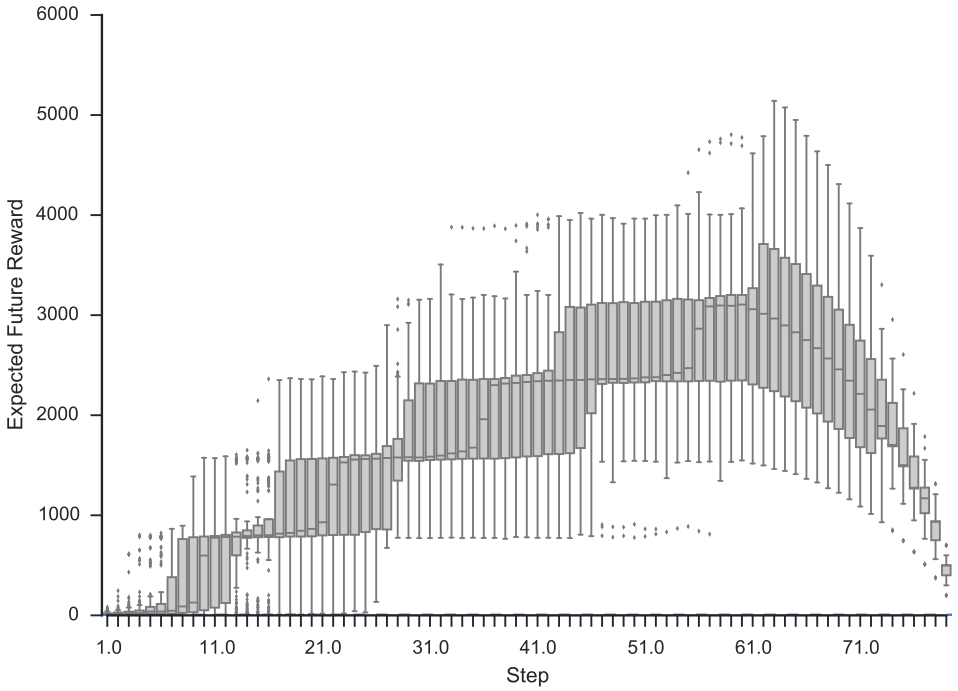


Figure 2.5: *Expected accumulated future reward at each step by the* MCTS *planner.*

The reward function instantiates the *getReward* operation of class *RewardFunction* in the OnPlan framework. Action implementations instantiate the *execute* operations of the corresponding subclasses of the *Action* class (e.g. move, pick up victim, etc.). A simulation about domain dynamics is provided to the simulation-based planner. It instantiates the *sampleSuccessor* operation of the *Simulation* class.

2.2.4.3 Estimation of Expected Future Reward

In a preliminary experiment, we observed the estimation of mean expected future reward. The MCTS planner increases the expected future reward up to step 60. Onwards from step 60 it decreases as the agent was informed about the end of the experiment after 80 steps. The planning depth $h_{\max} = 20$ thus detects the end of an experiment at step 60. The mean expected reward for executed actions is shown in Figure 2.5.

We also measured the increase in accuracy of the estimation of expected reward by MCTS. We measured the normalized coefficient of variation (CV) to investigate estimation accuracy, as the mean of expected future reward is highly fluctuating in the course of planning. The CV is a standardized measure of dispersion of data from a given distribution and independent from the scale of the mean, in contrast to standard deviation. Normalization of the CV renders the measurement robust to the number of samples. The normalized CV of a sample set is defined as quotient of the samples' standard deviation s and their mean $\bar{x}$, divided by the square root of available samples n. Note that the CV decreases as n increases, reflecting the increased accuracy of estimation as more samples become available.

$$\frac{s/\bar{x}}{\sqrt{n}} \tag{2.4}$$

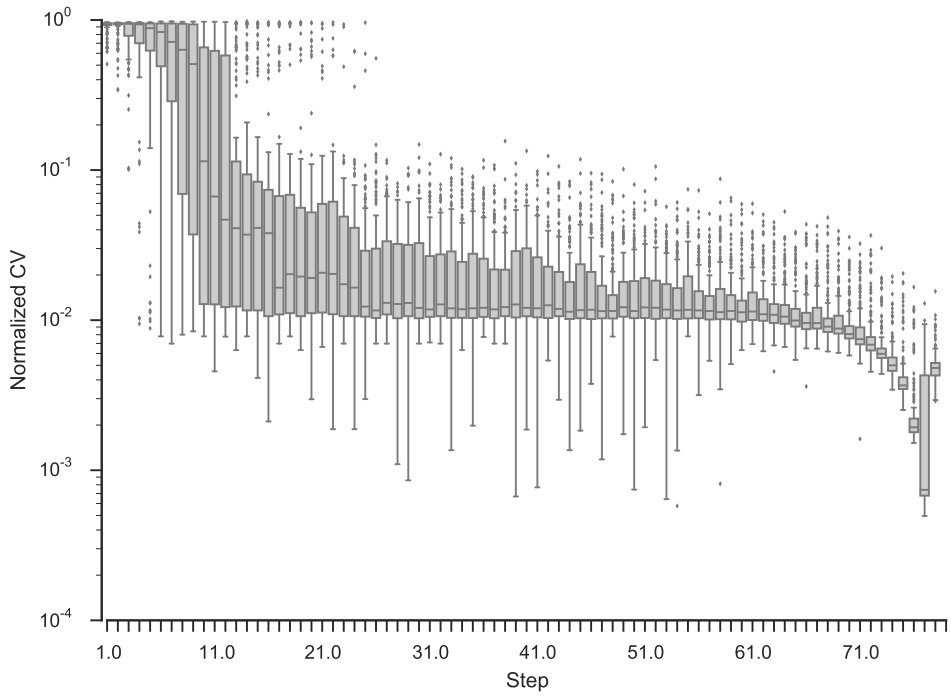


Figure 2.6: *Normalized coefficient of variation of expected reward estimation for actions executed by the* MCTS *planner. Note the logarithmic scale on the y-axis. After about 20 steps, estimation noise resembles the noise level inherent to the domain (up to 5% action failures and average spread of fires).*

We recorded mean $\bar{r}$ and standard deviation s_a of the expected reward gathered from simulation episodes for each potential action a, along with the number of episodes where a was executed at the particular step n_a. The normalized CV of an action then computes as follows.

$$\frac{s_a/\bar{r}}{\sqrt{n_a}} \tag{2.5}$$

Figure 2.6 shows the normalized CV w.r.t. the expected reward of the actions executed by the agent at a given step in the experiment. We observed that MCTS steadily improves its estimation accuracy of expected reward. After about 20 steps, estimation noise resembles the noise level inherent to the domain (up to 5% action failures and average spread of fires).

2.2.4.4 Autonomous System Behavior

In a baseline experiment, we evaluated ONPLAN's ability to synthesize autonomous behavior according to the given reward function. Figure 2.7 shows the average ratio of victims that was at a safe position w.r.t. the number of actions performed by the agent, within a 95% confidence interval. The Figure also shows the ratio of victims that are located at a burning position. No behavioral specification besides the instantiation of our planning framework has been provided to the agent. It can be seen that the planning component is able to generate a strategy that yields sensitive behavior: The robot transports victims to safe positions autonomously.

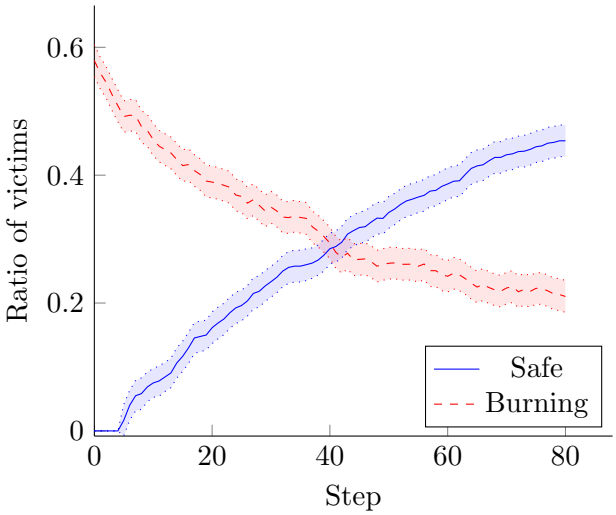


Figure 2.7: *Autonomous agent performance based on an instantiation of the* OnPlan *framework with a* MCTS *planner. Reward is given for victims at safe positions. Dotted lines indicate 0.95 confidence intervals.*

2.2.4.5 Robustness to Unexpected Events

In a second experiment we exposed the planning agent to unexpected events. This experiment is designed to illustrate robustness of the OnPlan framework to events that are not reflected by the simulation P_{sim} provided to the planning component. In this experiment, all victims currently carried by the robot fall to the robot's current position every 20 steps. Also, a number of fires ignite such that the total number of fires accumulates to 10. Note that these events are not simulated by the agent while planning. Figure 2.8 shows the agent's performance in the presence of unexpected events with their 95% confidence intervals. It can be seen that transportation of victims to safety is only marginally impaired by the sudden unexpected changes of the situation. As MCTS is used in an online manner that is based on replanning at each step, the planning framework is able to recover from the unexpected events efficiently.

2.2.4.6 System Goal Respecification

A third experiment highlights the framework's ability to adapt behavior synthesis to a system goal that changes at runtime. Before step 40, the agent was given a reward for keeping the number of fires low, resulting in a reduction of the number of burning victims. Onwards from step 40, reward was instead provided for victims that have been transported to safety. Besides respecification of the reward function to reflect the change of system goal no additional changes have been made to the running system. I.e., only the *rewardFct* reference of the planner was changed. This change impacts the weighting of episodes (see Algorithm 5, lines 3 and 11). The different weighting in turn impacts the updating of the planner's current strategy.

Figure 2.9 shows system performance in this experiment, together with 95% confidence intervals. The results indicate that the framework indeed is able to react adequately to the respecification of system goals. As system capabilities and domain dynamics remain the same throughout the experimental runtime, all high-level specifications such as action capabilities (i.e. the action space $\mathcal{A}$) and knowledge about domain dynamics (i.e. the generative model P_{sim}) are sensibly employed to derive valu-

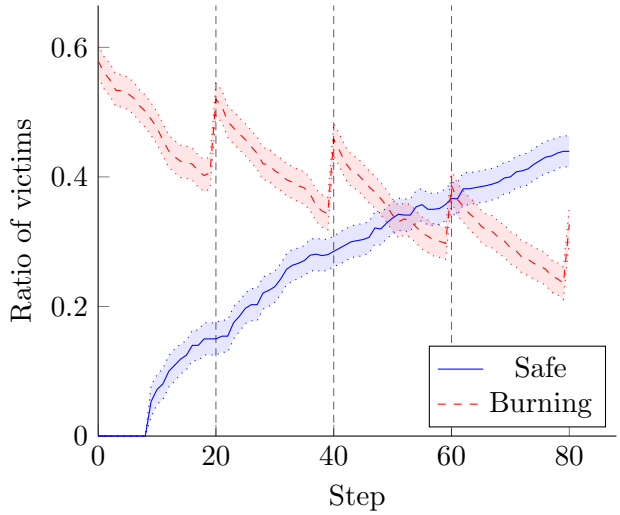


Figure 2.8: *Autonomous agent performance despite unexpected events at runtime. Every 20th step, all victims carried by the agent fall to the ground, and the number of fires raises to 10. Dotted lines indicate 0.95 confidence intervals.*

able courses of actions, regardless of the current system goal. OnPlan thus provides a robust system adaptation mechanism for runtime goal respecifications.

2.3 Framework Instantiation in Continuous Domains

We now focus on an instantiation of the OnPlan framework that works in continuous space and action domains. I.e. states and actions are represented as vectors of real numbers $\mathbb{R}^n$, for some $n \in \mathbb{N}$. This means that state and action spaces are of infinite size. In this section we show how Cross Entropy Open Loop Planning (CEOLP) [WL13, Wei14] instantiates our planning framework, and illustrate how information obtained from simulations in the planning process can be used to identify promising ares of the search space in continuous domains. CEOLP works by optimizing action parameters w.r.t. expected payoff by application of the cross entropy method.

2.3.1 Cross Entropy Optimization

The cross entropy method for optimization [dBKMR05, RK13] allows to efficiently estimate extrema of an unknown function $f : X \to Y$ via importance sampling. To do so, an initial probability distribution (that we call sampling distribution) $P_{\text{sample}}(X)$ is defined in a way that covers a large region of the function's domain. For estimating extrema of f, a set of samples $x \in X$ is generated w.r.t. the sampling distribution (i.e. $x \sim P_{\text{sample}}(X)$). The size of the sample set is a parameter of the cross entropy method. For all x in the set, the corresponding $y = f(x) \in Y$ is computed. Then samples are weighted w.r.t. their relevance for finding the function extrema. For example, when trying to find maxima of f, samples x are weighted according to $y = f(x)$. Typically this involves normalization to keep sample weights in the $[0; 1]$ interval. We denote the weight of a sample x_i by w_i. The weighted sample set is used to update the sampling distribution $P_{\text{sample}}(X)$ by minimizing the distributions' cross entropy. Minimizing cross entropy yields a distribution that is more likely to generate samples in X that are located close to the maxima of f. Minimizing of cross entropy has been shown to be

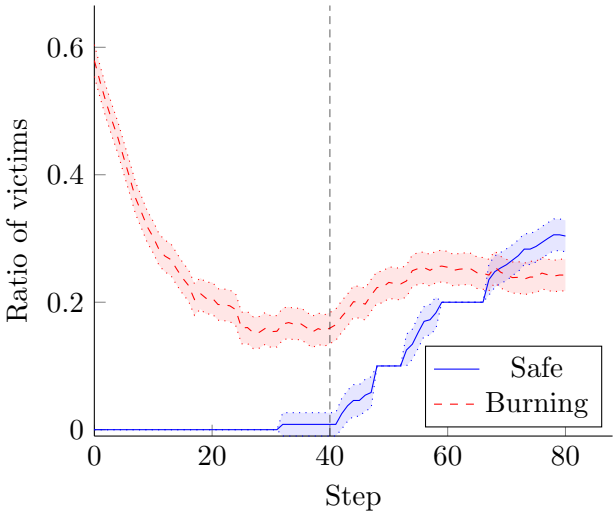


Figure 2.9: *Autonomous agent performance with a respecification of system goal at runtime. Before step 40, the agent is given a reward for keeping the number of fires low, resulting in a reduction of the number of burning victims. Onwards from step 40, reward is provided for victims that have been transported to safety. Dotted lines indicate 0.95 confidence intervals.*

equivalent to maximizing the likelihood of the samples x weighted by $f(x)$ [dBKMR05]. Sampling, weighting and building new sampling distributions by maximum likelihood estimation are repeated iteratively. This yields an iterative refinement of the sampling distribution which increases the probability to sample in the region of the maxima of f, thus providing a potentially better estimate thereof. While convergence of the CE method has been proven for certain conditions, it is not easy to establish these conditions in the most practically relevant settings [Mar05]. However, empirical results indicate that the CE method provides a robust optimization algorithm which has been applied successfully in a variety of domains (see e.g. [dBKMR05, Kob12, LWM15])

Figure 2.10 illustrates the idea of iterative refinement of the sampling distribution to increase the probability to generate samples in the region of the maxima of the unknown function f. In this example, a Gaußian sampling distribution was chosen. The sampling distribution is shown as solid line, while the unknown target function is shown as dashed line. While in this Figure the target function has a Gaußian form as well, this is not required for the cross entropy method to work. Initially, the sampling distribution has a large variance, providing a well spread set of samples in the initial generation. Then the samples are weighted w.r.t. their value $f(x)$ and a maximum likelihood estimate is built from the weighted samples. This yields a Gaußian sampling distribution that exposes less variance than the initial one. Repeating this process finally yields a distribution that is very likely to produce samples that are close to the maximum of the unknown target function.

Sampling from a Gaußian distribution can for example be done via the Box-Muller method [BM58]. Equation 2.6 shows a maximum likelihood estimator for a Gaußian distribution (μ, σ^2), given a set I of n samples $\vec{a}_i \in \mathcal{A}, i \in \{0, ..., n\}$, each weighted by $w_i \in \mathbb{R}$. This yields a new Gaußian distribution that concentrates its probability mass in the region of samples with high weights. Samples with low weights are less

influential on the probability mass of the new distribution.

$$\mu = \frac{\sum_{(\vec{a}_i, w_i) \in I} w_i \vec{a}_i}{\sum_{(\vec{a}_j, w_j) \in I} w_j}$$
$$\sigma^2 = \frac{\sum_{(\vec{a}_i, w_i) \in I} w_i (\vec{a}_i - \mu)^T (\vec{a}_i - \mu)}{\sum_{(\vec{a}_j, w_j) \in I} w_j} \tag{2.6}$$

Summarizing, the requirements for the cross entropy method are as follows.

1. A way to weight the samples, i.e. a way to compute $f(x)$ for any given $x \in X$.

2. An update mechanism for the distribution based on the weighted samples has to be provided. Typically, this is a maximum likelihood estimator for the sampling distribution.

Note that the cross entropy method is not restricted to a particular form of probability distribution. Also discrete distributions or other continuous ones than a Gaußian can be used to model the sampling distribution [dBKMR05].

2.3.2 Cross Entropy Open Loop Planning

The key idea of Cross Entropy Open Loop Planning is to use cross entropy optimization on a *sequence* of actions. The agent's strategy $P_{\text{act}}(\mathcal{A}|\mathcal{S})$ is thus represented by a *vector* of multivariate Gaußian distributions over the parameter space of the actions $\mathcal{A} \subseteq \mathbb{R}^{\mathbb{N}}$.

In the context of our framework for simulation-based planning, we want to find the maxima of a function that maps sequences of actions to expected rewards, that is $f : \mathcal{A}^* \to \mathbb{R}$. The simulation $P_{\text{sim}}(\mathcal{S}|\mathcal{S} \times \mathcal{A})$ and the reward function $R : \mathcal{S} \to \mathbb{R}$ allow us to estimate $f(\vec{a})$ for any given $\vec{a} \in \mathcal{A}^*$: We can generate a sequence of states $\vec{s} \in \mathcal{S}^*$ by sampling from the simulation and build an episode $e \in \mathcal{E}$ from $\vec{a}$ and $\vec{s}$. We can then evaluate the accumulated reward of this episode by computing the discounted sum of gathered rewards $R_{\mathcal{E}}(e)$ (see Equation 2.1).

In OnPlan, we generate $e_{\max}$ episodes and weight them by accumulated reward as shown in Algorithm 5. The sampling of actions from the strategy (Algorithm 5, line 9) is done by generating a sample from the Gaußian distribution over action parameters at the position of the strategy vector that matches the current planning depth (i.e. the number of iteration of the for-loop in Algorithm 5, line 6). The Gaußians that form the strategy $P_{\text{act}}(\mathcal{A}|\mathcal{S})$ are updated after generating and weighting $e_{\max}$ episodes, as stated in Algorithm 4. The update is performed via maximum likelihood estimation for each Gaußian in the strategy vector as defined in Equation 2.6.

2.3.3 Framework Instantiation

Cross Entropy Open Loop Planning instantiates the OnPlan framework based on the following considerations.

1. *Strategy::sampleAction(State) : Action* generates samples from the current vector of Gaußians that represents P_{act}. As CEOLP is state agnostic and only accumulates action parameters w.r.t. planning depth, this depth is the only information that is used for conditioning the distribution: I.e. when sampling at depth d, the d-th component of the plan distribution is used to generate a value for the action.

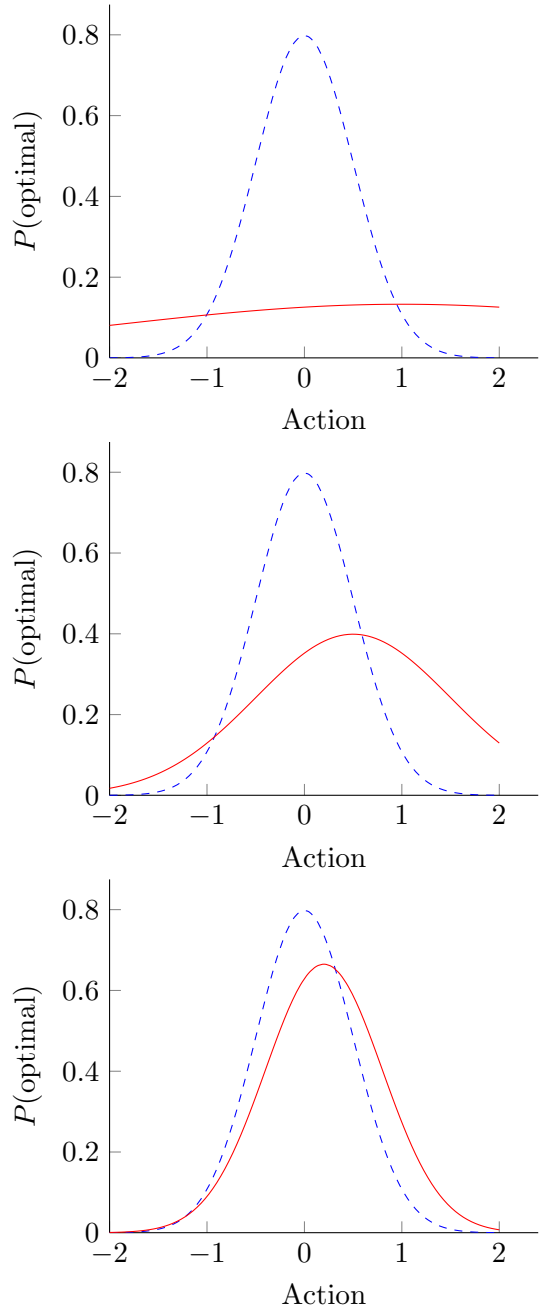


Figure 2.10: *Illustration of the cross entropy method with a Gaußian sampling distribution. The dashed line represents the unknown target function. The solid line represents the Gaußian sampling distribution that is iteratively refined by maximum likelihood estimation based on samples from the previous iteration, weighted by their target function values.*

2. *SimPlanner::updateStrategy(Set(WEpisode)) : Strategy* refines the Gaußians in the strategy by maximum likelihood estimation w.r.t. the samples from the previous generation, weighted by the accumulated reward (see Equation 2.6). This yields a strategy that is likely to produce high-reward episodes.

3. The remaining plug-points – *execute* of class *Action*, *getReward* of class *RewardFunction* and *sampleSuccessor* of class *Simulation* – have to be instantiated individually for each domain and/or system use case.

2.3.4 Empirical Results

We compared an instantiation of our framework with CEOLP with a vanilla Monte Carlo planner that does not perform iterative update of its strategy. The latter proposes actions from a strategy distribution that is the best average w.r.t. weighted simulation episodes. However, in contrast to the CEOLP planner, it does not refine the strategy iteratively while simulating to concentrate its effort on promising parts of the search space.

2.3.4.1 Experimental Setup

We provided the same number of simulations per action to each planner. The one that instantiates ONPLAN updates the strategy distribution every 30 simulations (i.e. $e_{\max} = 30$) and does this 10 times before executing an action. Planning depth was set to $h_{\max} = 50$. The vanilla Monte Carlo planner uses the aggregated result of 300 episodes generated w.r.t. the initial strategy to decide on an action, without updating the strategy within the planning process. It only builds a distribution once after all samples have been generated and evaluated to decide on an action. Action duration was fixed at one second. The discount factor was set to $\gamma = 0.95$ in all experiments.

2.3.4.2 Example Domain

We used the continuous example domain introduced in Chapter 1 to evaluate the ONPLAN approach for continuous domains. In this search and rescue setting, and agent that has to collect stochastically moving victims in a real-time setting. To do so, it has to determine its optimal velocity and rotation rate w.r.t. its current situation. For more detailed information on the sample scenario, we refer to Section 1.3.2 in Chapter 1.

The reward function providing unit reward to a planner on collecting a victim instantiates the *getReward* operation of class *RewardFunction* in the ONPLAN framework. Action implementations instantiate the *execute* operations of the corresponding subclasses of class *Action*. The simulation provided to the simulation-based planner instantiates the *sampleSuccessor* operation of the *Simulation* class.

2.3.4.3 Iterative Parameter Variance Reduction

Figure 2.11 shows an exemplary set of actions sampled from P_{act} for the first action to be executed. Here, the effect of updating the sampling strategy can be seen for the two-dimensional Gaußian distribution over the action parameters speed (x axis) and rotation rate (y axis). While the distribution is spread widely in the initial set of samples, updating the strategies according to the samples' weights yields distributions that increasingly concentrate around regions of the sample space that yield higher

expected reward. The figures also show Spearman's rank correlation coefficient of the sampled action parameters to measure the dependency between the two action variables (speed and rotation rate). It can be seen that the degree of correlation increases with iterations. Also, the probability that there is no statistically significant correlation of the parameters decreases: From 0.94 in the initial set of samples to 0.089 in the tenth set.

2.3.4.4 Estimation of Expected Reward

Figures 2.12 and 2.13 show the effect of iteratively updating the strategy on simulation episode quality. We evaluated the magnitude of effect depending on the degree of domain noise. Domain noise is given by movement speed of victims in our example. We compared victim speed of 0.1 and 1.0 (m/sec). Figure 2.12 shows the average accumulated reward of the episodes generated in a particular iteration, grouped by domain noise. The result is shown as a factor of the value in the initial iteration. The data shows that the episodes' average accumulated reward increases with iterations of strategy updates. The magnitude of the effect depends on domain noise. Figure 2.13 shows the corresponding coefficient of variation (CV), the quotient of standard deviation and mean of a sample set. This data is also grouped by domain noise. The CV of accumulated reward per episode shows a tendency to be reduced with iterations. This means that the estimation of the target value (accumulated reward per episode) is likely to increase its accuracy due to iterative strategy refinement. Again, the magnitude of the effect depends on domain noise.

2.3.4.5 Comparison to Vanilla Monte Carlo

Figure 2.14 shows the time needed to collect the victims by the OnPlan and vanilla Monte Carlo (Vmc) planners. Both are able to autonomously synthesize behavior that leads to successful completion of their task. System autonomy is achieved in a highly dynamic continuous state-action space with infinite branching factor and despite the noisy simulation. However, the planner using our framework is collecting victims more effectively. The benefit of making efficient use of simulation data by cross entropy optimization to drive decisions about actions becomes particularly clear when only a few victims are left. In these situations, only a few combinations of actions yield goal-oriented behavior. Therefore it is valuable to identify uninformative regions of the sampling space fast in order to distribute simulations more likely towards informative and valuable regions.

2.4 Related Work

The MAPE-K approach to autonomic computing is a high level architectural blueprint for designing autonomous systems [Kep03, KC03]. As OnPlan, it proposes to use a separate planning component (the *autonomic manager*) to control an agent (the *managed element*). The autonomic manager comprises four control activities: Monitoring, analysis, planning and execution. These activities are based on the centrally available knowledge of a component. Figure 2.15 illustrates the approach.

While MAPE-K provides an intuitive description of tasks to be performed by an autonomous system, it does not provide any proposition about *how* to realize them in an algorithmic sense. I.e. MAPE-K remains more abstract than the OnPlan framework. OnPlan suggests a particular approach to system autonomy by combining

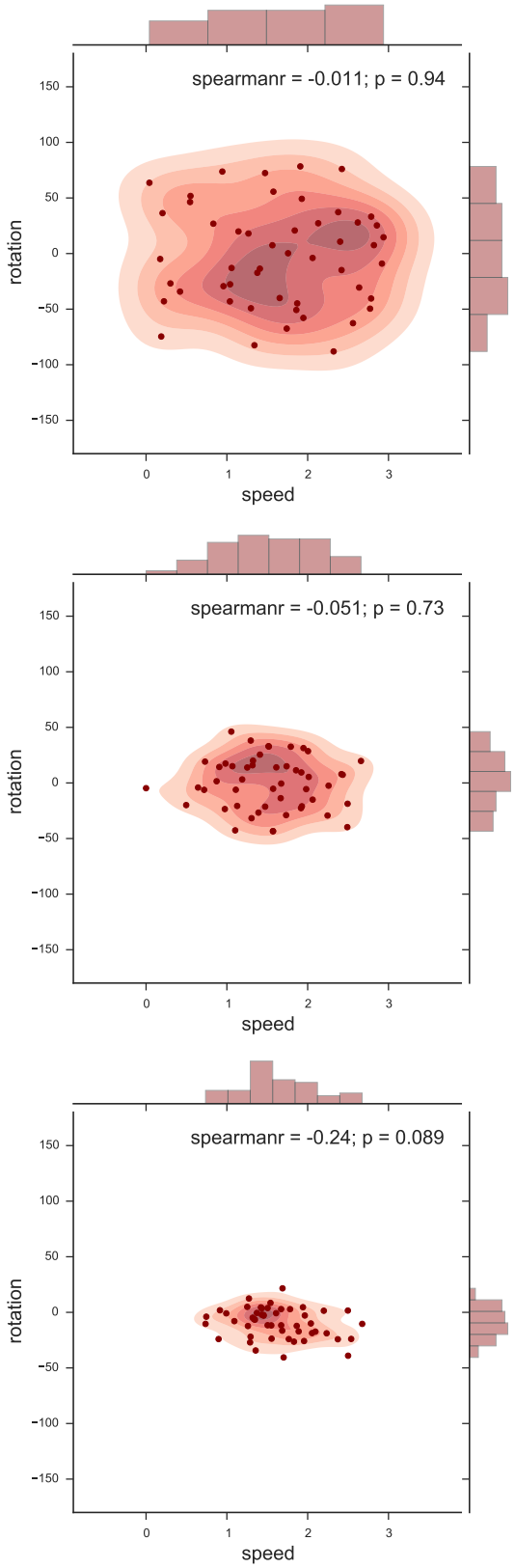


Figure 2.11: *Actions sampled from P_{act} for the first action to execute at iterations one, five and ten.*

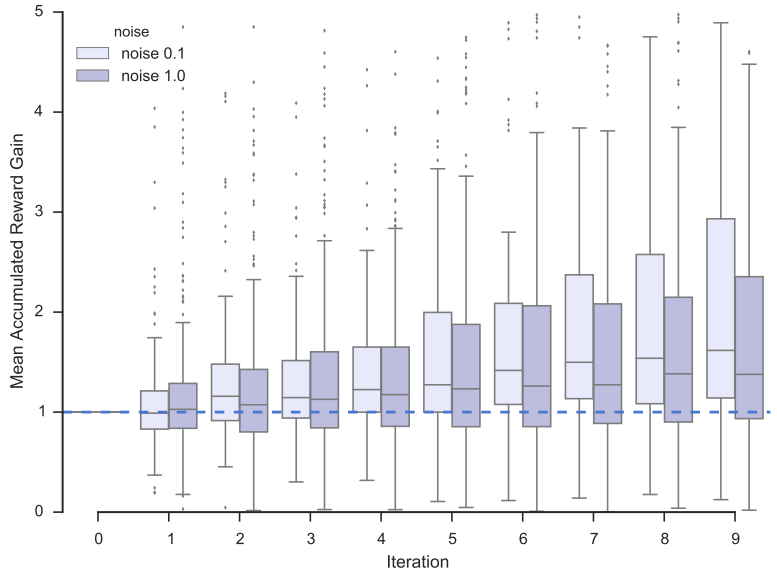


Figure 2.12: *Mean accumulated reward of sampled episodes per iteration. Results are shown as factor (i.e. gain) of mean accumulated reward in the initial iteration. The data shows a tendency to increase episode quality with iterative updating of the sampling strategy. The magnitude of the effect depends on domain noise. Boxes contain 50% of measured data, whiskers 99.3%.*

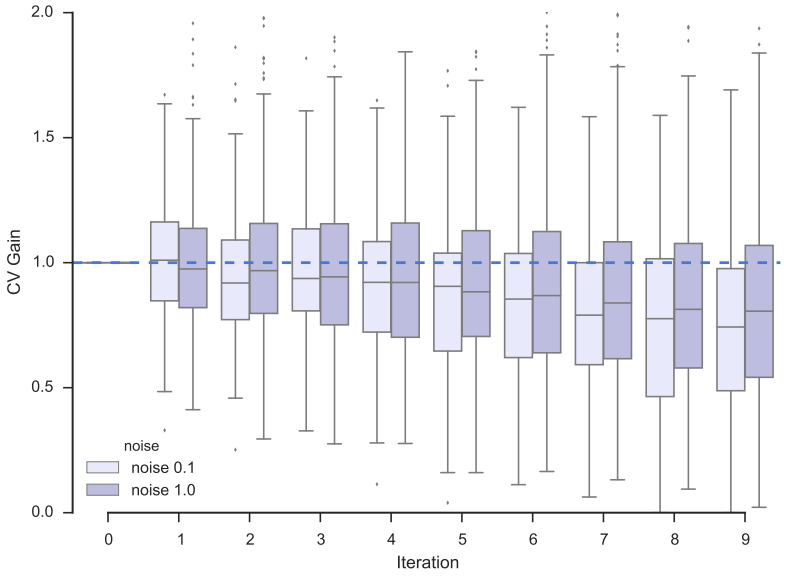


Figure 2.13: *Coefficient of variation (CV) of mean accumulated reward from the sampled episodes per iteration. Results are shown as factor (i.e. gain) of CV of mean accumulated reward in the initial iteration. The data shows a tendency to increase estimation accuracy with iterative updating of the sampling strategy (i.e. decreasing CV). The magnitude of the effect depends on domain noise. Boxes contain 50% of measured data, whiskers 99.3%.*

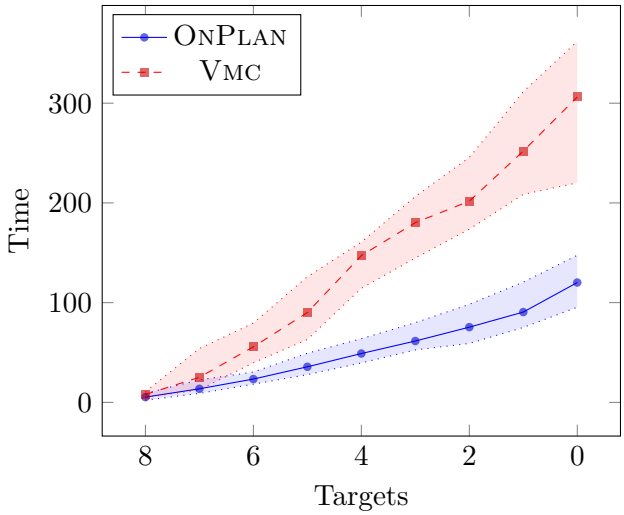


Figure 2.14: *Comparison of an instance of* OnPlan *using* Ceolp *with a vanilla Monte Carlo planner (*Vmc*). Lines show the median, dotted lines indicate interquartile range (comprising 50% of measured data).*

online planning, probabilistic representations and importance sampling as discussed in this Chapter.

A more concrete architecture for real time model-based reinforcement learning is rtmba [HQS12, HS13]. As OnPlan it parallelizes planning and execution in separate computational threads. rtmba also comprises a thread for learning an environmental model from observations. We will discuss a particular approach to model learning in Chapter 4. In contrast to OnPlan, rtmba is proposing Monte Carlo Tree Search as a particular approximate planning algorithm. This constrains the architecture to tree based stochastic search procedures. OnPlan proposes to use the more general approach of importance sampling to efficiently approximate optimal behavioral choices in dynamically changing environments. As has been shown in this Chapter, this allows to treat discrete and continuous domains in a uniform manner. In a sense, OnPlan can be seen as a generalization of rtmba in terms of the approach to planning.

2.5 Summary & Outlook

Modern application domains such as cyber-physical systems are characterized by their immense complexity and high degrees of uncertainty. This renders unfeasible classical approaches to system autonomy which compile a single solution from available information at design-time. Instead, the idea is to provide a system with a high-level representation of its capabilities and the dynamics of its environment. The system then is equipped with mechanisms that allow to compile sensible behavior according to this high-level model and information that is currently available at runtime. I.e., instead of providing a system with a single predefined behavioral routine it is given a *space* of solutions and a way to evaluate individual choices in this space. This enables systems to autonomously cope with complexity and change.

In this paper we proposed the OnPlan framework for realizing this approach. It provides simulation-based system autonomy employing online planning and importance sampling. We defined the core components for the framework and illustrated its behavioral skeleton. We showed two concrete instantiations of our framework: Monte

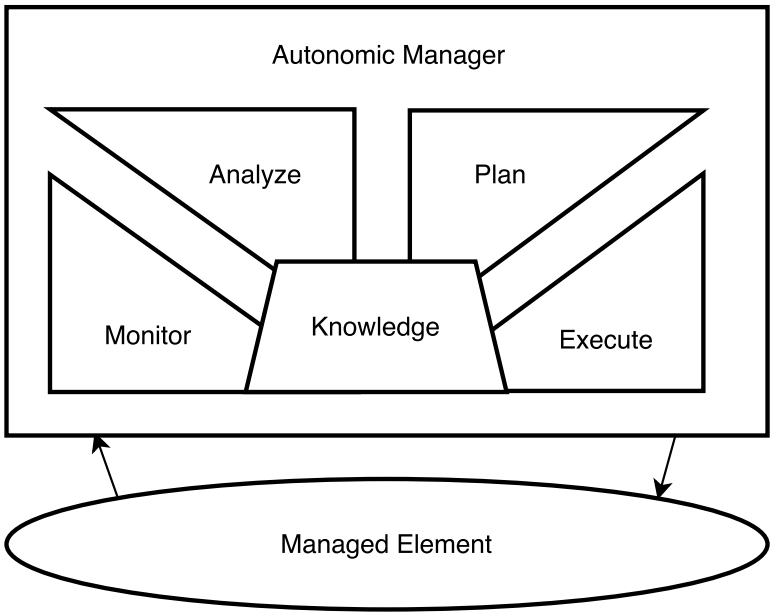


Figure 2.15: *The MAPE-K approach to autonomic computing (adapted from [KC03]).*

Carlo Tree Search for domains with discrete state-action spaces, and Cross Entropy Open Loop Planning for continuous domains. We discussed how each instantiates the plug points of OnPlan. We showed the ability of our framework to enable system autonomy empirically in a search and rescue domain example.

An important direction of future work is to extend OnPlan to support learning of simulations from observations at runtime. Machine learning techniques such as probabilistic classification or regression provide potential tools to accomplish this task (see e.g. [HS13]). Also, other potential instantiations of the framework should be explored, such as the GOURMAND planner based on labeled real-time dynamic programming [KDMW12, BG03], sequential halving applied to trees (SHOT) [KKS13, PCWL14], hierarchical optimistic optimization applied to trees (HOOT) [MWL11] or hierarchical open-loop optimistic planning (HOLOP) [Wei14, WL12]. It would also be interesting to investigate possibilities to extend specification logics such as LTL or CTL [BK$^+$08] with abilities for reasoning about uncertainty and solution quality. Model checking of systems acting autonomously in environments with complexity and runtime dynamics such as the domains considered in this paper provides potential for further research. Another direction of potential further research is simulation-based planning in collectives of autonomous entities that are able to form or dissolve collaborations at runtime, so-called ensembles [WHKM15, HG15]. Here, the importance sampling approach may provide even more effectiveness as in a single-agent context, as the search space typically grows exponentially in the number of agents involved. Mathematically identifying information that is relevant in a particular ensemble could provide a principled way to counter this combinatorial explosion of the search space in multi-agent settings.

Chapter 3

Monte Carlo Action Programming

We consider the problem of sequential decision making in highly complex and changing domains. These domains are characterized by large probabilistic state spaces and high branching factors. Additional challenges for system design are occurrence of unexpected events and/or changing goals at runtime.

A state of the art candidate for responding to this challenge is behavior synthesis with online planning [BPW+12, KDMW12, KH13]. Here, a planning agent evaluates possible behavioral choices w.r.t. current situation and background knowledge at runtime. At some point, it acts according to this evaluation and observes the actual outcome of the action. Planning continues, incorporating the observed result. Planning performance directly correlates with search space cardinality.

This Chapter discusses *Monte Carlo Action Programming* (MCAP) to reduce search space cardinality through specification of heuristic knowledge in the form of procedural nondeterministic programs. MCAP is based on stochastic interpretation of nondeterministic action programs by Monte Carlo Tree Search (MCTS) [BPW+12, CDJSU06]. Combining search space constraints and stochastic interpretation enables program evaluation in large probabilistic domains with high branching factors.

From the perspective of Monte Carlo Tree Search, MCAP provides a formal nondeterministic action programming language that allows to specify plan sketches for autonomous systems. From the perspective of action programming, MCAP introduces stochastic interpretation with MCTS. This enables effective program interpretation in very large, complex domains.

We will discuss Monte Carlo Tree Search and action programming in Section 4.1. Section 3.2 introduces Monte Carlo Action Programming. In Section 3.4 we empirically compare plain MCTS and MCAP specification for online planning. We conclude and outline venues for further research in Section 5.6.

3.1 Preliminaries

We briefly review Monte Carlo Tree Search in Section 3.1.1 and action programming in Section 3.1.2.

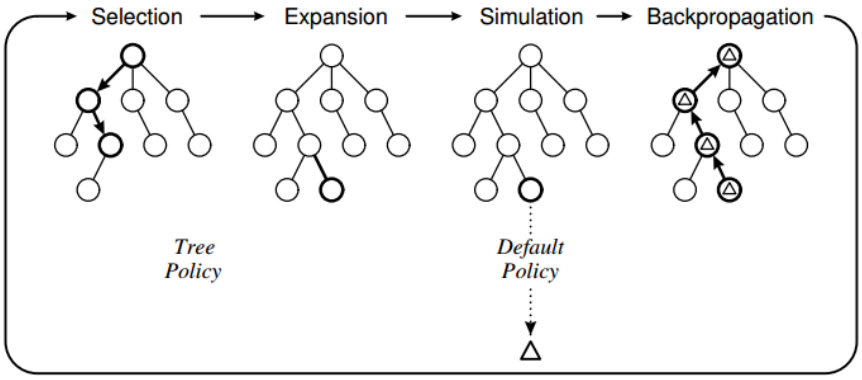


Figure 3.1: *Monte Carlo Tree Search [BPW+12].*

3.1.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a framework for statistical search in very large state spaces with high branching factors based on a generative model of the domain (i.e. a simulation). It yields good performance even without heuristic assessment of intermediate states in the search space. The MCTS framework originated from research in computer Go [CDJSU06, SSM13]. The game Go exposes the mentioned characteristics. Also, not many good heuristics are known for Go. Nevertheless, specialized Go programs based on MCTS are able to play on the level of a human professional player [GKS+12]. Recently, a combination of MCTS and deep neural networks has been able to defeat a human professional Go player [SHM+16]. MCTS is also commonly used in autonomous planning [KDMW12, KH13] and has been applied successfully to numerous other search tasks [BPW+12].

MCTS adds nodes to the tree iteratively. Nodes represent states and store metadata about search paths that lead through them. Gathered metadata comprises mean reward (i.e. node value) and the number of searches that passed through the node. It enables assessment of exploration vs. exploitation: Should search be directed to already explored, promising parts of the search space? Or should it gather information about previously unexplored areas?

Figure 3.1 shows the basic principle of MCTS. Based on node information, MCTS selects an action w.r.t. a given *tree policy*. The successor state is determined by simulating action execution. Selection is repeated as long as simulation leads to a state that is represented by a node in the tree. Otherwise, a new node representing the simulated outcome state is added to the tree (expansion). Then, a *default policy* is executed (e.g. uniform random action execution). Gathered reward is stored in the new node (simulation or *rollout*). This gives a first estimation of the new node's value. Finally, the rollout's value is backpropagated through the tree and the corresponding node values are updated. MCTS repeats this procedure iteratively. Algorithm 6 shows the general MCTS approach in pseudocode. Here, v_0 is the root node of the search tree. v_l denotes the last node visited by the tree policy. Δ is the value of the rollout from v_l according to the default policy.

MCTS can be interrupted at any time and yields an estimation of quality for all actions in the root state. The best action (w.r.t. node information) is executed and its real outcome is observed. MCTS continues reusing the tree built so far. Eventually, nodes representing past states are pruned from the tree.

Algorithm 6 General MCTS approach [BPW+12]

```
1: procedure MCTS(s_0)
2:     create root node v_0 with state s_0
3:     while within computational budget do
4:         v_l ← TREEPOLICY(v_0)
5:         Δ ← DEFAULTPOLICY(v_l)
6:         BACKUP(v_l, Δ)
7:     end while
8:     return a(BESTCHILDv_0())
9: end procedure
```

3.1.2 Action Programming

Nondeterministic action programs define sketches for system behavior that are interpreted at runtime, leaving well-defined choices to be made by the system at runtime. Interpreting an action program typically provides a measure of quality for particular instantiations of theses sketches. Concrete traces are then executed w.r.t. this quality metric.

Well-established action programming languages are GOLOG [DGLL00, GLLS09] and FLUX [Thi05]. Each is interpreted w.r.t. a particular formal specification of domain dynamics: The situation calculus and the fluent calculus are concerned with specification of action effects and domain dynamics in first order logic [BRS+00, Thi98]. For both GOLOG and FLUX, Prolog interpreters have been implemented.

Previous work of the author proposed an action programming framework in *rewriting logic* [Bel13, Bel14]. It is based on formal algebraic specification of domain dynamics in rewriting logic, incorporating order sorted representations, equational abstractions and symbolic computation [Mes12]. Action programs are written in the Maude language and interpreted by term rewriting [CDE+07].

The MCAP framework differs from these formalisms and their respective languages:

1. MCAP does not provide nor require a specific formal representation of domain dynamics. Rather, any form of domain simulation suffices.

2. MCAP does not explore the search space exhaustively. Rather, programs are interpreted stochastically by MCTS. The search space is explored iteratively. Program interpretation is directed to promising areas of the search space based on previous interpretations. Search can be interrupted any time yielding an action recommendation accounting for current situation and a given program. Recommendation quality depends on the number of simulations used for search [BPW+12].

3.2 Monte Carlo Action Programming

This Section introduces Monte Carlo Action Programming (MCAP), a nondeterministic procedural programming framework for autonomous systems. The main idea of the MCAP framework is to allow to specify behavioral blueprints that leave choices to an agent. An MCAP is a nondeterministic program. MCAP programs are interpreted probabilistically by MCTS. MCAPs constrain the MCTS search space w.r.t. a procedural nondeterministic program.

Framework parameters are defined in Section 3.2.1. MCAP language syntax is introduced in Section 3.2.2. Section 3.2.3 defines language semantics.

3.2.1 Framework Parameters

The MCAP framework requires the following specification.

1. As for the ONPLAN framework (cf. Chapter 2), we require sorts for states $\in \mathcal{S}$ and actions $\in \mathcal{A}$.

2. Actions have a duration $d : \mathcal{A} \to \mathbb{R}$.

3. MCAP requires a *generative domain model* $P(\mathcal{S} \mid \mathcal{S} \times \mathcal{A})$ that captures the probability distribution of successor states w.r.t. current state and executed action. The model does not have to be explicit: The framework only requires a simulation that allows to query one particular successor state. I.e. this probability distribution is equivalent to the simulation required by the ONPLAN framework (cf. Chapter 2, Section 2.1.2).

4. MCAP requires a reward function R that encodes the quality of a state w.r.t. system goals (Equation 3.1). This reward function is equivalent to the one required by the ONPLAN framework (cf. Chapter 2, Section 2.1.1).

$$\mathcal{R} : \mathcal{S} \to \mathbb{R} \tag{3.1}$$

5. A discount factor $\gamma \in [0; 1]$ weights the impact of potential future decision on the current situation. A discount factor of zero means that only immediate consequences of action are considered. A discount factor of one means that all future consequences influence the current decision equally, regardless of their temporal distance (cf. Chapter 2, Section 2.1.2).

6. MCAP requires a maximum search depth $h_{\max} \in \mathbb{N}$. This defines the maximum depth of simulation when evaluating behavioral alternatives (cf. Chapter 2, Section 2.1.2).

3.2.2 Syntax

Equation 3.2 defines the basic syntax of the MCAP language. ϵ is the empty program, $\mathcal{A}$ denotes the specified action space, **;** is a sequential operator, + represents non-deterministic choice, and $\|$ denotes interleaving concurrency. Q denotes the query space for conditional evaluation (see Equation 3.23). ? denotes querying the current execution context. $\circ$ denotes a conditional loop.

$$\begin{aligned} \mathcal{P} := \epsilon \mid \mathcal{A} \mid \mathcal{P}\,;\mathcal{P} \\ \mid \mathcal{P}{+}\mathcal{P} \mid \mathcal{P} \parallel \mathcal{P} \\ \mid ?(\mathcal{Q})\{\mathcal{P}\} \mid \neg?(\mathcal{Q})\{\mathcal{P}\} \\ \mid \circ\{\mathcal{Q}\}\{\mathcal{P}\} \end{aligned} \tag{3.2}$$

Sequential composition is associative, non-deterministic choice and parallel composition are associative and commutative.

3.2.2.1 MCAP Normal Form

We define a conjunctive normal form for MCAPs. Equations 3.3 to 3.10 define a term reduction system that transform programs to their normal form, and to resolve parallel composition. The following Equations (3.3 to 3.6) show reduction of programs to their conjunctive normal form.

$$\epsilon ; p = p \tag{3.3}$$
$$p + p = p \tag{3.4}$$
$$(p_1 + p_2) ; p = (p_1 ; p) + (p_2 ; p) \tag{3.5}$$
$$p ; (p_1 + p_2) = (p ; p_1) + (p ; p_2) \tag{3.6}$$

Equation 3.3 defines ϵ as left identity element of programs. We do not define right identity, as we use ϵ as a program termination symbol when interpreting an MCAP (cf. Equation 3.25). Equation 3.4 defines program choice idempotent. The following two Equations 3.5 and 3.6 transform sequences of choices into a disjunctive normal form.

The following four Equations (3.7 to 3.10) reduce parallel compositions to programs only containing sequences and choices. Parallel composition is resolved by non-deterministic interleaving of actions. We do not define any identity for ϵ and parallel composition to keep the number of equations minimal. It is treated by Equation 3.3 after reduction of parallelism to sequences and choices.

$$p \parallel (p_1 + p_2) = (p \parallel p_1) + (p \parallel p_2) \tag{3.7}$$
$$(a_1 ; p_1) \parallel (a_2 ; p_2) = (a_1 ; (p_1 \parallel (a_2 ; p_2))) + (a_2 ; ((a_1 ; p_1) \parallel p_2)) \tag{3.8}$$
$$a_1 \parallel (a_2 ; p) = (a_1 ; a_2 ; p) + (a_2 ; (a_1 \parallel p)) \tag{3.9}$$
$$a_1 \parallel a_2 = (a_1 ; a_2) + (a_2 ; a_1) \tag{3.10}$$

These reductions were checked to be locally confluent (and sort decreasing using sorts for actions and programs, respectively) with the Maude Church-Rosser Checker [DM10]. Termination can be shown by multi-set (or recursive) path ordering [Der79] with precedence order $\parallel \succ ; \succ +$ on function symbols (see Appendix A for a proof). Thus, every MCAP has a unique normal form that can be determined in finite time.

3.2.3 Semantics

This Section formalizes MCAP semantics in the context of MCTS interpretation.

3.2.3.1 Search Tree

We introduce a formal representation of the search tree. Its purpose is to accumulate information about computation traces w.r.t. simulation and action choices. Tree nodes represent states $\in \mathcal{S}$ and actions $\in \mathcal{A}$. State nodes $\mathcal{V}_\mathcal{S}$ and action nodes $\mathcal{V}_\mathcal{A}$ alternate (Equations 3.11 and 3.12). Nodes contain aggregation of metadata $\mathcal{D}$ that guides further search. Aggregated data are visitation count and node value (Equation 3.13).

$$\mathcal{V}_\mathcal{S} \subseteq \mathcal{S} \times \mathcal{D} \times 2^{\mathcal{V}_\mathcal{A}} \times \mathcal{P} \tag{3.11}$$
$$\mathcal{V}_\mathcal{A} \subseteq \mathcal{A} \times \mathcal{D} \times 2^{\mathcal{V}_\mathcal{S}} \times \mathcal{P} \tag{3.12}$$
$$\mathcal{D} \subseteq \mathbb{N} \times \mathbb{R} \tag{3.13}$$

While it is possible to use a DAG instead of a tree [SCM12], we will concentrate on the tree setting for the sake of simplicity.

3.2.3.2 Framework Operations

Equations 3.14 to 3.18 show the functional signatures of MCAP framework operations. We will define each one in the rest of this Section. Selection returns an action node for a given state node. Expansion builds a new state node given a state, its value and an MCAP to execute in the new state. Rollout is performed according to a MCAP defining the default policy, from a particular state for a certain number of steps.

$$\text{select} : \mathcal{V}_{\mathcal{S}} \to \mathcal{V}_{\mathcal{A}} \tag{3.14}$$

$$\text{expand} : \mathcal{S} \times \mathcal{P} \times \mathbb{R} \to \mathcal{V}_{\mathcal{S}} \tag{3.15}$$

$$\text{rollout} : \mathcal{S} \times \mathcal{P} \times \mathbb{N} \to \mathbb{R} \tag{3.16}$$

$$\text{update} : \mathcal{V}_{\mathcal{S}} \to \mathcal{V}_{\mathcal{S}} \tag{3.17}$$

$$\text{update} : \mathcal{V}_{\mathcal{A}} \to \mathcal{V}_{\mathcal{A}} \tag{3.18}$$

3.2.3.3 Action Selection

Equation 3.22 shows UCB1 action selection. It is a popular instantiation of the MCTS tree policy based on regret minimization [KS06, ACBF02]. $q(v_a)$ denotes the current value aggregated in the metadata of action node v_a. Let $v_a = (a, d, \vec{v_s}, p) \in \mathcal{V}_{\mathcal{A}}, d = (n, q) \in \mathcal{D}, \vec{v_s} \in 2^{\mathcal{V}_{\mathcal{S}}}$. The following equation then defines the accessor function for action quality metadata of an action node.

$$q((a, (n, q), \vec{v_s}, p)) = q \tag{3.19}$$

$\#(v_s)$ and $\#(v_a)$ denote the number of searches that visited the corresponding node stored in its metadata (see also Algorithm 7, lines 2 and 10). Let $v_s = (s, d, \vec{v_a}) \in \mathcal{V}_{\mathcal{S}}, v_a = (a, d, \vec{v_s}, p) \in \mathcal{V}_{\mathcal{A}}$ and $d = (n, q) \in \mathcal{D}$. The following equations define the accessor functions for count metadata of state and action nodes.

$$\#((s, (n, q), \vec{v_a}, p)) = n \tag{3.20}$$

$$\#((a, (n, q), \vec{v_s}, p)) = n \tag{3.21}$$

UCB1 favors actions that expose high value (first term of the sum), and adds a bias towards actions that have not been well explored (second term of the sum). The parameter c is a constant to control the tendency towards exploration.

$$\text{select}(v_s) = \text{argmax}_{v_a \in \vec{v_a}(v_s)} \left(q(v_a) + c \cdot \sqrt{\frac{2 \ln \#(v_s)}{\#(v_a)}} \right) \tag{3.22}$$

3.2.3.4 Queries

Our framework requires specification of a query representation and a satisfaction function of queries and states to enable conditional computation. Let $\mathcal{V}$ denote variables in the query, and let $\mathcal{O}$ denote objects (i.e. entities) in the domain. Let $\Theta : \mathcal{V} \to \mathcal{O}$ denote substitutions of first order variables, including the empty substitution $\emptyset \in \Theta$. Given these definitions, queries Q are evaluated w.r.t. a given state $\in \mathcal{S}$ and yield a set of substitutions in Θ for query variables (Equation 3.23). It returns the set of substitutions for variables in the query for which the query holds in the state. In case the query is ground and holds, the set containing the empty substitution $\{\emptyset\}$ is returned. If the query does not hold, it returns the empty set $\emptyset$. We write $\vdash$ in infix notation and $s \nvdash q \Leftrightarrow s \vdash q = \emptyset$.

$$\vdash : \mathcal{Q} \times \mathcal{S} \to 2^{\Theta} \tag{3.23}$$

3.2.3.5 Interpretation of MCAPs

Expansion of the tree is constrained by a given MCAP through interpreting it w.r.t a given state. The *potential program function* constrains the search space w.r.t. a given action program and current system state. It maps an MCAP and a given state to the set of normalized MCAPs that result from (a) non-deterministic choices and (b) interpretations of queries.

$$\text{pot} : \mathcal{S} \times \mathcal{P} \to 2^{\mathcal{P}} \tag{3.24}$$

Equations 3.25 to 3.31 define MCAP interpretation by the potential program function inductively over the structure of $\mathcal{P}$. Sum notation denotes multiple choices. We assume that substitution sets resulting from evaluating a query are finite.

$$\text{pot}\,(s, \epsilon) = \emptyset \tag{3.25}$$

$$\text{pot}\,(s, a) = \{a\,;\,\epsilon\} \tag{3.26}$$

$$\text{pot}\,(s, p\,;\,p') = \bigcup_{p'' \in \text{pot}(s,p)} (p''\,;\,p') \tag{3.27}$$

$$\text{pot}\left(s, \sum_i p_i\right) = \bigcup_i \text{pot}(s, p_i) \tag{3.28}$$

$$\text{pot}(s, ?\{q\}\{p\}) = \bigcup_{\theta \in s \vdash q} \text{pot}\,(s, \theta(p)) \tag{3.29}$$

$$\text{pot}(s, \neg?\{q\}\{p\}) = \begin{cases} \text{pot}(s, p) & \textbf{if } s \nvdash q \\ \emptyset & \textbf{otherwise} \end{cases} \tag{3.30}$$

$$\text{pot}(s, \circ\{q\}\{p\}) = \text{pot}(s, ?\{q\}\{p\}\,;\,\circ\,\{q\}\{p\}) \tag{3.31}$$

Reducing the empty program in any state yields the empty set. Reducing a single action yields a program consisting of the action followed by the empty program.

The other reductions are defining the potential programs for composite programs, non-deterministic choices and operations involving queries. They are recursively applying the potential program function, and build sets of potentially resulting programs. These sets represent the non-determinism of choices and query interpretations (i.e. variable substitutions).

3.2.3.6 Search Tree Expansion

Equation 3.32 shows the MCAP expansion mechanism. $s \in \mathcal{S}$ denotes the state for which a new node is added. p is the MCAP to be executed in state s. Potential programs $pot(s, p)$ in normal form define the set of action node children for actions $a \in \mathcal{A}$. These also comprise the corresponding tail program $p' \in \mathcal{P}$. Thus, an MCAP effectively constrains the search space by restricting the set of action node children of a state node. The search tree is expanded when an episode reaches a state that is not represented in the tree. The value of the new node representing this state is estimated by the value $r \in \mathbb{R}$ of the corresponding episode. Note that the initial count is set to zero, and that the metadata of all action node children is initialized with count and value set to zero.

$$\text{expand}(s, p, r) = (s, (0, r), \vec{v}_a, p)$$
$$\text{where } \vec{v}_a = \bigcup_{(a,p') \in \text{pot}(s,p)} (a, (0, 0), \emptyset, p') \tag{3.32}$$

3.2.3.7 Rollout

After expansion a rollout is performed. A number of simulation steps is performed (i.e. until maximum search depth $h_{\max}$ is reached) and the reward for resulting states is aggregated. An MCAP $p \in \mathcal{P}$ defines the rollout's default policy. Actions and corresponding tail programs are selected uniformly random from the set of potential programs in each state $s \in \mathcal{S}$ encountered in the rollout.

$$\text{rollout}(s, p, h) = \begin{cases} R(s) & \textbf{if } h = h_{\max} \\ R(s) + \gamma \cdot \text{rollout}(s', p', h + 1) & \textbf{otherwise} \end{cases}$$
$$\textit{where } (a, p') \sim_{\text{uniform}} \text{pot}(s, p) \wedge s' \sim P(\cdot|s, a) \tag{3.33}$$

3.2.3.8 Value Update

The value of the expanded node is then incorporated to the search tree by value back-propagation along the current search path. In general any kind of value update mechanism is feasible, e.g. a mean update as used by many MCTS variants.

$$\text{update}(v_a) \leftarrow v(v_a) + \frac{r - v(v_a)}{\#(v_a)} \tag{3.34}$$

$$\text{update}(v_s) \leftarrow v(v_s) + \frac{r - v(v_s)}{\#(v_s)} \tag{3.35}$$

Another option for updating node values is dynamic programming (i.e. a Bellman update) [Bel57a]. An action's value is the weighted sum of its successor states' values (Equation 3.36). A state's value is the currently obtained reward and the value of the currently optimal action discounted w.r.t γ and the action's duration (Equation 3.37).

$$\text{update}(v_a) = \sum_{v_s \in \vec{v_s}(v_a)} \frac{\#(v_s)}{\#(v_a)} v(v_s) \tag{3.36}$$

$$\text{update}(v_s) = R(s(v_s)) + \max_{v_a \in \vec{v_a}(v_s)} \gamma^{d(a)} q(v_a) \tag{3.37}$$

Algorithm 7 shows the interplay of selection, aggregation of metadata, simulation, expansion, rollout and value update for Monte Carlo Action Programming.

Algorithm 8 shows the integration of MCAP with online planning. While the system is running, a given MCAP is repeatedly evaluated and executed until termination (lines 2 – 4). Evaluation is performed by MCTS until a certain budget is reached (lines 6 – 8). The currently best action w.r.t. MCAP interpretation is determined (line 9). If there is no such action, the program terminated (line 10). Otherwise, the best action is executed and the outcome observed (lines 13 and 14). In case the new state is already represented in the search tree, the corresponding state node is used as new root for further search (lines 15 and 16). Otherwise, a new root node is created (line 18).

Algorithm 7 Monte Carlo Action Programming

Require: $h_{\max}, R, \text{pot}, \text{simulate}$

procedure MCAP(v_s, h)
  $\#(v_s) \leftarrow \#(v_s) + 1$ ▷ increase state node count
  if $h = h_{\max}$ **then**
    return ▷ reached maximum search depth
  end if
  if $\vec{v_a}(v_s) = \emptyset$ **then**
    return ▷ no action is available
  end if
  $v_a \leftarrow \text{select}(v_s)$ ▷ select action node
  $\#(v_a) \leftarrow \#(v_a) + 1$ ▷ increase action node count
  $s' \sim P(\cdot|s, a)$ ▷ simulate action outcome
  if $\exists v_{s'} \in \vec{v_s}(v_a) : s(v_{s'}) = s'$ **then** ▷ successor state node exists
    MCAP($v_{s'}, h + 1$) ▷ recursive call through the tree
    $v_a \leftarrow \text{update}(\text{v}_\text{a})$ ▷ update action quality
    $v_s \leftarrow \text{update}(\text{v}_\text{s})$ ▷ update state value
  else
    $r \leftarrow \text{rollout}(s', p(v_a), h)$ ▷ estimate state node value
    $v_{s'} \leftarrow \text{expand}(s', p(v_a), r)$ ▷ create successor state node
    $\vec{v_s}(v_a) \leftarrow \vec{v_s}(v_a) \cup \{v_{s'}\}$ ▷ add new state node to successors
  end if
end procedure

Algorithm 8 Online MCAP

Require: initial state s_{init}, MCAP p_{init}

$v_{\text{init}} \leftarrow \text{expand}(s_{\text{init}}, p_{\text{init}})$ ▷ initial state node (contains action nodes)
while running **do**
  ONLINE-MCAP(v_{init})
end while

procedure ONLINE-MCAP(v_s)
  while budget **do**
    MCAP(v_s) ▷ repeatedly update v_s w.r.t. MCAP
  end while
  $v_a^{\max} \leftarrow \text{argmax}_{v_a \in \vec{v_a}(v_s)} q(v_a)$ ▷ get optimal action
  if $v_a^{\max} = \text{null}$ **then**
    return ▷ MCAP terminated
  end if
  execute $a(v_a^{\max})$ ▷ execute action
  observe s' ▷ observe the outcome
  if $v_{s'} \in \vec{v_s}(v_a^{\max})$ **then** ▷ previously considered situation
    $v_s \leftarrow v_{s'}$ ▷ reuse planning result
  else ▷ previously unconsidered situation
    $v_s \leftarrow \text{expand}(v_s, p(v_a^{\max}), 0)$ ▷ new root, no value information
  end if
end procedure

3.3 MCAP Extension of the OnPlan Framework

in this Section, we extend the OnPlan framework introduced in Chapter 2 to provide support for Monte Carlo Action Programming. As action choice in Mcap is depending on the given program and its possible execution traces, we have to adapt the OnPlan framework when acting and generating episodes to keep track of the remaining tail program to execute. Therefore, we extend the basic components for agent and planner to maintain a reference to the current program to be executed. We accordingly adapt the generation of episodes. We also show how Mcap instantiates the operations for sampling actions from a strategy and for updating the strategy based on generated episodes. I.e. we show how to sample from a strategy that is formalized as an Mcap search tree, and how to update such a strategy based on such samples.

3.3.1 OnPlan Support for MCAP

In order to express Mcap in the OnPlan framework, we extend the component model and the corresponding behavioral algorithms to support online execution of non-deterministic Monte Carlo action programs.

3.3.1.1 Extended Component Model

We extend the component model from Chapter 2 by class *McapPlanner* that specializes *SimPlanner* to represent a planning entity using the Mcap approach. We also introduce a class *Program* that represents Monte Carlo action programs. An *McapPlanner* maintains a reference to such a program. Figure 3.2 shows the extended component model of OnPlan.

A key change to the original framework is that the strategy now is a probability distribution of programs depending on a given state and a given program. This reflects that fact that interpretation of a program in a particular state yields a set of choices which are themselves programs. We require that the strategy always yields a program when sampling from it. Thus, P_{act} now changes its form accordingly to $P_{\text{act}}(\mathcal{P}|\mathcal{S} \times \mathcal{P})$.

3.3.1.2 MCAP Agent Behavior

The agent behavior of the original OnPlan framework is adjusted to support Mcap execution. When an action is required, the agent now samples a normal form program from its planner's current strategy. It then sets the planners reference to the new tail program, before executing the first action of the sampled program. The other parts of the original OnPlan agent behavior do not change, implementing the online planning approach of interleaved execution and deliberation at runtime. Algorithm 9 shows the Mcap adjusted agent behavior in the OnPlan framework.

3.3.1.3 MCAP Episode Generation

When generating episodes from the simulation, two changes affect the OnPlan framework when incorporating Monte Carlo action programming.

1. Episodes now have to comprise information about the program to be executed at each step. This is necessary, as the search tree representing the strategy is expanded by a new node when an episode leaves the tree. This new node carries information about the program to be executed from there on, so we keep it in the episode.

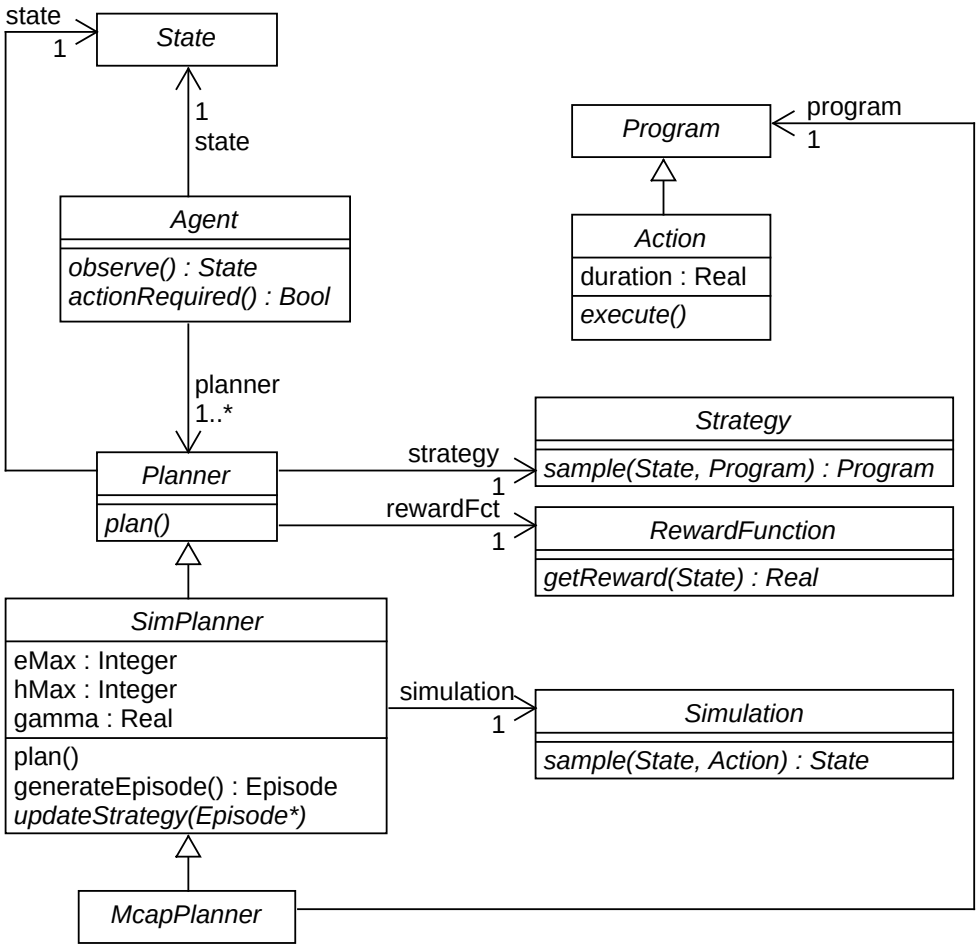


Figure 3.2: MCAP *extension of the* ONPLAN *framework*

Algorithm 9 MCAP Agent Behavior

Require: Local variable action $\in \mathcal{A}$

```
1: while program ≠ nil do
2:     while !actionRequired() do
3:         state ← observe()                          ▷ observe current state
4:         planner.state ← state                      ▷ inform planner
5:     end while
6:     (action ; program) ← planner.strategy.sampleProgram(s, p)
                                                      ▷ sample new program from strategy
7:     planner.program ← program                      ▷ inform planner
8:     action.execute()                               ▷ execute head of new program
9: end while
```

2. Episode generation should be constrained by the planner's Monte Carlo action program. Thus, we now sample a program to be executed from the current strategy w.r.t. current simulation state and currently simulated program.

MCAP Episodes As we need remaining programs when updating the strategy given as a search tree, we will keep track of the corresponding information in the episodes. We adjust the definition of an episode to support MCAP programs. An episode is now a list of triples. Each triple contains the current state, the program to execute (in order to preserve choices for expansion), and the effectively simulated action.

$$\mathcal{E}_{\mathcal{P}} \subseteq (\mathcal{S} \times \mathcal{P} \times \mathcal{A})^* \tag{3.38}$$

Episode Generation We reformulate the generation of episodes (see Algorithm 5 in Chapter 2) to support Monte Carlo Action Programs. The planner has to (1) sample successor states from the simulation according to the current strategy which yields a program when samples while (2) keeping track of the remaining program tail to be executed when generating episodes.

Algorithm 10 reflects these tasks. The generation starts with the planner's current state and program (lines 2 – 4). Episodes are generated up to the maximum search depth $h_{\max}$ (line 5). If the action program terminated, generation of the episode stops (lines 6 – 8). Otherwise, the current program is interpreted w.r.t. current state and strategy. I.e. a new program to be executed is sampled from the current strategy (line 9). Then, current state, program and action to be executed are appended to the generated episode (line 10). The simulation is queried for a successor state (line 11), and the program used for further episode generation is set to be the tail program sampled from the strategy (line 12).

When the maximum search depth has been reached or the action program terminated, the final state and the remaining program (which may be a nil program) are added to the episode (line 14). As no action is executed, the corresponding data remains unset in the triple.

Algorithm 10 Generating weighted episodes in MCAP

Require: Local variables s $\in \mathcal{S}$, p, p' $\in \mathcal{P}$, a $\in \mathcal{A}$, e $\in \mathcal{E}_{\mathcal{P}}$

```
 1: procedure generateEpisode
 2:     s ← state
 3:     p ← program
 4:     e ← nil
 5:     for 0 ... h_max do
 6:         if p = nil then
 7:             break
 8:         end if
 9:         (a ; p') ← strategy.sampleProgram(s, p)
10:         e ← e :: (s, p, a)
11:         s ← simulation.sampleSuccessor(s, a)
12:         p ← p'
13:     end for
14:     e ← e :: (s, p, nil)
15:     return e
16: end procedure
```

3.3.2 MCAP Search Trees as Strategies

When instantiating the OnPlan framework with Mcap, strategies are represented as search trees. Recall that strategies represent probability distributions of programs conditioned to a current state and a current program: $P_{\text{act}}(\mathcal{P}|\mathcal{S}\times\mathcal{P})$. In fact, a strategy should reflect the interpretation of a program in a particular state (yielding potential choices) and at the same time weight these choices to yield high-value choices with a high probability.

We define a set of search trees $\mathcal{T}$. Each search tree $t \in \mathcal{T}$ consists of two sets of state nodes and actions nodes as defined in the Mcap framework (Equations 3.11 and 3.12).

$$\mathcal{T} \subseteq 2^{\mathcal{V}_\mathcal{S}} \times 2^{\mathcal{V}_\mathcal{A}} \tag{3.39}$$

In the following, we denote a state node $(s, d, \vec{v}_a, p) \in \mathcal{V}_\mathcal{S}$ by $v_{s,p}$. Similarly, we denote an action node $(a, d, \vec{v}_s, p) \in \mathcal{V}_\mathcal{A}$ by $v_{a,s,p}$.

We say a node is an element of a search tree if the corresponding set contains the node. Let V_s be a set of state nodes, and V_a a set of action nodes. Let $t = (V_s, V_a) \in \mathcal{T}$ be the corresponding search tree. Let $v_{s,p} \in \mathcal{V}_\mathcal{S}$ be a state node, and $v_{a,s,p} \in \mathcal{V}_\mathcal{A}$ an action node.

$$v_{s,p} \in t \Leftrightarrow v_{s,p} \in V_s \tag{3.40}$$
$$v_{s,p} \notin t \Leftrightarrow v_{s,p} \notin V_s \tag{3.41}$$
$$v_{a,s,p} \in t \Leftrightarrow v_{a,s,p} \in V_a \tag{3.42}$$
$$v_{a,s,p} \notin t \Leftrightarrow v_{a,s,p} \notin V_a \tag{3.43}$$

In the following, we discuss how to sample a particular program choice from such a strategy represented by an Mcap search tree, given a current state and an Mcap program, and how to generate episodes based on such a strategy (Section 3.3.2.1). We then show how a strategy is updated w.r.t. a generated episode (Section 3.3.2.2).

3.3.2.1 MCAP Sampling

Sampling a program from the strategy represented as a search tree then follows the principle of Monte Carlo Tree Search. When we sample a program in a situation where a node is maintained to collect statistics about this configuration of state s $\in \mathcal{S}$ and program p $\in \mathcal{P}$, then we use the UCB1 selection mechanism to decide on the program to execute next. It is in fact the program (a ; p') that is given by the action node child $v_{a,s,p'}$ which exposes the maximal UCB1 score. If we sample for a situation that is not yet represented in the tree, we use a uniform random selection of potential programs to execute further, i.e. (a ; p') is uniformly sampled from pot(s, p). Algorithm 11 shows the sampling of programs from a strategy represented by a search tree $t \in \mathcal{T}$.

3.3.2.2 MCAP Updating

Updating a strategy represented as an Mcap search tree based on a generated episode consists of two main operations.

1. Add a new state node to the search tree where the episode leaves the current search tree. Initialize this state node's children w.r.t. the program to execute stored in the episode at that point. Set the corresponding state node's value to the discounted accumulated sum of rewards of the remainder of the episode.

Algorithm 11 Sampling programs in MCAP

Require: Search tree $t \in \mathcal{T}$

```
1: procedure samplePROGRAM(s, p)
2:     if v_{s,p} ∈ t then                                  ▷ inside the tree
3:         v_{a,s,p'} ← select(v_{s,p})
4:         return (a; p')
5:     else                                                 ▷ outside the tree
6:         return (a; p') ∼_uniform pot(s, p)
7:     end if
8: end procedure
```

2. Update count and value of all nodes that have been traversed by the episode.

Discounted Accumumlated Epsiode Rewards To enable the first task, we define a function $R_e : \mathcal{E}_\mathcal{P} \to \mathbb{R}$ that determines the discounted accumulated sum of rewards of a given episode. This function is recursively defined using the reward function $R : \mathcal{S} \to \mathcal{A}$ from the framework specification. Let $s \in \mathcal{S}, p \in \mathcal{P}, a \in \mathcal{A}$ and $e \in \mathcal{E}_\mathcal{P}$.

$$R_e(\text{nil}) = 0 \tag{3.44}$$

$$R_e((s, p, a) :: e) = R(s) + \gamma^{d(a)} R_e(e) \tag{3.45}$$

Updating the Search Tree In the MCAP framework, the search tree is updated after generation of each episode, thus $e_{\max} = 1$. Accordingly a single epsiode is passed as an argument to the *updateStrategy* operation. Updating the strategy is done by performing the following two tasks.

1. Adding a new node to the search tree where the episode leaves the tree.
2. Updating the values of all previously traversed nodes w.r.t. reward of the episode.

Algorithm 12 shows the updating of a strategy search tree w.r.t. a given episode. The algorithm extracts the first triple (s, a, p) of the current episode containing state s, program p to be executed and effectively executed action a (line 5). The algorithm then checks whether the search tree contains a corresponding node $v_{s,p}$ (line 6). Two possible results may occur.

1. In case the node $v_{s,p}$ is part of the tree, the current episode triple (s, p, a) is cached (line 7). Then the procedure is called iteratively on the remainder of the episode (line 8) before updating values of the nodes in the tree that correspond to the current triple (s, a, p) (line 9). The recursive call in line 8 effectively causes a bottom up procedure, which first adds a new node if the episode leaves the existing search tree at some point before updating the traversed nodes in a bottom up fashion.
2. In case the node $v_{s,p}$ does not exists as part of the tree, it is created according to node expansion as defined in Equation 3.32 (line 11). The new node's value is set to $R_e(e)$ of the currently remaining episode. The new node is then added to the tree: Either as a new root node if no previously executed episode triple has been cached in case it was the first triple (lines 12 – 13), or as a child node of the action node that was traversed by the episode as last node before leaving the tree (line 15).

Algorithm 12 MCAP strategy update

Require: Search tree $t \in \mathcal{T}$

1: **procedure** UPDATESTRATEGY(e)
2: **if** e = nil **then**
3: **return** ▷ done
4: **end if**
5: (s, p, a) ← head(e)
6: **if** $v_{s,p} \in t$ **then** ▷ state node in tree
7: $(s_\mathrm{p}, p_\mathrm{p}, a_\mathrm{p}) \leftarrow (s, p, a)$ ▷ set predecessor
8: UPDATESTRATEGY(tail(e)) ▷ update tree bottom up
9: BELLMANUPDATE($v_{s,p}, v_{a,s,p}$) ▷ update node metadata
10: **else** ▷ state not in tree
11: $v_{s,p} \leftarrow$ expand(s, p, R_e(e)) ▷ create new node
12: **if** $(s_\mathrm{p}, p_\mathrm{p}, a_\mathrm{p}) =$ nil **then** ▷ no predecessor
13: $t \leftarrow t \cup \{v_{s,p}\}$ ▷ add new node as tree root
14: **else** ▷ predecessor exists
15: $\vec{v}_s(v_{a_\mathrm{p},s_\mathrm{p},p_\mathrm{p}}) \leftarrow \vec{v}_s(v_{a_\mathrm{p},s_\mathrm{p},p_\mathrm{p}}) \cup \{v_{s,p}\}$ ▷ add new node to predecessor
16: **end if**
17: **end if**
18: **end procedure**

Updating node counts and values can be done by a Bellman backup. The corresponding operation is shown in Algorithm 13. Both count and value updates affect the UCB1 scores of the corresponding nodes, potentially changing the selection of program execution choices in future episode generation.

Algorithm 13 MCAP Bellman update

1: **procedure** BELLMANUPDATE(v_s, v_a)
2: $\#(v_a) \leftarrow \#(v_a) + 1$
3: $v(v_a) \leftarrow \sum_{v'_s \in \vec{v}_s(v_a)} \left(\frac{\#(v'_s)}{\#(v_a)} v(v'_s) \right)$
4: $\#(v_s) \leftarrow \#(v_s) + 1$
5: $v(v_s) = R(s(v_s)) + \max_{v'_a \in \vec{v}'_a(v_s)} \gamma^{d(a(v'_a))} v(v'_a)$
6: **end procedure**

Alternatively, node counts and values can be updated in an incremental mean fashion by replacing the call to BELLMANUPDATE in line 9 of Algorithm 12. In this case, the update operation has an additional argument yielding the reward $R_{\mathcal{E}}(e)$ of the currently remaining episode e when calling the operation. Algorithm 14 shows the corresponding procedure.

3.4 Experimental Evaluation

We empirically evaluated the effectiveness of MCAP by comparing it to a pure MCTS planner where the search space is not constrained by a non-deterministic program.

Algorithm 14 MCAP mean update

1: **procedure** MEANUPDATE(v_s, v_a, r)
2: $\#(v_a) \leftarrow \#(v_a) + 1$
3: $v(v_a) \leftarrow v(v_a) + \frac{r - v(v_a)}{\#(v_a)}$
4: $\#(v_s) \leftarrow \#(v_s) + 1$
5: $v(v_s) \leftarrow v(v_s) + \frac{r - v(v_s)}{\#(v_s)}$
6: **end procedure**

3.4.1 Setup

We used the search and rescue domain introduced in the introduction (see Section 1.3.1 in Chapter 1) for evaluating the effectiveness of the MCAP framework. We will shortly recall the domain here. The domain consists of a connected graph of 20 positions. The degree of connectivity is 0.3, resulting in 6 to 7 connections per position. Each position may be on fire, initially ten fires are placed. Ten victims are randomly located, as well as three ambulances (pos.safe = true). An ambulance position never catches fire. Victims are initially located at positions that do not carry an ambulance. A rescue agents is located at a random non-ambulance position. It can move to connected, non-burning positions and lift or drop victims. The number of victims an agent can carry is limited by its capacity, which is set to two in the experiments. At every time step the fire attribute of a position may change depending on how many of the position's neighbors are on fire.

An agent can execute different possible actions. Note that, due to parameter instantiation, effectively a much larger number of action instances is executable in each concrete state.

1. *Move*(R, P): Robot R moves to target position P if it is connected to the robot's current position and is not on fire.
2. *Extinguish*(R, P): Robot R extinguishes fire at a neighbor position P.
3. *Lift*(R, V): Robot R lifts victim V (at same location) if it has capacity left.
4. *Drop*(R, V): Robot R drops lifted victim V at the current location.
5. *Noop*: Does nothing.

Effectiveness of the MCAP framework was evaluated empirically for the rescue domain. A simulation of the domain was used as generative model. Reward $R(s)$ was defined as the number of victims located at safe positions in state s. Maximum search depth was set to $h_{\max} = 20$ and the discount factor was set to $\gamma = 0.9$. These parameters settings yielded sensible results in preliminary experiments, where they were manually tuned. Experiments were conducted with randomized initial states, each consisting of twenty positions with 30% connectivity. Three positions were safe, ten victims and ten fires were located randomly on unsafe positions. Robot capacity was set to two. This setup yields a state space containing more than 10^{19} possible states. Fires ignited or ceased probabilistically at unsafe positions. Actions succeeded or failed probabilistically (p = 0.05). This yields a branching factor of $2 \cdot 2^{17}$ for each action.

We used incremental mean node value update in all experiments (see Algorithm 14). In the experiments using plain MCTS all potentially executable actions were evaluated at each step. Algorithm 15 shows pseudocode for the program used to determine

the action to evaluate in the experiments with MCAP. $\mathcal{A}_{\text{exec}}$ denotes the set of all executable actions in the current state, where action parameters have been instantiated accordingly (line 7). Both MCTS and MCAP were provided 0.2 seconds at each step for action evaluation.

Algorithm 15 Pseudocode of the MCAP used in the experiments

```
1: while true do
2:     if self.position.safe ∧ self.victims ≠ ∅ then
3:         ∑_{v∈self.victims} self.drop(v)
4:     else if ¬(self.position.safe) ∧ self.position.victims ≠ ∅ then
5:         ∑_{v∈self.position.victims} self.lift(v)
6:     else
7:         ∑_{a∈A_exec} a
8:     end if
9: end while
```

3.4.2 Results

The experiments yielded the following empirical results.

3.4.2.1 Accumulated Reward

Accumulated reward while system execution is the absolute metric to be optimized by any online planner. We compared the absolute accumulated reward of both the MCAP agent and the MCTS agent per step of the simulation. Figure 3.3 shows the measured data as boxplots for every fifth step of system execution. The MCAP agent is able to accumulate reward more effectively that its pure MCTS counterpart. This isshows the effectiveness of constraining the search space according to expert knowledge that is available at system design time by a non-deterministic Monte Carlo action program.

3.4.2.2 Estimation of Expected Reward

As pure Monte Carlo Tree Search, MCAP is used to estimate expected future reward of actions based on simulation data. Figure 3.4 shows the mean expected reward of actions that were actually executed by the agents. The estimate is increasing in absolute magnitude due to the stepwise summation of reward (reward is given for save victims at each step, so starting estimation in a step where already many victims are safe yields a high future reward estimation). From step 60 onwards, expected reward declines as both agents were informed about termination of the experiment and terminated their simulations accordingly.

Figure 3.4 shows that MCAP and MCTS show very different absolute value estimates. Therefore, standard deviation for itself does not provide an accurate measure for estimation quality. We computed the normalized coefficient of variation (CV) which is a measure of dispersion of a data sample that is comparable also if sampled populations expose a different absolute mean. Let $\bar{r}_a$ denote the average value of an executed action a, let σ_a denote the standard deviation of the rewards measured for that node and let n_a be the node's visitation count. Then, the normalized CV computes as follows.

$$\frac{s_a/\bar{r}}{\sqrt{n_a}} \tag{3.46}$$

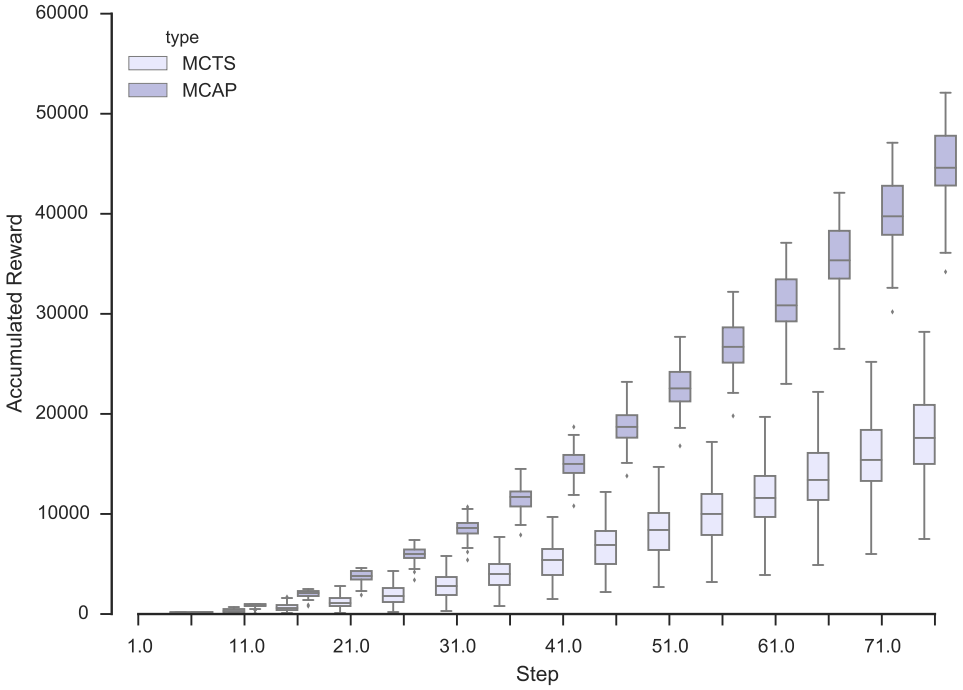


Figure 3.3: *Reward accumulated by* MCAP *and* MCTS

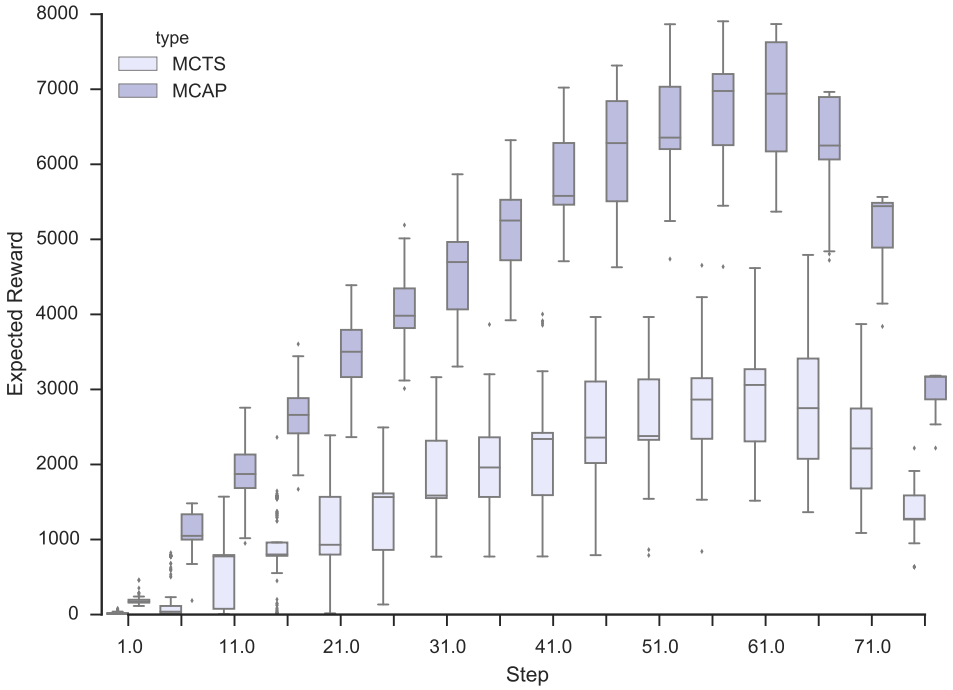


Figure 3.4: *Expected reward per step for* MCAP *and* MCTS.

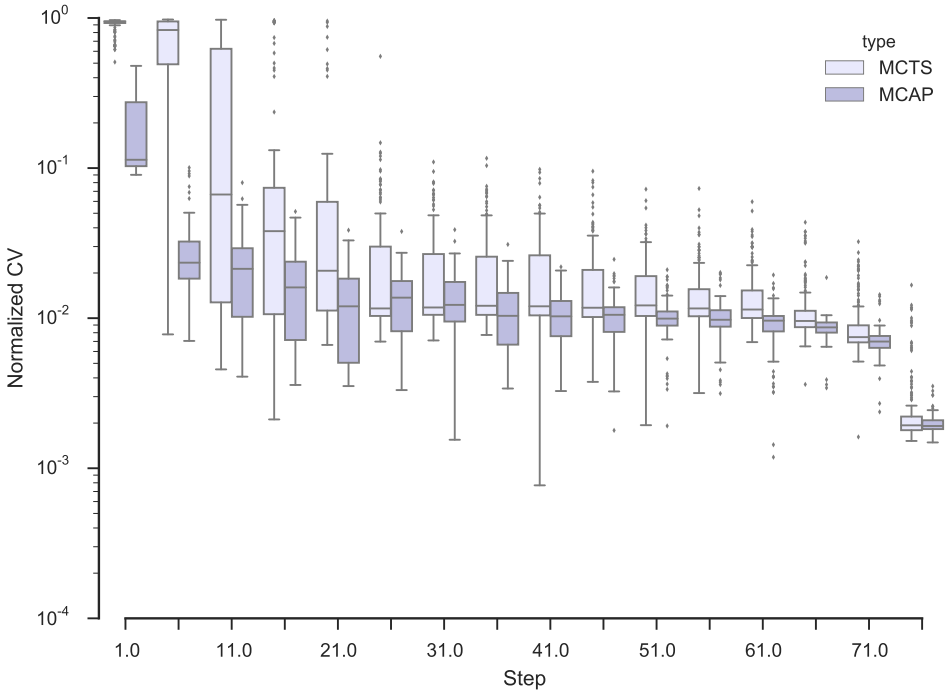


Figure 3.5: *Normalized CV per step for* MCAP *and* MCTS

A low normalized CV thus indicates high accuracy of the estimation. Accuracy thus increases by either (a) reducing standard deviation of samples or (b) increasing samples drawn from the distribution.

Figure 3.5 compares the corresponding normalized CV values of actions executed by the MCAP and MCTS planning agents at a each step of the experiment. The MCAP planner is able to increase its estimation accuracy much faster that the MCTS planner. In comparison to pure MCTS search, the search space that is constrained by the MCAP program yields lower variance and thus lower standard deviation of reward accumulated in the generated episodes. This in turn yields more stable estimates after shorter amounts of system execution time.

3.4.2.3 Estimation Gain

One of the key ideas of simulation based online planning is to compile information from simulations to increase the estimation quality of a strategy before actual action execution. We measured the amount of gain, both in terms of expected reward and normalized CV for all root state nodes that were traversed in system execution. The gain of such a measure was defined as the quotient of the measure directly after executing an action (i.e. before generating any episode from the current root state node) and directly before executing the next action (i.e. after generating episodes and updating the strategy for 0.2 seconds).

Figure 3.6 shows the corresponding gain of average expected value of the root state node at all steps of the experiment. MCAP is able to increase the estimated value of the current state better than pure MCTS.

Figure 3.7 shows the gain of normalized CV for all states traversed in the experiments. MCAP reduces estimation quality stronger than pure MCTS in the beginning of

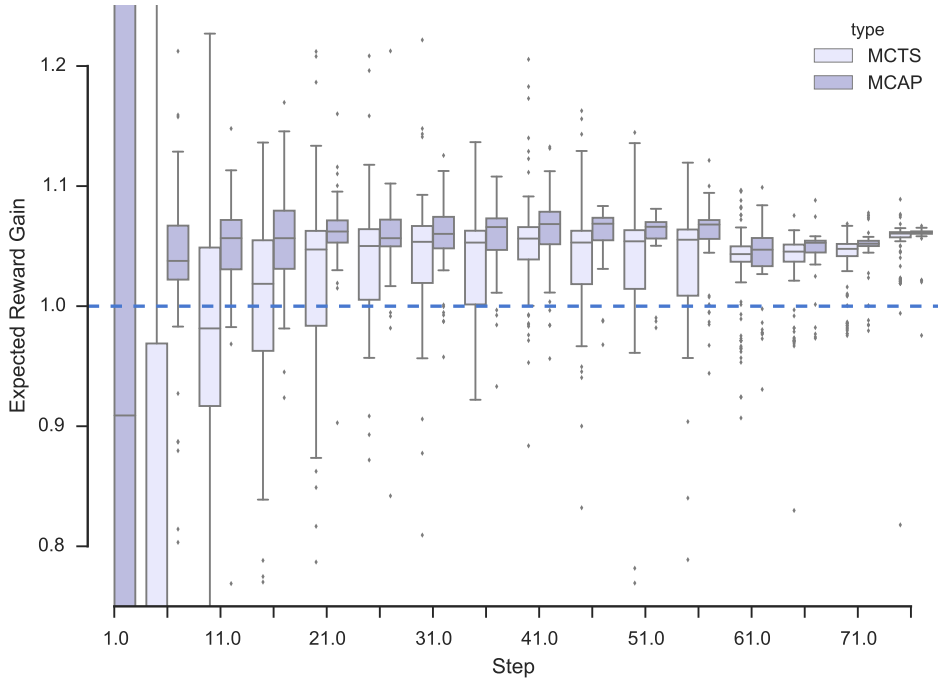


Figure 3.6: *Expected reward gain per step for* MCAP *and* MCTS

an episode (for about the first 15 steps). After 15 steps, while still some gain is observable, it starts to become smaller for MCAP. The same effect can be observed for MCTS after about 45 steps. This effect – the *diminishing return* – occurs as at some point, the planner has compiled so much information from the previous simulation episodes already that future episodes only provide fewer additional information. MCAP reaches this point earlier than MCTS due to the constrained search space.

3.4.2.4 System Performance

System performance was measured with the statistical model checker Multivesta [SV13]. Two metrics of system behavior with and without MCAP search space constraints were assessed: Ratios of safe victims and burning victims.

Figure 3.8 compares the average results for behavior synthesis with plain MCTS and with MCAP within a 0.95 confidence interval. The effect of MCAP search space reduction on system performance can clearly be seen. The configuration making use of online MCAP interpretation achieves larger ratios of safe victims and manages the reduction of burning victim ratios better than the configuration not making use of MCAP. With plain MCTS, search is distracted by low reward regions due to avoiding burning victims. MCAP search identifies high reward regions where victims are saved within the given budget.

A similar experiment with unexpected events illustrates robustness of the approach. Here, every twenty steps all currently carried victims fell to the ground (i.e. were located at their carrier's position). Also, fires ignited such that overall at least ten fires were burning immediately after these events. Note that the simulation of the domain used for plain MCTS and MCAP did *not* simulate these events. The planning system managed to recover from the unexpected situations autonomously (Figure 3.9). As for

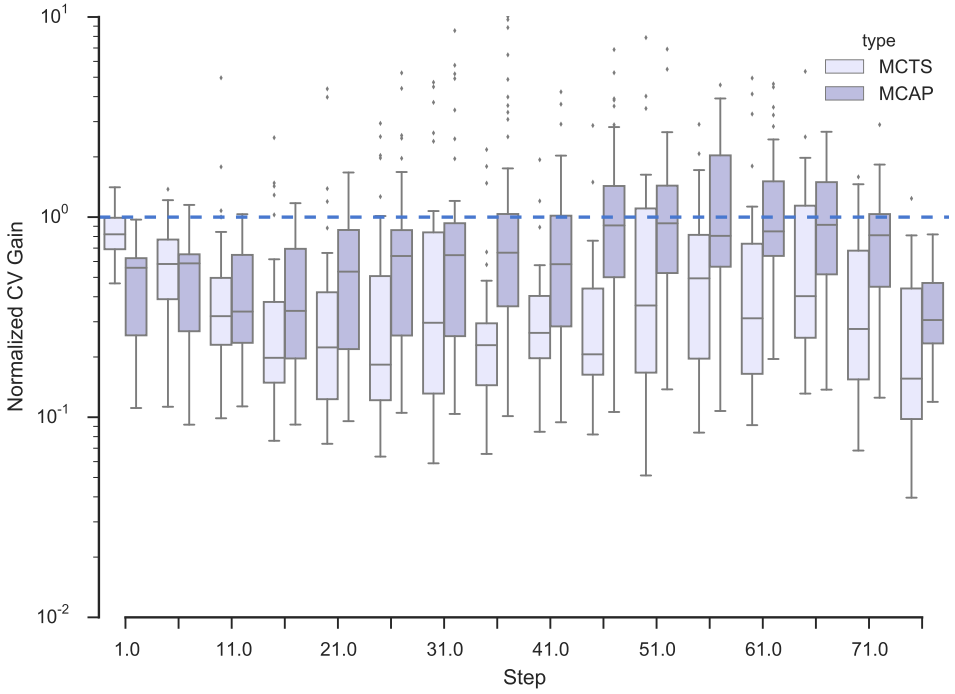


Figure 3.7: *Normalized CV gain per step for* MCAP *and* MCTS

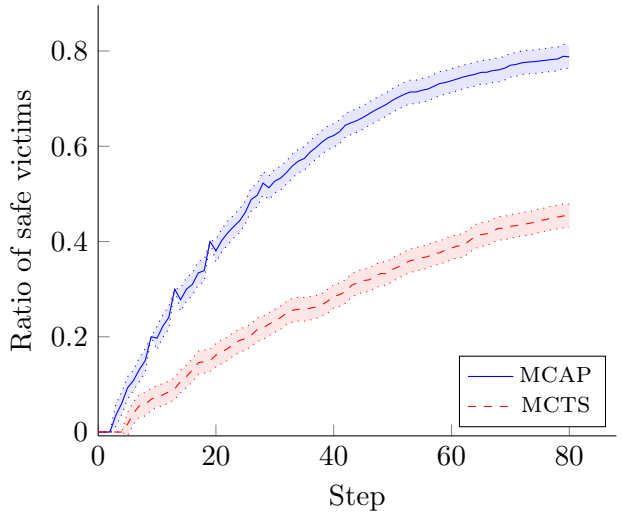
the basic experiment, the configuration with MCAP performed significantly better that the configuration using plain MCTS.

In a third experiment the reward function was changed unexpectedly for the system. Before step 40, a reward is provided exclusively for avoiding burning victims. Onwards from step 40 on the reward function from the previous experiments was used, providing reward for safe victims. The planner did *not* simulate the change of reward when evaluating action traces. The MCAP system outperformed the plain MCTS planner by reacting more effectively to the change of reward function. Figure 3.10 shows the results of this experiment.

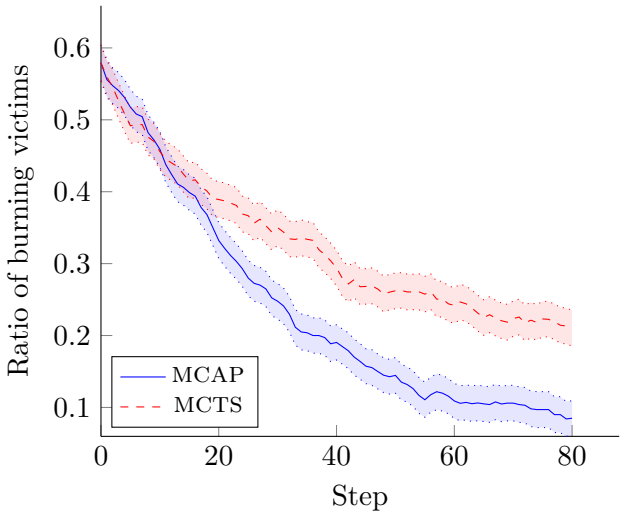
3.5 Related Work

To the best of the author's knowledge, there is no other non-deterministic action programming language that is probabilistically interpreted at runtime in an online planning manner. For in-depth information on Monte Carlo Tree Search, we refer to [BPW+12]. Action programming with symbolic interpretation (i.e. as logic programs) has been exhaustively investigated in the literature. For more information on symbolic action programming, we refer to GOLOG [DGLL00, GLLS09], FLUX [Thi05], or action programming in rewriting logic [Bel13, Bel14]. In contrast to these approaches, MCAP makes use of online planning and importance sampling to solve the exploitation-exploration problem, enabling scalability to very large domains which would be intractable with purely symbolic approaches.

PROBLOG is a probabilistic logic programming language where horn clauses in the knowledge base are explicitly annotated with numerical values corresponding to their uncertainty [DRKT07, KDDR+11]. Bayesian inference is used to determine the uncer-

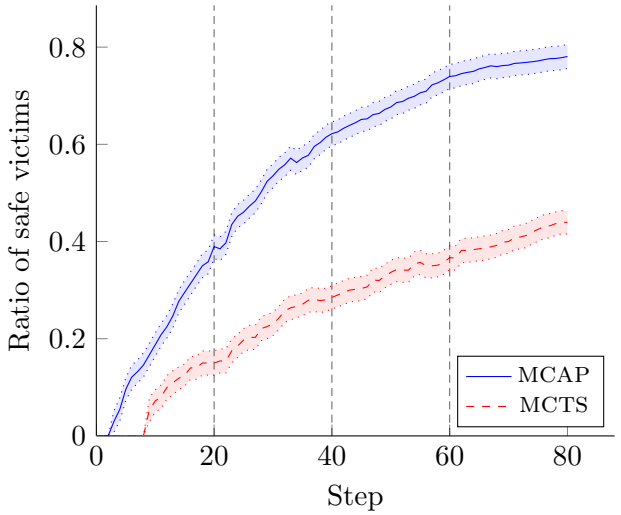


(a)

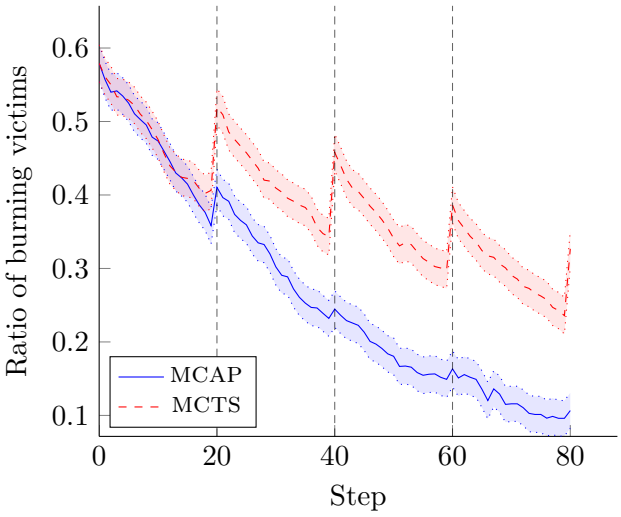


(b)

Figure 3.8: *Comparison of (a) safe and (b) burning ratios for* MCTS *and* MCAP.

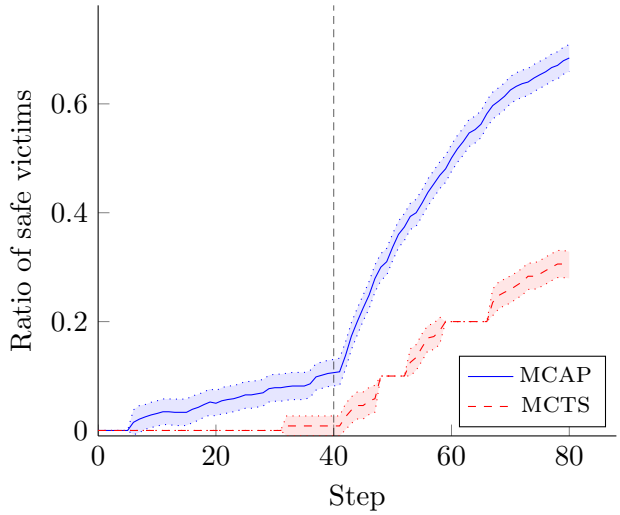


(a)

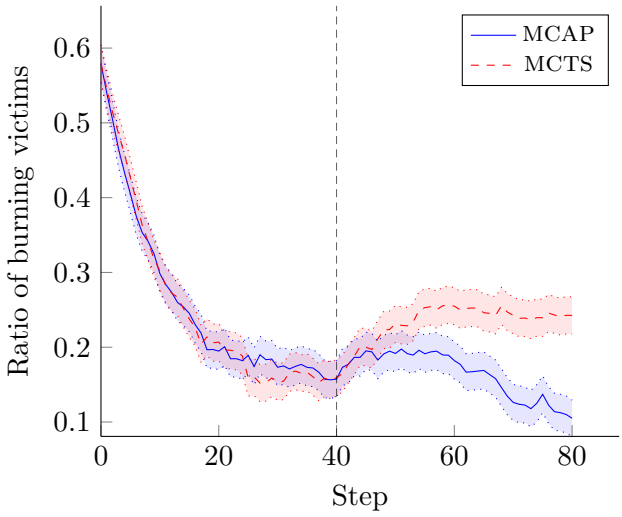


(b)

Figure 3.9: *Comparison of (a) safe and (b) burning ratios for* MCTS *and* MCAP *despite unexpected events every 20 steps. See text for details.*



(a)



(b)

Figure 3.10: *Comparison of (a) safe and (b) burning ratios for* MCTS *and* MCAP *despite unexpected change of system goal at step 40. See text for details.*

tainty about some given query. The language also supports to incorporate observations into the inference process. In contrast to MCAP, PROBLOG is a declarative language for performing symbolic logical inference with uncertainty. However, this uncertainty is not exploited to decide where to put evaluation effort, as inference in logic programming is not necessarily used for decision making. Nevertheless, an application of PROBLOG to online decision making in the spirit of MCAP would be an interesting venture.

3.6 Summary & Outlook

This Chapter introduced Monte Carlo Action Programming, a programming language framework for autonomous systems that act in large probabilistic state spaces. It comprises formal syntax and semantics of a nondeterministic action programming language. The language is interpreted stochastically via Monte Carlo Tree Search. The effectiveness of search space constraint specification in the MCAP framework was shown empirically. Online interpretation of MCAPs provides system performance robustness in the face of unexpected events.

A possible venue for further research is the extension of MCAP to domains with continuous time and hybrid systems. Here, discrete programs are interpreted w.r.t. continuously evolving domain values [ACH+95]. Another important aspect is to investigate specification and verification of system goals (e.g. as LTL formulae) in utility based program interpretation as provided by MCAP. It would also be interesting to evaluate to what extent manual specification techniques as MCAP could be combined with online representation learning (e.g. statistical relational learning [Get07] and deep learning [HOT06]): How to constrain system behavior if perceptual abstraction is unknown at design time or changes at runtime? Also, application of the importance sampling principle for evaluation of continuous choices (cf. Chapter 5) would be an interesting direction for further research.

Chapter 4

Relational Probabilistic Action Forests

Online planning as discussed in the previous Chapters enables autonomous system behavior synthesis at runtime w.r.t. specified system goals, accounting for probabilistic action effects and unexpected events. A popular variant of online planning is Monte-Carlo Tree Search [BPW+12]. Possible future trajectories depending on system action choices are repeatedly sampled from a predictive model of the system domain (i.e. a simulation) and evaluated to form an action recommendation. The recommended action is executed, the result observed, and the planning process repeats.

Performance of online planning correlates with quality of prediction [HS13]. A predictive domain model manually specified at design time may yield poor predictive quality at runtime, either due to specification errors or due to unexpected change of domain dynamics. Learning a predictive model from observations made at runtime allows to identify and recover from predictive inaccuracy due to erroneous specification or unexpected change of domain dynamics. Learning capabilities enable an online planner to maintain performance autonomously.

In this Chapter, we propose *Relational Probabilistic Action Forests* (RPAF), an algorithm that learns decision forests [CSK12] as a predictive model for action effects from relational probabilistic runtime observations.[1] In effect, the algorithm is able to generalize from observations that are given in form of a complex data type. The algorithm generalizes over discrete relational training data, yielding fast learning convergence by exploiting structure in perception data. In particular, relational learning generalizes faster than learning with propositional representations if the domain can be represented relationally [DRB01, Dri10]. We consider discrete state and time settings. We concentrate on compilation of relational perception data into a decision forest. The forest yields a probabilistic predictive model of domain dynamics and effects of system interaction with the environment.

RPAF learns human-readable white-box models of system domains, which may provide information valuable for specifying and training subsequent system generations. Learned white-box models can be communicated to humans that have to interact with

[1] This Chapter is based on a previous publication by the author together with Alexander Neitz [BNar]. The author of this thesis contributed the idea for the approach, structure and writing of the previously published paper, and the formalization of the approach. Work on conceptual details, implementation and experimental evaluation were realized by both authors. Alexander Neitz studied preliminary ideas on the approach in his Bachelor thesis, which was supervised by the author of the thesis on hand.

the system, which may in turn increase acceptance of the autonomous adaptive system by potential human collaborators [SGG+13].

This Chapter provides the following contributions.

1. We propose *Relational Probabilistic Action Forests*, an approach for learning probabilistic predictive action models for relational data with random forests.
2. We propose *set-adjusted Gini impurity* for assessment of relational generalization in the context of relational probabilistic action forests.
3. We perform an empirical evaluation of relational probabilistic action forests on a relational example domain and integrate the approach with a Monte Carlo tree search planner.

Section 4.1 briefly reviews decision trees and random forests. Section 4.2 introduces relational probabilistic action forests and an algorithm for learning them from relational training data. Section 5.3 presents empirical results for various experiments. We discuss related work in Section 4.4. We conclude and sketch venues for further research in Section 5.6.

4.1 Preliminaries

4.1.1 Decision Trees

A decision tree is a classifier that assigns a discrete probability distribution for given data D belonging to a particular class of data C.

$$D \to P(C) \tag{4.1}$$

A decision tree can be learned from a set $X \subseteq D \times C$ of training samples.

$$2^X \to (D \to P(C)) \tag{4.2}$$

Learning a decision tree statistically identifies equivalence classes on sample data. These are defined by queries Q. Queries are associated with nodes in the tree. A path in the tree can be considered as a conjunction of queries. Potential queries are a parameter of the learning algorithm. A query maps data D to a discrete, finite value V (e.g. a boolean value or an enumeration type).

$$Q = D \to V \tag{4.3}$$

For each distinct value resulting from querying a training sample, a new child node is added to the current node. Samples are distributed to child nodes w.r.t. the query result.

$$\text{split} : 2^X \times Q \to (2^X)^{\mathbb{N}} \tag{4.4}$$

A decision tree learning algorithm starts with a single root node containing all sample data. A number of queries is chosen as candidates for an adequate split. Query candidates are evaluated statistically: The query that partitions the sample set in a way that maximizes some notion of order in the resulting sets is chosen from the candidates.

$$\text{evaluate} : Q \times 2^X \to \mathbb{R} \tag{4.5}$$

Typical evaluation metrics are Gini impurity or Shannon information entropy [Bre96]. Assessing and choosing split candidates and distributing samples to child nodes according to the best candidate repeats until no further improvement of order can be found or a given maximum tree depth is reached.

Figure 4.1 shows a flow graph of decision tree learning. It visually depicts the steps of the learning process as outlined above.

1. In the beginning, a single sample set corresponding to the root node of the tree is available.
2. From the set chosen in the first step, candidate queries for splitting the set are generated.
3. The quality of each candidate is determined by observing how much it would contribute to the degree of order of sample sets that would result from splitting the current sample set according to the candidate.
4. If there is some gain possible for any candidate, the candidate providing the most gain is used to split the sample set. For each outcome of the sample's query, a new node is added as a child node to the node that corresponds to the split sample set.
5. If the tree has not yet grown to its maximum depth, the procedure is recursively executed for each child node.
6. Building child nodes from leaves is executed until there is no gain possible from splitting, or if the maximum tree depth is reached.

4.1.2 Decision Forests

A decision forest is an ensemble of decision trees [CSK12, Bre01]. Typically each tree is trained with a randomly chosen subset of training data. All learned trees are considered when classifying data. Let T denote the set of trees in a forest. Let $p_c^t(d)$ denote the probability that a particular tree $t \in T$ assigns class c to data d. Equation 4.6 shows the probability of a forest assigning class c to data d.

$$p_c(d) = \frac{1}{|T|} \sum_{t \in T} p_c^t(d) \tag{4.6}$$

Also, when choosing new split candidate queries when growing a tree in a decision forest only a fraction of possible queries is chosen for evaluation. This enforces independence between trees in a forest and enhances generalization capabilities [Bre01]. The fraction is a parameter of a decision forest learning algorithm. Other parameters are the number of trees and their maximum depth.

4.2 Relational Probabilistic Action Forests

A probabilistic action forest is a function that maps a given state $\in S$ to a probability distribution of successor states for particular executed actions $\in A$.

$$P(\mathcal{S}|\mathcal{S} \times \mathcal{A}) \tag{4.7}$$

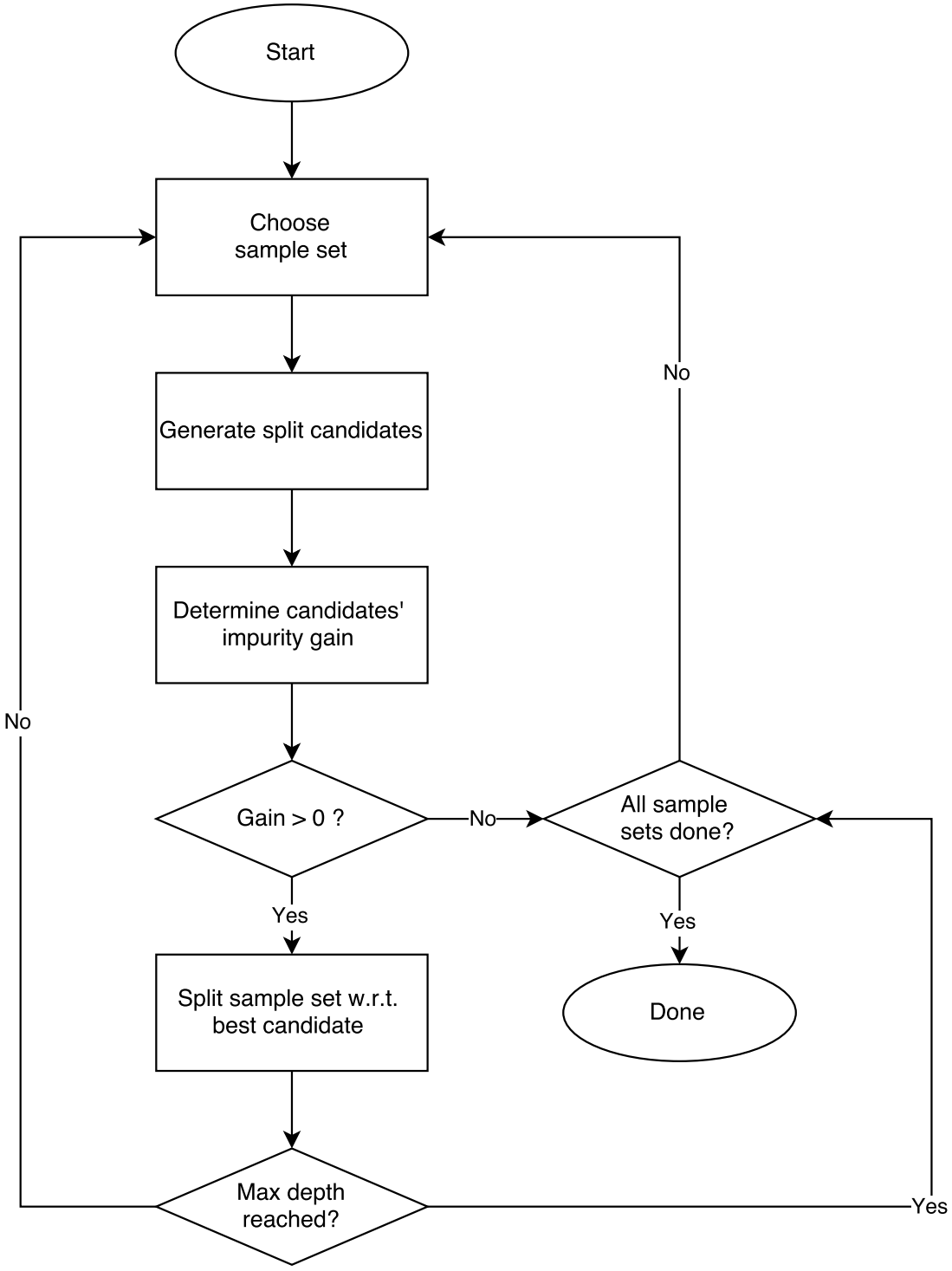


Figure 4.1: *Decision tree learning flow graph.*

Note that the learned probability distribution provides a simulation of the domain P_{sim} in the OnPlan framework (cf. Chapter 2). It can be used for generating potential future system traces to be evaluated by a simulation-based online planner.

A *relational* probabilistic action forest allows for relational representation of states and actions. These representations encode objects and their relationships. Exploiting relational structure may yield faster learning rates and improve generalization capabilities in comparison to propositional data representations [Dri10].

The general idea is to train decision forest classifiers to encode the conditional probability distribution of action effects based on observations made at runtime. Action effects may vary depending on the situation in which they are executed. Action effects may also be probabilistic: The effect of executing the same action in the same situation may vary at each execution.

Learning is performed with a set of observations. An *observation sample* is a triple of current state, executed action and observed effects. An effect is the difference of states before and after action execution. We represent effects as sets of effect atoms $\in E$ (see Section 4.2.2). We denote observations by O.

$$O \subseteq S \times A \times 2^E \tag{4.8}$$

Training a probabilistic action forest with a set of observations yields conditional probabilistic action effect distributions. Our algorithm learns a forest for each action type $\in A$. Action effects are iteratively grouped by splitting observation samples w.r.t. set-based queries and minimizing the resulting sets' Gini impurity. This procedure identifies and generalizes conditional and probabilistic action effects based on observation frequency and provides a predictive function as defined in Equation 4.7.

$$\text{learn} : 2^O \rightarrow \text{predict} \tag{4.9}$$

Section 4.2.1 introduces an illustrative example domain. Section 4.2.2 introduces effect atoms. Section 4.2.3 introduces relational queries. Section 4.2.4 discusses insertion of samples into the tree and corresponding variable substitutions. Section 4.2.5 proposes set-adjusted Gini impurity for relational grouping of observation samples and building of probability distributions for effect generalization. Section 4.2.6 describes action effect prediction.

4.2.1 Example Domain

We used the search and rescue domain introduced in Chapter 1 for evaluation of Rpaf. We shortly recap the scenario here.

A robot agent can move around a connected graph of positions and lift or drop victims. The number of carried victims is constrained by a robot's capacity. A position may be on fire, in which case a robot cannot move there. At every time step the fire attribute of a position may change depending on how many of the position's neighbors are on fire. A *safe* position never catches fire.

In particular, we list possible agent actions with their respective parameters.

- *Move*(R, P): Moves robot R to target position P if it is connected to the robot's current position and is not on fire.
- *Extinguish*(R, P): Robot R extinguishes fire at a neighbor position P.
- *Lift*(R, V): Makes robot R lift victim V at the same location.

Name	**Effect**
Attribute change	**update** $c.a \leftarrow v$ if $c.a \neq v$
Reference update	**update** $c.r \leftarrow d$
Reference addition	**add** d **to** $c.r$
Reference deletion	**delete** d **from** $c.r$

Table 4.1: *Effect atoms. c and d are objects in the domain, r is a reference and a a boolean attribute of object c, and v is a boolean value. Reference changes only occur for references that have a single object as value. Reference additions and deletions describe changes to references with arbitrary multiplicity.*

- *Drop*(R, V): Robot R drops lifted victim V at the current location.
- *Noop*: Does nothing.

4.2.2 Effect Atoms

An action effect is a set of *effect atoms* $\in E$. They are determined by building the difference of states before and after execution of an action. We consider four different effect atoms. They are listed in Table 4.1.

4.2.3 Relational Queries

To allow for relational grouping of effects depending on preconditions, we introduce relational queries in the nodes of the decision tree. They allow to exploit relational structure of the input data when searching for split criteria that reduce Gini impurity in the set of observations.

Query candidate construction is a function of a given relational data structure and a set of introduced identifiers. The relational data structure can for example be defined in terms of a class diagram (see Figure 1.9 in Section 1.3.1, Chapter 1 for a class diagram of the discrete example domain).

Queries can only use identifiers as arguments that are parameters of the learned action or have been introduced in an ancestor node. Let Δ denote relational data structures, ID identifiers and Q queries. Equation 4.10 shows the functional signature of query construction.

$$\Delta \times 2^{\mathrm{ID}} \rightarrow 2^{Q} \tag{4.10}$$

We distinguish two types of queries:

1. *Identifier queries* $\in Q_{\mathrm{ID}}$ introduce identifiers for sets of objects in observations. An identifier query is a function that takes a state and a set of identifier substitutions (see Section 4.2.4) as arguments and returns a new substitution. Let Θ denote substitution space. Equation 4.11 shows the functional signature of an identifier query.

$$S \times 2^{\Theta} \rightarrow \Theta \tag{4.11}$$

2. *Split queries* $\in Q_{\text{split}}$ separate data w.r.t. to introduced identifier sets. A split query is a function that takes queries sample substitutions for a particular identifier (see Section 4.2.4) and returns a boolean value. Equation 4.12 shows the functional signature of a split query.

$$\text{ID} \times 2^{\Theta} \to \mathit{Bool} \tag{4.12}$$

In the following, we define queries that are used in the experiments, together with their semantics. *IsEmpty* is a split query, the others are identifier queries. Let Y be a freshly introduced identifier. W and X are identifiers introduced in an ancestor of the current node or parameters of the learned action.

1. **Node:** Reference
   (a) **Type:** Identifier query
   (b) **Notation:** $Y \leftarrow X.r$
   (c) **Semantics:** Y represents the set $\{y \mid x \in X \wedge y \in x.r\}$. $x.r$ is the set of all objects that are referenced by object x through reference r. The type of Y is specified by the type of r.
2. **Node:** Filter
   (a) **Type:** Identifier query
   (b) **Notation:** $Y \leftarrow \textbf{filter}(a = v, X)$
   (c) **Semantics:** Y represents the set $\{y \mid y \in X \wedge y.a = v\}$ where $y.a$ is the value of the discrete finite-valued attribute a in object y. The type of Y is the type of X.
3. **Node:** Intersection
   (a) **Type:** Identifier query
   (b) **Notation:** $Y \leftarrow W \cap X$
   (c) **Semantics:** Y represents the set of objects present in both sets W and X, which must have the same type. The type of Y is the type of W.
4. **Node:** IsEmpty
   (a) **Type:** Split query
   (b) **Notation:** $\textbf{isEmpty}(X, \vec{\theta})$**?**
   (c) **Semantics:** Evaluates to $\exists(X \leftarrow \emptyset) \in \vec{\theta}$.

In general, relational probabilistic action forests could be extended to use other queries such as set union, type checks or cardinality constraints.

4.2.4 Sample Insertion & Substitution Management

Object set identifiers introduced by queries provide a way to generalize over observed effects. Therefore we record identifier substitutions when inserting an observation sample into the tree. We denote observation samples with their corresponding substitutions by O_{Θ}.

$$O_{\Theta} \subseteq S \times A \times 2^{E} \times 2^{\Theta} \tag{4.13}$$

When a sample reaches a node while being inserted into a tree, the node's query is evaluated w.r.t. the sample. Evaluation has different results for identifier and split queries.

1. Evaluating an identifier query adds a new substitution to a sample. Let $(s, a, \vec{e}, \vec{\theta}) \in O_\Theta$ and $q \in Q_{\text{ID}}$.

$$\text{eval} : O_\Theta \times Q_{\text{ID}} \to O_\Theta \tag{4.14}$$

$$\text{eval}((s, a, \vec{e}, \vec{\theta}), q) = \left(s, a, \vec{e}, \vec{\theta} \cup \{q(s, \vec{\theta})\}\right) \tag{4.15}$$

2. Evaluating a split query does not change the sample itself. Instead, the sample is subsequently sorted into the branch corresponding to query evaluation.

We define the evaluation operation also for sets of observation samples. Evaluating an identifier query yields a set of samples with substitutions changed according to Equation 4.15. The resulting sample set is passed to the single child node of the current node containing the identifier query.

$$\text{eval} : 2^{O_\Theta} \times Q_{\text{ID}} \to 2^{O_\Theta} \tag{4.16}$$

Evaluating a split query yields a pair of sets of samples. One set contains all samples where split query evaluation yields true, the other set contains all samples where query evaluation yields false. The resulting sample sets are passed to the corresponding child node of the current node containing the split query.

$$\text{eval} : 2^{O_\Theta} \times Q_{\text{split}} \to 2^{O_\Theta} \times 2^{O_\Theta} \tag{4.17}$$

4.2.5 Impurity Assessment for Relational Sample Sets

Relational probabilistic action forests generalize in two ways.

1. Ground objects in observed effects may be substituted with variables introduced by identifier queries, yielding more abstract effect representations.

2. Split queries provide generalization by minimizing the impurity of sample sets resulting from the corresponding split.

We measure the Gini impurity of sample sets resulting from a variable substitution or a split due to a query. This provides an assessment of the degree of generalization each query provides. The most generalizing query is chosen as new node query, and the process repeats for one or two new child nodes (depending on query type).

4.2.5.1 Effect Variables, One-of Interpretation and Sample Effect Groups

For generalization, we replace ground objects in the effect atoms of a sample with *effect variables* from the sample's substitutions. This is done if the replaced object is an element of the set associated with the variable. Let $(s, a, \vec{e}, \vec{\theta}) \in O_\Theta$ be an observation sample in a leaf, let o be a ground object and let $\vec{o}$ denote a set of objects. Equation 4.18 shows substitution of a ground object by an effect variable.

$$\begin{aligned}(s, a, \vec{e}, \vec{\theta}) \to (s, a, \vec{e}\backslash[o \leftarrow \dot{X}], \vec{\theta}) \\ \textbf{if } (X \leftarrow \vec{o} \in \vec{\theta}) \wedge (o \in \vec{o})\end{aligned} \tag{4.18}$$

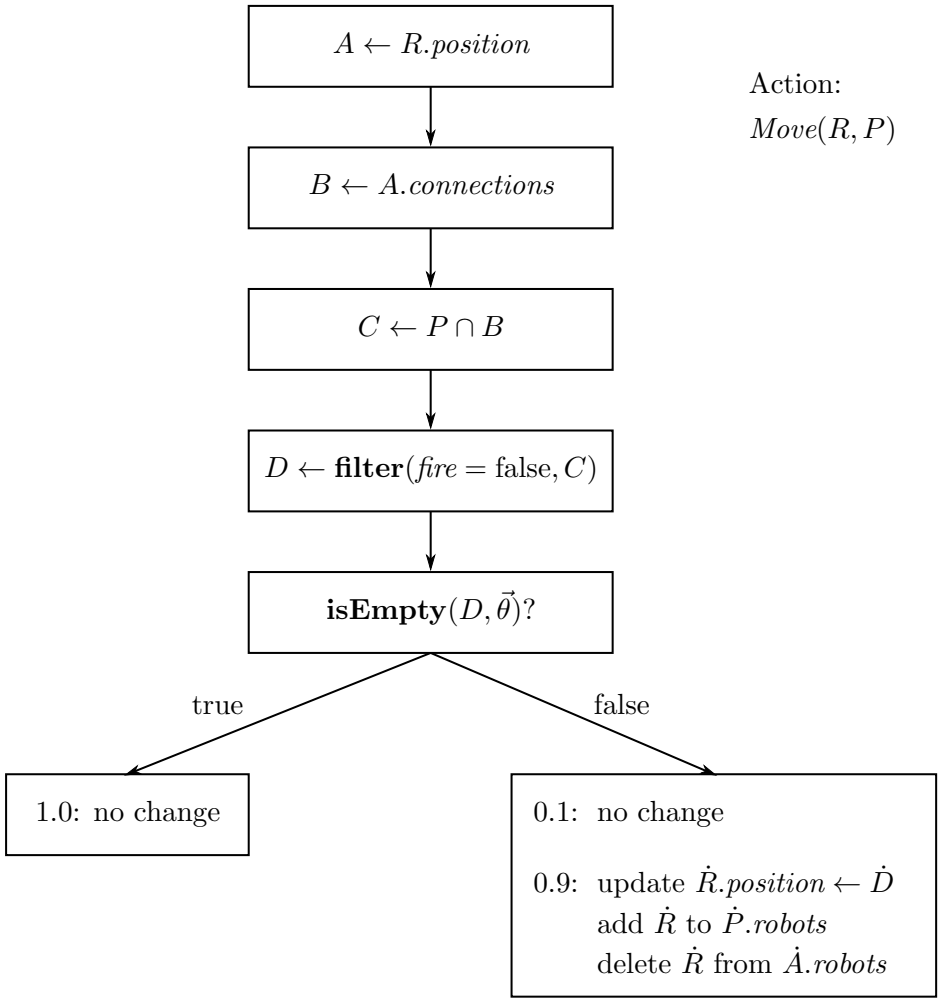


Figure 4.2: *Exemplary tree for action Move in the rescue domain (Section 4.2.1). The first four nodes introduce new identifiers. The fifth node performs a split w.r.t. a sample's substitutions $\vec{\theta}$. Queries are selected due to optimal reduction of sample Gini impurity. This effectively extracts the action's generalized probabilistic effects w.r.t. their preconditions.*

The meaning of a dotted variable in an effect is that *one of* the objects in the substituted set is subject to the effect. We denote effect variables with a dot on top (e.g. $\dot{X}$) to emphasize the *one-of* interpretation. We will consider this interpretation both when assessing Gini impurity and when predicting an action effect. The introduction of effect variables may result in different samples with syntactically equal effect. We refer to these samples as a *sample effect group*. A sample effect group is effectively an equivalence class of samples w.r.t. their effects. Samples associated with a node can always be partitioned into pairwise disjoint sample effect groups.

Figure 4.2 shows a tree where different sample effect groups have been learned: The false-child of the split node contains two sample effect groups. One implies no change due to the action, the other one implies that one of the objects in each of the object sets associated with the variables ($\dot{R}, \dot{D}, \dot{P}$ and $\dot{A}$) is subject to the effect. The first effect group (no effect) has been observed in 10 % of the training samples, the second effect group in 90 %.

4.2.5.2 Set-Adjusted Gini Impurity

We generalize over ground objects by introducing one-of variables in the effects. Given the object sets associated with these variables in the samples are not too general (i.e. are small enough), variable introduction provides an adequate degree of generalization.

Gini impurity is used to assess the degree of generalization a particular split or variable introduction provides. Conceptually, Gini impurity is defined by the probability that the effect of a randomly drawn sample from our training set is predicted incorrectly if the prediction is sampled randomly from the probability distribution associated with the sample. The introduction of object set substitutions and sample effect groups should be reflected by the computation of Gini impurity. We propose an adjustment of Gini impurity to allow for impurity assessment of relational generalization as discussed in this paper.

Let $\vec{x}$ denote the set of observed training samples. $\mathcal{G}(\vec{x})$ denotes the partition of $\vec{x}$ into sample effect groups. $\mathcal{X}(g)$ is the set of samples in the equivalence class of a sample effect group $g \in \mathcal{G}(\vec{x})$. The cardinality $|g|$ of a sample effect group $g \in \mathcal{G}(\vec{x})$ is the number of samples in the equivalence class of g. The group cardinality $|x|$ of a sample $x \in \vec{x}$ is the product of the cardinality of sets that are associated with the one-of variables in the effect of x (according to the substitutions recorded with x). Equation 4.19 defines *set-adjusted Gini impurity*.

$$I(\vec{x}) := \frac{1}{|\vec{x}|} \sum_{g \in \mathcal{G}(\vec{x})} \left[\frac{|g|}{|\vec{x}|} \left(\sum_{x \in \mathcal{X}(g)} \frac{|x| - 1}{|x|} \right) + \left(1 - \frac{|g|}{|\vec{x}|} \right) |g| \right] \tag{4.19}$$

4.2.5.3 Query Chains

Eventually, a single query may not provide enough information to allow for a sensible assessment of impurity reduction. This is a typical problem of decision trees and can be solved by performing a lookahead: We conditionally evaluate subsequent queries under hypothetical choice for one of the actual candidates [EM04, SDP06]. We realize lookahead in our approach by creating *query chain* candidates instead of using single query candidates. Query chains consist of a number of subsequent identifier queries. The last element of a chain may be a split query. The maximum length of each chain is a parameter of the algorithm.

4.2.5.4 Probabilistic Leaves

If maximum tree depth is reached or no query provides any further improvement, a leaf is generated instead of an inner node. The effect variables minimizing set-adjusted Gini impurity are determined and applied to the effects of the training examples which have been passed to the leaf. Let $\vec{x} \in 2^{O_\Theta}$ be the set of samples sorted into a leaf. We assign as probability to each sample effect group $g \in \mathcal{G}(\vec{x})$ the relative frequencies of samples in the group versus samples in the leaf. This is shown in Equation 4.20.

$$P(g) = \frac{|g|}{|\vec{x}|} \tag{4.20}$$

4.2.6 Effect Prediction

To predict an action effect distribution $\vec{e}_?$ for action a in state s, a sample $(s, a, \vec{e}_?, \emptyset) \in O_\Theta$ is sorted into the tree and corresponding variable substitutions are recorded (see

Section 4.2.4). When reaching a leaf a sample effect group is chosen probabilistically w.r.t. its probability (Equation 4.20). This effect group's *one-of* variables are substituted with the object sets resulting from sample insertion (Equation 4.15).

The probability of a particular ground effect is distributed uniformly over all possible ground instantiations. Let g be the chosen sample effect group with effect e. Let vars(e) denote the set of one-of variables occurring in e. Let $\vec{\theta}$ be the set of substitutions that were made when inserting the prediction sample in the tree. Let $\dot{e}_{\vec{\theta}}$ be e where all one-of variables have been replaced with an arbitrary object from the corresponding substitution sets in $\vec{\theta}$. Equation 4.21 states the probability of $\dot{e}_{\vec{\theta}}$.

$$P(\dot{e}_{\vec{\theta}}) = \frac{1}{\prod_{\dot{X} \in \text{vars}(e):(X \leftarrow \vec{o}) \in \theta} |\vec{o}|} \tag{4.21}$$

The predicted effect probability distribution $\vec{e}_?$ is then given by the function mapping a ground effect to its probability: $\vec{e}_? = P(\dot{e}_{\vec{\theta}})$.

4.3 Experimental Evaluation

We evaluated our approach empirically on the domain introduced in Section 4.2.1. Actions failed with an average probability of 0.05 in the experiments. The ignition probability of a position ranged from 0.05 to 0.95 depending on the number of fires at neighbor positions.

4.3.1 Evaluation of Predictive Quality

We examined the relationship between the training set size and predictive quality. Model learning was performed for 10 positions, 5 victims and 5 initial fires. We used a single robot with a maximum capacity of two. Samples were produced by instantiating actions randomly uniform w.r.t. all objects present in the domain. The instantiated action was executed and the effect observed to build a sample.

We determined the mean Brier scores for models trained with different training set sizes. The Brier score is a proper scoring rule used to evaluate the quality of probabilistic predictions [Bri50]. The idea is to compare the predictions made by a probabilistic classifier like RPAF with actually observed events. In the case of RPAF, this means that we compare for a given number of observations the prediction of action effects made by the learned RPAF with the effects that actually occurred.

Let $N \in \mathbb{N}$ be the number of samples used for determining the Brier score. Let $R \in \mathbb{N}$ be the number of potential effects in the prediction made by the RPAF forest learned for the executed action. Let $p_{n,r} \in [0;1]$ be the predicted probability of effect r for sample n. Note that prediction probabilities and number of predicted effects may differ from sample to sample, as the sample is sorted into the trees according to the state before action execution and to the particular instantiation of the executed action's parameters. Let $o_{n,r} \in \{0,1\}$ denote whether the predicted effect j was observed in sample n ($o_{n,r} = 1$) or not ($o_{n,r} = 0$). Then, the Brier score is defined as follows.

$$BS = \frac{1}{N} \sum_{n=1}^{N} \sum_{r=1}^{R} (p_{n,r} - o_{n,r})^2 \tag{4.22}$$

We determined the Brier scores by letting each model predict the outcomes of 1024 evaluation examples which were generated by the same simulation process as the

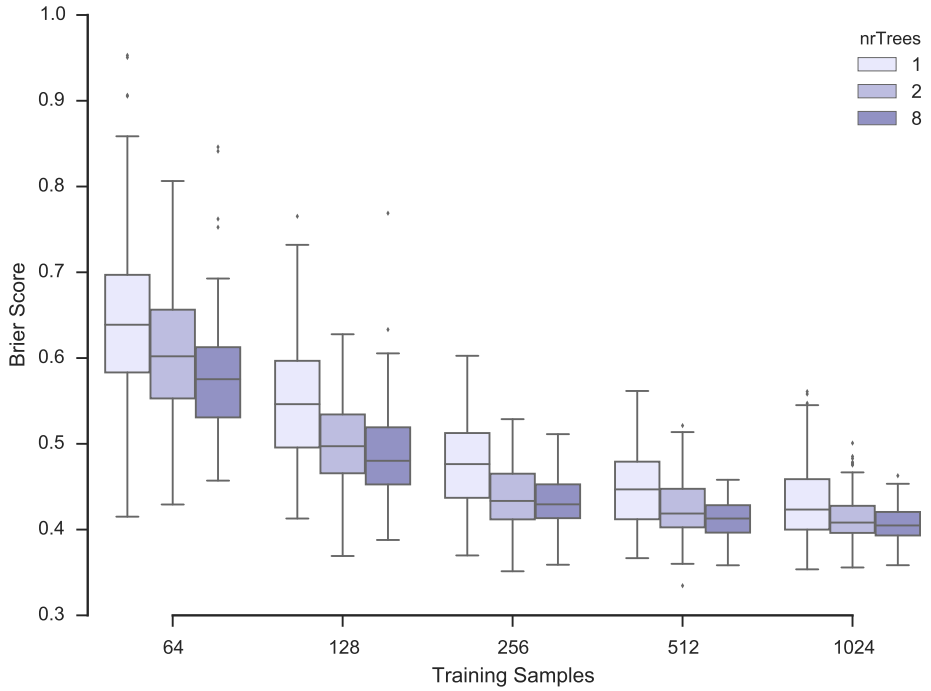


Figure 4.3: *Measured Brier scores w.r.t. training set size for various forest sizes.*

training examples. In case the observed effect was not in the set of predicted effects, we increased the score by 1.

We also evaluated the impact of the forest size on predictive performance. We set the fraction of candidates tested per node to 0.25 of all potential queries. Figure 4.3 shows Brier score w.r.t. training samples for various forest sizes (denoted by *nrTrees* in the legend). Increased numbers of trees yield faster learning rates.

4.3.2 Integration with Online Planning

We evaluated the usefulness of learned models for online planning. We provided an MCTS planner [BPW+12] with trained forests. The agent got reward for transporting victims to safe positions. The planner only had access to the learned model to explore its potential choices. As a measure of the success of the learning stage we used the reward the agent was able to accumulate. The induced models were generated from training sets of different sizes. Figure 4.4 shows the results. Gathered reward increases with number of training samples. With more than 256 samples, performance of the planner using the learned model was close to performance when using a perfect simulation of the environment. Using more trees increased expected gathered reward especially in cases where few training samples (less than 256) were available.

4.4 Related Work

Using relational queries in top-down learning of decision trees to increase generalization and predictive performance has been proposed with the TILDE algorithm [BR98]. TILDE requires as input a so-called *language bias* which defines the pool of possible

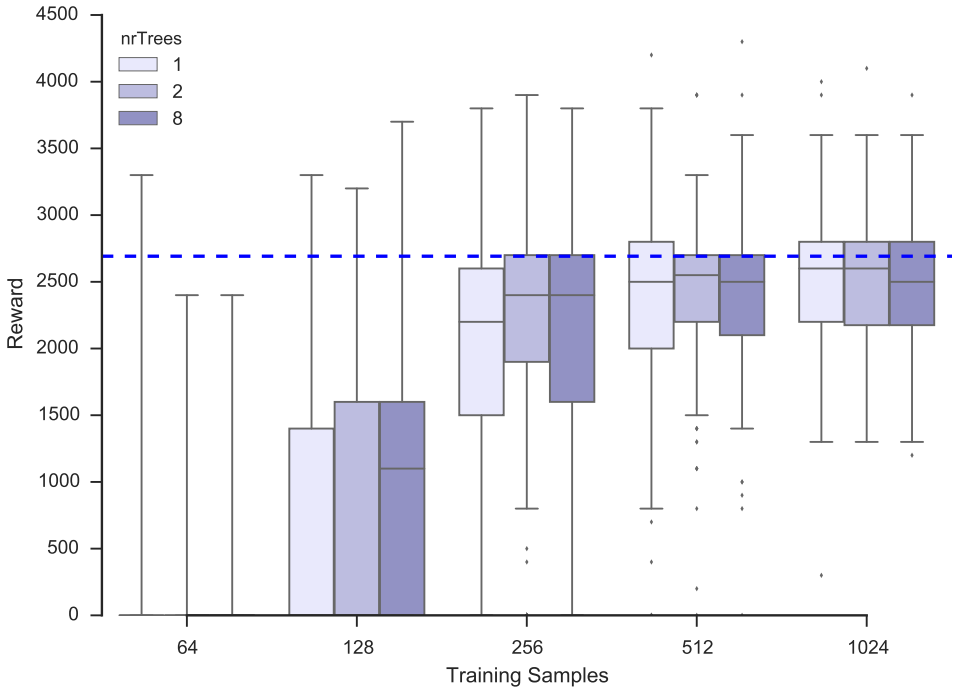


Figure 4.4: *Measured reward w.r.t. training set size for various forest sizes. The dashed line shows the median of gathered reward using a perfect simulation for planning.*

relational queries used for training data separation. The query construction mechanism described in this paper is a sort of language bias, despite not being formalized in the same way as in TILDE. TILDE has not been used in a forest setting, and was not employed for learning predictive action models.

A successor of the TILDE algorithm that was proposed in the context of relational reinforcement learning is the RRL-TG algorithm [Dri10]. In contrast to the work described in this paper, RRL-TG is an instance of Q-learning, which is *model-free*: RRL-TG does not provide an explicit encoding of conditional action effects, but rather evaluates actions in a black-box manner. The algorithm learns a relational decision tree that is used for assessment of the quality of a particular action in a given state directly. Also other algorithms of the RRL family are model-free [DR03, GDR03].

An alternative to RPAF is learning rule based action models, see e.g. work by Pasula et al. [PZK07] or *Relational Action Rules* by Rodrigues et al. (RAR) [RGRS10]. These learn declarative relational rules instead of decision forests to model domain dynamics. RAR is intended for incremental learning where new training data is streamed into the current model online. It would also be interesting to use it in combination with an online planner as in our experiments. Also, using ensembles of RAR similar to the forest approach would be an interesting use of the approach.

The combination of decision forest learning and online planning is part of the TEXPLORE algorithm [HS13, HQS12]. Here, a tree is learned per state feature, while our approach learns a forest for each action type. We argue that our approach results in smaller trees and stronger generalization. Conditional sample separation should be more coherent when only generalizing observation data of a single action. State features typically are influenced highly variable depending on which action is the cause for the

change. However, we cannot provide empirical evidence for this claim yet. TEXPLORE does not provide support for complex data types when learning domain dynamics from observations.

4.5 Summary & Outlook

We proposed *relational probabilistic action forests*, an approach for learning probabilistic predictive action models for relational data with decision forests. In contrast to existing algorithms, relational probabilistic action forests are able to cope with training samples given in form of a complex data type. This is achieved by introducing variables for object-set and queries defined over these variables. We proposed set-adjusted Gini impurity to allow for precise impurity assessment of complex relational sample data. We empirically demonstrated the effectiveness of the approach. We showed that the algorithm can be integrated with an online planner. It learns a simulation of the domain that can be used for evaluation of autonomous system decisions when planning.

An interesting venue for further research would be to investigate options for incremental learning when training data is streamed into the algorithm online. Also, while relational probabilistic action forests work well for representations specified a-priori, it would be interesting to see how the approach could be combined with relational data representations that are inferred at runtime, e.g. by statistical relational learning [Get07] or deep learning [HOT06].

Chapter 5

Time-Adaptive Cross Entropy Planning

This Chapter introduces *Time-Adaptive Cross Entropy Planning* (TACE) to increase flexibility of online planning agents in continuous state, action and time domains with infinite state-action spaces and branching factors.[1] TACE reduces simulation effort of planning with cross entropy optimization by maintaining and adapting a probability distribution over the optimal planning horizon. This allows to identify temporally local problems in a global context, and to subsequently concentrate on the solution of the local problem. We show the effectiveness of TACE by comparing it empirically to a state-of-the-art online planner for continuous domains.

We consider the problem of sequential decision making in highly complex and dynamically changing continuous domains with infinite probabilistic state spaces and branching factors. Recently, Cross Entropy Open-Loop Planning (CEOLP) has been proposed as an approach to tackle this challenge [Wei14, WL13]. First experiments have shown remarkable success on hard multi-dimensional planning problems such as robot locomotion planning. As an online planner, CEOLP interleaves planning with execution of the first action of the current plan. After execution, planning is restarted to determine the next action to execute.

CEOLP works by iteratively aggregating qualitative information about candidate plans by simulation of their consequences. Candidate plans are generated from an initial probability distribution over possible plans. Each candidate is evaluated by simulating and assessing its potential outcome. By aggregating information about multiple plans, a new probability distribution for candidate plans can be constructed by maximum likelihood estimation. This yields a distribution where potentially promising plans are more likely to be drawn. Iterating plan generation, assessment by simulation and according refinement of the generating distribution concentrates its probability mass around high value plans. CEOLP can be seen as a variant of importance sampling based on the given simulation and the evaluation function for plans.

Typically, simulation of candidate plan consequences is the most expensive part of CEOLP. *Time-Adaptive Cross Entropy Planning* (TACE) aims to reduce simulation effort based on cross entropy optimization. It adaptively optimizes plan length and individual action duration to increase planning efficiency and flexibility.

TACE enables flexible planning effort based on temporal locality. Early iterations

[1] This Chapter is based on a previous publication by the author [Belar]. The contribution is completely work of his own. Continuous Cross-Entropy Control (see Section 5.4) has not been previously published.

of cross entropy optimization sample the global solution space. By explicitly preferring shorter plans, subsequent iterations increasingly focus optimization effort on promising local partitions of the solution space. Overall, temporal adaptation identifies local problems in the global context and subsequent concentration of effort. This enables more efficient optimization when solving a close, temporally highly important local problem.

Section 5.1.1 formally introduces CEOLP. Section 5.2 discusses the key idea of time adaptation in cross entropy planning. In Section 5.3 we illustrate the effectiveness of TACE by comparing it empirically to CEOLP. We outline related work in Section 5.5. We conclude and sketch venues for further research in Section 5.6.

5.1 Preliminaries

This Section summarizes Cross Entropy Open Loop Planning [Wei14, WL13], which serves as a foundation for Time-Adaptive Cross Entropy Planning.

5.1.1 Cross Entropy Open Loop Planning

Let $\mathcal{S}$ denote the state space, and let $\mathcal{A} \subseteq \mathbb{R}^{\mathbb{N}}$ denote the action space. We denote probability distributions over actions $P(\mathcal{A}) = P(\mathbb{R}^{\mathbb{N}})$ as Φ. In general, this may be any probability distribution; we assume multivariate Gaussians in the remainder of this Chapter. A *plan* is a vector of actions $\vec{a} \in \mathcal{A}^{\mathbb{N}}$. We call a vector of distributions over actions $\vec{\phi}_{\vec{a}} \in \Phi^{\mathbb{N}}$ a *plan distribution*.

CEOLP [Wei14, WL13] is an online planner based on the cross entropy method [dBKMR05]. It is shown in Algorithm 16. CEOLP works by generating plans from an iteratively refined plan distribution. Generated plans are evaluated based on a simulation of the domain. A potential trace of consequences is generated by a simulation. Here, a simulation of domain dynamics is a probability distribution over states and actions, yielding the corresponding successor states after executing a particular action: $P(\mathcal{S}|\mathcal{S} \times \mathcal{A})$. Traces are evaluated w.r.t. an evaluation function $\text{EVAL} : \mathcal{S} \times \mathcal{A}^{\mathbb{N}} \to \mathbb{R}$. A trace's value weights the corresponding evaluated plan (Algorithm 17).

Algorithm 16 CEOLP (adapted from [WL13])

Require:
$\text{EVAL} : \mathcal{S} \times \mathcal{A}^{\mathbb{N}} \to \mathbb{R}$,
$\text{FIT} : 2^{(\mathcal{A}^{\mathbb{N}} \times \mathbb{R})} \to \Phi^{\mathbb{N}}$,
$i_{\max}, e_{\max} \in \mathbb{N}$

1: **procedure** CEOLP($s \in \mathcal{S}, \vec{\phi}_{\vec{a}} \in \Phi^{\mathbb{N}}$)
2: **for** $0...i_{\max}$ **do**
3: $E \leftarrow \emptyset$
4: **for** $0...e_{\max}$ **do**
5: $\vec{a} \sim \vec{\phi}_{\vec{a}}$
6: $v \leftarrow \text{EVAL}(s, \vec{a})$
7: $E \leftarrow E \cup (\vec{a}, v)$
8: **end for**
9: $\vec{\phi}_{\vec{a}} \leftarrow \text{FIT}(E)$
10: **end for**
11: **return** $\vec{\phi}_{\vec{a}}$
12: **end procedure**

Algorithm 17 Plan Evaluation through Simulation

Require:
$R : \mathcal{S} \to \mathbb{R}$,
$P(\mathcal{S}|\mathcal{S} \times \mathcal{A})$

```
1: procedure EVAL(s ∈ S, ā ∈ A^N)
2:     r ← 0
3:     while |ā| > 0 do
4:         s ← P(·|s, ā_0)                ▷ execute first action
5:         r ← r + R(s)                   ▷ observe reward
6:         ā ← ā_{1..|ā|}                 ▷ remove action from plan
7:     end while
8:     return r
9: end procedure
```

The initial distributions over action parameters should provide good coverage of the sample space. In our case, this would be the case when using a uniform distribution or a Guaßian distribution with large variance over possible action parameters.

When a given number $e_{\max} \in \mathbb{N}$ of individual episodes has been generated, evaluated and aggregated in the set E, the plan distribution is fitted at each step $t \in \mathbb{N}$ to the weighted plans in E by maximum likelihood estimation for multivariate Gaussian distributions (Equation 5.1).

$$\begin{aligned} \textsc{fit}(E) &= (\vec{\mu}, \vec{\Sigma}) \text{ where} \\ \mu_t &= \frac{\sum_{(\vec{a}^i, w_i) \in E} w_i \vec{a}_t^i}{\sum_{(\vec{a}^j, w_j) \in E} w_j} \\ \Sigma_t &= \frac{\sum_{(\vec{a}^i, w_i) \in E} w_i (\vec{a}_t^i - \mu_t)^T (\vec{a}_t^i - \mu_t)}{\sum_{(\vec{a}^j, w_j) \in E} w_j} \end{aligned} \tag{5.1}$$

The new distribution is used for plan generation in the next iteration, likely to be generating more high value plans than the previous distribution. After $i_{\max} \in \mathbb{N}$ iterations, the resulting plan distribution is returned. The agent then executes an action drawn from the first element of this distribution and the planning process is restarted.

CEOLP does not require any discretization of state or action space. Planners directly optimizing continuous representations show superior performance compared to open-loop planners that require a-priori discretization of continuous state and action spaces [MWL11, WL12].

5.2 Time-Adaptive Cross Entropy Planning

We propose Time-Adaptive Cross Entropy Planning (TACE) that renders CEOLP time-adaptive in two ways:

1. By optimizing *planning horizon* (i.e. search depth) to reduce simulation effort per planned action and to orient the agent in the search space when no local problem is present (Section 5.2.1).

2. By optimizing *duration* of each action to adapt the number of required planning steps (Section 5.2.2).

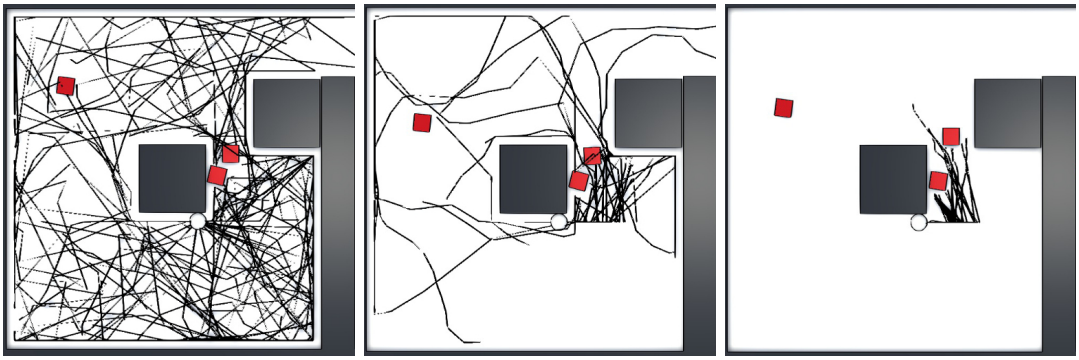


Figure 5.1: *Illustration of iterative variance reduction and depth adaptation of plan distributions with* TACE *planning. Black lines show simulation traces. Iterations one, five and ten are shown from left to right. In the course of planning, the agent (white circle) identified a local problem (red boxes to be collected) and concentrated its evaluation effort accordingly. Note the Gaussian movement of the target boxes.*

5.2.1 Adaptive Planning Horizon

In highly dynamic domains, it may be valuable to concentrate planning effort on current local problems. To this end, time adaptation is useful to identify local problems in a global context (when planning with large horizons), and to solve local problems effectively (when planning with small horizons). Figure 5.1 illustrates this idea: Early iterations sample from a distribution with a large planning horizon to detect and estimate temporally local problems. Subsequent iterations decrease the planning horizon accordingly. Of course, this argument can be turned around: If an agent does not find any local problem, it can adaptively increase its planning horizon to orient itself in the search space.

TACE is shown in Algorithm 18. Adaptation of the planning horizon is realized by maintaining an additional probability distribution $\phi_h \in P(\mathbb{N})$ over potentially promising search depths. Instead of fixing the planning horizon as a parameter of the algorithm, TACE renders the length of plan vectors $\vec{a} \in \mathcal{A}^{\mathbb{N}}$ flexible: The length of each plan $l \in \mathbb{N}$ is drawn from the distribution of horizons ϕ_h (line 5). Individual plans $\vec{a}$ are then drawn from $\vec{\phi}_{\vec{a}}$ as in CEOLP (line 6) with the corresponding length $|\vec{a}| = l$, thus $|\vec{a}| \sim \phi_h$.

The distribution over planning horizons ϕ_h is fitted by maximum likelihood estimation (Equation 5.1) in the same way as the plan distribution (line 9). We used a univariate normal distribution for experimental purposes, but other probability distributions can be used as well, provided an adequate fitting procedure is defined. TACE uses smooth updating of distributions (lines 10 and 11) [dBKMR05]. The parameters α_h and $\alpha_{\vec{a}}$ (both $\in [0;1]$) allow to control the impact of new information gathered from the current fitting operations on the already existing distributions of planning depth and action parameters.

As simulation is the most expensive operation of the algorithm, TACE is tuned to reduce the number of planning steps to reduce simulation effort. TACE prefers shorter plans by adapting their weight w.r.t. to their length when evaluating a trace (line 7).

$$\text{EVAL'}(s, \vec{a}) = \text{EVAL}(s, \vec{a}) + \frac{c}{|\vec{a}|} \tag{5.2}$$

Here, $c \in \mathbb{R}$ is a constant parameter to weight the impact of trace length w.r.t. potential trace value.

Algorithm 18 Time-Adaptive Cross Entropy Planning

Require:
EVAL' : $\mathcal{S} \times \mathcal{A}^{\mathbb{N}} \to \mathbb{R}$,
FIT : $2^{(\mathcal{A}^{\mathbb{N}} \times \mathbb{R})} \to \Phi^{\mathbb{N}}$,
$i_{\max}, e_{\max} \in \mathbb{N}$

procedure TACE($s \in \mathcal{S}, \vec{\phi}_{\vec{a}} \in \Phi^{\mathbb{N}}, \phi_h \in P(\mathbb{N})$)
  for $0...i_{\max}$ **do**
    $E \leftarrow \emptyset$
    for $0...e_{\max}$ **do**
      $h \sim \phi_h$
      $\vec{a} \sim (\vec{\phi}_{\vec{a}}, h)$
      $v \leftarrow \text{EVAL'}(s, \vec{a})$
      $E \leftarrow E \cup (\vec{a}, v)$
    end for
    $\phi_h \leftarrow (1 - \alpha_h)\phi_h \cdot \alpha_h \cdot \text{FIT}_h(E)$
    $\vec{\phi}_{\vec{a}} \leftarrow (1 - \alpha_{\vec{a}})\phi_{\vec{a}} \cdot \alpha_{\vec{a}} \cdot \text{FIT}_{\vec{a}}(E)$
  end for
  return $\vec{\phi}_{\vec{a}}$
end procedure

5.2.2 Adaptive Action Duration

As planning an action is expensive, we want to reduce the number of planned actions without sacrificing an agent's flexibility to perform less coarse action coordination. Figure 5.2 shows a setting where an agent can choose movement and rotation speed. Its designated goal is to reach the static target in the top right corner (red box). Theoretically, two action commitments would suffice for reaching the target box: Move to a turning point (either upper left or lower right corner), and then rotate towards the target and approach it. As one can see, setting a fixed action duration may not be optimal for a given task (which may be unknown at system design time). While Figure 5.2 shows an example where fixed action duration of the CEOLP agent (top row) was too low to be optimal, other situations may also require to adaptively reduce action durations to increase planning efficiency.

Thus, for additional optimization of the number of planned actions and to increase the planning agent's flexibility w.r.t. the current task, TACE adapts each action's duration (which in turn influences the re-planning rate) by cross entropy optimization: We remove action duration as a parameter of the algorithm (as in CEOLP) and render it an adaptive parameter of the actions themselves, which are optimized by TACE. $\mathcal{A}_d$ denotes the action parameter space extended by action duration.

$$\mathcal{A}_d \subseteq \mathcal{A} \times \mathbb{R} \tag{5.3}$$

5.3 Experimental Evaluation

We evaluated the effectiveness of time adaptive planning as proposed in Section 5.2 empirically by comparing CEOLP and TACE agents for various configurations. We describe our experimental setup in Section 5.3.1. We discuss our observations in Section 5.3.2.

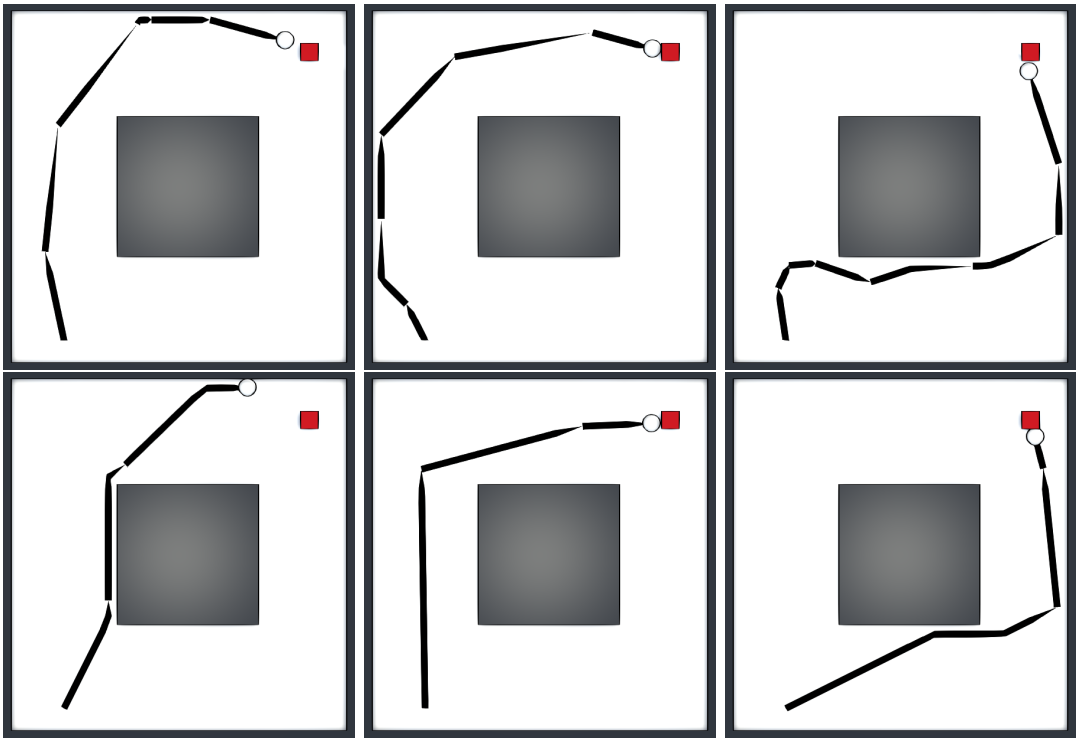


Figure 5.2: *Motivation for adaptive action duration: As planning an action is expensive, we want to reduce the number of planned actions without sacrificing flexibility. The white circle represents the agent, the small square in the upper right corner of each image is the target to be reached by the agent. The dark area in the center is a static obstacle. Black lines show agent traces, each line segment shows a planned and executed action. The upper row shows traces of a* CEOLP *agent with a fixed action duration. The bottom row shows traces of a* TACE *agent that adaptively optimizes action duration.*

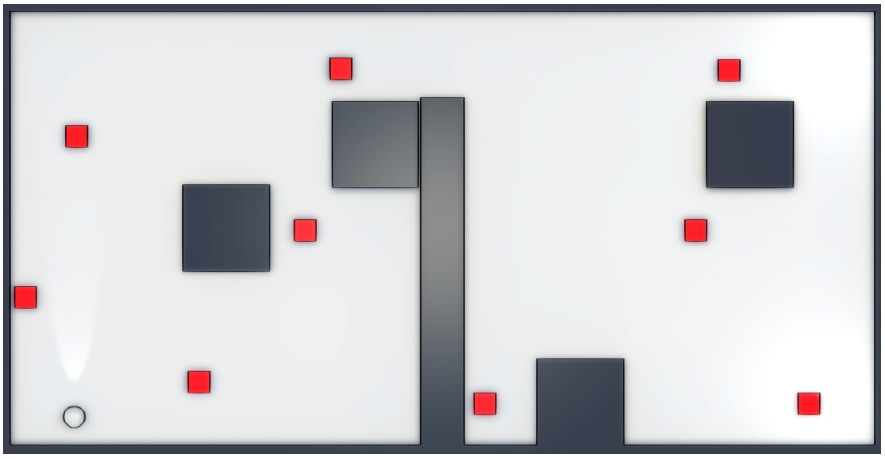


Figure 5.3: *Experimental setup. Grey areas are static obstacles. Small boxes are targets with Gaussian motion. The agent is the circle bottom left.*

5.3.1 Setup

We used the continuous example domain introduced in Chapter 1 for evaluation. The initial setup we used in our experiments is shown in Figure 5.3. We shortly recall the domain here.

An agent (shown as small white circle) is operating in a continuously modeled environment consisting of static obstacles (shown as dark rectangular areas) and dynamically moving targets (shown as small red boxes). The area has a size of $10 \times 20m$. The agent's task is to collect the targets as fast as possible, which is realized when the agent collides with a target. Target movement was determined from a Gaussian distribution. Velocity, rotation rate and duration to re-sampling of these parameters were drawn independently from a normal Gaussian distribution (i.e. mean and standard deviation of one; in m/s, deg/s or s, respectively).

Agent action parameters were velocity $v \in [0, 3]$ m/sec and rotation $r \in [-90, 90]$ deg/sec. Values sampled by TACE were correspondingly clamped. As described in Section 5.2, the TACE agent optimized two additional parameters: (a) The duration in seconds $d \in \mathbb{R}$ for each action and (b) the planning horizon $h \in \mathbb{N}$. Agents executed rotation and forward movement in sequence. Both rotation and movement were executed for the predefined (CEOLP) or determined (TACE) duration.

Agents were provided with a simulation of their environment. We implemented a black-box simulation that sampled the real execution model at given time steps and performed simple geometric inference to determine targets that had been collected by agents for a given simulation step. Simulation and real execution model exposed difference w.r.t. accuracy of collision detection and position computation – the simulation did not perfectly predict the real situation, which is a common situation in real-world tasks.

For trace evaluation we used a time-discounted reward aggregation model. Targets were collected by agents on collision. Each collected target provided a reward of γ^t, were $\gamma \in [0, 1]$ is a discount factor and $t \in \mathbb{R}$ is the time to collection. This model lets agents with lower γ prefer traces where targets are collected early.

We evaluated CEOLP with a planning horizon $h = 50$. Lower planning horizons regularly lead to situations were the agent did not find a far target due to physical distance, resulting in low performance. Correspondingly, we choose the optimization

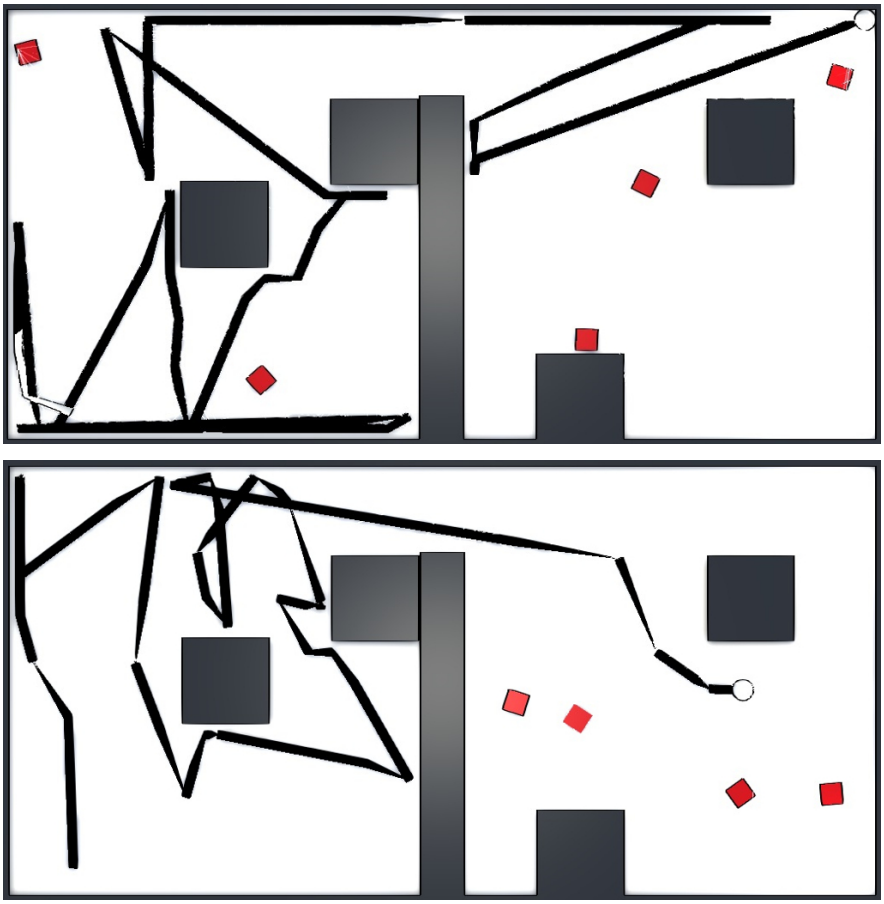


Figure 5.4: *The black lines show exemplary traces of* CEOLP *(top,* $h = 50, d = 4$*) and* TACE *(bottom,* $h \in [2, 50], d \in [0.5, 4]$*) agents after 20 action executions. Note the straight movement of the* TACE *agent when traversing from one room to another, while preferring shorter, more flexible movement when facing a local problem.*

range of the horizon for TACE planning to be $h \in [2, 50]$. We evaluated CEOLP for various action durations ($d = 1, 2, 4$). We choose the ranges for action duration accordingly for the TACE planner ($d \in [0.5, 1], [0.5, 2], [0.5, 4]$). Both planners refined their action parameter distributions 10 iterations á 30 plan samples per planned and executed action ($i_{\max} = 10, e_{\max} = 30$). The initial distribution of action parameters was uniformly random within the given parameter ranges.

5.3.2 Results

We could observe the additional flexibility of TACE when compared to CEOLP in the experiments. Figure 5.4 shows exemplary traces of a CEOLP configuration ineffective due to large action duration, and a flexible TACE planner. We could observe the changing focus of TACE between local, very reactive planning, and global, rather proactive movement towards the next local subtask.

5.3.2.1 Expected Reward Gain

We measured the gain of expected reward w.r.t. episode samples drawn from the initial distribution. Each generation of the plan distribution is build by maximum

likelihood estimation from the samples drawn from the previous generation as defined by Equation 5.1. The gain of expected reward is a measure of how effectively the iteratively reshaped plan distributions guide plan search towards high-value regions of the state-action space.

Let $\bar{r}_i$ denote the average reward gathered from episodes drawn from the Gaussian plan distribution at generation i. We define the *gain of expected reward in generation i* as quotient of average reward of the current generation i and average reward gathered by the initial distribution where actions have been drawn from a uniform random distribution.

$$\textit{Expected reward gain in generation } i = \frac{\bar{r}_i}{\bar{r}_0} \tag{5.4}$$

We compared the gain of expected reward per generation both for TACE and CEOLP. Figure 5.5 shows the median of gain of expected reward of a distribution per generation, together with a 0.95 confidence interval. We observed a slight degradation of TACE solutions generated from the first two refinements of the plan distribution. This is likely due an initial orientation in the larger search space. The search space is larger due to the action duration parameter that is additional optimized by TACE. However, onwards from generation four, the solutions provided by time-adaptive cross entropy planning provide a larger gain than the CEOLP equivalents. The solution quality achieved by TACE reflects (a) the additional flexibility of planning due to adaptively optimized action durations and (b) that concentration of optimization efforts to locally relevant situations due to adaptive simulation depth yields more reward than it loses by dropping information from the planning process that is currently not relevant.

As one of the key ideas of TACE is to speed up the planning process by reduction of simulation steps, we additionally compared the gain of expected reward per time, rather than per generation. As each generation is evaluated faster by TACE than by CEOLP due to adaptive optimization of simulation depth, the gain of expected reward is growing much faster for time-adaptive cross entropy planning. Figure 5.6 shows the same data as Figure 5.5, but sorted into discrete time bins. Obviously, the reduction of simulation depth does not only positively affect the gain of expected reward per generation, but also has a positive effect on the time that is needed to achieve this gain.

5.3.2.2 Estimation Precision

We measured the gain of estimation precision per generation. As a measure for estimation precision, we used the coefficient of variation (CV), which provides a degree of precision that is independent from the sample mean. The CV is defined as the quotient of standard deviation and the sample average, which is the average reward in our case. Let $\bar{r}_i$ denote the average accumulated reward from episodes drawn from generation i of the plan distribution. The coefficient of variation of generation i, CV_i is the defined as follows.

$$CV_i = \frac{s_{\bar{r}}}{\bar{r}} \tag{5.5}$$

We are interested in the gain of this measure per generation. We define the *gain of coefficient of variation in generation i* as quotient of coefficient of variation of the current generation i and the coefficient of variation gathered by the initial distribution

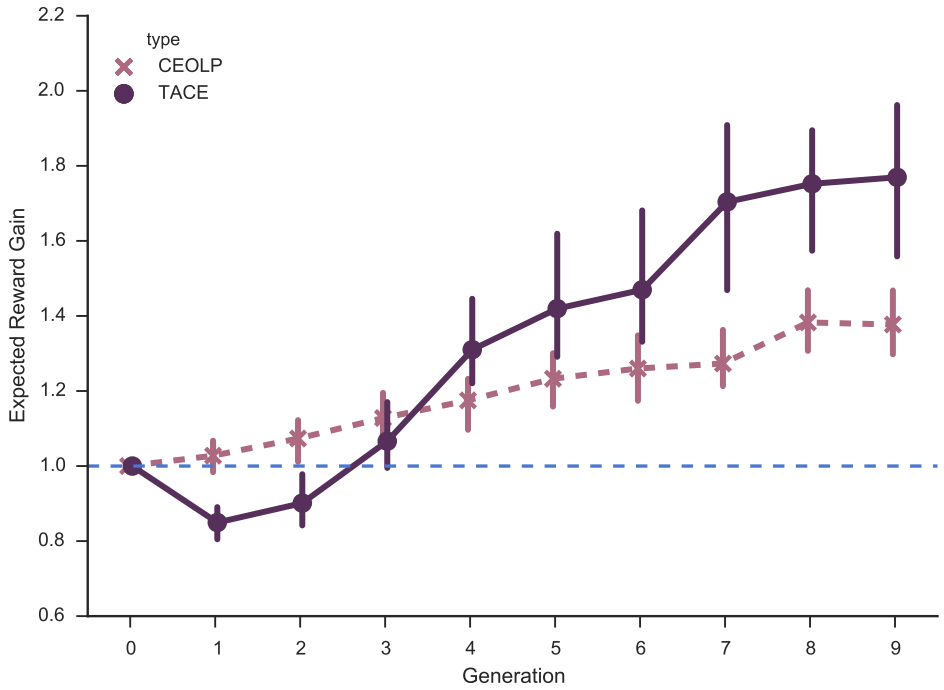


Figure 5.5: *Comparison of* TACE *and* CEOLP *w.r.t. gain of estimated future reward per generation. Data points show the median, vertical bars represent 0.95 confidence intervals.*

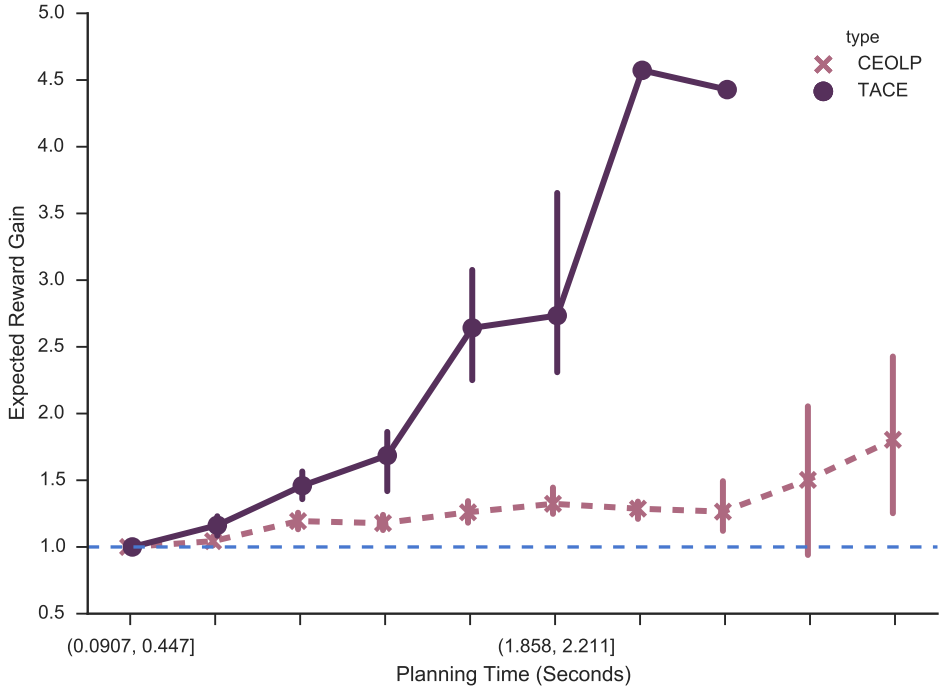


Figure 5.6: *Comparison of* TACE *and* CEOLP *w.r.t. gain of estimated future reward per planning time. Data points show the median, vertical bars represent 0.95 confidence intervals.*

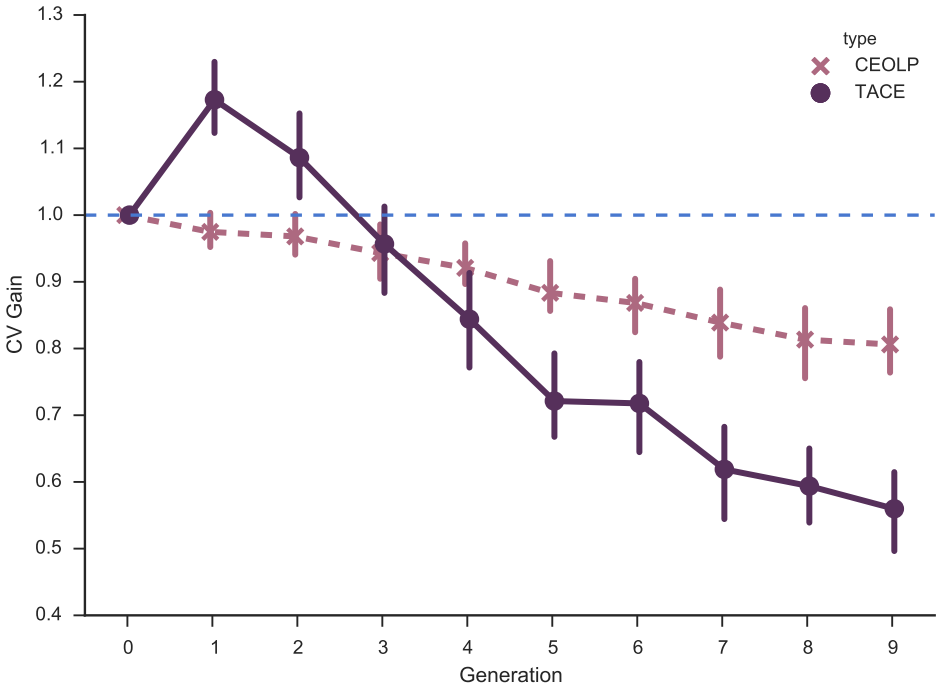


Figure 5.7: *Comparison of* TACE *and* CEOLP *w.r.t. gain of coefficient of variation per generation. Data points show the median, vertical bars represent 0.95 confidence intervals.*

where actions have been drawn from a uniform random distribution.

$$CV \text{ gain in generation } i = \frac{CV_i}{CV_0} \tag{5.6}$$

Figure 5.7 shows our observations with regard to gain of estimation precision. The median of the data is shown together with 0.95 confidence intervals. As for the expected reward, the first two generations show a declining value of precision in comparison to the initial uninformed samples. However, as for expected reward estimation, after four generations the gain of CV is much more effective for TACE in comparison to CEOLP.

As mentioned above, we are not only interested in the gain of precision per generation. In a real time application domain, it is also important how effectively the gain of estimation precision is generated w.r.t. planning time. Figure 5.8 shows the corresponding results. The median of the data is shown together with 0.95 confidence intervals. The results are generated from the same data as in Figure 5.7, but here the CV gain is shown w.r.t. ten equally spaced planning time bins. One can see that TACE is much more effectively using time to increase the estimation precision in terms of CV than CEOLP.

5.3.2.3 Planning Time

A central motivation for adaptation of simulation steps is that simulation typically is the most expensive operation within the planning process. We measured the impact of time adaptation by comparing planning times per action for CEOLP with TACE. Both algorithms use the same procedure for optimization of the sampling distribution of action parameters, i.e. cross entropy minimization of a multivariate Gaußian distribution.

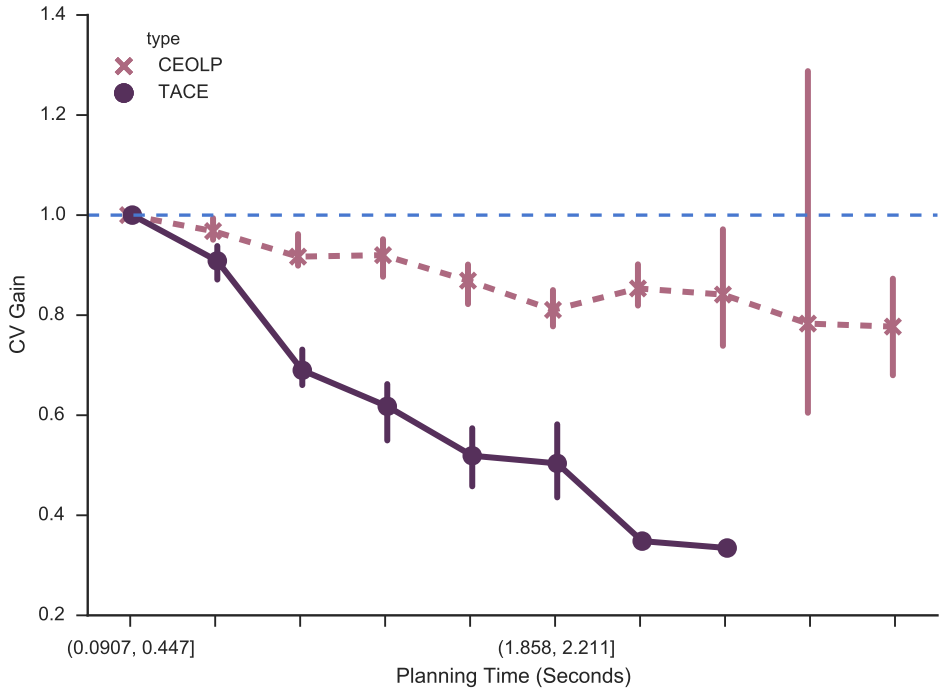


Figure 5.8: *Comparison of* TACE *and* CEOLP *w.r.t. gain of coefficient of variation per planning time. Data points show the median, vertical bars represent 0.95 confidence intervals.*

However, TACE additionally optimizes sampling depth and action duration, effectively reducing simulation effort.

Figure 5.9 shows histograms of the measured times needed by CEOLP and TACE to complete planning of an action for a given number of episodes ($e_{\max} = 30$) and refinement iterations ($i_{\max} = 10$). As can be seen, the reduced simulation cost of TACE yields planning times below those of CEOLP. The lower density of the TACE distribution reflects its flexibility w.r.t. adapting its planning depth to the current situation.

5.3.2.4 Performance

In terms of performance, TACE clearly outperformed CEOLP. Figure 5.10 shows the time needed to collect a particular number of targets. Figure 5.10a shows the strongest and weakest configuration of CEOLP compared to the most flexible configuration of TACE. In our experiments, increasing flexibility (in terms of action duration range) showed a positive effect on performance, even though the optimization problem gets harder (Figure 5.10b). All variants of TACE outperform any CEOLP configuration.

5.3.2.5 Planned Actions

We also observed that TACE is able to complete its task with less actions, and thus less re-planning steps (Figure 5.11a). This is important as it indicates superiority of TACE not only w.r.t. the metric we required the agent to optimize (time to collection), but also w.r.t. structure and flexibility of the solutions that were found. In terms of required actions, increasing flexibility in form of action duration range showed the same

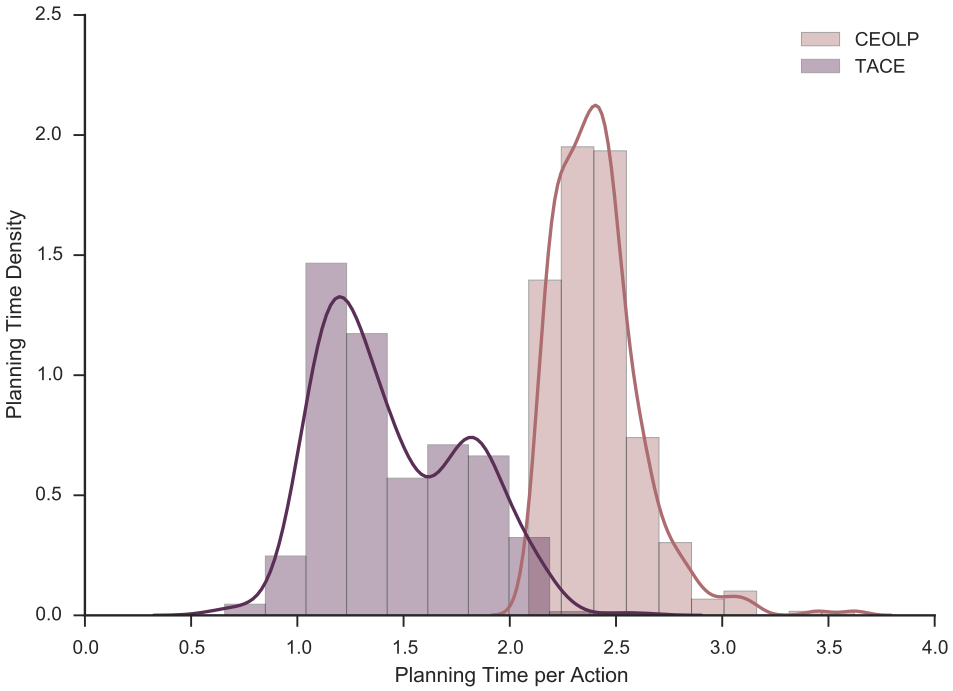


Figure 5.9: *Comparison of* TACE *and* CEOLP *w.r.t. planning time per action.*

tendency as for time to collection: Increased flexibility yielded less re-planning steps (Figure 5.11b).

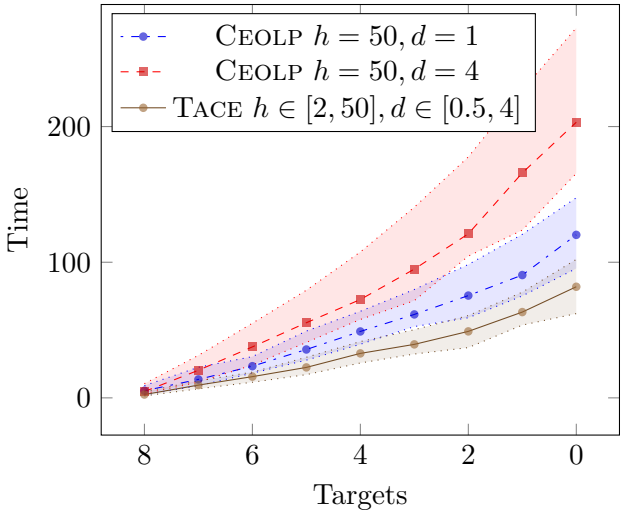
5.3.2.6 Simulation Steps

TACE performed only around half as many simulation steps as CEOLP (Figure 5.12a). While this is not surprising due to optimization of planning horizon, it is an important finding: Typically, simulation is the most expensive part of cross-entropy planning [WL13]. In comparison to typical simulation computations, the actual optimization via maximum likelihood estimation can be neglected in terms of both computation time and memory requirements. Thus, reduction of simulation steps is an important property of TACE. We also observed that all tested configurations of TACE required less simulation steps than any CEOLP configuration (Figure 5.12b). These findings indicate that TACE is better able to draw relevant information from simulations than CEOLP by incorporating temporal structure of the task at hand into the planning task.

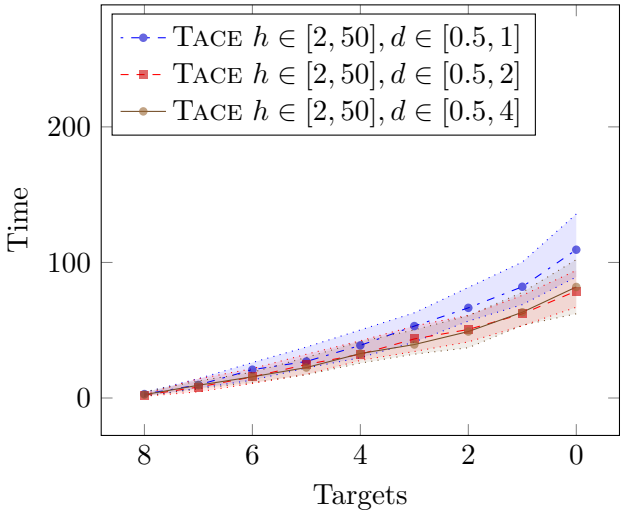
5.4 Continuous Time Cross Entropy Control

In its original formulation, CEOLP interleaves planning and action execution. A promising action to be executed in the current situation is evaluated by running simulations and refining the simulation strategy. In the course of planning, the agent does not execute any action. On the other hand, while an action that has been recommended by the planner's strategy is executed, the planning process is interrupted.

In continuous time domains, not doing anything is the same as executing a no-operation. Also, the time used for execution could effectively be used for planning in

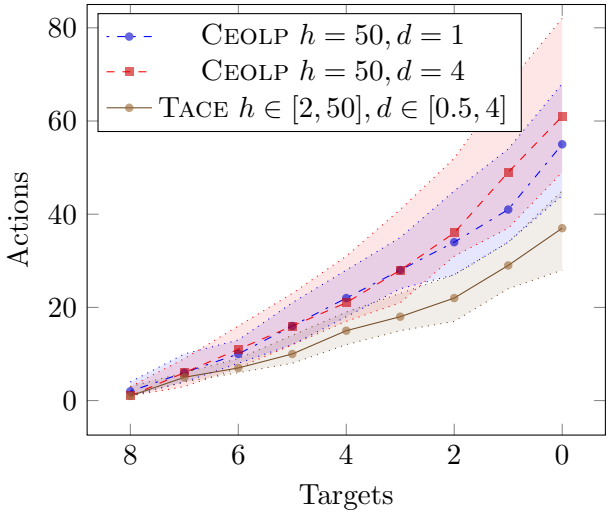


(a)

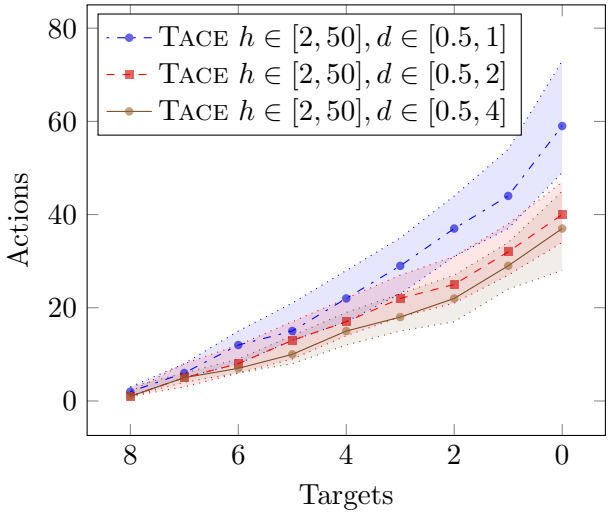


(b)

Figure 5.10: *Performance of* TACE *(a) versus weakest and strongest configurations of* CEOLP *and (b) for various adaptation ranges. Marks show median of observed data from 100 runs, bands show interquartile range.*

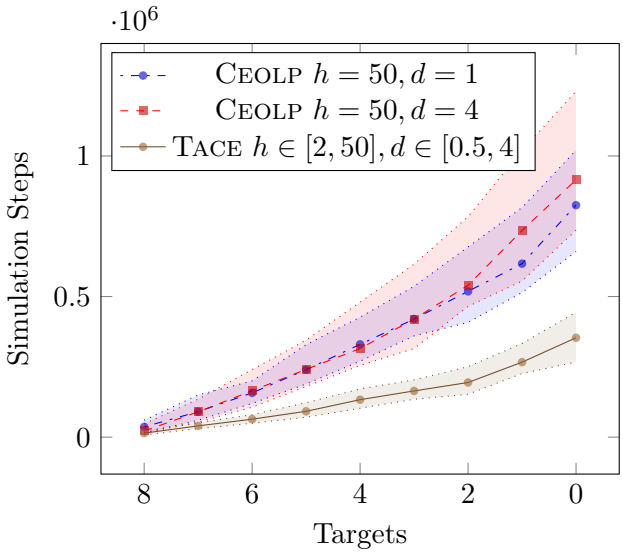


(a)

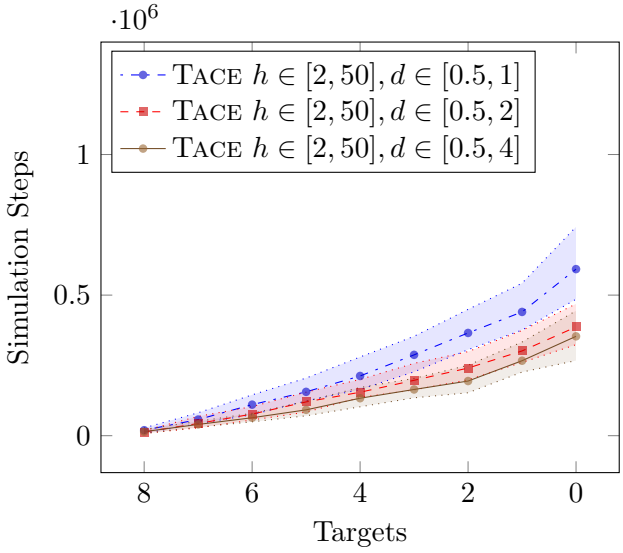


(b)

Figure 5.11: *Actions planned and executed by* TACE *(a) versus weakest and strongest configurations of* CEOLP *and (b) for various adaptation ranges. Marks show median of observed data from 100 runs, bands show interquartile range.*



(a)



(b)

Figure 5.12: *Simulation steps performed by* TACE *(a) versus weakest and strongest configurations of* CEOLP *and (b) for various adaptation ranges. Marks show median of observed data from 100 runs, bands show interquartile range.*

parallel. The question that arises is:

> *How to update the current strategy and execute actions taking into account the continuous flow of time?*

In this Section, we introduce the C^3 algorithm (Continuous Time Cross Entropy Control) to answer this question. The key idea is to exploit the explicit modeling and optimization of action durations as maintained by TACE. We assume that an autonomous system is perceiving (i.e. observing) its environment at a certain – potentially dynamically changing – rate. In other words, the system observes its current state every δt seconds, where δt may vary from observation to observation. We say that observing is performed in *frames*. A frame then takes δt seconds of *frame time*. In a typical continuous time system, δt tends to be small (i.e. in the millisecond range). Nevertheless, the ideas that C^3 are based on also apply for larger frame times.

If a system can measure its frame time, we can use this information to update the estimated optimal duration of the first planned action by reducing the corresponding distribution mean by δt_i for each frame i. If the mean of the duration of the first planned action falls below zero, we drop the action from the strategy.

A system that uses C^3 performs this frame based update of action duration of the first planned action, and at the same time executes an action w.r.t. the distribution of the other parameters besides duration (e.g. speed and rotation rate). It also performs simulations based on the current plan distribution to optimize action parameters w.r.t. the current situation.

For example, assume that the optimal duration estimate of the first action yields a mean of 1 second. A system based on C^3 then executes the corresponding action (i.e. using the other estimated optimal parameters) for a single frame – that is, until it is able to observe its environment the next time. Assume this frame took 0.1 seconds. Then, the mean of the optimal duration estimate of the first action is reduced accordingly from 1 second to 0.9 seconds. Optimization in the spirit of TACE is then done by sampling and simulating actions w.r.t. the new duration mean that has been reduced by the current frame time.

Figure 5.13 shows the distributions of accumulated action durations of the first two actions of a strategy. Only the duration dimension of the multivariate distribution of action parameters is shown for the sake of clearer visualization. Note that the mean of the second distribution that is shown in Figure 5.13 is *not* the actual mean μ_1 of duration of the second action that is maintained in the strategy. It is rather the sum of the means of the first and second action, representing their *accumulated* duration (i.e. $\mu_0 + \mu_1$). The other parameter dimensions besides duration are treated exactly the same as for CEOLP and TACE when optimizing.

Figure 5.14 shows the update of the first action duration mean as described above after a frame that took δt frame time. Note that only the mean of the first action in the strategy has to be reduced. This reflects the fact that this action has been executed in the current frame for δt time. The new mean of the first action is the difference of current mean μ_0 and frame time δt.

$$\mu_0 \leftarrow \mu_0 - \delta t \tag{5.7}$$

Figure 5.15 illustrates the situation where the mean of the first action duration falls below zero. This means that based on previous optimization, the current first action should be replaced by the subsequent one in the strategy. Therefore, when the mean of

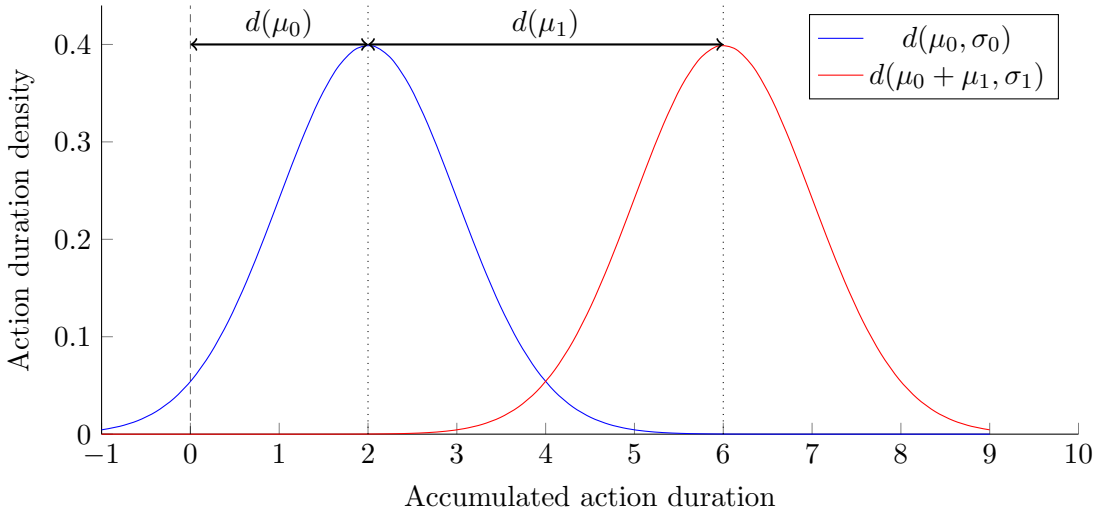


Figure 5.13: *Sequence of distributions over plan durations.*

the first action duration drops below zero, the strategy is changed correspondingly by dropping the multivariate distribution of the first action. The process of frame based updating is the repeated, now for the previously second action that has become the new first one.

5.4.1 The C^3 Algorithm

Algorithm 19 shows the formalization of C^3. It extends TACE (see Algorithm 18) by incorporating frame time to duration of the first action (lines 2 – 5). The current frame time is now a parameter of the algorithm, which is repeatedly called by the planning agent. The mean of the duration distribution of the first (i.e. currently executed) action is updated as follows.

1. The current frame time is subtracted from the mean duration of the current estimate of the first action in the strategy (line 2).
2. Check whether the corresponding mean drops below zero (line 3). If so, proceed with the following steps.
   (a) Drop the first (current) action from the strategy (line 4).
   (b) Append a new action to the strategy (i.e. a prior distribution) (line 5).

After updating the duration of the current action estimate, cross entropy minimization is performed on the strategy as in the TACE algorithm. Each call to C^3 generates and evaluates one generation of samples (lines 7 – 12), and updates the current strategy and search depth distribution correspondingly (lines 13 – 14).

5.4.2 Empirical Results

We evaluated C^3 empirically, and compared the results with the ones obtained for TACE (see Section 5.3). We used the same experimental domain and reward function as for TACE. We set $i_{\max}$ to 10, meaning that ten samples were generated per frame. Experiments were performed on a standard laptop, yielding frame times between 25 and 50 milliseconds. We used a learning rate of $\alpha = 0.6$ to enable smooth updating of the strategy. Target movement speed was set to 1 m/sec.

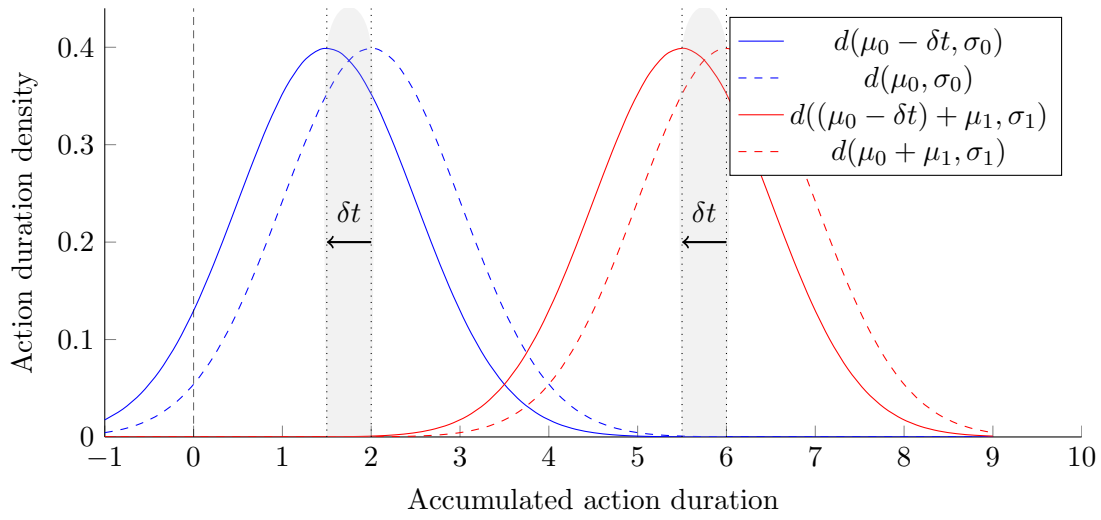


Figure 5.14: *Adjusting the initial distribution of action duration after a frame that took δt time. Dashed lines denote the distributions of action duration before the update, solid lines show the corresponding distributions after the update.*

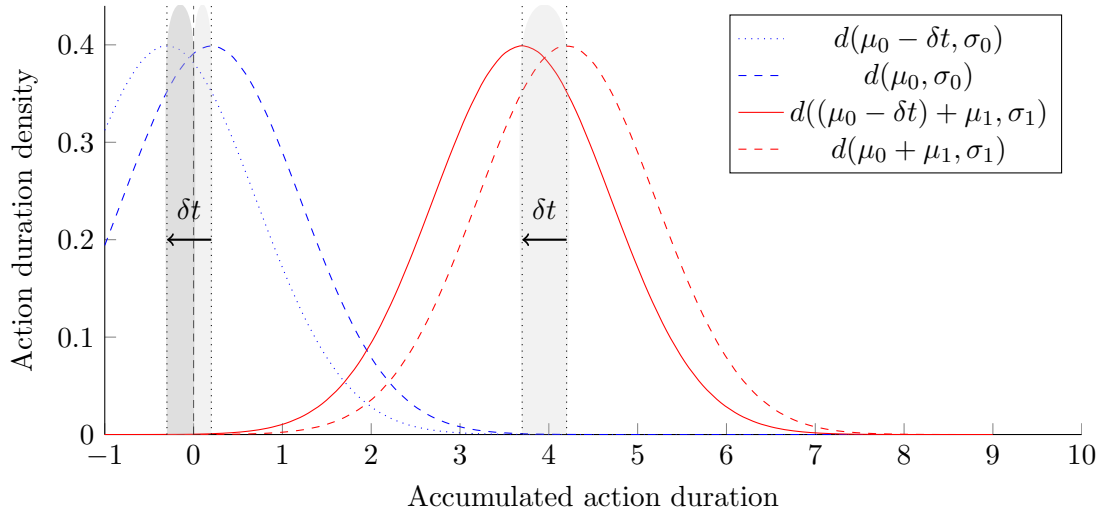


Figure 5.15: *Dropping the initial distribution of action duration from the sequence when its mean drops below zero. Dashed lines denote the distributions of action duration before the update, solid lines show the corresponding distributions after the update. The dotted line shows the dropped distribution.*

Algorithm 19 Continuous Time Cross Entropy Control

procedure $\mathrm{C}^3(s \in \mathcal{S}, \phi_{\vec{a}} \in \Phi^*, \phi_h \in P(\mathbb{N}), \delta t \in \mathbb{R})$
  $\mu_d(\mathrm{head}(\phi_{\vec{a}})) \leftarrow \mu_d(\mathrm{head}(\phi_{\vec{a}})) - \delta t$ ▷ Update duration mean of first action
  if $\mu_d(\mathrm{head}(\phi_{\vec{a}})) \leq 0$ **then**
    $\phi_{\vec{a}} \leftarrow \mathrm{tail}(\phi_{\vec{a}})$ ▷ Drop first action from strategy
    $\phi_{\vec{a}} \leftarrow \phi_{\vec{a}} :: \phi_{\vec{a}}^{\mathrm{prior}}$ ▷ Append new prior action to strategy
  end if
  $E \leftarrow \emptyset$
  for $0...e_{\max}$ **do**
    $h \sim \phi_h$
    $\vec{a} \sim (\vec{\phi}_{\vec{a}}, h)$
    $v \leftarrow \textsc{eval}'(s, \vec{a})$
    $E \leftarrow E \cup (\vec{a}, v)$
  end for
  $\phi_h \leftarrow (1 - \alpha_h)\phi_h \cdot \alpha_h \cdot \textsc{fit}_h(E)$
  $\vec{\phi}_{\vec{a}} \leftarrow (1 - \alpha_{\vec{a}})\phi_{\vec{a}} \cdot \alpha_{\vec{a}} \cdot \textsc{fit}_{\vec{a}}(E)$
  return $\vec{\phi}_{\vec{a}}$
end procedure

5.4.2.1 System Performance

Figure 5.16 shows the time needed by C^3 to collect the targets in comparison to TACE. It is able to perform the desired task much faster. This is not surprising, as the acting and planning are executed in parallel. Also, as the agent acts w.r.t. a strategy that is continuously updated with information about the current state of the environment, reactions to permanent target movement can be found in a much more flexible way by C^3 than by TACE.

5.4.2.2 Simulation Steps

C^3 also need less simulation steps to compile the information needed for task completion than TACE. This reflects the fact that information from the simulation is immediately compiled into an action, which is continuously updated according to the latest simulation results. On the other hand. TACE acts after performing a certain number of simulations, which may have provided information that is already outdated when executing the determined action due to domain noise (i.e. target movement also takes place while the agent is planning). Figure 5.17 shows median and interquartile range of data about simulation steps measured performed by C^3 and TACE in our experiments.

5.4.2.3 Executed Actions

We also observed the number of actions that were optimized and executed by an agent during operation. For TACE , we measured the number of planned and executed actions. For C^3 , we measured the number of times the duration mean of the currently executed action (the first action in the strategy) dropped below zero. Figure 5.18 shows the median and the interquartile range of the data measured in the experiments. Even though C^3 needs much less time to collect the targets, it plans and optimizes about three times as many distinguishable actions than TACE. This observation reflects the higher flexibility of C^3 in comparison to TACE due to using more actual information

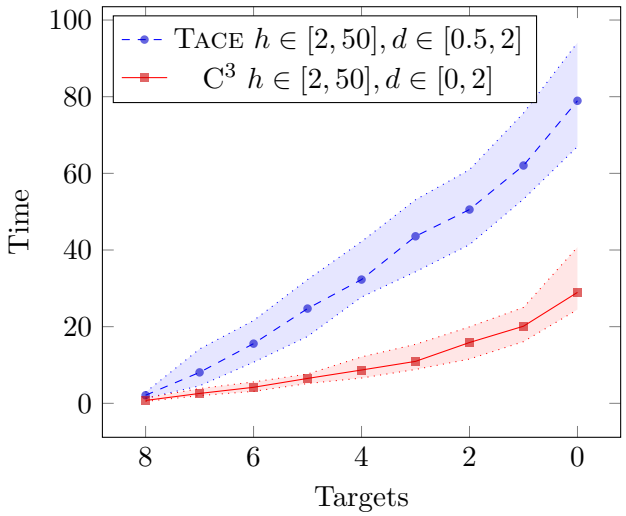


Figure 5.16: *Performance comparison of* TACE *and* C^3

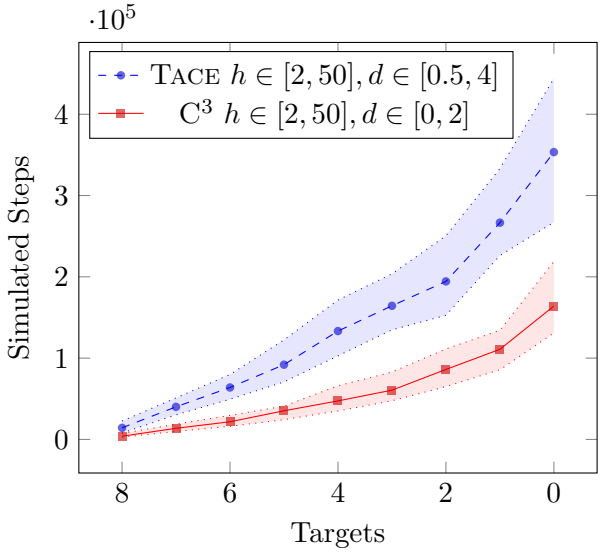


Figure 5.17: *Simulation steps performed by* TACE *and* C^3

when acting and planning in parallel, thus being more reactive to current environmental circumstances.

5.5 Related Work

HOOT [MWL11] and its online variant HOLOP [Wei14, WL12] are simulation based planners for continuous domains based on Hierarchical Optimistic Optimization [BSSM09]. They select actions for simulation by explicitly maintaining a search tree that is built in a Monte Carlo fashion in the course of planning. The search tree representation brings along increased memory and computation costs when selecting actions and updating their respective return, and information from simulations does only impact the resulting plan distribution locally. In contrast, cross entropy optimization updates the whole plan distribution in each generation.

In contrast to planning for continuous domains, discrete planning has a vast body

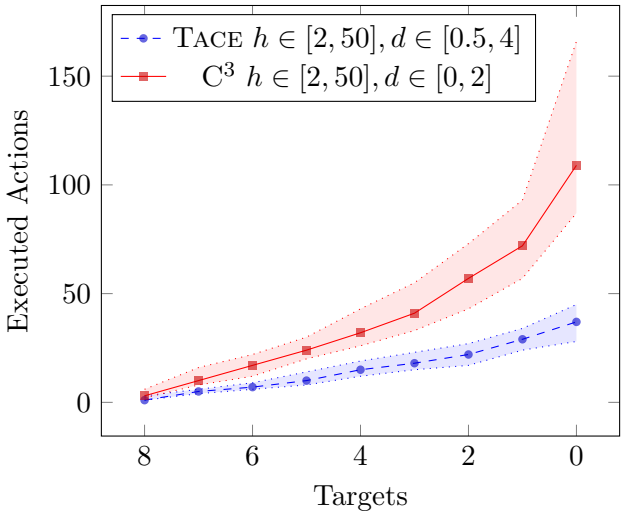


Figure 5.18: *Actions executed by* TACE *and* C^3

of research. Recent discrete online planners are e.g. PROST [KH13] based on Monte Carlo Tree Search [BPW$^+$12] and UCT [KS06], and GOURMAND [KDMW12] based on Labeled Real-Time Dynamic Programming [BG03]. They have both proven successful in the ICAPS international planning competition's discrete tracks recently. In fact, the most recent variant of GOURMAND was partially motivated by the question of dynamically determining the planning horizon, which is a fixed parameter in PROST. The work on TACE and C^3 is also rooted in these ideas from discrete planning.

The cross entropy method has originally been proposed in the context of rare event simulation, and has subsequently been extended to become a tool for stochastic combinatorial optimization [dBKMR05, RK13]. It has been successfully applied to robotic motion planning [Kob12], and recently for generating agent policies w.r.t. goal specifications in linear temporal logic [LWM15].

5.6 Summary & Outlook

This Chapter introduced Time-Adaptive Cross Entropy planning (TACE) for temporally flexible planning in continuous domains with infinite state-action spaces and branching factors. It is based on Cross Entropy Open Loop Planning (CEOLP) and is based on the idea of increasing planning flexibility by optimizing planning horizon as well as duration of individual actions. Doing so yields planners that first sample the global solution space for local subtasks, and subsequently concentrate their planning resources on those subtasks to solve them efficiently. We evaluated TACE empirically by comparing it to CEOLP for various configurations. We found that TACE planning agents performed more effectively than CEOLP agents. TACE required both fewer re-planning and simulation steps. As simulation typically is the most expensive operation in cross entropy based planning, this property effectively improves flexibility of agents.

This Chapter also introduced Continuous Time Cross Entropy Control (C^3), an approach to parallelization of action execution and deliberation for frame-based online planning agents. The idea is to maintain distributions about optimal durations of actions, and to adjust the parameters of these optimized distributions in concordance with the observed passing of time. In the case of cross entropy planning, the mean

of the duration parameter distribution is shifted w.r.t. passing time while optimizing action parameters as in TACE at the same time. Is was shown empirically that an agent planning with C^3 is able to pursue its tasks effectively and flexibly in continuous state, action and time domains.

An interesting direction for future research would be the investigation of continuous time series regression for plan aggregation. For example, dynamic time warping allows to define a distance measure on time series, enabling flexible temporal pattern detection in ordered data structures [KR05, PFW+14]. Interpreting a plan and the corresponding expected reward as a multivariate time series could enable corresponding data mining techniques for online analysis of plans and their consequences.

It would also be interesting to combine time adaptive planning and continuous control as discussed in this Chapter with alternative meta-heuristic search techniques based on importance sampling. An interesting candidate are for example genetic algorithms [GH88, DM13]. In this case genomes could represent sequences of actions (i.e. plans). A genetic algorithm would maintain a number of genomes in its population. These genomes would be evaluated by simulation (as for TACE and C^3). The principles of selection, mixture and variation would yield populations that in average increase their expected reward. Time adaptation and continuous control could probably be used in this framework straightforwardly by including action duration and genome length as parameters to optimize, and by continuously aging genomes based on frame times.

When using cross-entropy, plans are built iteratively and are efficiently represented in terms of a matrix that is easily communicated. Therefore, TACE and C^3 lend themselves to planning in multi-agent systems. Applying TACE or C^3 for coordination of many autonomous agents would be an interesting direction for future research.

Chapter 6

Conclusion and Outlook

> Coming back to where you started is not the same as never leaving.
>
> Terry Pratchett, *A Hat Full of Sky*

6.1 Conclusion

In this thesis, we studied the problem of building systems that are able to act autonomously, adequately and efficiently in dynamically changing, complex application domains (c.f. problem statement in Chapter 1, section 1.2). The key idea is to provide a system with a space of solutions, based on a high-level model of the domain dynamics and a goal specification. In Chapter 1 we discussed how by applying Monte Carlo sampling techniques through a simulation of the domain, computation of useful behavioral solutions at runtime becomes feasible even for large domains. In particular, using importance sampling to concentrate deliberation effort to high-value regions of the behavioral search space allows to efficiently compile goal-driven system behavior. By taking an online planning approach that parallelizes planning and execution, unexpected changes are incorporated into the planning process, and planning effort is guided towards regions of the search space that are relevant w.r.t. the current situation.

In terms of technical contributions, the thesis comprises four main parts. In Chapter 2 we discussed OnPlan, a framework for simulation based system autonomy that is centered around the ideas of online planning and importance sampling of the behavioral space in order to efficiently compile goal-driven system behavior. We described the framework both in terms of a mathematical definition and corresponding component model. We also defined the central algorithms that realize the online planning and importance sampling concepts within the framework. We showed framework instantiation in discrete domains by Monte Carlo Tree Search, and in continuous domains by Cross Entropy Open Loop Planning. We evaluated both instantiations empirically, showing the effective policy value estimation capabilities of the framework and its flexibility w.r.t. unexpected events or changing system goals at runtime.

In Chapter 3 we discussed Monte Carlo Action Programming (Mcap), a nondeterministic procedural action programming language for restricting the behavioral search space by expert knowledge. Mcap is interpreted at runtime in an online planning fashion by Monte Carlo Tree Search, allowing to evaluate consequences of program choices in very large and complex application domains. We outlined an how Mcap can

be regarded as an extension to the OnPlan framework. We evaluated the effectiveness of search space reduction for autonomous systems with Mcap empirically. Mcap yields measurable gain in behavioral effectiveness and flexibility w.r.t. system goal realization in comparison to an unconstrained Monte Carlo Tree Search planner.

The approach to system autonomy studies in this thesis relies on a simulation of the application domain. While this simulation may be defined and implemented at design time of a system, it is also possible to learn a simulation from runtime observations of domains dynamics. This allows a system to adapt the simulation used by its deliberation process to changes in the environmental dynamics, but also to overcome potentially erroneous design time specifications of domain simulations. In Chapter 4 we discussed Relational Probabilistic Action Forests (Rpaf), an approach to learning domain simulations from runtime observations that are available as discrete object-oriented data. Rpaf builds a probabilistic decision forest classifier to allow for predictions annotated with the degree certainty that is associated to each prediction, which is related to observation frequency of a particular dynamic event. Rpaf exploits relational structure in observations, which allows for a large degree of generalization when appropriate. The effectiveness of Rpaf for simulation based planning was demonstrated empirically by showing that a planner using a simulation learned from runtime observations achieves performance to a planner that uses a perfect simulation of the application domain.

In Chapter 5 we discussed Time-Adaptive Cross Entropy Planning (Tace). It is based on two central ideas: First, Monte Carlo sampling is typically the most expensive operation in simulation based online planning. Therefore, Tace proposes to optimize sampling depth in parallel to system action parametrization in order to concentrate the sampling effort on local problems that have been identified in previous global sampling steps. Second, Tace incorporates *timing* as a first-class property to online system behavioral planning. Deliberating not only about what to do, but also about when to do it may provide additional flexibility to any system that acts autonomously. Tace formalizes this concept for continuous state, action and time domains by incorporating action duration as an explicit action parameter to be optimized. Based on the concept of action duration and its corresponding optimization, we derived Continuous Cross Entropy Control (C^3) that enables continuous real-time optimization of system behavior in parallel to system execution. We evaluated both Tace and C^3 empirically, finding that they greatly outperform an existing state-of-the-art online planner (Cross Entropy Open Loop Planning) for continuous domains in terms of simulation effort, flexibility and planning performance.

6.2 Limitations & Possibilities

We will conclude this thesis by discussing limitations and possibilities of simulation-based online planning.

6.2.1 Limitations

The statistical nature of the simulation-based approach to system autonomy yields a number of limitations. We will discuss these in the following sections, and propose potential solutions.

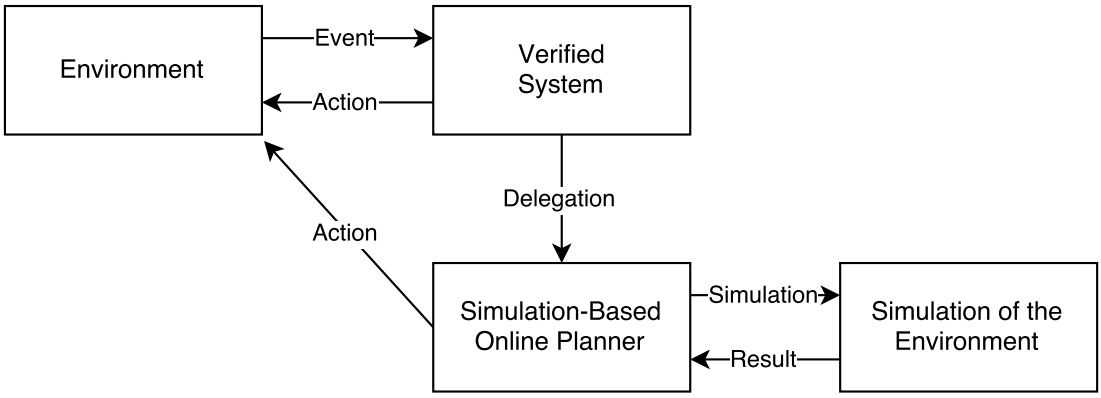


Figure 6.1: *Combining formally verified and statistically synthesized behavioral components. Arrows indicate the flow of events.*

6.2.1.1 Approximate Results & Safety Guarantees

Behavior generated by simulation-based online planning is based on approximation and empirical evidence rather than rigorous symbolic deduction. This means that results are only valid up to some level of statistical confidence. However, due to the inherent uncertainty of modern application domains, we consider it reasonable, if not necessary, to deal with this uncertainty explicitly and to exploit it in the reasoning process. Also, techniques for statistical decision making can probably be used in parallel with exact logical or algebraic reasoning methods.

However, in particular situations rigorous guarantees about system behavior are valuable or even necessary. For example, in safety critical systems formal verification of behavior may be a key factor. Legal issues may heavily rely on formal verification approaches which at least allow to guarantee that in the case some bad thing happens it was not the fault of the autonomous system.

To this end, an architecture that is built of different behavioral components would be an approach to allow for integration of safety critical, verifiable behavior and flexible, autonomous system behavior determined by simulation-based online planning. In this approach, a top level system could query the current state for particular situations that have to be dealt with in a formally verified way. In a safe situation, the verified component may then decide to trigger a component that is more cognitively involved. Such a cognitive component could for example be driven by a simulation-based online planner such as MCAP or TACE. Figure 6.1 sketches an according component model. Note that this approach could straightforwardly be extended to larger stacks of behavioral components.

6.2.1.2 Finite Horizons in Perpetual Domains

In its current state, simulation-based online planning uses a finite horizon for generating informations from simulations. When the horizon is reached, a zero reward is assumed. Figure 6.2 illustrates this operation. Obviously, this is an overly simplified approximation of future expected reward in perpetually progressing scenarios.

One approach to deal with this problem is to exploit available simulations of the application domain to learn an approximation of the value function $V : \mathcal{S} \to \mathcal{R}$. A value function maps states to expected future rewards. An approximation of this function can be learned by executing a simulation-based online planner in a meta-simulation of

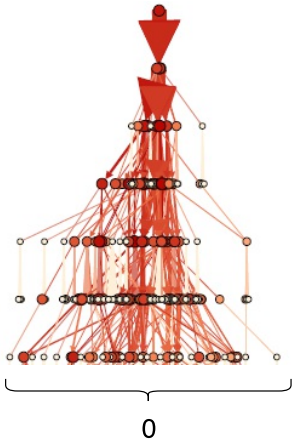


Figure 6.2: *Basic simulation-based online planners provide no expected reward when reaching the finite simulation horizon.*

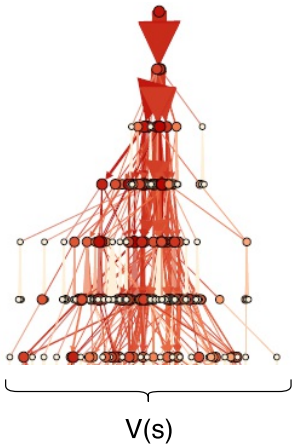


Figure 6.3: *An approximation of the value function $V : \mathcal{S} \rightarrow \mathcal{R}$ mapping states to expected future rewards can be learned from meta-simulation of a planner in a perpetual domain. The learned function can be used to provide final simulation states with a more accurate value estimation.*

the environment. This yields a dataset mapping states to observed future rewards. The approximation of the value function can be learned based on this simulated dataset. A planner can then use this approximation of the value function to provide final simulation states with a more accurate value estimation. Figure 6.3 sketches this approach by a correspondingly changed finite horizon search tree of a simulation-based online planner.

6.2.1.3 High Dimensional Domains & Adaptive Abstraction

The kind of abstraction used for representation of perceptional data, actuator domains and problem encoding has a crucial impact on the behavioral synthesis of a system at runtime. The level of abstraction defines how much of raw sensory data is used in the search and optimization process of online planning. Using less data means a reduction of the search space and thus a potential gain of efficiency when searching and evaluating potential behavioral alternatives. For example, stochastic optimization techniques such as cross entropy minimization or similar meta-heuristic search techniques are known to work well for problems of small to medium size and complexity. On very large

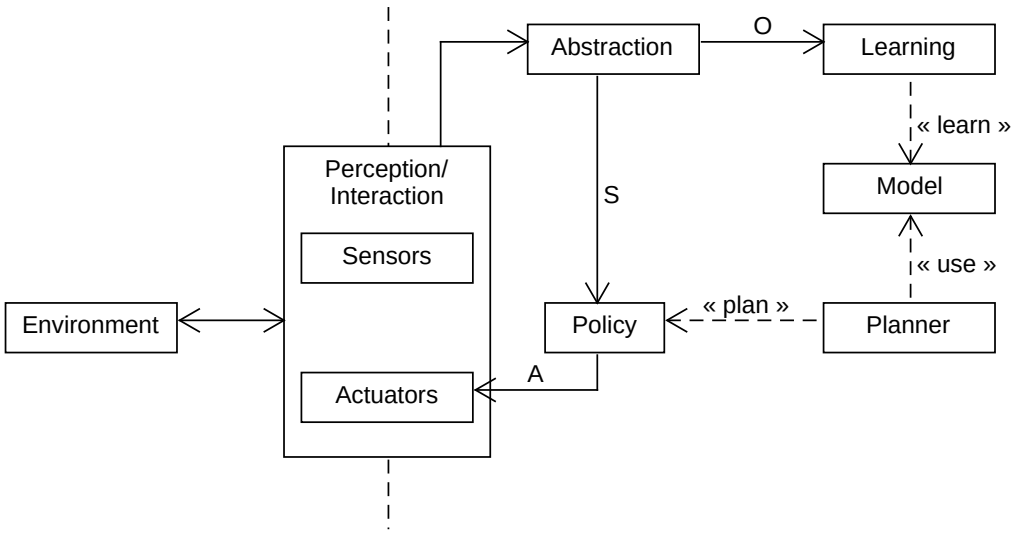


Figure 6.4: *Abstraction in an autonomous system.*

problems with many representational dimensions, their random search initialization may not provide an initial hook into the target landscape in a reasonable amount of time. On the other hand, reducing the problem dimensionality by too much abstraction may result in ignoring relevant information, yielding weak system performance.

For these reasons, it may be highly valuable to balance the trade-off between problem size reduction and representation quality w.r.t. application domain and system goals. While this abstraction process can be performed in an ad-hoc manner manually (as when e.g. modeling a class diagram for system perception), it seems valuable to enable autonomous systems to decide on an adequate degree of abstraction adaptively in order to (a) cope with the sheer amount of features and (b) with changing optimal degrees of abstraction at runtime, be it due to a change in the environment or a change of the current system goals.

Abstraction techniques such as unsupervised or semi-supervised learning try to exploit inherent variation and structure in data to decide on which features and correlations to use as relevant, and which ones to drop from the information transforming processes such as planning. Recently, such techniques extracted useful abstractions for tasks such as digit recognition or speech synthesis [HOT06, DSY+10]. The achieved performance was close to that of humans at times [KSH12].

Figure 6.4 outlines the role of representational abstraction in a schematic view of an autonomous system (cf. figure 1.3 in Chapter 1). The abstraction process can be seen as a kind of preprocessing step before applying the techniques that yield system autonomy as studied in this thesis. The learning and planning operations are then based on the abstractions chosen by the system for the particular task currently to be realized.

6.2.2 Possibilities

Simulation-based online planning provides much potential for extensions in various directions. We will outline some of these directions in the following.

6.2.2.1 Quantum Monte Carlo Methods

Quantum computation exploits quantum-mechanical properties to allow encoding and manipulation of data in form of so called quantum bits (*qubits*) [Fey82, NC10]. In contrast to classic bits which store information by realizing one of two discrete states, qubits are able to encode much more information by representing any of a quantum's superpositions. While a classic computer with n bits is able to represent one of 2^n states at a time, a quantum computer with n qubits is able to represent and manipulate 2^n states in parallel. When measuring the result of such a computation, the current configuration is collapsed into a single state to serve as computation output. In an informal sense, qubit states can be regarded as a quantum-physical representation of a probability distribution of its state.

The relation to the work presented in this thesis thus becomes clear. The ability to represent and manipulate probability distributions directly on a quantum-physical level renders the computation and evaluation of probabilistic simulations extremely efficient. While this computational speedup has been known in theory for some time, realizing quantum computation faces very hard physical challenges. Nevertheless, various steps towards building quantum computers have been reported in recent years (see e.g. [VYH$^+$14, CMS$^+$15]). Just recently, an empirical study has shown the speedup of using a quantum computer over classical approaches such as simulated annealing to reach a factor of ca. 10^8 for a particular optimization problem [DBI$^+$15]. System autonomy as outlined in this thesis is based on probabilistic online optimization, and the application of quantum computers for simulation based system autonomy seems a natural direction for potential future work.

6.2.2.2 Surprisal

This thesis has been concerned with efficiently generating information about potential behavioral decisions to enable flexible system autonomy. While the approach discussed in this thesis emphasized importance sampling by value, another way to efficiently generate valuable information about decision making at runtime is to concentrate deliberation based on *surprise* [IB05, IB09, RD09, BI10] or related concepts such as curiosity and creativity [Sch06, Sch12, FLS$^+$13]. In contrast to the approach to simulation-based online planning discussed in this thesis, accounting for surprise in an information-theoretic sense would incorporate the degree of change that is observed into the planning process.

Informally, surprise can be seen as a gradient of a probability distribution. That is, if a system encounters situations that produce perceptional data in proportion to a distribution that is anticipated due to some deliberation process, it may simply continue to execute the behavior it found to be optimal, as nothing surprising happened. This also means that while executing and not encountering any surprising situations, the system can use its resources for deliberating about completely different alternatives, future plans, or any other computationally intensive tasks.

Monitoring the degree of surprise (i.e. the deviation of actual from anticipated events) in the course of execution allows to identify situations that have not been anticipated by former deliberation processes. In such a case, the system could allocate computational resources to adapt its anticipation to the current course of events. This would effectively enables adaptive resource allocation for planning processes w.r.t. the relation of anticipation and current perception.

Also within the simulation-based planning process surprise could potentially be

used to steer the simulation process towards high-surprise regions in the search space, rather than concentrating on high-value regions only. The idea is that if we already identified a high-value region, it may be more valuable to explore the search space in other directions, potentially based on the degree of surprise. This may yield an interesting approach to the exploration-exploitation trade-off (see e.g. [AMS09]), with deep roots in Bayesian statistics, information theory and probability theory.

6.2.2.3 Distribution, Coordination & Emergence

In this thesis, we concentrated our studies on system autonomy realized by a single agent. That is, simulation and corresponding evaluation of behavioral choices are performed in a unique component serving as a single point of information. While this restriction allows to study the general requirements and concepts of autonomy, it is usually the case that multiple autonomous systems interact with each other at runtime. This necessitates coordination of systems, and incorporation of expectations about potential future behavior of other systems into the decision making process.

Autonomous system coordination can be seen from two different perspectives.

1. Task and resource allocation. Systems have to identify independent and mutually conditional tasks to be completed, and to assign available resources in ways that enable or alleviate goal satisfaction. This step can be seen as a kind of problem decomposition.

2. System coordination and execution. After tasks and resources have been assigned, execution has to be monitored and system actions have to be coordinated in order to enable or alleviate the realization of particular tasks. Coordination can be seen as solving the partial problems resulting from the decomposition of the original top-level problem.

In application scenarios that exhibit unexpected change at runtime, task and resource allocation as well as system coordination should be consequently optimized w.r.t. existing knowledge about domain dynamics. In fact, it is possible to solve the allocation and coordination problems in the same online manner as the decision problem. Here, consistently monitoring the current situation in parallel to efficiently computing solutions to allocation and coordination would allow for autonomous assignment of tasks, resources and systems in complex and dynamically changing environments.

Scientific frameworks that study distributed optimization through task allocation, resource distribution and behavioral coordination are distributed constraint optimization (see e.g. [ML04, MSTY05]), multi-agent optimization [NO09] and consensus theory [NOP+10, SLY+14]. It seems an interesting direction for future research to leverage ideas from these frameworks into distributed simulation-based system autonomy.

An additional challenge for distributed system autonomy is the corresponding distribution of information. An autonomous system can only decide on its behavior based on the currently available information. This means that it is crucial to define what information to share when for a distributed autonomous system that is to operate efficiently and scalable. In terms of the perspectives of allocation and coordination mentioned above, information could be seen as a resource to be assigned.

Also, to enable efficient decision making by approaches as discussed in this thesis it is necessary to find algorithms and mechanisms that allow to effectively adapt the search space to relevant, high-value regions. A potential venue in the context of simulation-based system autonomy would be to enable agents to share or request their respective

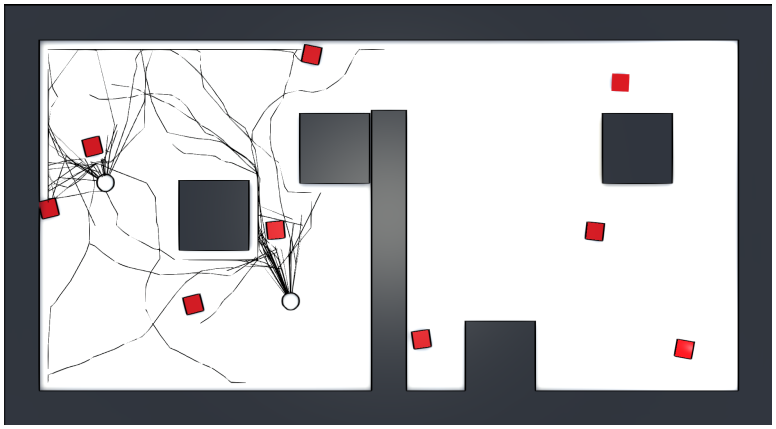


Figure 6.5: *Two* TACE *agents planning and acting in parallel.*

current probabilistic strategies. This would enable individual agents to incorporate the probabilistic choices of their counterparts into their simulations (i.e. their search process), thus enabling efficient adaptation of their own behavioral policies

Figure 6.5 shows two TACE agents operating in a shared environment. By enabling them to exchange information about their current optimized policies, the potential high-value choices of the agents could be aligned to increase chances of reaching a global optimum.

Appendix A

MCAP Normal Form Termination

The following Equations define a reduction system for Monte Carlo Action Programs to their normal form (cf. Chapter 3). This reduction system was shown to be locally confluent (and sort decreasing using sorts for actions and programs, respectively) with the Maude Church-Rosser Checker [DM10].

$$\epsilon ; p = p \tag{A.1}$$
$$p + p = p \tag{A.2}$$
$$(p_1 + p_2) ; p = (p_1 ; p) + (p_2 ; p) \tag{A.3}$$
$$p ; (p_1 + p_2) = (p ; p_1) + (p ; p_2) \tag{A.4}$$
$$p \parallel (p_1 + p_2) = (p \parallel p_1) + (p \parallel p_2) \tag{A.5}$$
$$(a_1 ; p_1) \parallel (a_2 ; p_2) = (a_1 ; (p_1 \parallel (a_2 ; p_2))) + (a_2 ; ((a_1 ; p_1) \parallel p_2)) \tag{A.6}$$
$$a_1 \parallel (a_2 ; p) = (a_1 ; a_2 ; p) + (a_2 ; (a_1 \parallel p)) \tag{A.7}$$
$$a_1 \parallel a_2 = (a_1 ; a_2) + (a_2 ; a_1) \tag{A.8}$$

We show termination by multi-set (or recursive) path ordering [Der79] with precedence order $\parallel \succ ; \succ +$ on function symbols. We denote the multi-set extension of $\succ$ by $\gg$ [DM79, HO80]. As the reduction system is locally confluent and terminating, every MCAP has a unique normal form that can be determined in finite time.

Equations A.1 and A.2 are trivial.

Equation A.3 (and similarly Equation A.4).

$$\begin{aligned}(p_1 + p_2) ; p \succ (p_1 ; p) + (p_2 ; p)\\ \textbf{as } ; \succ +\\ \textbf{and } (p_1 + p_2) ; p \succ p_1 ; p\\ \textbf{and } (p_1 + p_2) ; p \succ p_2 ; p\end{aligned}$$

Equation A.5.

$$\begin{aligned}p \parallel (p_1 + p_2) \succ (p \parallel p_1) + (p \parallel p_2)\\ \textbf{as } \parallel \succ +\\ \textbf{and } p \parallel (p_1 + p_2) \succ p \parallel p_1\\ \textbf{and } p \parallel (p_1 + p_2) \succ p \parallel p_2\end{aligned}$$

Equation A.6.

$$(a_1 ; p_1) \parallel (a_2 ; p_2) \succ (a_1 ; (p_1 \parallel (a_2 ; p_2))) + (a_2 ; ((a_1 ; p_1) \parallel p_2))$$
$$\textbf{as} \;\; \parallel \succ +$$
$$\textbf{and} \;\; (a_1 ; p_1) \parallel (a_2 ; p_2) \succ a_1 ; (p_1 \parallel (a_2 ; p_2))$$
$$\textbf{and} \;\; (a_1 ; p_1) \parallel (a_2 ; p_2) \succ a_2 ; ((a_1 ; p_1) \parallel p_2)$$

$$(a_1 ; p_1) \parallel (a_2 ; p_2) \succ a_1 ; (p_1 \parallel (a_2 ; p_2))$$
$$\textbf{as} \;\; \parallel \succ ;$$
$$\textbf{and} \;\; (a_1 ; p_1) \parallel (a_2 ; p_2) \succ a_1$$
$$\textbf{and} \;\; (a_1 ; p_1) \parallel (a_2 ; p_2) \succ p_1 \parallel (a_2 ; p_2)$$

$$(a_1 ; p_1) \parallel (a_2 ; p_2) \succ p_1 \parallel (a_2 ; p_2)$$
$$\textbf{as} \;\; \parallel = \parallel$$
$$\textbf{and} \;\; \{(a_1 ; p_1), (a_2 ; p_2)\} \gg \{p_1, (a_2 ; p_2)\}$$
$$\quad \textbf{as} \;\; a_1 ; p_1 \succ p_1$$

$$(a_1 ; p_1) \parallel (a_2 ; p_2) \succ a_2 ; ((a_1 ; p_1) \parallel p_2)$$
$$\textbf{as} \;\; \parallel \succ ;$$
$$\textbf{and} \;\; (a_1 ; p_1) \parallel (a_2 ; p_2) \succ a_2$$
$$\textbf{and} \;\; (a_1 ; p_1) \parallel (a_2 ; p_2) \succ (a_1 ; p_1) \parallel p_2$$

$$(a_1 ; p_1) \parallel (a_2 ; p_2) \succ (a_1 ; p_1) \parallel p_2$$
$$\textbf{as} \;\; \parallel = \parallel$$
$$\textbf{and} \;\; \{(a_1 ; p_1), (a_2 ; p_2)\} \gg \{(a_1 ; p_1), p_2\}$$
$$\quad \textbf{as} \;\; a_2 ; p_2 \succ p_2$$

Equation A.7.

$$a_1 \parallel (a_2 ; p) \succ (a_1 ; a_2 ; p) + (a_2 ; (a_1 \parallel p))$$
$$\textbf{as} \;\; \parallel \succ +$$
$$\textbf{and} \;\; a_1 \parallel (a_2 ; p) \succ a_1 ; a_2 ; p$$
$$\quad \textbf{as} \;\; \parallel \succ ;$$
$$\quad \textbf{and} \;\; a_1 \parallel (a_2 ; p) \succ a_1$$
$$\quad \textbf{and} \;\; a_1 \parallel (a_2 ; p) \succ a_2 ; p$$
$$\textbf{and} \;\; a_1 \parallel (a_2 ; p) \succ a_2 ; (a_1 \parallel p)$$
$$\quad \textbf{as} \;\; \parallel \succ ;$$
$$\quad \textbf{and} \;\; a_1 \parallel (a_2 ; p) \succ a_2$$
$$\quad \textbf{and} \;\; a_1 \parallel (a_2 ; p) \succ a_1 \parallel p$$

Equation A.8 is trivial. □

Bibliography

[ACBF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 138(1):3–34, 1995.

[AMS09] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19):1876 – 1902, 2009. Algorithmic Learning Theory.

[ANW12] Alekh Agarwal, Sahand Negahban, and Martin J Wainwright. Stochastic optimization and sparse statistical recovery: Optimal algorithms for high dimensions. In *Advances in Neural Information Processing Systems*, pages 1538–1546, 2012.

[BB12] Chaithanya Bandi and Dimitris Bertsimas. Tractable stochastic analysis in high dimensions via robust optimization. *Mathematical programming*, 134(1):23–70, 2012.

[BC12] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, 5(1):1–122, 2012.

[Bel57a] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

[Bel57b] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[Bel13] Lenz Belzner. Action programming in rewriting logic. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5-Online-Supplement), 2013.

[Bel14] Lenz Belzner. Verifiable decisions in autonomous concurrent systems. In Eva Kühn and Rosario Pugliese, editors, *COORDINATION*, volume 8459 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2014.

[Belar] Lenz Belzner. Time-adaptive cross entropy planning. In *31st ACM Symposium on Applied Computing 2016*, to appear.

[BG03] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS*, volume 3, pages 12–21, 2003.

[BHW15] Lenz Belzner, Rolf Hennicker, and Martin Wirsing. OnPlan: A framework for simulation-based online planning. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, pages 1–30, 2015.

[BI10] Pierre Baldi and Laurent Itti. Of bits and wows: a bayesian theory of surprise with applications to attention. *Neural Networks*, 23(5):649–666, 2010.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[BK+08] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[BM58] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *Ann. Math. Statist.*, 29(2):610–611, 06 1958.

[BNar] Lenz Belzner and Alexander Neitz. Learning relational probabilistic action models for online planning with decision forests. In *31st ACM Symposium on Applied Computing 2016*, to appear.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[BR98] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1–2):285 – 297, 1998.

[Bre96] Leo Breiman. Technical note: Some properties of splitting criteria. *Machine Learning*, 24(1):41–47, 1996.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[Bri50] Glenn W Brier. Verification of forecasts expressed in terms of probability. *Monthly weather review*, 78(1):1–3, 1950.

[BRS+00] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, Sebastian Thrun, et al. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI/IAAI*, pages 355–362, 2000.

[BSSM09] Sébastien Bubeck, Gilles Stoltz, Csaba Szepesvári, and Rémi Munos. Online optimization in x-armed bandits. In *Advances in Neural Information Processing Systems*, pages 201–208, 2009.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[CDJSU06] GMJB Chaslot, Steven De Jong, Jahn-Takeshi Saito, and JWHM Uiterwijk. Monte-carlo tree search in production management problems. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 91–98. Citeseer, 2006.

[CMS+15] AD Córcoles, Easwar Magesan, Srikanth J Srinivasan, Andrew W Cross, M Steffen, Jay M Gambetta, and Jerry M Chow. Demonstration of a quantum error detection code using a square lattice of four superconducting qubits. *Nature communications*, 6, 2015.

[CSK12] Antonio Criminisi, Jamie Shotton, and Ender Konukoglu. *Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning.* Now, 2012.

[DBI+15] Vasil S Denchev, Sergio Boixo, Sergei V Isakov, Nan Ding, Ryan Babbush, Vadim Smelyanskiy, John Martinis, and Hartmut Neven. What is the computational value of finite range tunneling? *arXiv preprint arXiv:1512.02206*, 2015.

[dBKMR05] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tutorial on the cross-entropy method. *Annals OR*, 134(1):19–67, 2005.

[Der79] Nachum Dershowitz. Orderings for term-rewriting systems. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 123–131. IEEE, 1979.

[DGLL00] Giuseppe De Giacomo, Yves Lespérance, and Hector J Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1):109–169, 2000.

[Dia09] Persi Diaconis. The markov chain monte carlo revolution. *Bulletin of the American Mathematical Society*, 46(2):179–205, 2009.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DM10] Francisco Durán and José Meseguer. A church-rosser checker tool for conditional order-sorted equational maude specifications. In *Rewriting Logic and Its Applications*, pages 69–85. Springer, 2010.

[DM13] Dipankar Dasgupta and Zbigniew Michalewicz. *Evolutionary algorithms in engineering applications.* Springer Science & Business Media, 2013.

[DR03] Kurt Driessens and Jan Ramon. Relational instance based regression for relational reinforcement learning. In *ICML*, pages 123–130, 2003.

[DRB01] Kurt Driessens, Jan Ramon, and Hendrik Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In *Machine Learning: ECML 2001*, pages 97–108. Springer, 2001.

[Dri10] Kurt Driessens. Relational reinforcement learning. In *Encyclopedia of Machine Learning*, pages 857–862. Springer, 2010.

[DRKT07] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467, 2007.

[DSY+10] Li Deng, Michael L Seltzer, Dong Yu, Alex Acero, Abdel-rahman Mohamed, and Geoffrey E Hinton. Binary coding of speech spectrograms using a deep auto-encoder. In *Interspeech*, pages 1692–1695. Citeseer, 2010.

[Dur10] Rick Durrett. *Probability: theory and examples*. Cambridge university press, 2010.

[EM04] Saher Esmeir and Shaul Markovitch. Lookahead-based algorithms for anytime induction of decision trees. In *Proceedings of the twenty-first international conference on Machine learning*, page 33. ACM, 2004.

[Fey82] Richard P Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6):467–488, 1982.

[FLS+13] Mikhail Frank, Jürgen Leitner, Marijn Stollenga, Alexander Förster, and Jürgen Schmidhuber. Curiosity driven reinforcement learning for motion planning on humanoids. *Frontiers in neurorobotics*, 7, 2013.

[GDR03] Thomas Gärtner, Kurt Driessens, and Jan Ramon. Graph kernels and gaussian processes for relational reinforcement learning. In *Inductive Logic Programming*, pages 146–163. Springer, 2003.

[Get07] Lise Getoor. *Introduction to statistical relational learning*. MIT press, 2007.

[GH88] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[GKS+12] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Commun. ACM*, 55(3):106–113, 2012.

[GLLS09] Giuseppe Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina. Indigolog: A high-level programming language for embedded reasoning agents. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming:*, pages 31–72. Springer US, 2009.

[GMT14] Virginie Gabrel, Cécile Murat, and Aurélie Thiele. Recent advances in robust optimization: An overview. *European Journal of Operational Research*, 235(3):471–483, 2014.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[GS11] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[Ham13] John Hammersley. *Monte carlo methods*. Springer Science & Business Media, 2013.

[Has70] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[HG15] Matthias M. Hölzl and Thomas Gabor. Continuous collaboration: A case study on the development of an adaptive cyber-physical system. In *1st IEEE/ACM International Workshop on Software Engineering for Smart Cyber-Physical Systems, SEsCPS 2015*, pages 19–25, 2015.

[HO80] Gérard Huet and Derek C Oppen. Equations and rewrite rules. *Formal language theory: perspectives and open problems*, pages 349–405, 1980.

[HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[HQS12] Todd Hester, Michael Quinlan, and Peter Stone. Rtmba: A real-time model-based reinforcement learning architecture for robot control. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 85–90. IEEE, 2012.

[HS13] Todd Hester and Peter Stone. Texplore: real-time sample-efficient reinforcement learning for robots. *Machine learning*, 90(3):385–429, 2013.

[IB05] Laurent Itti and Pierre F Baldi. Bayesian surprise attracts human attention. In *Advances in neural information processing systems*, pages 547–554, 2005.

[IB09] Laurent Itti and Pierre Baldi. Bayesian surprise attracts human attention. *Vision research*, 49(10):1295–1306, 2009.

[Jay03] Edwin T Jaynes. *Probability theory: the logic of science*. Cambridge university press, 2003.

[KC03] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KDDR+11] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language problog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011.

[KDMW12] Andrey Kolobov, Peng Dai, Mausam Mausam, and Daniel S Weld. Reverse iterative deepening for finite-horizon mdps with large branching factors. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling, ICAPS*, 2012.

[Kep03] J Kephart. An architectural blueprint for autonomic computing. *IBM*, 2003.

[KH13] Thomas Keller and Malte Helmert. Trial-based Heuristic Tree Search for Finite Horizon MDPs. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*, pages 135–143. AAAI Press, June 2013.

[KKS13] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1238–1246, 2013.

[Kob12] Marin Kobilarov. Cross-entropy motion planning. *I. J. Robotic Res.*, 31(7):855–871, 2012.

[KR05] Eamonn Keogh and Chotirat Ann Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3):358–386, 2005.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[KW08] Malvin H Kalos and Paula A Whitlock. *Monte carlo methods*. John Wiley & Sons, 2008.

[Lee12] Peter M Lee. *Bayesian statistics: an introduction*. John Wiley & Sons, 2012.

[LWM15] Scott C. Livingston, Eric M. Wolff, and Richard M. Murray. Cross-entropy temporal logic motion planning. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15*, pages 269–278, 2015.

[Mac03] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[Mar05] Leonid Margolin. On the convergence of the cross-entropy method. *Annals of Operations Research*, 134(1):201–214, 2005.

[MCM13] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[Mes12] José Meseguer. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 81(7-8):721–781, 2012.

[ML04] Roger Mailler and Victor Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 438–445. IEEE Computer Society, 2004.

[MSTY05] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1):149–180, 2005.

[MWL11] Christopher R. Mansley, Ari Weinstein, and Michael L. Littman. Sample-based planning for continuous action markov decision processes. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS*, 2011.

[NC10] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[NO09] Angelia Nedić and Asuman Ozdaglar. Distributed subgradient methods for multi-agent optimization. *Automatic Control, IEEE Transactions on*, 54(1):48–61, 2009.

[NOP+10] Angelia Nedić, Asuman Ozdaglar, Pablo Parrilo, et al. Constrained consensus and optimization in multi-agent networks. *Automatic Control, IEEE Transactions on*, 55(4):922–938, 2010.

[PCWL14] Tom Pepels, Tristan Cazenave, MarkH.M. Winands, and Marc Lanctot. Minimizing simple and cumulative regret in monte-carlo tree search. In Tristan Cazenave, MarkH.M. Winands, and Yngvi Björnsson, editors, *Computer Games*, volume 504 of *Communications in Computer and Information Science*, pages 1–15. Springer International Publishing, 2014.

[PFW+14] François Petitjean, Germain Forestier, Geoffrey Webb, Ann E Nicholson, Yanping Chen, Eamonn Keogh, et al. Dynamic time warping averaging of time series allows faster and more accurate classification. In *IEEE International Conference on Data Mining (ICDM)*, pages 470–479. IEEE, 2014.

[PZK07] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, pages 309–352, 2007.

[RD09] Ananth Ranganathan and Frank Dellaert. Bayesian surprise and landmark detection. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 2017–2023. IEEE, 2009.

[RGRS10] Christophe Rodrigues, Pierre Gérard, Céline Rouveirol, and Henry Soldano. Incremental learning of relational action rules. In *Machine Learning and Applications (ICMLA), 2010*, pages 451–458. IEEE, 2010.

[RK11] Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*, volume 707. John Wiley & Sons, 2011.

[RK13] Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[Sch06] Jürgen Schmidhuber. Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2):173–187, 2006.

[Sch12] Jürgen Schmidhuber. A formal theory of creativity to model the creation of art. In *Computers and Creativity*, pages 323–337. Springer, 2012.

[SCM12] Abdallah Saffidine, Tristan Cazenave, and Jean Méhat. Ucd: Upper confidence bound for rooted directed acyclic graphs. *Knowledge-Based Systems*, 34:26–33, 2012.

[SDP06] Jan Struyf, Jesse Davis, and David Page. An efficient approximation to lookahead in relational learners. In *Machine Learning: ECML 2006*, pages 775–782. Springer, 2006.

[SGG+13] Dieter Spath, Oliver Ganschar, Stefan Gerlach, Moritz Hämmerle, Tobias Krause, and Sebastian Schlund. *Produktionsarbeit der Zukunft-Industrie 4.0*. Fraunhofer Verlag, 2013.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SLY+14] Wei Shi, Qing Ling, Kun Yuan, Gang Wu, and Wotao Yin. On the linear convergence of the admm in decentralized consensus optimization. *Signal Processing, IEEE Transactions on*, 62(7):1750–1761, 2014.

[SSM13] David Silver, Richard S. Sutton, and Martin Müller. Temporal-difference search in computer go. In Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors, *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI, 2013.

[SV10] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pages 2164–2172, 2010.

[SV13] Stefano Sebastio and Andrea Vandin. Multivesta: Statistical model checking for discrete event simulators. In *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools*, pages 310–315. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[Thi98] Michael Thielscher. Introduction to the fluent calculus. *Electron. Trans. Artif. Intell.*, 2:179–192, 1998.

[Thi05] Michael Thielscher. Flux: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming (TPLP)*, 5(4-5):533–565, 2005.

[VYH+14] M Veldhorst, CH Yang, JCC Hwang, W Huang, JP Dehollain, JT Muhonen, S Simmons, A Laucht, FE Hudson, KM Itoh, et al. A two qubit logic gate in silicon. *arXiv preprint arXiv:1411.5760*, 2014.

[Wei14] Ari Weinstein. *Local Planning for Continuous Markov Decision Processes*. PhD thesis, Rutgers, The State University of New Jersey, 2014.

[WF05] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[WHKM15] Martin Wirsing, Matthias Hölzl, Nora Koch, and Philip Mayer, editors. *Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project*, volume 8998 of *Lecture Notes in Computer Science*. Springer Verlag, Heidelberg, 2015.

[WL12] Ari Weinstein and Michael L. Littman. Bandit-based planning and learning in continuous-action markov decision processes. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling, ICAPS*, 2012.

[WL13] Ari Weinstein and Michael L. Littman. Open-loop planning in large-scale stochastic domains. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[WvO12] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-art*, volume 12. Springer Science & Business Media, 2012.