Relational Regression Methods
to Speed Up Monte-Carlo Planning

Teresa Böpple

Course of Study: Informatik

Examiner: Prof. Dr. rer. nat. Marc Toussaint

Supervisor: M.Sc. Hung Ngo,
Ph.D. Vien Ngo

Commenced: September 1, 2016
Completed: March 3, 2017

CR-Classification: I.2.8
Abstract

Monte-Carlo Tree Search is a planning algorithm that tries to find the best possible next action by using random simulations and estimating the return by them. One big advantage of Monte-Carlo Tree Search is that it can be used with very little domain knowledge, is implemented easily and is applicable for many problems. This thesis shows how Monte-Carlo planning is speed up by applying Relational Regression. A relational domain is given with states that consist of facts. In order to use regression on relation states, features have to be created that map the state to a vector that can be used by regression. Therefore, training data are created that contain states, possible actions and an estimated return of these state-action pairs. These data are created by a standard Monte-Carlo planner. All facts that occur in any state from the training data are written into a factlist. With the help of that factlist, features are created. Every row of the feature stands for a fact. If this fact occurs in a state, the feature of this state contains a 1 in the corresponding row. If not, the row contains a 0. Other features that are created also consider combinations of facts or actions. With the help of regression, these features can be mapped to a real value that corresponds to the expected return of the state or state-action pair. This is evaluated by testing it on a test dataset. The results of this test are that for a big enough and accurate training dataset the return calculated by regression is very close to the one calculated by the planner for the test data.

Because of this promising results, the regression is actually integrated into a Monte-Carlo planner. For a well chosen set of training data, that contain a wide range of both terminal- and dead-end states, the planner improves in many ways. First, in average the modified planner needs less steps to reach a terminal state than the original planner. Second, the modified planner reaches a terminal state more often than the original planner, because the planner gets into a dead-end state less often now. With this the actual goal of this work is achieved and it is demonstrated that the planning process speeds up.
Contents

1 Introduction .................................................. 9
  1.1 Goals ................................................... 9
  1.2 Structure .............................................. 10

2 Background and Related Work .................. 11
  2.1 Monte-Carlo Tree Search ......................... 11
  2.2 Regression ........................................... 12
  2.3 TILDE ................................................ 15
  2.4 IRL .................................................. 16

3 Program Setup ........................................... 19
  3.1 Data Representation ................................ 19
  3.2 Planner Example .................................. 20

4 Creation and Evaluation of Features ........ 23
  4.1 Factlist .............................................. 23
  4.2 First Feature: F .................................. 24
  4.3 Second Feature: FF² ................................ 24
  4.4 Third Feature: FA .................................. 25
  4.5 Fourth Feature: FF²A .............................. 25
  4.6 Regression .......................................... 25
  4.7 Evaluation .......................................... 27

5 Integration in the Monte-Carlo Planner .... 33

6 Evaluation ................................................ 35
  6.1 Feature F ............................................ 35
  6.2 Feature FA .......................................... 42

7 Summary and Outlook ............................... 45

A Kurzfassung .............................................. 47

B Bibliography .............................................. 49
List of Figures

2.1 The four steps of Monte-Carlo Tree Search .......................... 12
3.1 Rule releasing ............................................................. 20
3.2 Part of data file ............................................................. 21
4.1 Extract from the factlist ..................................................... 24
4.2 Results: \(\chi_{\text{train}}\) for all four features .......................... 29
4.3 Results: \(\chi_{\text{test}}\) for all four features .......................... 30
4.4 Results: \(\chi_{\text{train}}\) for all four features for bigger training dataset ........... 31
4.5 Results: \(\chi_{\text{test}}\) for all four features for bigger training dataset ........... 32
5.1 Integration of the Regression in the Monte-Carlo Planner .............. 34
6.1 Percentage of successful runs for standard Monte-Carlo planner and Monte-Carlo planner that uses regression with feature F ............. 36
6.2 Steps until terminal state is reached: Monte-Carlo planner versus Monte-Carlo planner that uses regression with feature F .................... 37
6.3 \(\chi_{\text{train}}\) and \(\chi_{\text{test}}\) for New Training Data ......................... 39
6.4 Percentage of successful runs for standard Monte-Carlo planner and Monte-Carlo planner that uses regression on new training data and feature F 40
6.5 Steps until terminal state is reached: Monte-Carlo planner versus Monte-Carlo planner that uses regression with new data and feature F .................... 41
6.6 Percentage of successful runs for standard Monte-Carlo planner and Monte-Carlo planner that uses regression with new data and feature FA .... 42
6.7 Steps until terminal state is reached: Monte-Carlo planner versus Monte-Carlo planner that uses regression with new data and feature FA .... 43
1 Introduction

Machine learning gained more and more importance over the past years. Instead of only following instructions, machine learning programs use the knowledge they get from examples to find the best action strategy. The goal of this is that the programs can adjust to a changing environment or task without being modified, and are thereby more flexibly usable. Machine learning is used for many different applications like speech recognition, game playing or economic movement prediction. One promising approach to improve planners like Monte-Carlo Tree Search is combining them with learning methods. Monte-Carlo Tree Search is a planner that is used in a wide range of domains and requires only very little domain knowledge. The Monte-Carlo planner tries to find the best possible action for a given state by applying randomized actions starting from that state and considering the results. The action with the most promising returns is chosen. In order to get robust and accurate results, many Monte-Carlo steps have to be performed. In large domains this can take a long time. One possibility to speed it up is to apply machine learning techniques within Monte-Carlo tree search. The Monte-Carlo Tree Search planner executes a few steps and passes its results to a learner application. The learner uses this results to deduce the return in further steps. Thereby the planner only needs to perform less steps for the planning in the future.

In a relational domain the states are represented by a set of grounded literals. Actions can change the state, the relational world is currently in. For example robot setups can be modeled as a relational world. A start state or current state has to be given. With the help of machine learning, the robot should be able to perform a task when it knows a termination rule. When a target state gets associated with a reward, the robot should be able to find the best strategy to reach such a state itself through learning.

1.1 Goals

In this thesis Relational Regression is used to calculate the expected return. A Monte-Carlo planner is used to calculate expected returns for relational state-action pairs. This planner works on a robot setup with three robot arms and three objects that have to be fixed together. With the help of this data, features are created over the relational states.
1 Introduction

Afterwards, Relational Regression is used to map these features to the expected return. In the next step, the creation of the features and the regression are integrated into the Monte-Carlo planner. The main goal is to speed up the Monte-Carlo planning by this.

1.2 Structure

In the next chapter the background and some related work is introduced. Monte-Carlo Tree Search and Regression are explained in more detail. In addition, similar concepts like TILDE and IRL are presented. In chapter 3 the program setup is described shortly. Especially the relational domain is explained. The main contribution of this work, the creation of the different features is explained in chapter 4. To evaluate these features, some tests were executed. These tests and results are also presented in chapter 4. The next step, the integration of the regression into the Monte-Carlo planner is shortly described in chapter 5. To evaluate the improved planner and compare it to the pure Monte-Carlo planner, more tests were executed. This is described in chapter 6. Last, the conclusion consists of a short summary and an outlook.
2 Background and Related Work

2.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search is, as its name implies, a tree search variant. Tree-Search algorithms are used to find the optimal action sequence for a given problem. In a search tree, states are represented as nodes. An action changes the state. So, in a search tree, actions are edges that connect the node that represents the state before the action was executed with the node for the state after the action was performed. Whereas most tree-search algorithms use a positional evaluation function, Monte-Carlo Tree Search only uses Monte-Carlo simulations [Cha10]. Random actions are taken, until a leaf node is reached. When this is executed often, it estimates the reward for the node, where the random actions started. Monte-Carlo Tree Search consists of four steps: First, the selection step, where a selection strategy is applied. Second, the expansion step that adds nodes to the tree. Third, the simulation step, which applies randomized actions until termination. In the last step, the back-propagation step, the results from the simulation step are propagated back through all visited nodes to the start node.

In the selection step a selection strategy has to be chosen and applied. The selection strategy aims to find a good balance between exploration and exploitation. If the selection strategy prefers exploration, actions that look less promising than others are taken in order to explore the tree. On the other hand, exploitation always takes the most promising action to receive the best reward. First, the selection strategy is applied at the root node. Then it is applied recursively, until a node is reached that is not part of the tree. At this point the expansion step starts.

The expansion step adds new nodes to the tree. An expansion strategy is used to decide how many child nodes of a given node are stored in the tree. The most popular expansion strategy is, that one new node is added for each simulated game [Cha10].

The simulation step selects the random actions that are taken. This actions can be random or pseudo-random. Pseudo-random actions are chosen according to a simulation strategy. Pure random actions are often unrealistic and weak. If the simulation strategy is too deterministic, the simulations get biased and the level of the Monte-Carlo program decreases as well [Cha10]. Therefore, the simulation strategy has to find a good balance
2 Background and Related Work

![Monte-Carlo Tree Search Diagram](image)

**Figure 2.1:** The four steps of Monte-Carlo Tree Search [CBSS08]: Selection, Expansion, Simulation and Back-Propagation

between random and more deterministic actions. According to the chosen strategy, the actions are taken until the end of the game.

In the back-propagation step, the result from the simulation is propagated back to the node, where the simulation started. To compute the value of this node when more than one simulation starts from the node, a back-propagation strategy is used. Two common strategies are to use either the average result from all simulations or the maximum. There are also other, more complex strategies, but not evaluated further in this work.

Typically, many Monte-Carlo steps need to be performed, before the action that is actually taken can be selected. Afterwards, the best child node is selected. In addition to the value, the visit count of the nodes is often also considered when selecting the best child. If that is the case, a more robust child is selected.

In huge domains, holding a complete tree with all possible outcomes in memory is impossible. With Monte-Carlo tree search, it is not necessary to construct such a tree. That is why especially for huge domains with high branching factor like computer GO, Monte-Carlo tree search produces good results [Cha10]. Another advantage is that no positional evaluation function is needed, so the same algorithm can be used on different domains.

2.2 Regression

Regression models the relationship between an input vector $x$ and an output value $y$. Therefore, a vector $\beta$ is calculated by which the output can be estimated for any input. For given data $D = \{(x_i, y_i)\}_{i=1}^{n}$ with $x_i \in \mathbb{R}^d$, where $d$ is the dimension, and $y_i \in \mathbb{R}$,
2.2 Regression

Regression finds a function \( f \) for an input vector \( x \in \mathbb{R}^d \) with \( f(x_i) \approx y_i \). The parameter \( \beta \) is calculated for this, so that:

\[
f(x) = \beta_0 + \sum_{j=1}^{d} \beta_j x_j, \beta \in \mathbb{R}^{d+1}
\]

To find the optimal \( \beta \), the least square cost of the training data \( D \) is considered. The least square cost squares the error, the function makes for \( D \):

\[
L(\beta) = \sum_{i=1}^{n} (f(x_i) - y_i)^2
\]

The best choice for \( \beta \) is, when the least square cost is minimal. That is for

\[
\beta = (X^T X)^{-1} X^T y
\]

\( X \) contains all \( x_i \) from \( D \):

\[
X = \begin{pmatrix}
   x_1^T \\
   x_2^T \\
   \vdots \\
   x_n^T
\end{pmatrix}
\]

Instead of just using \( x \) for regression, a feature \( \phi(x) \) can be created and used as input. Such features can be extremely powerful, especially when they are non-linear for example polynomial or radial basis functions. An example of a polynomial feature is:

\[
\phi(x) = \begin{pmatrix}
   1 \\
   x \\
   x^2 \\
   x^3
\end{pmatrix}
\]

The \( X \) used to calculate \( \beta \) then contains all features:

\[
X = \begin{pmatrix}
   \phi(x_1)^T \\
   \phi(x_2)^T \\
   \vdots \\
   \phi(x_n)^T
\end{pmatrix}
\]

Now a closer look is taken on Ridge Regression and Relational Regression. Ridge Regression tries to improve the model function \( f \) by regularization. In this work, relational states are used as input. To be able to use regression on them, Relational Regression is needed.
2 Background and Related Work

2.2.1 Ridge Regression

To use regression, sample data are taken from an unknown function. With this data, a model estimate is created. If other data from the same function are sampled, regression returns a different model estimate. Ridge Regression tries to regularize the result to get a better solution. Under the assumption that the data are noisy with variance $\text{Var}\{y\} = \sigma^2 I_n$ and $\hat{\beta} = (X^T X)^{-1} X^T y$, the variance of $\beta$ is:

$$\text{Var}\{\hat{\beta}\} = (X^T X)^{-1} \sigma^2$$

In most cases, $\sigma$ is unknown, but it can be estimated based on the deviation from the learned model:

$$\hat{\sigma} = \frac{1}{n - d - 1} \sum_{i=1}^{n} (f(x_i) - y_i)^2$$

Ridge Regression adds a regularization to the least square cost:

$$L_{\text{ridge}}(\beta) = \sum_{i=1}^{n} (y_i - \phi(x_i)^T \beta)^2 + \lambda \sum_{j=2}^{k} \beta_j^2$$

The new optimum is now:

$$\hat{\beta}_{\text{ridge}} = (X^T X + \lambda I)^{-1} X^T y \quad (I = I_k)$$

As $\beta_1$ is usually not regularized, $I_{1,1} = 0$ then. The estimator variance also changes and is now:

$$\text{Var}\{\hat{\beta}\} = (X^T X + \lambda I)^{-1} \sigma^2$$

Next, the optimal $\lambda$ has to be chosen. When $\lambda$ is set $\lambda = 0$, $\beta$ is calculated the same way as in standard regression. In that case, the training data error is lowest. To choose the optimal $\lambda$, the generalization error has to be estimated. For this estimation, test data $D$ are needed. One possibility is the k-fold cross validation: At first, the data are partitioned in k equal sized subsets $(D_1...D_k)$. For each of these subsets $\hat{\beta}_i$ is computed on $D\setminus D_i$. The error is then computed on validation data $D_i$:

$$l_i = L_{\text{ls}}(\hat{\beta}_i, D_i)$$

Last, the mean squared error is computed:

$$\hat{l} = \frac{1}{k \sum_i l_i}$$

The $\lambda$ for which $\hat{l}$ is smallest, is chosen.
2.2.2 Relational Regression

Regression gets an $x \in \mathbb{R}^d$ as input. In relational regression, the regression is over a relational state $s$, so the input is $s$. Similar to standard regression, a feature $\phi(s)$ can be constructed. One possibility for such a feature is a binary feature like:

$$\phi(s) = \begin{cases} [s \in B_1] \\ [s \in B_2] \\ \vdots \\ [s \in B_m] \end{cases}$$

$B_i$ are properties that some states have and some do not have. Then $f(s)$ can simply be calculated as:

$$f(s) = \phi(s)^T \beta$$

2.3 TILDE

Top-down induction of decision trees (TDIDT) is the best known and most successful machine learning technique and used to solve numerous practical problems [BD98]. It is not yet so popular within first-order learning because of discrepancies between the clausal representation employed within inductive logic programming and the structure underlying a decision tree. Top-down induction of first-order logical decision trees (TILDE) uses TDIDT on first-order logic trees. For this, first-order logic decision trees are translated into logic programs. Then, attribute-value learners are upgraded to the first-order logic context. The learner learns from interpretations. Learning from interpretations starts from a given set of classes $C$, a set of classified examples $E$ and a background theory $B$. Now the task is to find a hypothesis $H$ (in this case a Prolog program), such that for all $e \in E$:

$$H \land e \land B = c \quad \text{and} \quad H \land e \land B \not= c'$$

where $c$ is the class of the example $e$ and $c' \in C - \{c\}$. First-order logical decision trees (FOLDTs) are binary decision trees. Its nodes contain a conjunction of literals. Different nodes may share variables but a variable that is introduced in a node must not occur in the right branch of that node. A tree $T$ is either a leaf with class $k$ ($T=$leaf($k$)) or an internal node with conjunction $c$, left branch $l$ and right branch $r$ ($T=$inode($c,l,r$)). For classification on an example $e$ it has to be checked whether a query $C$ succeeds in $e \land B$. When the example is sorted to the left, $C$ is updated by adding the conjunction. [BD98]
To translate FOLDTs into logic programs, a newly invented nullary predicate as well as a query is associated with each internal node. The query can use all predicates that were defined in higher nodes. For leaves only a query is associated. This associated query of a node succeeds for an example if and only if that node is encountered during classification. That means if a query associated with a leaf succeeds, the leaf indicates the class of the example. For the right subtree, the algorithm adds the negation of the invented predicate $p_i$ and not the negation of the conjunction itself to a query. [BD98]

TILDE works like C4.5 [Qui93] for binary attributes. The basic idea behind C4.5 is to create a decision tree where each node splits a training data sample with the attribute that leads to the highest information gain. As soon as all cases belong to the same class, a leaf is created identifying this class. [Qui93]

TILDE employs a classical refinement operator under $\theta$-subsumption to compute the set of test that are considered at a node. An operator $\rho$ maps clauses onto sets of clauses such that: For any clause $c$ and $\forall c' \in \rho(p)$, $c$ $\theta$-subsumes $c'$. A clause $c_1$ $\theta$-subsumes another clause $c_2$ if and only if there is a variable substitution $\theta$ such that $c_1 \theta \subseteq c_2$. To refine a node with associated query $\leftarrow Q$, TILDE computes $\rho(\leftarrow Q)$ and chooses that query $\leftarrow Q'$ that results in the best split. The conjunction put in the node consists of the literals that have been added to $Q$ in order to produce $Q' = (Q' \cdot Q)$. A set of specifications is provided, indicating which conjunctions can be added to a query, the maximal number of times it can be added and the modes and types of the variables in it. [BD98]

2.4 IRL

Inverse reinforcement learning (IRL) tries to find an explanation of the behavior in order to learn complex skills. Thereby it can handle changes in the world dynamics. Imitation learning, on the other hand, is learning directly the behavior. [MPG+15] introduces an IRL algorithm for relational domains, called Cascaded Supervised IRL (CSI). The task of CSI is to find an operator that gets a reward function out of demonstrations. For this, operators are defined that go from demonstrations to an optimal quality function and from an optimal quality function to the corresponding reward function [MPG+15]. For a given dataset $D = (s_k, a_k)$, a classification algorithm is used to get a decision rule that uses the states $s_k$ as inputs and the actions $a_k$ as labels. For example a Score Based Classification algorithm can be used for this. Such an algorithm outputs a score function $q_c = \mathbb{R}^{S \times A}$ from which a decision rule can be inferred by taking the action with the highest score for each state:

$$\forall s \in S, \pi(s) \in \arg\max_{a \in A} q_c(s, a)$$
The reward can then be computed in the following way [MPG+15]:

$$R_c = q_c(s, a) - \gamma \sum_{s \in S} P(s'|s, a) \max_{b \in A} q_c(s', b)$$

When the world dynamics $P$ are provided, $R_c$ can be computed exactly. If $P$ is not provided, $R_c$ is estimated by regression. For this, a regression dataset is constructed from a mix of non-expert and expert samples. The output is an estimation of the target reward $R_c$. For regression, the data are projected onto a hypothesis space. As there are many optimal candidates for the score based classification step among which the one that will be projected with the smallest error in the hypothesis space during regression is chosen. The quality of the regression step is further improved by a reward shaping step. In this step the optimal policy is not changed anymore, only the reward shape is modified. Because $\forall R \in \mathbb{R}^{S \times A}, \forall t \in \mathbb{R}^S, q_t(s, a) = Q^*_R(s, a) + t(s)$ and $q_t(s, a) = Q^*_R(s, a)$ share the same optimal policies, a function $t \in \mathbb{R}^S$ has to be found such that the expected error of the regression step is minimal [MPG+15]. To make CSI non-parametric in the end, a Relational Regression method like TILDE can be used.
3 Program Setup

3.1 Data Representation

The relational environment that is used to construct and evaluate the features represents a robot setup. The first order logic world consists of agents that represent robot arms and objects. There are three agents A1, A2 and A3 and three objects Handle, Long1 and Long2. For each agent or object some predicates can hold. The predicate "busy" means the agent or object is involved in an ongoing activity. For example, if A1 is involved in an activity, then (busy A1) would be true. Another predicate is "free". Free always holds when the agent's robot hand is free. The predicate "held" is for objects that are held by any agent. When an agent X holds an object Y, (grasped X Y) holds. (grasped A3 Long2) means, that the agent A3 has the object Long2 in his hand. In that case, (held Long2) also holds, because Long2 is held by A3. An agent can also grasp a screw. This is indicated by the predicate "hasScrew". Objects can be fixed together. If the objects Handle and Long1 are fixed together, the predicates (fixed Handle Long1) and (fixed Long1 Handle) hold. An agent can perform an action that changes the state. There are four different actions: grasping, graspingScrew, releasing and fixing. Rules describe the impact an action has on the current state.

Figure 3.1 shows the rule for releasing an object. There is one agent X and one object Y involved in this action. The first braces (second line of activateReleasing) surround the preconditions. The action (releasing X Y) can only be taken if (releasing X Y) is not already part of the state. Another precondition is that (grasped X Y) must hold. An agent can only release an object if it has grasped it before. The third and fourth precondition is that neither (busy X) nor (busy Y) holds. Any agent or object can only perform one action a time. When all preconditions hold, the action can be taken. The second braces enclose the postcondition. When the action is taken, the facts (releasing X Y) = 1.0, (busy X) and (busy Y) are added to the state. (releasing X Y) = 1.0 means, that the action (releasing X Y) takes 1.0 time units. After that time, (Terminate releasing X Y) is invoked. The rule for Terminate releasing also has preconditions (only (Terminate releasing X Y)) and postconditions. The postconditions now describe the changes in the world after the action was performed. In this case the facts (releasing X Y) and (grasped X Y) no longer hold. Agent X is now free (free X) and object Y no longer held
3 Program Setup

Figure 3.1: Rule releasing: Contains the preconditions and postconditions for the action (releasing X Y)

(held Y)!. Furthermore both the agent X and the object Y are no longer busy. There are such rules for every action. These rules are quite evident. Only the action grasp can fail with a probability of 10%. In that case, after the action is terminated, the precondition holds again. The action WAIT waits until the first action terminates and executes the corresponding termination rule.

The world starts in the following start state:

\[
\text{state}\{\text{(agent A1), (free A1), (agent A2), (free A2), (agent A3), (free A3),}
\]

\[
\text{(object Handle), (object Long1), (object Long2)}\}
\]

Terminal states are all states that contain the following facts:

\[
\text{(fixed Handle Long1), (fixed Handle Long2)}\]

3.2 Planner Example

The project "Planner Example" is a Monte-Carlo planner for relational worlds. For each rollout, the planner starts in the start state and randomly chooses actions until a terminal state is reached. For every such rollout, the first action and the return are stored. The number of rollouts, the planner performs, is selectable. The action with the best return is chosen and executed. Afterwards, the same process starts from the following state. With the help of the planner, it is possible to create a data file. For every step, the state and every possible action with the best return that the planner found for this action, is written into the file.

Figure 3.2 shows the first part of a data file that was created by the planner. It shows the start state and the 13 possible actions that can be taken from it with the corresponding
3.2 Planner Example

Figure 3.2: Part of data file that shows the start state with all possible actions and the corresponding returns

```
{ state [(agent A1), (free A1), (agent A2), (free A2), (agent A3), (free A3),
         (object Handle), (object Long1), (object Long2), ]

action (activate_grasping A1 Handle), return=-14.4299
action (activate_grasping A1 Long1), return=-12.4718
action (activate_grasping A1 Long2), return=-11.7795
action (activate_grasping A2 Handle), return=-12.3057
action (activate_grasping A2 Long1), return=-12.2445
action (activate_grasping A2 Long2), return=-12.443
action (activate_grasping A3 Handle), return=-12.2597
action (activate_grasping A3 Long1), return=-12.5206
action (activate_grasping A3 Long2), return=-12.3505
action (activate_graspingScrew A1), return=-12.3122
action (activate_graspingScrew A2), return=-12.290
action (activate_graspingScrew A3), return=-12.2604 }
```

return. The first action stands for the WAIT action. For every other action, the name is shown. Based on this, the best action is selected. In this case, the action activate_grasping A1 Long2 would be chosen. With −11.2187 it has the best return value. The features are later created on the basis of this data.
4 Creation and Evaluation of Features

To be able to execute regression on a relational domain, features have to be created that map the relational state to a real value. In this chapter, different options to create such features are introduced. Afterwards, the evaluation of the different features is presented.

All features are based on facts. Facts occur in relational states. For example the start state \{((agent A1), (free A1), (agent A2), (free A2), (agent A3), (free A3), (object Handle), (object Long1), (object Long2) )\} consists of nine facts. The first one is "(agent A1)". The next ones are "(free A1)", "(agent A2)" and so on. A factlist is created, based on which, the features can be constructed.

4.1 Factlist

The factlist contains all facts that occur in a training dataset. To create this list, all states from training data are considered. Every fact that occurs in a state is added to the list after checking that it is not already part of the list.

Figure 4.1 shows the beginning of the factlist. At first the facts from the start state are listed. Next, there are the facts that occur in the second state, but not in the start state. These facts depend on the dataset. Some entries represent the last action that was taken. For example: decision(activate_grasping A3 Long1). In the creation of the dataset, the action activate_grasping A3 Long1 was first taken. These actions are also part of the factlist and therefore used in the regression. Some facts also have a duration. For example: (grasping A3 Long1)=5 indicates, that the fact (grasping A3 Long1) holds for 5 time units. The durations are listed in the factlist but are not further considered. Every fact is only stored once in the list. (grasping A3 Long1)=5 and (grasping A3 Long1)=4 are considered the same fact. If both occur in the training data, one is regarded as duplicate and not stored.
4 Creation and Evaluation of Features

4.2 First Feature: F

The first feature that was created, is a vector that considers all facts. Each vector row represents one fact. When the fact is true in a given state, the corresponding row of the feature is 1. When the fact does not occur in this state, the feature row is 0.

For example the start state: The first fact of the factlist is (agent A1). This occurs in the start state, so the first row of the feature is 1. With the facts (free A1), (agent A2), (free A2), (agent A3), (free A3), (object Handle), (object Long1), (object Long2), decision(activate_grasping A3 Long1), (grasping A3 Long1)=5 (busy A3), (busy Long1), decision(activate_grasping A2 Handle), (grasping A2 Handle)=5 (busy A2), (busy Handle),

\[ \phi(\text{startstate}) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \ldots \end{pmatrix}^T \]

In the following, this feature will be called "F".

4.3 Second Feature: FF^2

In the next step, a second feature is created. The second feature apart from facts also considers combinations of two facts. Every fact from the fact list is combined with every
other fact. The resulting list of combinations looks like this:
{(agent A1), (agent A1)}
{(agent A1), (free A1)}
{(agent A1), (agent A2)}
...
{(free A3), (held Handle)}
...
A row of a feature is 1 when both facts occur in the state. If one fact does not occur or both facts do not occur in the state, the feature row is 0. The actual feature is a concatenation of the first feature with only facts (F) and the two-fact combinations (F^2).

4.4 Third Feature: FA

Another possibility is to use features that also consider actions. Therefore, the next step is to create a list of actions. Similar to the list of facts, the list of actions contains all actions that occur in the training dataset. That comprises all possible actions from any state in the dataset. Now every state-action pair has its own feature. The first part of the feature is like the first feature F. The second part of the feature represents the actions. Every row in the feature vector stands for one action. If the action is taken in the state-action pair, the corresponding row contains a 1. If it is not taken, there is a 0. This feature will be called "FA" in the following.

4.5 Fourth Feature: FF^2A

The last feature is a combination of all features that were already created. Like the feature FA, it also considers state-action pairs. At first it considers all the facts like the first feature (F). Then the combinations of two facts like the second feature (F^2). Last, it also considers the actions. Like the third feature it uses the action list to create the last part of the feature (A). This feature is the longest and most complex feature and is referred to as FF^2A.

4.6 Regression

After the features were created, regression is used to calculate a heuristic for the estimated return. The training data for the regression are given by the output of
the Monte-Carlo planner. For all features, $\beta$ is calculated by Ridge Regression: 

$$\beta = (X^T X + \lambda I)^{-1} X^T y$$

The robot setup can be viewed as a Markov Decision Process. With a Markov Decision Process the state value function (V-function) can be calculated:

$$V(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) \cdot V(s')$$

where $V(s)$ is the value of state $s$, $\pi$ is the agents policy, $R(s, \pi(s))$ the reward for the state-action pair $(s, \pi(s))$, $\gamma$ the discount factor and $P(s'|s, \pi(s))$ the transition probability. Additionally there is a state-action value function (Q-function):

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \cdot Q(s',\pi(s'))$$

For the planning data that are used, the Q-function can be approximated as:

$$\hat{Q}(s,a) \approx \max_i \hat{R}_i(s,a)$$

where $\hat{R}_i$ is the return for the state-action pair $(s,a)$ calculated by the $i$-th Monte-Carlo rollout where the action $a$ is taken first. The V-function can then be estimated by:

$$\hat{V}(s) \approx \max_a \hat{Q}(s,a)$$

### 4.6.1 Features F and FF$^2$

The data for the regression have the following form: $D = \{(s_i, y_i)\}_{i=1}^{n}$ where $s_i$ is the state and $y_i$ the return. In the training dataset, for every state, all possible actions are listed with the best return, the planner found. The return for the state-action pair corresponds to the state-action value function (Q-function). For regression the return for a state is needed, that corresponds to the state value function (V-function). In order to get this value function, the maximum return of all possible actions for the state $s_i$ is taken as $y_i$. The matrix $X$ looks like this:

$$X = \begin{pmatrix}
\phi(s_1)^T \\
\phi(s_2)^T \\
\vdots \\
\phi(s_n)^T
\end{pmatrix}$$

$\phi$ can be either the feature F or the feature FF$^2$. The vector $y$ contains the return of the best action for each state. Ridge Regression is used to calculate $\beta$. For the evaluation, seven different values for $\lambda$ are used.
4.7 Evaluation

4.6.2 Features FA and FF$^2$A

Now, the data for regression have the following form: $D = \{(sa_i, y_i)\}_{i=1}^n$ where $sa_i$ is a state-action pair and $y_i$ the return. In the training dataset, the return is given for each state-action pair (Q-function). This time, the return that is given in the data, can be used as $y_i$ without any change. The matrix $X$ now contains the features for all state-action pairs and has considerably more rows than $X$ for the features $F$ and FF$^2$, because there are more state-action pairs than states.

$$X = \begin{pmatrix}
\phi(sa_1)^T \\
\phi(sa_2)^T \\
\vdots \\
\phi(sa_n)^T
\end{pmatrix}$$

In this case, $\phi$ is either the feature FA or the feature FF$^2$A. Now the vector $y$ also contains the returns for each state-action pair. Like for the other features, Ridge Regression is used with different $\lambda$s.

4.7 Evaluation

In order to evaluate the regression, two different datasets are created by the Monte-Carlo planner. One of these datasets is used as a training dataset to calculate $\beta$. The other one is a test dataset to test the estimated $V$- or $Q$-function on different data.

4.7.1 $\chi$

To measure the quality of the regression, a measurement is needed. At first, the mean is calculated:

$$\mu = \frac{1}{n} \sum_{i} y_i$$

Now, with the help of the mean, $\sigma^2$ can be computed:

$$\sigma^2 = \frac{1}{n-1} \sum_{i} (y_i - \mu)^2$$

Furthermore, the Least-Square cost is calculated and divided by $n$ ($f(s_i)$ is the model function evaluated for $s_i$):

$$L = \frac{1}{n} \sum_{i} (y_i - f(s_i))^2$$
4 Creation and Evaluation of Features

In the next step, $\chi$ can be calculated:

$$\chi = \frac{\sqrt{L}}{\sqrt{\sigma^2}}$$

The minimum value, $\chi$ can take is 0. That happens when $L$ is 0. When $L = \sigma^2$, $\chi$ would be 1. So, $\chi = 1$ means that the model function is as good as the function $f(s_i) = \mu$, because $L = \sigma^2$ then, too. Consequently, $\chi$ should be between 0 and 1 and as close to 0 as possible.

4.7.2 Results

At first, two datasets were created by the Monte-Carlo planner. In both cases the planner executed 100 rollouts per step and performed 100 steps. So both datasets consist of 100 states. The first dataset is used as a training dataset, the second one as test dataset. The factlist, X and y are calculated for the training data. In the next step, $\beta$ is calculated by regression with X and y. Then $\chi$ can be computed: $\chi$ is always calculated for the training data ($\chi_{train}$) and the test data ($\chi_{test}$). $\sigma^2$ is only calculated once for the training data. For $\chi_{test}$ the same $\sigma^2$ is used. For the test data, only L is calculated newly:

$$\chi_{train} = \frac{\sqrt{L_{train}}}{\sqrt{\sigma^2}}$$

$$\chi_{test} = \frac{\sqrt{L_{test}}}{\sqrt{\sigma^2}}$$

The diagram in Figure 4.2 shows $\chi_{test}$ for all features with different $\lambda$s.

As the diagram shows, $\chi_{train}$ is very close to 0 for all settings. The feature FF$^2$ (green line) has the lowest $\chi_{train}$, where $\chi \approx 0$. This feature is also the longest feature with a feature length of 3540. As expected, all features are best for the smallest $\lambda$ ($\lambda = 10^{-10}$), because the least-square cost is minimal for $\lambda = 0$. FA (red line) and FF$^2$A (yellow line) ($\chi \approx 0.025$) are a bit worse than F and FF$^2$, because there are about 300 state-action pairs in the training data and only 100 states. Sometimes there are only 10 rollouts per state-action pair, so the chance, that the planner does not find the optimal return is higher. Noticeable is also that the bigger the feature, the less changes through the regularization factor $\lambda$. F (blue line) only has a length of 59 and for $\lambda \geq 1$ it achieves the worst results. For $\lambda = 0.1$ or smaller, it has the second best results though ($\chi \approx 0.01$). The longest feature (FF$^2$A) with a length of 3562 has about the same $\chi$ for every $\lambda$.

Figure 4.3 shows the results for the test data.
4.7 Evaluation

Figure 4.2: Results: $\chi_{train}$ for all four features. The feature FF$^2$ (green) shows the best results, followed by F (blue). The features FA (red) and FF$^2$A (yellow) are a bit worse, because the dataset is different for them. The for higher $\lambda$s the small features get worse fast, whereas the longer features do not alter much.

Obviously, $\chi_{test}$ is higher than $\chi_{train}$, but it is still close to 0. Here, $\chi$ is the best for $\lambda \approx 10^{-2}$ except for the feature FF$^2$A. The shorter features F and FA ($\chi \approx 0.3$ for F and $\chi \approx 0.5$ for FA) have much better results than the longer ones FF$^2$ and FF$^2$A. This seems surprising at first. The longer features should lead to a more accurate heuristic. For the training data, the feature FF$^2$ had a $\chi$ of almost 0 ($\approx 2 \cdot 10^{-13}$). For the test data, $\chi$ is over 0.25. The reason for this is, that FF$^2$ and FF$^2$A consider two-fact combinations. While (nearly) all facts that occur in the test data, have also occurred in the training data, the combinations of two facts that occur in the training data might be different from those occurring in the test data. The $\beta$ that is calculated by regression, is 0 at all digits that stand for combinations that do not occur in training data. This problem can be solved by using a bigger training dataset.

For the second test, the training data is created that consist of 500 states. This time 1000 rollouts were executed per step to get more reliable data, especially for the state-action pairs. Sometimes there are up to 13 possible actions in a state. When 1000 rollouts are executed per step there are still about 80 rollouts for every action in that state, so the
4 Creation and Evaluation of Features

Figure 4.3: Results: $\chi_{test}$ for all four features. Both shorter features F (blue) and FA (red) get better results than the longer ones FF$^2$ (green) and FF$^2$A (yellow). They seem to be better suited for short datasets. The features that do not consider actions still get slightly better results than the ones that consider actions.

returns are more reliable than with only 8 (or even less) rollouts per action. The test data consist of 100 states again, but this time also with 1000 rollouts per step.

Figure 4.4 shows the evaluation results for the new training data.

For all features, $\chi_{train}$ is slightly higher than for the first data. Again, FF$^2$ shows the best results ($\chi \approx 0.005$) and F is a bit worse ($\chi \approx 0.01$). FA and FF$^2$A have similar $\chi$s for $\lambda \leq 0.1$ ($\approx 0.025$). Altogether the results are very similar to those in the first test. $\chi$ is a bit higher, because there are more data and the heuristic is not able to match all of them anymore, as the feature length does not increase that much. For higher $\lambda$s the same effect as in the first test eventuates: For the smaller features, $\chi$ gets higher very fast, especially for F. Whereas for the longer features, $\chi$ is low even for higher $\lambda$s. For FF$^2$A $\chi$ does nearly stay the same for higher $\lambda$s. Whereas $\chi_{train}$ is similar compared to the one for the first data, $\chi_{test}$ differs more. That can be seen in the next figure.

Figure 4.5 shows the results for the test data.
4.7 Evaluation

Figure 4.4: Results: $\chi_{train}$ for all four features for bigger training dataset. For the new training dataset, FF$^2$ (green) still shows the best results followed by F (blue). The features FA (red) and FF$^2$A (yellow) have similar results.

$\chi_{test}$ for F and FA does not change significantly, compared to the first test. It only gets slightly better. For F it improves from about 0.03 to 0.02, for FA it improves from 0.05 to 0.03. As expected, FF$^2$ and FF$^2$A are much better now, because of the bigger training dataset. FF$^2$ improves form about 0.25 to 0.02 and FF$^2$A from about 0.2 to less than 0.04. For both features FF$^2$ and FF$^2$A, the regularization factor $\lambda$ is very important. The results are not good without regularization ($\lambda = 10^{-10}$). With regularization, they get better very fast. For every four features, between $\lambda = 10^{-4}$ and $\lambda = 0.1$, all $\chi$s are very similar. In this range, the features are also very similar among each other.

The features all have a higher length than in the first test. As more training data are used, more facts are in the factlist. F has now a length of 81 instead of 59 in the first test. Both the features FF$^2$ and FF$^2$A have a length of more than 6600. Because of that, the regression takes a very long time. $\beta$ also has a length of over 6600 then, so applying the estimated Q- or V-function also takes its time. The added value for this features is very low or even nonexistent compared to the other two features. The results of F are very close to those of FF$^2$ and the ones from FA are even better than those of FF$^2$A. For this reason, the features FF$^2$ and FF$^2$A are less suitable than F and FA for this application.
Figure 4.5: Results: $\chi_{test}$ for all four features for bigger training dataset. F (blue) and $\text{FF}^2$ (green) have a similar $\chi_{test}$. The results for FA (red) and $\text{FF}^2\text{A}$ (yellow) are slightly worse, but still very close to 0. For the longer features $\text{FF}^2$ and $\text{FF}^2\text{A}$ this is a huge improvement compared to the results for the old training dataset. These longer features need a higher regularization factor $\lambda$ than the shorter features.
5 Integration in the Monte-Carlo Planner

After getting promising results for estimating the return by regression, it can be used by an actual application. This chapter describes how the regression was integrated into the Monte-Carlo planner. For this, the same planner is used, that has already created the datasets for regression. Before the planning starts, training data are used to calculate the fact list and the action list. With the help of the fact list, the features for the states of the training data are created. Only F and FA are used for this as the features FF and FF_A are much longer and it takes a long time to use regression for them. The goal of this work is to speed up Monte-Carlo planning. Because these features take so much time, the planning would actually become slower. Moreover, F and FA achieve similar results than FF and FF_A, so it would not be worth to take the effort for the longer features. All created features for the states in training data are stored in the matrix X. Then the vector y is created, that contains the returns for all states or state-action pairs. In the next step, Ridge Regression is used to calculate $\beta$. Afterwards, the actual planning starts. Figure 5.1 shows how the modified planner works now and which steps it performs.

At the beginning, the planner is in the start state. The Monte-Carlo planner starts the rollouts from there. Now, the planner is modified so that the return that is calculated by regression replaces part of these rollouts. Before the regression is used, a few Monte-Carlo steps are performed. For the evaluation either 5 or 10 steps were performed. After these steps are performed, the feature of the state, in which the planner is after the Monte-Carlo steps, is calculated. The next step depends on the feature that is used. When F is used, only the state is needed to calculate the return. The function that was created by regression now is the estimated V-function. It is then used to estimate the return of that state. This return is multiplied by a discount factor, because the state is a few steps away. The result is added to the return from the Monte-Carlo steps. An exception is the case that a terminal state is already reached during the execution of the Monte-Carlo steps. In that case the return is only calculated by regression and the discount factor. The return of the Monte-Carlo step is discarded then. Otherwise the return would be twice as much as the reward for termination.

When FA is used, a state-action pair is needed to calculate the return. The function that was created by regression is the estimated Q-function now. In this case, for every possible action, the return is calculated by this function. For every action, the corresponding
Figure 5.1: Integration of the Regression in to Monte-Carlo Planner. At first some Monte-Carlo rollouts are performed. Then the estimated Q-function (when the feature FA is used) or V-function (when F is used) is used to calculate the expected return from there. The return is multiplied with the discount factor and added to the Monte-Carlo return. The exception with the terminal state is applied as explained for feature F. When the return is calculated for all possible actions, the estimated V-function is calculated. Therefore, the Q-function with the highest value is taken as return for the state.

Like for normal Monte-Carlo planning, many rollouts have to be performed for this. So the described procedure is repeated. For every action from the first state, the best return is stored. The most promising action is taken. That is the one with the highest return. Then the planner continues from the following state. This is repeated until the planner reaches a terminal state.
6 Evaluation

In this chapter, the modified Monte-Carlo planner that uses regression, is evaluated. Regression is integrated as described in Chapter 5. The first training dataset that is used to create the fact list, action list and features is the one that was created in Chapter 4.7. It comprises 500 states and was created by the Monte-Carlo planner with 1000 rollouts per step. For this dataset, the lowest $\chi^2_{test}$ was reached for $\lambda = 10^{-10}$. Therefore this $\lambda$ is used for Ridge Regression. The planner always executes the action with the highest return until either a terminal state or a dead-end state is reached. In the used robot setup, dead-end states are all states where more than one agent holds a screw. This is because there is no action that can be executed when two or more agents have a screw except for "WAIT". The "WAIT" action does not change the state anymore when all previous actions are done. Moreover, there is no action like "releaseScrew" to get out of this state again.

6.1 Feature F

6.1.1 First Test

At first, the planner using feature F for regression was tested. Figure 6.1 shows the portion of times how often the planner was successful and a terminal state was reached. For this experiment, every setting of the planner was executed 100 times. The results are compared to those of the standard Monte-Carlo planner that was also executed 100 times.

The diagram shows the standard Monte-Carlo planner (green bar), the modified one that executes 10 Monte-Carlo steps at first (blue bar) and the one that executes 5 Monte-Carlo steps before regression (red bar). The standard planner and the modified one with 10 Monte-Carlo steps before regression show a similar success rate. For 10 rollouts per step, they both reach the terminal state in a bit over 70% of times. This seems quite good considered that there are states with more than 10 possible actions. So in this states there are action for which no rollout is executed at all. For 50 and 100 rollouts per step, the success rate even is over 90%. It is also noticeable, that it does not
6 Evaluation

Figure 6.1: Percentage of successful runs for standard Monte-Carlo planner and Monte-Carlo planner that uses regression with feature F. The standard planner (green) and the planner that uses regression after 10 rollout steps show similar results. The more rollouts are executed per step, the better the results. The planner that uses regression after 5 rollout steps rarely reaches the terminal state regardless the number of rollouts per step.

make a big difference whether there are 50 or 100 rollouts. The percentage of successful runs is only a little lower for 50 rollouts. However, when the Monte-Carlo planner only executes 5 steps before using regression, the results worsen drastically. A success rate of under 40% is considered as not acceptable for such a planner.

Not only the success rate of the planner is interesting, but also how many steps it takes to get to a terminal state. Figure 6.2 shows how many steps the planner needs to reach a terminal state. For this diagram, only successful runs are taken into account.

The diagram shows the mean and the standard deviation of the number of steps that the planner has to take to reach a terminal state. The optimum the planner can achieve for this task is 13 steps. The standard Monte-Carlo planner (green bar) needs more than 30 steps on average when executing 10 rollouts per step. The planner that uses regression after 10 rollout steps (blue bar) is slightly better. When the planner executes 50 rollouts every step, the results for the improved planner with 10 rollout steps get much better. For 100 rollouts per step, these results do not significantly change anymore. About 17
Figure 6.2: Steps until terminal state is reached: Monte-Carlo planner versus Monte-Carlo planner that uses regression with feature F. On average, the standard Monte-Carlo planner (green) needs the most steps. The modified planner with 10 rollout steps shows better results, especially for the 50 rollout setting. The planner that uses regression after 5 rollout steps (red) shows the best results and only needs very few steps to reach a terminal state, regardless of the number of rollouts per step.

Steps until the terminal state is reached is already very close to the optimum of 13 steps. The standard Monte-Carlo planner still needs over 20 steps for the 100 rollout setting. What all settings have in common is, that the standard deviation is highest for 10 rollouts per step. For the standard Monte-Carlo planner and the one with 10 rollout steps before regression, it is over 13 steps. Mostly there are two or less rollouts for every action, therefore the optimal return is only found rarely. Whether an expedient action is taken is rather incidental. For the setting with 100 rollouts the standard deviation decreases to about 9 steps for the standard planner and 3 steps for the modified planner with 10 rollout steps. What is also conspicuous is that the planner with 5 Monte-Carlo steps before regression (red bar) has the best results as it needs the least number of average steps. Moreover it has a very low standard deviation ranging from about 5 steps to about 2 steps depending on the setting. In most cases, the terminal state is never reached, because the planner gets into a dead-end state. However, when the terminal state is reached, this happens really fast. An explanation for this phenomenon could be that the
training data are created by a nearly optimal planner (Monte-Carlo planner with 1000 rollouts). This planner never gets into a dead-end state. Because of this, the training data do not contain such a state, but only states from which a terminal state can still be reached. The return for a dead-end state, that is calculated by the heuristic, cannot be accurate, because the training data does not contain such states. When 10 Monte-Carlo steps are executed before regression, the Monte-Carlo planning compensates this effect. It would be desirable to also get good results for the planner that uses regression after 5 rollout steps, because that would further speed up the planning process. The creation of new training data, that also contain dead-end states, could solve this problem. Therefore new training data were created and the test was repeated with this new data.

6.1.2 Creation of New Training Data

To create new training data, the Monte-Carlo planner is manipulated so that dead-end states can be reached. For this, the planner does not always select the best action, but a random action. To get precise return data, the planner still uses 1000 rollouts per step. To create the training data, the planner with this settings is executed five times. When a dead-end state is reached the planner stops and writes the data into the datafile. Afterwards, the standard Monte-Carlo planner is also executed five times. Because of this, both terminal states and dead-end states are part of the training data. The resulting training dataset consists of 207 states.

Before the planner uses this data, it is tested how good the regression works for this. For this, $\chi_{\text{train}}$ and $\chi_{\text{test}}$ are calculated. This is only done for the features F and FA. FF and FF$^2$A are not used because they are not efficient as shown in chapter 4.7. As test dataset, the dataset with 100 states that was created with 1000 rollouts per step from chapter 4.7 is used. Figure 6.3 shows the results for $\chi_{\text{train}}$ and $\chi_{\text{test}}$.

The diagram shows that for small $\lambda$ (< $10^{-1}$), the features F (blue lines) and FA (red lines) achieve similar results. Both $\chi_{\text{train}}$ and $\chi_{\text{test}}$ are slightly better for F, but the difference is only about 0.007 in both cases. Again, for higher $\lambda$s, the results worsen fast. Compared to the old training data, the results are a bit worse. One reason is that the new training dataset does not contain as many states as the old one. Especially, part of it should improve the estimation for dead-end states. The test data do not contain dead-end states, so this effect is not tested by this experiment.

6.1.3 Second Test

To see whether the intended effect occurs, the new training data are used for the modified planner. As for the old data, it is tested how often a terminal state is reached
6.1 Feature F

Figure 6.3: $\chi_{\text{train}}$ and $\chi_{\text{test}}$ for New Training Data. $\chi_{\text{train}}$ shows better results than $\chi_{\text{test}}$. Feature F that does not consider actions is slightly better than FA that does consider actions.

and, when it is reached, how many steps the planner takes to get there. Again, the planner is executed 100 times for each setting and compared to the standard Monte-Carlo planner. The $\lambda$, for which the lowest $\chi_{\text{test}}$ was reached, is taken as a regularization factor. Here this is $\lambda = 10^{-10}$. Figure 6.4 shows the ration between the times a terminal state is reached and the times a dead-end state is reached.

The diagram shows that for 10 rollouts per step, both Monte-Carlo planners that use regression are significantly better than the standard Monte-Carlo planner without regression. Whether there are 5 or 10 rollout steps executed before regression, does not make a big difference. Whereas the standard Monte-Carlo planner only reaches a terminal state in slightly over 70% of the performed planning experiments, the modified planners reached a terminal state in about 95%. For 50 and 100 rollouts per step, the modified planners always reach a terminal state. That is slightly better than the standard Monte-Carlo planner as well. For the planner that uses regression after 5 rollout steps, this is a huge improvement compared to the run with the old training data, where the terminal state was reached in less than 40% for all settings.
Figure 6.4: Percentage of successful runs for standard Monte-Carlo planner and Monte-Carlo planner that uses regression on new training data and feature F. Both planners that use regression nearly always reach a terminal state now.

Next, the number of steps, the planners need are compared. Figure 6.5 shows the average number of steps, the planners need to reach a terminal state and the standard deviation. Again, only successful runs are considered.

Like for the old training data, both planners that use regression have better results than the standard Monte-Carlo planner. Especially for 50 and 100 rollouts per step, the results for these planners are considerably better. The planner that uses regression after 5 rollout steps achieves the best results now. For 10 rollouts per step the results seem worse than for the old dataset. But now a terminal state is nearly always reached. Instead of going into a dead-end state, the planner now reaches a terminal state, but needs more steps. When this planner is executed with 50 rollouts per step, it reaches the terminal state faster than the standard Monte-Carlo planner that uses 100 rollouts per step. The standard deviation is with about 7 steps for 50 rollouts also lower than the one for the standard planner with 100 rollouts per step which is about 9 steps. That means the new training data cause the intended effect on the planner. Not only is the terminal state always reached now, but the planner reaches it also really fast.

In the experiments the planner that uses regression after 5 rollout steps and uses the new training dataset reaches the terminal state in an average of about 19 steps, for
6.1 Feature F

Figure 6.5: Steps until terminal state is reached: Monte-Carlo planner versus Monte-Carlo planner that uses regression with new data and feature F. The best results are reached by the planner that uses regression after 5 rollout steps (red). The planner that performs 10 rollout steps before using regression (blue), still shows better results than the standard Monte-Carlo planner (green).

50 rollouts per step. It never got into a dead-end state. In comparison, the standard Monte-Carlo planner needs about 22 steps when executing 100 rollouts per step. Before executing the first step, the planner that uses regression has to calculate $\beta$ to get the estimated V-function. This takes some time, but for the feature F that has a length of less than 100, it is quite fast. Then the planner only has to execute 5 steps for each rollout. The standard planner has to execute steps until a terminal state is reached. In this scenario, from the start state these are a minimum of 13 steps. As the actions are selected randomly, in most cases much more than 13 steps are executed though. Therefore, the actual planning of the planner that uses regression is faster. The planning is further sped up, because less rollouts per step have to be executed to get the same planning quality. In this example, when only 50 rollouts per step are used the modified planner is still better than the standard Monte-Carlo planner with 100 rollouts. When only half the rollouts have to be executed, the speed about doubles.
6 Evaluation

Figure 6.6: Percentage of successful runs for standard Monte-Carlo planner and Monte-Carlo planner that uses regression with new data and feature FA. The modified planners that use regression, both show better results than the standard Monte-Carlo planner here. For 50 or 100 rollouts per steps, a terminal state was always reached.

6.2 Feature FA

In addition to the planner that uses the feature F, also the one that uses the feature FA is evaluated. Like it is described in chapter 5, for every possible action, after the rollout steps, the return is estimated by regression. Again, \( \lambda = 10^{-10} \) is used, because that achieved the best results for \( \chi_{test} \). For every setting, the planner was executed 100 times. This time the experiment is only conducted for the newly created data that also contain dead-end states. Figure 6.6 shows how often a terminal state is reached out of 100 tries. The values for the standard Monte-Carlo planner are depicted for comparison again.

Again, the diagram shows the standard Monte-Carlo planner (green bar), the modified one that executes 10 Monte-Carlo steps at first (blue bar) and the one that executes 5 Monte-Carlo steps before regression (red bar). Like for the feature F, when the planner only executes 10 rollouts per step, both modified planners are significantly better than the standard Monte-Carlo planner. The planner with 5 steps before regression shows the
6.2 Feature FA

Figure 6.7: Steps until terminal state is reached: Monte-Carlo planner versus Monte-Carlo planner that uses regression with new data and feature FA. For 10 rollouts per steps, all planners need a similar average number of steps to reach a terminal state. For 50 and 100 rollouts per step, both modified planners show considerably better results than the standard planner.

best results, reaching a terminal state in 98% of the runs. The planner with 10 steps before regression still has a success rate of 90% (the standard Monte-Carlo planner only reaches 73%). For 50 and 100 rollouts per step, both modified planners always reach a terminal state in this experiment and are therefore slightly better than the standard Monte-Carlo planner. Here the same results than for the feature F are achieved.

Next, the number of steps, the planner needs to reach a terminal state is evaluated and shown in Figure 6.7. Like for feature F, only successful runs are considered for this.

Like the diagram for feature F, this diagram shows the mean and standard deviation of the number of steps that the planner has to take to reach a terminal state. When the planner executes only 10 rollouts per step, all planners show similar result. They all need an average of about 30 steps to reach a terminal state. The standard deviation is very similar too and with about 15 steps comparatively high. The planner that executes 10 Monte-Carlo steps before regression is slightly better than the other two planners. For 50 and 100 rollouts, the planners show similar results as for feature F. Now the modified planners are considerably better than the standard Monte-Carlo planner. The modified
planners need less than 20 steps on average to reach a terminal state, whereas the standard Monte-Carlo planner needs an average of more than 26 steps, for 50 rollouts per step. For 100 rollouts the modified planners only need an average of about 17 steps to reach a terminal state, what is very close to the optimum of 13. The standard Monte-Carlo planner still needs about 22 steps. The standard deviation also gets lower. For the modified planner it is only 5 steps for 50 rollouts and 3 steps for 100 rollouts per step. For comparison: The standard deviation for the standard Monte-Carlo planner is with 12 steps for 50 and 9 steps for 100 rollouts per step still much higher. Compared to the feature F, the results are slightly better for FA. For 50 and 100 rollouts per step the planner that uses the feature FA is about 1 step faster on average. The standard deviation also is about 1 step lower. On the other hand, the speed-up is not as high as for F, because regression has to be used, instead of only for the state, for every possible state-action pair from that state.
7 Summary and Outlook

This work presented an approach to apply regression in order to speed up Monte-Carlo planning. For that, four different features were created that map a relational state to a vector. With the help of Ridge Regression, the expected return for this state can be estimated. After these four features were created, they were evaluated. Training data and test data have been created for this by a very precise Monte-Carlo planner. Regression was used to create a model function from the training data. Then it was compared how good the return estimated by regression coincides with the actual return from the training and test data. The two features that were considered best, were actually integrated into a Monte-Carlo planner. It was then evaluated how well the modified Monte-Carlo planner works compared to the standard one. It turned out that, using the modified planner, less rollouts have to be performed to get the same result as for the standard Monte-Carlo planner with more rollouts. That means that not only the rollouts are faster itself, because less rollout steps have to be executed, but also less rollouts are needed per step. Both speeds the Monte-Carlo planning up.

Outlook

Further work in this area could include generalization to other domains and improving the Monte-Carlo planner. For the robot setup that was used to evaluate the Monte-Carlo planner which was modified by adding regression after a few steps, good results were reached. This was only tested for one domain. One could check whether this concept reaches results in other domains that are similar promising than for this domain. Different domains could include similar robot domains as well as different domains like games (chess, Go, ...). Another possible advancement could be to improve the Monte-Carlo planner. The planner that is considered best, executes 5 Monte-Carlo steps before using regression. If a simulation strategy is used for that instead of only taking random actions, the planner might get better. One possible simulation strategy could be the $\epsilon$-greedy strategy that is proposed in [GHSD14]. This could further speed up the planning because less rollouts might be needed. A further approach could be to test more complex ways to create features like TILDE.
A Kurzfassung


Bibliography


Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature