Learning Heuristic Functions in Classical Planning

Master Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
http://ai.cs.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Martin Wehrle, Silvan Sievers

Cedric Geissmann
cedric.geissmann@unibas.ch
10-056-026

February 12, 2015

Acknowledgments

I would like to express my thanks to Dr. Martin Wehrle and Silvan Sievers for their support and supervision of my work. I would also like to thank Prof. Dr. Malte Helmert, who allowed me to write this thesis in the field of artificial intelligence. Furthermore I would like to thank my colleagues and my family for supporting me during this time. Special thanks goes to Miro Steiner who proofread this thesis.

Abstract

The goal of classical domain-independent planning is to find a sequence of actions which lead from a given initial state to a goal state that satisfies some goal criteria. Most planning systems use heuristic search algorithms to find such a sequence of actions. A critical part of heuristic search is the heuristic function. In order to find a sequence of actions from an initial state to a goal state efficiently this heuristic function has to guide the search towards the goal. It is difficult to create such an efficient heuristic function. Arfaee et al. [1, 10] show that it is possible to improve a given heuristic function by applying machine learning techniques on a single domain in the context of heuristic search. To achieve this improvement of the heuristic function, they propose a bootstrap learning approach which subsequently improves the heuristic function.

In this thesis we will introduce a technique to learn heuristic functions that can be used in classical domain-independent planning based on the bootstrap-learning approach introduced by Arfaee et al. [1, 10]. In order to evaluate the performance of the learned heuristic functions, we have implemented a learning algorithm for the Fast Downward planning system. The experiments have shown that a learned heuristic function generally decreases the number of explored states compared to *blind-search*. The total time to solve a single problem increases because the heuristic function has to be learned before it can be applied.

Table of Contents

1 Introduction

Classical planning is the task of finding a sequence of actions that lead from a given initial state to a desired goal state. This sequence of actions is called a plan. Finding a plan can be a challenging and time consuming task for a human, because the problems have a huge amount of states and each state has many possible actions. To deal with the large state space of such a problem, domain specific solvers can be developed which use the power of a computer to solve these problems efficiently. Such a domain specific solver can be very efficient as it can take advantage of domain specific knowledge that is provided by the user. Exploiting this knowledge, a solver can search for a plan in these large state spaces efficiently. However, this is a domain specific approach and developing a new solver for every domain is costly. Therefore domain-independent planning systems that are able to solve arbitrary domains were developed. The challenge of developing a domain-independent planning system based on heuristic search is to have heuristic functions that work well on a diverse set of domains. Heuristic search is the systematic exploration of the state space, guided by a heuristic function. A heuristic function estimates the distance from a given state to the goal. Planning systems based on heuristic search use the heuristic function to guide the search towards a goal. To be useful in practice, heuristic functions have to be computed efficiently and work on many domains the planning system tries to solve. To achieve these properties a loss of accuracy has to be taken into account.

Different domain independent and fast to evaluate heuristic functions that have a reasonable accuracy exists. Still there are problems that cannot be solved in reasonable time with these heuristic functions. Arfaee et al. [1, 10] show that it is possible to learn a strong heuristic function with common machine learning techniques, from a given initial heuristic function. This process is called bootstrap-learning of heuristic functions. With a bootstrap learned heuristic function it is possible to solve instances that are not solvable by existing heuristic functions.

Inspired by the capabilities of bootstrap-learning of heuristic functions, this thesis will provide an adoption of the bootstrap-learning approach presented by Arfaee et al. [1, 10] and apply it to classical domain-independent planning. Since there is an upper time-limit to solve a single instance in classical domain-independent planning, the bootstrap learned heuristic function has to be computed online during the search. In classical planning a goal state has to fulfill some goal criteria. Therefore it is possible and likely to have multiple goal states which is not the case in the original bootstrap-learning approach applied to problems with a single goal state. Furthermore, the bootstrap learned heuristic function has to be domain independent, since it will be used for classical domain-independent planning.

This thesis is organized as follows. After showing the notation for planning and presenting machine learning techniques that will be used in this thesis, we will present an approach to learn heuristic functions for classical domain-independent planning, including an improvement based on *BiSS*. We will continue by comparing the learning heuristic functions for classical domain-independent planning approach to some state-of-the-art heuristic functions. Afterwards, we will evaluate different variants of the learning heuristic functions for classical domain-independent planning approaches. In the end of this thesis we will summarize the results and suggest further possibilities to improve the learning heuristic functions for classical domain-independent planning.

2 Background

This chapter defines the notation that will be used throughout this thesis. Furthermore, it gives an introduction to machine learning techniques.

2.1 Planning

This section introduces the formal setting and notation used for planning in this thesis. Planning tasks in this thesis will be formalized in a SAS^+ like finite domain representation, as introduced by Bäckström and Nebel [2].

Definition 1 (planning task)**.** *A planning task is a 4-tuple* $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A} \rangle$*:*

- $\mathcal{V}$ *is a finite set of* ***state variables*** v *which have finite domains* $D(v)$*.*
- *A* ***partial state*** s *is a variable assignment for each variable* $v \in \mathcal{V}$ *from* $D(v) \cup \{u\}$*, where* $u \notin D(v)$ *represents an undefined variable assignment.*
- *A* ***variable assignment*** *of a specific variable* $v \in \mathcal{V}$ *for a given partial state* s *is defined as* $s[v] \in D(v) \cup \{u\}$*.*
- *Let* s *be a partial state. The set of defined variables in* s *is defined as* $var(s) := \{v \in \mathcal{V} \mid s[v] \neq u\}$*.*
- *A state* s *is a partial state for which* $var(s) = \mathcal{V}$
- s_0 *is the* ***initial state****.*
- $s_\star$ *is a partial state that defines the* ***goals****.*
- $\mathcal{A}$ *is a finite set of* ***actions****. An action is a triple* $a = \langle pre(a), eff(a), cost(a) \rangle$*:*
  - $pre(a)$ *is a partial state defining the* ***preconditions****.*
  - $eff(a)$ *is a partial state defining the* ***effects****.*
  - $cost(a) \in \mathbb{R}_0^+$ *is the* ***cost*** *of applying action* a*.*

Actions are also called operators. Actions do not have to be invertible.

The transition function *result*(a, s) produces a new state s', which is called the successor state of s, if a is an *applicable* action in s. An action a is applicable in s if s contains the preconditions $pre(a)$. In order to produce the successor state s' out of s, the transition function *result*(a, s) replaces all variables $v \in var(\mathit{eff}(a))$ which get a new variable assignment $s'[v] = \mathit{eff}(a)[v]$.

A planning task can be represented as a state space. For this thesis the state space will be defined as follows.

Definition 2 (state space)**.** *A state space is a 6-tuple* $\mathrm{S} = \langle S, s_0, S_\star, A, T, c\rangle$*:*

- *S is a finite set of states s.*
- *$s_0 \in S$ is the initial state.*
- *$S_\star \subseteq S$ is a set of goal states.*
- *$A(s) \subseteq A$ denotes actions applicable in a state s of S.*
- *$T(a, s) : S \times A \mapsto S$ denotes a transition function defined for all states s of S and actions a of $A(s)$.*
- *$c(a) : S \mapsto \mathbb{R}_0^+$ is the cost function defined for each action $a \in A$.*

A planning task $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}\rangle$ defines a state space S_Π with states $\mathcal{S}(\Pi)$, initial state s_0, goal states $\{s \in \mathcal{S}(\Pi) | s_\star \subseteq s\}$, actions $\mathcal{A}$, transition function *result*(a, s) and cost function $cost(a)$.

A problem is formalized in a planning task in order to be solved. To solve a planning task means to find a *plan* in its state space.

Definition 3 (plan)**.** *Given a state space $\mathcal{S}_\Pi$ of a planning task $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}\rangle$, a* plan *is a sequence $\pi = a_0, \ldots, a_n$ of actions in A. Moreover, to be a* valid *plan for the planning task Π, π must generate a path of transitions in the state space that starts in s_0 and ends in a goal state $s_\star$ of $S_\star$.*

Planning describes the task of searching for a plan π in the state space $\mathcal{S}_\Pi$. If such a plan π is found, the search was successful. To obtain an optimal plan, the sum of the cost of each action in π has to be minimal among the cost of all plans. *Satisficing planning* prefers to find plans with low cost but also allows to find suboptimal plans, where *optimal planning* only allows optimal plans. This thesis deals with *satisficing planning*.

A planner based on heuristic search starts exploring the sate space from s_0. The *reachable state space* is this part of the state space that can be reached by any sequence of transition functions starting from s_0.

Definition 4 (random walk)**.** *A random walk is a sequence of transitions with randomly chosen actions $a \in A(s)$, which are applicable in the current state s of the random walk, starting at the initial state s_0. The endpoint s_{end} of a random walk, is the last state generated by the sequence of transitions.*

2.2 Machine Learning

This section introduces machine learning in general and defines the notation used in this thesis based on the textbooks by Murphy [14] and Hagan et al. [6]. This section also introduces artificial neural networks based on the textbooks Hagan et al. [6] and Nielsen [15], which is the machine learning technique used for this work.

Machine learning is the field of study in computer science that can learn from data as well as making predictions on data. Other than just following static program instructions, machine learning algorithms build models from input data in order to predict the correct output data. Building such a model, which is also called learning, can be categorized into 3 different types of learning.

Supervised learning The learning algorithm gets example input data with their corresponding correct output value assigned. The goal of the learning algorithm is to develop a general rule that maps from the input data to the output value.

Unsupervised learning The learning algorithm only gets the input data with no corresponding output value. The goal of the learning algorithm is to find patterns in the input data.

Reinforcement learning The learning algorithm has to interact with a dynamic environment to perform a specific task. The learning algorithm receives no external information on how close it is to performing this task.

This thesis only uses supervised learning. The example input data and their corresponding output values used in supervised learning are called data set.

Definition 5 (data set)**.** *A data set is the set U of tuples $u = \langle \vec{x}, y \rangle$:*

- *$\vec{x}$ is a vector of input values $(x_1, \ldots, x_n)$.*
- *y is the corresponding target output value given the input $\vec{x}$.*

Given such a data set U supervised learning can accomplish two different tasks.

classification In classification the learning algorithm assigns the input values $\vec{x}$ to different class labels y.

regression In regression the learning algorithm imitates a function f that maps input values $\vec{x}$ to output values y.

In this thesis we are only interested in regression, since we are looking for a heuristic function h that maps some features $\vec{x}$ of a given state s to a heuristic value y.

2.2.1 Learning and Generalization

This section discusses how learning works in the context of regression. Furthermore, this section introduces the problem of over-fitting, and how this will be avoided in this thesis.

Regression is the task of finding a function f that approximates the function $f^\star$ as closely as possible. A tuple from the data set U can be represented as $y = f^\star(\vec{x})$, where $\vec{x}$ is the given input data from the data set U, y is the target output value assigned to its input $\vec{x}$ and $f^\star$ is the target function that explains the data set exactly. The function f can then be written as $\hat{y} = f(\vec{x})$, where $\vec{x}$ is the given input data from the data set U and $\hat{y}$ is the output data, generated by the function f. The goal of this function f is to learn to predict the correct output value $\hat{y}$ given its corresponding input $\vec{x}$. This information is provided by the data set U. To learn the function f that approximates $f^\star$ the best, a performance measure can be introduced. The overall prediction error on the complete data set U can be used as a performance measure. The goal is to minimize the overall prediction error to get the function f that approximates the function $f^\star$ the best.

If there are no restrictions made for the function $f^\star$, there are many possible functions which can explain the given data perfectly. However, these functions can perform very badly on yet unseen data. This problem is called over-fitting. To avoid over-fitting, a learning algorithm should be able to generalize. Different ways to achieve the goal of generalization exist. This thesis prevents over-fitting by splitting the data set U in two parts. A larger part that remains as training set T and a smaller validation set V which will be used to compute the performance measure. The validation set V only contains data not used for training, to measure how good the function $f^\star$ performs in terms of generalization. The training set T and the validation set V have to fulfill the following conditions:

$$T \cup V = U \tag{2.1}$$

$$T \cap V = \emptyset \tag{2.2}$$

$$|T| > |V| \tag{2.3}$$

For convenience we will use the term training set to refer to the data set in future references. A training set has to provide enough relevant information in order to successfully learn a function f, which means there have to be enough data points in the training set and the data points have to provide useful information. For instance, if the data set only contains 2 data points and no further information. Almost any function f can fit through these 2 points but does not match the target function $f^\star$ at all. If the data set only contains data points in a specific part of the problem space no statement can be made for the parts of the problem space where no information is available.

2.2.2 Neural Networks

In this section we will give a brief introduction to artificial neural networks (ANN) and how this thesis uses ANN.

The human brain is a complex biomechanical machine that is capable of learning complex tasks. It can be seen as a biological neural network. A neural network is an interconnected net of neurons transmitting signals. These signals are received via dendrites and processed by the neuron body. Based on these input signals, a neuron decides to trigger an output signal which will be send out via an axon. Given these basic concepts, an artificial neural network can be built to process input signals. An artificial neural network, or ANN, is an interconnection of artificial neurons. The simplest form of an ANN is a single artificial neuron, which will be discussed next.

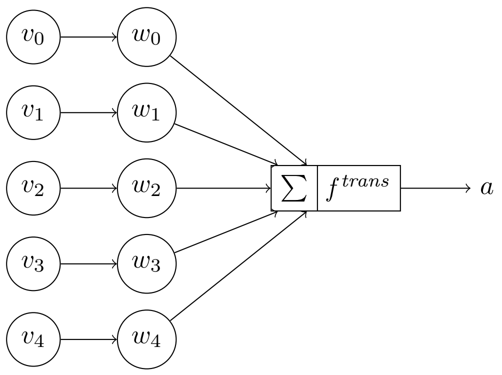


Figure 2.1: Single neuron with input values v_i, weights w_i, output value a and neuron body $f^{trans}(\sum_i(v_i * w_i))$.

2.2.3 Neuron

An artificial neuron, which can be seen in Figure 2.1, from now on called neuron, builds the simplest form of an artificial neural network, and can be used for simple linear classification. A neuron has multiple input values v_i, where each input value v_i has an assigned weight w_i. The input values v_i are the artificial equivalent of the dendrites and the assigned weights w_i are the strength of the connectivity of these dendrites. A neuron has only a single output value a which corresponds to the axon. This output value a is the activation value that determines if a neuron fires or not. The neuron body consists of a decision function f^{trans}. This decision function f^{trans}, also called transfer function, takes the sum of all input values v_i multiplied by their assigned weight w_i and maps it to an output value a.

$$a = f^{trans}(\sum_{i=1}^{n}(v_i * w_i)) \tag{2.4}$$

Equation 2.4 shows how the neuron body works. The weighted sum of the inputs can be abbreviated as $z = \sum_{i=1}^{n}(v_i * w_i)$. There are many different possibilities for such a transfer function f^{trans}. The two transfer functions used in this thesis are the sigmoid-like function $f^{sigmoid}(z) = \tanh(z)$ and a linear function $f^{linear}(z) = z$

Figure 2.2 shows the sigmoid-like transfer function $\tanh(z)$. This transfer function has some sigmoid-like properties that will be used by the neuron. For $z < -2$, $\tanh(z)$ returns -1. For $z > 2$, $\tanh(z)$ returns $+1$. These properties give the sigmoid-like transfer function $\tanh(z)$ the behavior of a decision function. This decision function is used to determine if the neuron should fire or not.

A neuron can be used to solve many different independent problems. However, to be able to solve a specific problem, a neuron has to learn from the problem it is designed to solve and adopt its behavior. At the beginning, the weights w_i assigned to each input value v_i of the neuron are chosen randomly. To achieve the desired output a, corresponding to a specific input $\vec{v}$, the neuron has to adopt the weights w_i assigned to each of its input values v_i. By changing the weights w_i the neuron is able to distinguish between more and less important input values v_i and can therefore assign the correct output a to its corresponding input $\vec{v}$. The change in the weights w_i can be computed exactly for the actual input $\vec{v}_{act}$. If the weights w_i are now adopted to exactly produce the last seen output a_{act}, the prediction

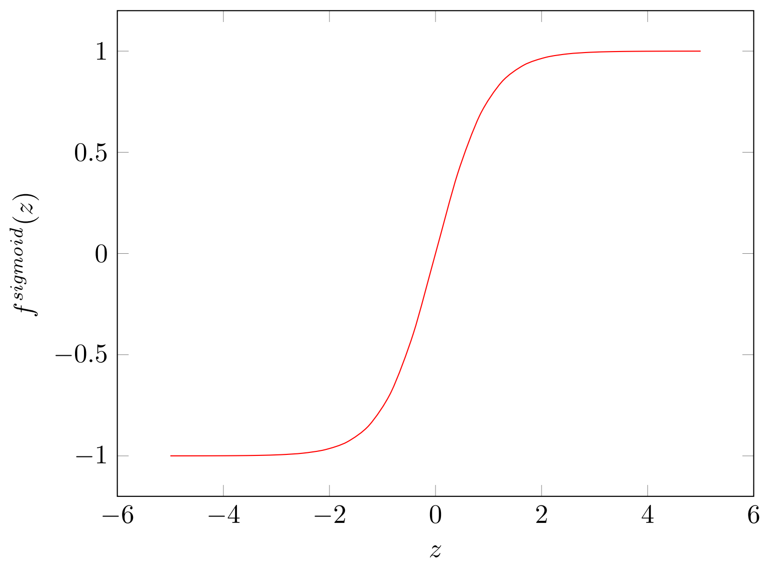


Figure 2.2: The sigmoid-like transfer function $\tanh(z)$.

to previously seen inputs $\vec{v}_{old}$ is most likely to get worse. Because of this behavior, the learning rate η is introduced to adjust the weights in the correct direction. Changes in the weights w_i are multiplied by a small factor $\eta \in [0, 1)$ which is called learning rate.

Since a single neuron is only able to solve linear separable problems, which is not enough to solve complex problems, such as approximating a heuristic function, neurons can be interconnected to each other to build an artificial neural network.

2.2.4 Multilayer Artificial Neural Network

A multilayer artificial neural network, from now on referred to as artificial neural network or ANN, is an interconnected network of neurons aligned in different layers. There are 3 different types of layers in an ANN: the input layer, where each neuron in this layer represents a single input value v_i^0; the hidden layer which can consist of multiple layers with an arbitrary number of neurons in each layer; and the output layer, that can contain one or multiple neurons, which provide the output a^L of the total ANN. In this thesis an output layer with a single neuron is chosen and a single hidden layer with an arbitrary number of neurons. The ANN is fully connected which means that each neuron is connected to each neuron in the previous layer. Connections only exist to the previous layer and there are no cycles in the network. This type of ANN is also called feed forward ANN. Figure 2.3 shows an example of such a feed forward artificial neural network.

Given these concepts the neurons Equation 2.4 can be reformulated to

$$a_j^l = f^{trans}(\sum_{i=1}^{n}(a_i^{l-1} * w_{ij}^l)), \tag{2.5}$$

where l is the actual layer and w_{ij}^l is the weight assigned to neuron j in layer l connected to neuron i in the previous layer $l-1$. The root for this recursive equation is defined as the

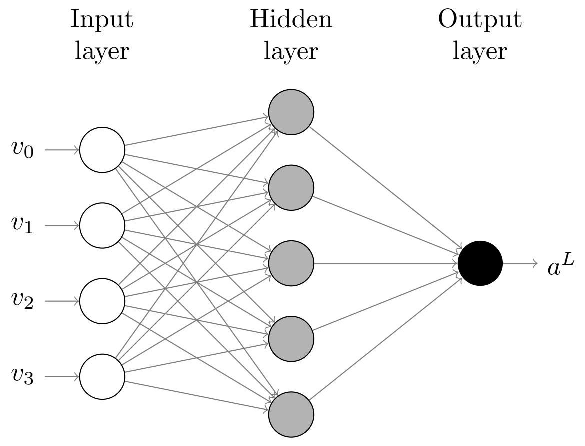


Figure 2.3: Example for a multilayer feed foreword ANN with an input layer, a single hidden layer and an output layer with a single neuron.

input values $\vec{v}$. Therefore the equation for the input layer looks as follows:

$$a_j^0 = f^{trans}(\sum_{i=1}^{n}(v_i^0 * w_{ij}^0)) \tag{2.6}$$

All neurons in the input and hidden layers are sigmoid neurons, they use the sigmoid function as their transfer function f^{trans}. The neuron in the output layer is a linear neuron with the linear function as its transfer function f^{trans}. As the ANN consists of interconnected neurons, like a single neuron, the ANN has to learn from the problem it is designed to solve. Since the weights w_i at the inputs v_i of a neuron are the only thing that can be adopted, the weights w_i have to be learned correctly to predict the correct output a^L to its corresponding input $\vec{v}$. The state-of-the-art algorithm to train an ANN introduced by Rumelhart et al. [16] is the so called *back-propagation* algorithm which will be discussed next.

2.2.5 Back-Propagation

Back-propagation is an algorithm introduced by Rumelhart et al. [16] that will adjust all the weights w_{ij}^l in an ANN so the ANN will be able to to predict the correct output value a^L given its corresponding input $\vec{v}$. This section will give a brief overview of how *back-propagation* works.

The goal of *back-propagation* is to minimize the total error E of the ANN. E can be computed with an arbitrary error function. In this thesis the error function is the mean square error (MSE)

$$E = \frac{1}{n}\sum_{m=1}^{n}(a_m^L - y_m)^2, \tag{2.7}$$

where y_m is the target output value for the input $\vec{v}_m$, a_m^L is the predicted value of the ANN for the given input $\vec{v}_m$ and m indicates the m-th entry U_m in the data set U.

To achieve this goal, *back-propagation* has to adopt the weights w_{ij}^l of the ANN with respect to minimizing the error E of the ANN. In particular *back-propagation* computes the partial derivatives $\frac{\partial E}{\partial w_{ij}^l}$ by back propagating the error E through the network.

$$\delta_j^L = \frac{\partial E}{\partial a_j^L} f'(z_j^L) \tag{2.8}$$

Equation 2.8 shows how to compute the error δ_j^l for each neuron j in the output layer. This equation is easily computable, since each component is known. To propagate the error δ_j^l further back through the ANN, the error δ_j^l can be defined recursively as:

$$\delta_j^l = \sum_i w_{ji}^{l+1} \delta_i^{l+1} f'(z_j^l) \tag{2.9}$$

Using equation 2.8 as the root for the recursion in equation 2.9, the error δ_j^l can be computed for every layer l in the ANN. Given the error δ_j^l in each layer l, the partial derivatives $\frac{\partial E}{\partial w_{ij}^l}$ can be computed as follows:

$$\frac{\partial E}{\partial w_{ij}^l} = a_i^{l-1} \delta_j^l \tag{2.10}$$

Given these 3 equations the *back-propagation* algorithm can be formalized as shown in Algorithm 1.

Algorithm 1: *back-propagation*

```
    Input: $U$, ANN
    Output: ANN
1 while stopping condition not met do
2     $a^L \leftarrow feed\_forward(\vec{x})$;
3     $e \leftarrow compute\_error(a^L, y)$;
4     $backpropagate\_error(e)$;
5     $adjust\_weights()$;
6 end
```

The stopping condition used for *back-propagation* in this thesis consists of 2 parts. The first part is to stop to *back-propagation* algorithm if the total error on the validation set monotonically increases for several consecutive iterations, which are called epochs. This part of the stopping condition is called early stopping and is used to prevent over-fitting. The second part of the stopping condition sets an upper bound for the number of epochs.

3 Learning Heuristic Functions for Classical Planning

In this chapter we will introduce a learning algorithm for heuristic functions in the context of classical domain-independent planning, based on bootstrap-learning of heuristic functions introduced by Arfaee et al [1, 10]. Furthermore we will introduce an extension to the learning algorithm to decrease training set generation time with a solution cost predictor called bidirectional stratified sampler (*BiSS*).

3.1 Bootstrap-Learning of Heuristic Functions

This section briefly describes the bootstrap-learning approach for heuristic functions introduced by Arfaee at al. [1, 10].

Bootstrap-learning describes the process of learning a new heuristic function from an initial weak heuristic function. As described by Arfaee et al. [1, 10] bootstrap-learning is an iterative process where each iteration returns an even stronger heuristic function compared to the previous iteration. To accomplish this task, a new heuristic function has to be learned for the current problem. To be able to learn a new heuristic function, a training set U with enough relevant state space information has to be provided. Given the initial heuristic function h_0, it is only possible to solve easier initial instances. If such an initial instance can be solved, all the states on the path with their distance to the goal are added to the training set U. If the training set contains enough relevant state space information, a stronger heuristic function h_1 that is able to solve more instances and more difficult instances in less time, can be learned from the training set. If the instances were too hard to solve for the initial heuristic function h_0 and therefore the training set U does not contain enough relevant state space information to generate a stronger heuristic function h_1, more easy-to-solve instances are generated by performing random walks with the goal state as their initial state.

To apply bootstrap-learning to classical planning, a lower limit on the training set size cannot be set[1]. Furthermore a set of initial easy-to-solve instances may not be available for every domain, so the initial set of easy-to-solve instances has to be generated for each domain independently by the bootstrap-learning approach. Regarding these points, a learning algorithm for heuristic functions in classical domain-independent planning can be introduced. We will call this algorithm *LHFCP* which is short for **L**earning **H**euristic **F**unctions in **C**lassical domain-independent **P**lanning.

3.2 Learning Heuristic Functions in Planning

LHFCP is initialized with a single start state s_0, a condition for the goal states $\mathcal{S}_\star$ and an initial heuristic function h_0. Algorithm 2 describes the *LHFCP* algorithm in detail.

Algorithm 2: *LHFCP* with search

```
Input: $S_\star, s_0, h_0, random\_walk\_length, number\_of\_random\_walks$
Output: $h^{ann}$
1  $instances \leftarrow []$;
2  $training\_set \leftarrow []$;
3  for $i \leftarrow 0$ to $number\_of\_random\_walks$ do
4      $s_\star \leftarrow generate\_random\_goal\_state()$;
5      $s_{end} \leftarrow end\_point\_of\_random\_walk(s_\star)$;
6      $instances.\ add(s_{end})$;
7  end
8  foreach $State\ s \in instances$ do
9      $\pi \leftarrow search\ from\ s\ with\ h_0$;
10     if $solution\ found$ then
11         foreach $State\ p \in \pi$ do
12             $training\_set.\ add(p, distance\_to\_goal(p))$;
13         end
14     end
15 end
16 $h^{ann} \leftarrow train\_ann(training\_set)$;
```

From line 3 to 7 the easy-to-solve instances are created domain-independently. A random state $s_\star \in \mathcal{S}_\star$ which fulfills the goal conditions is created and a random walk with $s_\star$ as the initial state is performed. The end point s_{end} of this random walk is added to the set of instances if s_{end} is not already in the set of instances[2]. Given the set of instances the algorithm can proceed and create the training set U from line 8 to 15. In line 9 *LHFCP* solves an instance from the set of instances with the initial heuristic function h_0. If a solution is found, line 11 to 13 will add every state p in the plan π with its distance to the goal, to the training set U. In line 16 the training set U is complete and a new heuristic function h^{learn} will be learned from the training set U.

[1] Since some domains with less easy-to-solve instances than the lower limit for the training set size might exist, and *LHFCP* will be applied to all domains independently, it cannot be guaranteed that a lower limit on the training set size can be reached in every domain.

[2] A search from a fixed start state to a fixed goal state with a fixed deterministic heuristic function, always returns the same plan. Therefore the computational overhead of solving the same instance multiple times can be saved, by only including a state once in the set of instances.

This *LHFCP* approach depends on the strength of the initial heuristic function h_0 and its ability to solve enough instances. Solving many of these instances is a time consuming process. Since a random walk starts at a goal state $s_\star \in \mathcal{S}_\star$, the distance from the goal for every state visited during the random walk is known. Because actions are not always invertible, the distance from this state to the goal does not have to be the shortest. However, it can be used as an estimate for the distance to the goal. Exploiting this property, the *LHFCP* algorithm can be reduced to just sampling the state space $\mathcal{S}_\Pi$ with random walks from a goal state $s_\star \in \mathcal{S}_\star$. Algorithm 3 describes this approach in detail.

Algorithm 3: *LHFCP* with random walks

```
Input: $S_\star, s_0, random\_walk\_length, number\_of\_random\_walks$
Output: $h^{ann}$
1 $training\_set \leftarrow []$;
2 for $i \leftarrow 0$ to $number\_of\_random\_walks$ do
3     $s_\star \leftarrow generate\_random\_goal\_state()$;
4     foreach $State\ p \in random\_walk(s_\star)$ do
5         $training\_set.\ add(p, distance\_to\_goal(p))$;
6     end
7 end
8 $h^{ann} \leftarrow train\_ann(training\_set)$;
```

Algorithm 3 is a shorter version of Algorithm 2. There is no initial heuristic function needed and there is no search involved. From line 2 to 7 the training set will be created directly using the random walks. In line 8 the new heuristic function h^{learn} will be learned from the training set U. The advantage of this algorithm is that the time consuming search can be omitted, however there is a disadvantage too. The distance for a state to the goal generated by a random walk does not have to be the shortest possible distance. Obviously, since the distance is generated by random walks, a state that would be right next to a goal, can occur at the end of a random walk, and therefore have a bigger distance than the true distance would be. Furthermore, since actions are not always reversible, the random walk cannot always be reversed to get from s_{end} to $s_\star$. There is also another factor. Since goal states are created at random, it could be that a goal state $s_\star$ is used as the initial state for a random walk where the search would never go towards this goal state $s_\star$. So the state space $\mathcal{S}_\Pi$ could be sampled in uninteresting regions. To deal with this issue, a new random goal state $s_\star \in S_\star$ is created for every new random walk as its initial state. This will increase the probability of sampling interesting regions of the state space $\mathcal{S}_\Pi$.

The second approach looks promising due to the fact that the time consuming search can be omitted but has some issues on its own. There is a third approach: Lelis et al. [11] present a method to predict the optimal distance from a state s to the goal $s_\star$. The third approach uses such a predictor to improve the accuracy of the estimate of the distance of the states sampled by the random walk starting from a goal state $s_\star \in \mathcal{S}_\star$. In Algorithm 4 this approach is described in detail.

Algorithm 4 and Algorithm 3 are essentially the same algorithms. The only difference is in line 5 where the distance to the goal is not the distance generated by the random walk but an estimate of the distance from the current state to the goal. This estimate can be done

Algorithm 4: *LHFCP* with heuristic guess

```
Input: $S_\star, s_0, h_0, random\_walk\_length, number\_of\_random\_walks$
Output: $h^{ann}$
1 $training\_set \leftarrow []$;
2 for $i \leftarrow 0$ to $number\_of\_random\_walks$ do
3     $s_\star \leftarrow generate\_random\_goal\_state()$;
4     foreach $State\ p \in random\_walk(s_\star)$ do
5         $training\_set.\ add(p, heuristic\_value(p, h_0))$;
6     end
7 end
8 $h^{ann} \leftarrow train\_ann(training\_set)$;
```

by every heuristic function. However, Lelis et al. [11] pointed out that a heuristic function is not accurate enough and proposed to use a solution cost predictor instead. The solution cost predictor used in this thesis will be discussed next.

3.3 Bidirectional Stratified Sampling

In this section we will introduce the bidirectional stratified sampling (*BiSS*) approach and how it can be used to decrease training set generation time for *LHFCP*.

Chen [3] introduced a method to estimate some properties of a search tree, called stratified sampling. If the search tree is balanced, it is enough to walk along a single path to collect some information about the search tree. As most search trees are not balanced, a stratifier is introduced in the form of a type system. A type system groups nodes with similar properties together.

Definition 6. *Let $S(s_0)$ be the set of nodes in the search tree rooted at s_0. $T = t_1, \dots, t_n$ is a type system for $S(s_0)$ if it is a disjoint partitioning of $S(s_0)$. For every $s \in S(s_0)$, $T(s)$ denotes the unique $t \in T$ with $s \in t$.*

To estimate the distance from an initial state s_0 to a goal state $s_\star$ a special goal type is introduced which only maps to goal states. Given these goal types, stratified sampling starts sampling the search tree at the initial state s_0 and expands all states s with a different type $T(s)$ for every level in the search tree, until a state $s_\star \in t_{goal}$ is found. The level where this state $s_\star \in t_{goal}$ appears is the estimate for the distance from the initial state s_0 to $s_\star \in t_{goal}$. Figure 3.1 shows an example of stratified sampling, where a goal $s_\star \in t_{goal}$ is found after 3 levels of expansions. The Figure 3.1 shows every state on every layer of the search tree as a circle. The filled circles are states which are expanded by the search. The hollow circles are the states which are not expanded because they have the same type as another state on the same layer.

Lelis et al. [11] showed in their experiments that the estimates for the solution cost produced by stratified sampling were more than double the optimal solution cost. To get a more accurate estimate of the optimal solution cost, Lelis et al. [11] have come up with a different solution called bidirectional stratified sampling *BiSS*. As the name already suggests, stratified sampling is executed in two directions. One starting from the start state s_0 sampling forwards and the other starting from the goal state $s_\star$ sampling backwards. The

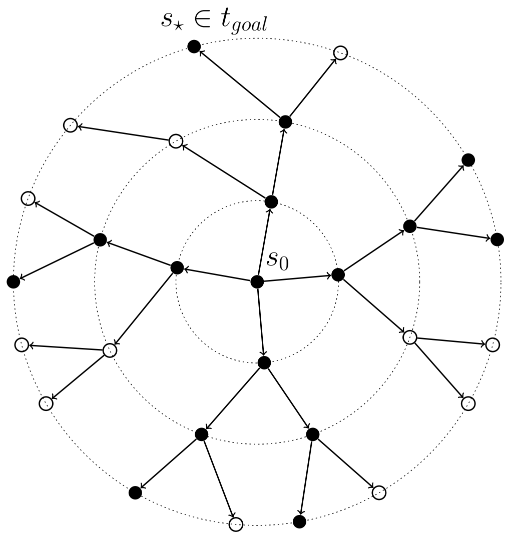


Figure 3.1: Example for stratified sampling for 3 levels of expansions.

forward as well as the backward sampling, expand every state s with distinct types $T(s)$ for a complete layer. *BiSS* checks if the most recent expanded layers, have the same type $t = T(s)$ in common. If there is a type t in common the sum of the level of the most recent layers will be used as estimate for the optimal solution cost. Intuitively, such a type t can occur near to the goal state $s_\star$ and to the initial state s_0, even if the path from the initial state s_0 to the goal state $s_\star$ is very long. To ensure that such an underestimation does not occur, multiple consecutive layers need to have same types t_n in common, which is called a match. Figure 3.2 is an illustration of a match.

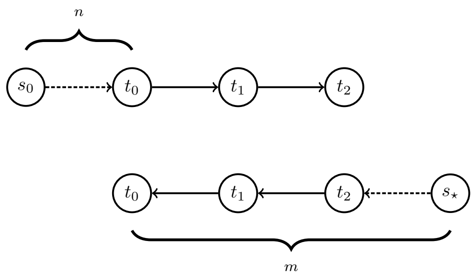


Figure 3.2: Illustration of a match for $K = 2$. Each circle represents a set of types at a level of search. Each t_i denotes just one of the types in the corresponding set.

To apply *BiSS* to classical planning, multiple goal states have to be taken into account. *BiSS* for planning uses a single forward search starting at the initial state s_0 and multiple backward searches starting at a random goal state $s_\star \in \mathcal{S}_\star$. If a type t has occurred in the most recent layer of the forward search as well as in the most recent layer of one of the backward searches, the search stops there. To get a match, *BiSS* for planning advances the forward search and checks if any of the backward searches has still a type t in common on the corresponding layer. If a match did not occur the forward search is set back to its state before the match, and expansion for the forward search, as well as for the backward searches continues as normal.

4 Experiments

Experiments were setup to show the integration of *LHFCP* in a state-of-the-art planning environment. This chapter will present the implementation details and the basic configuration of *LHFCP* as well as the experimental environment. In the reminder of this chapter we will discuss the results of the experiments. We first compare a basic configuration of the *LHFCP* algorithm to some baseline-configuration, to see how the *LHFCP* algorithm performs. In further experiments we will compare different configurations of the *LHFCP* algorithm to examine the different parameters and find good values for these parameters. Furthermore we will discuss the different *LHFCP* approaches and their advantages and disadvantages.

4.1 Implementation Details

This section presents the evaluation system as well as the details for the different *LHFCP* approaches.

The *LHFCP* algorithm was integrated into Fast Downward. Fast Downward is a state-of-the-art planning system introduced by Helmert [8], which is often used as an evaluation system for classical planning. The modularity of Fast Downward allows adding new heuristic functions to the framework easily. Fast Downward already implements several heuristic functions which can be used for comparison.

We added 3 new components to the Fast Downward planner: the *LHFCP* algorithm, the ANN, which is based on an implementation by Miller [13], and *BiSS* for planning. The resulting system has several parameters that can be adjusted by the user: length of random walks as well as number of random walks which control the size of the training set U, the initial heuristic function h_0 which is used to generate the training set U, the learning rate η which controls the learning speed of the ANN, and the topology of the ANN which controls the complexity of the learned heuristic function h^{learn}.

All the experiments were run on a computer with Intel Xeon E5-2660 CPUs running at 2.2 GHz. We set an upper limit for computation time to 30 minutes and a memory bound of 2 GB, as it is common for International Planning Competitions (IPC). The experiments in this thesis considered planning tasks of all domains from IPC up to 2011 from the benchmark suite provided with Fast Downward.

4.2 Results

In this section we will present and discuss the results of the experiments. We will start with the big picture by evaluating the *LHFCP* algorithm based on search with different configurations and comparing it to some baseline configurations.

4.2.1 Big Picture

In this section we will discuss the results of the first *LHFCP* approach, from now on referred to as *search-strategy*, and compare it to some baseline configurations. As baseline configurations we use *blind-search* as a comparison to show that learning is possible. Furthermore we compare *LHFCP* against h^{FF} to see whether it is possible to learn a very strong heuristic function. These experiments were also run to determine the influence of the different *LHFCP* parameters.

$\boldsymbol{h_0}$ This is the initial heuristic function which is used by *LHFCP* to solve initial-instances. Depending on the strength of this heuristic function, more initial-instances can be solved and more of the state space can be explored, which leads to a bigger training set.

number of random walks This parameter controls the size of the training set as well as the exploration of the state space. If there is a large state space with a large branching factor, it is important to perform enough random walks to generate a training set which provides enough information about the state space.

random walk length This parameter sets the maximum length for a random walk. It controls the difficulty of the generated initial-instances. A longer walk can result in more hard-to-solve initial-instances.

topology This parameter sets the number of neurons in each layer of the ANN in the form *input-hidden-output*. The more neurons are added to the hidden layers the more complex functions can be approximated. But with more neurons in the ANN training needs more time to complete. Furthermore the more complex a function can get the higher the probability gets that over-fitting will occur.

$\boldsymbol{\eta}$ This parameter controls the learning rate of the ANN. It is fixed for all experiments.

epochs This parameter sets the maximum number of epochs for the ANN training. This value is fixed for all experiments.

Table 4.1 shows all the parameters that can be chosen for a run of the *LHFCP* algorithm, as well as the default parameter values that will be used in all experiments, if nothing else is mentioned.

Table 4.2 shows the overall performance of the *LHFCP* algorithm against *blind-search* and h^{FF}. The values in the table are computed as follows: *coverage*, *cost* and *memory* are computed as the sum across all domains; and *evaluations*, *expansions*, *generations*, *search time* and *total time* are computed as the geometric mean across all domains. The *coverage* is the number of planning tasks solved by a configuration. A planning task is not solved if it

Parameter	**Value**
method	search
random walk length	50
number of random walks	200
ANN topology	n-50-1
h_0	*FF*
η	0.002
epochs	500

Table 4.1: Basic configuration which is used in all experiments, if no other values are specified.

exceeds the memory limit or the time limit. *LHFCP* solved 13 tasks more than *blind-search* but not as many as h^{FF}. *Plan cost* is the sum of all actions in all plans over all tasks where each configuration found a plan. The configuration for *LHFCP* produces cheaper plans than h^{FF} over all domains but not as cheap as *blind-search* which produces optimal plans. In order to measure the quality of a heuristic function, the following properties are taken into account: *evaluations*, which is the number of states that are evaluated during the search; *expansions*, which is the number of states expanded during the search and *generations*, which is the number of states generated during the search. A heuristic function is better the smaller these values are. Over all domains, the configuration for *LHFCP* requires about half of *expansions*, *evaluations* and *generations* than *blind-search*. The performance of the h^{FF} configuration cannot be reached by the *LHFCP* configuration. Since *expansions*, *evaluations* and *generations* all behave in a similar manor, we will restrict ourself just on *expansions*. *Search time* is the time that is used to search for a plan. We can see that h^{FF} has the lowest value for *search timei* among these configurations. The *blind-search* configuration is slightly faster than the *LHFCP* configuration. *Total time* is the time used to solve a problem, including the generation of the training set and learning of the heuristic function h^{learn}. Table 4.2 shows that the *total time* for the *LHFCP* configuration is higher than the *total time* for the *blind-search* configuration as well as for the h^{FF} configuration. The high *total time* value is caused by the generation of the training set and the learning of the new heuristic function h^{learn}, which is considered as preprocessing. Because of this preprocessing the memory consumption for *LHFCP* is also higher than for *blind-search* and h^{FF}. Total

	blind-search	***LHFCP***	$\boldsymbol{h^{FF}}$
Coverage	680	693	1308
Cost	8150448	8150988	8150992
Evaluations	85955.71	49785.85	1156.04
Expansions	53369.73	24990.02	282.75
Generated	339075.28	158430.39	1937.95
Memory	97249300	110422044	17366928
Search time	1.31	1.62	0.25
Total time	1.35	8.66	0.27

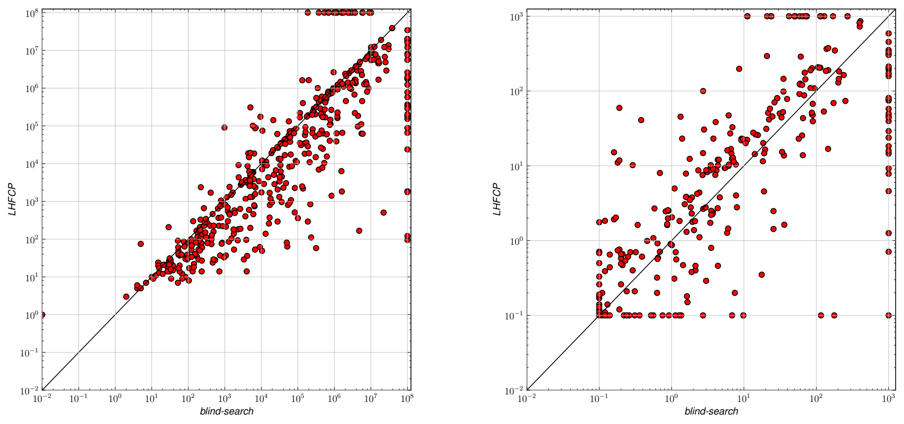
Table 4.2: Results of the experiments for the *search-strategy* against the baseline configuration *blind-search* and h^{FF}.

time and memory consumption will not be used as a quality measure for the new learned heuristic function h^{learn}.

In Table 4.3 we can see that *LHFCP* can solve slightly more instances on some domains than *blind-search*, which shows that it is possible to learn a stronger heuristic function on these domains. Considering the domains **blocks**, **driverlog** and **miconic** we can see that *LHFCP* solves more instances than *blind-search*. Furthermore we can see that the training set provided for these domains is large, which indicates that learning from the training set was possible and a stronger heuristic function h^{learn} could be generated. There are also domains where *LHFCP* could not solve as many instances as *blind-search*. For instance on the domains **mprime**, **mystery**, **no-mprime** and **no-mistery**, *LHFCP* failed to provide a good enough training set and therefore *LHFCP* was not able to solve as many instances as *blind-search* on these domains. The *LHFCP* configuration solves less instances on most domains than h^{FF}. On the domains **gripper**, **movie**, **psr-small**, **storage** and **woodworking-11** *LHFCP* can solve as many instances as h^{FF}.

Coverage	***blind-search***	***LHFCP***	h^{FF}	**Training set size**
airport (50)	21	21	31	6
barman-11 (20)	0	0	0	7041
blocks (35)	18	20	29	36062
depot (22)	4	4	14	6876
driverlog (20)	7	9	15	27359
elevators-08 (30)	2	2	6	21559
elevators-11 (20)	0	0	0	7877
floortile-11 (20)	0	0	7	32
freecell (80)	15	17	74	183
grid (5)	1	1	3	2030
gripper (20)	7	7	7	15276
logistics00 (28)	10	10	21	50862
logistics98 (35)	2	2	8	26767
miconic (150)	50	55	143	155794
movie (30)	30	30	30	1104
mprime (35)	19	15	32	9227
mystery (30)	15	13	19	3807
no-mprime (35)	17	11	30	5307
no-mystery (30)	15	12	19	3806
nomystery-11 (20)	3	3	20	5854
openstacks-08 (30)	12	11	12	1157
openstacks-11 (20)	0	0	0	0
openstacks (30)	7	7	9	169
parcprinter-08 (30)	10	10	20	110
parcprinter-11 (20)	0	0	10	44
parking-11 (20)	0	0	9	1139
pathways (30)	4	4	5	13137
pathways-noneg (30)	4	4	5	13747
pegsol-08 (30)	27	27	30	826
pegsol-11 (20)	17	17	20	426
pipesworld-notankage (50)	14	15	37	31208
pipesworld-tankage (50)	11	13	21	6198
psr-small (50)	49	49	49	36142
rovers (40)	5	7	10	18246
satellite (36)	5	6	19	24094
scanalyzer-08 (30)	12	11	21	28431
scanalyzer-11 (20)	4	3	11	23357
schedule (150)	13	23	33	60253
sokoban-08 (30)	11	11	29	174
sokoban-11 (20)	3	3	19	152
storage (30)	14	16	16	5963
tidybot-11 (20)	3	0	11	0
tpp (30)	6	6	7	1355
transport-08 (30)	6	6	8	21387
transport-11 (20)	0	0	0	7910
trucks (30)	6	6	16	5420
visitall-11 (20)	0	0	0	0
woodworking-08 (30)	4	6	9	15657
woodworking-11 (20)	1	1	1	7187
zenotravel (20)	8	8	13	28208
Sum (2131)	680	693	1308	

Table 4.3: Domain-wise coverage of *LHFCP*, *blind-search* and h^{FF}. And domain-wise training set size of *LHFCP*.



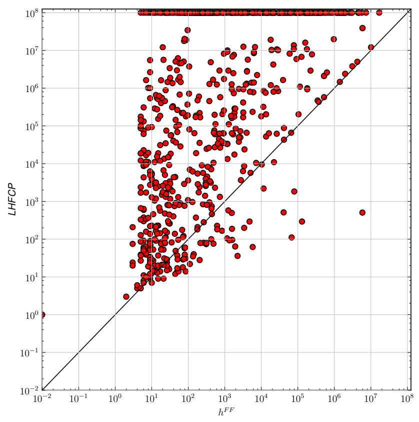
(a) Expansions (b) Search Time

Figure 4.1: Shows the performance of *LHFCP* against *blind-search* for all domains.

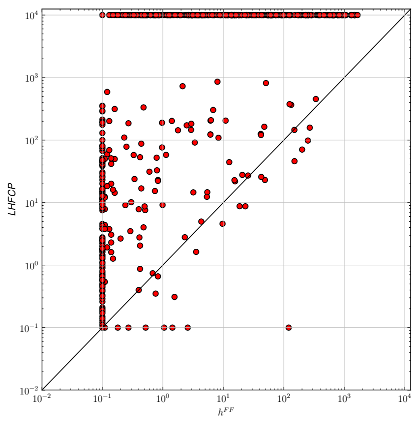
Figure 4.1(a) shows the performance of the *LHFCP* configuration against the *blind-search* configuration for each problem in terms of *expansions*. The *LHFCP* configuration produces less *expansions* on most domains than *blind-search*. For some domains *blind-search* needs less *expansions* than *LHFCP*. On these domains a new heuristic function h^{learn} could not successfully be generated. There are many problems along the diagonal, where a heuristic function h^{learn} was generated that performs about the same as *blind-search* with a slight shift towards *LHFCP*. Figure 4.1(b) shows that the *search time* for most problems is smaller for the *blind-search* configuration than for the *LHFCP* configuration. For problems where *LHFCP* was able to learn a strong heuristic function h^{learn} *LHFCP* finds the solution faster than *blind-search*. For problems that are near to the diagonal in terms of *expansions*, *blind-search* performs better in terms of *search time*. This behavior is caused by the overhead of computing the heuristic value for every state in the search with the ANN, compared to the *blind-search* which always returns a fixed value.

Figure 4.2(a) shows the performance of *LHFCP* against h^{FF} for every problem in all domains in terms of *expansions*. We can see that the majority of the problems is located above the diagonal which means that h^{FF} performs better in terms of *expansions* than *LHFCP*. But there are some problems below the diagonal where *LHFCP* was able to learn a heuristic function h^{learn} which performs better than h^{FF}. These problems come from the domains **gripper** and **psr-small**. In Figure 4.2(b) we can see the performance of the *LHFCP* against h^{FF} in terms of *search time*. We can see that the majority of the problems lie above the diagonal which means that h^{FF} is faster on most of the problems than *LHFCP*. We have already seen that *LHFCP* performs better on the domains **gripper** and **psr-small** in terms of *expansions*. Since there are less *expansions* on the domains **gripper** and **psr-small** the *search time* is lower for *LHFCP* on these domains.

The experiments show that it is possible to learn a strong heuristic function h^{learn} on the domains **gripper** and **psr-small**, because the parameters for the *LHFCP* approach had good values. We will investigate the influence of the different parameters for *LHFCP* next.



(a) Expansions



(b) Search Time

Figure 4.2: Shows the performance of *LHFCP* against h^{FF} for all domains.

4.2.2 Parameter setting

In this section we will discuss the influence of the different parameters that can be chosen for *LHFCP*. The different parameters and their default settings are shown in Table 4.1. The names for the different configurations will use the following naming scheme: *parameterName-value*. The default configuration is called *default* and will use the values specified in Table 4.1. Additional information and discussion for the following experiments can be found in Appendix A.

Table 4.4 shows the results of an experiment where the length of the random walks varies. The *default* configuration can solve slightly more instances than *rwl-20* and *rwl-100*. When it comes to *cost*, all 3 configurations perform more or less the same. The small difference that can be observed in terms of *cost* is because of noise. Therefore we cannot make conclusions about the performance in terms of *cost* depending on the length of the random walks. Table 4.4 shows that *rwl-100* performs better in terms of *expansions* than *rwl-20* and *default*, which is caused by the bigger training set. The *rwl-100* configuration has the lowest *search time* compared to the configurations *rwl-20* and *default*. The low *search time* is caused by the lower value in *expansions*. In Table 4.4 we can see that *rwl-20* has a lower *total time* than the *default* and *rwl-100* configurations. This is because the

Random walk length (rwl)	***rwl-20***	***default***	***rwl-100***
Coverage	687	693	690
Cost	8152460	8152552	8152378
Expansions	29221.89	29205.89	27534.29
Search time	1.87	1.84	1.79
Total time	9.32	9.49	10.02
Training set size	396.76	457.66	519.24

Table 4.4: Results of the experiments for the *search-strategy* with different length for random walks.

execution of the random walk needs less time for *rwl-20.* In terms of *training set size* the configuration *rwl-100* produces the biggest training set and *rwl-20* produces the smallest training set. Since every state on the path which is produced by the preprocessing search starting from a random walk end point to the goal is included in the training set, the training set will be bigger for longer paths. This experiment shows that longer random walks provide a bigger training set with more information where it is possible to learn a stronger heuristic function h^{learn}. The experiment also shows that long random walks produce an overhead in preprocessing and therefore for some problems the timeout is reached before the actual search can start.

Number of random walks (nrw)	***nrw-100***	***default***	***nrw-500***
Coverage	701	693	675
Cost	8152220	8152505	8162028
Expansions	26726.12	25701.05	24337.35
Search time	1.76	1.73	1.70
Total time	7.79	8.95	11.79
Training set size	311.93	466.80	912.01

Table 4.5: Results of the experiments for the *search-strategy* with different number of random walks.

Table 4.5 shows the results of an experiment which investigates the influence of the number of random walks on the quality of the learned heuristic function h^{learn}. We see that an increase in the number of random walks also results in an increase of the *training set size.* As we expected, *nrw-500* generates a heuristic function h^{learn} that performs better in terms of *expansions* than the *nrw-100* and *default* configurations because of the bigger training set. Because of the better performance in terms of *expansions*, *nrw-500* also performs slightly better in terms of *search time* than *nrw-100* and *default*. The overhead in the number of random walks causes *nrw-500* to spend more time in preprocessing than *nrw-100* and *default* which results in the high value for *total time* for the *nrw-500* configuration. In Table 4.5 we can see that *nrw-100* can solve the most instances among the configurations *nrw-100*, *default* and *nrw-500.* We would expect that *nrw-500*, which generates the strongest heuristic function h^{learn}, can solve the most instances but since *nrw-100* has the least overhead in preprocessing, *nrw-100* can spend more time for the actual search than *default* and *nrw-500.* Therefore *nrw-100* has the highest value for *coverage.* In terms of *cost* we can see that *nrw-100* and *default* perform more or less the same, the difference can be caused by noise, but they produce cheaper plans over all than *nrw-500.*

Figure 4.3 investigates the influence of the number of random walks on the size of the training set per domain. As we already know from earlier experiments, *LHFCP* is not able to produce a training set on every domain. For instance on the domains **airport**, **floortile-11** and **visitall-11** *LHFCP* cannot provide a training set. Figure 4.3 shows the expected behavior that the *training set size* increases linearly with the number of random walk on most domains, like for instance **blocks**, **depot** and **zenotravel**. There is a third type of domains, where the *training set size* does not increase with the number of random walks.

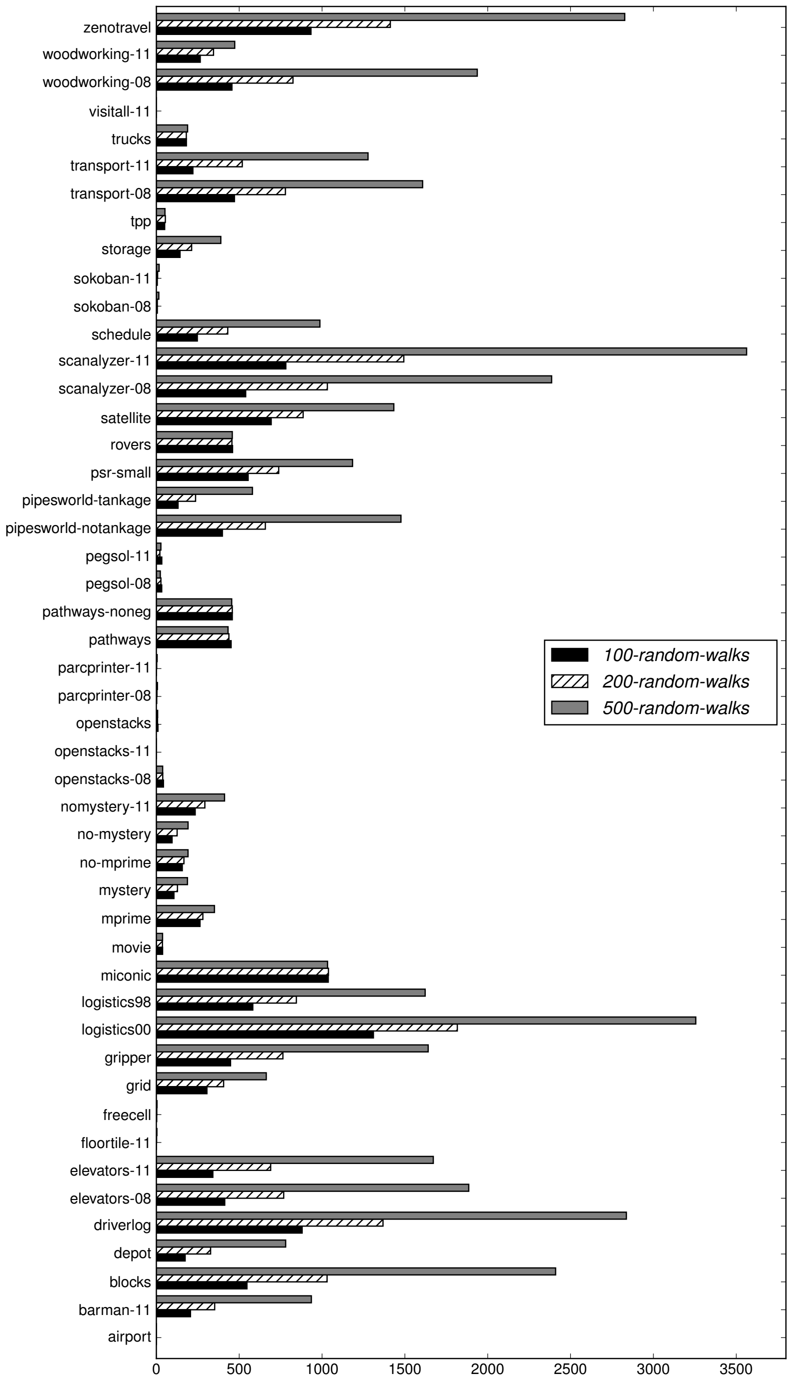


Figure 4.3: Training set size of *LHFCP* with different number of random walks.

Examples for these type of domains are **movie**, **pathways** and **rovers**. Since more random walks produces an overhead in preprocessing, and the information in the training set cannot be enhanced with more random walks, the number of random walks for these domains should be as small as possible to save time for the actual search. This experiment shows that it is

ANN-topology (topo)	***topo-n-20-1***	***default***	***topo-n-100-1***
Coverage	679	693	689
Cost	8152255	8152552	8152356
Expansions	35034.57	28690.82	26111.88
Search time	1.60	1.81	2.31
Total time	7.86	9.43	12.05
Training set size	454.60	453.10	452.80

Table 4.6: Results of the experiments for the *search-strategy* with different ANN topologies.

not possible to chose a value for the number of random walks domain-independently.

The next experiment investigates the influence of the topology of the ANN on the quality of the learned heuristic function h^{learn}. In Table 4.6 we can see the results for this experiment. We consider only ANN topologies with a single hidden layer. The number of the neurons in the hidden layer is changed for this experiment. We will first discuss the configuration *topo-n-20-1*, which uses a topology with 20 neurons in the hidden layer. We can see in Table 4.6 that the configuration *topo-n-20-1* cannot solve as many instances as the *default* configuration. In terms of *cost* the configurations *topo-n-20-1* and *default* produce about the same *cost* and the small difference is caused by noise. Table 4.6 shows that the *topo-n-20-1* configuration needs more *expansions* overall than the *default* configuration. This indicates that the goal heuristic function $h^{\star}$ is a more complex function than the configuration *topo-n-20-1* can produce. To be able to approximate the goal heuristic function $h^{\star}$ better, more neurons have to be added to the hidden layer. Furthermore we can see that the *total time* has the lowest value for the *topo-n-20-1* configuration. This small value for *total time* is because there are less neurons in the ANN and therefore *back-propagation* has to adopt fewer weights which results in a decrease in training time. The same behavior can be observed for the *search time*. Although for *search time* the overhead of more neurons is only produced by feed-forward, we can see that the *search time* is better for *topo-n-20-1* than for the configuration *default* and *topo-n-100-1*.

Table 4.6 shows that the *topo-n-100-1* configuration produces a stronger heuristic function h^{learn} in terms of *expansions* than the *default* and *topo-n-20-1* configuration. Since there are more neurons in the ANN, *back-propagation* has to adopt more weights and the training time increases for the *topo-n-100-1* configuration. Because of the increase in training time, the *topo-n-100-1* configuration spends too much time in preprocessing and can therefore not solve as many problems as the *default* configuration, although it performs slightly better in terms of *expansions*. The value for *cost* is about the same for the *default* configuration as well as for the *topo-n-100-1* configuration. The small difference in *cost* is caused by noise. Since the topology of the ANN has no influence on the generation of the training set, the size of the training set is the same for all 3 configurations. This experiment shows the expected behavior for the topology of the ANN. The more neurons we add to the hidden layer, the better the heuristic performs in terms of *expansions*. But the training time also increases with the number of neurons and *coverage* gets worse.

We can use different heuristic functions as h_0 for the *search-strategy*. The different heuristic functions used in this experiment are: *blind-search*, *FF*, *ipdb* introduced by Haslum

initial heuristic (h_0)	***h_0-blind-search***	***h_0-FF***	***h_0-ipdb***	***h_0-lm-cut***
Coverage	683	705	604	699
Cost	8148963	8149074	8149088	8148928
Expansions	20814.37	19795.51	21012.24	20664.96
Search time	1.48	1.39	1.41	1.40
Total time	10.79	5.53	51.03	6.46
Training set size	181.35	295.55	242.87	234.74

Table 4.7: Results of the experiments for the *search-strategy* with different heuristic functions h_0.

et al. [7] with the implementation of Sievers et al. [19], and *lm-cut* introduced by Helmert and Domshlak [9]. We can see in Table 4.7 that the h_0-*FF* configuration can solve the most problems among all configurations in this experiment. The configuration h_0-*lm-cut* solves 6 instances less than h_0-*FF* and 16 instances more than h_0-*blind-search*. The configuration h_0-*ipdb* solves the least instances among all configurations in this experiment. The low value for *coverage* for the h_0-*ipdb* configuration, is because *ipdb* needs much time to setup which causes an overhead in preprocessing. For the h_0-*ipdb* configuration more instances exceed the maximum time limit in preprocessing than for the other configurations. In terms of *cost* all configurations in this experiment have similar values. The small differences in *cost* are because of noise. Table 4.7 shows that h_0-*FF* produces slightly less *expansions* than all other configurations in this experiment. The configuration h_0-*ipdb* produces the most expansions among all the configurations in this experiment. Furthermore we can see in Table 4.7 that h_0-*FF*, h_0-*ipdb* and h_0-*lm-cut* have about the same value for *search time*. Only h_0-*blind-search* has a higher value in *search time* than the other configurations. The configuration h_0-*ipdb* has the highest value in *total time*. As we already discussed, this high value is caused by the *ipdb* initialization. The *total time* value for the h_0-*blind-search* configuration is significantly higher compared to the configurations h_0-*FF* and h_0-*lm-cut*. Because *blind-search* needs more time to solve the initial instances than *FF* and *lm-cut*, the h_0-*blind-search* configuration also needs more time in total. The configuration h_0-*blind-search* generates the smallest training set among all the configurations in this experiment because *blind-search* cannot solve as many initial instances as the other heuristic functions. We can see that the h_0-*FF* configuration can solve the most initial instances and therefore provides the biggest training set in this experiment.

This experiment shows that the stronger the initial heuristic function h_0 is, the better the learned heuristic function h^{learn} performs. A stronger heuristic function h_0 also generates a bigger training set. Since the initial heuristic function h_0 is used in preprocessing, it should have a small initial time to reduce the overhead.

4.2.3 Walk Strategy

In this section we will present the results of the experiments for the second *LHFCP* variant. We will from now on refer to this approach as *walk-strategy*.

In table 4.8 we can see that the *walk-strategy* produces a newly learned heuristic function

	search-strategy	***walk-strategy***
Coverage	693	692
Cost	8152339	8174179
Expansions	25367.74	25153.76
Search time	1.65	1.61
Total time	8.80	7.82
Training set size	453.09	1850.41

Table 4.8: Results of the experiments for the *walk-strategy* compared with the *search-strategy*.

h^{learn}-*walk* which performs slightly better than the heuristic function h^{learn}-*search* in terms of *expansions*. The *walk-strategy* solves only 1 instance less than the *search-strategy* over the complete set of domains. In terms of *cost* we can see that the *search-strategy* produces overall cheaper plans than the *walk-strategy*. If we look at the size of the training set, we can see that the *walk-strategy* produces a much bigger training set than the *search-strategy*. This is because in the *search-strategy* there are only states in the training set that occur in a plan produced in the *LHFCP* search preprocessing phase. In the *walk-strategy*, states in the training set that are not part of such a plan also exist. If we now look at table 4.8 we can see that the improvement in terms of *expansions* of the heuristic function h^{learn}-*walk* is not as big as we expected given the much bigger training set. Despite the much bigger training set we can see that the *total time* is lower for the *walk-strategy* than for the *search-strategy*. This indicates that the search done in preprocessing to generate the training set is a time consuming process. The *search time* over all domains is slightly lower for the *walk-strategy* which is as we expected since there are less expansions for the *walk-strategy* than for the *search-strategy*.

This experiment shows that it is possible to learn a strong heuristic function h^{learn}-*walk* only relying on sampling the state space with random walks without using any heuristic functions.

4.2.4 Prediction Strategy

In this section we will present and discuss the results of the third approach of *LHFCP*. From now on we will refer to the third approach as *prediction-strategy*. In the previous

	search-strategy	***prediction-strategy***
coverage	49	19
cost	249	249
expansions	21.21	53.91
search time	0.10	0.10
total time	0.36	33.87
training set size	274.89	760.42

Table 4.9: Results of the experiments for the *prediction-strategy* compared to the *search-strategy* on the domain **psr-small**.

section, we already saw the advantages of sampling the state space $\mathcal{S}_\Pi$ and omit the search. Now we also want to enhance the accuracy of the training set. This experiment is executed only on the domain **psr-small** because it was possible to learn a strong heuristic function on this domain. Because our implementation of *BiSS* for planning is resource hungry the experiments are restricted only to the domain **psr-small**. We will use the following type system in these experiments: $T_c(s) = (h(s), c(s, 0), \ldots, c(s, H))$, where $h(s)$ is the heuristic value for state s, $c(s, k)$ is the number of children of s which have the heuristic value k and N is the maximum heuristic value. N was set to 100 for these experiments and the heuristic value is computed with $h_0 = FF$.

Table 4.9 shows the overall performance of the *prediction-strategy* compared to the *search-strategy* for the domain **psr-small**. We can see that the *prediction-strategy* solves 30 instances less than the *search-strategy*. Because the implementation of *BiSS* for planning uses so many resources, *LHFCP* exceeds the time limit for most instances already in the preprocessing phase, and can therefore not solve as many instances as the *search-strategy*. In terms of *cost* the *prediction-strategy* as well as the *search-strategy* produce the same cost over all problems that could be solved. We can see in Table 4.9 that the *prediction-strategy* produces more *expansions* than the *search-strategy* which is not as we expected. The *search time* is the same for both approaches. The *total time* for the *prediction-strategy* is much higher than the *total time* for the *search-strategy* which is because of the resource hungry implementation of *BiSS* for planning. Table 4.9 shows that the training set is much bigger for the *prediction-strategy* than for the *search-strategy* which is because the *prediction-strategy* adds all states visited during the random walks to its training set, and the *search-strategy* only adds states that occur in a plan produced by the preprocessing search to its training set.

This experiment shows that *BiSS* for planning is too resource hungry to be used with *LHFCP*.

4.3 Discussion

In this section we will give a brief wrap up on the experiments that we have run in this chapter and give an overview of the influence of the different parameters.

The experiments have shown that it is possible to learn a heuristic function which produces less *expansions* than *blind-search*. The experiments have also shown that the overall time to solve a problem with the learned heuristic function increases which has to be taken into account if a heuristic function should be learned. The experiments that were run to determine the influence of the different parameters, have shown that it is not possible to be domain-independent with every parameter. The random walk length produces a slightly bigger training set with increasing length. The *training set size* increases almost linear with the number of random walks on most domains. But there are also domains where an increase in the number of random walks does not increase the size of the training set and therefore the number of random walks should be as low as possible on these domains to reduce the overhead in training set generation. Further experiments have shown that more neurons in the hidden layer of the ANN increase the performance of the heuristic function. But there is a trade off. The more neurons there are the more time takes the training and less time can be used to solve an instance. The experiments that investigated the influence of different initial heuristic functions h_0 showed that a stronger initial heuristic function h_0 can solve more instances from the set of initial-instances and therefore provide a bigger training set with more information which generates a slightly stronger heuristic function h^{learn}. The experiments have shown that the best set of parameters cannot be chosen easily. There is a trade off between less *expansions* and better *coverage*. The best trade off for the parameter settings is the *default* configuration with less random walks.

The next set of experiments have shown that the search phase can be omitted in preprocessing which leads to a decrease of the *total time* and an increase in the *training set size*. The increase in the *training set size* is because all states visited by the random walks are included in the training set. The performance of the learned heuristic function for the *walk-strategy* slightly improves compared to the *search-strategy*. Further experiments have shown that *BiSS* for planning does not improve the learned heuristic function. Moreover, it shows that *BiSS* for planning is too expensive and the time limit is often exceeded during the preprocessing phase.

5 Related Work

Arfaee et al. [1, 10] describe a machine learning approach to learn a heuristic function during the search, which they call bootstrap-learning of heuristic functions. Bootstrap-learning of heuristic functions starts with a weak initial heuristic function h_0, a set of initial problem instances and a time limit to solve these initial problem instances. The instances that can be solved by bootstrap-learning in the given time using the initial heuristic function h_0, provide the training examples to learn a stronger heuristic function. The stronger heuristic function replaces the weaker heuristic function and the bootstrap-learning process is repeated with the stronger heuristic function, to obtain an even stronger heuristic function. This process is repeated until a heuristic function that is strong enough to solve hard instances in the given amount of time is returned by the bootstrap-learning process. If the bootstrap-learning process cannot solve enough instances in the given amount of time, a random walk procedure produces easy to solve instances, starting from the goal state.

Arfaee et al. [1] introduce an approach to solve single problem instances using the bootstrap-learning approach, without the need of a training phase in the order of days.

Lelis et al. [12] propose a way to reduce the learning time used by the bootstrap-learning approach from the order of days to minutes. Their approach is to estimate the distance to the goal directly, instead of letting the bootstrap-learning approach solve the given problem instances. They use these predictions to learn the new heuristic function.

Samadi et al. [17] introduce a method where a heuristic function is learned during search from a set of other heuristic functions. Instead of picking the maximum heuristic value out of the given heuristic functions, the new heuristic function is computed with an artificial neural network (ANN), from a set of different input heuristic functions. The ANN is precomputed and used during the search to compute the heuristic value for a state.

Thayer et al. [21] propose a method to use suboptimal search algorithms to reduce the solving time by sacrificing guaranteed optimality. They present a technique for improving the heuristic function during search based on an approach introduced by Sutton [20] called temporal difference learning. Since they apply the heuristic function for suboptimal search, the learned heuristic function can be inadmissible. They applied the on-line learned heuristic function to greedy best-first search introduced by Doran and Michie [4]. This new learned heuristic function produced better solutions faster than previous heuristic functions used with greedy best-first search.

Sarkar et al. [18] introduce a method for learning a heuristic function while solving a problem, to solve subsequent problems faster. They assume that these subsequent problems come from the same distribution and therefore the distribution can either be estimated in an off-line training phase or on-line while solving the actual problem.

Fink [5] proposes a method of learning a heuristic function out of several given heuristic functions. One possible way of getting a heuristic value out of other heuristic values is by choosing the maximum heuristic value among the given heuristic values. In his paper he describes an approach where he computes the new heuristic value as a weighted sum of the different base heuristic values. The weights of the new composite heuristic function are trained on-line by using the known distance to the nodes which were already visited during the search.

6 Conclusion

This chapter gives a conclusion of the work done in this thesis. The last section of this chapter presents possible future work and improvements of the *LHFCP* approach.

6.1 Overall Results

We introduced a framework that is capable of learning heuristic functions in the context of classical domain-independent planning. The framework is inspired by an approach called bootstrap-learning for heuristic functions introduced by Arfaee et al. [1, 10]. The subject of their approach as well as of our approach was to use common machine learning techniques to learn a strong heuristic function which can be used to speed up heuristic search.

We have shown that it is possible to learn a heuristic function in the context of classical domain-independent planning by relying on common machine learning techniques like artificial neural networks. We have also shown that it is possible to learn such a heuristic function, without relying on any other heuristic function by just sampling the state space with random walks starting from a goal state. Furthermore we have added a solution cost predictor as an improvement for our approach, based on a suggestion for the original bootstrap-learning approach made by Lelis et al. [11] which is called bidirectional stratified sampling (*BiSS*). We showed that *BiSS* can be adapted to planning tasks which have more than a single goal state. The experiments have shown that *BiSS* for planning produces a too big overhead to be used with *LHFCP*. Furthermore, the experiments showed that a domain-independent approach is not possible with *LHFCP*. There are many parameters that have a different optimal value for different domains. Since a new heuristic function has to be learned for every problem individually, there is always an overhead in total time that has to be taken into account.

Although *LHFCP* did not perform as good as already existing heuristic functions and has some issues with domains with non invertible operators, it seems a viable alternative in domains where existing heuristic functions perform badly or to improve the heuristic function in specific domains, without additional knowledge of the domain itself.

6.2 Future Work

The modularity of the *LHFCP* framework allows to substitute the learning component with each other machine learning technique. Although ANNs are very powerful and can approximate almost every function, it can be very difficult to choose the correct topology for the current task. There might be some other machine learning techniques that are better suited for this task. In common machine learning tasks the efficiency of a learner can be improved by the introduction of features which expose more sophisticated domain knowledge to the learner. Until now the features are the raw values of a state, but the framework allows to easily add some other features. The size and the quality of the training set depend on the number of random walks as well as the length of the random walks. Since these parameters have other optimal values for each problem, they can be estimated in a preprocessing phase. Stratified sampling introduced by Chen [3] which can estimate some properties of a search tree could be used to estimate the length of the random walks, as this is related to the length of the solution cost, as well as the number of random walks which is related to the branching factor. Exposing this knowledge, a training set which provides the necessary state space informations can be built more efficiently for every domain and the overhead in training set generation can be reduced to save more time for the actual search.

Bibliography

[1] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16):2075–2098, 2011.

[2] Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[3] Pang C Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21(2):295–315, 1992.

[4] Jim E Doran and Donald Michie. Experiments with the graph traverser program. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 294, pages 235–259. The Royal Society, 1966.

[5] Michael Fink. Online learning of search heuristics. In *International Conference on Artificial Intelligence and Statistics*, pages 114–122, 2007.

[6] Martin T Hagan, Howard B Demuth, Mark H Beale, et al. *Neural network design*. Pws Pub. Boston, 1996.

[7] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, Sven Koenig, et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, volume 7, pages 1007–1012, 2007.

[8] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, 26: 191–246, 2006.

[9] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: what's the difference anyway? In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.

[10] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. Bootstrap learning of heuristic functions. In *Third Annual Symposium on Combinatorial Search*, 2010.

[11] Levi Lelis, Roni Stern, Ariel Felner, Sandra Zilles, and Robert C Holte. Predicting optimal solution cost with bidirectional stratified sampling. In *ICAPS*, pages 155–163, 2012.

[12] Levi HS Lelis, Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. Learning heuristic functions faster by using predicted solution costs. 2012.

[13] David Miller. Neural network tutorial. http://inkdrop.net/dave/docs/neural-net-tutorial.cpp, 2011. [Online; accessed 09-02-2016].

[14] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[15] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[16] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5:3, 1988.

[17] Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristics. In *AAAI*, pages 357–362, 2008.

[18] Sudeshna Sarkar, Partha P Chakrabarti, and Sujoy Ghose. Learning while solving problems in best first search. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 28(4):535–541, 1998.

[19] Silvan Sievers, Manuela Ortlieb, and Malte Helmert. Efficient implementation of pattern database heuristics for classical planning. In *SOCS*, 2012.

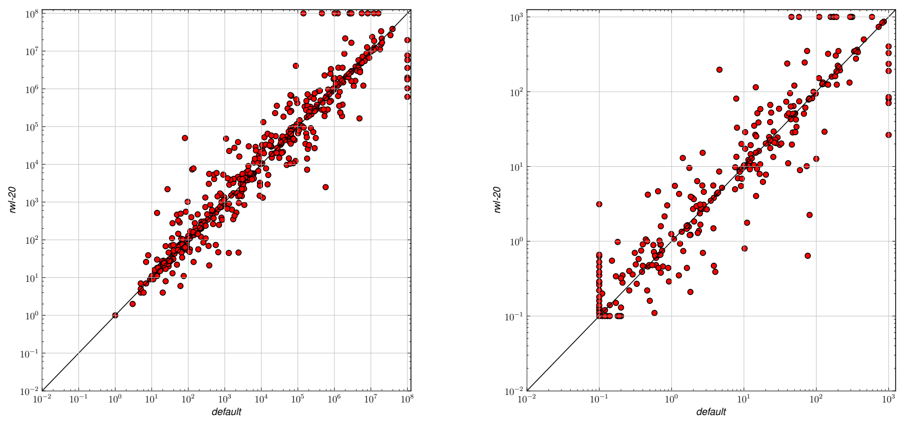
[20] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[21] Jordan Tyler Thayer, Austin J Dionne, and Wheeler Ruml. Learning inadmissible heuristics during search. In *ICAPS*, 2011.

A
Appendix

A.1 Problem Wise Evaluation

In this section we will discuss the results of the experiments for the different configurations of *LHFCP* in more detail. We will investigate the performance of the different parameters: random walk length, number of random walks and topology of the ANN in terms of *expansions* and *search time* with problem wise distribution.

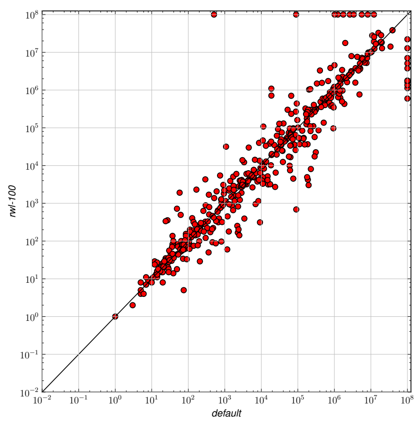


(a) Expansions (b) Search Time

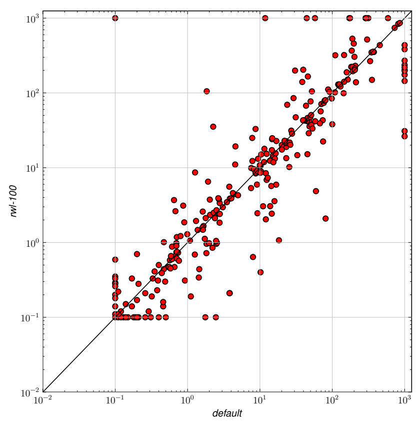
Figure A.1: Shows the performance of the *default* configuration against *rwl-20* for all domains.

In Figure A.1(a) we can see that all the problems are aligned along the diagonal. Some problems can be solved with less *expansions* by the *default* configurations, other produce less *expansions* for the *rwl-20* configuration. Both configurations perform about the same over all problems. Figure A.1(b) shows that the *default* configuration can solve more problems at a minimum *search time* than the *rwl-20* configuration. Most of the problems are aligned along the diagonal.

Figure A.2(a) shows the performance of the *default* configuration compared to the *rwl-100* configuration. All problems are aligned along the diagonal, but there is a small shift to the bottom which means that the *rwl-100* configuration performs slightly better in terms of *expansions*. In Figure A.2(b) we can see that more problems are located below the diagonal which means that the *rwl-100* configuration has a lower *search time* in general than the *default* configuration.



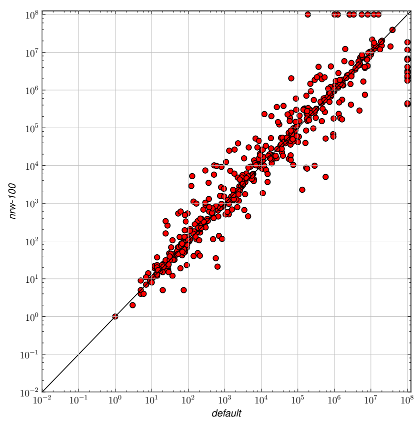
(a) Expansions



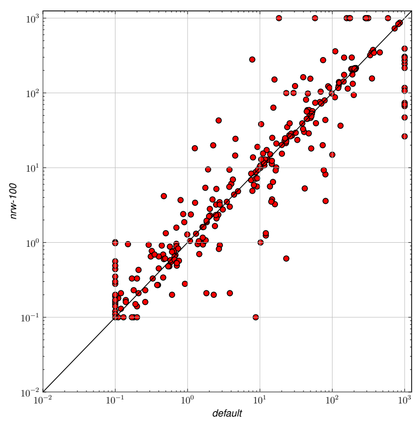
(b) Search Time

Figure A.2: Shows the performance of the *default* configuration against *rwl-100* for all domains.

We already saw in Table 4.5 that the difference in terms of *expansions* and *search time* is marginal. In Figure A.3(a) we can see that this marginal difference is because all the problems are near to the diagonal, so none of the configurations *nrw-100* and *default* can be preferred in terms of *expansions*. Figure A.3(b) shows that *nrw-100* and *default* have a similar search time for most problems. There are some problems that can be solved faster



(a) Expansions

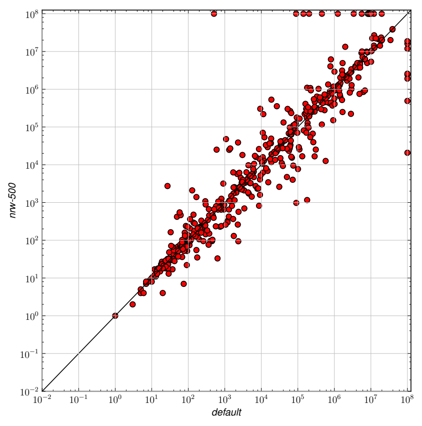


(b) Search Time

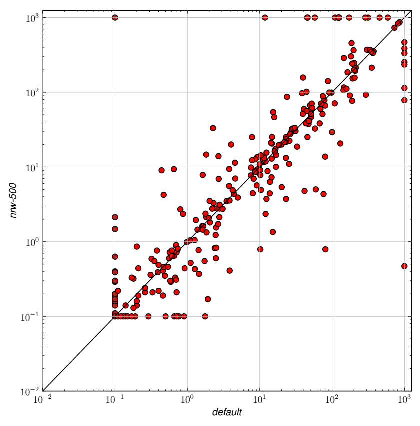
Figure A.3: Shows the performance of the *default* configuration against *nrw-100* for all domains.

by the *default* configuration but there are also problems that can be solved faster by the *nrw-100* configuration. Neither of the configurations *nrw-100* nor *default* can be preferred in terms of *search time*.

In Figure A.4(a) we can see that the *default* configuration and the *nrw-500* configuration have about the same number of *expansions* for most problems. The same behavior can be observed in Figure A.4(b) in terms of *search time*.



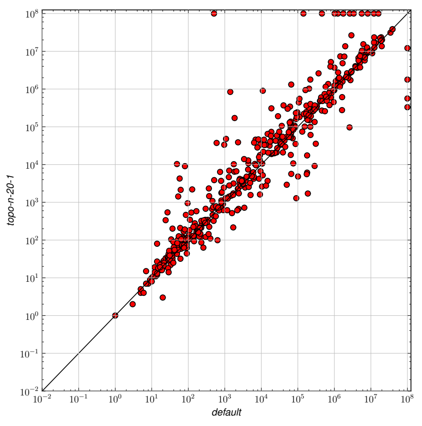
(a) Expansions



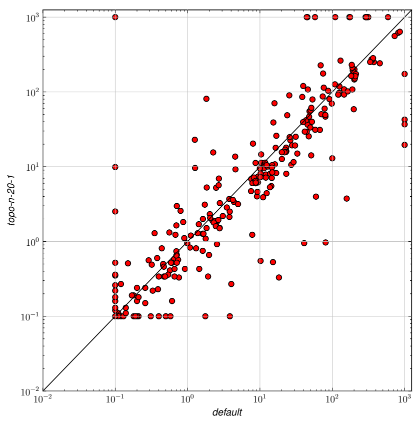
(b) Search Time

Figure A.4: Shows the performance of the *default* configuration against *nrw-500* for all domains.

Figure A.5(a) shows the performance of the *topo-n-20-1* configuration against the *default* configuration. We can see that slightly more problems are located above the diagonal, which means that the *default* configuration produces less *expansions* over all problems than the



(a) Expansions

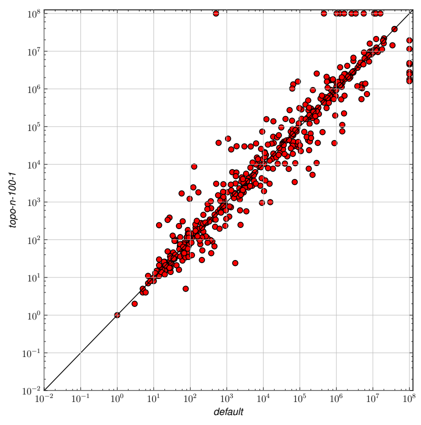


(b) Search Time

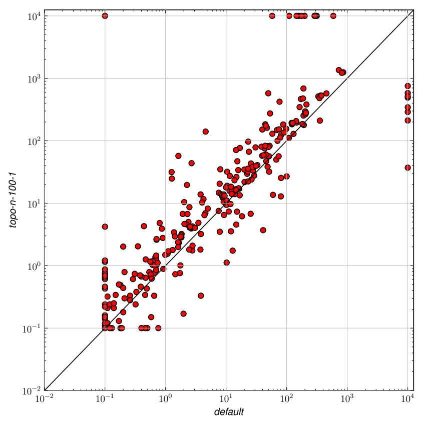
Figure A.5: Shows the performance of the *default* configuration against *topo-n-20-1* for all domains.

topo-n-20-1 configuration. There are some problems which are located below the diagonal. For these problems the *topo-n-20-1* configuration was able to produce a heuristic function h^{learn} which produces less expansions than the *default* configuration. This experiment shows that there are some problems that can be approximated with an ANN with few neurons, but there are also other problems that need more neurons in the hidden layer to be approximated. In Figure A.5(b) we can see that the majority of the problems are located below the diagonal. This behavior can be observed because the *default* configuration has more neurons in the hidden layer than the *topo-n-20-1* configuration which means that calculating the heuristic value with an ANN with more neurons takes more time than with fewer neurons. There are some problems in Figure A.5(b) that are above the diagonal. Despite the computational overhead of more neurons the *default* configuration is faster, because it produces less expansions on these problems.

In Figure A.6(a) we can see that the problems are aligned along the diagonal with a slight shift to the bottom which means that *topo-n-100-1* performs better in terms of *expansions* than the *default* configuration. We can observe a similar behavior as for the *topo-n-20-1* and *default* configuration. There are some problems that need more neurons in the hidden layer to be approximated correctly, but there are also problems that can be approximated with fewer neurons in the hidden layer. Figure A.6(b) shows the performance of the *default* configuration against the *topo-n-100-1* configuration in terms of search time. As we expected, most of the problems are located above the diagonal because there are more neurons in the *topo-n-100-1* configuration than in the *default* configuration which cause an overhead in computation when the heuristic function is evaluated.



(a) Expansions



(b) Search Time

Figure A.6: Shows the performance of the *default* configuration against *topo-n-100-1* for all domains.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Cedric Geissmann

Matriculation number — Matrikelnummer
10-056-026

Title of work — Titel der Arbeit
Learning Heuristic Functions in Classical Planning

Type of work — Typ der Arbeit
Master Thesis

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, February 12, 2015

[illegible]

Signature — Unterschrift