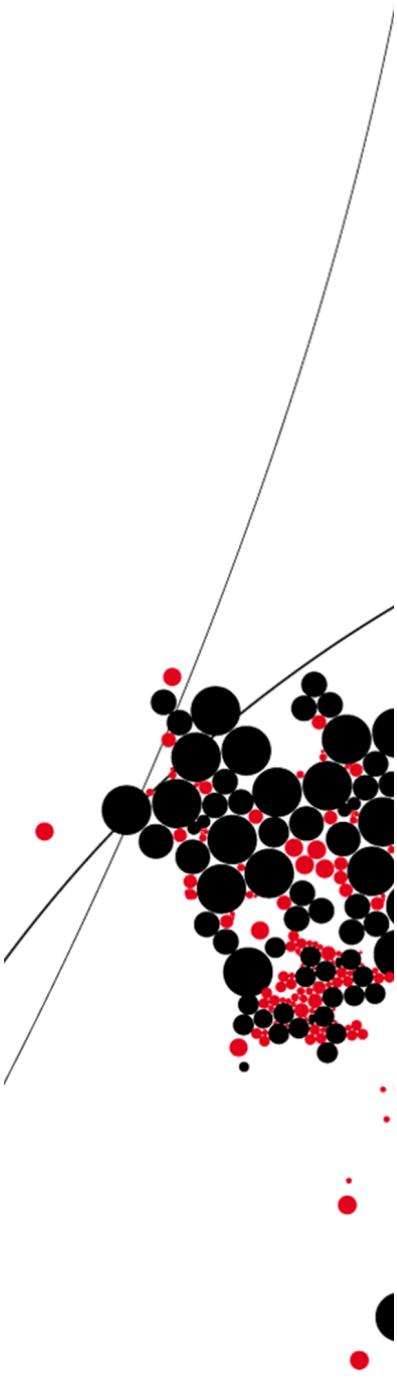
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**



On a Framework for Domain Independent Heuristics in Graph Transformation Planning

Jochem W. Elsinga

M.Sc. Thesis
August 2016

Supervisors:
prof. dr. ir. A. Rensink
dr. ir. J. Kuper

Formal Methods & Tools Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

ABSTRACT

Planning is the task of finding a sequence of actions which, for an initial state, reach a predefined goal. A technique to model planning problems is graph transition systems. Graph transition system are an extension of a transition system in which states are associated with a unique graph and transitions are expressed by graph transformations and graph morphisms.

A technique to solve planning problems is forward state space exploration. However, for large problem instances a state space may become unbearably large and uninformed state space exploration is no longer feasible. It becomes necessary to have some functionality which guides the exploration. For with we use a heuristic, which is a function which estimates the distance of a state to the goal.

In this work we developed and present two distinct domain independent heuristic approaches which are based on the underlying graph transformation planning problem modeling paradigm. The first of these is the NENTUPLE approach which is based on the comparison of graph abstractions. The second approach is linearization abstraction which is based on calculating distance metrics from linearized abstracted state spaces.

We have furthermore designed and implemented a framework in GROOVE for heuristics functions. In this framework we have implemented he rustics functions based on the above mentioned approaches.

To evaluate the effectiveness of the heuristic approaches developed in the thesis we had run experiments using three planning problems. From the results we found that our heuristics provide a significant improvement over uninformed exploration and furthermore perform equal or better than heuristics presented in related work (in our problem set).

Finally, this work has made a initial classification of distinguishing features of graph transformation planning problems and correlated these to the effectiveness of the heuristics approaches presented in this work.

ACKNOWLEDGMENTS

The sentence:

> “I would like to thank everybody."

Is all I originally had written in my acknowledgments. However, I feel that this does not do justice to all the people in that have been there for me during my work on this thesis, my whole student time, and my life in general. So, in this short foreword I would like to take some time and thank some people individually for all they have done for me.

The first people I would like to thank are my parents Watze and Angelique who have supported me in many ways and have always been there for their first and favorite son. I also guess this is the part where I mention the fact that I proved my mother wrong when she tried to convince me university might be to challenging for me after my first year.

I would also like to thank my brothers; Rinse, Jesse, Hidde and Hessel who I cherish greatly. Furthermore, I want to thank Judith who has been very loving and supporting, even during my moody phases.

Next, I would like to thank all house mates I have had. Especially some of my former house mates with whom I have been or am still very close friends. In no particular order these are: Dani, Simon, Remke, Meine and Annika. I think what I miss most from my student life is the times we had together.

I would also like to thank Kaspar, Robin and David for all the good times we have had together. I feel they motivated and pushed me to achieve all I could from my studies. I want to especially mention David, who during my the time working on my thesis was always ready to discuss my work and answer any questions I had. He and Vincent also took the time to offer me feedback on my work and for that I thank you both.

I would like to thank my advisors Arend and Jan for all they have done for me during my master thesis. For all the guidance and feedback they have given. I have learned a lot in this period and you both challenged me to produce my best work. I am proud of the work I have done, and it would not have been possible without their help.

Finally, I would like to thank Li Ruonan and Güner Orhan with whom I shared an office during my thesis work. Besides always having time for a chat or a funny story they were always ready to provide me with a key-card so I could get some much needed coffee.

For all the others that I have failed to mention, I would refer them back to my original acknowledgments and have them know they are not forgotten.

Jochem W. Elsinga
August 2016

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

Everyday we make decisions in every aspect of our life, some of these are made automatically subconsciously and some require a lot of thought. In any case, one can consider a sequence of decisions (or actions) as a *plan* to achieve some desired goal from some starting situation. A trivial example would be putting on ones shoes before leaving the house. The actions one might take (in order) are: `put on left shoe`, `put on right shoe`, `tie left shoe` and `tie right shoe`. The task of coming up with such a sequence of actions, also called plan or path, to achieve the goal is called *planning*. People are generally not aware they are solving planning problems or even dealing with the complexity involved in planning. In the world of computer science planning is a research field which is critical to sectors such as manufacturing, space systems, software engineering, robotics, education, and entertainment [1].

The planning research field formalizes the components that we have intuitively presented above. There are three components, these are: an *initial state*, a number of *actions* and a *goal*. These components all exist within the *environment* of the problem. Thus, for our example the environment consist of a left and right foot, a corresponding shoe which may be on or off, and a lace which is either tied or untied. The actions are those given in the previous paragraph and the initial state is the one with both shoes off. Finally, the goal is that both shoes are on and tied. These three components and the environment together are called a planning problem. Furthermore, the set of combinations of values which each element of an environment can take is known as the *state space* of a problem. Intuitively, the state space thus represents all possible states a problem may be in.

The objective of a planning problem is to determine a plan which leads from the initial state to some state in the problem state space that meets the goal requirements. A plan is thus the solution to a planning problem. Planning research concerns itself with *automated* methods of solving planning problems. For this reason, planning is sometimes also referred to as automated planning.

While our example has a trivial solution, planning can, for larger problems, become complex relatively quickly. This complexity stems from three factors which are described below.

- The first factor is the number of possible actions, as the number of actions increase more sequences need to be considered in order find a solution. For example, if we not only have actions to put on and tie shoes but also for putting on socks, the number of possibilities increases and thus so does the complexity of the problem.
- The second factor is the size of the environment, which depends on the first factor. As the number of elements in the environment increase the more possible actions become possible. For example, imagine not a single pair of feet and shoes as in our example but several. In this case there are many more possible combinations to consider to find a plan, which again increases the complexity of the problem.
- The third factor is the solution requirements. While some planning may simply require any plan as solution, one could also introduce additional constraints such as that the solution should be the shortest in terms of number of actions. Consider actions have a cost, (i.e. time, distance, currency) a constraint could be to find the cheapest or most expensive plan. This factor becomes relevant for planning problems such as routing (i.e. navigation software) or the traveling salesman problem.

The first two factors cause the phenomenon known as *state space explosion* to occur in planning problems. State space explosion occurs when the size of the state space grows out of control due the increase of combination between states and actions. In this work we are not concerned with the third factor and will only be interested in finding any solution to for a planning problem.

This increased complexity means that finding a solution to a planning problem becomes increasingly more difficult (in terms of time and memory) as the problem size increases. To deal with this issue one typically makes use of *heuristics* which are used as a form of artificial intelligence and aid in the automated solving of planning problems. Heuristics assist by estimating which actions, for some state, lead to a state more closely resembling a state which satisfies this goal. The number of actions is known as the *distance* from a state to the goal. To develop a heuristic one should determine metrics which can be used to estimate this distance.

Heuristics are not an exact science and thus, predicting or determining whether a heuristic is effective is not trivial. One approach is to design a heuristic explicitly for a problem, making it rely on problem specific knowledge. The drawback of this however is that a new heuristic is needed for each problem. A second approach is to use a more catch-all approach in which the heuristic is based on underlying problem modeling knowledge. This also known as a *domain independent heuristic*.

This thesis is on planning, particularly the goal is to develop domain independent heuristics to solve planning problems that are modeled as graph transition systems. The following section gives the context in which we attempt to achieve our goal.

1.2 PROBLEM CONTEXT

In order to formalize the planning components of real-life problems and systems, we will use model representations of those systems. *Modeling*, in the scientific sense, is a technique of formally and systematically representing concepts in an abstracted manner for the purpose of getting a better understanding of those concepts. Within computer science, modeling and models have varying definitions. In essence, modeling is a broad and general term which is used to indicate a technique in which one can emulate reality within some boundaries. Starfield *et al.* [2] give a general definition of a model which captures how broadly the field of modeling can be applied, while still expressing what modeling achieves. The definition is as follows:

> A model is a representation of a concept. The representation is purposeful: the model purpose is used to abstract from the reality the irrelevant details.[2]

In this work we will use modeling to create formal and abstracted representations of systems relevant for planning problems. The model is formal in the sense that we have mathematically defined the environment and other components of a planning problem. The model is abstracted in the sense that we have limited the scope of the planning problems environment.

A technique for modeling is transition systems. A *transition system* is a mathematical model which can be used to describe the behavior of discrete systems. It consists of states and transitions between states. A state represents some situation of the system and transitions correspond to actions which change the situation of the system (and thus lead to another state).

There is a clear correlation between transition systems and planning problems. In both cases, states amount to the same thing, and actions are in fact transitions. Finally, we only require to specify an initial state, which is the start situation in a planning problem, and define some method in order to determine if a goal condition has been met. A transition system is thus another way of representing the state space of a problem. One can solve planning problems modeled as transition system by exploring the transition system in search for a state that satisfies the goal condition of the planning problem. The result would then be the path taken during the exploration.

We have now given a technique in which we can represent the components for a planning problem. However, we still need to actually represent these components, i.e. define the environment of the problem. Intuitively, we need a way to formally represent how a state or action models some real-life state or action. In this work we will do this by using *graphs*. Every state has an associated graph which models its situation.

The use of graphs to represent states in a transition system is called a graph transition system (GTS). *Graph transition systems* extend traditional transition systems by associating a graph representation with states and representing transitions as graph transformations combined with graph morphisms. Intuitively, a state of a system is modeled as a graph and a transition to another state is modeled as a graph transformation. Planning problems modeled using graph transition systems are called graph

transformation planning problems. In this thesis, we use the underlying graph structure for modeling the planning problems as a basis for domain independent heuristics.

In this work we use the tool GROOVE[1] [3] to model and solve graph transformation planning problems. GROOVE is a graph transformation tool that offers modeling and model checking capabilities for graph production systems. A *production system* consists of a set of graph transformation rules and an initial graph, referred to as the start graph. A graph transformation system can be generated in GROOVE given a production system. GROOVE generates the state space of the system by iteratively applying all valid graph transformation rules to the graphs associated to known states. GROOVE currently supports some basic exploration strategies, without heuristics, for finding goal states in a graph transition system.

1.3 PROBLEM STATEMENT

In this research we work towards a framework in which different domain independent heuristics can be used to solve planning problems in the context of graph transition systems. These heuristics can be used to guide forward state space exploration, which is an approach for solving planning problems.

In the context of graph transformation planning problems and domain independent graph heuristics we formulate the following problem statement:

To develop heuristic approaches for the purpose of planning using graph transformations and implement these in GROOVE as a framework for solving planning problems.

1.3.1 RESEARCH QUESTIONS

To achieve the desired objective we propose the following research questions to augment the problem statement.

RQ1. How can we define and solve planning problems in the context of graph transition systems and what advantages and disadvantages does this provide?

RQ2. In which ways can we exploit the underlying structure and formalisms of graphs and graph transformations in a graph transition system to develop meaningful metrics to estimate distances between a state and a goal?

RQ3. How can we implement planning in GROOVE with sophisticated problem solving techniques supported by a framework of heuristics?

The *first* research question focuses on how planning fits within the paradigm of graph transition systems. We need to consider the aspects of a planning problem and how they can be modeled using graphs and graph transformations. Furthermore, we must look into techniques to solving planning problems modeled as graph transition

[1] http://groove.cs.utwente.nl/

systems. And finally, we have to consider what form the solution of such a problem takes. This research question is examined and answered in Chapter 2 of this work.

The *second* research question deals with the development of domain independent graph-based planning heuristics. We consider aspects of the underlying modeling paradigm (graph transition systems) and determine universal characteristics which can be used to define distance metrics. Furthermore, we consider an abstraction of the original problem as an approach to reduce the problem state space. The answer to this research question is a collection of heuristic schemes which, given a graph and a goal, can estimate the distance from the state associated with the graph to a state which satisfies the goal. This research question is answered in Chapter 3 of this work.

The *third* research question considers the implementation of planning as defined according to RQ1 and heuristics as defined for RQ2 in the graph transformation tool GROOVE. We aim to design and implement a simple and extensible framework in GROOVE for heuristic functions and extend the exploration capabilities in GROOVE to support informed search strategies. This research question is answered in Chapter 4 of this work.

In order to evaluate the effectiveness of the to be developed heuristics we will attempt to solve a variety of planning problems using the GROOVE implementation. In this way can measure and compare how different heuristic approaches are suited for different problems. Furthermore, we will compare our results to those presented in papers which also promote graph transformation based planning tools. Finally, we compare the results of our approach to planners developed for the established planning language PDDL.

1.4 OUTLINE

The remainder of this thesis is structured as follows:

Chapter 2 gives an introduction to the central concepts of this work. This includes formal definitions pertaining to graph transition systems and planning, as well as the establishment of planning and exploration in the context of graph transformations. Finally, in the chapter we provide an overview and example of the GROOVE tool.

Chapter 3 presents two domain independent graph heuristic approaches developed within this final project. In this chapter we give a theoretical definition of each approach and define concrete heuristic functions, this is followed by a practical example and concluded with a discussion of the approaches advantages and disadvantages.

Chapter 4 presents the development and implementation of planning functionality and a framework for heuristics in the GROOVE tool.

Chapter 5 presents a set of test problems we have used for evaluation and, furthermore, discusses some distinguishing features of these test problem. Following this we present the results of planning in GROOVE using the heuristic functions developed in this

work. Finally, we provide an evaluation of the results and compare these to the results of other work.

Chapter 6 gives an overview of work related to this thesis.

Chapter 7 gives the conclusions and final remarks of this works. It reiterates and summarizes the answers to the research questions, discusses the contributions of this work to the field of planning and heuristic and closes with possibilities for future work.

1.5 AVAILABILITY

It is possible to obtain the source and/or binary files of the implementation presented in work as well as the GROOVE files of the planning problems used for experimentation by directly contacting the author[2] or through the FMT research group at the University of Twente[3].

It is the hope that at some point the work presented in this thesis will be integrated into the main GROOVE tool.

[2] j.w.elsinga@student.utwente.nl
[3] http://fmt.cs.utwente.nl/contact/

CHAPTER 2

BACKGROUND

In Section 2.1 we provide a series of formal definitions of concepts concerning graph transition systems and graph transformation planning problems. These concepts are also intuitively described. Furthermore this section serves as an introduction to the basic concepts of state space exploration in Section 2.2 and planning in Section 2.3. In Section 2.4 the graph transformation tool GROOVE is discussed and some insight into the basics of modeling graph transition systems with the tool is given. Finally this chapter also introduces a running example system modeled in GROOVE, which will be used to demonstrate the possibilities of planning in GROOVE. The concepts discussed in this chapter should be seen as a reference point for the remainder of this work.

2.1 DEFINITIONS

We begin by setting up the concept of production systems, which consist of a set of graph transformation rules and an initial graph. The graph transformation rules can be applied to the initial graph and its successors to construct the graph transition system of the production system. The theory of these concepts is based on graphs and graph morphisms.

Notation 1. $\mathcal{L}$ denotes the universal set of all labels.

Definition 1 (Graph)**.** A graph $G = \langle V, E, src, tgt, lbl \rangle$ consists of a set of nodes V, a set of edges E, a source and target functions $src, tgt : E \to V$ and a label function $lbl : E \to \mathcal{L}$.

Notation 2. $\mathcal{G}$ denotes the universe of graphs. Components of a graph $G \in \mathcal{G}$ are often denoted V_G, E_G, etc.

In a graph, nodes may be labeled by a self-loop edge with a label. We thus partition $\mathcal{L}$ into the label set for nodes $\mathcal{L}_V$ and the label set for edges $\mathcal{L}_E$, such that $\mathcal{L}_V \subset \mathcal{L}$ and $\mathcal{L}_E = \mathcal{L} \setminus \mathcal{L}_V$, where $\mathcal{L}_V$ is only used for self-loops.

Definition 2 (Domain,Range,Image)**.** Let $f : X \to Y$ be the function. We say X is the domain of a function, also written as $dom(f)$, Y is the range of the function, also written as $rg(f)$, and $\{f(x) \mid x \in X\}$ is the image of the function, also written as $im(f)$.

Definition 3 (Graph Morphism)**.** A graph morphism $\psi : G_1 \to G_2$ between two graphs is a pair of mappings $\psi = (\psi_E, \psi_V)$ with $\psi_E : E_{G_1} \to E_{G_2}$ and $\psi_V : V_{G_1} \to V_{G_2}$ such that $\psi_V \circ src_{G_1} = src_{G_2} \circ \psi_E$, $\psi_V \circ tgt_{G_1} = tgt_{G_2} \circ \psi_E$ and $lbl_{G_1} = lbl_{G_2} \circ \psi_E$. We say ψ is partial if both ψ_E and ψ_V can be partial functions.

Graph morphisms are used in graph transformation rules to indicate which graph elements are created, destroyed or preserved in a graph transformation.

Definition 4 (Graph Transformation Rule)**.** A *simple* graph transformation rule $r = \langle L, R, p \rangle$ consists of two graphs L and R, called left-hand side (LHS) and right-hand side (RHS), and a partial graph morphism $p : L \to R$.

Notation 3. $\mathcal{R}$ is used to denote a set of rules.

Notation 4. L_r, R_r and p_r are used to denote the left-hand side graph L, right-hand side graph R and partial graph morphism p of a graph transformation rule $r = \langle L, R, p \rangle$.

The application of a rule r to a graph G requires finding a match $\mu : L \to G$, which typically is a morphism from L to G (note this assumes a rule of this simple form).

Notation 5. $\mathcal{M}$ is used to denote the set of matches.

The applicability of a rule r to a graph G thus depends on whether a match μ from L_r to G can be found. The application of rule is called a graph transformation.

Definition 5 (Graph Transformation)**.** Given a graph G, a rule $r = \langle L, R, p \rangle$ and a match $\mu : L \to G$, a graph transformation is a tuple $G \overset{r,\mu}{\Rightarrow} H$ where H is a graph uniquely determined by G, r and μ.

$$\begin{array}{ccc} L & \xrightarrow{\ p\ } & R \\ \downarrow{\scriptstyle \mu} & & \downarrow{\scriptstyle \mu'} \\ G & \xrightarrow{\ f\ } & H \end{array}$$

Figure 2.1.1: Graph Transformation

Intuitively, H is equal to G where an image of L, determined by μ, has been replaced by R. For the purpose of this thesis, it is not necessary to go into any detail about how this is done precisely. The relation between graphs and graph morphisms involved a in graph transformation are shown in Figure 2.1.1.

The applicability of a rule may be restricted by the uses of negative application conditions (NACs).

Definition 6 (Negative Application Condition, Satisfies)**.** Let $r = (L, R, p)$ be a graph transformation rule, G a graph and $\mu : L \to G$ a match. A negative application condition (NAC) is a graph morphism n with $dom(n) = L$. If there is no graph morphism $m : rg(n) \to G$ such that $m \circ n = \mu$, then μ satisfies NAC, this is written as $\mu \models n$.

A graph transformation for a given G, r and μ is only valid if $\mu \models n$ for all NACs n in r. The relation between all graphs and morphisms involved in determining the satisfiability of NACs are shown in Figure 2.1.2. The morphism m is shown as a dotted line since its own existence determines if $\mu \models n$.

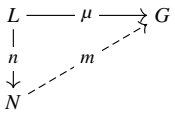


Figure 2.1.2: Satisfiability of a NAC

We now have all components necessary to define the concept of a graph transition system (GTS).

Definition 7 (Graph Transition System)**.** A graph transition system $S = \langle Q, A, \rightarrow, q_0 \rangle$ is an extension of a transition system where,

- Q is a set of states ranged over by q. Each state $q \in Q$ has a uniquely defined associated graph representation G_q;
- A, derived from $\mathcal{R} \times \mathcal{M}$, is the alphabet of S;
- $\rightarrow \subseteq Q \times A \times Q$ is a set of labeled transitions, elements of which are written as $q \xrightarrow{\alpha} q'$. Each transition $q \xrightarrow{\alpha} q'$ implies $G_q \overset{r,\mu}{\Rightarrow} G_{q'}$, where $\alpha = (r, \mu)$;
- $q_0 \in Q$ is an initial state.

Notation 6. The elements of the transition relation $\rightarrow$ in a graph transition system are called actions.

Graph transition systems are generated from so-called production systems.

Definition 8 (Production System)**.** A production system is a tuple $P = \langle \mathcal{R}, G_0 \rangle$ consisting of a set of graph transformation rules $\mathcal{R}$ and an initial graph G_0.

In addition to the production system we require the notion of a control program to generate the graph transition system S of the production system. A control program is a mechanism which schedules graph transformation rules. By applying the graph transformation rules $r \in \mathcal{R}$ of a production system to the initial graph G_0 and its resulting successor graphs, in combination with a control program, one is able to generate S. S is also known as the state space of the production system.

We have now defined all the concepts related to graphs, graph transformations and production systems. We use these concepts to define planning problems in the context of graph transformations. Graph transformation planning problems consist of a production system and a goal and results in a solution. These notions are defined below.

Definition 9 (Goal,Goal State)**.** A goal is a predicate ϕ_{goal} over graphs. A goal graph G_{goal} is a graph which satisfies ϕ_{goal}. A goal state q is a state for which G_ϕ is a goal graph.

Definition 10 (Path)**.** Given a transition system and two states q_s, q_t, a path is a finite sequence of successive actions from q_s to q_t.

Definition 11 (Graph Heuristic)**.** A graph heuristic is a function $H : \mathcal{G} \to \mathbb{R}_0^+$.

Definition 12 (Distance)**.** Given a transition system and two states p, q, the distance from p to q is the number of actions required to reach q from p.

A graph heuristic H typically represents an estimated distance measure between a graph $G \in \mathcal{G}$ and a predefined goal ϕ_H. We can estimate the distance from some state q to some goal state in a state space by using H to estimate the distance between the graph representation G_q and ϕ_H. The estimated distance from some state q to some goal is called the heuristic value of q.

Definition 13 (Graph Transformation Planning Problem, Solution)**.** A graph transformation planning problem consists of a production system P and a goal ϕ_{goal}. A solution for such a problem is a path in the state space generated from P, from q_0 to an arbitrary goal state q_{goal}.

2.2 EXPLORATION

In this section we explain and discuss the concept of state space exploration in the context on graph transformation systems and planning problems.

Given a production system, state space exploration is the process of systematically and iteratively calculating the successors (also known as expanding) of known states, thereby generating new states until a newly generated state satisfies some predefined condition or until there are no more new successor states. Exploration can thus be used to generate the graph transition system corresponding to a production system.

Definition 14 (Successor)**.** The successor set of a state q is the set of states Q' for which there exists an action α such that $q \xrightarrow{\alpha} q'$ for $q' \in Q'$.

In this work we consider exploration with input a production system and is parameterized by an acceptor that represents the condition on which exploration should be halted and a strategy that specifies how and in which order states are expanded. In Algorithm 1 we give a pseudo code algorithm which serves as the basis of a single iteration of state space exploration.

Algorithm 1: Single Exploration Iteration

```
1 select state q to expand;
2 if q fulfills acceptor then
3 |   break;
4 end
5 expand state q by calculating all its successors Q';
```

2.2.1 ACCEPTOR

The acceptor is a condition that is used to specify the aim of exploration. After a state q is selected to be expanded the acceptor condition is check in the exploration algorithm, if it is met (termed fulfilled) then exploration ends and otherwise continues. This is seen on lines 2 in Algorithm 1.

An acceptor is not stateless, or intuitively, the acceptor remembers previous iterations in exploration. An acceptor condition can range from a requirement on the state space (e.g. x number of states generated), to a requirement on the transitions outgoing from (or incoming to) a state (e.g. final state or certain transition possible) and finally a requirement on the representation of the state (e.g. a predicate over graphs, in the case each state q has a corresponding graph representation G_q).

It is also possible to set the acceptor unfulfillable. In this case exploration continues uninterrupted until the full graph transition system corresponding to the initial production system is generated. Furthermore, it is possible to specify how often an acceptor condition should be fulfilled before halting exploration. For example in the case of a final state acceptor one could specify that n number of final states should be found before halting.

2.2.2 STRATEGY

The exploration strategy, also known in literature as a search or traversal strategy, has two responsibilities within exploration. Both of these responsibilities pertain to the arrangement of generating successor states.

The first responsibility specifies whether all successor states of a state should be generated in a single exploration iteration or if a single successor state should be generated per iteration. For both cases there are advantages and disadvantages in terms of calculation time, size of explored subspace and implementation. For example, an advantage of the second approach is that only the minimum number of states are generated, the drawback however is that the algorithm needs to keep track of how far a certain state has been expanded. An extensive discussion of all benefits and disadvantages of both approaches falls outside the scope of this work. We have chosen to generate all successor states in a single iteration. This can be seen in the pseudo code of Algorithm 1 on line 5.

The second responsibility specifies the order in which states should be explored. There are many well known exploration strategies with a varied range of approaches to this specification. This is implemented on line 1 in Algorithm 1. For this work they can be divided into two categories, uniformed and informed strategies. Uninformed exploration can be classified by the fact that the algorithm is given no information about the problem other than its definition. This means that the algorithm does not know if one state is more promising than another state in terms of fulfilling the acceptor condition. Some well known uniformed strategies are breadth-first search, depth-first search and random search.

In contrast to uninformed exploration, informed exploration do use addition problem knowledge in order to find a solution. Having additional problem knowledge allows the algorithm to find solutions (i.e. states that fulfill the acceptor) more efficiently

than uninformed algorithm. Informed search algorithms follow the same general approach with the addition an informed method of choosing the next state to expand. Additional problem knowledge can come from two sources. The first (which is the one we consider in this work) is from the problem itself, specifically we will consider the underlying graph structure and graph transformation rule applicability to the graph G corresponding to a state q. The second source is problem context provided *in addition* to the problem specification. The additional information provided is processed by a function known as a heuristic function. As we are considering graph based transition systems we use graph heuristic functions defined in Definition 11. Given the two sources of knowledge we can define problem independent and problem dependent graph heuristics, respectively.

The heuristic makes an estimation on how near a state is to a state in which the acceptor condition is satisfied. The informed strategy then uses this information to decide which state should be expanded first. Some well known informed strategies are greedy best-first search, A* search and hill-climbing search.

The exact specification, advantages and implementations of each exploration strategy fall outside of the scope of this work. For a detailed explanation of exploration strategies see [4]. This work will focus on exploration using informed search algorithms, specifically the greedy best-first search strategy. Greedy best-first search simply defines that the state with the best heuristic value is expanded first. Note that the reason we only consider a single exploration strategy is that previous research in the field of graph transformation planning problems has shown that choosing different informed exploration strategies has only a minor effect on the planning result, in contrast to the choice of heuristic function [5, 6, 7].

2.3 PLANNING

Automated planning is a discipline within artificial intelligence which concerns finding a sequence of actions to achieve some goal. A planning problem is solved by using a planning strategy. The problem is defined in a planning language. Beside the problem definition it is possible to define additional problem context, which could speed up the problem solving time and reduce memory use. The information presented in this section is primarily taken from [4].

Planning problems are modeled by states, actions and goals. A planning strategy explores the planning state space in search of a goal state. The solution of a planning problem is then a sequence of actions which from a start state leads to a goal state. How the states, actions and goals are represented is called the language of the planning problem. In this project, we will represent planning problems using graph transformation systems. This means planning problems will be modeled using graphs and graph production systems. This allows us to solve planning problems using GROOVE. We refer to these as graph transformation planning problems.

What differentiates planning from straightforward problem solving is that the planning strategy has planning language (domain-independent) and possible problem context (domain-dependent) knowledge. This enables a strategy to apply heuristics to guide exploration and make educated decisions when searching for a solution.

In the following subsections, we will consider the implementation of planning problems as graph production systems. These sections make use of the definitions presented in Section 2.1. In Section 2.3.1, we will discuss the language in which these problems will be modeled. In Section 2.3.2, we discuss different planning approaches and consider which is most suited for graph transformation planning problems. In Section 2.3.3 we will consider what effect the definition of the goal has on the performance of solving planning problems. Finally, in Section 2.3.4, we will discuss what a solution to a graph transformation planning problem looks like.

2.3.1 PLANNING LANGUAGE

In the field of artificial intelligence and planning there already exist some well known planning languages used to represent planning problems. One such language is the STRIPS language [8] which was first presented in the 1970s. Another such language is PDDL [9] which attempts to standardize planning languages. The most current version of PDDL is v3.1.

In this research we will step away from the standard planning languages and use graph transformation to represent planning problems. This is not a new idea and has already been explored in literature. Edelkamp *et al.* show for the Gossiping Girl problem that GROOVE without any planning heuristics performs better than the heuristic search planner FF [11]. Furthermore a link between graph transformation systems and PDDL is presented by Meijer to transform planning problems in PDDL to graph transformation systems as implemented in GROOVE. Tichy *et al.* presents techniques to transform self-adaptive systems modeled as graph transformations to the PDDL language.

The advantage of using graph transformation as a basis for the planning language is the intuitive nature in which systems can be modeled using graphs and graph transformations. Furthermore, this thesis hopes to show how problem domain-independent knowledge obtained from graph transformation systems can be advantageous in determining a good heuristic. A reason why graphs and graph transformations are used to represent planning problems instead of well established languages such as PDDL is because PDDL does not allow for the dynamic creation of objects (in terms of graphs these would be nodes). Formal definitions of how planning problems are expressed using graphs and graph transformations are given in Section 2.1.

2.3.2 PLANNING STRATEGY

In this section we introduce different approaches one can take to solving planning problems. A planning strategy consists of an algorithm to systematically work towards finding a solution.

The three planning well known strategies are: forward state space search, backward state space search and partial-order planning. In *forward state space search*, also known as progression planning, the algorithm starts in the initial state and considers sequences of actions until it finds a sequence that reaches a goal state. *Backward state space search*, also known as regression planning, starts from a goal state and then by applying the inverse of actions looks for a sequence which reaches a start state. Finally

partial-order planning is an approach which allows decomposition of the problem. The algorithm delays the decision of ordering actions until it becomes absolutely necessary. This allows the algorithm to create subgoals and solve these independently without worrying about the order in which actions are applied.

In this work we will implement graph transformation planning problems with progression planning. Specifically this corresponds to the exploration as described in Section 2.2. As mentioned in the section on exploration we will use the greedy best-first search exploration strategy, and as acceptor we use goal conditions as defined in the following section.

For a more complete overview of why a progression planning approach is chosen to solve graph transformation planning problems see the research topics related to this work.

2.3.3 GOAL

In Definition 9 and Section 2.3.1 we defined a goal as a predicate over graphs. This is a very broad definition for a goal with respect to graph transition systems. Given such a broad definition we are able to define a wide scope of different goals which is very desirable. However, using graph heuristics which estimate some distance from the current state to a goal based on the underlying graph structure, it is required to make some tangible comparisons between a graph and a goal. For this reason, although a goal may be very general and minimalistic, we believe a goal must be detailed enough to make heuristic distance estimations relevant.

We say there is a hierarchy of approaches of formulating a goal. The hierarchy is as follows:

1. **Predicate over graphs** – ϕ_{goal}, The general definition of a goal.
2. **Partial negation graph** – $\phi_{goal} = (N, \{NAC\})$. In this case a goal is formulated using two components, a graph N for which there exists a graph morphism mapping N to G_{goal} and NACs $\{NAC\}$ which express elements that may not be present in a G_{goal}.
3. **Partial graph** – $\phi_{goal} = P$. In this case a goal is formulated using a graph P for which there exists a graph morphism mapping P to G_{goal}.
4. **Complete graph** – $\phi_{goal} = C$. In this case a goal is formulated using a graph C for which there exists a graph isomorphism mapping C to G_{goal}. An example of this would be Figure 2.4.2c.

Each goal representation in the hierarchy can be expressed as a predicate over graphs. However, $2-4$ only represent a subset of all possible goal formulations that are possible using 1. Furthermore, goal representation 4 has the explicit additional requirement that a goal state exactly match the goal (i.e. there may be no additional node or edge elements present in a goal state), this is not case for the other goal representation. Additionally, from the first and second level in the hierarchy the representation of a goal shifts from a logical expression towards graphs. It is possible that for a goal multiple goal states exist. Furthermore different formulations of a goal do not only affect the expressiveness of a goal but also the effectiveness and complexity of heuristics.

We discuss different types of graph transformation heuristic approaches and how well they are suited for certain goal formulations in Chapter 3.

2.3.4 SOLUTION

In Definition 13 we define a solution as a path from the start start to a goal state. A path is defined in Definition 10 as a sequence of actions from one state to another state. A path can be determined once a goal state has been discovered. The planning strategy can then from the goal state backtrack to the start state (by using the record information about parent states) and record the actions. These actions are a combination of a transformation rule and a matching from the graph of the source state to the graph of the target state.

It is possible that a goal state is never discovered. This may occur in two different scenarios. The first is that the goal is simply never satisfied in any state in the whole state space. In this case the algorithm would not return a path as solution but something to the effect that the a goal is not reachable. The second scenario is an infinite state space in which the goal has yet to be discovered. In this case the planning algorithm would continue to explore the state space indefinitely until reaching some predefined state or time threshold, or until it is stopped manually. The resulting solution would again be something to the effect that no goal state been found and thus no path from the start state to a goal state.

2.4 GROOVE

For this thesis, we will use the graph transformation tool GROOVE in which we will implement algorithms to solve graph transformation planning problems. In this section we will introduce GROOVE and explain some of its modeling capabilities as well as give an example of how a puzzle system is modeled in GROOVE.

GROOVE stands for GRaph-based Object-Oriented VErification, and is a tool originally developed for software model checking of object-oriented systems. It is set apart from other model checking approaches by the fact that it is based on graph transformations. GROOVE uses graphs to represent snapshots of a state, and transitions arise from applying graph transformations to a graph. This process results in graph transition systems (see Definition 7) as computational models.

In GROOVE graphs nodes can have a type, flags and attributes. Flags and types are matched to a node using self-edges with specific labels from the label set $\mathcal{L}_V$. GROOVE offers a graphical interface to model production systems (defined in Definition 8). This consists of four parts, a type graph, a start graph, transition rules and a control program. The control program has previously been introduced in Section 2.1 and is henceforth considered outside the scope of this work. The remaining elements are discussed below.

Type Graph
A type graph is a graph which specifies the allowed structure as well as node hierarchy

of the graphs and can be considered similar to a class diagram. The exact details of type graphs are not important to this thesis.

Start Graph
The start graph, also known as a host graph, is a graph (as defined in Definition 1) which models the start state of the system. An example of a start graph can be seen in Figure 2.4.2b.

Transition Rules
A production system in GROOVE is a set of graph transition rules which can be graphically modeled in the Simulator component. In GROOVE a rule as introduced in Section 2.1 is modeled as a graph that specifies how a host graph should be transformed. Using colors and line thickness the left-hand and right-hand side graph L and R, as well as NACs, are visually represented as single graph. Examples of graph transition rules can be seen in Figure 2.4.1.

There are four element types (nodes, edges or attributes) in GROOVE graph transition rules. These are the reader, the embargo, the creator and the eraser.
Readers are elements that are required to be present in L and remain present in R. These elements are represented in black and have a normal line thickness.
Embargoes are elements that are part of a NAC and thus should not be present in L. Embargoes are represented in red with a thicker dotted line.
Creators are elements which are not present in L but are created in the R. Creator elements are represented using thick green lines.
Erasers are elements which are present in L but not in R, these elements are thus deleted in the transformation. These elements are represented by blue dashed lines.

In GROOVE it is also possible to define conditionals as rules. These are rules where the RHS is equal to the LHS. Such a condition as rule, while modeled graphically in GROOVE, is actually a predicate over graphs and as such is a goal as defined in Definition 9. We can thus write graph transformation planning problems (see Definition 13) using GROOVE.

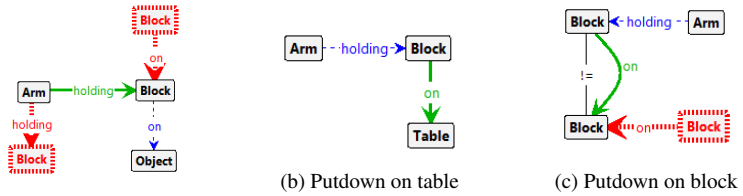
GROOVE currently supports some straightforward planning problem solving capabilities. These exists in the form of a state space exploration strategy such as breadth-first or depth-first search, together with a so-called acceptor. An acceptor can be considered as a goal (Definition 9) with respect to graph transformation planning problems (Definition 13) and can be represented in several forms within GROOVE. If no acceptor, is given the whole state space is explored. During exploration, each discovered state is matched to the acceptor. If they match, then the goal state is reached and a path can be given.

Next we introduce a GROOVE example which shows the basic GROOVE modeling and planning capabilities. The example is given as a production system, and is only a small system which illustrates the possibilities of GROOVE in terms of state space generation. In addition, a goal graph is given to create a graph transformation planning problem the example.

2.4.1 BLOCK WORLD

Block World is a system which exists exclusively of blocks, a table and an arm. The type graph which expresses these objects as nodes and their relations as edges is given in Figure 2.4.2a. The blocks can be set on the table or on another block by the arm, which can pick up and put down blocks. In case of stacking blocks, the bottom block must always be on the table (no floating stacks) and only one block may be on top of any given block. Figure 2.4.1a, 2.4.1b and 2.4.1c show these rules modeled in GROOVE.

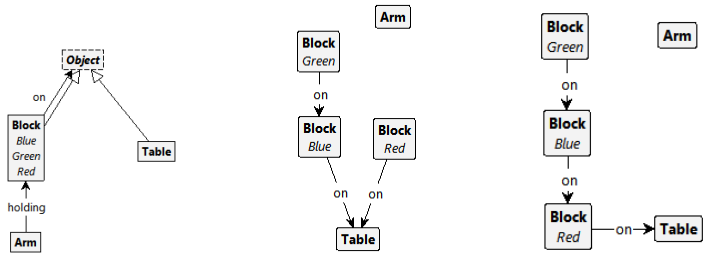
The size of the Block World puzzle is dependent on the number of blocks in the world. We also introduce block colors to create unique groups of blocks. In this example a size of 3 blocks with 3 different colors is chosen. Figure 2.4.2b gives the graph representation of a possible start state of the Block World system. This in combination with the transition rules represents a production system of a 3 block Block World puzzle.



(a) Pickup block rule (b) Putdown on table rule (c) Putdown on block rule

Figure 2.4.1: Transformation rules for Block World problem

Figure 2.4.2c shows the graph representation of a goal graph we have for the Block World example. The goal in this case is predicate over graphs which expresses that all blocks are stacked with the Red block on the bottom followed by the Blue block and finally on top the Green block, and an empty arm.



(a) Type graph (b) Start graph (c) Goal graph

Figure 2.4.2: Type, Start and Goal graph for 3-block Block World problem

Figure 2.4.3 shows the explored state space of the 3-block Block World problem, generated by applying the exploration of production system. This is done in a breath-first approach were each applicable rule is first applied to the original graph and then is applied to each following node. We see that the state $s19$ is labeled with `goal` (the

name of the conditional rule expressing the goal graph) which indicates that $s19$ is a goal state. In this small example one can manually determine a path from the start state $s0$ to the goal state $s19$. One should note that each action in the state space of Figure 2.4.3 is labeled with only the rule corresponding to the action taken. In fact each action shown is a combination of a rule and a match, as defined in Definition 7. However, the match is not shown in the visual representation of the state space.

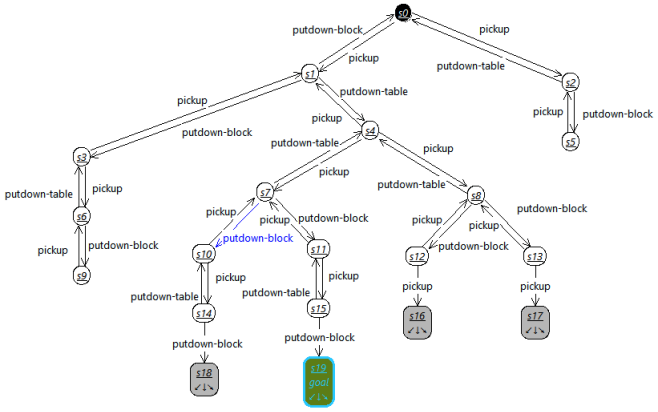


Figure 2.4.3: Complete state-space for 3-block Block World problem

The small state space of the 3-block Block World system already provides an insight into the issues of finding a solution to graph transformation planning problem. The major issue is the size of the state space. In this example the problem is so small that solution can be found simply by looking at the problem. However, this will no longer be the case if the size of the problem grows. Yet even for such a small problem with only 22 states, when using using breadth-first exploration, the goal state is the 19th state to be discovered. We can already see how state space explosion can occur. The worst-case complexity of the puzzle is in the order $O(n! * 2^n)$ where n is the number of blocks (this complexity is due to the order in the stacking of blocks and the combinations in which sets of colored blocks are stacked). This shows how fast the state space can grow and that a sophisticated exploration strategy is an important aspect in quickly discovering a goal state.

CHAPTER 3

HEURISTICS

In this chapter lies the main focus of this thesis. In Section 3.1 we give an introduction to the concept of heuristics and their use within exploration strategies given in Section 2.1 and 2.2. The heuristics presented in this chapter are all original work of the author.

In Section 3.2 and 3.3 we define graph heuristic approaches based on graph element abstraction and graph transition abstraction, respectively. From these approaches we show how specific graph heuristic schemes can be defined which can be used to generate graph heuristic functions as defined in Definition 11. Furthermore, in both Section 3.2 and 3.3 we provide examples of how such a heuristic function is generated and how they may be used to calculate the heuristic value of a state. Finally, we provide a discussion of the advantages and disadvantages for each approach and hypothesize on how well specific heuristic schemes will perform in solving planning problems for different problem types.

3.1 INTRODUCTION

Heuristics come down to using additional problem knowledge to make an estimation of how close a state is to satisfying a goal. Furthermore, exploration strategy may be supplemented by a heuristic in order to improve state space exploration. In the introduction to Section 2.3, we say a strategy may have domain-independent and domain-dependent knowledge, and a heuristic may make use of either of these sources. In this research effort we consider heuristics functions based on domain-independent knowledge. We aim to develop heuristics which make use of the planning language, specifically graphs and graph transformations. This allows us to develop heuristics which are applicable to a wide range of problems modeled as graph transition systems.

Two important properties of heuristic functions are *admissibility* and *consistency* [4]. Admissible means that a heuristic never overestimates the distance to reach the goal or, in other words the heuristic distance to reach the goal is not higher than the actual distance to reach the goal. Consistent means that for every state q and every successor state q' of q, the estimated distance of reaching the goal from q is no greater

than the distance of getting to q' and the estimated distance of reaching the goal from q'. If a function is admissible and consistent then we can guarantee an optimal solution. Due to flexibility of graph transformations (i.e. a single rule may add/remove several elements) it may be very difficult to specify heuristics which are admissible and consistent. Therefore, while we acknowledge these properties we will not attempt to satisfy these requirements in order to make define a heuristic which leads to an optimal solution. This may be considered a limitation of this work.

In Section 2.1 we introduced the concept of a graph heuristic as a function $H : \mathcal{G} \to \mathbb{R}_0^+$. In this section we will discuss possible implementations of this function. The goal to which $H(G)$ makes a distance estimation is in a form discussed in Section 2.3.3. To do this we introduce three levels of implementation with respect to graph heuristics. These are; **a)** a heuristic approach, which specifies the knowledge domain the heuristic is based on, **b)** a heuristic scheme, which specifies how an heuristic approach may be implemented for a specific goal type, and **c)** a heuristic function, which is a function generated from a heuristic scheme and a goal.

A graph heuristic can be seen in two flavors. The first is an abstraction of the actions, and the second is an abstraction of the graphs associated with states. In Section 3.2 we describe the NENTUPLE (or NEN) approach which is based on the abstraction of graphs and in Section 3.3 we describe the Linearization Abstraction (or LA) approach which is based on the abstraction of actions. For both of these approaches we define heuristic schemes for all suitable goal types.

Finally, for informed exploration of a production system the heuristic value (see Section 2.1) of a state q is determined by calculating the value of the heuristic function, used by the exploration strategy, for the graph G_q.

3.2 NENTUPLES

NENTUPLES is a graph heuristic approach which is based on graph abstraction and element counting. This approach is suited for heuristics with the goal type in the form of a graph. We refer to this graph heuristic scheme as HS_{NEN}. Concretely, we define two schemes HS^P_{NEN} and HS^C_{NEN} using the NENTUPLE approach, these require a goal of the type partial graph and complete graph respectively. Intuitively these schemes create a decomposition of $\langle node, edge, node\rangle$ triples in both G and ϕ_H and then compare the similarity of these decompositions to estimate the distance from G to ϕ_H.

In Section 3.2.1 we will give the mathematical definitions of the decomposition of G into NENTUPLES, HS^P_{NEN} and HS^C_{NEN}. After that we will, in Section 3.2.2, give a practical example of how a graph modeled in GROOVE is decomposed into NENTUPLES and how graph heuristics H^P_{NEN} and H^C_{NEN} are calculated. Finally we will discuss the advantages and disadvantages of this approach as well as possible extensions in Section 3.2.3.

3.2.1 THEORY

3.2.1.1 ABSTRACTION: EDGES AND NODE OBJECTS

For this approach we define the triple $\langle node,edge,node \rangle$, called a NENTUPLE, which represents pairs of node objects which are connected by an edge object. An edge object is simply the label of the edge, given by $lbl(e)$. A node object $[v]$ of a graph G is the set of node labels $\mathcal{L}_V$ for a node $v \in V_G$. Given a graph G we define Equation (3.2.1) to determine $[v]_G$ for a given node element $v \in V_G$.

$$[v]_G = \{lbl(e) \mid e \in E_G, src(e) = tgt(e) = v, lbl(e) \in \mathcal{L}_V\} \tag{3.2.1}$$

Subsequently, we define N_G which is the set of all node objects $[v]$ of G.

$$N_G = \{[v]_G \mid v \in V_G\} \tag{3.2.2}$$

These edge and node objects are an abstraction from their original respective elements in G. Combined in the form of a NENTUPLE we create a second level of abstraction. Intuitively in a NENTUPLE abstraction, a graph G is represented by NENTUPLES instead of its original form given in Definition 1.

3.2.1.2 DECOMPOSITION

Given the definition of NENTUPLES we define the function $\text{NEN}(G)$ which, given a graph G, calculates the multiset of all NENTUPLES in G. Note that a multiset, also commonly referred to as a bag, is a set in which may contain multiple instances of the same element, and we denote a multiset using square brackets.

$$\text{NEN}(G) = \left[\left\langle [src(e)], lbl(e), [tgt(e)] \right\rangle \mid e \in E_G\right] \tag{3.2.3}$$

We call the determining of the multiset of NENTUPLES of G the decomposition of G. The decomposition is an abstract representation of a graph and is used to calculate the similarity (thus estimated distance) between two graphs.

3.2.1.3 NENTUPLE HEURISTIC SCHEMES

We use decomposition to define the graph heuristic schemes HS^P_{NEN} and HS^C_{NEN}. Given a goal parameter ϕ_H of the correct goal type, we can generate the heuristic functions which have ϕ_H as goal parameter. Intuitively, heuristic functions generated from HS_{NEN} estimate the distance between G to a goal ϕ_H by calculating the distance between the decomposition of both G and ϕ_H. Exactly how this distance calculation is made differs for each heuristic scheme.

The heuristic scheme HS^P_{NEN} requires a goal ϕ_p of the type partial graph. If the goal is partial graph we are only concerned with the distance from G to ϕ_p. Thus, the distance between G and ϕ_p is calculated by taking the cardinality of the multiset consisting of the NENTUPLES in ϕ_p and not in G. Given a goal ϕ_p of the type partial graph, we thus define HS^P_{NEN} as shown in Equation (3.2.4).

$$HS^P_{NEN}(\phi_p) : G \mapsto |\left[\text{NEN}(\phi_p) - \text{NEN}(G)\right]| \tag{3.2.4}$$

The heuristic scheme HS^C_{NEN} requires a goal ϕ_c of the type complete goal. If the goal is a complete graph then we should consider the absolute distance from G to ϕ_c. This calculation is achieved by taking the cardinality of the symmetric difference between the decompositions of ϕ_c and G. Given a goal ϕ_c of the type complete graph, we thus define HS^C_{NEN} as shown in Equation (3.2.5).

$$HS^C_{NEN}(\phi_c) : G \mapsto |\big[\textsc{Nen}(\phi_c) - \textsc{Nen}(G)\big] + \big[\textsc{Nen}(G) - \textsc{Nen}(\phi_c)\big]| \tag{3.2.5}$$

Intuitively, we consider the absolute distance between two graphs as the total difference between them. Thus, in terms of NENTUPLES heuristics we can calculate this as the number of tuples which are present in one graph but not the other. This also underlines the major difference between the heuristic scheme for complete graph and for partial graph goals. With partial graph goals we are not interested in additional NENTUPLES in a graph that are not part of the goal. This is due to the fact that the goal is only a partial representation of a goal state. In the scenario of complete goal graphs this is not the case, and thus we must consider to complete similarity between graphs in terms of NENTUPLES.

3.2.2 BLOCK WORLD EXAMPLE

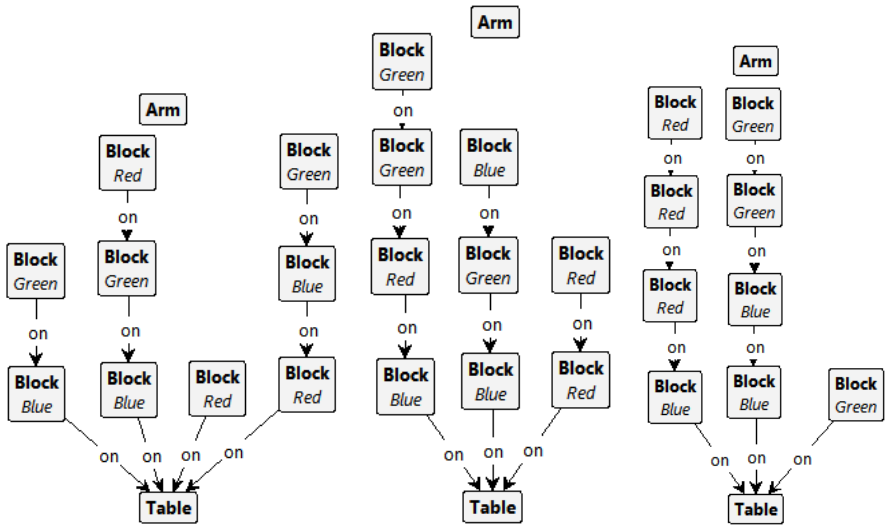
In this section we will give examples of decomposition of graphs G and how we can use the heuristic schemes of HS_{NEN} to generate heuristic functions. In this section we will make use of the Block World problem. A full description of this problem and how it is modeled in GROOVE is given in Section 2.4.1. We extend the problem size from 3 blocks in Section 2.4.1 to 9 blocks, with 3 blocks of each color (*Red,Blue* and *Green*).

Given a start graph with 9 blocks and the graph transformation rules shown in Figure 2.4.1 we define a 9-block Block World production system P_{BW}. From this production system we can theoretically generate (the extremely large) state space S_{BW} of the 9-block Block World problem. In Figure 3.2.1 we give the graph representation G_1, G_2 and G_3 of three possible states in S_{BW}.

For the 9-block Block World problem we can define a goal ϕ for which we want to find a corresponding goal state G_{goal} in S_{BW}. In this example we introduce two goals, one of type complete graph and the other of type partial graph. These two goals, ϕ_C and ϕ_P respectively, are shown in Figure 3.2.2. In the case of ϕ_C the goal state is exactly a state in which all blocks are stacked on the table sorted by color. In the case of ϕ_P the goal state is any state in which all red blocks are stacked.

Given P_{BW} and the two goals ϕ_C and ϕ_P, we define two graph transformation planning problems for the 9-block Block World. To solve these planning problems in this example we use an exploration strategy which makes use of heuristic functions based on NENTUPLES. Therefore, we give a heuristic function for the exploration strategy for each planning problem which are generated from the heuristic schemes HS^C_{NEN} and HS^P_{NEN}.

First we will generate the heuristic functions for goals ϕ_C and ϕ_P (given in Figure 3.2.2) using heuristic schemes HS^C_{NEN} and HS^P_{NEN} respectively. Following this we



(a) Graph G_1 (b) Graph G_2 (c) Graph G_3

Figure 3.2.1: GROOVE graph representation of states in 9-block Block World state space S

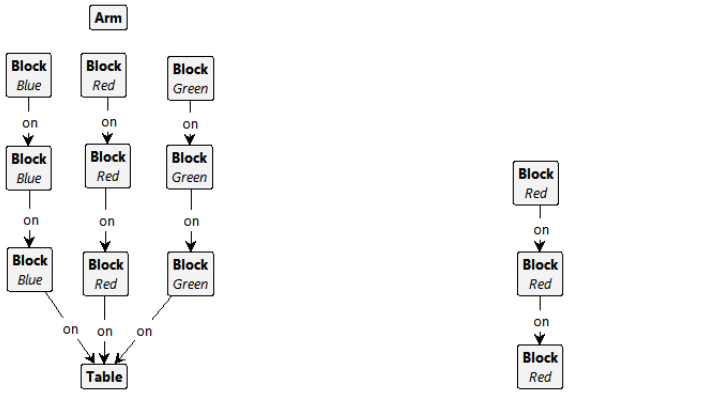
consider the decomposition of graphs in the Block World problem. Specifically we will deduce all possible NENTUPLES in a 9-block instance of Block World. Furthermore, we calculate the decomposition of G_1, G_2, G_3, ϕ_C and ϕ_P. Finally, using these results we calculate the heuristic values for the states in S_{BW} corresponding to G_1, G_2 and G_3 using $HS^C_{NEN}(\phi_C)(G)$ and $HS^P_{NEN}(\phi_P)(G)$ to give an indication of how this is done in exploration strategies.

Next we consider NENTUPLES in the Block World problem. In order to do this we need to abstract the edges and nodes in the graph representations of a Block World problem to edge objects, $lbl(e)$, and node objects, $[v]$. We do this by evaluating the type graph of the problem, which can be found in Figure 2.4.2a.

In this example we textually represent the node objects N as:

$$\{\textbf{Arm, Table}, \textit{Red, Blue, Green}\} \tag{3.2.6}$$

All NENTUPLES possible in this instance of the Block World problem are given in Table 3.2.1.



(a) Complete graph goal ϕ_C for a 9-block Block World problem

(b) Partial graph goal ϕ_P for a 9-block Block World problem

Figure 3.2.2: Complete graph and partial graph goal for 9-block Block World problem

Table 3.2.1: All possible NENTUPLES of Block World with 3 colors

$\langle Red, \text{on}, Red\rangle$	$\langle Blue, \text{on}, Red\rangle$	$\langle Green, \text{on}, Red\rangle$
$\langle Red, \text{on}, Blue\rangle$	$\langle Blue, \text{on}, Blue\rangle$	$\langle Green, \text{on}, Blue\rangle$
$\langle Red, \text{on}, Green\rangle$	$\langle Blue, \text{on}, Green\rangle$	$\langle Green, \text{on}, Green\rangle$
$\langle Red, \text{on}, \textbf{Table}\rangle$	$\langle Blue, \text{on}, \textbf{Table}\rangle$	$\langle Green, \text{on}, \textbf{Table}\rangle$
$\langle \textbf{Arm}, \text{holding}, Red\rangle$	$\langle \textbf{Arm}, \text{holding}, Blue\rangle$	$\langle \textbf{Arm}, \text{holding}, Green\rangle$

We can calculate decompositions for each of the graphs given in Figure 3.2.1. Recall $\text{NEN}(G)$ is the multiset of all NENTUPLES in G. Thus, the same tuple may be present more than once in the decomposition multiset of a graph. In Equation (3.2.7), (3.2.8) and (3.2.9) we give the decomposition of graphs G_1, G_2 and G_3 respectively. Note that, for the sake of readability we have further shortened node object representation to the first letter of a nodes uniquely identifying label.

$$\begin{aligned}\text{NEN}(G_1) = [&\langle R, \text{on}, G\rangle, \langle R, \text{on}, \mathbf{T}\rangle^2, \langle B, \text{on}, R\rangle, \\ &\langle B, \text{on}, \mathbf{T}\rangle^2, \langle G, \text{on}, B\rangle^3]\end{aligned} \tag{3.2.7}$$

$$\begin{aligned}\text{NEN}(G_2) = [&\langle R, \text{on}, R\rangle, \langle R, \text{on}, B\rangle, \langle R, \text{on}, T\rangle, \\ &\langle G, \text{on}, R\rangle, \langle G, \text{on}, B\rangle, \langle G, \text{on}, G\rangle, \\ &\langle B, \text{on}, G\rangle, \langle B, \text{on}, \mathbf{T}\rangle^2]\end{aligned} \tag{3.2.8}$$

$$\begin{aligned}\textsc{Nen}(G_3) = [&\langle R,\text{on},R\rangle^2, \langle R,\text{on},B\rangle, \langle B,\text{on},B\rangle,\\ &\langle B,\text{on},\mathbf{T}\rangle^2, \langle G,\text{on},B\rangle, \langle G,\text{on},G\rangle,\\ &\langle G,\text{on},\mathbf{T}\rangle]\end{aligned} \tag{3.2.9}$$

We can see that each decomposition has 9 NENTUPLES, which is to be expected as each G_q has 9 edges.

Next we calculate the decompositions of the goals as shown in Figure 3.2.2. In Equation (3.2.10) and (3.2.11) the decomposition of the goals ϕ_C, ϕ_P respectively are given.

$$\begin{aligned}\textsc{Nen}(\phi_C) = [&\langle R,\text{on},R\rangle^2, \langle R,\text{on},\mathbf{T}\rangle,\\ &\langle B,\text{on},B\rangle^2, \langle B,\text{on},\mathbf{T}\rangle,\\ &\langle G,\text{on},G\rangle^2, \langle G,\text{on},\mathbf{T}\rangle]\end{aligned} \tag{3.2.10}$$

$$\textsc{Nen}(\phi_P) = [\langle R,\text{on},R\rangle^2] \tag{3.2.11}$$

Finally, given the decompostions of each graph G_1, G_2, G_3 and both goals ϕ_C, ϕ_P, we can calculate the heuristic values of the states $q \in S_Q$ corresponding to G_1, G_2 and G_3 in the case of both heuristic functions. These values are given in Table 3.2.2.

Table 3.2.2: Heuristic values of graphs from Figure 3.2.1 for heuristic functions generated from heuristic schemes in Equation (3.2.5) and (3.2.4) for goals ϕ_P and ϕ_C respectively.

	G_1	G_2	G_3
$HS^C_{NEN}(\phi_C)(G)$	14	10	6
$HS^P_{NEN}(\phi_P)(G)$	2	1	0

These values give an indication of the similarity between the graphs G_1, G_2, G_3 and the goal graph ϕ_P and ϕ_C. Intuitively, the NENTUPLE heuristic approach assumes the more similar two graphs, the shorter the distance between them. For complete graph goals the actual heuristic value is the number of NENTUPLES that both the input graph and the goal graph do not have in common. For partial graph goals the value is the number of NENTUPLES the goal graph has but the input graph does not. We see that in both goal cases G_3 is determined to the nearest to the goals; in the case of the partial graph goal ϕ_P, G_3 is even the graph representation of a goal state.

During exploration using greedy best-first search, such heuristic values are calculated for every state not yet expanded. The exploration algorithm then selects the state with the lowest corresponding heuristic value to expand next. In this, for the goal ϕ_C the algorithm would expand the state corresponding to G_3 next. For the goal ϕ_P the algorithm would have terminated at the point the state corresponding to G_3 was generated.

3.2.3 DISCUSSION

The aim of this section is to discuss the advantages and disadvantages of the NENTUPLE heuristic approach as a basis for graph heuristics. Furthermore, we discuss some current limitations to the approach as presented in Section 3.2.1 and describe a possible extension of the work.

3.2.3.1 ADVANTAGES

The main advantage that NENTUPLES provide is a relatively low-time-cost real-time pruning of the subspace during exploration. Exploration guarantees that, without an acceptor, eventually the whole state space of a production system is generated. The heuristics based on NENTUPLES serves as a guide showing which states are expected to be more promising in leading to an acceptor. In this way we achieve a pruning of the subspace (because we expand the most promising states first). The abstraction of a graph (associated with a state) and comparing NEN tuples such as in HS^{P}_{NEN} and HS^{C}_{NEN} are computationally relatively cheap in terms of time compared to generating additional states. This trade-off allows for a slight increase in exploration time in exchange for a significant reduction in explored subspace.

The second advantage of the NENTUPLE approach is that it is extensible. Concretely we define the $\langle node,edge,node \rangle$ triple in Section 3.2.1 however, by expanding or fine tuning exactly what such a tuple represents and contains it may be possible to get a more accurate abstraction of a graph. Intuitively, we can change the granularity of the abstraction to more accurately abstractly represent a graph. For example coarser abstraction could be tuples only representing only labels. Conversely, a finer abstraction could be tuples representing not a single edge with its source and target node but also all the incoming or outgoing edges of those nodes. In this sense we scale how much of the graph context is encompassed by a single tuple. Thus, using a finer level of abstraction gives provides a more accurate representation of the original graph. The trade-off at this point however is between calculating and comparing the abstractions verse the accuracy of the abstraction.

3.2.3.2 DISADVANTAGES

The main drawback of the NENTUPLE approach is the goal type it requires. NENTUPLE heuristics determine distance from a goal by comparing how closely the graph abstaction of a state compares to the graph abstraction of goal. Therefore, the goal is required to also be in the form of a graph. This limits the applicability of the approach for graph transformation planning problems in general. A second disadvantage, which follows from the first, is that in order to make an accurate comparison the goal (assumed in the form of graph at this point), there must be enough information present in the goal graph abstraction. Therefore, a goal should be as descriptive as possible to make this approach effective. This occurrence is expanded upon in Section 5.1.4. Both of these disadvantages limit the use of the NENTUPLE heuristics depending on the goal of a planning problem.

A final disadvantage is that comparisons between the NENTUPLE abstractions of graphs are local. Intuitively, this means that while both graphs may have the same

NENTUPLE the abstraction says nothing about if these are the same elements in the actual graphs. Thus, for the heuristic it may seem that two graphs are similar while in fact some crucial elements are either not in the graph or in the wrong location. The effect of this is that a NENTuple heuristic may lead exploration down a false path, possibly for some time before realizing its mistake. An example of this can be seen in the Block World example given previously. For the heuristic function $HS^{C}_{NEN}(\phi_C)(G)$ we see that for input G_3 that both graphs have the 3 red blocks stacked on top of each other. However in G_3 these are stacked on another block while the goal expects them to be on the table. We know that several operations are required to correct this however, the heuristic does not have this context. So, in the eyes of the heuristic function the stack of red blocks looks like its almost reached the goal.

3.2.3.3 LIMITATIONS AND EXTENSION

Apart from the limitation that the goal has to be in the form of a graph, there is currently at least one other theoretical limitation to the NENTUPLE approach. This is the abstraction of node objects which do not have any incoming or outgoing edges. Since these nodes are not captured by the abstraction in any way, they are not considered by heuristic and thus may impact its effectiveness. This limitation again has to do with the granularity of the graph abstraction previously mentioned.

Furthermore, there are also some limitation in the implementation of both the abstraction of graphs to NEN tuples and the comparison between NENTUPLES used in heuristic schemes within GROOVE. The two main implementation limitations concern two advanced graph modeling techniques in GROOVE which are beyond the definitions of graphs and simple graph transformation rules given in Section 2.1. However, for the sake of completeness they are mentioned. Both inheritance and nested structures in graphs are currently not supported in the implementation of the NENTUPLE approach in GROOVE.

Currently there is one goal type in the form of a graph that is not supported by a NENTUPLE heuristic scheme. Defining a heuristic scheme for a goal in the form of a partial negation graph would be a relevant extension to the current work.

3.3 LINEARIZATION ABSTRACTION

The concept of the linearization abstraction approach consists of three parts.

The first part is the definition of an abstract graph which is an extension of a graph and keeps track of elements that may have been created or deleted after a graph transformation. The abstracted graph thus in a sense keeps a history of the result of graph transformations applied to it and its predecessors.

The second part consists of creating a relaxation of the graph transformation planning problem. This is done by creating an abstraction of the graph transition system.

The third part is the parallel application of applicable abstract rules and corresponding matches to an abstract graph. This in a sense creates a linear state space of abstract graphs.

We use the linearization abstraction approach to calculate the abstract linear state space of a graph. Linearization abstraction is thus an algorithm which requires a graph G, rules set $\mathcal{R}$ and a WITNESS function which determines which elements (*witnesses*) of an abstract graph satisfy a goal ϕ. During the calculation of the abstract linear state space of G we implement bookkeeping functionality which keeps track of size of the state space and what abstract rules are applied to what abstract graph elements for each parallel graph transformation. The result of linearization abstraction is thus the bookkeeping variables and the sets of *witnesses*.

We define three distance metrics, one naive, the other two more sophisticated, based on the results of linearization abstraction. These serve as the basis of the heuristic functions of linearization abstraction approach. We will give six heuristic schemes which are a combination different goal types and distance metrics. We refer to heuristic schemes based on linearization abstraction in general as HS_{LA}. Given such a heuristic scheme and goal of the suitable type we can generate a heuristic function.

We will first formally define the linearization abstraction heuristics scheme HS_{LA} in Section 3.3.1. Following this in Section 3.3.2 we give an example of a heuristic function generated from a linear abstraction heuristic scheme. Finally we will discuss the expected advantages and disadvantages of linearization abstraction heuristic schemes in Section 3.3.3.

3.3.1 THEORY

3.3.1.1 GRAPH ABSTRACTION

Definition 15 (Abstract Graph)**.** An abstract graph $\hat{G} = \langle V, \hat{V}, E, \hat{E}, src, tgt, lbl \rangle$, is an extension of a graph, where $\hat{V} \subseteq V$ and $\hat{E} \subseteq E$ represent sets of graph elements that may have been created or deleted after a graph transformation, and $src(e) \in \hat{V} \Rightarrow e \in \hat{E}$ and $tgt(e) \in \hat{V} \Rightarrow e \in \hat{E}$.

We refer to V, E as the *total* sets, to $\hat{V}, \hat{E}$ as the *maybe* sets, and to $V \setminus \hat{V}, E \setminus \hat{E}$ as the *real* sets. These names correspond to what information is retained about the possible existence of elements after a graph transformation.

Notation 7. $\mathcal{AG}$ denotes the universe of abstract graphs.

Definition 16 (Graph Abstraction)**.** Let G be a graph, and A and abstract graph. A abstracts G, written as $G \sqsubseteq A$, if $\{V_A \setminus \hat{V}_A\} \subseteq V_G \subseteq V_A$ and $\{E_A \setminus \hat{E}_A\} \subseteq E_G \subseteq E_A$.

3.3.1.2 ABSTRACT GRAPH TRANSFORMATION RULES

Definition 4 gives the definition of a graph transformation rule. For abstraction we will extend this concept to define abstract graph transformation rules $\hat{r}$.

Definition 17 (Abstract Graph Transformation Rule)**.** An abstract graph transformation rule $\hat{r}$ is a tuple $\langle L, \hat{R}, \hat{p} \rangle$, where L is a graph, $\hat{R}$ is an abstract graph and $\hat{p} : L \to \hat{R}$ is a total morphism from L to $\hat{R}$.

Definition 18 (Graph Transformation Rule Abstraction)**.** Let $r = \langle L, R, p \rangle$ be a graph transformation rule then, $\hat{r} = \langle L, \hat{R}, \hat{p} \rangle$ is the abstraction of r where,

– L is the original LHS graph in r.

– $\hat{p} : L \to \hat{R}$ is an extension of p such that it is a total graph morphism, and each edge $e \in \big(E_L \setminus dom(p_E)\big)$ and node $v \in \big(V_L \setminus dom(p_V)\big)$ is mapped to a fresh edge or node with respect to R for $\hat{p}_E$ and $\hat{p}_V$ respectively.

– $\hat{R}$ is the abstraction of graph R such that,

- $V_{\hat{R}} := \hat{p}_V(V_L)$;
- $\hat{V}_{\hat{R}} := \big(V_R \setminus rg(p_V)\big) \cup \hat{p}_V\big(V_L \setminus dom(p_V)\big)$;
- $E_{\hat{R}} := \hat{p}_E(E_L)$;
- $\hat{E}_{\hat{R}} := \big(E_R \setminus rg(p_E)\big) \cup \hat{p}_E\big(E_L \setminus dom(p_E)\big)$.

It is possible to extend the abstraction of simple graph transformation rules to also include NACs as defined in Definition 6. In order to include NACs we need to define abstract satisfiability of NACs.

Definition 19 (Abstract Satisfaction)**.** For an abstract graph $\hat{G}$ and match $u : L \to \hat{G}$, a NAC n is abstractly satisfied if $\neg\exists m' : im(n) \to \hat{G}$ such that $rg(m'_E) = \{e \mid e \in E_{\hat{G}} \setminus \hat{E}_{\hat{G}}\}$, $im(m'_V) = \{v \mid v \in V_{\hat{G}} \setminus \hat{V}_{\hat{G}}\}$ and $m' \circ n = \mu$. This is written as $\mu \models_{\text{ABS}} n$.

The difference between satisfiability and abstract satisfiability is thus that for abstract satisfiability we loosen the constraint such that any morphism m' should only map elements from N to *real* elements in $\hat{G}$.

Definition 20 (Match on Abstract Graph)**.** Let $\{\hat{r}, NAC\}$ be an abstract graph transformation rule with a set of NACs, $\hat{G}$ be an abstract graph and μ a graph morphism. μ is a match of r to $\hat{G}$ if $\mu : L_{\hat{r}} \to \hat{G}$ is a total graph morphism, and $\mu \models_{\text{ABS}} n$, $\forall n \in NAC$.

3.3.1.3 ABSTRACT GRAPH TRANSFORMATIONS

In Section 2.1 a graph transformation has been defined as the tuple $G \overset{r,\mu}{\Rightarrow} H$ where r is a graph transformation rule and μ is a match from L in r to G. Note that for linearization abstraction we will assume that rules are in the simplest form and thus a match μ is a graph morphism.

$$\begin{array}{ccc} L & \xrightarrow{\quad p' \quad} & \hat{R} \\ \downarrow{\scriptstyle \mu} & & \downarrow{\scriptstyle \mu'} \\ \hat{G} & \xrightarrow{\quad f \quad} & \hat{H} \end{array}$$

Figure 3.3.1: Abstract Graph Transformation

We now define an abstract graph transformation which given an abstract graph G, an abstract rule $\hat{r}$ and a match $\mu : L \to G$ is the tuple $\hat{G} \overset{\hat{r},\mu}{\Rightarrow} \hat{H}$. The relations between graphs, abstract graphs and graph morphisms involved are shown in Figure 3.3.1.

The resulting abstract transformed graph $\hat{H}$ of an abstract graph transformation given $\hat{G}, \hat{r}$ and μ is generated in the same way as H would be generated in a graph transformation with the addition that for $\hat{H}$ the sets $\hat{V}_{\hat{H}}$ and $\hat{E}_{\hat{H}}$ need to be defined. The definition is given by;

$$\hat{V}_{\hat{H}} := f(\hat{V}_G) \cup \mu'(\hat{V}_{\hat{R}}) \tag{3.3.1}$$

$$\hat{E}_{\hat{H}} := f(\hat{E}_G) \cup \mu'(\hat{E}_{\hat{R}}) \tag{3.3.2}$$

These *maybe* element sets in $\hat{H}$ are thus a combination of the image of previous *maybe* element in abstract graph $\hat{G}$ and the image of elements created and deleted by $\hat{r}$. In this way $\hat{H}$ not only stores the information of which elements are newly created and deleted after a graph transformation but also keeps a history of the effects of previous graph transformations.

In terms of the abstraction of the state space S of a production system, we can say of the abstracted state space: if $G \overset{r,\mu}{\Rightarrow} H$ and $G \sqsubseteq \hat{A}$ then $\hat{A} \overset{\hat{r},\mu}{\Rightarrow} \hat{B}$ such that $H \sqsubseteq \hat{B}$. Furthermore, if $G \sqsubseteq \hat{A}$ and $\hat{A} \overset{\hat{r},\mu}{\Rightarrow} \hat{B}$ then $G \sqsubseteq \hat{B}$. Intuitively, what the first statement means is that if there exists a transition from state G to H in the state space then there also exists an abstract transition from an abstract graph $\hat{A}$ of G to an abstract graph $\hat{B}$ of H. The second statement implies that in this abstract state space that graph abstraction is preserved by abstract graph transformations. In other words, if $\hat{A}$ is an abstract graph of G and there exists an abstract transformation from $\hat{A}$ to $\hat{B}$ then also $\hat{B}$ must be an abstraction of G. This is due the fact that graph elements are never deleted by abstract graph transformation rules.

3.3.1.4 TRACKERS & DEPENDENCY + ABSTRACT GRAPH TRANSFORMATIONS

We now introduce the concept of a tracker which serves the purpose of bookkeeping. This bookkeeping functionality will allow us to set up different metrics to determine the "distance" for the heuristic H_{LA}.

Definition 21 (Tracker)**.** A tracker t is a triple $\langle \hat{r}, \mu, iteration\# \rangle$ such that,

– $\hat{r}$ an abstract rule;

– μ is a match;

– $iteration\# \in \mathbb{N}$ is an abstract graph transformation iteration.

We use $\mathcal{T}$ to indicate a set of trackers.

These tracker triples t indicate the dependency of a node or edge with respect to which abstract rules $\hat{r}$ are required in order to create or delete the element. Each node v and edge e of an abstract graph derived through a sequence of rule applications has a set of trackers ($\mathcal{T}_e$ and $\mathcal{T}_v$).

An element has a set of trackers because its creation may be dependent on a series of rule applications (i.e. the application of a rule enables the rule which creates the element). The tracker set $\mathcal{T}$ of an element thus captures a history of all rules required to create it and is transitively closed.

Next we define the dependency relation. Intuitively, a dependency maps graph elements to sets of trackers.

Definition 22 (Dependency)**.** A dependency $d : \hat{G} \to 2^{\mathcal{T}}$ is a mapping of abstract graph elements $(e \in \hat{E}_{\hat{G}}, v \in \hat{V}_{\hat{G}})$ to sets of trackers $\mathcal{T}$.

Initially the dependency relation for an initial abstract graph (before any abstract graph transformations) is a mapping from all graph elements to the empty set. However, every graph transformation will create tracker triples, and newly created elements in $\hat{V}$ and $\hat{E}$ will be dependent on the $\hat{r}$ and μ in the abstract graph transformation and possible previous abstract graph transformations. Therefore, in an abstract graph transformation we need to also maintain this dependency relation.

To achieve this we make sure that an element, along with obtaining the tracker corresponding to the graph transformation $(\hat{r}, \mu)$ and iteration which is responsible for creating it, also inherits the trackers of all elements which are present in the match μ in the graph transformation.

We define $(\hat{G}, d) \overset{\hat{r},\mu,i}{\Longrightarrow} (\hat{H}, d')$ as the dependency plus abstract graph transformation (or DAG transformations) which is an extension of an abstract graph transformation. The input for such a transformation is an abstract graph $\hat{G}$, dependency relation d, abstract rule and corresponding match $\hat{r}, \mu$ and an iteration counter i. The result of a DAG transformation is the abstract graph transformation as defined in Section 3.3.1.3 as well as an update of the dependency d' with respect to the abstract rule application $\hat{r}$ given μ.

This update to d', given an abstract graph transformation and d, consists of two parts. The first is defining a new tracker which corresponds to the abstact graph transformation. This is given as:

$$t_{new} =< \hat{r}, \mu, i >$$

Next we need to update d to d' such that there is a mapping from each new element to its tracker set and for each existing element we adjust for element remapping in f. We thus define d'_E and d'_V as follows:

$$\begin{aligned}
d'_V(v) &:= d_V(f^{-1}(v)) && \text{for } \forall v \in \{V_{\hat{H}} \setminus \mu'(\hat{V}_{\hat{R}})\} \\
d'_V(v) &:= \{t_{new}\} \cup \left(\bigcup d_v(L)\right) && \text{for } \forall v \in \mu'(\hat{V}_{\hat{R}}) \\
d'_E(e) &:= d_E(f^{-1}(e)) && \text{for } \forall e \in \{E_{\hat{H}} \setminus \mu'(\hat{E}_{\hat{R}})\} \\
d'_E(e) &:= \{t_{new}\} \cup \left(\bigcup d_e(L)\right) && \text{for } \forall e \in \mu'(\hat{E}_{\hat{R}})
\end{aligned}$$

3.3.1.5 Linearization of Abstract Graph Transformations

Given the definitions of an abstract graph, abstract graph transformation rules and abstract graph transformations we can now combine these to define parallel abstract graph transformations. We call this a linearization because this approach creates a linear state

space of abstract graphs instead of a branching state space. In addition to abstract graph transformations we can also include transformation book keeping in the form of dependencies. We thus create parallel DAG transformations.

Note that since DAG transformations are an extension of abstract graph transformations we can simply ignore (set to `null`) all dependency related variables to achieve parallel abstract graph transformations from parallel DAG transformations.

The application of DAG transformations is shown in Algorithm 2. It takes as input an abstract graph $\hat{G}$ and the set M of all abstract rules $\hat{r}$ and corresponding valid matches μ as well as the dependency d and an external iteration counter i. The algorithm returns the successor abstract graph $\hat{G}'$ which is the abstract graph to which abstract graph transitions are applied in parallel and its respective dependency relation.

Algorithm 2: Parallel DAG Transformations

input : $\hat{G}$ – initial abstract graph;
M – set of abstract rule and with match $(\hat{r},\mu)$ applicable to $\hat{G}$;
d – dependency relation for $\hat{G}$;
i – iteration of parallel abstraction graph transformation.

output : $\hat{G}'$ – abstract graph after linearization of graph transformations;
d' – dependency relation for $\hat{G}'$.

1 **for** $(\hat{r}_j,\mu_j) : M$ **do**
2 Let $(\hat{G}'_j,d'_j)$ be the abstract graph such that $(\hat{G},d) \xRightarrow{(\hat{r}_j,\mu_j,i)} (\hat{G}'_j,d'_j)$;
3 **end**
4 **return** $\left(\bigcup_j \hat{G}'_j, \bigcup_j d'_j\right)$;

In Algorithm 2 we see the successor abstract graph $\hat{G}'$ of $\hat{G}$ for M is given by calculating the successor $\hat{G}'_j$ for every $j \in M$ and then taking the union of all these successors. By construction there can never be a conflict between abstract graph transformations. A conflict is caused by the fact that one transformation has a requirement which would be unsatisfied if another transformation is performed. An example would be if a transformation r_1 has the condition which adds an edge to a certain node n and another transformation r_2 deletes that specific node n. Using standard transformations these two rules would not be able to be applied in parallel since there is a conflict of actions on that certain node n. However, in abstract graph transformations these transformations can be applied in parallel because n is not actually deleted but only placed in the maybe node set. The argument is similarly true for d'.

3.3.1.6 WITNESSES AND GOAL MATCHES

In this section we define the concept of *witnesses*. A *witness* is a set of elements, (i.e. subgraph) in an abstract graph which satisfies a goal. Intuitively, for an abstract graph $\hat{G}$ and goal ϕ, we say $\hat{G}$ is a goal graph if we can find a witness. The WITNESS function calculates all witnesses in $\hat{G}$. Determining if a witness satisfies a goal is goal type dependent and is done by the MATCH function, which is a helper function within WITNESS.

We use an abstract variable x which serves as goal type parameter. This variable can be concretely instantiated to a goal type which are defined in Section 2.3.3.

Equation (3.3.3) gives the generic witness function which has as parameters x, representing a goal type, and ϕ, representing a goal of type x. WITNESS takes as input an abstract graph $\hat{G}$ and returns a set of graph element sets or witnesses. These witnesses represent the subgraphs in $\hat{G}$ which match the goal ϕ. These matches are graph morphisms μ which are captured by the MATCH function.

$$\text{WITNESS}^x_\phi : \hat{G} \mapsto \{im(\mu) \mid \mu \subseteq \text{MATCH}^x_\phi(\hat{G})\} \tag{3.3.3}$$

The condition for determining which subgraphs match a goal is goal type dependent. Thus, for each goal type we can define the conditions for which a subgraph matches a goal. In the following definitions we define the MATCH function for the goal types: **a)** partial negation graph ($x := n$), **b)** partial graph ($x := p$) and **c)** complete graphs ($x := c$).

Definition 23 (Partial Negation Goal Match)**.** Let $\phi = (N, \{NAC\})$ be a negation partial goal (where N is a graph and $\{NAC\}$ a set of NACs), and μ a graph morphism. $\mu \in \text{MATCH}^n_\phi(\hat{G})$ if $\mu : N \to \hat{G}$ is a total graph morphism , and $u \models_{\text{ABS}} nac, \forall nac \in \{NAC\}$.

Definition 24 (Partial Goal Match)**.** Let $\phi = P$ be a partial goal (where P is a graph), and μ a graph morphism. $\mu \in \text{MATCH}^p_\phi(\hat{G})$ if $\mu : P \to \hat{G}$ is a total graph morphism.

Definition 25 (Complete Goal Match)**.** Let $\phi = C$ be a complete goal (where C is a graph) and μ a graph morphism. $\mu \in \text{MATCH}^c_\phi(\hat{G})$ if $\mu : C \to \hat{G}$ is a total graph morphism such that $\{E_{\hat{G}} \setminus \hat{E}_{\hat{G}}\} \subseteq rg(\mu_E)$ and $\{V_{\hat{G}} \setminus \hat{V}_{\hat{G}}\} \subseteq rg(\mu_V)$.

If in the actual problem a graph G satisfies the goal ϕ then for all abstract graphs $G \sqsubseteq \hat{G}$ in the abstraction of the problem to following should hold: $\text{MATCH}^x_\phi(\hat{G}) \neq \emptyset$. Intuitively, this means that all actual goal states are also goal states in problem abstraction. Thus, in our over approximation we ensure we completeness in terms of goal states.

We hypothesize that it is also possible to define a concrete MATCH function for goals of the type predicate of graphs. However, this is not trivial and is beyond the scope of this work.

3.3.1.7 LINEARIZATION ABSTRACTION

We can now tie all previously defined concepts of abstraction together to perform the linearization abstraction of a graph G and witness function. For linearization abstraction we assume the known parameters: the rule set $\mathcal{R}$ and a loop threshold THRESHOLD. The result of linearization abstraction is a dependency relation d, which is a mapping from $\hat{G}$ to 2^T, a set of witnesses W (which is the first non-empty set result of WITNESS on input $\hat{G}$), and the iteration count *iter* which is the number of parallel abstraction graph transformations required to find a witness. Note that $\hat{G}$, which an abstract graph of G on which all abstract transformations are performed, is not part of the result. The reason for this is that all necessary information to determine a heuristic value based on the linearization abstraction approach is already contained in the dependency relation, witness set and iteration count.

Pseudo code of the linearization abstraction algorithm is given in Algorithm 3. The function `ParallelTransformation` on Line 13 is a call to Algorithm 2. Furthermore, the WITNESS function on Line 7 depends on the goal type x of ϕ. In the concrete instantiation of the algorithm we require x to be set to a concrete goal type. The goal and goal type used in the WITNESS function of linearization abstraction is set by its respective heuristic scheme.

The algorithm itself consists of two parts. The first part is the initialization of variables. This consists mostly of the abstraction of the input and parameters and is done as defined in the previous sections. The second part is the `While`-loop from Line 6 to Line 14. In this loop we repeatedly do the following:

1. check if the current abstract graph $\hat{G}$ is a goal graph (Line 8)
2. find all abstract rules and matches M valid on $\hat{G}$ (Line 12)
3. apply the parallel DAG transformation for all abstract rules and matches in M (Line 13)

Algorithm 3: Linearization Abstraction

input : G – initial graph;
WITNESS_ϕ^x – Witness function for goal ϕ where goal type is x;
$\mathcal{R}$ – rule set; ; `// Global variable`
1 THRESHOLD– loop iteration threshold. ; `// Global variable`
output : d – dependency relation;
W – Set of *witnesses*.

2 $\hat{G} \leftarrow G$ extended with $\hat{V}, \hat{E} := \emptyset$;
3 $\hat{\mathcal{R}} \leftarrow$ abstraction of $\mathcal{R}$;
4 $d \leftarrow \{(v, \emptyset) \mid v \in V_{\hat{G}}\} \cup \{(e, \emptyset) \mid e \in E_{\hat{G}}\}$;
5 iter $\leftarrow 0$;
6 **while** iter $<$ THRESHOLD **do**
7 $\quad W \leftarrow \text{WITNESS}_\phi^x(\hat{G})$;
8 $\quad$ **if** $W \neq \emptyset$ **then**
9 $\quad\quad$ **return** (iter, d, W);
10 $\quad$ **end**
11 $\quad$ iter $\leftarrow$ iter $+ 1$;
12 $\quad M \leftarrow \{(\hat{r}, \mu) \mid \hat{r} \in \hat{\mathcal{R}}, \mu : L_{\hat{r}} \rightarrow \hat{G}\}$;
13 $\quad (\hat{G}, d) \leftarrow$ `ParallelTransformation`$(\hat{G}', M, \text{iter}, d)$;
14 **end**
15 **return** (iter, d, W);

The loop only exits in two cases, the first is when the loop threshold THRESHOLD is met (line 15) and the second is if $\hat{G}$ is a goal graph (line 9). We include a loop threshold since it may be possible that there does not exist a path from input G to a

goal graph G_{goal} given $\mathcal{R}$. In this case it may also be that no such path exists given $\hat{\mathcal{R}}$ and abstract graph transformations. So we require a threshold to avoid an infinite loop.

As mentioned, in the second case the loop is exited when $\hat{G}$ is a goal graph. In this case we have achieved the objective of the algorithm and we return the result.

In the next section we will show how the result of the linearizion abstraction algorithm may be used a to estimate the distance from a graph G to a goal ϕ. These distance estimates serve as basis for graph heuristics based on linear abstraction.

3.3.1.8 DISTANCE METRICS

We define three distance metrics based on linearization abstraction. The first is a naive approach which is the number of parallel abstract graph transformation iterations needed to reach an abstract graph which is a goal graph, given ϕ, from G. We call this metric “iteration count" (or IC for short). This is a naive approach since parallel abstract graph transformations allow multiple abstract graph transformations in a single iteration, so we may greatly underestimate the actual distance if there are many independent graph transformations required.

Therefore, we introduce a second distance metric which aims to refine IC. To more accurately estimate the distance between graph G and a goal we count all relevant abstract rules needed such that $\hat{G}$ is a goal graph. In this way the distance metric becomes the exact number of abstract rules needed to reach a goal in an abstract graph. We call this metric “dependency count" (or DC for short).

Finally, we introduce a third distance metric which is a midway between the IC and DC metrics. This third metric counts all rule applications applied during linearization abstraction. We call this metric “match count" (or MC for short). The two purposes of this metric is to provide a metric more accurate than iteration count and less computation heavy than dependency count. The first purpose is easily achieved as the number of iteration generally has a correlation to number of matches. The second approach is also achieved since this metric does not rely on witnesses, thus in the implementation we can reduce the number of calculations needed to calculate the match count value of a graph.

We can calculate these distance metrics for G using the trackers and witnesses created during the linearization abstraction process. From the result (d, W) of the linearization abstraction we can obtain all relevant trackers and from these calculate the corresponding heuristic value. For all metrics we give pseudo-code algorithms which, given a dependency relation and set of witnesses for an abstract graph calculates the heuristic value. These algorithms are Algorithm 4, Algorithm 5 and Algorithm 6 for iteration count, match count and dependency count respectively.

As one can see in Algorithm 4 the iteration count algorithm simply returns the iteration count i from the input.

In Algorithm 5 the result is the number of trackers in the image of the dependency relation. For each rule application a unique tracker is created, and the dependency relation tracks which elements have which trackers. In this case we want to know all rule applications thus we want all trackers, this is achieved by taking the image of d.

Algorithm 6 is slightly more complex. Firstly we are only concerned with the trackers corresponding to elements which are also part of the goal graph. These elements

Algorithm 4: Heuristic: Iteration Count (IC)

input : i – iteration count of Linearization Abstraction algorithm;
d – dependency relation for $\hat{G} \to 2^{\mathcal{T}}$;
W – set of *witnesses* for $\hat{G}$ given ϕ.
output : $\mathbb{R}_0^+$ – heuristic value.

1 **return** i;

Algorithm 5: Heuristic: Match Count (MC)

input : i – iteration count of Linearization Abstraction algorithm;
d – dependency relation for $\hat{G} \to 2^{\mathcal{T}}$;
W – set of *witnesses* for $\hat{G}$ given ϕ.
output : $\mathbb{R}_0^+$ – heuristic value.

1 **return** $|\bigcup\{d(x) \mid x \in \hat{G}\}|$;

Algorithm 6: Heuristic: Dependency Count (DC)

input : i – iteration count of Linearization Abstraction algorithm;
d – dependency relation for $\hat{G} \to 2^{\mathcal{T}}$;
W – set of *witnesses* for $\hat{G}$ given ϕ.
output : $\mathbb{R}_0^+$ – heuristic value.

1 **return** $\texttt{Min}\left\{\bigcup|d(w)| \mid w \in W\right\}$;

correspond to a witness $w \in W$. We use the dependency relation to determine all trackers corresponding to the elements in a witness. From this set we take the cardinality as the heuristic value. It may be possible there are multiple witnesses for a goal graph. The heuristic value should correspond to the witness which has the lowest number of trackers. The function `Min` stands for the minimum operation which takes the smallest value from a set.

3.3.1.9 LINEARIZATION ABSTRACTION HEURISTIC SCHEMES

We use the three linearization abstraction distance metrics defined above to define three goal type generic heuristic schemes. We define the heuristic scheme $HS^x_{LA-IC}(\phi)$ based of the naive iteration count metric in Equation (3.3.4), $HS^x_{LA-MC}(\phi)$ based on the efficient match count metric in Equation (3.3.5) and $HS^x_{LA-DC}(\phi)$ based on the more sophisticated dependency count metric in Equation (3.3.6).

$$HS^x_{LA-IC}(\phi) : G \mapsto \textsc{Ic}\big(\textsc{La}(G, \textsc{Witness}^x_\phi)\big) \tag{3.3.4}$$

$$HS^x_{LA-MC}(\phi) : G \mapsto \textsc{Mc}\big(\textsc{La}(G, \textsc{Witness}^x_\phi)\big) \tag{3.3.5}$$

$$HS^x_{LA-DC}(\phi) : G \mapsto \textsc{Dc}\big(\textsc{La}(G, \textsc{Witness}^x_\phi)\big) \tag{3.3.6}$$

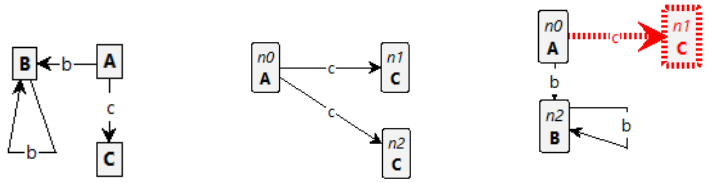
The heuristic schemes as given in Equation (3.3.4), (3.3.5) and (3.3.6) are generic since they are require a concrete goal type. The goal type for these schemes is defined by setting the parameter x to a concrete goal type, this also instantiates the WITNESS function and, also the MATCH helper function, with a concrete goal type. In Section 3.3.1.6 we defined concrete instances of MATCH, and thus WITNESS and HS_{LA}, for the goal types: **a)** partial negation graph ($x := n$), **b)** partial graph ($x := p$) and **c)** complete graphs ($x := c$). Therefore, Equation (3.3.4), Equation (3.3.5) and (3.3.6) together actually represent nine concrete heuristic schemes. We can use such a concrete heuristic scheme in combination with a goal ϕ to create heuristic functions based on linearization abstraction.

3.3.2 LINEARIZATION ABSTRACTION EXAMPLE

In this section we give an example of the linearization abstraction algorithm and how algorithm output is used to calculate heuristic values for each linearization abstraction heuristic scheme HS^x_{LA}. In order to do this we introduce a new trivial problem which is given as a production system and a partial graph goal. The reason a trivial problem is used as example is that due to the complexity and many steps in the process of linearization abstraction a larger problem quickly becomes convoluted.

We present the problem used in this example in the form of graphs and graph transformations in GROOVE. The type graph and start graph of the problem are given in Figure 3.3.2a and Figure 3.3.2b respectively. This example problem is thus a world of A, B and C nodes which are connected by b and c edges. The start graph consists of a

single A node which is connected to two C nodes by a c edge. In the start graph we also show the node numbering, this is done so we can visually track nodes during the iterations of linearization abstraction later on. Figure 3.3.2c shows the partial graph goal for the example problem. The goal represents a state in which the A node connected to a B node by a b edge and the B node has a self b edge, furthermore, the A node the goal requires the A edge is not connected to any C nodes by a c edge.



(a) Type Graph (b) Start Graph G_0 (c) Partial Graph Goal ϕ_P

Figure 3.3.2: Type, Start and Goal graph of sample linearization abstraction example

Figure 3.3.3 show the transformation rules $\mathcal{R}_{ex}$ of the example problem. These are labeled r_1, r_2 and r_3. They are relatively trivial rules which simply create or delete some elements from the host graph.

(a) r_1: new **B** node with **b** edge from **A** node (b) r_2: new **b** edge from **B** node (c) r_3: remove **C** node and **c** edge from **A** node

Figure 3.3.3: Transformation rules for small linearization abstraction example

We will use linearization abstraction to this example problem and using heuristic schemes HS^p_{LA} to determine the heuristic values from the start state corresponding to G_0 to a goal state for ϕ_P. We thus want to calculate three heuristic values. To do this we first need to define the witness function. This is given as $\text{WITNESS}^p_{\phi_P}(\hat{G})$, where n refers to the goal type and phi_P is the goal we wish to find witnesses for in $\hat{G}$. Next we can define the heuristic functions we will use to calculate heuristic values. These are:

1. $HS^p_{LA-IC}(\phi_P)(G, \text{WITNESS}^p_{\phi_P}(\hat{G}))$,
2. $HS^p_{LA-MC}(\phi_P)(G, \text{WITNESS}^p_{\phi_P}(\hat{G}))$, and
3. $HS^p_{LA-DC}(\phi_P)(G, \text{WITNESS}^p_{\phi_P}(\hat{G}))$.

which calculate results based on the distance metrics of iteration count, match count and dependency count of linearization abstraction respectively. Each of these functions requires as input the result of linearization abstraction. Therefore, in this example we next show this process, as described in Algorithm 3, is applied to our example problem.

To do this we consider the input as defined above. In this example the THRESHOLD input parameter is not relavent and thus we ignore it.

The first step is to create the abstract graph $\hat{G}_0$ which is an initial abstraction of G_0 (line 2 of Algorithm 3). Secondly, we create the abstract rules of the problem by applying the abstraction of transformation rules as defined in Definition 18 (line 3 Algorithm 3).

Once we have abstracted the input graph and created the abstraction on the transformation rules we create an new dependency set d which initially maps all elements in the abstract graph to an empty set of trackers. Furthermore, we initiate the ITER to zero.

The initial abstraction of G_0 to $\hat{G}_0$ simply states that G_0 is extended which two maybe sets for nodes and edges. The result is practically the same graph shown in Figure 3.3.2b. The abstraction of the transformation rules $\mathcal{R}_{ex}$ can be as considered as that all creator elements of the rules in Figure 3.3.3 represent the mapping to a maybe element in the abstract graph and that all eraser elements are matched on real elements and mapped to maybe element in the abstract graph.

At this stage we have initialed all necessary variables for linearization abstraction and can apply iterations of parallel DAG transformation until the witness set W is non empty (lines 6-14 of Algorithm 3).

Figure 3.3.4 give a visual representation of the abstract graph $\hat{G}_0$ for each iteration in the abstract graph transformation. Where gold colored graph elements represent maybe elements.

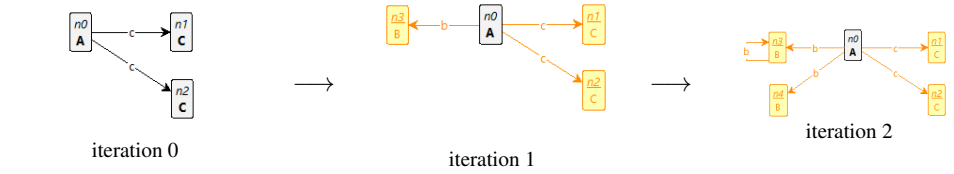


Figure 3.3.4: Linearization Abstraction of small example

In the first iteration of linearization abstraction loop three abstract graph transformations are applied to $\hat{G}_0$. These are a single application of $\hat{r_1}$ and two applications of $\hat{r_3}$, the first matched on node $n1$ and the second on $n2$. This whole step is captured in a single parallel DAG transformation, which in addition to the transformation also creates trackers and updates the dependency set. In the first iteration, 3 trackers are created, one for each abstract rule application.

$$\langle \hat{r_1}, n0, 1\rangle, \langle \hat{r_3}, n1, 1\rangle, \langle \hat{r_3}, n2, 1\rangle \tag{3.3.7}$$

Each maybe element in $\hat{G}_0$ of iteration 1 is mapped to the tracker responsible for

moving it to the the maybe set. This is done in the dependency relation d given below.

$$\begin{aligned}\bigcup d(\hat{G_0}) = \{&n1 \to \{\langle \hat{r_3}, n1, 1\rangle\},\\ &n2 \to \{\langle \hat{r_3}, n2, 1\rangle\},\\ &n3 \to \{\langle \hat{r_1}, n0, 1\rangle\},\\ &(n0 \xrightarrow{c} n1) \to \{\langle \hat{r_3}, n1, 1\rangle\},\\ &(n0 \xrightarrow{c} n2) \to \{\langle \hat{r_3}, n2, 1\rangle\},\\ &(n0 \xrightarrow{b} n3) \to \{\langle \hat{r_1}, n0, 1\rangle\}\}\end{aligned} \tag{3.3.8}$$

In the second iteration two abstract graph transformations are applied to $\hat{G_0}$. These are $\hat{r_2}$ matched on node $n3$ and $\hat{r_1}$ matched on node $n0$. For both these transformations a new tracker is created.

$$< \hat{r_1}, n0, 2 >, < \hat{r_2}, n3, 2 > \tag{3.3.9}$$

Furthermore, the dependency set is updated as shown below.

$$\begin{aligned}\bigcup d(\hat{G_0}) = \{&n1 \to \{\langle \hat{r_3}, n1, 1\rangle\},\\ &n2 \to \{\langle \hat{r_3}, n2, 1\rangle\},\\ &n3 \to \{\langle \hat{r_1}, n0, 1\rangle\},\\ &n4 \to \{\langle \hat{r_1}, n0, 2\rangle\},\\ &(n0 \xrightarrow{c} n1) \to \{\langle \hat{r_3}, n1, 1\rangle\},\\ &(n0 \xrightarrow{c} n2) \to \{\langle \hat{r_3}, n2, 1\rangle\},\\ &(n0 \xrightarrow{c} n3) \to \{\langle \hat{r_1}, n0, 1\rangle\},\\ &(n0 \xrightarrow{b} n4) \to \{\langle \hat{r_1}, n0, 2\rangle\},\\ &(n3 \xrightarrow{b} n3) \to \{\langle \hat{r_1}, n0, 1\rangle, \langle \hat{r_2}, n3, 2\rangle\}\}\end{aligned} \tag{3.3.10}$$

In the third iteration a witness can be found which satisfies ϕ_P (as defined in Definition 24). The set of witnesses W is given as follows.

$$W = \{\{n0, n3, (n0 \xrightarrow{c} n3), (n3 \xrightarrow{b} n3)\}\} \tag{3.3.11}$$

Thus, at this point the linearization abstraction algorithm terminates. The result is a tuple of the iteration count, the dependency relation and witness set. given as $\langle iter, d, W\rangle$. We can now use this result as input for the heuristic functions defined earlier in this example.

In Equation (3.3.12), (3.3.13) and (3.3.14) we calculate the heuristic value which estimates the distance from the initial state corresponding to G_0 to a goal state which satisfies ϕ_P. Each function uses a different metric to estimate the difference.

$$HS^p_{LA-IC}(\phi_P)(G_0, \textsc{Witness}^p_{\phi_P}(\hat{G_0})) = 2 \tag{3.3.12}$$

$$HS^p_{LA-MC}(\phi_P)(G_0, \textsc{Witness}^p_{\phi_P}(\hat{G_0})) = 5 \tag{3.3.13}$$

$$HS^p_{LA-DC}(\phi_P)(G_0, \textsc{Witness}^p_{\phi_P}(\hat{G_0})) = 2 \tag{3.3.14}$$

Given these heuristic values it is important to note that while there is a correspondence between values they are not directly comparable. That is to say, for example, while

both the IC and DC metric heuristic functions give the same result, this does not mean they are both equally well suited as heuristic for this problem. The heuristic value should be put in context of what the distance metric is measuring.

With this in mind, we see that both IC and DC heuristic function have a value of 2 and MC heuristic function has a value of 5. What this means is that in the linearization abstraction of the problem only two iterations are needed to reach a goal state from the start state. By refining the metric to rule applications we see that 5 abstract rule applications are required to reach a goal state. Finally, in dependency count we see that in fact only two abstract rule applications are needed to create an abstract graph which corresponds a goal state.

In this example one can clearly see the levels of refinement in the distance metrics. Concretely, considering the example we can easily see that two rule applications are needed to satisfy the goal. In the abstraction this is most accurately shown by the dependency count heuristic. The match count heuristic is an all encompassing over approximation. The iteration count on the other hand is simplification of the whole abstraction which corresponds to the number of dependent rule applications.

3.3.3 DISCUSSION

The aim of this section is to discuss the advantages and disadvantages of the linearization abstraction heuristic approach as a basis for graph heuristics. Furthermore, we discuss some current limitations to the approach as presented in Section 3.3.1 and give an option of possible extension of the work.

3.3.3.1 ADVANTAGES

The linearization abstraction provides an advantage as heuristic approach by attempting to reduce the explored state space of a planning problem, which is achieved by two components. The first is the problem abstraction. By creating an abstraction of graphs which store information about which graph elements may be present in a current graph we can actually represent a set of graphs (and thus states) by a single abstract graph. This form of abstraction, which is supported by abstract graph transformation rules allows us to create an abstracted form of the problem state space in which we keep track of which abstract transformations are responsible for creating certain graph elements. The second component is linearization of abstract rule applications. By allowing parallel application of independent abstract transformation rules we further reduce the abstracted state space into a linear entity. We can permit this abstraction because abstract graphs record which elements are effected in transformations using maybe sets. Furthermore, using the dependency relation we keep track of exactly which elements are effected by which abstract transformation rule.

A further advantage to the approach is that with this form of abstraction we are able to keep some graph context, this is different to NENTUPLES. Furthermore, linearization abstraction generates an abstract graph from a given start graph to an abstract graph which matches a goal. In terms of heuristics we use the results from this algorithm to the distance between a graph and a goal using distance metrics. However, the results

from linearization abstraction, include the abstract graph, could also be used to calculate a path of abstract rule application from a start graph to a goal. Such information can be used to create a plan for a planning strategy.

A final advantage of the linearization abstraction approach is that from its result it is possible to define several different distance metrics which can be used to calculate heuristic values. These metrics may be combined and allow for extensibility of the approach.

3.3.3.2 DISADVANTAGES

The major disadvantage of the linearization abstraction approach is the computation complexity. Each linearization abstraction is actually a simplified generation of the problem state space. Thus, while abstraction significantly reduces the size of the problem, when a linearization abstraction has to be applied to the graph of every expanded state during an exploration a great deal of computations are still required.

Furthermore, as the size of abstract graph grows, finding all applications of abstract rules becomes more time consuming. This is partially due to the fact that the abstraction and linearization shifts part of the state space explosion commonly found in graph transition system from the state space to abstract graphs. Intuitively, this comes down to finding abstract rule matches in a single abstract graph instead of expanding several states within a state space.

The linearization abstraction approach allows for the application of all independent abstract graph transformation using parallel abstract graph transformations. This provides an advantage in terms of space reduction for problems with many independent actions. However, if the problem requires mostly dependent actions to reach a goal then the whole linearization does not provide any advantage.

In terms of distance metrics based of the results of linearization abstraction there is a trade-off between the accuracy of distance and computation complexity. Especially between match count and dependency count there is an additional level of complexity in the current implementation. This is caused by the implementation of the WITNESS function. The issue is caused by the fact that iteration count and match count only require the witness set to be non-empty while dependency count actually uses the witness set for its calculations. Thus, in the implementation these first two metrics only perform a simple check while the dependency count metric calculates the whole witness set. This, is an expensive operation in proportion to the number of witnesses.

The conclusion from this is as follows. While dependency count provides the most accurate distance estimation it is also the most computation intense to calculate. This is a trade-off a user must seriously consider when deciding which heuristic function to use for a certain problem. For smaller problems or problems with a extremely partial goal graph it may be more efficient to expand some possibly less rewarding states using linearization abstraction heuristic functions based of match count metrics instead of dependency count metrics.

3.3.3.3 LIMITATIONS AND EXTENSION

The main current theoretical limitation of linearization abstraction is that we have not defined the condition in which an abstract graph satisfies a predicate over graphs. In terms of implementation, a major limitation of the linearization abstraction heuristic approach is that as the size of the graph increases, finding all matches for abstract graph transformations becomes more time consuming (this is due to there being more possible matches). This means that the higher the number of iterations in linearization abstraction needed, the longer the whole algorithm takes. At some point it is no longer feasible to calculate the linearization abstraction of a graph in a planning problem. Therefore, we can assume, in the current implementation there is some upper limit with respect to problem size at which point linearization abstraction heuristics no longer provide any performance enhancement. Theoretically, the worst case scenario would be a problem which requires many dependent actions (requiring many iterations) while also including possible independent actions that are not relavent in the solution (cause an increase in total matches found in each iteration).

Abstraction is a trade-off between problem size and problem context. In this approach we have chosen to reduce the problem size by not distinguishing between newly created or deleted maybe elements in an abstract graph. A possible extension of the linearization abstraction approach is to reuse the concepts of parallel DAG transformations and distance metrics with another graph and graph transformation abstraction. By refining the granularity of the abstraction (such as the history of elements) it may be possible to define more accurate heuristic schemes for planning problems.

Apart from the use of linearization abstraction for defining heuristic schemes, it is also a possibility to use this approach as a method for aiding a planning strategy on a global level instead of state level. This can be done by considering a path that solves the linearization abstraction and using this to guide the actual exploration. Thus, in effect solving a simplified version of the problem to better understand the problem. In planning this is also known as creating a plan [4].

Finally, a direct and relevant extension to linearization abstraction could be the addition of sophisticated distance metrics. It could be possible to define an advanced metric which takes into account if actual transformation rules are conflicting, and if their abstracted counterparts are for example part of the trackers in the dependency count, could add a distance penalty to such a transformation. This should be possible, since the dependency relation and trackers record which rule and match were applied to create or delete an element for every iteration of linearization abstraction.

CHAPTER 4

PLANNING IN GROOVE

GROOVE currently supports graph transformation planning problem solving in the form of state space exploration. The basic functionality to solve planning problems thus already exists in GROOVE. however, currently GROOVE is limited to uninformed exploration strategies. This of course presents a major drawback to solving planning problems. Since the state space of a planning problem may be quite large, an uninformed exploration strategy is generally very inefficient.

In this chapter we will discuss what components makes up a graph transformation planning problem, which of these components are already present in GROOVE and how we can incorporate more advanced exploration strategies to improve solving planning problems. The result will be a planning framework in GROOVE which is describes **a)** how planning problems can be formulated in GROOVE, **b)** what capabilities GROOVE offers to solve planning problems and **c)** how solutions are presented.

In Section 4.1 we describe the required components needed to support solving graph transformation planning problems. Next, in Section 4.2 we discuss how GROOVE should be extended in order to more fully support planning problem solving. Finally, we review some practical implementation aspects of linearization abstraction in Section 4.3.

4.1 PLANNING COMPONENTS

In Figure 4.1.1 we see a feature diagram which specifies the three components which make up framework of graph transformation planning problems. These three components are **a)** the planning problem, **b)** the solution strategy and **c)** the solution. These concepts were previously introduced and described in Section 2.3. In this section we will discuss what each component entails with respect to implementation in GROOVE and describe the relation between the components. As a whole these represent the pillars of the planning framework.

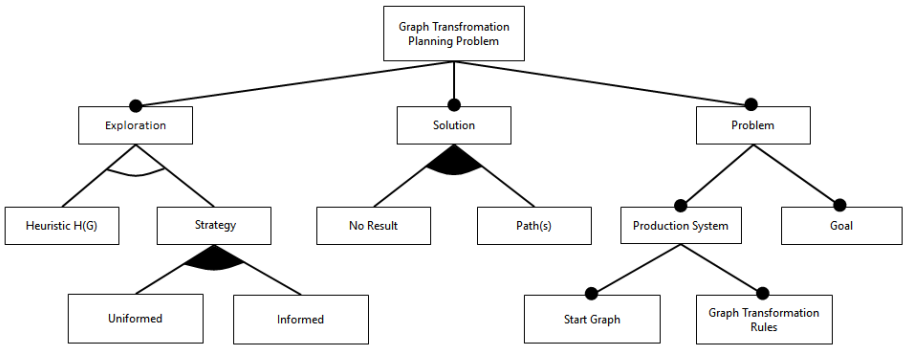


Figure 4.1.1: Graph transformation planning problem feature diagram

4.1.1 PLANNING PROBLEM

The right branch of Figure 4.1.1 specifies the planning problem. This component consists of a production system (see Definition 8) and a goal (see Definition 9).

In terms of implementation, the production system concept exists within GROOVE and is implicitly contained within the `Grammar` object. From the Grammar object we can obtain the set of rules for the problem, as well as the start graph of a problem and a corresponding type graph. It is possible to model a planning problem using the GUI to define a grammar in GROOVE. In Section 2.3.3 we have defined a spectrum of goal types. For the implementation in GROOVE we limit ourselves to goal types in the form of a graph. We implement such goals using the already defined `Rule` objects in GROOVE. A goal can thus be modeled as a graph transformation rule without the deletion or creation of elements, thus given a rule $r = (L, R, p)$ that the left-hand side L is isomorphic to R. Intuitively, this means that we in fact have a rule where nothing changes. For partial negation graph goals, we also include NACs in the rule.

The goal becomes input for a possible heuristic scheme used during exploration. Furthermore, the goal becomes the acceptor (see Section 2.2) for the exploration used to solve the problem. The `Acceptor` class exists in GROOVE, for which subclasses define a range of possible acceptor types and specifies their fulfillment conditions. For each goal type we should create an acceptor with the following fulfillment conditions for each type respectively;

- partial negation graph – Given goal $\phi_{goal} = (N, \{NAC\})$ and state q, the acceptor is fulfilled if there exists a total graph morphism μ such that, $u : N \to G_q$ and $\mu \models n, foralln \in NAC$.
- partial graph – Given goal $\phi_{goal} = P$ and state q, the acceptor is fulfilled if there exists a total graph morphism μ such that, $\mu : P \to G_q$.
- complete graph – Given goal $\phi_{goal} = C$ and state q, the acceptor is fulfilled if C is isomorphic to G_q.

In the case of the partial negation graph and partial graph goal types, this can be

implemented in GROOVE by finding a graph transformation (see Definition 5) for the graph representation a state G_q and the goal encapsulated in a graph transformation rule. In the case of the complete graph goal type an actual isomorphism check between the graph corresponding to a state G_q and the goal C should be performed. In this work we only implement acceptors of the partial negation graph and partial graph goal types. Implementation of the third acceptor is possible by using the `areIsomorphic()` function in the `IsoChecker` class in GROOVE to check isomorphism.

In terms of input for this component of graph transformation problems, the system requires a production system and an additional rule representing a goal in the form of a grammar. In order to parse the goal (modeled as a rule) the system furthermore requires the name of the rule in the grammar and the goal type.

4.1.2 SOLUTION

The center branch of Figure 4.1.1 specifies the solution component of graph transformation problems. A solution of a planning problem is conceptually considered in Section 2.3.4. There are two possibilities, either an acceptor is fulfilled in which case the result should be a path(s) from the start state to the a goal state (see Definition 10), or the acceptor is never fulfilled in which there is no result.

In terms of implementation, the results of an exploration are contained in the `ExploreResult` object, which is a variable of the Acceptor object. In every step of the exploration the Acceptor is checked if its condition is satisfied; if so, then the relevant solution information is stored in the `ExploreResult`. When the acceptor is fulfilled the solution to the exploration is returned in the form of the `ExploreResult`.

In the case of planning problems the relevant information is a path from the start state to a goal state. Such a path is the list of actions (see Definition 7) required to reach a goal state from the start state. In this work we limit acceptors to finding at most a single path.

In the case that exploration halts without fulfilling the acceptor, there is no solution and thus the `ExploreResult` would be empty. Textual representation of the an solution is not trivial. An action consists of a rule and match. A rule can be easily represented by its unique name (a string), however for a match this is not straightforward. A match μ can be uniquely identified by is its root node and edge set, which represents $im(\mu)$. Unfortunately, this is textually not very readable. This work, textually presents the (limited) solution by printing only the names of all applied rules.

4.1.3 EXPLORATION STRATEGY

The left branch of Figure 4.1.1 specifies the solution approach component of graph transformation planning problems. In Section 2.3.2 we have made the implementation choice to use forward state space exploration, and in Section 2.2 we have described the relevant aspects of this exploration.

This component consists of two elements, an exploration strategy and a heuristic function $H(G)$ which may be used to guide exploration. Exploration already exists within GROOVE (as shown in Section 2.4.1). It is implemented in the same way that exploration is represented in this work. In Section 4.1.1 we express how an acceptor is modeled in GROOVE and in Section 4.1.2 we express the result of exploration in GROOVE.

GROOVE currently only supports uniformed exploration strategies (i.e. without heuristics). Examples of these are breadth-first search and depth-first search. These basically specify the order in which states are generated. We extend these strategies with the greedy best-first strategy (see Section 2.2.2) which makes use of heuristic functions. There is currently no support for heuristics in GROOVE. We will implement the heuristic schemes from Chapter 3. This is discussed in the following section (Section 4.2).

4.2 HEURISTIC FRAMEWORK

In this section we discuss the software development objectives and design goals for implementing graph heuristic in GROOVE. We consider design aspects which are important to developing a framework for defining heuristic functions. Furthermore, we examine how heuristics will practically be incorporated within the existing exploration components in GROOVE. Finally, we present the design choices and patterns used to achieve our objectives and design goals.

4.2.1 OBJECTIVE

"A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software." [14]

Our objective is to develop a framework which lets one generate heuristic functions from the set of implemented heuristic schemes and to allow straightforward extension of this set by implementing additional heuristic approaches and schemes.

A sub-objective is; it should be possible to combine heuristic schemes to generate hybrid heuristic functions using a range of operators. Examples of such operators are; addition, $min()$ and $max()$. Furthermore, it should be possible to combine and order the values of heuristic schemes to a single output. Such an ordering of values may be done by a lexicographic ordering. In addition to these features it should be able to add weights to heuristic schemes and normalize heuristic values with respect to the values of other heuristic functions within a hybrid function.

The heuristic function is a variable of the exploration strategy, which for each generated state calls a calculate method which calculates its respective heuristic value. An informed exploration strategy should thus be instantiated with a heuristic function.

The two main criteria we aim to achieve in our implementation are extensibility and flexibility. These criteria can be seen as implied specifications of a framework. In following sections we present an overview the how the framework is implemented

and discuss design choices which aim to accomplish our framework criteria. In Section 4.2.2 we give a high level class diagram of the classes which form part of the heuristic framework in GROOVE and show their dependencies. We explain the functionality and relations between classes and use this as a reference point to show how certain design choices were implemented. In Section 4.2.3 we highlight design choices we have made in the implementation to achieve an extensible and flexible framework. Furthermore, we provide some additional information certain implementation aspects.

4.2.2 FRAMEWORK OVERVIEW

Figure 4.2.1 shows a high level class diagram of our framework implementation. In this section we discuss the responsibility of the classes and there relation to each other.

The **`GBFSStrategy`** class is an extension of the preexisting strategy framework in GROOVE. What differentiates this strategy class from others in GROOVE is that it has a `Heuristic` which it uses to calculate the heuristic values for states during exploration.

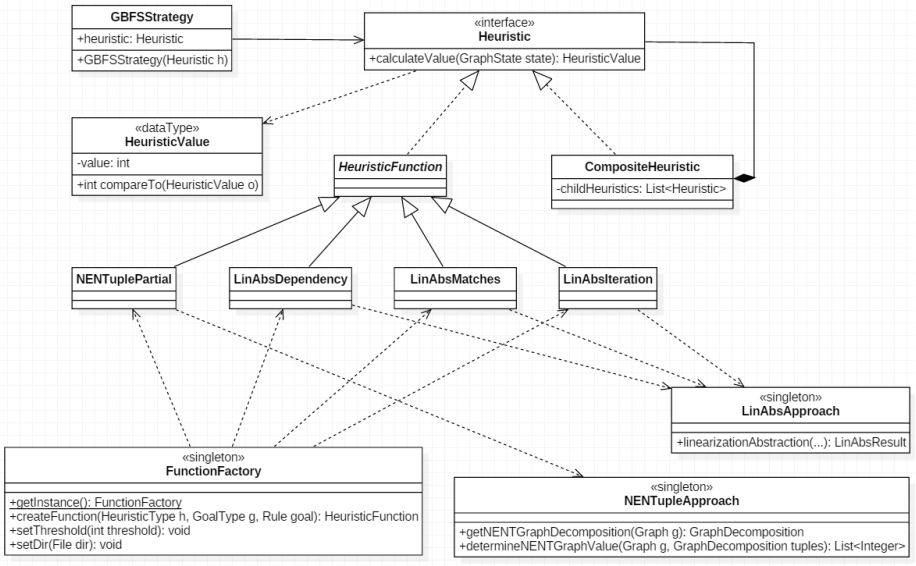


Figure 4.2.1: Class Diagram of Heuristic Framework in GROOVE

The **`Heuristic`** interface is the abstractly models a heuristic function $H(G)$ as defined in Definition 11. The interface contains a single method `CalculateValue()` which given a state q returns the heuristic value corresponding to G_q (the graph G_q can be obtained from the `graphState` object) and a predefined goal. The method returns the heuristic value for that graph in the form of the `HeuristicValue` object. By using an interface to abstractly model the general functionality of a heuristic function we

allow for a lot of flexibility in the actual implementation of a heuristic function as well as simply extensibility. This interface in combination with `CompositeHeuristic` and `HeuristicFunction` form a composite pattern which provide further flexibility in defining heuristic functions. More details on the benefits of this pattern are given in Section 4.2.3.

The **`HeuristicValue`** class models the heuristic value data type. It extends the `Comparable` interface.

The **`CompositeHeuristic`** class is used to model a collection of heuristic functions. This class forms part of the heuristic function composite pattern. In its current form it models a list of `Heuristic`s and for its heuristic function is the sum of the list its heuristic functions. The purpose of this class is to model hybrid heuristic functions. However, currently the class is only a stub as these capabilities are not yet implemented.

The **`HeuristicFunction`** abstract class is used purely to create a level of separation between the composite design pattern and the concrete implementation of leaf heuristic function classes one level below. This is done so that there is a clear separation of concerns between the pattern and implementation and furthermore to reduce the cluttering effect with the addition of new leaf heuristic functions.

The **`NENTuplePartial`**, **`LinAbsDependency`**, **`LinAbsMatches`** and **`LinAbsIteration`** classes are concrete implementation of `Heuristic` interface. These classes form the leafs of the composite pattern.

Currently, we have implemented 7 heuristic functions in 4 classes. These are the NEN heuristic scheme with a partial graph goal (Equation (3.2.4)), and all schemes for the linearization abstraction metrics with goal types of negation partial graph and partial graph (Equation (3.3.4), (3.3.5) and (3.3.6)). In the case of linearization abstraction, the two mentioned goal types evaluation of the goal is implemented in the same manner and therefore the two corresponding heuristic schemes can be implemented using a single class.

The **`NENTupleApproach`** and **`LinAbsApproach`** classes are singleton classes which implement the computational aspects for their respective heuristic approaches. The instance of these classes are used by the leaf heuristic functions to compute required input variables to calculate a heuristic function.

Concretely, `NENTupleApproach` contains two methods which are used to calculate the NEN decomposition of a graph (Equation (3.2.3)) and to calculate the difference between two NEN tuples. `LinAbsApproach` contains a method which is the implementation of the linearization abstraction algorithm given by Algorithm 3.

The **`FunctionFactory`** is a factory class which is used to instantiate heuristic functions.

4.2.3 DESIGN CHOICES

In this section we discuss in some more detail design and implementation choices we have made for the heuristics framework in GROOVE.

COMPOSITE PATTERN

We use the structural composite pattern to model heuristic functions in GROOVE. This patterns consists of the `Heuristic` interface, `CompositeHeuristic` composite class and the `HeuristicFunction` abstract leaf class (implemented by concrete heuristic function classes). The reason for the use of this pattern is the high level of extensibility in both defining new heuristics (creation of a new leaf class) or the combination of heuristic functions to form hybrid heuristics (by extending `CompositeHeuristic`). The composition of heuristic functions allows one to in a easy and centralized manner define hybrid heuristics (as discussed in the objective section).

HEURISTIC VALUE

While in Definition 11 we define a graph heuristic value to be a number (specifically $\mathbb{R}_0^+$) we have implemented the heuristic value in GROOVE as a data type object. the class `HeuristicValue` implements the interface `Comparable` and has a single Integer attribute `value`. The reason we have implemented the heuristic value as an object with a `compareTo()` method is so that it can easily to extends to include a range of values (possibly of different types) and still be evaluated. This allows for the formation of hybrid heuristic functions which rely on a lexicographical ordering of individual heuristic functions within the hybrid function.

Thus, we create an extensible environment in which additional functionally in terms of what attributes in heuristic values are evaluated and how they are evaluated. This can be achieved by creating a subclass of `HeuristicValue` designed to represent heuristic values for a specific hybrid function.

SEPARATION OF COMPUTATION CONCERNS

We have chosen to create a separation between the functionality of heuristic functions and the logic required to implement this functionality. There are three reasons we have chosen to do this. The *first* is the reuse-ability of code. Several concrete heuristic functions require the same calculations to be made (specifically linearization abstraction Algorithm 3 for all linearization abstraction heuristics). The *second* reason is that by separating the logic extension and alteration of specific approaches becomes easily without the need to alter (possibility multiple) heuristic function classes. The third reason is that it architecturally creates a neat separation between the definitions of heuristic schemes as given for example by Equation (3.2.4) and (3.3.5), and the algorithms required to calculate the needed input variables.

HEURISTIC FUNCTION ENCODING

GROOVE uses string encodings to represent all components of exploration, examples are exploration strategies and acceptor types. At point of instantiation these strings are parsed by its respective encoding parser and then instantiated with the parameter defined in the encoding. We have chosen to extend on this design approach for heuristic function encoding.

Foremost, we have introduced a new strategy value encoding `gbfs` which encodes the `GBFSStrategy`, additionally we define that this encoding is parameterized with a heuristic encoding. This sting encoding looks as follows: `gbfs:HEUR`, where `HEUR` is a string defined by the syntax in Listing 4.1.

Listing 4.1: Heuristic Encoding Syntax

```
HEUR := la_X(TYPE,rulename>threshold) | nen(TYPE,rulename);
TYPE := c | p | n;
   X := i | m | d;
```

In Listing 4.1 we currently have defined a syntax in which it is possible to encode either a NENTUPLE or linearization abstraction heuristic scheme, where a goal is in the form of a graph and modeled as a graph transformation rule in GROOVE. This syntax currently only offers the minimal functionality in terms of defining heuristics. However, by defining a formal syntax (and corresponding parser) we offer a platform in which expanding the functionality is straightforward. Such expansion could be considered encoding of additional heuristic schemes or encoding of possible functions to define hybrid functions.

Currently, we have implemented a straightforward string parser which is only loosely correlated (i.e. no direct connection) to the syntax we have defined in Listing 4.1. So, we provide simply a proof of concept in this work of the possibilities of using a heuristic encoding. In order to make this approach better suited to extensibility and flexibility (i.e. easily updating or redefining the syntax) a parser should be used which is concretely linked to the defined syntax. Note that this is possible in GROOVE as there exists a framework for defining a syntax and corresponding parser in GROOVE.

HEURISTIC INSTANTIATION

We use the creational factory method pattern for the instantiation of a heuristic function. In the parsing of the encoded heuristic an instance of the `HeuristicFactory` is created and used to instantiate the heuristic corresponding to the encoding. The `createFunction()` method of the factory has a heuristicType, goalType and goal parameter. These are used to determine the correct heuristic approach and scheme and then to initialize the scheme with the goal. To do this we use a nested switch function. Furthermore, we include additional set methods for parameters which might be approach specific. Concretely, in the implementation of linearization abstraction we require the directory of the Grammar which represents the problem and an iteration threshold. In Figure 4.2.1 the relation between the factory and the respective concrete heuristic functions can be seen.

4.3 IMPLEMENTATION OF ABSTRACTION

In Section 3.3.1 we introduce the concepts of abstract graphs (Definition 15) and abstract graph transformations rules (Definition 17) as well as how to abstract graphs and rules (Definition 16 and Definition 18). In this section we discuss how we have implemented this abstraction in GROOVE. There are two important features which characterize this abstraction. The first is the introduction of the maybe element sets in abstract graphs. The second is the mapping of elements between graphs within a subset (for example the maybe or real set), this is important in graph transformation rules and abstract satisfaction (Definition 19).

In GROOVE we have chosen to not fundamentally implement abstraction but instead simulate it using the current programming and modeling capabilities. The reason for this is that such a fundamental implementation would require substantial programming work to extend GROOVE functionality to support abstraction. Furthermore, without known the effectiveness of heuristic functions based on this form of abstraction a great deal of fruitless work may have been done. Therefore, instead we have designed an approach which simulates the abstraction using modeling features of GROOVE.

We simulate abstract graphs in GROOVE by extending the graph with additional node and edge labels which we implicitly consider labels of nodes and edges in the maybe sets of the abstract graph. To do this we assume the production system has a type graph (see Section 2.4). We extend the type graph of production system by introducing “abstract" nodes and edges. This is done by creating new element types which are labeled *label+“SUB"* to indicate an element in the maybe set. In the first row of Table 4.3.1 we show how this is modeled in the GROOVE type graph.

We now discuss how we can differentiate between maybe and real elements in an abstract graph simulated in GROOVE. In terms of nodes we use sub-typing. Sub-typing is a functionality in GROOVE graphs which allows for the sub-typing of node types to other node types. Concretely, we simulate maybe node elements as sub-types of all nodes. Intuitively, given an abstract graph, a node with its original label (i.e. *node*) is in the graphs total node set and a node with the label *nodeSUB* would be in the maybe node set. Given an abstract graph $\hat{G}$, we can reference a node by referring to the super node, we can reference a maybe node by referring to the sub node and finally we can reference a real node by referring explicitly only the super node. For edges there is no such sub-typing mechanism, therefore we must use an edge formula $?[edge, edgeSUB]$ to reference an edge, we can reference a real or maybe edge by referring to the respective edge label. Thus, we shown how we can represent an abstract graph $\hat{G}$ such as defined in Definition 15 in GROOVE.

We now show how we simulate graph transformations rules abstaction in GROOVE using the deliberate approach to simulate abstract graphs given above. In Table 4.3.1, one can see how the abstract version of each possible relavent aspect of a rule simulated in a GROOVE rule model. The combination of these aspects simulate a graph transformation rule abstraction as given in Definition 18. In the case of NACs (which corresponds to embargo objects in GROOVE) we require that the elements in a NAC n are mapped to real elements in an abstract graph (see Definition 19). This is simulated by ensuring embargo elements reference only elements in the real sets of an abstract

graph.

We have chosen to implement this simulation of abstraction in GROOVE at the point where a GROOVE model is interpreted as a grammar (i.e. production system) in the code. The class `GrammarModel` is responsible for parsing and creating graph and transformation rule objects in GROOVE. We have extended this class creating `AbstractGrammarModel` which creates an abstracted version of a production system.

Specifically we override the method `createGraphModel()` that has as input an `AspectGraph` which is an object representing a graph. The `createGraphModel()` creates `GraphModel` objects from these `AbstractGraph`s. The `AspectGraph` models all forms of graphs which can be modeled in GROOVE, such as a graph, a type graph or a rule graph (see Section 2.4 and 2.4.1). Our approach is to alter and extend these `AspectGraph`s before they are used to create graph models. We do this by rewriting element labels and creating/deleting elements in accordance to the transformations shown in Table 4.3.1.

By simulating the abstraction using the creation of additional labels for maybe elements and the referencing of specific maybe, real and total elements in abstract graph transformation rules as shown in Table 4.3.1 we can emulate the abstraction that we have defined for the linearization abstraction heuristic approach. Furthermore, using this approach we can use the existing match finding functionality in GROOVE to for abstract graph transformations.

While this implementation captures the abstraction defined in Section 3.3.1, it does limit some of the functionality of modeling in GROOVE. An example of this is that we require problems to be modeled using a type graph. Furthermore, the rewriting of labels in `AspectGraph`s is done on a case-by-case basis such that more complex modeling syntax cannot be altered correctly. Similarly, there are certain issues that arise with preexisting sub-type structures in a standard type graph. These however are GROOVE and implementation limitations and not limitations of the abstraction in general. To fully incorporate linearization abstraction it would be best to implement such functionality in the fundamental basis of the code instead of using a simulation. However, as discussed at the beginning of this section, such an implementation falls outside the scope of this work.

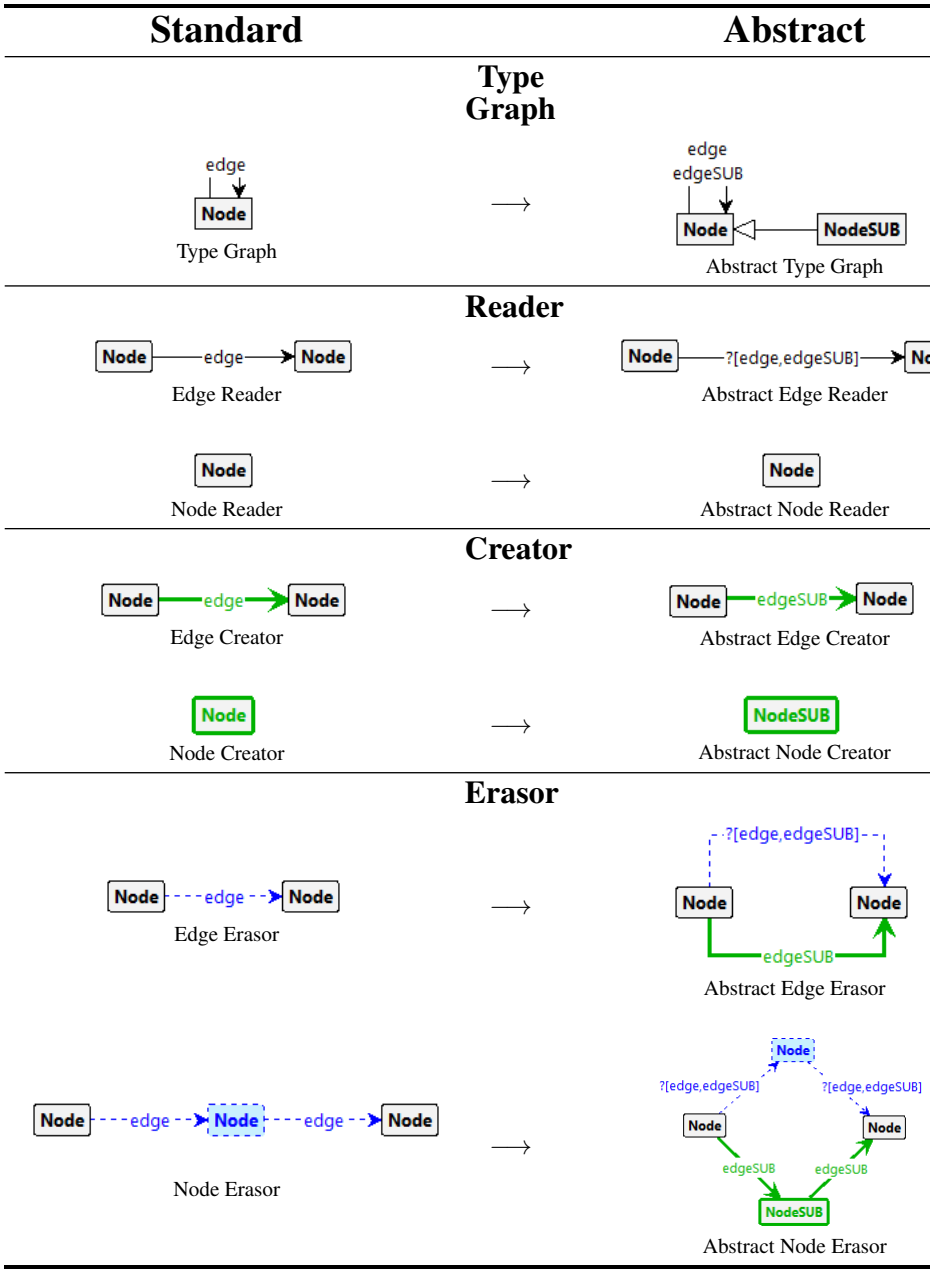


Figure 4.3.1: Transformations for Abstract of Type Graph and Rules in GROOVE

CHAPTER 5

EVALUATION

The previous chapters have designed domain independent graph heuristic functions to aid state space exploration to solve graph transformation planning problems. Furthermore, we have shown how these heuristics, as well as planning in general, are implemented in the tool GROOVE. In this chapter we will evaluate the effectiveness of these heuristics on a range of problems.

The evaluation chapter is divided into four parts.

In Section 5.1 we present the planning domains we will use to evaluate our heuristics. Furthermore, we define some distinguishing features of planning domains and hypothesize on the correlation of these features and the effectiveness of our heuristics approaches.

In Section 5.2 we define the metrics we use to evaluate the effectiveness of each heuristic. Furthermore, we give our experiment approach and setup.

In Section 5.3 we present the results of the experiments for each problem domain. For each domain we discuss the results and relate them to the hypothesize we made in Section 5.1. Also, we provide a comparision of our results to those of related works and to a competitive planner tool using the PDDL language.

Finally, in Section 5.4 we give an overall discussion in which we compare the performance of our heuristics and offer some closing remarks on the results presented in the work.

5.1 PLANNING DOMAINS

The goal of this section is to give a brief introduction to the problem domains that will be used in the experiments to evaluate the heuristic schemes presented in this thesis. For each domain, if not already done in a previous chapter, we give the type graph and transformation rules as well as the start and goal graph for a specific problem instance. Furthermore, we describe how we define instances of each problem for the experiments. Finally, we discuss the distinguishing features of different domains and attempt to make a classification of domains. We aim to define a classification such that

it is possible to create assumptions on which heuristic schemes are best suited for a certain domain.

5.1.1 BLOCKS WORLD

The Block World problem domain has previously been given in Section 2.4.1.

We can alter the size and difficulty of a problem in Block World by altering the number of blocks in the world or by the number of possible colors. For the experiments we will use 9 different problem sizes (4, 6, 8, 10, 12, 14, 16, 18 and 20 blocks). For problems with 14 blocks or less we use three colors while for problems with 16 blocks or more we use 4 colors. For each problem we aim to divide the total number blocks into colored blocked as evenly as possible.

For each problem size we have two problem instances which are randomly chosen initial configurations of the blocks. For each problem we use the same goal configuration which is all blocks of the same colors stacked on top of each other with the block stacked on the table. In the context of graph transformation planning problems and heuristic schemes, we express this as a partial graph goal.

For evaluation of the heuristic schemes presented in this work, we compare the experiment results of the Block World domain to those found in [5], [10] and [15].

5.1.2 SLIDING PUZZLE

The Sliding Puzzle, also known as the *N*-puzzle problem, is a common puzzle game in which n numbered tiles are positioned on an $n+1$ sized grid from a random initial positioning. The goal of the puzzle is to move the tiles such that they become ordered (in a game setting these tiles may, when ordered, form an image). There are only four possible actions one may use in the puzzle; these are "sliding" a tile up, down, left or right from its current position into an adjacent empty position.

The GROOVE type graph of the Sliding Puzzle problem for $n = 8$ is given in Figure 5.1.1.

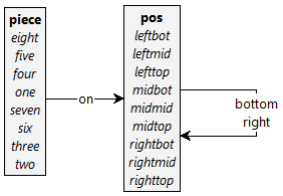
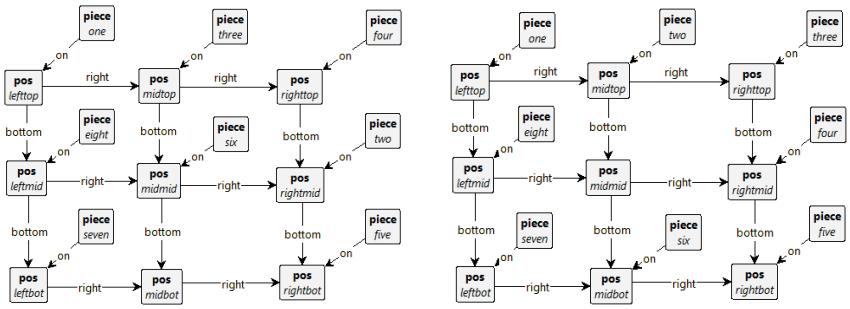


Figure 5.1.1: Type graph of 8-piece Sliding Puzzle problem

Here we define piece nodes which represent tiles and pos nodes which represent positions on the grid. Furthermore, a tile has an on-edge to a position and each position has an edge to its right and bottom neighbor, in this way it knows its relative position. We use flags to uniquely label pos nodes and we number piece nodes.

Figure 5.1.2a and 5.1.2b show a start and goal graph for an 8-piece Sliding Puzzle problem.

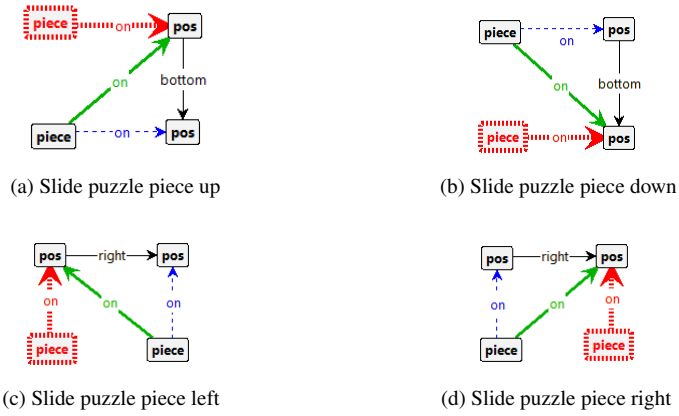


(a) Start graph

(b) Goal graph

Figure 5.1.2: Start and Goal graph of an instance of Sliding Puzzle problem

In Figure 5.1.3 we can see the 4 actions allowed in the Sliding Puzzle problem modeled as graph transformation rules.



(a) Slide puzzle piece up

(b) Slide puzzle piece down

(c) Slide puzzle piece left

(d) Slide puzzle piece right

Figure 5.1.3: Transformation rules for Sliding Puzzle problem

The complete state space of a Sliding Puzzle can be calculated and is dependent on the number of tiles. The formula for the state space size is $n!/2$, this corresponds to half of all possible numbered tile placements. The reason we need to divide by half is that only half of the initial placements are solvable (i.e. lead to the same goal state) [16]. In the case of the 8-piece Sliding Puzzle problem the states space is thus $8!/2 = 20160$ states, for the next puzzle size $n = 15$ the state space becomes a staggering amount (slightly over half a trillion states).

For the experiments we will consider the 8-piece Sliding Puzzle with four difficulties. We rate these difficulties easy, medium, hard and worst. These difficulty levels correspond with the number of steps required to find a goal state from the initial state. For each of these cases we consider the same goal, which is given as a graph in Figure 5.1.2b. While this may also be considered a complete graph goal we will consider it

a partial graph goal. This is due to the fact that complete graph goal heuristic schemes have not been implemented in GROOVE yet.

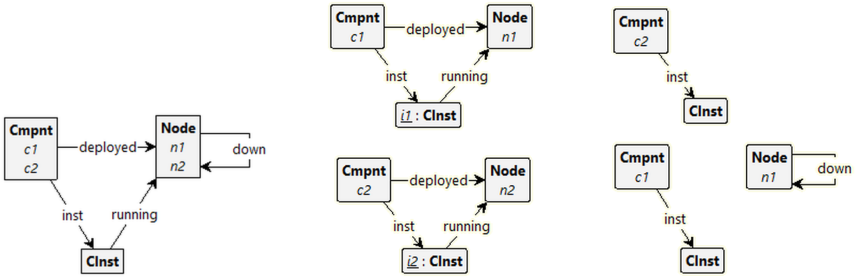
Also, we will experiment with solving planning problems in the $n = 15$ size Sliding Puzzle domain. For these problems we define instances by first defining the goal graph which is the ordering of tiles starting from the top left most position and moving right. From this goal state we generate initial states by making 100 random moves, the planning problem thus becomes to find a path back. We create four problem instances using this approach and similarly to the $n = 8$ variant we define the goal in the planning problem as a partial graph goal.

For evaluation of the heuristic schemes presented in this work we can compare the experiment results of the Sliding Puzzle domain to those found in [6] and [7].

5.1.3 Electronic Control Units

The Electronic Control Units (ECUs) is a problem domain taken from work in [5]. It is an application example which considers the reconfiguration of Electronic Control Units in automotive systems. In essence, the domain deals with the runtime reconfiguration of software components which run on a Runtime Environment (RTE) middleware, which serves as link that connects software components to Basic Software (BSW) that controls hardware. The problem in terms of planning is to find a plan to reconfigure the system in case of hardware failure (i.e. shutdown an ECU) or rebooting of a RTE after a software upgrade (i.e. redeployment in soft real-time).

We model the ECU domain using 3 components; **a)** Node, which represent ECU modules that may be shut down, **b)** Cmpnt, which represent software components that are deployed on ECUs and **c)** CInst, which represent instances of software components running on ECUs. In Figure 5.1.4a we give a type graph representation of the problem modeled in GROOVE. We use flags to uniquely define Cmpnt and Node instances. Furthermore, in Figure 5.1.4b we give an initial scenario of an ECU problem. In this scenario there are two ECUs, both with a single software Cmpnt deployed on them and both Cmpnts running an instance of the software on their respective ECU.



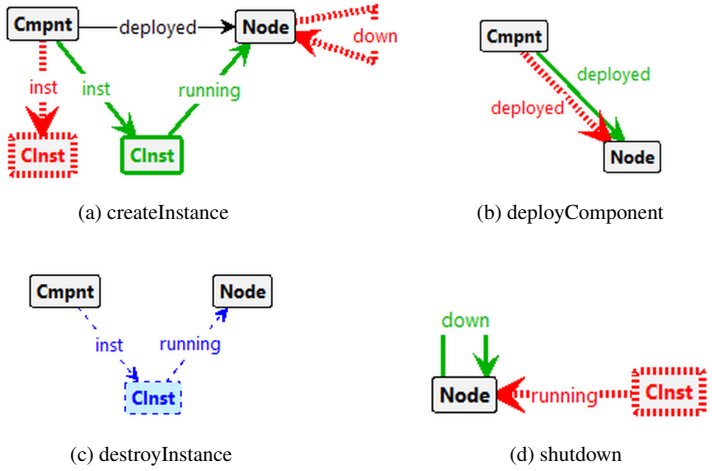
(a) Type graph (b) Start graph (c) Goal graph

Figure 5.1.4: Type graph for ECU problem and instance Start and Goal graph

In Figure 5.1.4c we give a possible goal (in the form of a partial graph) for the ECU problem scenario of Figure 5.1.4b. The goal expresses that ECU $n1$ should be

shut down and both software components $c1$ and $c2$ should have an instance running.

Figure 5.1.5 show the graph transformation rules relevant to the ECU domain we are modeling. There are four different transformations. The first rule, given in Figure 5.1.5a, models the instantiation of a software instance for a respective software component and ECU, this instantiation is only valid if the ECU is not shut down and the component is not already running an instance. The second rule, given in Figure 5.1.5b, models the deployment of a software component on an ECU, deployment is only valid if the component is not already deployed on said ECU. The third rule, given in Figure 5.1.5c, models the destruction of a software instance of a component running on some ECU. The forth and final rule, given in Figure 5.1.5d, models the shutting down of an ECU, shutdown is only valid if no software instances are running on it.



(a) createInstance

(b) deployComponent

(c) destroyInstance

(d) shutdown

Figure 5.1.5: Transformation rules for ECU problem

For the experiments in the ECU domain we use 4 different problem sizes. The problem size is determined by the number of ECUs in the problem, in our experiments we have problems with 2, 3, 4 and 5 ECUs. For each problem size we have two different instances (v1 and v2). The fist group of instance v1 have same number of component instances running in the initial configuration as ECUs that are available. Figure 5.1.4b, for example, is the start graph of problem instance ECU2-v1. The second group of instance v2 have additional components and instances running, equal to the number of ECUs available divided by two and rounded down. Additional components allow for several more applications of transformation rules 1-3, so the problem difficulty between instances is greatly increased.

As goal for each start instance we define that half the total number of ECUs must be shut down and that every component must have an instance. In the context of graph transformation planning problems we define this as a partial graph goal. For the problem instance ECU2-v1 the corresponding goal is given in Figure 5.1.4c.

For evaluation of the heuristic schemes presented in this work we can compare the

experiment results of the ECU domain to those found in [5].

5.1.4 Domain Distinctions

In this section we present and discuss features that we have identified as distinguishing features of graph transformation planning problems. We hypothesize that certain heuristics schemes are better suited to solving planning problems with specific features compared to other schemes. In the following subsections we introduce and discuss the following three features: **a)** type of actions, **b)** dependency of actions and, **c)** goal expressiveness. We will relate these features to the planning domains used in the evaluation. Furthermore, we will speculate on how these features relate to the effectiveness of heuristic approaches we defined in Chapter 3.

Classic vs. Dynamic Planning Problem

We define classic planning problems as planning problems with a predefined element set. Intuitively, these are problems which do not involve element instantiation or destruction. Thus, in the problem start state, all elements are already defined and actions only change the properties and relations of these elements. In terms of graphs, in a classic planning problem the start graph G_{q_0}, contains all nodes that will ever exist in any graph G_q and actions only create and delete edges between nodes.

This problem type stands in contrast to dynamic planning problems. We define these as planning problems in which elements may be instantiated or destroyed. Thus, the size of the state may differ within the state space of the problem. Intuitively, this corresponds to actions which create or delete elements. Thus, dynamic graph transformation planning problem contains graph transformations which create (thus instantiating) or delete (thus destroying) node elements. As mentioned in Section 2.3.1 modeling dynamic planning problems is not possible in PDDL and is an advantage of using the graph transformation modeling paradigm.

While both heuristic approaches in this work seem appropriate for classic and dynamic planning problems, we hypothesize that exploration using the linearization abstraction heuristic approach is better suited for dynamic problems and NENTuple approach is better for classic problems. The reason for this lies in the techniques used to evaluate a state to determine its heuristic value.

The NENTuple approach creates an abstraction of a graph and goal to NEN tuples, and compares the two according to the specific heuristic scheme. This fits well for classic problems because the graph size in classic problems is set in terms of nodes. Furthermore, actions directly influence the NEN decomposition of a graph since they only create or delete edges in a graph. These two properties of classic problems mean that, the NEN decompositions of the graphs of two relative states are likely to vary and thus, depending on the goal, have a different heuristic value. Considering that the NENTuple approach has an meaningful distance estimation, this means that during exploration, using GBFS, we will not have a lot of unnecessary branching due to heuristic values being equal.

The linearization abstraction approach creates an abstraction of the whole planning problem and executes this with independent actions in parallel. The result of this abstract execution is then used to calculate a heuristic value. By simulating, although in an abstract form, the planning problem we create an abstracted form of the state space. This is well suited for dynamic problems since we also simulate the growth of a graph due to the instantiation of elements.

Both the Block World and Sliding Puzzle domains are classic planning problem domains. Both begin with a predefined set of node elements and actions simply alter the relation between these nodes by creating and deleting edge elements. The ECU domain is a dynamic planning problem domain. In this domain we have actions which either create or delete CInst nodes.

DEPENDENT VS. INDEPENDENT ACTIONS

In Section 2.1 we define the concept of an action, with respect to graph transition systems, as an element of the transition relation. Thus, in graph transformation problems at graph level an action corresponds to a graph transformation rule and a match. The dependency of actions is a subjective measure of how much actions rely upon each other to be applicable. Intuitively, we say actions are independent if they are always applicable and dependent if they require other actions before being valid. Correspondingly, independent actions cause branching in the state space of a problem and are one of the causes of state space explosion.

We hypothesize that the dependence of actions has no relevance to the NENTUPLE heuristic approach. This is due to the fact that this approach is not concerned with actions and thus should not be effected by their dependency. However, inaccuracy in the distance estimation and branching may cause the unnecessary generation of irrelevant states when using NENTUPLE heuristics functions.

We believe that the linearization abstraction heuristic approach is better suited to problems with fewer independent actions. This is due to the fact that the linearization abstraction approach simulates an abstraction of the problem. Due to the way we define the abstraction of transformation rules we relax the dependency between actions by relaxing the requirement for a valid match. This causes an increase in the abstract graph and state space size. We attempt to reduce the abstract state space by applying independent actions in parallel. However, finding all these matches for each abstract graph transformation rule still costs some time. So by having a greater number of matches we require additional computation time which makes linearization abstraction heuristic functions less time effective for problems with a large number of independent actions.

In terms of graph transformation planning problems, in order to estimate the dependency of actions in a problem one should not only consider the rule set of the problem but also possible matches of those rules to graphs corresponding to states of the state space. With respect to the problems used in the evaluation of this work, we see that the problems of the Blocks World domain contains relatively many independent actions.

As the problem size grows, actions become less dependent as more matches for each transformation rule become possible. The Sliding Puzzle domain has four transformation rules which are independent. However, we see that for each rule only a single match may exist in every possible state. Thus, branching is kept to a minimum. The ECU domain in contrast has some dependent actions, specifically actions with the createInstance and destroyInstance rule are dependent with respect to matches on the same Cmpnt and Node elements.

GOAL EXPRESSIVENESS

In this feature we only consider goals in the form of graphs. Goal expressiveness is the measure of how complete the goal represents a goal state. Intuitively, a complete graph goal is very expressive while partial and negation partial graph goals may vary in expressiveness.

Goal expressiveness is an important feature in heuristic functions which compare the input graph to the goal graph. This is the case in with the NENTUPLE heuristic approach. However, it is not relevant in the linearization abstraction approach, since the goal is not used in a comparison, but only as part of MATCH function.

For the NENTuple approach we hypothesize that corresponding heuristic functions are better suited for problems with expressive goals. This is since the NENTuple approach compares the input graph to the goal graph. The more expressive the goal is the more complete the NEN decomposition and thus the more accurate the distance measurement between the two.

For the problem domains used in the evaluation we see that the Block World and Sliding Puzzle problems both have very expressive goals. In both cases, the goal type is a partial graph, however, they both express most or the complete graph representation of a goal state. Note, we have purposefully chosen to make these goals so expressive in order to get the best results for our planning problems. From preliminary experiments we have been able to confirm our hypothesis with respect to goal expressiveness.

The goal of the problems in the ECU domain are greatly less expressive as they only specify partial relations of elements of a goal state. This allows for easier satisfiability of a goal.

COMBINATIONS AND CONCLUSIONS

A planning problem can thus be classified, but is not limited, by the features discussed above. While we believe that these features affect the suitability of a heuristic approach, there may be other unconsidered aspects which would overturn our hypotheses. Furthermore, it is in no way clear in which priority these problem features influence a heuristic approaches effectiveness.

Finally, a final difference determining which heuristic approach is more suitable for a problem is that, the linearization abstraction approach is computationally more demanding than the NENTUPLE approach. What is means is that while we expect linearization abstraction, specifically using the dependency count metric, to be the most

accurate distance measure, it is also the most time expensive. So, in considering the heuristic scheme for a specific problem one should consider the potential problem size compared to the solving time. Concretely, it may be more feasible to use the NENTUPLE approach and generate a greater number of irrelevant states but doing it relatively fast instead of using the linearization abstraction approach which generates fewer irrelevant states but takes a much longer time doing so. We call this the time-accuracy trade-off.

5.2 METRICS & MEASUREMENT

The aim of this section is to define the metrics we will use to evaluate the effectiveness of the heuristics proposed in the thesis on different problems and varying problem sizes. We also explain how these metrics are obtained from running a planning problem in GROOVE and what measures we take to ensure accurate results. Finally we give an overview of the system on which the experiments are performed.

5.2.1 EVALUATION METRICS

5.2.1.1 NO. OF STATES

No. of states represents the number of states generated during exploration. This is thus the subspace of the GTS explored in search of a goal state. This metric is a measurement of how well the heuristic used in exploration prunes the state space. Furthermore, this metric gives an indication of the memory use of a planning problem given the heuristic.

5.2.1.2 EXPLORATION TIME

Exploration time represents the time required to complete exploration and thus solve the planning problem. This metric gives is a measurement of how fast an exploration finds a goal state given a heuristic.

5.2.1.3 PATH LENGTH

Path length is simply the length of the path given as result of the planning problem. This metric is a measure of how quickly (in terms of actions) an exploration reaches a goal state given a heuristic if it never takes a wrong branch. Note that in these experiments we are not interested in the shortest path so it may be that a path is rather inefficient and devious.

5.2.1.4 (PATH LENGTH : NO. OF STATES) RATIO

The path length to number of states ratio is a metric which gives a measurement of how effectively a heuristic is in finding relavent states. The lower this ratio, the more "on-track" an exploration is, and thus the fewer unnecessary calculation are needed.

5.2.2 MEASUREMENT APPROACH

In GROOVE the planning is executed by performing an exploration of the production system of the planning problem. As input this requires a production system, exploration strategy (with heuristic) and goal. As exploration strategy we use greedy best first search as described in Section 2.2. The production system, heuristic, and goal are variables which we change during experimentation.

The measurement of the experiment is done by calculating the value of each metric for every planning problem. The number of states and path length are elements of result of the planning problem. The runtime of the planning is calculated by measuring the time needed to execute the key component of the exploration (i.e. the function `play()`) in the `Exploration` class in GROOVE. Finally, the path length to number of states ratio is calculated after the experimentation results are obtained.

The experiment is set up such that first a few (between 1 and 5) initial runs are executed to allow the JIT compiler to run, then several more (between 3 and 10) real runs are executed. Note that the exploration in GROOVE is deterministic (i.e. the same problem instance explored multiple times produces the same state space) so the number of states and path length in each run is the same. The exploration time for a single problem is calculated by taking the average runtime of all real runs excluding the longest and shortest real run. The number of initial and real runs are varied according to problem size, as the size of the problem increase so does the runtime and the difference between runtimes becomes less significant. Therefore, for larger problems the total number of runs is fewer than for small problems.

For the results in Section 5.3 we only present the average values of each metric for every problem instance.

5.2.3 EXPERIMENTAL SETUP

The experiments were conducted on a laptop computer with an Intel i7-3517U CPU with 8GB of RAM. The experiment were executed on a JVM running on the computer with VM option `-XX:SoftRefLRUPolicyMSPerMB=10` to avoid excessive garbage collection for larger models and state spaces.

5.3 RESULTS

In this section we present and discuss the results of the experiments used to evaluate the effectiveness of the heuristics presented in this work. Each subsection corresponds to a planning problem, and for each we will discuss the results for the chosen evaluation metrics, reflect on our expectations as given in Section 5.1.4, and finally compare these results to those of other works. The results from other works for the planning problems also used in this work are given in Appendix A.

Finally, in Section 5.3.4 we compare the results achieved for the problems modeled as graph transformation planning problems compared to PDDL.

For the planning problems we evaluate the effectiveness of three different heuristic functions. These are the NENTUPLE heuristic scheme, and the linearization ab-

straction heuristic scheme using the match count and dependency count, which are all instantiated for a goal of the type partial graph goal. We denote greedy best-first exploration using these heuristics as NEN, LA-M and LA-D respectively. Furthermore, we use LA to represent the use of the linearization abstraction heuristic approach in GBFS exploration in general.

5.3.1 BLOCKS WORLD DOMAIN

Table 5.3.1 presents the results for each metric of the Block World planning problems. Furthermore, Figure 5.3.1 and Figure 5.3.2 plot the number of states and exploration time metric respectively for each problem size and heuristic function.

Primarily, in Table 5.3.1 we can see that NEN is able to solve all problems, while both LA approaches are only able to solve smaller instances of domain. Furthermore, we can see that for the instances solved using LA are outperformed in each metric by NEN. We expect this to be due to the fact that this is a classic problem and the actions in this problem are independent, which we theorized would be disadvantageous for the linearization abstraction heuristic approach.

Table 5.3.1: Blocks World domain planning results

	NEN				LA-D				LA-M			
	#States	Time (s)	|Path|	ratio	#States	Time (s)	|Path|	ratio	#States	Time (s)	|Path|	ratio
Blocks-4-v1	15	0.02	6	2.50	15	0.20	6	2.50	16	0.23	6	2.67
Blocks-4-v2	38	0.04	8	4.75	24	0.19	10	2.40	35	0.23	10	3.50
Blocks-6-v1	35	0.02	12	2.92	62	0.59	14	4.43	147	1.92	16	9.19
Blocks-6-v2	31	0.02	8	3.88	97	1.35	16	6.06	147	2.11	18	8.17
Blocks-8-v1	94	0.04	18	5.22	175	6.36	30	5.83	212	5.84	22	9.64
Blocks-8-v2	109	0.05	16	6.81	114	3.80	28	4.07	313	9.51	36	8.69
Blocks-10-v1	141	0.06	26	5.42	488	310.32	32	15.25	-	-	-	-
Blocks-10-v2	159	0.04	22	7.23	406	243.97	34	11.94	-	-	-	-
Blocks-12-v1	179	0.05	26	6.88	-	-	-	-	-	-	-	-
Blocks-12-v2	106	0.02	16	6.63	-	-	-	-	-	-	-	-
Blocks-14-v1	186	0.14	30	6.20	-	-	-	-	-	-	-	-
Blocks-14-v2	240	0.09	28	8.57	-	-	-	-	-	-	-	-
Blocks-16-v1	180	0.10	28	6.43	-	-	-	-	-	-	-	-
Blocks-16-v2	156	0.05	28	5.57	-	-	-	-	-	-	-	-
Blocks-18-v1	226	0.06	32	7.06	-	-	-	-	-	-	-	-
Blocks-18-v2	240	0.04	32	7.50	-	-	-	-	-	-	-	-
Blocks-20-v1	347	0.05	36	9.64	-	-	-	-	-	-	-	-
Blocks-20-v2	316	0.06	38	8.32	-	-	-	-	-	-	-	-

For the instance Blocks-4-v1 one can clearly see how much slower the linearization abstraction approach is compared to NENTUPLES. This can be concluded from the fact that while approximately the same number of states are explored in each exploration the runtime for LA is a factor 10 greater than that of NEN. This becomes painfully clear in Figure 5.3.2, where on logarithmic scale we see the runtime for LA-M and LA-D increase exponentially as the problem size increases. A cause of this could be that as the problem size grows more independent actions become possible. Thus, we can conclude that the linearization abstraction heuristic approach is not suitable for this problem domain. We see that LA-D performs better than LA-M, this is according to expectations, since the dependency count metric is more accurate compared to the match count metric.

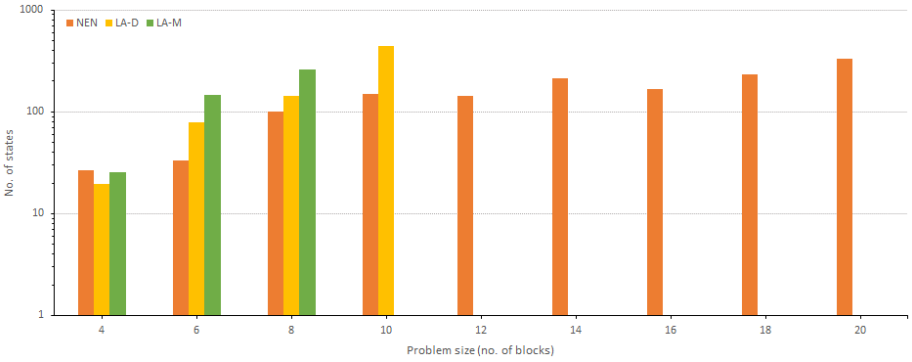


Figure 5.3.1: Average number of explored states for planning in Blocks World domain

In contrast, the NENTUPLE heuristic approach seems to perform exceptionally well. We know from Section 2.4.1 that state space grows at an alarming rate as the problem size increase. However, in Figure 5.3.1 we see that there is little to no increase in the explored state space for NEN. Furthermore, in Figure 5.3.2 we can see that there is only a minimal increase in the runtime of NEN as the problem size increases. Clearly, the NENTUPLE heuristic approach provides a significant reduction in both space and time in solving Block World planning problems.

In Table 5.3.1 we see that for NEN the path length : no. of states ratio is not constant but increases linearly as the problem size increases. What this means is that as the problem sizes grows the heuristic becomes more inefficient since it causes the generation of proportionally more unnecessary states. This is to be expected as the NENTUPLE approach provides a context-free coarse graph abstraction, and with an increase of problem size the context of the graph becomes more important. It may be concluded, however, that this increase in inefficiency is acceptable compared to runtime and NEN still outclasses its linearization abstraction counterparts.

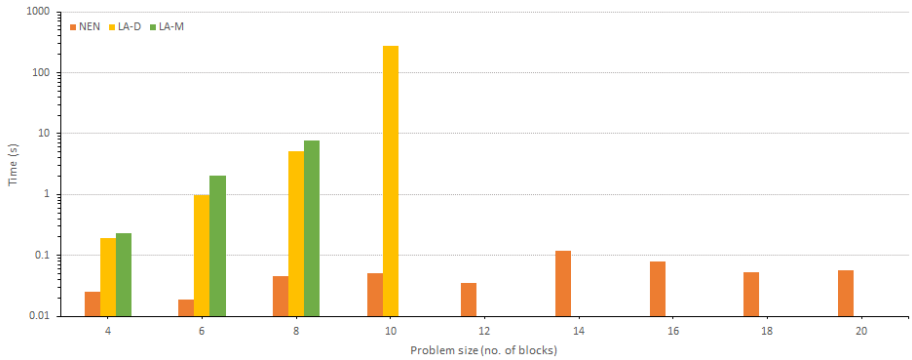


Figure 5.3.2: Average exploration time for planning in Blocks World domain

In Appendix A.1 we give the results of graph transformation planning in the Block

World domain from [5]. We cannot say anything on the similarity of individual problem instances the problem sizes match and we can therefore only make some performance comparisons. The ABS heuristic is an approach introduced in [5] and is conceptually similar to linearization abstraction. At first glance we can see that exploration using linearization abstraction presented in [5] performs poorly compared to the similar approach in [5] for the Block World domain. However, one can see that NEN vastly outperforms ABS in both number of states generated and runtime. Furthermore, the exploration runtime seems to be increasing exponentially as the problem size increases for ABS, whereas this is not the case for NEN.

5.3.2 SLIDING PUZZLE DOMAIN

In this domain, in addition to the NEN, LA-M and LA-D exploration, we also performed exploration using two additional heuristic functions. These are based on linearization abstraction and are a hybrid of the match count and dependency count metric. We combine these two metrics using lexicographical ordering. We introduce the heuristic function with DC > MC and the function with MC > DC. We refer to the greedy best-first search exploration using these heuristic functions as LA-(D+M) and LA-(M+D) respectively.

Table 5.3.2 present the results for each metric of the Sliding Puzzle planning problems. Furthermore, Figure 5.3.3 and Figure 5.3.4 plot the number of states and exploration time metric respectively for each problem size and heuristic function.

Similar to the results for the Block World domain we see that NEN outperforms all LA explorations. This is most clearly seen for the 15-piece problem instances. These problems were only solvable using NEN. However, we see that for this domain the result metrics are similar across approaches.

Table 5.3.2: Sliding Puzzle domain planning results

	NEN				LA-D				LA-M				LA-(D+M)				LA-(M+D)			
	#States	Time (s)	|Path|	ratio	#States	Time (s)	|Path|	ratio	#States	Time (s)	|Path|	ratio	#States	Time (s)	|Path|	ratio	#States	Time (s)	|Path|	ratio
8-easy	16	0.03	5	3.20	23	2.08	5	4.60	21	1.51	5	4.20	23	1.80	5	4.60	23	1.98	5	4.60
8-med	509	0.17	27	18.85	55	5.93	17	3.24	123	10.76	11	11.18	51	5.88	17	3.00	46	5.07	11	4.18
8-hard	483	0.12	28	17.25	53	5.49	18	2.94	127	11.66	12	10.58	49	4.78	18	2.72	45	4.61	12	3.75
8-worst	2567	1.06	64	40.11	763	91.18	62	12.31	1178	155.46	52	22.65	622	79.26	62	10.03	1070	132.12	56	19.11
15-v1	50659	48.44	328	154.44	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15-v2	36317	29.37	228	159.28	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15-v3	48430	44.35	226	214.29	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15-v4	10571	10.36	128	82.58	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

From the results in Table 5.3.2 one can see that NEN is much faster than any LA approach, this is shown obviously in Figure 5.3.4. There is about a factor 100 difference between runtimes. This is clearly in the advantage of NEN, however, for the number of explored states and ratio metric we see that LA performs better. What this means is that the linearization abstraction approach provides a more accurate heuristic compared to NENTUPLES. In Figure 5.3.3 one can see that, as the problem difficulty increases, the number of states explored by NEN rapidly rises.

Compared to NEN, all LA approaches have explored fewer states (excluding the 8-easy instance). The most accurate of these, LA-(D+M) is between a factor 5 and 10 more accurate. This is further emphasized by the ratio metric. These results are clearly

effected by the speed-accuracy trade-off discussed in Section 5.1.4.

In the Sliding Puzzle domain, the linearization abstraction heuristic approach is significantly more accurate and does not suffer much from independent actions for the smaller problem size (8-piece). However, for larger problem sizes (15-piece) which have a state space of half a trillion states, it becomes more effective to generate many slightly accurate states quickly. There are two other factors that are also influential. The first is that, for larger problem instances, linearization abstraction takes longer to calculate a witness set. The second is that, using GBFS, it is possible that if a state is wrongly evaluated (inaccurate heuristic value) then due to the nature of GBFS a erroneous path may be explored a long time before backtracking. These two combined cause a very long exploration time for large problem instances.

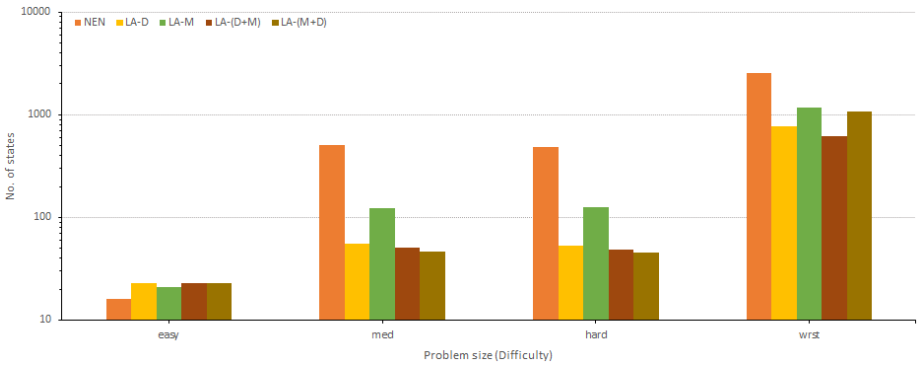


Figure 5.3.3: Average number of explored states for planning in Sliding Puzzle domain

We now consider the distinctions within the LA approach. We see that LA-D outperforms LA-M, which we also expected. This is due to the dependency metric being a more accurate distance measure compared to the matches metric. In combination with the fact that, due to problem modeling, there can only be a single MATCH found in linearization abstraction so both metrics are equally fast.

Furthermore, we see that LA-(D+M) is the most accurate. This is due to the fact that the heuristic function used is basically a refinement of a dependency count linearization abstraction heuristic function. This result shows that there may be advantages to combining heuristic schemes to define hybrid heuristic functions for exploration. Similarly, we see that LA-(M+D) shows a explored state reduction compared to LA-M. However, since the dependency count metric is more accurate than the match count metric we see that heuristic functions which first evaluate the DC perform better.

In Appendix A.2 we give the results of graph transformation planning in the Sliding Puzzle domain from [6, 7]. [7] makes use of two domain dependent heuristics and measures the number of states and transitions explored as well as the exploration time. When we compare the results using greedy best-first search we see that in terms of states (and thus accuracy) LA approaches have approximately the same accuracy range. While NEN is less accurate than the domain dependent heuristic, we see that they share the same magnitude in runtime. The limitation of these comparisons is that we do not know the exact problem instances with which [7] was performed, so a

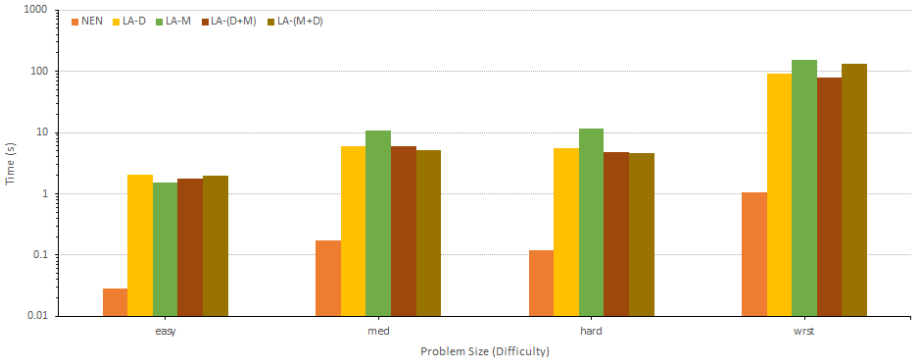


Figure 5.3.4: Average exploration time for planning in Sliding Puzzle domain

concrete comparison between number of explored states is not possible. Furthermore, comparing runtime is inaccurate due to the rounding used in [7] and different hardware.

In [6], the proposed Genetic Algorithm (GA) exploration is outperformed by A* in the Sliding Puzzle domain. The conclusion is that GA is not well suited for this domain. Our only overlapping result metric is runtime, and from the results in Table 5.3.2 we see NEN has approximately the same runtime magnitude as the A* exploration used in [6].

In conclusion, we see that LA is notably more accurate than NEN, however the runtime difference is significantly limiting allowing NEN to perform better overall. This is also the reason that the 15-piece problem instances are not solved by any LA approach within a timeout of 15 minutes. In all solved cases we see a great reduction of the explored state space, which at most are $9!/2$ and $15!/2$ states for the 8- and 15-piece instances respectively. Finally, while the plots show a relevant difference between NEN and LA they are slightly limiting since we can only show increase in problem difficulty and not problem size.

5.3.3 ECU Domain

Table 5.3.3 presents the results for each metric of the ECU planning problems. Furthermore, Figure 5.3.5 and Figure 5.3.6 plot the number of states and exploration time metric respectively for each problem size and heuristic function. Note that, in the plots we should consider all v1 and v2 problem instances as different sets since the problem sizes are not comparable.

In Table 5.3.3 we see that NEN does not perform well in this problem domain in any metric. As the problem size increases the number of explored states and consequently then path length to explored states ratio increases exponentially. This is caused by the problem type and the goal expressiveness. The ECU domain is a dynamic problem and one can calculate that for larger problem instances the number of states grows exponentially and the problem quickly becomes unsolvable. Furthermore, the inexpressiveness of the goal greatly reduces the accuracy, meaning an increase in unnecessarily explored

states. This can clearly be seen in the ratio metric.

Table 5.3.3: ECU domain planning results

	NEN				LA-D				LA-M			
	#States	Time (s)	|Path|	ratio	#States	Time (s)	|Path|	ratio	#States	Time (s)	|Path|	ratio
ECU-2-v1	22	0.02	4	5.50	27	0.27	4	6.75	18	0.16	5	3.60
ECU-3-v1	292	0.07	4	73.00	273	9.15	4	68.25	60	0.30	8	7.50
ECU-4-v1	49004	12.23	8	6125.50	-	-	-	-	384	4.80	23	16.70
ECU-5-v1	-	-	-	-	-	-	-	-	1170	30.68	42	27.86
ECU-2-v2	126	0.05	7	18.00	129	0.79	7	18.43	47	0.27	8	5.88
ECU-3-v2	9533	2.09	7	1361.86	-	-	-	-	188	1.39	16	11.75
ECU-4-v2	-	-	-	-	-	-	-	-	1755	35.20	35	50.14
ECU-5-v2	-	-	-	-	-	-	-	-	4753	134.10	50	95.06

Until problem instance ECU-3-v1 we see that LA-D is able to solve problems of the ECU domain. These are still relatively small problem instances. We also see that in terms result metrics the solutions are comparable to NEN, with the obvious exception of runtime. The reason LA-D is unable to solve larger problem instances is due to the goal inexpressiveness and the large number of witnesses for each linearization abstraction. As discussed in Section 3.3.3.2 the DC metric is very computation intensive, and especially so if the goal is inexpressive. Due to this effect we see that LA-D is not an effective exploration technique for the ECU problem domain.

Finally, we consider the LA-M exploration results. The result metrics show exceptional time performance compared to the other explorations. LA-M performs better than LA-D partially since it does not suffer from the witness set problem. However, this is not the only difference since we see that even for the smaller problems which LA-D is still able to solve LA-M is more accurate (i.e. fewer explored states and lower ratio). This effect may be due to the fact that for this problem domain the DC metric is too refined and causes the exploration of erroneous paths. By using a coarser metric we reduce this effect.

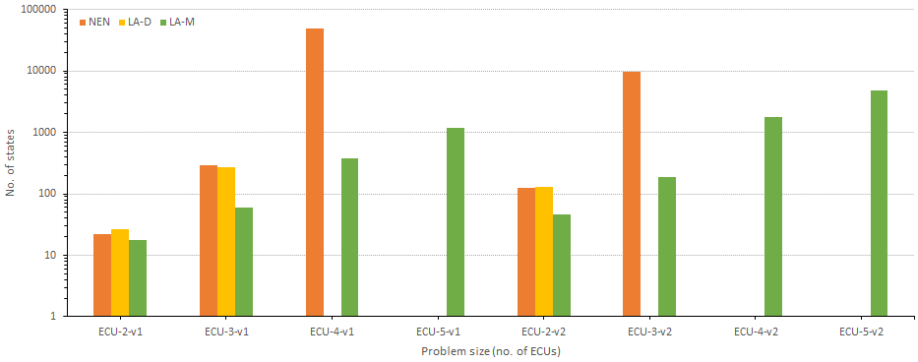


Figure 5.3.5: Average number of explored states for planning in ECU domain

In Figure 5.3.5 and Figure 5.3.6 we see that for NEN the number of explored states and time runtime increase at least exponentially on a logarithmic scale. The same

seems true for LA-D, however there are not enough data values to make any convincing conclusions. In both cases we see that these exploration approaches cannot keep up as the problem size increases.

In contrast we can see that for both sets of problem instance (v1 and v2) LA-M grows approximately at a linear rate on the logarithmic scale. This means that as the problem size increase the number of explored states and runtime increase exponentially. This is a considerable growth rate. However, compared to the other approaches LA-M performs very well. We should also consider that ECU-5-v1 and v2 may be consider relatively large systems. As an indication we consider that NEN (which already prunes the actual state space) explores almost 50,000 states for a small problem instance.

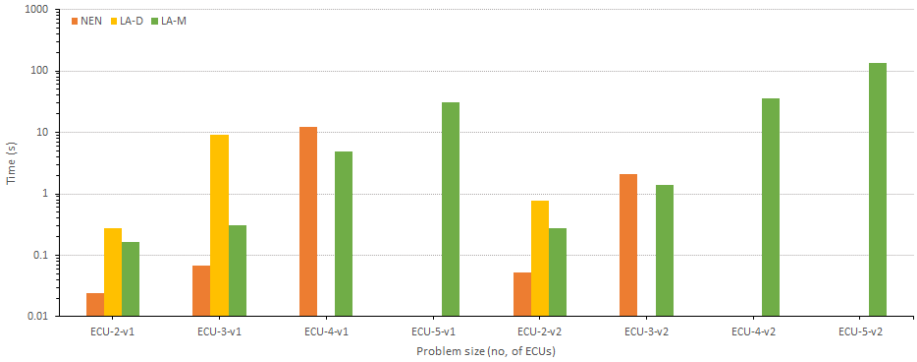


Figure 5.3.6: Average exploration time for planning in ECU domain

In Appendix A.3 we give the results of graph transformation planning in the ECU domain from [5]. As previously mentioned, the ABS heuristic is an approach introduced in that paper and is similar to linearization abstraction. In this work and in [5] the same problem sizes are used. However, this work used 2 unique problem instances per size and [5] uses 4 which are averaged. We only compare our exploration to exploration using the ABS heuristic.

In terms of number of explored states we see that LA-M is very similar to ABS, however for ABS it is not clear if there is a linear or exponential increase in the number of states as the problem size increases. It seems that of a problem size of 4 and 5 ECUs the number of states are approximately the same for EHC-ABS. This may be due to the results showing an average of four problem instances.

In terms of runtime we see that both our LA-M and exploration using ABS grows at an exponential rate in the ECU domain. However, we can see that for every instance the LA-M results are slightly faster than those of ABS using both exploration techniques. Unfortunately it is not possible to know if this is due to hardware or implementation of the approach without redoing the experiments in [5].

We must conclude that NEN and LA-D cannot be compared favorably to the results of [5] for the ECU domain since these approaches did not perform very well. Furthermore, since the linearization abstraction heuristic approach is conceptually similar to the ABS heuristic it is not surprising that they perform much the same.

5.3.4 PDDL COMPARISON

In this section we compare solving planning problems using graph transition system planning as presented in this work to using a planning solving based on a standardized planning language PDDL. In Section 2.3.1 differences between graph transformation planning problems and PDDL are explained. The results in this section give an indication how well our implementation in GROOVE compares a run-of-the-mill PDDL planner. Due to the many differences between the two approaches to planning, we are only interested in a general indication.

To achieve this we consider two problem instances. These are from the Block World and Sliding Puzzle domain respectively. We consider the Blocks-10-v1 and the 15-puzzle-v4 problem instances, the sizes of these problems represent a medium to large problem state space. In Appendix B we give the PDDL models for these problems. One should note that Blocks-10-v1 is modeled slightly differently in PDDL than as a production system. This is due to the fact that in a production system we have sets of colored blocks which are indistinguishable, and in PDDL each object has to be unique. Therefore, in the PDDL model we limit ourselves to defining a single start and goal instance (i.e. defining the exact position of a block).

Experiments are run using the LAMA-2011 PDDL planner [17], which was the winner of IPC-7 competition in sequential satisficing track; if it had competed in IPC-8 would have placed 12th out of 21 [18]. The results of the experiments, along with those using our implementation in GROOVE, are shown in Table 5.3.4.

Table 5.3.4: PDDL v GTS planning comparison

		#States	Time (s)	|Path|	ratio
Blocks-10-v1	PDDL	48	0.01	47	1.02
	NEN	141	0.06	26	5.42
	LA-D	488	310.32	32	15.25
15-Puzzle-v4	PDDL	14776	4.34	370	39.94
	NEN	10571	10.36	128	82.58
	LA	-	-	-	-

PDDL planners for the IPC competitions are designed to find the shortest path solution for a problem. This is different to the objective of greedy best-first search which we have implemented in GROOVE. In many cases finding the shortest path is more challenging and time-consuming than finding any path. LAMA-2011 uses three steps in solving a problem, these are; translation, preprocessing and search. While the details are not important for this work, search is done iteratively with continually adjusted techniques and parameters, continually seeking a shorter path. In order to compare PDDL planning to GTS planning as implemented in GROOVE we consider only the result from the first search iteration.

From Table 5.3.4 one can see that only NEN realistically compares to PDDL in terms of explored states and runtime. It seems that while PDDL performs better that the metrics are of the same order of magnitude. Clearly LA cannot compete with planners

using PDDL. This result is not unexpected as we hypothesized that the linearization abstraction heuristic approach would be better suited for dynamic planning problems, and furthermore is very computationally intensive.

It can be considered impressive that NEN manages to compete relatively well with PDDL considering that the NENTUPLE approach is rather naive (i.e. no real contextual awareness). This naivety can clearly be seen by the fact that PDDL is much more efficient. We see that for both problems the number of explored states to path length ratio is much smaller using PDDL. Again this is not surprising as PDDL is a language which is standardized for defining planning problems and developing planning solvers using PDDL has been a research and competition field since at least 1998.

The results in Table 5.3.4 show that planners using PDDL are better suited for classic planning. However, GTS planning allows for different problem types which cannot be expressed using PDDL. It has been shown in this work as well as others that these planning problems are solvable within reasonable time and space boundaries. Thus, GTS planning competes with PDDL on this front. Furthermore, improvements in heuristics and exploration techniques will improve GTS planning competitiveness with respect to PDDL planners.

5.4 DISCUSSION

The results from Section 5.3 clearly show that exploration using the NENTUPLE heuristic approach is faster than the linearization abstraction approach. However, linearization abstraction, specifically using the dependency count metric is a more accurate heuristic compared to the others presented in this work. From the results we can conclude that the hypotheses made in Section 5.1.4 were correct.

We have see that NEN performs better for classical planning problems while LA is better suited for dynamic problems. Furthermore, LA suffers heavily from independent actions, especially in large problem instances, this can be seen in the Block World domain. Similarly, NEN suffers from goal inexpressiveness, which is shown by the results from the ECU domain. Finally, from the results we can conclude that there is a clear time-accuracy trade-off between the NENTUPLE and linearization abstraction heuristic approaches.

Within the linearization abstraction approach we see that the dependency count metric is generally better than the match count metric. The exception is when calculating the witness set for DC is too time expensive. In this case the MC metric still offers a relatively accurate distance estimate and thus a decent alternative. Furthermore, from the results of the Sliding Puzzle domain we see that there is merit in combining heuristic schemes to define hybrid heuristic functions.

Finally, we have shown that PDDL planning is better than GTS planning as we have implemented it in GROOVE. Exploration using the NENTUPLE heuristics however comes relatively close in terms of explored state space and exploration runtime. The major difference however is that it is possible to define different types of problems in GTS planning. This adds possibilities to the planning domain which PDDL cannot address.

CHAPTER 6

RELATED WORK

In this chapter we give an overview of related work.

Ziegert [5] presents a heuristic based on rule abstraction. The heuristic is used greedy best-first search and enforced hill climbing search traversal strategies. The abstraction is done by relaxing the left-hand side of the transformation rule so that instead of deleting and creating nodes and edges they are only labeled deleted and created. Next all possible rules are applied to the abstracted state graph in parallel to create the abstract next state graph, this creates a linear state space. This process repeats till an abstract state in which the goal state holds is reached. The heuristic value is calculated by the number of labels that are attached to all elements in the goal match. The results show that while traversal strategy has little effect in reducing of number of states explored, the heuristic presented performs better than simpler variation of a abstraction heuristic.

Snippe [15] presents an A* implementation in GROOVE as a technique to solve graph transformation planning problems. Two heuristic functions are presented, the first is an unweighted element counting heuristic and the second is a weighted element counting with predefined weights. The weights are choose by the author and are problem specific. Results show that A* as traversal strategy with either heuristic preforms much better BFS and DFS. Furthermore it is shown that a weighted element counting heuristic performs better than an unweighted heuristic. The drawback however is that these weights problem specific and determined by author.

Estler *et al.* [7] present a framework designing element counting heuristics to guide A* and best-first traversal strategies to solve graph transformation planning problems. Several heuristics based on weighted element counting are given. Furthermore, a technique for trained element counting heuristics based on machine learning is presented. Results show that the weights are problem dependent and it is not always trivial to know the effect certain weights have on traversal efficiency. Furthermore, it is shown that trained weights can be effective.

Edelkamp *et al.* [19] present several types heuristics for informed search in graph transition systems, as well as introduction the concept of state space abstraction in order to create a database of distances from abstract states to goal states as distance measure. The abstract distance measure serves as transition cost for A* search. The main heuristic types presented are element counting, formula-based and hamming distance. Preliminary results show state space traversal with both best-first search and A* search. Each heuristic for both traversal strategies generally shows significant reduction of the number of states explored compared to DFS.

Edelkamp *et al.* [10] introduce graph transformation as a domain for modeling planning problems. GROOVE is presented as tool in which graph transformation systems and planning problems can be modeled. It is shown that using a heuristic an exponential gain in time is achieved for state space search search.

Yousefian *et al.* [6] present a model checking technique of graph transformation systems using the genetic algorithm (GA) to explore the state space. The genetic algorithm is a heuristic technique which guides state space exploration. The approach is used to check safety properties and search deadlocks of a system. GA is used to determine the next state to discover, for this state safety and deadlock properties are then checked, if an erroneous or deadlock state is discovered the path to this state is returned otherwise the algorithm determines the next state. It is shown that for several different problem types this approach provides a speed up time in finding erroneous/deadlock states. However the main advantage is the increase problem size this approach can handle compared to exhaustive state space exploration. The disadvantage is that if the algorithm does not return a result before the cutoff depth it is not guaranteed that a erroneous of deadlock state does not exist.

Below in Table 6 we give an overview of the planning problems used in each work. We differentiate between classic and dynamic problem types. These problems served as a basis for the planning problems we included in our experiments. This allowed us to compare the results achieved by our approach to known approaches.

Table 6.0.1: Overview of Planning Problems in Related Work

Classic Problems	Dynamic Problems
Block World Problem [5, 10, 15]	Electronic Control Units (ECU) [5]
Pac-Man Problem [6]	Car Platoon System (CPS) [6]
N-Puzzle Problem [6, 7]	Neue Bahntechnik Paderborn (NBP) [7]
Dining Philosophers Problem [6, 10]	Arrow Distributed Directed Protocol (ADDP) [19]
Girl's Gossip Problem [10]	

Close to the finalization of this work we were made aware of a master student in Brazil who is currently working on their thesis which is extremely closely related to the topics in this work. The work by Ramos *et al.* [20] is currently still in progress and thus unpublished. In this work several domain independent heuristics based on

graph abstraction are developed. From preliminary results we have seen that for all experiments these heuristics provide a decrease in the number of explored states in comparison to breadth first search for planning problems. We have done some experimental tests on the same problem set used by Ramos *et al.* using the NENTUPLE heuristic approach. We found that for several problems there are similarities in terms of explored state space. However, in some cases our approach was much less effective. We believe this is partially due to the modeling of the problems. For example, with the sliding puzzle model used by [20] our exploration using NENTUPLES generated a more states. However, using our modeling for the same problem instance our approach greatly outperformed the heuristics presented in [20]. Due to the recent knowledge of this work we have unable to perform a more comprehensive comparision. Furthermore, since the research is being done in Brazil the work will, unfortunately for us, be published in Portuguese.

CHAPTER 7

CONCLUSION

In this final chapter we look back on this work and the answers to the research questions given in Section 1.3 and discuss to what degree we have managed to solve our problem statement. Furthermore, we provide a general overview of this work and discuss what this works contributes. We conclude with a presentation of ideas for future work.

7.1 DISCUSSION

The goal of this section is to discuss the results of the research questions and problem statement of this work.

RQ1. How can we define and solve planning problems in the context of graph transition systems and what advantages and disadvantages does this provide?

This question is answered in Chapter 2. In Section 2.1 we formally defined the concepts concerning graph transition systems, as well as additional notions related to planning. In Definition 13 we defined the graph transformation planning problem and the solution of such a problem. In Section 2.2 we presented a planning strategy, forward state space search (implemented using greedy best first search), which can be used to solve graph transformation planning problems. Finally, in Section 2.3 we concretely put all previous graph transformation concepts in the context of planning. We discussed **a)** how graphs and graph transformations are suited as a planning language, **b)** which planning strategies are suited graph transformation planning, **c)** in which way the goal of a problem can be specified and, **d)** the form of the solution. In addition to these topics we also considered the advantages and disadvantages that graph transformation planning provide in comparison to other planning languages.

We have found that graph transition systems are a suitable paradigm for modeling planning problems. Graphs and graph transformations correspond to states and actions in a planning problem. Furthermore, we used known state space exploration techniques which are applicable to transition systems to solve graph transformation planning problems.

We argued that there are several advantages to graph transition planning problems,

such as an intuitive problem modeling approach and the possibility to model dynamic planning problems (see Section 5.1.4). Furthermore, we hypothesized that the underlying graph structure of the states and action would allow enable one to define accurate domain independent heuristics. We defined two such heuristic approaches in Chapter 3 and from the results in Section 5.3 we saw that these perform well for respective problem types. In Section 5.3.4 we gave a comparison of the GTS planning approach to the PDDL approach. We saw that while PDDLs performance is superior, GTS planning is not totally outclassed.

RQ2. In which ways can we exploit the underlying structure and formalisms of graphs and graph transformations in a graph transition system to develop meaningful metrics to estimate distances between a state and a goal?

We answered this question in Chapter 3 of this work. We provided two heuristic approaches which attempt to make use of the graph transition system model paradigm to estimate the distance between a graph and a goal. These two approaches rely on two different forms of abstraction. The NENTUPLE approach uses an abstraction of graphs and compares the decomposition of the graphs corresponding to a state and a goal respectively. The linearization abstraction approach uses abstraction of the graph transition system and its underlying components (graphs and graph transformation rules) to create a compressed (due to linearization) overestimation of a problem state space. We have defined distance metrics using information from the process linearization abstraction. By using these abstraction methods in NENTUPLES and linearization abstraction we were able to develop heuristic approaches which are domain independent.

We argued that both these approaches provide meaningful distance estimates. The NENTUPLE does this by abstractly comparing two graphs. The linearization abstraction approach has three distance metrics all which are based on the process of linearization abstract, counting either the number of iterations, the number of total actions or the number of goal specific actions during the process. Since linearization abstraction provides an over approximation of the problem state space there is a relation between the abstracted and the actual state space, and thus our distance metrics have a relation to the actual distance between a state and a goal.

RQ3. How can we implement planning in GROOVE with sophisticated problem solving techniques supported by a framework of heuristics?

We gave the answer to this question in Chapter 4. In order to implement planning in GROOVE we have extended its state space exploration capabilities to support informed exploration (GBFS), acceptors which fulfill graph goal types and exploration results which store a path.

We also implemented a framework in GROOVE for heuristic functions which can be used for informed exploration. The aim of this framework was to setup a software infrastructure that is extensible and flexible. To do this we used design patterns such as the composite pattern to allow for easy combination of heuristic functions to form hybrid functions and the addition of new heuristic approaches.

The above answered research questions aim to support the problem statement of this work which is repeated below.

To develop heuristic approaches for the purpose of planning using graph transformations and implement these in GROOVE as a framework for solving planning problems.

We have achieved our problem statement as we have developed two distinct heuristic approaches for graph transformation planning problems. We have furthermore, extended to GROOVE functionality by implementing these approaches and developing a framework in which they can be used to solve planning problems. We have then experimented with these approaches to solve different planning problems using greedy best-first search as exploration strategy. Finally, we have identified distinguishing features of different planning problems and correlated these to the suitability of the heuristic approaches we have developed.

7.2 CONTRIBUTIONS

In this section we describe the contributions this work has made to the field of planning and heuristics.

The main contribution of this work is the formal definition of two unique domain independent graph heuristics. We have defined two distinct graph heuristics and demonstrated their effectiveness on are range of different planning problems. We provide a detailed theoretical explanation of both approaches. This provides interested parties to easily extend upon our work.

In addition to we have demonstrated to benefits of using graph transition systems as a planning problem modeling paradigm in contrast to others such as PDDL. We provide further motivation to broaden the planner horizon to consider the possibilities of graph transition system modeling.

Our heuristic approaches are partially based upon those existing in other works. Developing more effective domain independent heuristics is a forever ongoing process with a lot of room for fine-tuning. Similarly, the approaches in this work may be extended to further the field of heuristics in graph transformation planning.

Moreover, in this work we have implemented a planning and heuristic proof-of-concept in the practical tool GROOVE. Thus, in this work we have contributed to the extension of the functionality of GROOVE. We have provided the possibility for generic users to solve their planning problems in GROOVE using the command line interface. Furthermore, we have implemented this functionality in such a manner that a more experienced user can implement their own heuristics in the GROOVE code within our framework. This allows for the general extension of the program functionality as well as the creation of tailor made heuristics for planning problems.

Finally, have begun on a rudimentary classification of graph transformation planning problems based on distinguishing features. We attempted to correlate such classifications with the effectiveness of our heuristics approaches. While there is still much room for improvement, extension and formalization of this classification, such a clas-

sification and correlation to heuristics provides a method to identify the effectiveness of heuristics for a certain problem. Such a contribution may be very valuable as the use of an unsuited heuristic to solve a planning problem may be more effective than using no heuristic at all.

7.3 FUTURE WORK

In this section we give an overview of several six possible directions for continuation of this work. The first three future work opportunities relate to the extension of theoretical components of this work. The other three opportunities detail implementation development of the heuristic framework in GROOVE.

Extensions of satisfiability conditions of goal types for heuristics and of acceptors in GROOVE to support goal types.

In Section 2.3.3 we introduce approaches to formulating a goal ϕ_{goal} and call these goal types. In Section 3.2.1 and Section 3.3.1 we define heuristic approaches which given a certain goal type can be used to generate heuristic schemes. The reason that a heuristic approach is tied to a goal type is due to the representation of specific goal types and how they are satisfied.

Currently, for NENTUPLES we only give heuristic schemes for partial and complete graph goal types, and for linearization abstraction we omit the predicate over graph goal type from our heuristic schemes.

An extension to this work would be defining all necessary components such that it is possible to apply our heuristic approaches for all possible goal types. Furthermore, it may be possible to define even more goal types. Finally, in Section 4.1 we define how the satisfiability for goal types in the form of graphs is defined. This can be extended for all goals and furthermore implemented in GROOVE.

Completeness and formal definitions of graph transformation planning problem distinguishing features.

In Section 5.1.4 we present and discuss what we call distinguishing features of graph transformation planning problems. Furthermore, we hypothesize on the effect these features have on our heuristics from Chapter 3. These features are presented in an informal manner and can generally only be measured subjectively. An extension to this work could be to give more rigorous or formal definitions of our proposed problem features. In this way it may become possible to more accurately correlate the effects features have on our heuristics. Formalization will also allow one to better verify our hypothesis on the effects of problem features.

A follow up to this extension is to consider additional distinguishing features. In this work we present three features and also discuss the effects of the combinations, since they are not mutually exclusive. However, it may be possible to imagine that features were overlooked or not thoroughly defined for all problem instances. Thus, a further extension of this work would be to consider the completeness of the distinguishing features presented in Section 5.1.4.

Extensions of heuristics and informed exploration strategies to offer a broader spectrum of planning problem solving capabilities.

A straightforward extension of this work would to be extend the number of heuristic functions given in this work. This can be done by defining additional heuristic schemes based on our presented approaches or by creating a new heuristic approach. In the case of extending NENTUPLES it might be an option to define another possibly relevant graph decomposition or in the case of linearization abstraction to slightly alter the level of information in the abstraction. An example of this could be to specifically record if an element in the maybe set was created or deleted.

In the case of new heuristic approaches it may be an option to consider user-defined graph patterns for graph decompositions or the use of machine learning to determine relevant structures.

Finally, in terms of exploration an extension to this work could be to consider more informed search strategies such as A*.

Implementation of operations to combine heuristics in GROOVE framework.

In Section 4.2 we present and implement an architectural framework for the use of heuristics in informed search within GROOVE. In this framework we use the composite pattern which allows for the combination of heuristic functions, we call these combinations hybrid functions. Currently, we have only implemented leaf functions. A possible extension to this work in terms of implementation would be to define and implement operators which meaningfully combine heuristic functions into hybrid heuristic functions. Furthermore, it is possible to extend the `HeuristicValue` return value of a heuristic function in order to define different ways in which heuristic values are evaluated.

Implementation of abstraction in GROOVE.

In Section 4.3 we discussed how we simulate linearization abstraction defined in Section 3.3 in GROOVE by altering the graph transformation rules used for the linearization abstraction heuristic. An extension of this work could to fundamentally implement abstract graphs and abstract transformations in GROOVE.

Optimization of implemented heuristics and extended testing of dynamic planning problems.

For this work we have provided a proof-of-concept implementation for certain NENTUPLE and linearization abstraction heuristic schemes. A purpose of this was to experiment with the effectiveness of our proposed approaches to solve planning problems. While speed and memory usage were considered during implementation, they were not the primary focus. A future work may be to reevaluate the implementation of the heuristic approaches in GROOVE and make code optimizations. Specifically, for the dependency count metric of linearization abstraction optimization improvements could go a long way in terms of speedup.

Furthermore, in our experimentation test suite we have currently only run tests on a single dynamic planning problem. A future work would be to extend our test suite with more dynamic problems to further evaluate and confirm our hypothesis on the effectiveness of linearization abstraction heuristics on this problem type.

Appendices

APPENDIX A

COMPARISON RESULTS

This appendix presents results of other works mentioned in this thesis for easier comparison.

A.1 BLOCK WORLD

In this section we show results from other works using heuristics to solve graph transformation planning problems in the Block World domain.

In Figure A.1.1 and A.1.2 we show graphs, as given in [5], of the results of Block World planning problems for average explored state space and planning time respectively. In the experiments two different exploration strategies are used and combined with two different exploration heurstics. These exploration strategies are greedy best-first search, denoted by GBF and enforced hill climbing, denoted by EHC. The two heuristics are SIM which is simple element counting as presented in [15] and ABS which is a heuristic similar to linearization abstraction presented in this thesis.

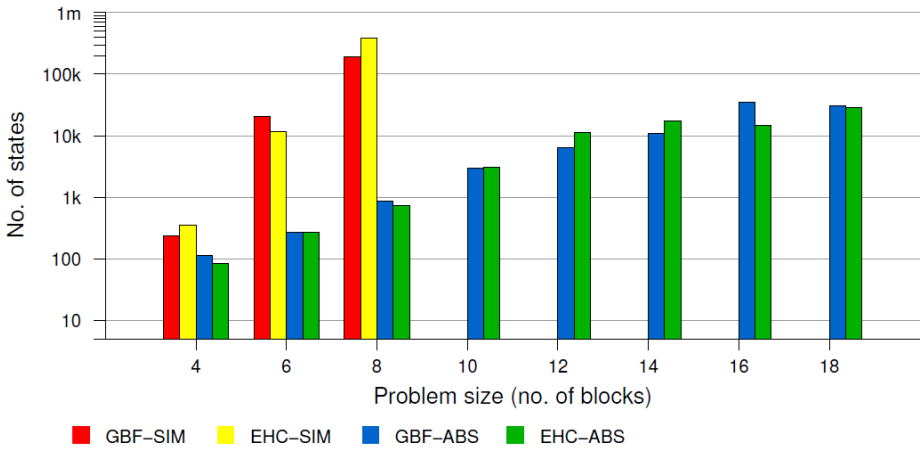


Figure A.1.1: Results from [5]: average number of explored states for planning in Blocks World domain

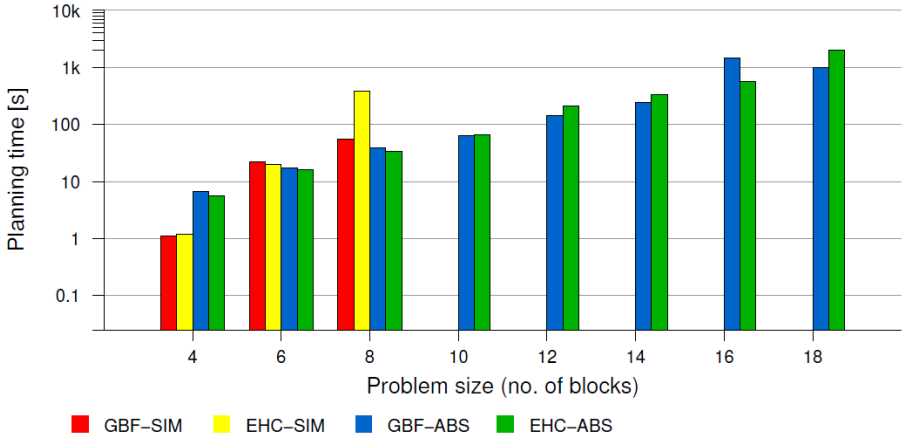


Figure A.1.2: Results from [5]: average exploration time for planning in Blocks World domain

A.2 Sliding Puzzle

In this section we show results from other works using heuristics to solve graph transformation planning problems in the Sliding Puzzle domain.

In Figure A.2.1 we show the results from [7]. The technique MC denotes a solving technique using model checking after complete generation of the state space. BF and A* denote the exploration strategy greedy best-first search and A* respectively. These exploration strategies are used with two heuristic functions. h^1_{Puz} is a heuristic which counts all misplaced pieces. h^2_{Puz} is a heuristic which calculates the sum of the

Manhattan distances of every misplaced piece.

The results show the number of states and transitions generated for each problem and the time needed to find a solution. No information is given about the path. Finally, problem instance 8puzzle-06 is purposefully modeled to be easily solvable.

	MC			*BF with* h^1_{Puz}			*BF with* h^2_{Puz}			*A* with* h^1_{Puz}			*A* with* h^2_{Puz}		
	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)
8puzzle-01	*	*	*	1453	2432	2	1688	2778	3	10207	17727	89	**662**	1031	<1
8puzzle-02	*	*	*	1389	2333	2	**251**	404	<1	82159	164840	6462	903	1505	25
8puzzle-03	*	*	*	**627**	1040	<1	955	1550	1	5926	10140	32	1032	1723	1
8puzzle-04	*	*	*	1086	1810	1	**117**	187	<1	27457	50070	649	2153	3631	4
8puzzle-05	*	*	*	1409	2344	1	**250**	401	<1	5311	9053	23	583	965	<1
8puzzle-06	*	*	*	**6**	8	<1	**6**	8	<1	**6**	8	<1	**6**	8	<1
15puzzle-01	~	~	~	~	~	~	**25967**	41225	336	~	~	~	%	%	%
15puzzle-02	~	~	~	~	~	~	**4997**	7717	22	~	~	~	%	%	%
15puzzle-03	~	~	~	~	~	~	**3561**	5468	13	~	~	~	%	%	%

* out of memory exception
~ not evaluated
% premature termination after 4 hours

Figure A.2.1: Results from [7] for planning in Sliding Puzzle domain

In Figure A.2.2 we show the results from [6] which proposes the Genetic Algorithm as exploration strategy, which is denoted with GA. Furthermore, results are presented for exploration using A*, DFS and BFS. In the case of A* the heuristic used in not specified. The results show the runtime to find a solution. This work focuses of finding error states and for the encoding of this problem a goal state is referred to as an error.

The results of executing the proposed GA for 8-puzzle.

8-Puzzle	GA						A*		DFS	BFS
	Depth searched	Result time (s)	Min time (s)	Max time (s)	# Runs/# errors	First error depth	Result time (s)	First error depth	Result time (s)	Result time (s)
	50	5.8	2	22.4	10/10	3	0.046	3	Out of memory	0.5
	50	15	2.4	27.5	10/10	4	0.037	4		0.55
	50	380	72.7	794.6	10/10	8	0.27	8		0.6
	50	612	82.6	1130.5	10/10	11	0.59	11		0.7

Figure A.2.2: Results from [6] for planning in Sliding Puzzle domain

A.3 ECU

In this section we show results from other works using heuristics to solve graph transformation planning problems in the ECU domain. These are results from [5]. An explanation of which exploration strategy and heuristics are used can be found in Appendix A.1.

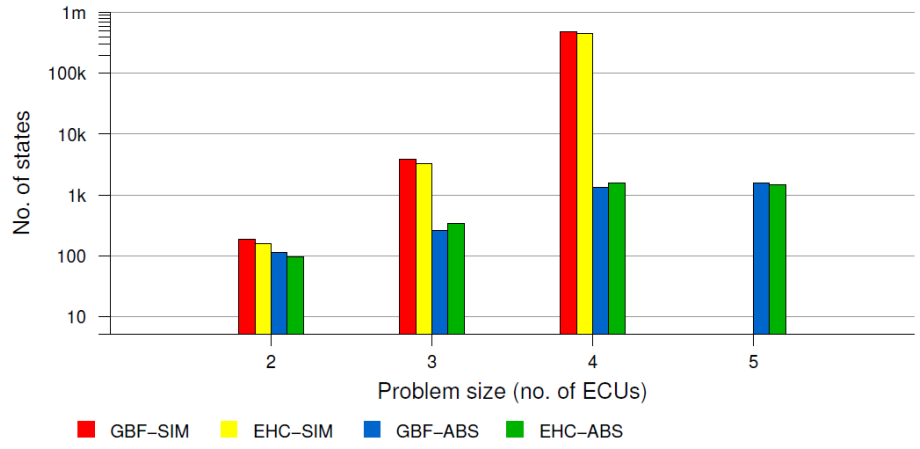


Figure A.3.1: Results from [5]: average number of explored states for planning in ECU domain

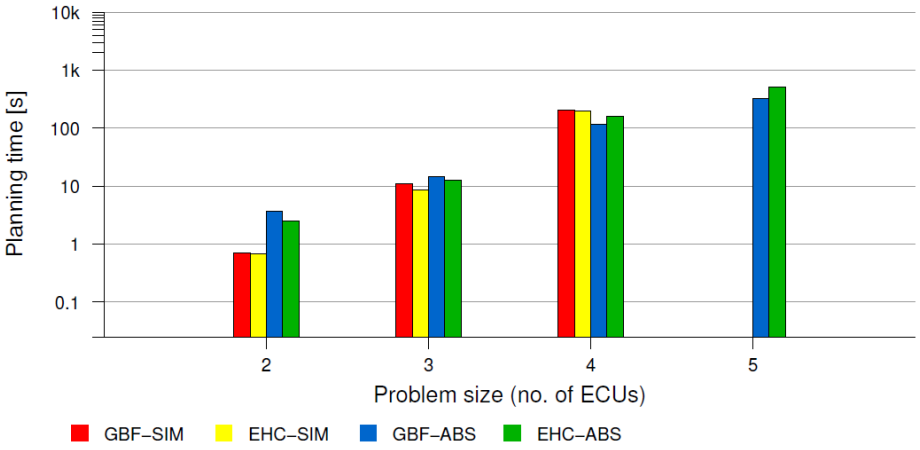


Figure A.3.2: Results from [5]: average exploration time for planning in ECU domain

APPENDIX B

PDDL

In this appendix we give the PDDL code used to compare results obtained for graph transformation planning problems to their PDDL equivalent.

B.1 BLOCK WORLD

Listing B.1: Block World Domain Defined in PDDL

```
(define (domain blocksworld)
  (:requirements :strips)
(:predicates (clear ?x)
             (on-table ?x)
             (arm-empty)
             (holding ?x)
             (on ?x ?y))

(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
               (not (arm-empty))))

(:action putdown
  :parameters  (?ob)
  :precondition (holding ?ob)
  :effect (and (clear ?ob) (arm-empty) (on-table ?ob)
               (not (holding ?ob))))

(:action stack
  :parameters  (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob)
               (not (clear ?underob)) (not (holding ?ob))))

(:action unstack
  :parameters  (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
  :effect (and (holding ?ob) (clear ?underob)
               (not (on ?ob ?underob)) (not (clear ?ob)) (not (arm-empty)))))
```

Listing B.2: 10-block Block World Problem in PDDL

```
(define (problem blocks-10-v1)
  (:domain blocksworld)

  (:objects
    R1 R2 R3 B1 B2 B3 G1 G2 G3 G4
  )

  (:init
    (on-table B3) (on R2 B3) (clear R2)
    (clear G1) (on G1 B1) (on B1 G2) (on G2 G3) (on G3 R1)
      (on R1 B2) (on B2 G4) (on G4 R3) (on-table R3)
    (arm-empty)
  )

  (:goal  (and
        (on R3 R2) (on R2 R1) (on-table R1)
        (on B3 B2) (on B2 B1) (on-table B1)
        (on G4 G3) (on G3 G2) (on G2 G1) (on-table G1)
        (arm-empty)
      )
  )
)
```

B.2 SLIDING PUZZLE

Listing B.3: Sliding Puzzle Domain Defined in PDDL

```
(define (domain SlidingPuzzle)
  (:requirements :strips :typing)

  (:types location
    piece
  )

  (:predicates
    (on  ?p - piece ?l - location)
    (bot ?x - location ?y - location)
    (right ?x - location ?y - location)
    (blank ?x - location)
  )

  (:action moveDown
    :parameters (?p - piece ?from  - location ?to - location)
    :precondition (and  (on ?p ?from)
          (bottom ?from ?to)
          (blank ?to)
        )
    :effect   (and  (not (on ?p ?from))
          (on ?p ?to)
          (not (blank ?to))
          (blank ?from)
        )
  )
  (:action moveUp
    :parameters (?p - piece ?from - location ?to - location)
    :precondition (and  (on ?p ?from)
          (bottom ?to ?from)
          (blank ?to)
        )
    :effect   (and  (not (on ?p ?from))
          (on ?p ?to)
          (not (blank ?to))
          (blank ?from)
```

```
        )
  )
  (:action moveLeft
    :parameters (?p - piece ?from - location ?to - location)
    :precondition (and  (on ?p ?from)
          (right ?to ?from)
          (blank ?to)
        )
    :effect   (and  (not (on ?p ?from))
          (on ?p ?to)
          (not (blank ?to))
          (blank ?from)
        )
  )
  (:action moveRight
    :parameters (?p - piece ?from - location ?to - location)
    :precondition (and  (on ?p ?from)
          (right ?from ?to)
          (blank ?to)
        )
    :effect   (and  (not (on ?p ?from))
          (on ?p ?to)
          (not (blank ?to))
          (blank ?from)
        )
  )
)
```

Listing B.4: 15-piece Slding Puzzle Problem in PDDL

```
(define (problem 15-puzzle)
  (:domain SlidingPuzzle)

  (:objects
    pos11 pos12 pos13 pos14 pos21 pos22 pos23 pos24
      pos31 pos32 pos33 pos34 pos41 pos42 pos43 pos44 - location
    p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 - piece
  )

  (:init
    (on p5  pos11) (on p14 pos12) (on p4  pos13) (on p8 pos14) (on p2 pos21)
    (on p3  pos22) (on p10 pos23) (on p13 pos24) (on p1 pos31) (on p6 pos32)
    (on p11 pos33) (on p15 pos34) (on p12 pos41) (on p9 pos42) (on p7 pos43)
    (blank pos44)
    (bot pos11 pos21) (bot pos12 pos22) (bot pos13 pos23) (bot pos14 pos24)
    (bot pos21 pos31) (bot pos22 pos32) (bot pos23 pos33) (bot pos24 pos34)
    (bot pos31 pos41) (bot pos32 pos42) (bot pos33 pos43) (bot pos34 pos44)
    (right pos11 pos12) (right pos12 pos13) (right pos13 pos14)
    (right pos21 pos22) (right pos22 pos23) (right pos23 pos24)
    (right pos31 pos32) (right pos33 pos33) (right pos33 pos34)
    (right pos41 pos42) (right pos43 pos43) (right pos43 pos44)
  )

  (:goal  (and
        (on p1  pos11) (on p2  pos12) (on p3  pos13) (on p4  pos14)
        (on p5  pos21) (on p6  pos22) (on p7  pos23) (on p8  pos24)
        (on p9  pos31) (on p10 pos32) (on p11 pos33) (on p12 pos34)
        (on p13 pos41) (on p14 pos42)                (on p15 pos43)
      )
  )
)
```

BIBLIOGRAPHY

[1] ICAPS. (). Main/icaps, [Online]. Available: http://www.icaps-conference.org/.

[2] A. M. Starfield, K. Smith, and A. L. Bleloch, *How to model it: Problem solving for the computer age*. McGraw-Hill, Inc., 1993.

[3] A. Rensink, "The groove simulator: a tool for state space generation", in *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, J. L. Pfaltz, M. Nagl, and B. Böhlen, Eds., ser. Lecture Notes in Computer Science, vol. 3062, Berlin: Springer Verlag, 2004, pp. 479–485.

[4] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003, ISBN: 0137903952.

[5] S. Ziegert, "Graph transformation planning via abstraction", *arXiv preprint arXiv:1407.7933*, 2014.

[6] R. Yousefian, V. Rafe, and M. Rahmani, "A heuristic solution for model checking graph transformation systems", *Applied Soft Computing*, vol. 24, pp. 169–180, 2014.

[7] H.-C. Estler and H. Wehrheim, "Heuristic search-based planning for graph transformation systems", *KEPS 2011*, p. 54, 2011.

[8] R. E. Fikes and N. J. Nilsson, "Strips: a new approach to the application of theorem proving to problem solving", *Artificial intelligence*, vol. 2, no. 3, pp. 189–208, 1972.

[9] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. E. Smith, *et al.*, "Pddl-the planning domain definition language", 1998.

[10] S. Edelkamp and A. Rensink, "Graph transformation and ai planning", 2007.

[11] J. Hoffmann and B. Nebel, "The ff planning system: fast plan generation through heuristic search", *Journal of Artificial Intelligence Research*, pp. 253–302, 2001.

[12] R. Meijer, "Pddl planning problems and groove graph transformations: combining two worlds with a translator", in *17th Twente Student Conference on IT 17*, 2012.

[13] M. Tichy and B. Klöpper, "Planning self-adaption with graph transformations", in *Applications of Graph Transformations with Industrial Relevance*, Springer, 2012, pp. 137–152.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0-201-63361-2.

[15] E. Snippe, "Using heuristic search to solve planning problems in groove", in *14th Twente Student Conference on IT, University of Twente. Available at fmt. cs. utwente. nl/education/bachelor/73*, 2011.

[16] W. W. Johnson and W. E. Story, "Notes on the" 15" puzzle", *American Journal of Mathematics*, vol. 2, no. 4, pp. 397–404, 1879.

[17] S. Richter, M. Westphal, and M. Helmert, "Lama 2008 and 2011", in *International Planning Competition*, 2011, pp. 117–124.

[18] M. Vallati, L. Chrpa, M. Grzes, T. L. McCluskey, M. Roberts, and S. Sanner, "The 2014 international planning competition: progress and trends", *AI Magazine*, vol. 36, no. 3, pp. 90–98, 2015.

[19] S. Edelkamp, S. Jabbar, and A. L. Lafuente, "Heuristic search for the analysis of graph transition systems", in *Graph Transformations*, Springer, 2006, pp. 414–429.

[20] A. S. Ramos, M. C. S. Boeres, and E. Zambon, "On applying guided search methods for state space exploration of graph grammars", in *48th Brazilian Symposium on Operational Research*, Unpublished MSc. Thesis, 2016.