Heuristic Search-Based Planning for Graph Transformation Systems

H.-Christian Estler[1] and **Heike Wehrheim**[2]
[1]ETH Zurich, christian.estler@inf.ethz.ch
[2]University of Paderborn, wehrheim@uni-paderborn.de

Abstract

Graph notations have proven effective in modeling complex systems. In recent years, it has therefore been proposed that graphs could also be used for modeling planning problems, *i.e.* using graphs to express states and actions. Translating these graphs into PDDL would be desirable since today's Graph Transformation tools are not as efficient as modern planning tools. Unfortunately, such a translation is not always possible as the formalisms have different expressiveness. In this work, we present an extension of a Graph Transformation tool with classic planning algorithms. Using two case studies, we show that this extension makes planning with graphs more feasible – without the need of translating into PDDL. Furthermore, we demonstrate how typical modeling artifacts, like meta-models, can be leveraged to semi-automatically develop heuristics for the planner.

Introduction

Graphical notations are widely used in computer science to model complex systems. Their depiction often allows for an easier and faster understanding of the structure of a system and they can be more accessible to non-experts compared to purely text-based notations. A well know example of such a notation is the *Unified Modelling Language* (UML) (OMG 2010) which has widespread use in industry and academia.

To leverage the advantages of graphical notations in the area of planning, researchers are investigating how such notations could be used to model planning problems. Tools like ItSimple (Vaquero, Tonidandel, and Silva 2005) allow the user to model a planning problem using different types of UML diagrams and subsequently generate planning problems in PDDL (Ghallab et al. 1998) which can be solved using off-the-shelf planning tools.

In this paper, we explore another approach towards solving planning problems which are modeled using UML diagrams. Rather than transforming the diagrams – which represent states and actions – into another language, we use them *as-is* when searching for a plan. The diagrams themselves are (directed labeled) graphs and we can thus use a graph transformation tool for applying actions to states, thereby generating the search space. The advantage of this approach is that we can use the full expressiveness of the graph formalism which, in our case, is different from the expressiveness of PDDL. However, as graph transformations are in general computationally expensive, we are facing the fundamental problem of planning tools: the need to minimize the number of states in the search space (*i.e.* the number graph transformations).

We have built a planning framework that uses heuristic search algorithms (currently A^* or *Best First*) to direct the search in a state space where new states are generated using a graph transformation tool. To the best of our knowledge, this is the first tool to experiment with such a combination. While it does not come as a surprise that a heuristic-driven approach typically uses less transformations than a non-heuristic approach, the evaluation of our framework yields insight of how efficient such a tool can perform in practice. Using our planning framework and two case studies, we will demonstrate i) that planning with graph transformation tools becomes more feasible when using heuristic search strategies instead of non-heuristic approaches, ii) how users can be enabled to write domain-specific heuristics for graph based planning problems and iii) how modeling artifacts, such as a meta-model, can be used to semi-automatically learn domain-specific heuristics.

Case Studies

The first case study we present in this paper is the n-puzzle problem. n numbered tiles are positioned on a square board. The objective is to place the tiles in order, using only *slide* moves, *i.e.* only a tile adjacent to the empty field can slide onto that field.

The second case study has more of a "real-world problem" character. It is based on the research project *Neue Bahntechnik Paderborn* (NBP) at the University of Paderborn. NBP aims at the development of a future railway system where small, driverless vehicles act completely autonomously with respect to individual goals. The vehicles are called *RailCabs*, referring to the idea that the transport of passengers or goods is demand

---

driven, as it is with regular cabs.

A typical planning problem for the NBP case study is the following (see Fig. 2): passengers and cargo needs to be transported from Paderborn to Berlin. Furthermore, passengers need to be transported from Paderborn to Leipzig. The RailCabs (RC1, RC2, RC3) which are needed to satisfy these request share part a part of their route. Whenever possible, the RailCabs should build a *convoy* by driving close together, therewith minimizing energy consumption. Furthermore, every RailCab is required to be in contact with a *Base Station* (BS1, BS2, …) to enable communication. Safety requirements such as “a RailCab with dangerous cargo is not allowed in a convoy” have to be met.

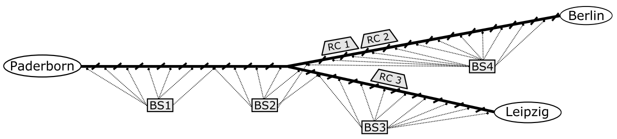

Fig. 2: Coordinating RailCabs constitutes a planning problem.

In total, the NBP planning problems consists of 15 possible actions (*e.g.* *move RailCab*, *create convoy*, *change Base Station*) and five rules which define safety requirements.

Graphs and Graph Transformations

The graphs we use to model planning problems are called *story patterns* (Fischer et al. 2000). They were developed as part of an extension of UML activity diagrams.

In its most simple form, a story pattern equals an UML object diagram. We use this simple form to model the start state of a planning problem. Fig. 1 shows a start state of a NBP planning problem. Nodes and edges of the graph are labeled, where a label consists of two strings which are separated by a colon. The front-string is the object name, whereas the rear-string defines the type of the object.

Story patterns are also used to define the actions of a planning problem. They describes how a graph can be transformed into another graph and are therefore also call it a *graph transformation rule.* An example for the “move” action of a RailCab is shown in Fig. 3.

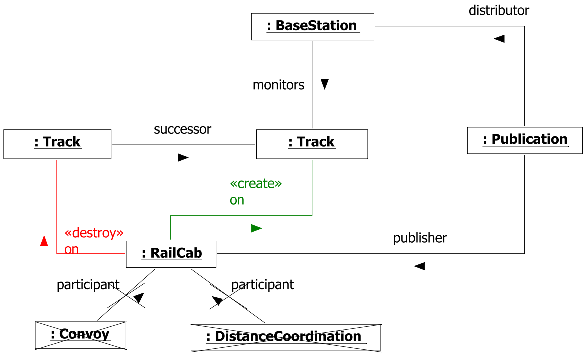

Fig. 3: Story pattern for the action *move*.

In addition to the nodes and edges of a regular object diagram, a graph transformation rule can use the special annotations `<<create>>` and `<<destroy>>` for nodes and edges. Furthermore, nodes and edges can be crossed out to define that the rule is only applicable to graphs which do not contain certain nodes or edges (called *Negative Application Conditions* (*NACs*)). In contrast to the start state graph from Fig. 1, the node labels omit a string in front of the colon. This omission defines that only the type of a node is of interest, not its specific object-name.

The execution of a transformation rule is performed in two steps: First, the graph to be changed (*e.g.* the one from Fig. 1) is searched for a subgraph which equals the graph of the transformation rule except for crossed out elements or those which are annotated with

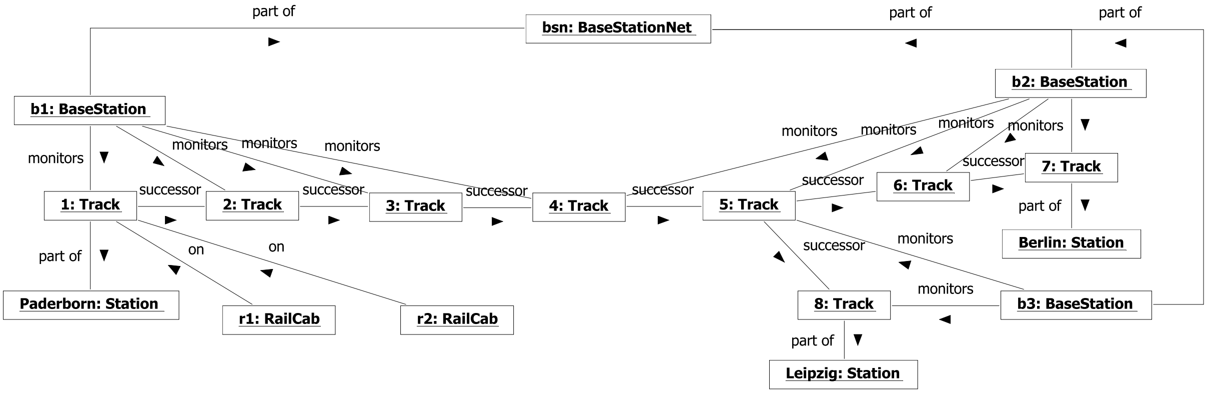

Fig. 1: Story pattern modeling a start state of a NBP planning problem.

`<<create>>`. If such a subgraph exists, then we have found a *matching*. Secondly, the subgraph will be modified by creating and deleting those elements which are annotated accordingly in the transformation rule.

Modeling goal states or states which are forbidden (*e.g.* due to safety requirements) is similar to modeling actions. The only difference is that we use story patterns without `<<create>>` or `<<delete>>` annotations. NACs, however, are allowed. The idea is to state only the properties that are of interest for a state, in order to be considered a goal state or a forbidden state. Fig. 4 shows an example of a goal state for the NBP problem.

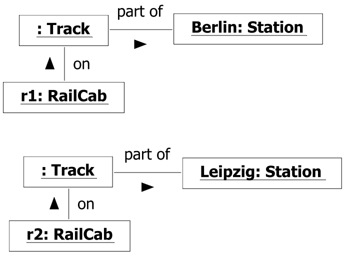

Fig. 4: Story patterns are used to define goal states.

The tuple (G, R), where G is a set of graphs and R is a set of graph transformation rules is called a *Graph Transformation System* (GTS).

It is worthwhile to mention the complexity of graph transformations: Establishing a matching between a transformation rule and a graph implies that one has to find graph homomorphisms. Deciding if a homomorphism exists is $\mathcal{NP}$-complete. Furthermore, we need to check if a newly generated state is already present in the search space, *i.e.* for every new graph, we check if there exists a graph isomorphism to a graph already in the search space. Deciding graph isomorphism is known to be in $\mathcal{NP}$.

Though story patterns alone are sufficient to model a GTS planning problem, our framework additionally requires the user to model a UML class diagram. This class diagram defines the types (classes), labeled edges (associations) and possible node connections (multiplicity constraints) in a planning problem. The class diagram needs to be provided before modeling the story patterns. It serves as a meta-model, *i.e.* only nodes and edges which are declared in the class diagram can be used in a story pattern, ensuring consistency between different story patterns. An example of a class diagram for the NBP case study is shown in Fig. 5.

The Planning Framework

Our planning framework utilizes two other tools. The first one is FUJABA[1] (*F*rom *U*ML to *J*ava *a*nd *b*ack *a*gain), an open-source tool for model-based software engineering. We use it as a front-end for the user input, *i.e.* a user models the story patterns and the class diagram within FUJABA.

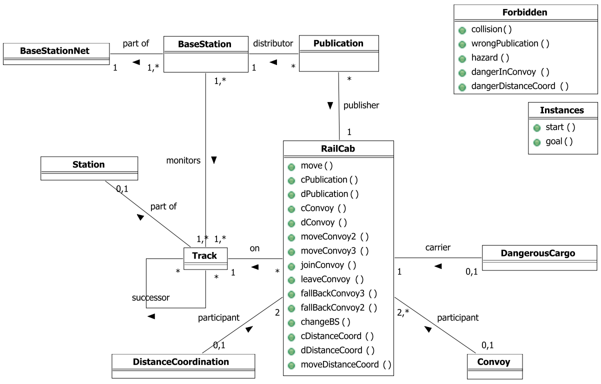

Fig. 5: UML class diagram for the NBP case study.

The second tool is GROOVE[2] (Rensink 2004). The main feature of GROOVE is its *simulator* which allows for generating and analyzing graph transition systems. While GROOVE also incorporates an editor to define graphs and transformation rules, we use it as a back-end only, *i.e.* we rely on it for graph transformations and isomorphism checks but we apply our own algorithms to control the generation of the state space.

The graph formalisms used by FUJABA and GROOVE are very similar but not identical. Therefore, we use a translation procedure presented by Röhs *et al.* in (Röhs 2009; Röhs and Wehrheim 2010). In principal, our framework can work without FUJABA in case a user prefers to model a GTS directly in GROOVE. Furthermore, we designed the planning framework to be as independent of a specific graph transformation tool as possible. For example, none of the original GROOVE source code has been modified. While it is not possible to simply exchange GROOVE for another graph transformation tool (some parts of the framework's implementation are GROOVE specific), large parts of the framework, could be reused as-is in case GROOVE should be replaced.

Writing Heuristics

The heuristic search algorithms of our framework – A^* and *Best First* – rely upon heuristics in order to perform efficiently. From a knowledge engineering standpoint it is desirable that users can define heuristics with the same notation they use to define the planning problem. As a first step, however, we have to identify the properties and functionalities which such a notation should provide. Therefore, we currently only provide an API with about 30 methods that ease the development of heuristic functions for GTS.

---

[1] http://www.fujaba.de

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

The design of the API was driven by the following observations:

- We think about graph nodes in terms of their types and their object names. For example, a node that represents a particular RailCab has the type "RailCab" and the object name "r1".
- The object name and the type are both labels of a node. Furthermore, a node has incoming and outgoing edges. These edges can have labels themselves. Source and target of an edge are nodes again.
- A node should allow us to easily access and analyze its close-by neighbors, *i.e.* the nodes which are at the source of an incoming edge or at the target of an outgoing edge.
- Checking reachabilty between nodes is crucial when developing heuristic functions for graphs. For example, we need to be able to check, if a RailCab node can reach a node that represents a station. The check should return the distance between the nodes or the list of nodes along the path. Furthermore, a reachability check should (optionally) take into account that edges are directed. It is also important that we can restrict the check in a way that only certain nodes are taken into account. For example, if we want to check reachability between a RailCab and a station, this check should use tracks but not Base Stations.

The API hides the internals of GROOVE's graph data structures. For example, nodes and edges are assigned unique identifiers internally but such identifiers are of no use to us as we do not know their meaning. We must also not forget that a user models a planning problem within FUJABA and thus might not even know anything about GROOVE and its internal graph representations.

In this paper, we evaluate four different heuristics which have been implemented using the API. Two well-known heuristics are for the 8-puzzle:

- h^1_{Puz} = the number of misplaced tiles, *i.e.* the number of tiles which are not in their goal position.
- h^2_{Puz} = the sum of the *Manhattan distances* of every misplaced tile.

Furthermore, two heuristics for the NBP problem:

- h^1_{NBP} = the sum of the shortest distance from the current position to the goal position for every RailCab; That is, we measure for each RailCab the minimal number of tracks between its current position and its goal position. Then, we add up all those values.
- $h^2_{NBP} = \infty$ if any RailCab can no longer reach its goal position from its current position. Otherwise return 0.

To give the reader an idea of how the API is used in practice, we provide an example for h^1_{NBP} in listing 1. Though the source code may not be completely self-explanatory, we do not explain the details in this paper. Rather, we provide the key observation from our experiments with the API: many properties of heuristic functions for GTS planning problems can be expressed using reachability tests between nodes in the graph.

We experimented with more sophisticated heuristics which, for example, take into account the possibility of building convoys and having different costs for different sorts of move actions (regular move, move in a 2-convoy, move in 3-convoy). We found that it quickly becomes quite cumbersome to write such heuristics by hand. Therefore, we developed a semi-automatic approach to writing heuristics that free the user from this burden.

Learning Heuristics

Using our API for writing heuristics, we can easily implement methods that extract feature values, *e.g.* the number of RailCabs or the number of tracks, from a given graph. Not having to decide how such feature values relate to the costs of solving a planning problem simplifies the development of a heuristic. For our framework, we developed an experience-based learning approach, *i.e.* we solve many problem instances and learn a heuristic estimate from experience. For instance, we define a set of features and store the values of such features in a so called *feature vector*. By providing many feature vectors together with the cost value of solving the corresponding problem, a learning algorithm can derive a function (a *regression function* to be precise) that predicts the costs based on a feature vector only. The approach we use for learning a regression function is called *Support Vector Machines* (SVM) (Boser, Guyon, and Vapnik 1992).

Instead of embedding a specific SVM implementation directly within our planning framework, we utilize a machine learning framework called WEKA[3] (Hall et al. 2009), which not only provides different SVMs but also other learning techniques. This provides the flexibility to experiment with different SVMs without the need to modify any code and also allows for future experiments with other learning approaches.

A learning algorithm can only yield meaningful results if it is trained on a sufficiently large data set. Asking the user to provide hundreds or thousands of different problem instances is undesirable and impractical. Our framework accounts for this by providing a *problem instance generator* which can generate many unique problem instances based on a single problem specification.

The language we use to write such a problem specification is ALLOY (Jackson 2002; 2006). It is based on first-order relational logic which facilitates an automatic analysis. ALLOY models can be executed and analyzed with the *Alloy Analyzer*[4]. The Alloy Analyzer translates an ALLOY model into a boolean formula and

---

[3] http://www.cs.waikato.ac.nz/ml/weka/

[4] http://alloy.mit.edu

```
public double getHeuriticWeight(Graph stateGraph, Set<HSRule> goalRules, Set<HSRule> actionRules) {
   Map<String, String> railCabStationMap = new HashMap<String, String>();
   double factor = 0;
   int distance = 0;

   // find out the costs for the ”move” rule
   for(HSRule r: actionRules) {
      if(r.getRuleName().equals(”move”)) {
         factor = r.getWeight(”Cost”); } }

   // get the goal graph
   Graph goalGraph = goalRules.iterator().next().getGrooveRule().lhs();

   // access the goal graph and find out where each RailCab shall go
   HSGraphAccess goalAcc = new HSGraphAccess(goalGraph);
   Set<HSNode> goalRailCabs = goalAcc.getAllNodesWithEdge(”RailCab”);
   for(HSNode goalRC: goalRailCabs) {
      HSNode goalTrack = goalRC.getSingleTargetOfEdgeLabel(”on”);
      String stationID = goalTrack.getSingleTargetOfEdgeLabel(”part_of”).getNodeID();
      railCabStationMap.put(goalRC.getNodeID(), stationID); }

   // determine the distance between the current RailCab positions and their goals
   HSGraphAccess acc = new HSGraphAccess(stateGraph);
   Set<HSNode> railCabs = acc.getAllNodesWithEdge(”RailCab”);
   Set<HSNode> tracks   = acc.getAllNodesWithEdge(”Track”);

   for(HSNode rc: railCabs){
      String goalStation = railCabStationMap.get(rc.getNodeID());
      HSNode goaltrack = acc.getNode(goalStation).getSingleSourceOfEdgeLabel(”part_of”);
      int i = rc.getSingleTargetOfEdgeLabel(”on”).canReach(goaltrack, tracks, true); }
      if( i > 0) // if i > 0 then goal is reachable
         distance += i;
   }
   return distance * factor;
}
```

Listing 1: Implementation of the heuristic h^1_{NBP}, using the heuristics API of our planning framework.

uses an “off-the-shelf” SAT solver to find satisfying assignments for such a formula. By enumerating over different satisfying assignments, we receive different instances of an ALLOY model. These instances can be used as training problems for the SVM.

We use the UML class diagram, which describes the general structure of a planning problem, to automatically generate a *skeleton* of an ALLOY specification. This skeleton needs to be manually extended with constraints such that an ALLOY instance represents a meaningful planning problem. Examples for such constraints would be: “Each RailCab can reach its goal station using tracks” or “If a track is monitored by more than one Base Station, than its successor and ancestor tracks have different Base Stations”.

After generating a sufficient amount of ALLOY instances, the planning framework automatically translates each instance into a GTS planning problem. Then, for each problem, a feature vector is created, based on the features specified by the user. After solving the problems optimally (*e.g.* by using A^* with an admissible heuristic), each resulting cost value is stored together with its corresponding feature vector in a WEKA input file. Based on this input file, we finally learn the regression function and encode it – using the API – as a heuristic for the planning framework.

An example of two different feature sets which we used to learn heuristic functions are:

- f^1_{Weka}: number of RailCabs; number of Tracks; number of stations in the goal rule; average distance to the goal station
- f^2_{Weka}: number of RailCabs not at their goal position; average distance to the goal station; average branching factor

Trained on 140 different planning problems, the SVM learned the following heuristic functions:

$$h^1_{Weka} = 5.4226{*}j_1{+}0.0769{*}j_2{-}0.8092{*}j_3{+}1.6148{*}j_4{-}2.5722$$

where j_1 = number of RailCabs; j_2 = number of tracks; j_3 = number of stations in the goal rule; and j_4 = average distance to the goal station.

$$h^2_{Weka} = 6.3639 * k_1 + 1.9876 * k_2 - 1.2123 * k_3 - 4.2353$$

where k_1 = number of RailCabs not at their goal position; k_2 = average distance to the goal station; and k_3 = average branching factor.

The entire process of generating the training data and learning the heuristic functions took about 35 minutes.

Related Work

Edelkamp and Rensink (Edelkamp and Rensink 2007) described the similarities and differences between planning tools and graph transformation tools. They found that “graph transformation systems provide a flexible, intuitive input specification for systems of change with a sound mathematical basis”. A performance comparison of the graph transformation tool GROOVE (Rensink 2004) and the heuristic search planner FF (Hoffmann and Nebel 2001) demonstrated that planners can vastly outperform the graph transformation tool. However, the paper also describes why a GTS planning problem might not be suited for translation into PDDL: PDDL does not support the creation/deletion of objects nor untyped domain objects.

Another paper by Edelkamp (Edelkamp, Jabbar, and Lafuente 2006) proposes several heuristic functions which can be used when performing heuristic search on GTS. While these heuristics – which can be encoded using our API – have the advantage of not being domain specific, they rely on the availability of a complete graph which defines the goal state. For the planning problem we are interested in, the goal states are typically defined using incomplete graphs which only state the properties of interest.

Röhs and Wehrheim (Röhs and Wehrheim 2010) have used GROOVE to solve planning problems using its built-in model checker. Their approach consists of the following steps: 1) modeling a planning problem using FUJABA, 2) translating the FUJABA graphs into GROOVE graphs, 3) using the GROOVE simulator to build a complete graph transition system, 4) finding a valid path from the start state to a goal state (valid means a path without forbidden states), 5) reporting the action names used along the path back to the user.

GROOVE's model checker (MC) is used to search for a counterexample to the following statement: "*there exists no path to a goal node without any forbidden states along the way*". If a counterexample can be found, it is a valid plan for the planning problem. The problem with the model checking approach is obvious: the generation of the entire state space (step 3) is very expensive and not necessary to solve a planning problem. We compare our heuristic search algorithms with the MC approach in the next section. Note that GROOVE allows to disable the exploration forbidden states using rule *priority*. We show the results of the MC approach with and without priorities.

Evaluation

We evaluated the implementation of our planning framework on different planning problems. As a point of reference, we used the model checking based planner developed by Röhs (Röhs and Wehrheim 2010). Both planners use GROOVE to perform the graph transformations and build the graph transition system. Considering the fact that our current implementation requires additional bookkeeping due to implementation details, we focus on the number of states and transitions rather than the runtime. Our experiments were carried out on a quadcore machine with an "Intel Core i7 Q820" processor (3.06GHz core speed), 8 GB RAM, running Windows 7 Professional 64 bit, Java 1.6.0_22 (32 bit) with a JVM heap size of 1.2 GB.

N-Puzzle Problems

The first problem we used for the evaluation was the n-puzzle. Remember that *Best First* (BF) returns the first solution it finds, whereas A^* returns an optimal solution (given an admissible heuristic).

One of the 8-puzzle problems, 8puzzle-06, is specifically modeled to be easily solvable. Only two slide actions are necessary to reach the goal state. The purpose of this problem instance is to demonstrate the disadvantage of checking for a goal state only after the entire state space has been generated, as it is done with the model checking planner (MC). The results of the experiments are shown in table 1

We did not try to solve the 15-puzzle using the MC planner or our planner with the h^1_{Puz} heuristic. With a state space of 1.3 trillion states, the problem is currently not feasible. For the 8-puzzle, both our algorithms find solutions for all problem instances. We observed that the model checking planner was able to generate the entire state space (approx. 180.000 states) but ran out of memory while performing the search for a counterexample.

NBP Problems

To evaluate the NBP case study, we used two different problems (NBP-B and NBP-M) and created twelve different problem instances. The instances vary in the number of tracks and the number of RailCabs used. The model checking planner was used a first time without any rule priorities and a second time with a priority of 1 for all rules which describe forbidden states. The heuristic h_{Empty} simply returns the value 0, *i.e.* the results show how the algorithms perform without any heuristic estimate. The heuristic h^1_{NBP} was modified to measure the distance-to-goal only (rather than costs) when used in combination with *Best-Frist*.

NBP-B Problems The first NBP problem models the situation that RailCabs have to move from Paderborn station to the stations Berlin and Leipzig, respectively. A single problem instance has the name "NBP-B-X-Y", where X represents the number of RailCabs and Y represents the number of tracks. The results of the experiments are shown in table 2.

NBP-M Problems In an NBP-M problem, RailCabs have to move from Paderborn station to Berlin station and Munich station, respectively. The track network allows to travel to Berlin directly or by taking a detour over Munich. It is, however, not possible to travel to Munich over the Berlin route. One RailCab is carrying dangerous cargo, which prevents it from joining a convoy.

Our findings for these problem instances are quite similar to ones from the NBP-B problems. The BF and A^* algorithms outperformed both model checking approaches. As we can see in table 2, the heuristic h^2_{NBP} performed very good for this particular problem, as it is likely for a RailCab to move along a route from which the goal station is unreachable.

Learned Heuristics Finally, we evaluated the learned heuristics h^1_{Weka} and h^2_{Weka} and compared them with the empty heuristic h_{Empty} and the manually written heuristics h^1_{NBP} and h^2_{NBP}. As the learned heuristics were used to estimate the costs, we only compared the results for the A^* algorithm. Table 3 shows

	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

Table 1: Results for the n-puzzle problem. For each search strategy the number of states, transitions and solving time is shown. The minimal number of explored states is highlighted in bold for each problem instance.

	MC without Prio			*MC with Prio*			*BF with h_{Empty}*			*BF with h^1_{NBP}*			*BF with h^2_{NBP}*			*A* with h_{Empty}*			*A* with h^1_{NBP}*			*A* with h^2_{NBP}*		
	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)
NBP-B-2-08	293	837	1	253	665	< 1	110	170	<1	**51**	74	<1	89	159	<1	268	642	<1	**88**	176	<1	204	457	<1
NBP-B-2-10	401	1011	1	315	797	< 1	**77**	108	<1	126	237	<1	177	312	<1	358	852	<1	**117**	217	<1	233	517	<1
NBP-B-2-15	776	1966	2	515	1247	1	200	377	<1	**123**	216	<1	138	193	<1	564	1326	<1	**199**	392	<1	293	647	<1
NBP-B-2-20	1301	3131	3	765	1797	2	202	320	<1	**131**	218	<1	135	196	<1	817	1892	1	**78**	132	<1	352	777	<1
NBP-B-3-08	7615	34182	59	4647	15773	21	291	610	<1	550	1473	1	**271**	432	<1	5782	19009	29	**233**	514	<1	3502	10136	13
NBP-B-3-10	12426	53300	132	6381	21029	29	908	2437	2	**206**	446	<1	535	855	<1	8249	25780	50	**356**	864	<1	3891	11020	15
NBP-B-3-15	34956	134530	234	13713	45017	50	**234**	244	<1	1689	5137	7	1587	3167	5	16706	53554	201	**2344**	6935	43	5031	14520	24
NBP-B-3-20	78052	277431	837	24795	80813	104	4428	13228	46	1232	3584	5	**346**	485	<1	29068	94063	638	**693**	1751	5	6171	18020	36
NBP-M-3-08	9332	54647	61	5329	23670	24	685	2945	2	3594	17526	30	**263**	443	<1	5281	22346	26	2252	7691	8	**1788**	6840	6
NBP-M-3-10	14008	78031	71	7570	32439	26	2409	10365	16	4557	21681	42	**737**	1655	1	8041	36001	57	**2418**	9826	11	2541	9558	10
NBP-M-3-15	39584	194855	*	17667	65815	281	6810	24655	107	5595	20744	52	**751**	1277	1	19877	75207	274	8943	31727	151	**5087**	15969	27
NBP-M-3-20	48587	227263	*	23477	83978	95	15662	55782	500	10220	36913	106	**2048**	4149	8	27191	99387	485	10253	33918	140	**5678**	17674	35

* Exception while executing Dijkstra's algorithm.

Table 2: Results for the NBP-B and NBP-M problem instances. For each search strategy the number of states, transitions and solving time is shown. The minimal number of explored states is highlighted in bold for *Best-First* and A^*, respectively.

the number of states, transitions and the cost-value of the resulting plan.

We observe that h^2_{Weka} performed above the average. For 6 out of 12 problems it used the fewest number of states to find a solution. Also, for 6 out of 12 problems it used the fewest number of states while returning an optimal solution. It worked particularly well on problems of category NBP-M and was otherwise only outperformed by h^1_{NBP}, which – on average – had higher cost values.

It may seem surprising that h^1_{Weka} performed very similar to h^2_{NBP}. We explain this with the fact that the "average distance" calculation within h^1_{Weka} also returns a very high value in case a RailCab can no longer reach its goal. This equals the implementation of h^1_{NBP}. Taking this detail into account, we can conclude that h^1_{Weka} only performs as a cut-off heuristic and is otherwise as ineffective in estimating the costs as h^2_{NBP} (which estimates 0).

The findings of the experiments demonstrates that a learned heuristic should cover dynamic aspects of the problem, as it is done with h^2_{Weka}. If we learn functions based on static feature like "number of RailCabs", the resulting function is of little use. Trained with the right features, however, a learned heuristic can be effective.

Conclusion

In this paper, we presented a framework for heuristic search-based planning for graph transformation systems. The planning framework uses the GROOVE graph transformation tool to perform the necessary graph transformations. Planning problems are modeled in FUJABA, using a graphical notation called story patterns. We developed an API that enables users to easily write heuristic functions for GTS planning problems. Furthermore, we presented an approach to semi-automatically learn heuristic functions using Support Vector Machines and constraint-based problem instance generation. The efficiency and effectiveness of the heuristic search-based planner was compared to a previously proposed model checking based planner. The results indicate that our planner is superior with respect to the number of states that need to be explored in order to find a solution.

The observation that a heuristic search algorithm expands less states than a model checker, in order to find a plan, does not come as a surprise. At the beginning of our research, however, we were confronted with the situation of not being able to encode heuristic information that would improve the solving of a GTS planning problem. Having overcome the technical issues, we were surprised by the actual performance improvements us-

	A^* with h_{Empty}			A^* with h^1_{NBP}			A^* with h^2_{NBP}			A^* with h^1_{WEKA}			A^* with h^2_{WEKA}		
	#states	#trans	Costs	#states	#trans	Costs	#states	#trans	Costs	#states	#trans	Costs	#states	#trans	Costs
NBP-B-2-08	268	642	17	88	176	21	204	457	17	185	404	17	**74**	114	21
NBP-B-2-10	358	852	23	117	217	23	233	517	23	226	488	23	**92**	135	23
NBP-B-2-15	564	1326	38	**199**	392	38	293	647	38	288	628	38	255	544	38
NBP-B-2-20	817	1892	53	**78**	132	53	352	777	53	348	758	53	315	674	53
NBP-B-3-08	5782	19009	26	**233**	514	31	3502	10136	26	3500	10210	26	604	1210	27
NBP-B-3-10	8249	25780	32	**356**	864	33	3891	11020	32	3891	11020	32	2660	6762	33
NBP-B-3-15	16706	53554	47	**2344**	6935	48	5031	14520	47	5032	14520	47	4462	12308	47
NBP-B-3-20	29068	94063	62	**693**	1751	63	6171	18020	62	6171	18020	62	5602	15808	62
NBP-M-3-08	5281	22346	19	2252	7691	21	1788	6840	19	1749	6584	19	**965**	3469	21
NBP-M-3-10	8041	36001	27	2418	9826	27	2541	9558	27	2583	9520	27	**1776**	6004	27
NBP-M-3-15	19877	75207	40	8943	31727	45	5087	15969	40	4978	15532	40	**4044**	12274	40
NBP-M-3-20	27191	99387	51	10253	33918	51	5678	17674	51	5672	17660	51	**5443**	16836	51

Table 3: Results for the learned heuristics. The third column of each search strategy shows the costs of the solution. A^* with h_{Empty} is optimal. The fewest number of states is highlighted in bold. The fewest number of states while being optimal is highlighted using underlining.

ing even simple heuristics as the ones described above. Furthermore, we found that the encoding of a planning problem using a GTS not only gives the advantage of having a graphical notation. It also aids the development of heuristics as one mostly thinks about the planning problem in terms simple properties such as nodes, neighbor-nodes or reachabilty between nodes.

As future work, we plan to analyze additional planning problems to further test our heuristics API. Once the API has reached a stable state, one can investigate how to write heuristics using the same graphical notations that are used to model the planning problem. Furthermore, we would like to explore potential performance benefits of using other graph transformation tools and changing our framework to account for multi-core architectures.

References

Boser, B. E.; Guyon, I. M.; and Vapnik, V. N. 1992. A Training Algorithm for Optimal Margin Classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, 144–152. New York, NY, USA: ACM.

Edelkamp, S., and Rensink, A. 2007. Graph Transformation and AI Planning. In Edelkamp, S., and Frank, J., eds., *Knowledge Engineering Competition (ICKEPS)*. Canberra, Australia: Australian National University.

Edelkamp, S.; Jabbar, S.; and Lafuente, A. 2006. Heuristic Search for the Analysis of Graph Transition Systems. In Corradini, A.; Ehrig, H.; Montanari, U.; Ribeiro, L.; and Rozenberg, G., eds., *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, 414–429. Springer.

Fischer, T.; Niere, J.; Torunski, L.; and Zündorf, A. 2000. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, 296–309. London, UK: Springer.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilins, D. 1998. PDDL - the Planning Domain Definition Language, version 1.2. Cvc tr-98-003/dcs tr-1165, Yale Center for Computational Vision and Control.

Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; and Witten, I. H. 2009. The WEKA data mining software: an update. *SIGKDD Explorations* 11(1):10–18.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Jackson, D. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* 11(2):256–290.

Jackson, D. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

OMG. 2010. OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. Technical Report formal/2010-05-03.

Rensink, A. 2004. The GROOVE Simulator: A Tool for State Space Generation. In Pfalz, J.; Nagl, M.; and Böhlen, B., eds., *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, 479–485. Springer.

Röhs, M. 2009. Sichere Konfigurationsplannung adaptiver Systeme durch Model Checking. Master's thesis, Universität Paderborn.

Röhs, M., and Wehrheim, H. 2010. Sichere Konfigurationsplanung selbst-adaptierender Systeme durch Model Checking. In Gausemeier, J.; Rammig, F.; Schäfer, W.; and Trächtler, A., eds., *Entwurf mechatronischer Systeme*, volume 272, 253–265.

Vaquero, T.; Tonidandel, F.; and Silva, J. 2005. The itSIMPLE tool for Modeling Planning Domains. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning and Scheduling (ICKEPS)*.