UNIVERSITÄT BASEL

# Generating and Evaluating Unsolvable STRIPS Planning Instances for Classical Planning

Bachelor's Thesis

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

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

Dietrich Zerr
d.zerr@stud.unibas.ch
11-058-864

02.08.2014

# Abstract

In classical planning, heuristic search is a popular approach to solving problems very efficiently. The objective of planning is to find a sequence of actions that can be applied to a given problem and that leads to a goal state. For this purpose, there are many heuristics. They are often a big help if a problem has a solution, but what happens if a problem does not have one? Which heuristics can help proving unsolvability without exploring the whole state space? How efficient are they? Admissible heuristics can be used for this purpose because they never overestimate the distance to a goal state and are therefore able to safely cut off parts of the search space. This makes it potentially easier to prove unsolvability.

In this project we developed a problem generator to automatically create unsolvable problem instances and used those generated instances to see how different admissible heuristics perform on them. We used the Japanese puzzle game Sokoban as the first problem because it has a high complexity but is still easy to understand and to imagine for humans. As second problem, we used a logistical problem called NoMystery because unlike Sokoban it is a resource constrained problem and therefore a good supplement to our experiments. Furthermore, unsolvability occurs rather 'naturally' in these two domains and does not seem forced.

# Table of Contents

# 1 Introduction

Domain-independent classical planning is the process of finding a sequence of actions for a given planning task that leads to a goal state. Such a sequence is called plan. Heuristic search is a popular approach to find such a plan more efficiently. Heuristics approximate the distance from any given state to a goal state, which can often be a big help in finding plans faster and with less memory consumption than blindly searching the whole search space. This works well if a problem has a solution, but what happens if a problem does not have one?

One objective of this bachelor's thesis was to find problem domains that can be used to create unsolvable problems, to think of how such problems could look like and to implement a problem generator that generates unsolvable problem instances in PDDL, which then can be used to run tests in the planning system Fast Downward [1]. The other objective was to find out which heuristics can help proving unsolvability faster and to see how much time and memory they need. We used several admissible heuristics for this purpose because they never overestimate the distance to a goal state and are therefore able to safely cut off parts of the search space where the distance is infinite. The resulting, smaller state space makes it potentially easier to prove unsolvability.

We decided to use the Japanese puzzle-game Sokoban[1] as the first problem domain. In Sokoban, the player takes the role of a warehouse keeper (Japanese: *sōkoban*) and has to push boxes onto goal fields in order to win. The player can only move vertically and horizontally and only push one box at a time. In particular, the player cannot pull boxes. As a result, boxes can easily get stuck in corners and along walls. The second problem domain we used was a logistical problem called NoMystery. It is a transportation problem with multiple locations, vehicles and packages. Each package needs to be moved from some location to another. The locations also have a certain amount of fuel, and a vehicle cannot leave if there is no more fuel left. We took these two problem domains because they are both simple to understand for humans while having a high complexity. The way unsolvability occurs in these domains is rather 'natural', meaning that such unsolvabilities are likely to appear when creating problem instances and are not just a result of carelessness

[1] http://www.sokoban.jp/

or unrealistic problem structure. Furthermore these domains complement each other well because NoMystery, unlike Sokoban, is a resource constrained problem.

# 2 Preliminaries

This chapter provides the general terms for planning and heuristic search as well as an introduction to the two planning problems we used in this bachelor's thesis: Sokoban and NoMystery.

## 2.1 Planning

In the area of artificial intelligence, the term *planning* describes the solving of *planning problems*. Planning problems are a formal description of problems we meet in real life such as navigation, logistical distribution of cargo or puzzle games like the sliding tile puzzle and the Towers of Hanoi. Such a planning problem is typically described by a *set of variables* where each of the variables can have different values. The initial assignment of these variables is called *initial state*, whereas *goal states* are the desired assignments of a given problem. There is a *set of actions* to change the assignments of variables. Each action has a *precondition* which requires the variables to have certain values and an *effect* that describes the changes to the variables after the action has been executed. An action can only be executed if its precondition is fulfilled. Furthermore all actions have non-negative *costs*. A sequence of actions that can be applied to the initial state and leads to a goal state is called a *plan*. The costs of a plan are the sum of the costs of all its actions. A plan is called *optimal* if it has the lowest possible costs. If there is no plan to a given problem then the problem is *unsolvable*.

## 2.2 Heuristics

Finding a plan can require huge amounts of time and memory, so it is desirable to avoid searching through the whole state space. Programs that are specified on solving problems are called *planners*. Many popular planners are based on heuristic search. When such a planner moves through the state space, it would like to know how far away a state is from a goal state. Since finding the exact distance would require to know the whole state space, a *heuristic function* is used to approximate the distance. Heuristics try to approximate the real distance as well as possible while still being fast and cheap in their execution. There are domain specific heuristics which use knowledge about a certain kind of problems and domain independent heuristics which try to be good at all kinds of problems without knowing what the problem is about.

In this bachelor's thesis we use several domain independent admissible heuristics. Heuristics are admissible if they never overestimate the distance to a goal state. In particular, if an admissible heuristic says that the distance for a given state is infinite, then the real distance must be infinite as well, which means that the goal is unreachable from this state. Using this property, a planner can safely cut off parts of the state space and make it smaller.

## 2.3 Sokoban

Sokoban is a puzzle game created by Hiroyuki Imabayashi in 1981. It was published in Japan in 1982 by the software house Tinkering Rabbit.[2, 3] In Sokoban, the player takes the role of a warehouse keeper (Japanese: *sōkoban*) and has to push boxes (in some versions called stones) onto goal fields in order to win. The player can only move vertically and horizontally and only push one box at a time. In particular, the player cannot pull boxes. An example of a Sokoban instance is given in figure 2.1.

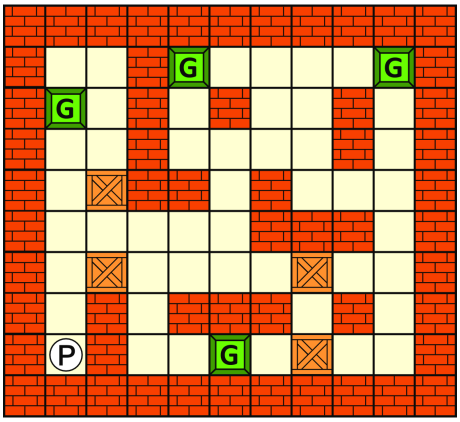


Figure 2.1: Example of a solvable Sokoban instance.
P = Player, G = Goal

A Sokoban field is surrounded by walls and usually has some walls in the field as well. The player can walk over empty fields and goal fields but not over walls or boxes. The player can freely decide which boxes to push on which goal fields and in which order to move them. As the player, boxes can only be moved over empty fields and goal fields but not over walls or other boxes. This also means that boxes can still be moved after they reach a goal field. Since boxes cannot be pulled, the player has to be careful not to push them into dead ends by moving them into corners, against walls or into certain constellations with other boxes. Sokoban instances for human players are created with the goal in mind to make them tricky. This means that there are usually only few ways to solve them and many other

[2] http://en.wikipedia.org/wiki/Sokoban
[3] http://www.sokoban.jp/

possible moves that lead to situations where a box gets stuck and the player cannot finish the instance anymore.

Sokoban is an interesting domain for artificial intelligence and planning because it has a high complexity and is yet easy to understand and to imagine for humans because it is close to the real world problem of moving boxes in a warehouse, which could also be done by a robot.

## 2.4 NoMystery

NoMystery is a transportation domain. The version we were using in our experiments was introduced in the International Planning Competition (IPC) 2011. It is the undisguised version of the Mystery domain created by Drew McDermott and introduced in the ICP 1998.[4] Mystery and NoMystery are essentially the same domain but with different names for actions and variables in order to disguise the true nature of the domain. Since our planner is domain independent, it is more convenient for us to use the NoMystery domain.

The domain is presented as a planar graph with multiple nodes which represent *locations*. There can also be *vehicles* and *packages* at these locations. The goal of a NoMystery problem is to move all packages to their respective goal location. Each vehicle has a *capacity* and can load packages up to its capacity limit if vehicle and package are on the same location. If two locations are connected, vehicles can move in both directions, but a vehicle can only leave a location if there is enough *fuel* on this location. Each time a vehicle leaves a location, the amount of fuel on this location is decreased by one unit. An example of a NoMystery instance is given in figure 2.2.

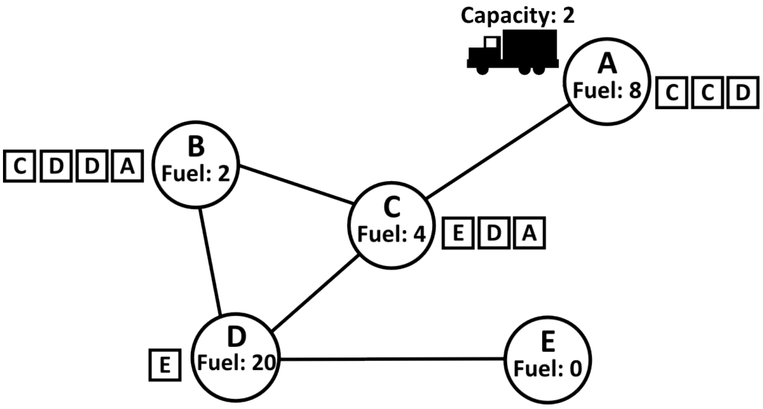


Figure 2.2: Example of a solvable NoMystery instance.
Circles are locations, squares are packages, letters on packages are destinations. For example, two packages must be transported from A to C and one from A to D.

We added NoMystery to our experiments because unlike Sokoban it is a resource constrained problem. We wanted to test whether heuristics perform differently on unsolvable problems of different domains.

[4] ftp://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html

# 3
# Generating Unsolvable Problems

The first two parts of this chapter explain the different properties of unsolvable problem instances for Sokoban and NoMystery and provide some examples. The third part explains how our problem generator creates unsolvable instances from a more technical side.

## 3.1 Unsolvable Sokoban Problems

When we started constructing different unsolvable instances for Sokoban, which we called *modes*, we noticed that some instances share crucial similarities in the way how unsolvability is achieved. We decided to put similar modes into *groups*. Overall we came up with 12 different modes, which are split into 4 groups. The constellation of boxes, goals, walls and free fields which makes a problem instance unsolvable, is called *unsolvability pattern*. 10 out of 12 modes have such patterns. The images in the detailed explanations below show all of them. In our case, the unsolvability pattern is always placed in the lower right corner of the Sokoban field, while the player is always placed at the opposite side, in the top left corner. The free space outside of the pattern may be filled randomly with boxes and goal fields, to increase the difficulty. We refrain from placing any random walls because it is important to make sure that the random placement is never cause of additional unsolvability, as it would distort the measurements. The easiest way to assure that randomly placed boxes are never cause of unsolvability is to never place two boxes in the 8-neighborhood of each other or at the borders of the field. Note that all patterns should include enough empty fields around their wall fields and boxes in order to keep all properties of the pattern. Otherwise, certain moves directly around the pattern might become unavailable on very small fields. This would make the pattern easier than intended and incomparable to larger field sizes. Those empty fields also ensure that randomly placed boxes are not too close to the pattern's walls and boxes and therefore cannot cause unwanted unsolvability. Goal fields can theoretically be placed anywhere outside of the pattern but we refrain from placing them below boxes or the player in order to have an easier visual representation of the generated instance in ASCII art.

All examples for the first two groups have one additional random box and goal because these problems need at least two boxes to show their different properties, while their patterns have only one box. Examples for the last two groups have no random boxes or goals as they are not necessary.

## Group 'one box cannot reach any goal' – modes 0 and 1

These are very simple unsolvabilities where one of the boxes can never reach any of the goals.

### Mode 0

One box is placed at the upper border of the Sokoban field without a goal field there. Since we cannot pull it away from the wall, it can never reach any of the goal fields. (There is no unsolvability pattern to show.)

### Mode 1

One box is isolated by walls together with one goal, so the player cannot reach it. An example for mode 1 is given in figure 3.1.

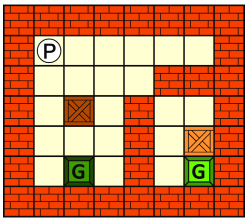


Figure 3.1: Mode 1 example - field size 6 × 5, one random box/goal
P = Player, G = Goal, darker colored box and goal are the random ones

## Group 'one goal cannot be reached' – modes 2, 3, 4, 5 and 6

This group of unsolvability makes it impossible to have all boxes on goal fields at the same time because one goal field is either missing or unreachable. In contrast to the previous group 'one box cannot reach any goal' it is possible to have each box on a goal field at some point (as long as it is more than one box) but not all at the same time.

### Mode 2

There is one goal field less than there are boxes. (There is no unsolvability pattern to show.)

### Mode 3

One goal field is isolated by walls. An example for mode 3 is given in figure 3.2.

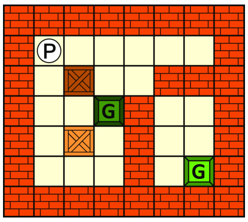


Figure 3.2: Mode 3 example - field size 6 × 5, one random box/goal
P = Player, G = Goal, darker colored box and goal are the random ones

### Mode 4

Boxes cannot be pushed onto one of the goal fields because it is almost completely enclosed by walls. There is only a narrow way for the player to reach it. An example for mode 4 is given in figure 3.3.

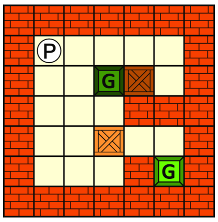


Figure 3.3: Mode 4 example - field size 5 × 5, one random box/goal
P = Player, G = Goal, darker colored box and goal are the random ones

### Mode 5

Boxes cannot be pushed onto one of the goal fields, but contrary to mode 4, the number of walls around it is kept at the minimum. An example for mode 5 is given in figure 3.4.

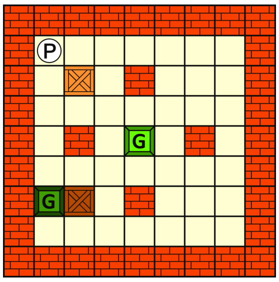


Figure 3.4: Mode 5 example - field size 7 × 7, one random box/goal
P = Player, G = Goal, darker colored box and goal are the random ones

### Mode 6

Similar to mode 5, but the condition for pushing a box onto the unreachable goal field is satisfied. The player could theoretically stand behind the box to push it but cannot reach that position once the box is in place. An example for mode 6 is given in figure 3.5.

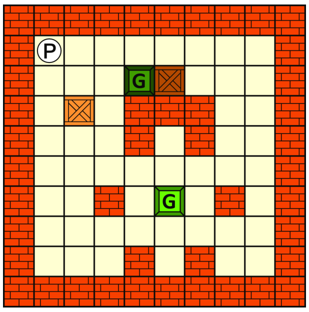


Figure 3.5: Mode 6 example - field size 8 × 8, one random box/goal
P = Player, G = Goal, darker colored box and goal are the random ones

### Group 'two boxes block each other' – modes 7, 8 and 9

This group of unsolvability involves two boxes that block each other from reaching a goal. The player would have to move one of the boxes into a dead-end position, where it can never reach any goal again, to be able to move the other box to a goal. So it is possible for each box to get to a goal field but not for all boxes at the same time. Problems of this group are similar to problems of the first group 'one box cannot reach any goal', with the difference that it is not clear from the start which box turns into that one box that cannot reach any goal.

#### Mode 7

In order to move a box to a goal, the player would have to push one of the boxes into a gap in the wall, where it cannot be pulled out anymore. An example for mode 7 is given in figure 3.6.

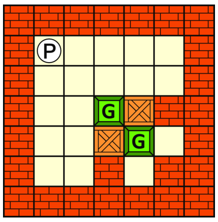


Figure 3.6: Mode 7 example - field size 5 × 5, no random boxes/goals
P = Player, G = Goal

#### Mode 8

Very similar to mode 7 but with fewer possible moves because the right box is placed at the right border and can only be moved along the wall. An example for mode 8 is given in figure 3.7.

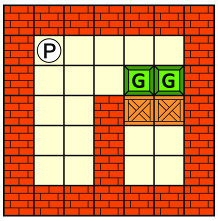


Figure 3.7: Mode 8 example - field size 5 × 5, no random boxes/goals
P = Player, G = Goal

### Mode 9

In contrast to modes 7 and 8, in mode 9 the player can still move the boxes along the wall after having pushed them into a dead end. Additionally, both boxes in the pattern have two possibilities to be pushed into a dead end, to the upper or the lower wall. The unsolvability pattern of mode 9 is also a good example for a pattern that would become significantly easier if the horizontal field size is not large enough. If the pattern does not at least contain two rows of empty fields on its left side, the boxes inside the pattern cannot leave it and random boxes cannot get inside through the entrance. This would divide the problem into smaller independent subproblems. An example for mode 9 is given in figure 3.8.

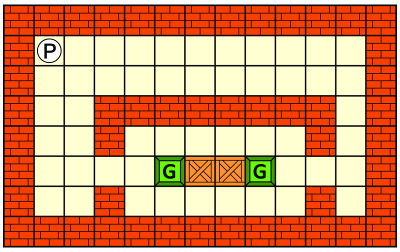


Figure 3.8: Mode 9 example - field size 11 × 6, no random boxes/goals
P = Player, G = Goal

## Group ’goal blocks player’ – modes 10 and 11

This group of unsolvability involves at least two goal fields that block the player’s path once they have a box on them, making it impossible to finish the remaining goals. In contrast to the previous groups, this time the player gets into some kind of dead end instead of a box.

### Mode 10

The structure of this pattern can be seen as two ’chambers’. The boxes are like doors, which can only be closed from inside the chamber. Once one of these boxes is on its goal position, the player cannot reach the other one. An example for mode 10 is given in figure 3.9.

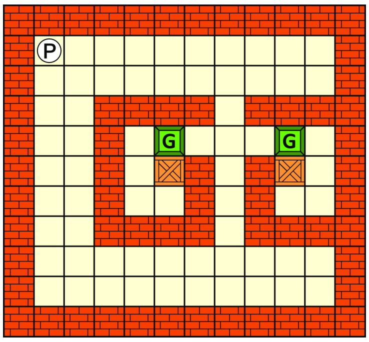


Figure 3.9: Mode 10 example - field size 10 × 9, no random boxes/goals
P = Player, G = Goal

### Mode 11

This pattern has four goals and one 'chamber' with two exits. Each exit must be approached from both sides to push the boxes on the goal fields. The player can only occupy up to three out of four goal fields with boxes, then both paths are blocked and the player is stuck either inside or outside the 'chamber'. An example for mode 11 is given in figure 3.10.

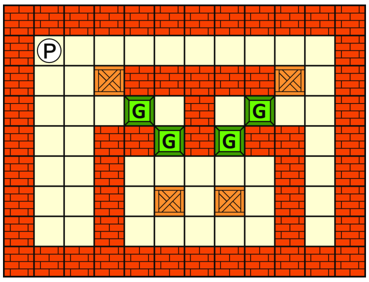


Figure 3.10: Mode 11 example - field size 10 × 7, no random boxes/goals
P = Player, G = Goal

## 3.2 Unsolvable NoMystery Problems

Compared with Sokoban, NoMystery does not have any groups, modes or patterns. The general structure of our NoMystery problems is simple. An instance is represented by two subgraphs where the nodes are locations. One of the subgraphs is solvable, whereas the other one is unsolvable. Each subgraph has one vehicle, at least two nodes and zero or more packages. The solvable subgraph is always a clique, which means that all its nodes are interconnected. The starting positions and destinations of its packages are chosen randomly, and it is ensured that there is enough fuel on all nodes. Start and destination are never the same location. The unsolvable subgraph can either be a clique or a star, where the node with the vehicle is connected to each other node of the subgraph. In both cases all packages are placed on the node with the vehicle. Their destinations are distributed equally across the other nodes of the subgraph. To create unsolvability in the unsolvable subgraph, there is one unit fuel less on the node with the vehicle than there are packages. Both vehicles have a capacity of one and leaving a node always costs one unit fuel, so the last package can never be delivered. All other nodes in the unsolvable subgraph have enough fuel. Additionally, the problem can be made potentially harder by connecting the two subgraphs with a single connection between the two nodes that have the vehicles on them, making it one large graph. An example is given in figure 3.11.

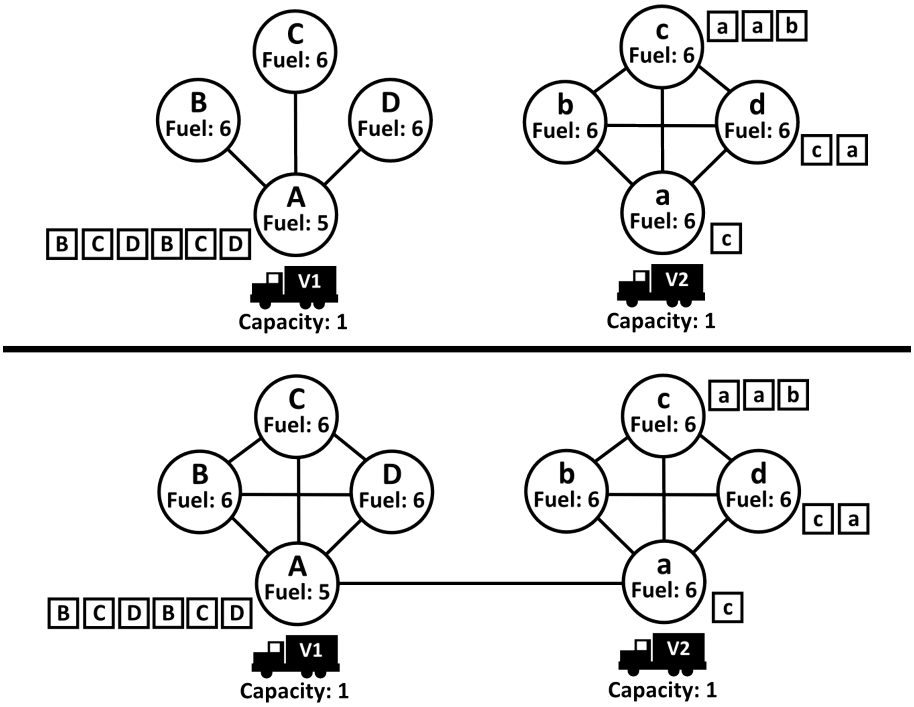


Figure 3.11: Example for two NoMystery instances with 4 locations and 6 packages in each subgraph. The left subgraphs are unsolvable, the right subgraphs are solvable. The upper graph shows the problem without connection between the two subgraphs and with the unsolvable subgraph as star. The lower graph shows the problem with a connection between the two subgraphs and with the unsolvable subgraph as clique.

The idea for this subgraph-structure stems from the paper *Fast Detection of Unsolvable Planning Instances Using Local Consistency* from Bäckström at al. (2013) [2] where they used a similar problem domain called *Trucks*. We adopted their structure, adjusted it to NoMystery and enhanced it as described above. In contrast to Sokoban, there is only one group of non-trivial unsolvability in NoMystery, which is lack of fuel. Trivial unsolvabilities like having no vehicle or separating a package or destination node completely from the graph are not interesting to us. They would likely be solved by our solver's pre-processing step, as we found out during our experiments on Sokoban problems from the first group 'one box cannot reach any goal', which have such a trivial structure. The decision to set the vehicle's capacity to one is not a limitation since higher capacity limits simply result in less movement and less fuel consumption. Having a capacity of one makes it easier to implement the generator and define the test cases. Limiting the distances of all packages to their respective destination to one move is no limitation either.

## 3.3 The Problem Generator

Our problem generator is written in Java and uses the strategy pattern [3]. Strategy pattern means we have a main class (*Problemgenerator*) and an interface (*Problem*). All other classes are specific problems that have to implement the interface *Problem*. This way, the program can be enhanced by more domains anytime without touching already existing code. The user just needs to provide the name of the specific problem class, e.g. Sokoban, to let the problem generator generate Sokoban problems. Our problem instances are formulated in the Planning Domain Definition Language (PDDL), but since the output string is generated by the specific problem classes, other problems could as well use any other representation. There is also a *properties* file where the user can specify the output directory, file name suffixes and file name endings. A call from the command line looks like this:

```
java Problemgenerator [⟨#problems⟩] ⟨problem-class-name⟩ [⟨prop1⟩] [⟨prop2⟩] [...]
```

The user can specify the number of problem instances that should be generated with the optional argument [⟨#problems⟩]. If it is missing, only one instance will be generated. The other optional arguments [⟨prop1⟩] [⟨prop2⟩] [...] are an indefinitely long list of integer numbers that can be used by the specific problem implementation as desired.

### Generating Sokoban Instances

When calling the problem generator for Sokoban, the user must provide a number for the mode. The mode is the only mandatory argument. It specifies which unsolvability pattern will be included into the problem. The mode numbers are the same as introduced previously in this chapter. Note that only one unsolvability pattern can be included into a problem. Optionally, the user can provide two more numbers for the field size (horizontally and vertically) and a forth number for the amount of boxes, which includes the boxes in the pattern. Default values will be used if these properties are not specified. If not all boxes and goal fields are used up by the pattern, the remaining boxes and goals are distributed randomly across the free space, as long as there is enough space for them. The size of the field will be automatically set to a minimum if the specified size is too small. The minimum field size depends on the selected mode and is always large enough to have free space for at least one random box and goal. Note that a mode will always place its fixed boxes and goals from the pattern, even if the specified number is smaller.

### Generating NoMystery Instances

When calling the problem generator for NoMystery, the user can provide the number of nodes and packages for each subgraph, decide whether the unsolvable subgraph should be a clique or a star and whether the two subgraphs should be connected. These arguments are all optional and default values will be used if they are not specified. While setting the number of packages to zero is allowed in both subgraphs, the user cannot set the number of nodes in a subgraph below two.

# 4 Experimental Results

The experiments were run with the planner Fast Downward [1] and the program lab[5] to define jobs (experiment setups as python scripts). To ensure comparable results, we used the computer cluster of the University of Basel, which has Intel Xeon E5-2660 CPUs running at 2.2 GHz.

We used different heuristics for the experiments to see how they compare with each other. Those heuristics were

1. the blind heuristic (or blind search) to have a baseline
2. the maximum heuristic ($\mathrm{h}^{\mathrm{max}}$) [4], which is for our purpose of identifying unsolvability identical to additive, FF [5] and LM-cut [6] heuristics, which are all based on delete relaxation
3. the Pattern Database heuristic (PDB) [7]
4. the Incremental Pattern Database heuristic (iPDB) [8]
5. the Merge-and-Shrink heuristic (M&S) [9][10][11]
6. the $\mathrm{h}^2$ (or generally $\mathrm{h}^{\mathrm{m}}$) [12] heuristic, where the implementation in Fast Downward is very slow at the time of this thesis. We still took it in to see how it fares.

All heuristics were run in eager greedy search with their respective default settings, except M&S, where we changed the settings[6] due to the fact that its computation of an abstraction consumed too much memory, even on small problems, while a smaller abstraction would already be enough to prove unsolvability.

## 4.1 Sokoban

In this section, we discuss our results for Sokoban. The first part explains how we decided on suitable parameters for the instances, the second part shows and explains the results of our experiments for the different Sokoban modes in detail.

[5] http://lab.readthedocs.org/en/latest/

[6] merge_strategy = merge_dfp, shrink_strategy = shrink_bisimulation(max_states = 50000)

### 4.1.1 Parameter Finding

To get useful experimental results on Sokoban, we first needed to find a suitable complexity for the test instances. Complexity in our case means the size of the field and the amount of boxes. The unsolvability patterns contribute to the complexity as well by adding boxes, walls and structures, so every mode has to be tuned separately. A suitable complexity is one that is not too hard, so that preferably all heuristics detect the unsolvability at some point, but also not too easy, so that there is enough space to see the differences between them. For example, if it is too easy and half of the heuristics finish in 0.1 seconds, we cannot tell which one is faster.

To actually find a suitable complexity, we simply tried around. The configurations used in the experiments that we describe here were found during those calibration runs. Additionally, it turned out that group 'one box cannot reach any goal' problems were already solved by Fast Downward's pre-processing step, which discovered the problem to be unsolvable independent of field size or number of boxes. Thus we omitted group 'one box cannot reach any goal' from further experiments and considered it being too trivial.

### 4.1.2 Results

For convenience reasons, we will from now on refer to "proving unsolvability" by using the term *solve*, since proving unsolvability is actually the solution to an unsolvable problem.

The memory limit was set to 3072 MB and the time limit to 30 minutes for all following experiments. We made 50 instances of each mode in two different complexities - so in the end 100 of each mode.

Note that only heuristics that were able to solve all 50 instances are shown in the bar charts below.

#### Group 'one goal cannot be reached'

For modes 2, 3 and 4, we created instances with field sizes $6 \times 6$ and $8 \times 8$, each with 3 boxes. For modes 5 and 6, we created instances with field size $8 \times 8$ with 2 and with 3 boxes.

The general trend within group 'one goal cannot be reached' problems is that the more fields are taken away by the pattern, the easier the problem becomes, since there is less space to move. Wall tiles in the middle of the Sokoban field also have more impact than wall tiles at the borders, as they influence more free fields around them.

**Heuristics comparison** Figure 4.1 gives an overview over the results for mode 2 and is representative for modes 3, 4, 5 and 6, which have very similar results.

Generally, group 'one goal cannot be reached' problems are solved very fast by iPDB and PDB in both complexities. $h^{max}$ is rather slow and becomes even slower with increasing complexity. All three of them need a lot more memory with increasing complexity. M&S is more efficient with harder problems and scales well with increasing complexity, but still needs longer than the other heuristics and takes a lot of memory. Thanks to its good scalability, it is at least faster than $h^{max}$ on the higher complexity versions. Blind search

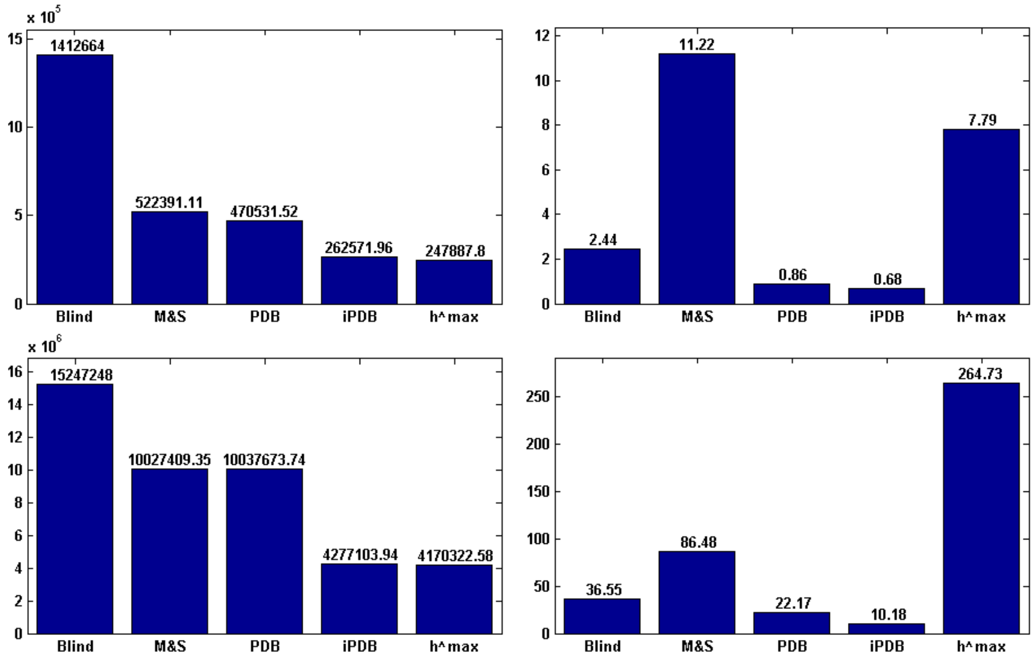


Figure 4.1: Group 'one goal cannot be reached', Mode 2 - average expansions (left) and average total time in seconds (right) over 50 instances each. The two top charts represent instances of field size 6 × 6, the two bottom charts of field size 8 × 8. All of them have 3 boxes.

needs a lot of memory and more expansions than the other heuristics but is still fast. The results for modes 5 and 6, where not the size but the amount of boxes changed, show a bigger jump in expansions and time consumption, except for M&S, which scales very well again. iPDB and $h^{max}$ have a very similar amount of expansions on all group 'one goal cannot be reached' problems. M&S and PDB even have an identical amount of expansions on all higher complexity problems and a very similar one on the lower complexity of modes 2 and 3, but M&S has more expansions on the lower complexity of modes 4, 5 and 6. $h^2$ could solve all lower complexity problems of mode 5 and 6 and two of mode 3. Since the implementation is slow, it cannot compete with the others in terms of time, but once it has calculated the heuristic, it recognizes the unsolvability instantly (0 expansions). That seems to be a general property of $h^2$ on the Sokoban domain as it holds true for every instance in every group it has solved in our experiments.

### Group 'two boxes block each other'

For modes 7 and 8, we created instances with field sizes 6 × 6 and 8 × 8, each with 4 boxes. For mode 9, we created instances with field sizes 11 × 6 and 11 × 8, each with 4 boxes.

Even though modes 7 and 8 are generally very similar, mode 7 has a larger search space because it has more possible actions. This has mainly to do with the fact that one of the boxes in mode 8 cannot be pushed away from the wall, reducing its movement dramatically. Mode 9 has a larger search space than modes 7 and 8 because the instance is larger and the box that has to be pushed to a wall can still be moved along the wall.

**Heuristics comparison** Figure 4.2 gives an overview over the results for mode 8 and is representative for mode 7 with the differences mentioned further below. Figure 4.3 gives an overview for the lower complexity version of mode 9 only, since the higher complexity version could not be solved by most heuristics.

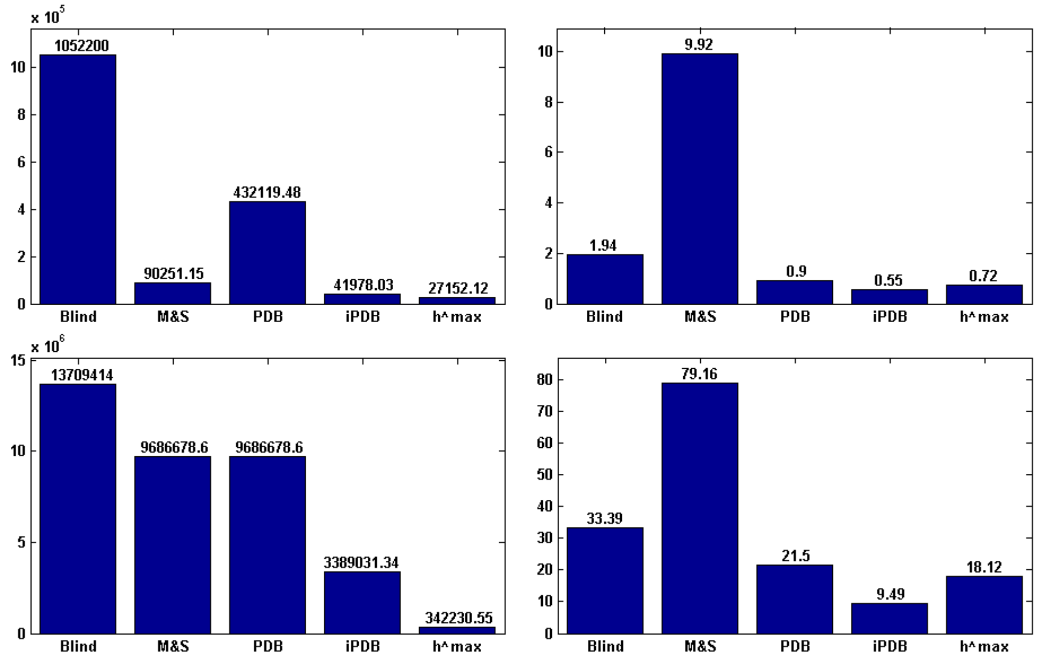


Figure 4.2: Group 'two boxes block each other', Mode 8 - average expansions (left) and average total time in seconds (right) over 50 instances each. The two top charts represent instances of field size 6 × 6, the two bottom charts of field size 8 × 8. All of them have 4 boxes.

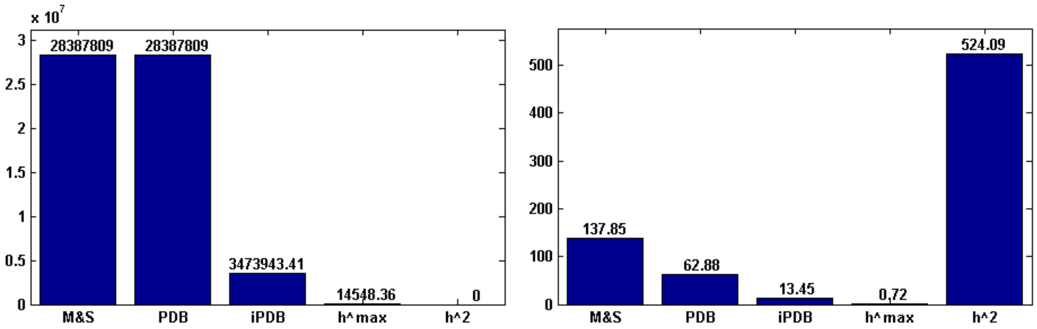


Figure 4.3: Group 'two boxes block each other', Mode 9 - average expansions (left) and average total time in seconds (right) over 50 instances each. The charts represent the lower complexity version of field size 6 × 6 with 4 boxes.

Generally, $h^{max}$ shows the best performance on group 'two boxes block each other' problems. It needs the fewest expansions, very little time and it scales a lot better than on group 'one goal cannot be reached'. M&S scales worse than it did on group 'one goal cannot be reached' problems but still better than all other heuristics. iPDB's results are very inconsistent, with some instances being solved in record time. Interestingly, the other heuristics do not show any sign of those quickly solved instances being easy. Actually some of them even seem to be harder. iPDB is also fast on the other instances, making it the fastest heuristic for modes 7 and 8. Even though mode 8 has a smaller search space than mode 7, both modes take a similar amount of time on comparable sizes for most heuristics except blind search and iPDB, which behave as expected, take longer and have more expansions on mode 7. Another difference between mode 7 and 8 is the performance of M&S and PDB. While they both need the same amount of expansions on both complexities of mode 7, PDB needs up to 10 times more expansions on the lower complexity of mode 8 but again exactly the same amount on its higher complexity.

For mode 9, the heuristics show a completely different behavior. It is extremely easily solved by $h^{max}$, which is very fast on both sizes. iPDB, PDB and M&S need much more resources so that they are even unable to solve the higher complexity problems. Blind search cannot solve any of the mode 9 problems due to its memory consumption. On lower complexity, $h^{max}$ manages to spot the unsolvability with up to 3500 times fewer expansions than PDB and M&S, which is quite remarkable (it was only up to 4 times less on modes 7 and 8). In the average, iPDB is clearly faster than PDB and M&S but its results are again extremely inconsistent, in some cases it is even slightly slower than of PDB. Again, we observe that M&S and PDB need the same amount of expansions. $h^2$ can solve 80% of the mode 9 higher complexity instances, which further confirms that this kind of problem is very easy for delete relaxation heuristics and hard for abstraction based ones.

### Group 'goal blocks player'

For mode 10, we created instances with field size 10 × 9 with 3 and 4 boxes. For mode 11, we created instances with a field size of 10 × 7 with 5 boxes. Additionally, we created 2 instances of mode 11 with 4 boxes (thus without random boxes/goals) with field sizes 10 × 7 and 10 × 8.

**Heuristics comparison** Figure 4.4 gives an overview over the results for mode 10.

Group 'goal blocks player' is more similar to the first group 'one goal cannot be reached' again, but with $h^{max}$ time wise closer to iPDB and PDB (which means better) and M&S being left behind. Even though $h^{max}$ is still slower, it needs less expansions than iPDB and PDB. iPDB is generally the fastest heuristic on problems of this group. For mode 10, M&S and PDB need the same amount of expansions again. The jump from 3 to 4 boxes in the different complexities of mode 10 is overall way smaller than the jump from 2 to 3 boxes in group 'one goal cannot be reached' problems. The two minimum-sized instances of mode 11 behave similarly to mode 10 and are both solved extremely fast by iPDB, but since it has shown inconsistent results, we cannot give any guarantee on these values. The results for the mode 11 instances with a random box and goal strongly vary depending on how close the random box and goal field are to each other. In our experiments, M&S and PDB were struggling with memory and could only solve half of the problems. Interestingly, $h^2$ managed to solve about 20% of them, even though it could not solve the two minimum-sized ones. Those it could solve were always instances that PDB could solve as well while M&S

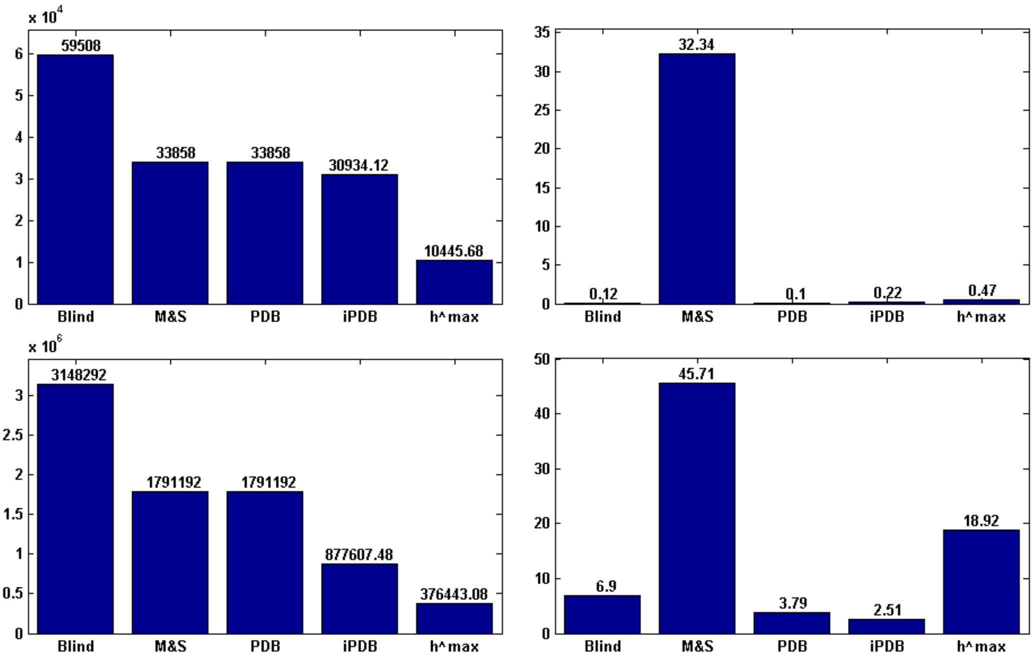


Figure 4.4: Group 'goal blocks player', Mode 10 - average expansions (left) and average total time in seconds (right) over 50 instances each. The two top charts represent instances with 3 boxes, the two bottom charts with 4 boxes. All of them are of field size 10 × 9.

could not. Those instances were the ones with the random box and goal being placed very far apart.

The time wise bad performance and good scalability of M&S on all Sokoban problems might be explained by the overhead which comes from the calculation of the abstraction. M&S wastes time if it calculates a more complex abstraction than necessary to solve the problem, especially on smaller instances. Given enough memory, M&S might actually become the fastest heuristic on large Sokoban instances.

## 4.2 NoMystery

In this section, we discuss our results for NoMystery. The first part explains how we decided on the suitable parameters for the instances, the second part shows and explains the results of our experiments in detail.

### 4.2.1 Parameter Finding

To find a suitable complexity for NoMystery, we basically tried around as we did with Sokoban. In case of NoMystery, complexity is determined by the number of nodes and packages as well as by the decision whether the two subgraphs are connected and whether the unsolvable subgraph is a clique or a star.

The calibration runs showed that instances with only one package in the unsolvable part were solved by the pre-processor because there was no fuel on the node. Since having no fuel on a node is equal to having no outgoing connections, this confirms our assumption that instances where an important connection is missing are too trivial. Instances without any packages in the solvable part were not trivial but still much easier than those with one or more packages, and it would be difficult to compare them with other instances, so we omitted them from further tests.

### 4.2.2 Results

As for Sokoban, the memory limit was set to 3072 MB and the time limit to 30 minutes for all following experiments. Our experiments combine different amounts of nodes and packages, use connected and unconnected subgraphs and the unsolvable subgraph is either a clique or a star. The number of nodes ranges from 2 to 4, the number of packages depends on the difficulty and ranges from 4 to 14 overall, distributed among the two subgraphs in different ratios.

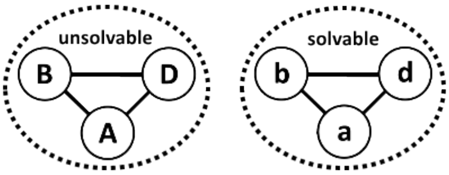
The results from blind search show that instances with equal number of nodes in both subgraphs are harder if the packages are distributed equally among both subgraphs and both subgraphs are cliques. Packages in the unsolvable subgraph have slightly more impact on the complexity. If the unsolvable subgraph is a star, it is obviously easier and therefore packages in the solvable subgraph have more impact. If one subgraph is larger than the other one, packages there have more impact regardless of whether it is a clique or a star. The results of some example setups are given in figure 4.5 and figure 4.6.

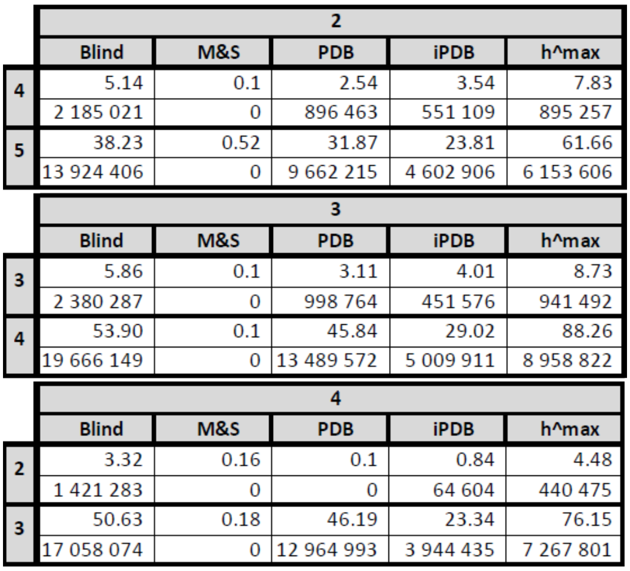
Overall PDB is the worst choice for unsolvable NoMystery problems. If the subgraphs are connected, PDB always needs the same amount of expansions as blind search and is slightly slower due to its calculation overhead. If the subgraphs are not connected and have few packages, PDB might have zero expansions and be very fast. However, with increasing number of packages, the needed number of expansions and time rise quickly, up to the point where it performs exactly as bad as with connected subgraphs. The transition where PDB starts being better than blind search is roughly around 7 packages overall.

With unconnected subgraphs, M&S needs almost always zero expansions and is much faster than the other heuristics. Only if the unsolvable subgraph becomes larger than two nodes and has enough packages, M&S needs more than zero expansions and becomes much slower. However, even on those instances, M&S still performs very well compared to the other heuristics. With connected subgraphs, M&S always needs more than zero expansions except for instances with very few packages. It is still one of the best heuristics, but similar to the cases with unconnected subgraphs, it becomes worse with larger unsolvable subgraphs.

iPDB is efficient in all situations except for very easy instances, where it might need one or two seconds to solve the problem, while others are ready almost instantly. It scales very well and is often the best heuristic when there are many packages. Thanks to its good scalability, iPDB performs especially well on connected subgraphs and is even better if the unsolvable subgraph is a clique and the solvable subgraph has many packages. On this kind of instances, it is the fastest heuristic and needs the fewest expansions.

$h^{max}$ performs badly with unconnected subgraphs. It needs fewer expansions than blind search but is up to twice as slow. With connected subgraphs, it is slightly faster than blind search and needs much fewer expansions, but it is still worse than iPDB in both aspects.





| | 2 | | | | |
|---|---|---|---|---|---|
| | Blind | M&S | PDB | iPDB | h^max |
| 4 | 5.14 | 0.1 | 2.54 | 3.54 | 7.83 |
| | 2 185 021 | 0 | 896 463 | 551 109 | 895 257 |
| 5 | 38.23 | 0.52 | 31.87 | 23.81 | 61.66 |
| | 13 924 406 | 0 | 9 662 215 | 4 602 906 | 6 153 606 |

| | 3 | | | | |
|---|---|---|---|---|---|
| | Blind | M&S | PDB | iPDB | h^max |
| 3 | 5.86 | 0.1 | 3.11 | 4.01 | 8.73 |
| | 2 380 287 | 0 | 998 764 | 451 576 | 941 492 |
| 4 | 53.90 | 0.1 | 45.84 | 29.02 | 88.26 |
| | 19 666 149 | 0 | 13 489 572 | 5 009 911 | 8 958 822 |

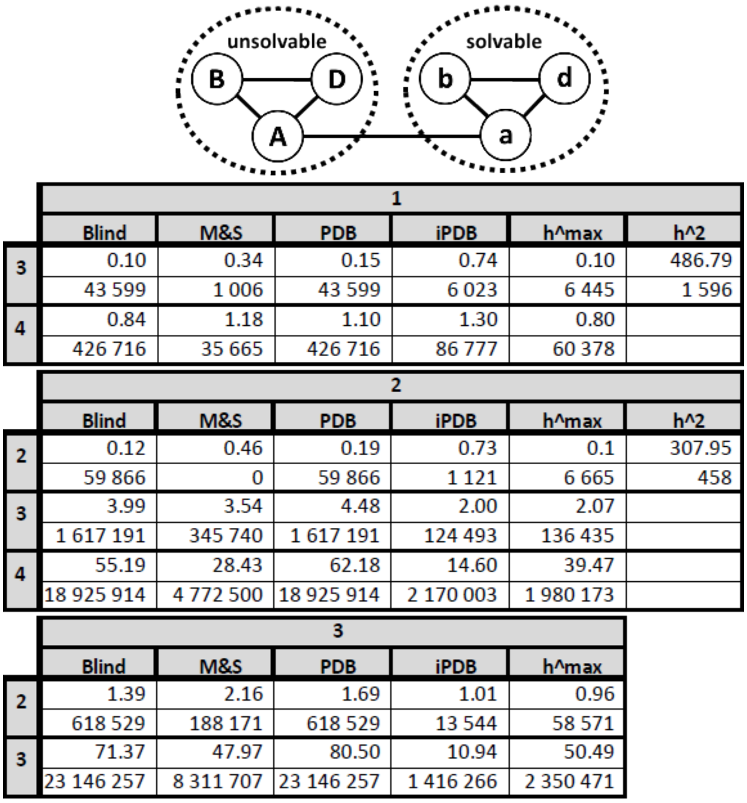
| | 4 | | | | |
|---|---|---|---|---|---|
| | Blind | M&S | PDB | iPDB | h^max |
| 2 | 3.32 | 0.16 | 0.1 | 0.84 | 4.48 |
| | 1 421 283 | 0 | 0 | 64 604 | 440 475 |
| 3 | 50.63 | 0.18 | 46.19 | 23.34 | 76.15 |
| | 17 058 074 | 0 | 12 964 993 | 3 944 435 | 7 267 801 |

Figure 4.5: Result of NoMystery problems with 3 nodes in both subgraphs. The subgraphs are not connected with each other and are both cliques. In each table, the number in the upper row represents the number of packages in the solvable subgraph, while the number in the left column represents the number of packages in the unsolvable subgraph. In each entry, the upper value is the average total time in seconds and the lower value the average amount of expansions. Both are rounded over 20 instances.

Only on easy instances with connected subgraphs and few packages in the unsolvable part, $h^{max}$ manages to be faster than all others. It is also better than M&S in terms of speed and expansions if the subgraphs are connected and there are more packages in the solvable subgraph.

The slow implementation of $h^2$ only manages to solve very easy instances. There is only one kind of instances in our experiments where $h^2$ can keep up with the other heuristics in terms of time and needs zero expansions, and those are instances with 3 nodes in each subgraph and 2 packages in the unsolvable subgraph, which additionally must be a star. If these subgraphs are connected, it is slightly slower but still very fast.

Unfortunately, we cannot compare our results with those in the paper from Bäckström at al., who originally came up with the idea for this subgraph-structure. The instance of Trucks they used had 2 nodes and packages in the unsolvable subgraph and 8 nodes and packages in the solvable subgraph, which was too large for most heuristics in their experiments. Only their own method as well as $h^2$ and $h^3$ were able to solve that instance. Interestingly, the implementation of $h^2$ they used finished in under a second. We had similar instances in our experiments with fewer nodes in the solvable subgraph but our $h^2$ did not finish in the given time limit of 30 minutes. The slow implementation aside, the reason for this might be the fact that in NoMystery, each move reduces the amount of fuel on a node. Hence, there


unsolvable
solvable
B
D
A
b
d
a

| | 1 | | | | | |
|---|---|---|---|---|---|---|
| | Blind | M&S | PDB | iPDB | h^max | h^2 |
| 3 | 0.10 | 0.34 | 0.15 | 0.74 | 0.10 | 486.79 |
| | 43 599 | 1 006 | 43 599 | 6 023 | 6 445 | 1 596 |
| 4 | 0.84 | 1.18 | 1.10 | 1.30 | 0.80 | |
| | 426 716 | 35 665 | 426 716 | 86 777 | 60 378 | |

| | 2 | | | | | |
|---|---|---|---|---|---|---|
| | Blind | M&S | PDB | iPDB | h^max | h^2 |
| 2 | 0.12 | 0.46 | 0.19 | 0.73 | 0.1 | 307.95 |
| | 59 866 | 0 | 59 866 | 1 121 | 6 665 | 458 |
| 3 | 3.99 | 3.54 | 4.48 | 2.00 | 2.07 | |
| | 1 617 191 | 345 740 | 1 617 191 | 124 493 | 136 435 | |
| 4 | 55.19 | 28.43 | 62.18 | 14.60 | 39.47 | |
| | 18 925 914 | 4 772 500 | 18 925 914 | 2 170 003 | 1 980 173 | |

| | 3 | | | | |
|---|---|---|---|---|---|
| | Blind | M&S | PDB | iPDB | h^max |
| 2 | 1.39 | 2.16 | 1.69 | 1.01 | 0.96 |
| | 618 529 | 188 171 | 618 529 | 13 544 | 58 571 |
| 3 | 71.37 | 47.97 | 80.50 | 10.94 | 50.49 |
| | 23 146 257 | 8 311 707 | 23 146 257 | 1 416 266 | 2 350 471 |

Figure 4.6: Result of NoMystery problems with 3 nodes in both subgraphs. The subgraphs are connected with each other and are both cliques. In each table, the number in the upper row represents the number of packages in the solvable subgraph, while the number in the left column represents the number of packages in the unsolvable subgraph. In each entry, the upper value is the average total time in seconds and the lower value the average amount of expansions. Both are rounded over 20 instances.

are many different states with nodes having different amounts of fuel, while the positions of vehicles and packages are the same. Trucks on the other hand does not have any resources. The states are defined by the positions of vehicles and packages only. Therefore, the state space of Trucks is smaller. Also, the structure of Trucks might suit h$^2$ better than that of NoMystery.

# 5
# Conclusion

In this thesis, we created many unsolvable problem instances for Sokoban and NoMystery. We investigated how helpful different admissible heuristics are on these instances. The general lesson is that admissible heuristics can be very helpful in some cases, but can also be worse than blind search in others. Sometimes it is a trade-off between time and memory consumption. Some heuristics might be slower than blind search but consume much less memory. The performance of different heuristics depends on the domain and on some very specific properties of the problem and its unsolvability.

Our experiments also show that using iPDB is never a bad choice and almost always better than using blind search in every aspect, at least for Sokoban and NoMystery. $h^{max}$ can be a very efficient heuristic for certain Sokoban instances. For NoMystery instances with unconnected subgraphs on the other hand, M&S is the most efficient heuristic. Nevertheless, blind search is still a good alternative. Given enough memory, it is often barely slower than the tested heuristics, especially on small instances.

Proving unsolvability in classical planning has not seen a lot of attention in research and there are certainly many open questions for future work. Sokoban is a domain with a lot of potential and it would be interesting to see some more exhaustive experiments. In our experiments we only used one configuration for each heuristic, which usually was the default one. Future experiments could try more different configurations, other heuristics or even completely new heuristics developed extra for unsolvable problem instances.

There is also the recent work *"Distance"? Who Cares? Tailoring Merge-and-Shrink Heuristics to Detect Unsolvability* by Hoffmann at al. [13], where they investigate certain M&S abstractions that aim to efficiently prove unsolvability. It would certainly be interesting to further investigate their concepts, also on our different Sokoban problems, since their evaluation did not include Sokoban at all.

# Bibliography

[1] Helmert, M. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246 (2006). URL http://www.fast-downward.org/.

[2] Bäckström, C., Jonsson, P., and Ståhlberg, S. Fast Detection of Unsolvable Planning Instances Using Local Consistency. In *SOCS*, pages 29–37 (2013).

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1 edition (1995). 37. Reprint (2009).

[4] Bonet, B. and Geffner, H. Planning as heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 129(1-2):5–33 (2001).

[5] Hoffmann, J. and Nebel, B. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302 (2001).

[6] Helmert, M. and Domshlak, C. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *ICAPS*, pages 162–169 (2009).

[7] Edelkamp, S. Planning with Pattern Databases. In *ECP*, pages 13–24 (2001).

[8] Haslum, P., Botea, A., Helmert, M., Bonet, B., and Koenig, S. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *AAAI*, pages 1007–1012 (2007).

[9] Helmert, M., Haslum, P., and Hoffmann, J. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *ICAPS*, pages 176–183 (2007).

[10] Helmert, M., Haslum, P., Hoffmann, J., and Nissim, R. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3):16:1–63 (2014).

[11] Sievers, S., Wehrle, M., and Helmert, M. Generalized Label Reduction for Merge-and-Shrink Heuristics. In *AAAI*, pages 107–115 (2014).

[12] Haslum, P. and Geffner, H. Admissible Heuristics for Optimal Planning. In *AIPS*, pages 140–149 (2000).

[13] Hoffmann, J., Kissmann, P., and Torralba, Á. "Distance"'? Who Cares? Tailoring Merge-and-Shrink Heuristics to Detect Unsolvability. In *ECAI* (2014). To appear.

# Declaration on Scientific Integrity
# Erklärung zur wissenschaftlichen Redlichkeit

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

**Author — Autor**

Dietrich Zerr

**Matriculation number — Matrikelnummer**

11-058-864

**Title of work — Titel der Arbeit**

Generating and Evaluating Unsolvable STRIPS Planning Instances for Classical Panning

**Type of work — Typ der Arbeit**

Bachelor's Thesis

**Declaration — Erklärung**

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

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

Basel, 02.08.2014

**Signature — Unterschrift**