Tailoring Pattern Databases for Unsolvable Planning Instances

Simon Ståhlberg
Department of Computer and Information Science
Linköping University, 581 83 Linköping, Sweden
simon.stahlberg@liu.se

Abstract
There has been an astounding improvement in domain-independent planning for solvable instances over the last decades and planners have become increasingly efficient at constructing plans. However, this advancement has not been matched by a similar improvement for identifying unsolvable instances. In this paper, we specialise pattern databases for dead-end detection and, thus, for detecting unsolvable instances. We propose two methods of constructing pattern collections and show that spending any more time constructing the pattern collection is likely to be unproductive. In other words, very few other pattern collections within the given space bounds are able to detect more dead-ends. We show this by carrying out a novel statistical analysis: a large computer cluster has been used to approximate the limit of pattern collections with respect to dead-end detection for many unsolvable instances, and this information is used in the analysis of the proposed methods. Consequently, further improvement must come from combining pattern databases with other techniques, such as mutexes. Furthermore, we explain why one of the proposed methods tends to find significantly more unsolvable variable projections, which is desirable since they imply that the instance is unsolvable. Finally, we compare the best proposed method with the winner and the runner up of the first unsolvability international planning competition, and show that the method is competitive.

1 Introduction
Over the past decades domain-independent planning has enjoyed a tremendous improvement and instances that earlier took an immense time to solve can now be solved very quickly. A few examples of planners are FAST FORWARD (Hoffmann and Nebel 2001), FAST DOWNWARD (Helmert 2006), and LAMA (Richter and Westphal 2010), all of which use heuristic functions to guide the search. However, most heuristic functions are not good at detecting dead-ends. Although a few heuristic functions that provide good dead-end detection, they were often not designed with that purpose in mind, e.g. the critical path heuristic $h_{cm}$ (Haslum and Geffner 2000). Detecting dead-ends is crucial for unsolvable planning instances since every state reachable from the initial state is a dead-end.

The development of heuristic functions has largely been driven by the international planning competitions (IPC). Historically, all benchmarks in IPC have been solvable, so planners participating in IPC had little use of techniques for identifying unsolvable instances. However, unsolvable instances are undoubtedly important in practice. Three examples are oversubscription planning, where all goals cannot be satisfied simultaneously and the objective is to maximise the number of satisfied goals (Smith 2004); penetration testing in computer security, where a system is considered safe if there is no solution (Boddy et al. 2005; Futuransky et al. 2010); and model checking where, given a specification, the task is to check if a model of a system meets the specification, e.g. deadlocks are impossible (Edelkamp, Leue, and Visser 2007). Model checking can be viewed as planning (Edelkamp 2003; Edelkamp, Kellershoff, and Sulewski 2010), and vice versa (Giunchiglia and Traverso 1999). In the light of the recent research on detecting unsolvability, IPC has introduced an unsolvability track.

The recent work by Bäckström et al. (2013) and Hoffmann et al. (2014) are both concerned with unsolvable instances. Bäckström et al. (2013) introduced a method for identifying unsolvable planning instances loosely based on consistency checking in constraint programming. The cornerstone of their method is variable projection. The method systematically enumerates every subset of variables and projects the instance onto it, and then checks whether the projection is unsolvable or not. If the projection is unsolvable, then the original instance must be unsolvable. Projections onto smaller subsets are faster to solve, so the method starts with every subset of size 1, then every subset of size 2, and so on. Consistency checking for planning is efficient if unsolvability can be proved for a reasonably small subset, otherwise it quickly becomes too time-consuming because of the sheer number of possible projections.

In the case when consistency checking fails (e.g. if the smallest subset that yields an unsolvable variable projection is too big), then working directly with the original instance might work. Hoffmann et al. (2014) specialised Merge & Shrink (M&S) (Helmert, Haslum, and Hoffmann 2007) to detect dead-ends and to prove unsolvability.

In this paper, we approach the problem of detecting unsolvability in a similar way but specialise pattern databases (PDBs) (Culberson and Schaeffer 1998; Edelkamp 2001;
We briefly present the SAS+ planning formalism (Bäckström 2004) to detect dead-ends. PDBs are, just as consistency checking, based on variable projection for providing heuristic guidance. Most current methods for deciding on PDBs are designed with solvable instances in mind and they perform poorly on unsolvable instances. Hence, we propose two methods of constructing pattern collections for dead-end detection (Section 4) and carry out a detailed statistical analysis (Section 5) on these. A typical statistical analysis of heuristic functions study instance coverage and run-time. One issue with this approach is that the limitation of the method is not clear. If we have a pattern collection that performs well on an instance, then we do not know if there is another pattern collection which would have performed considerably better. We address this issue by taking advantage of a large computer cluster and determine a distribution for every instance of how good projections are at dead-end detection. These distributions are then used to show that the proposed methods construct pattern collections that are amongst the best (Section 5.2). In other words, to improve dead-end detection further would require additional techniques. We show that the proposed methods differ in a significant way: one method tends to find more unsolvable projections than the other, which is important since an unsolvable projection implies that the instance is unsolvable (Section 5.3). We also show that mutexes are useful for improving how many dead-ends patterns can detect (Section 6). Finally, we run experiments and show that our best method is competitive with winner, Aidos, and the runner up, SymPA, of the first unsolvability IPC (Section 7). More precisely, the coverage of our method is comparable to SymPA, and is better than DE-PDBs, which is similar to our method and is part of Aidos. The experiments show that mutexes have a considerable effect on many instances.

2 Preliminaries

We briefly present the SAS* planning formalism (Bäckström and Nebel 1995). A SAS* instance is a tuple \( \Pi = (V, A, I, G) \) where:

- \( V = \{v_1, \ldots, v_n\} \) is the set of variables, and each variable is associated with a domain \( D_v \). A partial state is a set \( s \subseteq \bigcup_{v \in V} \{v, d\} : d \in D_v \} \) where every variable in \( V \) occurs in at most one pair. A total state is a partial state where every variable in \( V \) occurs in exactly one pair.
- \( A \) is the set of actions, and an action \( a \) has a precondition \( \text{pre}(a) \) and an effect \( \text{eff}(a) \), which are both partial states.
- \( I \) is the initial state, and is a total state.
- \( G \) is the goal, and is a partial state.

We write \( V(\Pi), A(\Pi), I(\Pi) \) and \( G(\Pi) \) for the set of variables, actions, the initial state and the goal of \( \Pi \), respectively.

The set of all total states of an instance \( \Pi \) is \( \text{StateSpace}(\Pi) \) = \( \{\{(v_1, d_1), \ldots, (v_n, d_n)\} : d_1 \in D_{v_1}, \ldots, d_n \in D_{v_n}, n = |V(\Pi)|\} \). We view partial states as partial functions (e.g. \( s[v] = d \) means that \( (v, d) \in s \)) and use the corresponding notation, in which case we use \( \mathcal{D}(s) \) to denote the domain of the partial function \( s \). We say that a partial state \( s_1 \) matches another partial state \( s_2 \) if \( s_1 \subseteq s_2 \). We define the composition of two partial states \( s_1, s_2 \) as \( s_1 \oplus s_2 = s_2 \cup \{(v, s_1[v]) : v \in \mathcal{D}(s_1) \setminus \mathcal{D}(s_2)\} \). An action \( a \) is applicable in a total state \( s \) if \( \text{pre}(a) \subseteq s \), the result of applying \( a \) in \( s \) is the total state \( s \oplus \text{eff}(a) \). Given two total states \( s_1, s_G \), a sequence of actions \( \omega = (a_1, \ldots, a_n) \) is called a plan from \( s_1 \) to \( s_G \) if and only if there exists a sequence of intermediate total states \( (s_1, \ldots, s_{n-1}) \), such that \( s_1 \) is the result of \( a_1 \) in \( s_1 \), \( s_i \) is the result of \( a_i \) in \( s_{i-1} \) for all \( 2 \leq i \leq n - 1 \), and \( s_G \) is the result of \( a_n \) in \( s_{n-1} \). A plan \( \omega \) is also a solution with respect to a goal \( G \) if \( G \subseteq s_G \). A state \( s \) is reachable with respect to an initial state \( s_I \) if there is a plan from \( s_I \) to \( s \).

The causal graph \( \text{CG}(\Pi) \) of a SAS* instance \( \Pi = (V, A, I, G) \) is the digraph \( (V, E) \) where an arc \( (v_1, v_2) \) belongs to \( E \) if and only if there exists an action \( a \in A \) such that \( v_2 \in \mathcal{D}(\text{eff}(a)) \) and \( v_1 \in \mathcal{D}(\text{pre}(a)) \cup \mathcal{D}(\text{eff}(a)) \).

The following definitions are central to this paper:

**Definition 1.** A state is a dead-end if there is no plan from \( s \) to any goal state \( g, G \subseteq s \) where \( G \) is the goal.

We define variable projection, or simply projection, in the usual way (Helmer 2004).

**Definition 2.** Let \( \Pi = (V, A, I, G) \) be a SAS* instance and let \( V' \subseteq V \). The variable projection of a partial state \( s \) onto \( V' \) is defined as \( s|_{V'} = \{(v, d) : (v, d) \in s, v \in V'\} \). The variable projection of \( \Pi \) onto \( V' \) is \( \Pi|_{V'} = (V', A|_{V'}, I|_{V'}, G|_{V'}) \), \( A|_{V'} = \{a_{V'} : a \in A\} \) where \( \text{pre}(a_{V'}) = \text{pre}(a)|_{V'} \) and \( \text{eff}(a_{V'}) = \text{eff}(a)|_{V'} \).

3 Pattern Databases & Collections

We give a brief introduction to pattern databases and present a strategy to build them. A pattern \( P \subseteq \text{StateSpace}(\Pi) \) is a subset of variables of an instance \( \Pi \), and the lowest plan cost of reaching a goal state from each state \( s \in \text{StateSpace}(\Pi|_P) \) is stored in a pattern database (PDB) \( h^P \). The database is used to efficiently provide an estimate of the real plan cost from some state to a goal state. We write \( h^P(s) \) to denote the cost associated with \( s \) in \( h^P \). A main problem with constructing a PDB is how to choose the pattern, since PDBs are often vastly different w.r.t. how informative they are (and there is an enormous amount of them).

As the name suggests, a pattern collection \( C \) consists of several patterns which, together, are used to provide a plan cost estimate: \( C(s) = \max\{h^P(s) : P \in C\} \). A popular strategy to build a pattern collection is iPDB (Haslum et al. 2007). The iPDB strategy uses hill-climbing search in the space of pattern collections, and we use a similar strategy. Hill-climbing search is an optimisation technique which attempts to maximise some score function by local search. In planning, the score function for solvable instances is often the average cost in the PDB. The search starts with a pattern of a single goal variable, and then attempts to find a better pattern by repeatedly adding a single variable to it. There might be several candidates, and the candidate whose PDB scored best is selected. This procedure is repeated until some condition is met. More precisely, let \( \Pi \) be an instance, then we use the following pattern selection strategy in this paper.

1. Let \( P = \{g\} \) where \( (g, d) \in G(\Pi) \), \( d \in D_g \) (i.e. \( g \) is a goal variable), and let \( b \) be a bound on the maximum PDB
size. We also have some function Score, where the input is a PDB and the output is a number;

2. Compute $\text{Score}(h^P, v)$ for every $v \in V(\Pi)$, where $v \notin P$, $v$ is weakly connected to some variable in $P$ in the causal graph, and $|D_v| \cdot \prod \{|D_{v'}| : v' \in P\} \leq b$.

3. If there are no candidates then return $P$; and

4. Let $P$ be the candidate with the highest score in step 2 and go to step 2. If there are several candidates with the same score then chose one arbitrarily.

The resulting pattern is added to the pattern collection, and then we start yet another search (with another initial pattern, if there is one). This is repeated until either: the collection is sufficiently large; or if the last 5 searches returned patterns that we have already seen before. The implementation tries to avoid candidates that were selected in previous searches.

4 Methods

In this section, we define the methods that we analyse in the paper. We propose two methods for constructing pattern collections. Both methods are based on hill-climbing search and the only difference is the score function. Before we define them, we give a definition of a central concept. Let $\Pi$ be a planning instance. The function $\xi(\Pi)$ generates a set of at most 20000 reachable states of $\Pi$ by breadth-first search. Note that, if $\Pi$ is unsolvable then every state of $\xi(\Pi)$ is a dead-end. The function $\xi$ is required to generate the same set every time for the same instance. Our methods are:

- $\text{HC}_{DE}(\Pi)$: A pattern collection is constructed for $\Pi$ by hill-climbing search with the scoring function:

  \[
  DE(h^P) = \frac{|\{s \in \text{StateSpace}(\Pi) : h^P(s) = \infty\}|}{|\text{StateSpace}(\Pi)|}.
  \]

  Which favours PDBs with a high ratio of dead-ends.

- $\text{HC}_{\text{Samples}}(\Pi)$: A pattern collection is constructed for $\Pi$ by hill-climbing search with a scoring function that attempts to maximise the number of different dead-end states that the pattern collection can detect. If two patterns are able to identify exactly the same states as dead-ends, then we only need one of them. The overlap of a new pattern with the collection is approximated with the help of $\xi(\Pi)$ in the following way. Let $P$ be a pattern and let $C$ be the current collection, then the scoring function is:

  \[
  \text{Samples}(h^P, C) = |\{s \in \xi(\Pi) : h^P(s) = \infty, C(s) \neq \infty\}|.
  \]

The function $\text{Samples}$ outputs the number of dead-ends that $C$ did not detect as dead-ends, but $h^P$ did detect. Note that, $\xi$ generate states close to the initial state. Detecting dead-end states close to the initial state is likely to prune away many reachable states from the search space. Hence, $\text{Samples}$ should favour patterns that can detect many, previously undetected, dead-end states of $\xi(\Pi)$. Furthermore, we resolve tiebreaks with $DE(h^P)$ (and if we still have a tiebreak then the successor is chosen arbitrarily).

5 Statistical Analysis

We present a detailed statistical analysis of how well pattern collections constructed by the methods defined in Section 4 perform on benchmarks. Roughly speaking, the analysis reveals the limit of pattern collections as a method for detecting dead-ends, and we attain this by letting a computer cluster1 generate PDBs using many years of CPU-time. The collected data lets us visualise the distribution of PDBs w.r.t. how many dead-ends they have — a crucial property for good dead-end detection. We show that the methods perform comparably but that $\text{HC}_{DE}$ tends to find more unsolvable projections, which is important since such projections prove that the instance is unsolvable. Because of this difference, we consider $\text{HC}_{DE}$ to be better than $\text{HC}_{\text{Samples}}$ on these benchmarks.

5.1 Domains

The statistical analysis is done on the benchmarks of unsolvable instances provided by Hoffman et al. (2014), which consists of 8 different domains. We give a brief explanation of the domain in each benchmark:

- 3SAT: Unsolvable formulas in the phase transition region.
- Bottleneck: $n$ agents have to move to their respective goal on a grid. However, when an agent moves to a tile it is flipped and agents cannot move to a flipped tile. Instances are unsolvable since only $n-1$ agents can reach their goal.
- Mystery and No mystery: Transportation domains there is not enough fuel to achieve the goal.
- Peg Solitaire (Pegsol): A tabletop game and the (impossible) goal is to have a single peg in the center of the board.
- Rovers: A fuel-restricted rover has to complete its mission before it runs out of energy. Normally, the rover is able to recharge but this is prevented in these instances.
- TPP: An agent is given a budget and can buy goods, drive between markets with different prices, and sell goods. The goal is to own a lot of goods, but its impossible to earn enough money for this.
- Tiles: A sliding tiles puzzle where the initial state of the puzzle is taken from an unsolvable part of the state space.

5.2 Dead-end detection

We briefly discussed a disadvantage of typical analysis methods in Section 1: the absence of a notion of optimality. For example, let $h^P$ be a PDB whose size is bounded by some integer $b$, and $s$ a dead-end state that $h^P$ cannot detect. Then there is no guarantee that there is another PDB, whose size is also bounded by $b$, that can detect $s$. An optimal pattern collection for the bound $b$ would contain a PDB for every dead-end that is detectable by a PDB whose size is bounded by $b$. Note that there might be dead-ends that an optimal pattern collection does not detect. In this section, we analyse how well the pattern collections constructed by

\footnote{The processors were 8-core Intel Xeon E5-2660 at 2.2GHz, provided by National Supercomputer Centre (NSC) at Linköping University. Website: \url{https://www.nsc.liu.se/}}
It is possible for HCDE to detect as many dead-ends as \( \phi \), whereas \( \phi \) fails for Tiles. We observe the effect of \( \mathcal{P}(\Pi) \) being incomplete in 3SAT, where HCDE and HC\text{Samples} detect many more dead-ends than \( \phi \). Due to space constraints we have not included how many dead-ends \( \phi \) detected per instance, but the experiments presented later shed some light on how successful HCDE is in practice. Typically, the frequency of dead-ends drop as the instance difficulty increase (roughly speaking, the difficulty of an instance is its state space size, or by the parameters used to generate it), i.e. good patterns become more rare. Consider this: \( \phi \) was given several weeks of CPU-time per instance to generate 20 million PDBs and construct a pattern collection, whereas HCDE and HC\text{Samples} were given mere 20 minutes and they were still able to compete with \( \phi \)!

The two outliers are Pegsol and 3SAT, but why is this? Figure 3 shows the histograms of how many dead-ends the PDBs of \( \mathcal{P} \) for 3 instances. The numbers within the parentheses denote specific instances in the benchmark. The percentage in the upper right corner is the average percent of dead-ends across all PDBs of the instance.
harder one which means that the PDBs for the easy instance are (relatively) a lot more informative than the PDBs for the hard instance. To get a more meaningful distribution for the hard instance we might have to consider larger PDBs or additional techniques (e.g. metaxes). The situation is different for Pegsol, where the distribution in Figure 3 is representative for every Pegsol instance, i.e. the PDBs are not very informative. The distributions are even worse for smaller PDBs, i.e. no PDB has any dead-end.

For Pegsol, where the distribution in Figure 3 is representative for every Pegsol instance, i.e. the PDBs are not very informative. The distributions are even worse for smaller PDBs, i.e. no PDB has any dead-end. The methods HCDE and HC\text{samples} might struggle when given these instances because their score functions count dead-ends for smaller state space sizes but the score is almost always 0. In other words, the search degrades to blind, uninform ed search and is therefore unlikely to find the few patterns which are useful for dead-end detection.

### 5.3 Unsolvable projections

If we encounter an unsolvable projection at some point during the construction of a pattern collection then we can terminate early: we know that the initial state is a dead-end and that the instance is unsolvable. Hence, unsolvable projections are very important. The proposed methods differ in this regard: HC\text{DE} found a total of 1458 unsolvable projections over all instances, whereas HC\text{samples} only found 499 (we did not terminate early, and every pattern collection contained at most 60 PDBs of unsolvable projections).

In this section, we investigate why HC\text{DE} finds more unsolvable projections than HC\text{samples}. Firstly, we explore whether the PDBs of unsolvable projections tend to have a lot of dead-ends in addition to the initial state, i.e. if such PDBs are likely to get a high score by DE. Secondly, we investigate how many "minimal" unsolvable projections there are. By minimal we mean that there is no unsolvable projection of fewer variables. Thirdly, we analyse if different unsolvable projections have many variables in common.

We now study whether there often is a correlation between unsolvability and the frequency of dead-ends. That is, for every instance of a domain we take the average percentile rank of the unsolvable projections, and then we take the minimum, the average or the maximum (of the aforementioned average) over the entire domain.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>3SAT</td>
<td>98</td>
<td>98</td>
<td>99</td>
</tr>
<tr>
<td>Bottleneck</td>
<td>49</td>
<td>67</td>
<td>93</td>
</tr>
<tr>
<td>Mystery</td>
<td>51</td>
<td>75</td>
<td>99</td>
</tr>
<tr>
<td>No mystery</td>
<td>96</td>
<td>97</td>
<td>99</td>
</tr>
<tr>
<td>Pegsol</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Rovers</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>TPP</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
</tbody>
</table>

Table 1: Minimum, average and the maximum average percentile rank of the ratio of dead-ends in the PDBs of the unsolvable projections in $\mathcal{P}(\Pi)$, for every instance $\Pi$. That is, for every instance of a domain we take the average percentile rank of the unsolvable projections, and then we take the minimum, the average or the maximum (of the aforementioned average) over the entire domain.

It is also important to consider how many variables the projections have. We define level $k$ of an instance as the set of projections with exactly $k$ variables. Furthermore, we say that a level is inconsistent if at least one projection of level $k$ is unsolvable. If the lowest inconsistent level is $k$ then a hill-climbing search need at least $k - 1$ iterations to encounter an unsolvable projection. The average frequency of unsolvable projections of 7 domains can be found in Table 2, where the frequencies are for the lowest inconsistent level $k$ and the next level. In contrast to $\xi$, we consider every projection at level $k$ (and $k + 1$). However, this was not possible for some instances (e.g. any 3SAT instance at level $k + 1$) due to the sheer number of projections. We observed that the frequencies did not vary much between instances of the same domain. Unsurprisingly, the frequencies are low at level $k$: 3SAT and Rovers had the lowest average frequency of about 0.002%; Mystery had the highest average frequency of about 9%; and the average frequency over every domain was about 1.76%. At level $k + 1$, the situation improved substantially: Rovers had the lowest frequency with about 0.005% unsolvable projections, which is almost 3 times more frequent than the previous level; The highest frequency was about 12.43%.

<table>
<thead>
<tr>
<th>Domain</th>
<th>% of unsolvable projections</th>
<th># of instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>3SAT</td>
<td>11 0.002%</td>
<td>5 0</td>
</tr>
<tr>
<td>Bottleneck</td>
<td>4 1.425%</td>
<td>20 20</td>
</tr>
<tr>
<td>Mystery</td>
<td>3.63 9.008%</td>
<td>8 8</td>
</tr>
<tr>
<td>No Mystery</td>
<td>4.72 1.646%</td>
<td>25 20</td>
</tr>
<tr>
<td>Pegsol</td>
<td>10.90 0.176%</td>
<td>21 2</td>
</tr>
<tr>
<td>Rovers</td>
<td>4 0.002%</td>
<td>3 3</td>
</tr>
<tr>
<td>TPP</td>
<td>4.2 0.039%</td>
<td>5 4</td>
</tr>
</tbody>
</table>

Table 2: The average frequency of unsolvable projections of every domain, for the lowest inconsistent level $k$, and if possible level $k + 1$. However, it was not computationally feasible to solve every projection in both levels for every instance. The last two columns detail for how many instances we were able to do this for. The Tiles domain is omitted since every projection that we solved was solvable.
Then $\Pi |^1$ the lowest inconsistent level many unsolvable neighbours an unsolvable projection has at able and we examine whether most unsolvable projections are of the lowest inconsistent level solvable projections have. The upper and lower histograms un-

Figure 4: Histograms of the ratio between number of unsolv-

able neighbours and the total number of neighbours that unsolvable projections have. The upper and lower histograms are of the lowest inconsistent level $k$ and $k+1$, respectively.

for the Mystery domain. The average frequency over every domain was 4.08% – slightly higher than 2 times the previous level. The outlier is Pegsol, whose frequency barely increased from level $k$ to level $k+1$.

We conjecture that unsolvable projections tend to have many variables in common. We define the following relationship between projections.

**Definition 3.** Let $\Pi|_{V_1}$ and $\Pi|_{V_2}$ be projections that both have weakly connected causal graphs and contain at least one goal variable each. Then $\Pi|_{V_1}$, $\Pi|_{V_2}$ are neighbours if $|V_1| = |V_2|$, $|V_1 \cap V_2| = |V_1| - 1$.

In other words, two neighbours share all but one variable and we examine whether most unsolvable projections are also neighbours. Figure 4 presents histograms over how many unsolvable neighbours an unsolvable projection has at the lowest inconsistent level $k$ and the next level ($k+1$), respectively. The averages in the histograms are significantly higher than the average frequency of unsolvable projections (Table 2) in most domains. This is interesting since it suggests that we are observing something which is typical for unsolvable projections. Due to space constraints, we have not included histograms of the percentage of unsolvable neighbours for every projection, but those histograms have a single tall bar at around 0% (mainly due to the fact that there are far fewer unsolvable projections than solvable projections). The average (normalised) percentage of unsolvable neighbours was about 7.85% and 28.17% for level $k$ and $k+1$, respectively, which are significantly higher than the corresponding averages of frequency of unsolvable projections (1.76% and 4.08%). When the percentage of unsolvable neighbours differs significantly, then we are able to reason about how likely a projection is to be solvable by looking at its neighbours: if we know that all of its neighbours are solvable, then the projection is very likely to be solvable as well. At level $k+1$ the percentage of unsolvable neighbours is even higher.

A natural follow-up question is what happens if we al-

low neighbours to differ in more than one variable. We de-

fine neighbourhood as the transitive closure of the neighbour relation on unsolvable neighbours. In other words, every projection in the neighbourhood is unsolvable. Intuitively, a neighbourhood is a set of projections that have many variables in common. Table 3 details how many neighbourhoods there are and how large they are. For every domain, except Pegsol, the number of neighbourhoods are very few and they can be quite large, especially at level $k+1$. The outlier is still Pegsol, where the histogram for level $k$ and level $k+1$ are about the same, and they have about the same average. For TPP, we note that, even though the unsolvable projection(s) at level $k$ has 0% neighbours, there is on average only one neighbourhood (of average size 1).

The difference between $HC_{DE}$ and $HC_{Samples}$ can be explained in the following way: $HC_{DE}$ will always try to select the individually best PDB, whilst $HC_{Samples}$ tries to select a PDB that works well with the current pattern collection – even if the PDB contains far fewer dead-ends. Since unsolvable projections tend to have many variables in common, it is therefore likely that their dead-end detection capabilities are similar. Hence, if the current pattern collection already contains a PDB in a neighbourhood of unsolvable projections, then $HC_{DE}$ is more likely than $HC_{Samples}$ to select a PDB in the same neighbourhood (since $HC_{Samples}$ might avoid PDBs with similar dead-end detection capabilities).

Since both methods identified about as many dead-ends and $HC_{DE}$ encountered more unsolvable projections, we consider $HC_{DE}$ to be the better (at least on this benchmark). Hence, we will only evaluate $HC_{DE}$ in Section 7.

<table>
<thead>
<tr>
<th>Domain</th>
<th># of neighbourhoods</th>
<th>Size of neighbourhoods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Level $k$</td>
<td>Level $k+1$</td>
</tr>
<tr>
<td>3SAT</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Bottleneck</td>
<td>2.35</td>
<td>1</td>
</tr>
<tr>
<td>Mystery</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>No Mystery</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pegsol</td>
<td>1474.95</td>
<td>1628</td>
</tr>
<tr>
<td>Rovers</td>
<td>2.33</td>
<td>1</td>
</tr>
<tr>
<td>TPP</td>
<td>1</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 3: The average number of neighbourhoods and their average size for every domain, at the lowest inconsistent level $k$ and at level $k$. The average number of unsolvable projections per instance can be calculated by taking the product of the number of neighbourhoods and their size.
Table 4: The sum of mutexes identified for every domain. Mutexes are identified by either \( h^2 \) or \( h^3 \). The sum does not include mutexes for instances whose goal matched a mutex.

<table>
<thead>
<tr>
<th>Domain (# instances)</th>
<th>Mutexes identified ( h^2 )</th>
<th>Mutexes identified ( h^3 )</th>
<th>Goal is mutex ( h^2 )</th>
<th>Goal is mutex ( h^3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3SAT (30)</td>
<td>42</td>
<td>38 554</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bottleneck (25)</td>
<td>51 679</td>
<td>248 340</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Mystery (9)</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>No mystery (25)</td>
<td>81 788</td>
<td>2 027 164</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Pegsol (24)</td>
<td>826</td>
<td>4 768</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Rovers (25)</td>
<td>19 431</td>
<td>234 424</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Tiles (20)</td>
<td>11 160</td>
<td>11 160</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TPP (25)</td>
<td>97 858</td>
<td>1 479 852</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 5: Histograms of the percent of dead-ends of the PDBs in \( \mathcal{P} \) for two instances. The PDBs in the histograms to the left did not use mutexes, whilst the PDBs in the histograms to the right did. The mutexes were generated by \( h^3 \). No mutex matched any goal.


6 Mutual Exclusions

A useful and well-known technique to improve the heuristic values of a PDB is to use mutexes: a mutex is a partial state which matches states that cannot be reached from the initial state. We generate mutexes with the critical path heuristic \( h^m \) (Haslum and Geffner 2000). More specifically, we let \( m = 2 \) or \( m = 3 \). In this section, we detail how many mutexes \( h^2 \) and \( h^3 \) there are and whether any of the mutexes can directly prove the instance unsolvable. If a mutex does not infer unsolvability, then it can still have a tremendous effect on increasing the number of dead-ends in a PDB.

Definition 4. A partial state \( s_m \) of a SAS* instance \( \Pi \) is a mutual exclusion, or mutex, if and only if there is no reachable state \( s \) from \( I(\Pi) \) such that \( s_m \subseteq s \).

A common definition of mutex in the literature is that there is no reachable total state which contains more than one variable-value pair of a mutex, but we use a more general definition. We get mutexes of size 2 and 3 from \( h^2 \) and \( h^3 \), respectively. Ideally, for an unsolvable instance \( \Pi \) we want to prove that some \( s_m \subseteq G(\Pi) \) is a mutex (i.e., every goal state is unreachable), but it is obviously PSPACE-hard to do so. Hence, if we use a polynomial-time algorithm to identify mutexes then the typical case is that the goal does not match any identified mutex.

To (naively) decide whether a partial state \( s_m \) is a mutex for an instance \( \Pi \), we let \( s_g \) be the goal and evaluate \( h^2 \) on \( I(\Pi) \). Table 4 details how many mutexes were found by \( h^2 \) and \( h^3 \), and how many instances were identified as unsolvable by a mutex. Every instance of the Mystery domain had a goal which matched a mutex as well as many instances of the Bottleneck domain. Mutexes from \( h^3 \) immediately identified 23 of 183 instances as unsolvable, whilst mutexes from \( h^3 \) identified 58 instances as unsolvable. For every instance, it took less than 4 seconds for \( h^2 \) to identify mutexes, and (except in a few cases) less than 5 minutes for \( h^3 \).

If a solution for a projection visits a state that matches a mutex then there is no corresponding solution for the original instance, and by avoiding such solutions we can generate PDBs with potentially more dead-ends. Figure 5 shows the impact of exploiting mutexes when generating PDBs for two instances, and the impact is a very significant increase in dead-ends: the average ratio of dead-ends for the 3SAT PDBs went from 15.7% to 84.2% and for the TPP PDBs it went from 3.6% to 88.4%. Figure 5 consists of only two instances, but the experiments in Section 7 show that many other instances benefit greatly, too, from using mutexes.

7 Experiments

In this section, we evaluate \( HC_{DE} \) on the benchmarks of unsolvable instances provided by Hoffmann et al. (2014). Our method and the consistency checking method is implemented in C#, and the other methods are part of the Fast Downward planner (Helmer 2006) which is implemented in C++. We gave every method 30 minutes and 4 GB of available memory for every instance, and they were run on an Intel i7 4790 at 3.6 GHz. The methods we compared were:

- **Blind**: Check whether a goal state is reachable without using any dead-end detection.
- **Mrg1** and **Mrg2** (Hoffmann, Kissmann, and Álvaro Torralba 2014): Two variants of the M&S heuristic that are optimised for dead-end detection.
- **CC** (Bäckström, Jonsson, and Ståhlberg 2013): An optimised version of the consistency checking method. We changed the search strategy to A* and the heuristic function is a pattern collection of every pair of variables.
- **HCBl**: A pattern collection is constructed by hill-climbing search with the scoring function:

\[
S(h^P) = \sum \left\{ h^P(s) : h^P(s) \neq \infty, s \in StateSpace(\Pi | P) \right\} \left[ StateSpace(\Pi | P) \right]
\]

The score function \( S \) is useful for solvable instances, but not so much for unsolvable. The planner used a depth-first search algorithm, and the pattern collection is used to detect dead-ends. This method serves as a baseline.

- **HCDE**: The pattern collection is constructed as described in this paper, and is using the same planner as **HCBl**. We also evaluate different PDB and pattern collection sizes, and how much PDBs benefit from mutexes generated by \( h^2 \) or \( h^3 \). These parameters are denoted as \( C_m \), where \( C \) is the pattern collection,
configuration was PDBs, which is also part of Aidos. Our best performing method was HCBL, which identified 83\% of the mutexes, while the second best method was DE-PDBs, which identified 75\% of the mutexes.

When HCDE did not have access to mutexes, then its performance was comparable to DE-PDBs. The reason why HCBL performs surprisingly well is because it stops once it finds an unsolvable projection, regardless of its score.

Table 5: Coverage results on the benchmarks. The second total is the coverage of every domain except 3SAT, Pegsol and Tiles.

<table>
<thead>
<tr>
<th>Domain (#)</th>
<th>Blind</th>
<th>M&amp;S</th>
<th>CC</th>
<th>Mrg1</th>
<th>Mrg2</th>
<th>HCBL</th>
<th>HCDE</th>
<th>SymPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCBL</td>
<td>HCDE</td>
<td>IPC contenders</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mrg1</td>
<td>Mrg2</td>
<td>- h^2</td>
<td>- h^3</td>
<td>- h^2</td>
<td>- h^3</td>
<td>- h^2</td>
<td>- h^3</td>
</tr>
<tr>
<td>3SAT (30)</td>
<td>15</td>
<td>15</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Bottleneck (25)</td>
<td>9</td>
<td>5</td>
<td>11</td>
<td>20</td>
<td>20</td>
<td>16</td>
<td>22</td>
<td>20</td>
</tr>
<tr>
<td>Mystery (9)</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Pegsol (24)</td>
<td>24</td>
<td>24</td>
<td>0</td>
<td>6</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Rovers (25)</td>
<td>0</td>
<td>18</td>
<td>3</td>
<td>11</td>
<td>11</td>
<td>14</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>Tiles (20)</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>TPP (25)</td>
<td>0</td>
<td>17</td>
<td>19</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>Total (183)</td>
<td>60</td>
<td>114</td>
<td>97</td>
<td>71</td>
<td>85</td>
<td>113</td>
<td>138</td>
<td>141</td>
</tr>
<tr>
<td>Total* (109)</td>
<td>11</td>
<td>65</td>
<td>74</td>
<td>61</td>
<td>69</td>
<td>64</td>
<td>89</td>
<td>92</td>
</tr>
</tbody>
</table>

Discussion

In this paper, we proposed and compared two methods of constructing pattern collections for dead-end detection. We showed by a statistical analysis and experiments that the methods performed very well: for most instances there is little point in spending more time to construct a better pattern collection, and the pattern collections outperformed the other methods in the experiments w.r.t. total coverage. Furthermore, since the pattern collections are often very good, we believe that further improvement is likely to come from enhancing patterns with other techniques (such as mutexes).

We showed that mutexes generated by h^3 had a tremendous impact on the number of dead-ends in PDBs, and it would be interesting to compare different mutex generation methods. Another possibility of improving PDBs is to simplify the instance such that the simplified instance is solvable if and only if the original instance is, e.g. redundant actions (Haslum and Jonsson 2000) are actions that can be removed since they are replaceable by a sequence of other actions. DE-PDBs and HCDE without mutexes perform similarly, and a difference is the order they consider patterns. DE-PDBs is not as aggressive as HCDE since it considers patterns in the same way as consistency checking, perhaps they differ significantly w.r.t. pattern collection construction time.

We showed that unsolvable projections tend to have many variables in common, and that unsolvable projections tend to have many dead-ends. Hence, another research direction is to devise a method that identifies those variables, and use the method to guide the hill climbing search. This would be useful when the score function for e.g. HCDE fails to provide a meaningful score, which is often the case for small PDBs with no dead-ends. However, we cannot expect a significant improvement since the proposed methods are already constructing good pattern collections.

It is worth nothing that solvable instances might also contain dead-ends and therefore the proposed methods might be useful for such instances, e.g. Sokoban (Pereira, Ritt, and Buriol 2014). Pattern collections can easily be tailored toward both heuristic guidance and dead-end detection; it is simply a matter of selecting different PDBs for either purpose. However, the cost is increased pattern collection construction time or increased memory usage. The latter might be addressed by compressing the PDBs (Felner et al. 2007).
Acknowledgements

Simon Ståhlberg is partially supported by the National Graduate School in Computer Science (CUGS), Sweden. The National Supercomputer Centre (NSC) at Linköping University provided access to a computer cluster, which was vital for this work. The anonymous reviewers provided very valuable and important feedback.

References


