Automated Creation of Pattern Database Search Heuristics

Stefan Edelkamp

Computer Science Department
University of Dortmund
stefan.edelkamp@cs.uni-dortmund.de

Abstract. Pattern databases are dictionaries for heuristic estimates storing state-to-goal distances in state space abstractions. Their effectiveness is sensitive to the selection of the underlying patterns. Especially for multiple and additive pattern databases, the manual selection of patterns that leads to good exploration results is involved.

For automating the selection process, greedy bin-packing has been suggested. This paper proposes genetic algorithms to optimize its output. Patterns are encoded as binary strings and optimized using an objective function that predicts the heuristic search tree size based on the distribution of heuristic values in abstract space.

To reduce the memory requirements we construct the pattern databases symbolically. Experiments in heuristic search planning indicate that the total search efforts can be reduced significantly.

1 Introduction

The ultimate goal for an efficient exploration is the automated creation of (admissible) search heuristics. By applying state space abstractions, heuristic estimates correspond to solutions in simplified problem spaces. The underlying problem graph is *abstracted*, e.g., in a form that nodes are contracted or new edges are inserted. Such abstractions are *homomorphisms*, i.e., for each path in the concrete state space there is a corresponding path in the abstract one. This notion of abstraction matches the one used in verification for abstraction refinement [4] and predicate abstraction [1,45].

Gaschnig [17] proposed that the cost of solutions can be computed by exact solution in abstract space. He observed that search with abstract information can be more time-consuming than with breadth-first search. Voltorta [49] has proven this conjecture, showing that heuristic search algorithms that explore state space abstractions for each encountered state from scratch (and that do not memorize abstract states) cannot possibly be better than blind search [20]. Absolver [40] was the first system to break the barrier imposed by his theorems. In order to reduce the number of revisits in abstract state one either has to memorize abstract state information on-the-fly or precompute it for the entire search space. Pattern databases [5] correspond to complete scans of the (inverted) abstract space *before* applying the search algorithm in the concrete space. A

S. Edelkamp and A. Lomuscio (Eds.): MoChart IV, LNAI 4428, pp. 35–50, 2007.

mixed strategy (between memorizing and precomputing) is considered in [30] and revisited in [27].

The success story of searching with pattern databases is long, starting with first optimal solutions in Rubik's Cube [35] and large savings in sliding-tile puzzles [5,36]. Applied for the multiple sequence alignment problem, pattern databases correspond to lookup tables for alignments of a smaller number of sequences [38,52]. In finding the best parse of a sentence [33], a pattern database entry correlates to the cost of completing a partial parse; the abstraction is derived by simplifying the grammar. TSP with asymmetric costs and precedence constraints has been analyzed by [24] using pattern database techniques. Pattern database heuristics have been applied for co-operative planning in computer games [47], where many agents search for individual paths but are allowed to help each other to succeed. First successful applications of pattern databases for verification are due to [11] (explicit-state model checking), [44] (symbolic model checking), and [39] (real-time model checking). In all approaches, even though the construction is automated, patterns were provided manually, such that, in essence, pattern selection remains a domain-dependent feature.

As they operate at the limit of main memory, a compact and space-efficient representation of pattern databases is essential. This paper exploits a space-efficient representation of pattern databases based on BDDs [3], which – by sharing binary state vectors – can lead to large memory savings. Instead of transforming an already constructed database, we apply a construction process that is throughout symbolic. Nonetheless, the main objective of this paper is to address the problem of automated pattern selection for an improved search. We embed our approach in domain-independent action planning, where automated pattern selection is mandatory. In this research area, the use of multiple [29] (often disjoint [36]) databases is frequent. So far, only greedy bin packing algorithms have been applied that terminates with the first established pattern set [7,21].

The number of possible patterns for selection is large. In case of state space abstractions that include don't cares in the state vector, the complexity is exponential in the number of remaining vector entries. In case of general relaxations of the state vector (e.g. by data abstraction, mapping variable domains to smaller sets) the number of possible choices almost becomes intractable. So even for the choice of a single pattern, we are facing a hard combinatorial optimization problem. If multiple abstractions are used, the number of choices is even worse.

In order to predict their pruning[1] effect, pattern databases have to be constructed. Unfortunately, the efforts for constructing pattern databases are high, as their sizes (measured in the number of abstract states) are large.

Especially in combinatorial problems with large state spaces and unknown structures, optimization algorithms adapted from nature – such as evolutionary

[1] Pattern database heuristics do not *prune* the exploration in the strong sense in that they eliminate transitions from the state space. If no error/goal is present, then there is no search reduction. On the other hand, if there is, then the enforced order of expansion can save many states to be looked at.

strategies or swarm optimization techniques [6] – are recommended. Given the discrete structure of the pattern selection problem, we have chosen genetic algorithms [26], which are widely used and adapt nicely: the proposed encoding of patterns into chromosomes is accessible for a human, and we can expect to obtain insights to important *schemas* for pattern selection. There are already some reports on attempts to unify planning and evolutionary computing [2,18,51,41,48], but all are concerned about plan finding (or plan refinement) and none of them addresses the problem of the automated creation of search heuristics.

The paper is structured as follows. First we review pattern databases as applied in optimal heuristic search planning. Then we turn to genetic algorithms for the automated inference of the patterns. Starting with the encoding of patterns into chromosomes, we present the design of genetic operators for the pattern selection problem. Experiments report on improving the mean heuristic value and on reducing the resulting search efforts for a selection of challenging planning domains. Finally, we draw conclusions and indicate further research avenues.

2 Pattern Databases in Planning

Action planning refers to a world description in logic[2]. A number of atomic propositions, *atoms* for short, describe what can be true or false in each state of the world. By applying operations in a world, we arrive at another world where different atoms might be true or false. Usually, only few atoms are affected by an operator, and most of them remain the same.

Let AP be the set of atoms. A planning problem (in STRIPS notation) [16] is a finite state space problem $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$, where $\mathcal{S} \subseteq 2^{AP}$ is the set of states, $\mathcal{I} \in \mathcal{S}$ is the initial state, $\mathcal{G} \subseteq \mathcal{S}$ is the set of goal states, and $\mathcal{O}$ is the set of operators that transform states into their successors. We often have that $\mathcal{G}$ is described by a simple list of atoms. Operators $O \in \mathcal{O}$ have preconditions $pre(O)$, and effects $(add(O), del(O))$, where $pre(O) \subseteq AP$ is the *precondition list* of O, $add(O) \subseteq AP$ is its *add list* and $del(O) \subseteq AP$ is its *delete list*. Given a state S with $pre(O) \subseteq S$, its successor $S' = O(S)$ is defined as $S' = (S \setminus del(O)) \cup add(O)$.

2.1 Admissible Heuristics in Planning

Admissible heuristics for planning underestimate the shortest path distance of the current state to the goal. They are important to guarantee optimality in heuristic search algorithms like A* and IDA*. The *max-atom* heuristic [22] is an approximation of the optimal cost for solving a relaxed problem in which the delete lists are ignored. Its extension *max-pair* improves the information without loosing admissibility, approximating the cost of atom pairs. The heuristic h^+ [25] is another extension to *max-atom* defined as the length of the shortest plan that

[2] For the sake of brevity, the presentation of the paper restricts to propositional planning. However, the design of planning pattern databases is applicable to complex planning formalisms too.

solves the relaxed problem with ignored delete lists. The heuristic is admissible, but solving relaxed plans is computationally hard.[3]

2.2 Explicit-State Planning Pattern Databases

Explicit-state planning pattern databases as proposed by [7,21] refer to state space abstraction, where some atoms are omitted from the problem description.

The basic idea for computing a heuristic with pattern databases is to analyze the abstraction of the concrete state space prior to the search [5]. In this abstract state space, a (complete) backward exploration (starting with the abstract goal) computes accurate goal distances and stores them in a lookup table[4]. This then information guides the concrete search process. More formally, the *abstraction* [34] of a planning problem $\mathcal{P} = <\mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G}>$ wrt. a set of atoms R is defined as $\mathcal{P}|_R = <\mathcal{S}|_R, \mathcal{O}|_R, \mathcal{I}|_R, \mathcal{G}|_R>$ with $\mathcal{S}|_R = \{S \cap R \mid S \in \mathcal{S}\}$, $\mathcal{G}|_R = \{G \cap R \mid G \in \mathcal{G}\}$, and $\mathcal{O}|_R = \{O|_R \mid O \in \mathcal{O}\}$, where $O|_R$ for $O \in \mathcal{O}$ is given as $(\mathit{pre}(O) \cap R, \mathit{add}(O) \cap R, \mathit{del}(O) \cap R)$. As the goal distance in abstract state space drop by not more than 1, pattern databases are consistent, leading to monotone cost functions in A* [42]. The principle of abstracting atoms has been extended to (automatically inferred and mutually exclusive) *atom groups* [7]. The approach reflects a multi-variate (finite domain) variable encoding of a state [23]. As an example, in Blocksworld the variable $on(X, a)$ (where X represents any available block a, b, c, or d), encodes the atoms $on(d, a)$, $on(c, a)$, $on(b, a)$ and $on(a, a)$. As only one block can lay on top of a, all atoms in a state variable (group) are mutually exclusive. Variables are distributed into *patterns*, where each pattern corresponds to an abstraction of the state space: abstract states are assignments of atoms to state variables of the chosen pattern. Using this approach each concrete state is mapped to an abstract state. As seen above the projection extends to operators, intersecting the precondition, add and delete list with the pattern. The selection of patterns that lead to the best search reduction is computationally hard and critically influences the quality of the estimate [21].

For multiple pattern databases [29], in each abstraction i, $i \in \{1, \ldots, k\}$, and for each state S we compute estimated costs $h_i(S)$. The maximum $h_m(S) = \max_{i=1}^{k} h_i(S)$ is a consistent estimate, the cumulation $h_a(S) = \sum_{i=1}^{k} h_i(S)$, however, is not necessarily admissible, since in general we cannot expect that each operator contributes to only one pattern database abstraction. In case an admissible heuristic is obtained by adding the values, we call the databases *disjoint* [36]. In order to resolve the admissibility problem in general, we have to grant that each operator has zero costs for all but one pattern databases. This induces that the backward in abstract space operates on a weighted problem graph. For this particular single-source shortest-paths problem, we adapt BFS. In each BFS level, each zero-cost operator is fired until a fixpoint is reached.

[3] The heuristic h^+ can, however, efficiently be approximated by the number of operators in a parallel plan that solves the relaxed problem. The applied approximation sacrifices the admissibility of the estimate making it inadequate for optimal planning.

[4] As inverse operator application is not always immediate, it is possible to apply backward search to the inverse of the state space graph generated in forward search [11].

2.3 Symbolic Planning Pattern Databases

The main limitation for applying pattern databases in practice is the restricted amount of (main) memory. Many strategies for leveraging the problem have been proposed. *Symmetries* allow reusing pattern databases [5], while lookups in *dual* pattern databases additionally apply to permutation problems [15]. Compressed pattern databases [14] approximate abstract states-to-goal distances. Given an upper bound on the optimal goal distance in the concrete state space, pattern database construction can be pruned [52]. *On-demand* pattern databases [13] suspend and resume a backward A* exploration of the abstract space.

A space-saving alternative for pattern databases that allow sharing of the state vector is the *trie* data structure. In a trie, each path from the root to a leaf corresponds to a scan of the state vector. Tries are commonly used in pattern databases for the multiple sequence alignment problem [46]. States with same prefixes share their representations. Tries can be *multi-variate* (each branch corresponds to a state vector entry of finite domain) or *binary* (each path corresponds to the binary representation of the state vector). Tries can be contracted by merging edges.

The main advantage of using BDDs [3] is an efficient and unique representation for sets of states. Intuitively, BDDs are binary tries in which further reduction rules have been applied to obtain a directed and acyclic graph structure. More precisely, a BDD represents the *characteristic function* of a set of states, which evaluates to 1 if and only if the binary state vector is a member of that set. The characteristic function is identified with the set itself.

Transitions are also formalized as relations, i.e., as sets of tuples of predecessor and successor states, or, more precisely, as the characteristic function of such sets. This allows to compute the image in form of a relational product. It conjoins the state set (formula) with the transition relation (formula) and quantifies the predecessor variable. This way, all states are determined, that can be reached by applying one action to a state in the input set. Iterating the process starting with the characteristic function of the initial state yields a symbolic implementation of BFS. The application of A* with BDDs has been initially proposed by [12], extensions are found in [31] (branching partitioning), [43] (weak heuristics), and [19] (ADDs). All implementations rely on small edge weights.

Symbolic pattern databases [8] are pattern databases that have been constructed symbolically for a latter lookup in either symbolic or explicit-state heuristic search. The bi-directional definition of the transition relation allows to change the search direction by quantifying the posterior variables set in the relational product. Each state set (in a breadth-first/shortest paths layer) is efficiently represented by a corresponding characteristic function. Different to the compression of the state set by *compiling* the outcome of an explicit-state pattern database, the symbolic construction operates on the compressed representation of the state and the action sets. External symbolic planning pattern databases [53] are symbolic planning databases that additionally exploit secondary storage devices such as hard disks to lessen the RAM load during construction and search.

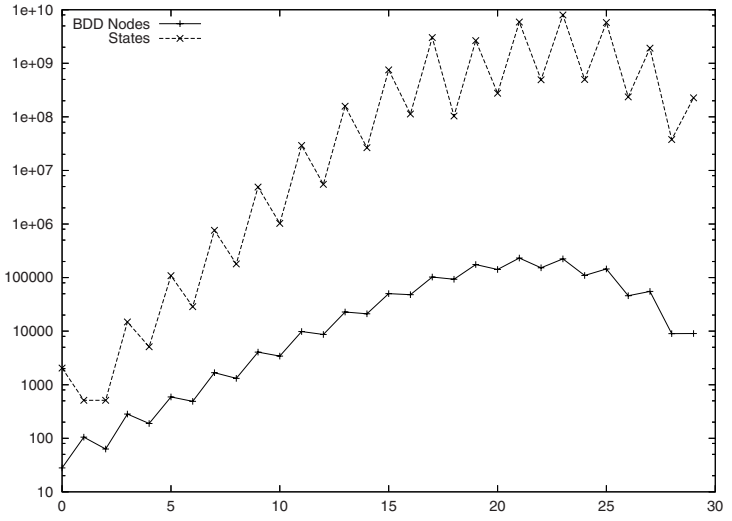

Fig. 1. Effect of symbolic pattern database construction

A better memory performance often favors symbolic to explicit-state pattern database construction: Fig. 1 displays an example of a typical Blocksworld problem instance. The number of BDD nodes is by far smaller than the number of represented states. Moreover, BDD nodes are small,[5] while the explicit-state sizes grow with size the problem instance. The mean heuristic value (for both cases) is 22.16. Besides memory savings the key performance of symbolic pattern databases for the purpose of this paper are time savings. For example, the above pattern database contains 27.22 billion entries generated in 109 seconds. Compared to explicit-state search, this corresponds to about 250 million expanded states per second.

3 Automated Pattern Selection

We have indicated that finding good patterns is involved, as there are many possible choices, especially if multiple pattern databases have to be considered. Manual pattern selection is tedious and implies that the planning process inevitably becomes problem-dependent. So far, automated pattern selection is a rather unresolved challenge, even if some recent progress has been made.

For explicit-state construction of multiple pattern databases, one has simplified the problem of finding a suitable partition to the *bin-packing* problem [7,21]. The general idea is to distribute state variables into bins in such a way that a minimal number of patterns is used; a state variable is added to an already existing bin, until the (expected) abstract state space size exceeds main memory.

[5] BDD nodes frequently consume a small number of bytes for encoding the level, the 0- and 1-successor and some auxiliary information like a hash value and markings.

```
Procedure GA
    t ← 0
    P^(t) ← Initialize
    Evaluate(P^(t))
    while (κ(P^(t)))
        P'^(t) ← Recombination(P^(t))
        P''^(t) ← Mutation(P'^(t))
        Evaluate(P''^(t))
        P^(t+1) ← Selection(P''^(t))
        t ← t + 1
```

Fig. 2. Standard genetic algorithm

Adding a variable to the pattern corresponds to a multiplication of its domain size to the (already computed) abstract state size (if possible). As a result, the bin-packing variant needed for automated pattern selection is based on multiplying variable domain sizes (rather than adding). Bin-packing is NP-complete, but efficient approximations like the first- or best-fit have been used.

For the implementation of automated pattern selection we adapt a genetic algorithm (GA) [26]. A generic implementation using the evolutionary operations for evaluation, recombination, mutation, and selection is shown in Fig. 2, where κ is the termination criterion, and t is the current iteration.

Representation. Patterns are represented as binary chromosomes of size $p \times n$, where n is the number of atoms (groups) and $p \leq n$ is the number of active patterns. In the columns, state variables are indexed, while in the rows patterns are selected. Therefore, a chromosome represents the distribution of state variables into multiple pattern databases. Fig. 3 illustrates an example: in the first pattern the groups 1, 5, 6, 8 and n are included, whereas in the second pattern the groups 3, 5 and 7 are present.

Chromosomes are *valid*[6] if all patterns respect the memory threshold M (the bin size). Formally speaking, if v_i denotes the set of atoms in state variable i, $i \in \{1, \ldots, n\}$, and $c_{i,j}$ is a bit indicating whether or not variable v_i is selected in pattern p_j, $j \in \{1, \ldots, p\}$, then for all j we have $\prod_{1 \leq i \leq n} c_{i,j} \cdot |v_i| \leq M$. (We assume that at least one variable is selected in each pattern, i.e., for all j we have $\sum_{i=1}^{n} c_{i,j} > 0$). For generating disjoint pattern databases, we additionally impose the condition that in each column there is exactly one 1, i.e., for all i we have $\sum_{j=1}^{p} c_{i,j} = 1$. Since the columns > 3 have more than one bit set, the chromosome in Fig. 3 is not disjoint.[7]

[6] Invalid chromosomes are assigned to a bad fitness value and discarded by Darwin's evolutionary rule for the *survival of the fittest*.

[7] As planning operators can modify more than one state variable at a time, in difference to the set of well-studied $(n^2 - 1)$-puzzle pattern databases [36] this condition is only a necessary but not a sufficient condition for disjointness. As checking disjointness based on the pattern selection may be involved, for each operator we assign cost 1 to only one abstraction.

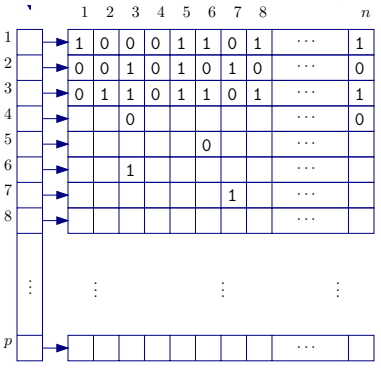

Fig. 3. Bitvector representation of a chromosome

Initialization. For the initialization phase we could generate random chromosomes, but we found that the amount of work to find an acceptable partitioning by performing a randomized assignment of the chromosomes is by far larger as with prior bin packing with no significant advantage within the overall search process. Therefore, we initialize the chromosomes with bin packing. To avoid all chromosomes of the initial population to be identical, we choose a random permutation to the atom groups prior to their automated partitioning into patterns. This leads to comparably good but different distributions of groups into patterns and a feasible initial population for the genetic algorithm.

Recombination. The motivation of *recombination* of two parent chromosomes is the hope, that the good properties of the one parent combines well with the good properties of the other. One of the simplest techniques that we looked at is *crossover*: the parent chromosomes exchange parts of their patterns. If the two parents have a different number of patterns, so do the two children.

Mutation. For *mutation* chromosome bits are flipped with a small probability. This corresponds to extending or reducing the corresponding abstract space. For our case, we had to allow the addition or deletion of entire patterns. In the bin packing analogy of multiple pattern partition, adding a pattern corresponds to opening a bin, and deleting a pattern corresponds to closing a bin.

Selection. During *selection* an enlarged population (as produced by either recombination or mutation) is truncated to its original size based on the fitness value(s). The normalized fitness evaluation for the population is interpreted as a distribution function, which governs the selection process for the next population. Chromosomes with a better fitness are chosen with higher probability.

Objective Functions. The objective function plays a central role in a genetic algorithm. It defines a fitness to determine the evolutionary strength of chromosomes. The construction of a meaningful objective function is often difficult, like in our case, where the conditions for *good* patterns are hardly accessible.

A fundamental question concerns the relationship between the contents of a pattern database, and the number of nodes expanded when the heuristic is used to guide the search. Korf [35] gives first insights in such performance predictions of pattern databases: he characterizes the effectiveness of a heuristic h by its expected value $\overline{h}$ (the mean) over the problem space. The main line of reasoning is the following: if the heuristic value of every state was equal to its expected value $\overline{h}$, then a search to depth d would be equivalent to searching to depth $d - \overline{h}$ without a heuristic, since the priority for every state would be its depth plus $\overline{h}$. This means that in most search spaces, a linear gain in $\overline{h}$ corresponds to an exponential gain in the search.

For the pattern selection problem we conclude that the higher the average heuristic value, the better the corresponding pattern database. As a consequence, we compute the mean heuristic value for each database. For one pattern database PDB we compute

$$\overline{h} = \sum_{h=0}^{\max} \frac{h \cdot \ |\{u \in PDB \mid h^*(u) = h\}|}{|PDB|},$$

where $h^*(u)$ is the accurate abstract goal distance stored for the abstract state u, and where the size of a pattern database (layer) is determined by counting the number of accepting paths in the BDD.[8] For multiple pattern databases, we have k distributions. As an example, the distributions of the heuristic estimates for three pattern databases are shown in Fig. 4 in the form of histograms. For computing the evolutionary strength for an entire chromosome we compute the mean heuristic value for each of the databases individually and cumulate (or maximize) the outcome. More formally, if PDB_i is the i-th pattern database, $i \in \{1, \ldots, k\}$, then the additive *fitness* of a chromosome is computed as

$$\mathit{fitness}(c) = \sum_{i=1}^{k} \sum_{h_i=0}^{\max_i} \frac{h_i \cdot \ |\{u \in PDB_i \mid h_i^*(u) = h_i\}|}{|PDB_i|}.$$

Using the mean heuristic estimate is not the only choice. We have also experimented with a derivate not based on the number of states that share the same heuristic value, but on the number of BDD nodes to represent them. Unfortunately, the results were consistently weaker.

A Note on Search Tree Prediction. Applying the mean heuristic value for the fitness extends to the formula for search tree prediction [37]. It approximates $E(N, c, P)$, the expected number of nodes expanded by IDA* up to depth c, given a problem-space tree with N_i nodes of cost i, and a heuristic characterized by the equilibrium distribution P. The formula denotes that in the limit of large c, we expect $E(N, c, P) = \sum_{i=0}^{c} N_i P(c - i)$ nodes to be generated. It has already been used for the analysis of pattern database performance [28] and to explain anomalies that many small pattern databases are often more effective than few

[8] There are linear time algorithms for performing model counting [3].

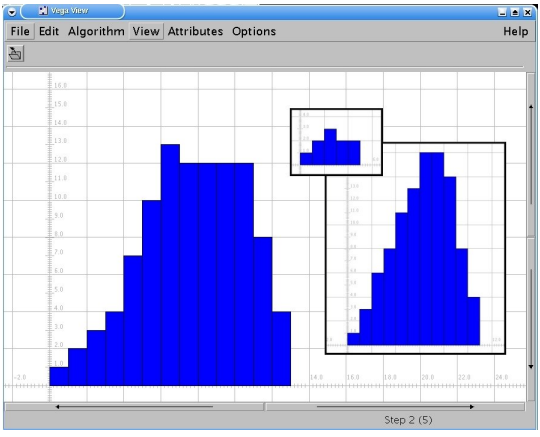

Fig. 4. Histograms of heuristic values

big ones of same total size [29]. However, the growth of search trees for general problems as addressed in action planning is not immediate [32]. Hence, we prefer $\overline{h}$ for the fitness evaluation.

4 Experiments

For implementing genetic algorithms we adapt the library GAlib[9] [50] to our hybrid (explicit-state and symbolic) planner MIPS [9]. One advantage is that it is portable to different operating systems. Another gain is that there is already a 2D chromosome data structure that satisfies our requirement. It was sufficient to provide an objective function. A fitness function could be derived automatically, using the fitness scaling approach. As a consequence, only the genetic operations for recombination mutation and selection are to be configured. These configurations have been implemented without modifying the existing source code for the standard genetic algorithm.

After some initial testing[10] we turned off recombination completely. This actually simplifies the genetic algorithm to a randomized local search strategy. As indicated above, the mutation operator adds and deletes groups to an existing pattern and allows to extend patterns in a disjoint partition. For the automated construction of both explicit-state or symbolic pattern databases, the maximum size of the abstract state spaces is taken as an additional parameter – actually the only information that has to be provided manually.

4.1 Explicit-State Pattern Databases

In a first test suite, we studied explicit-state pattern databases constructed with greedy bin packing and optimized genetic algorithms (with different parameters).

[9] `http://lancet.mit.edu/galib-2.4/`

[10] We conducted all experiments on a 3 GHz Linux PC. Time in CPU seconds was limited to 1,800; space was limited to 1 GB.

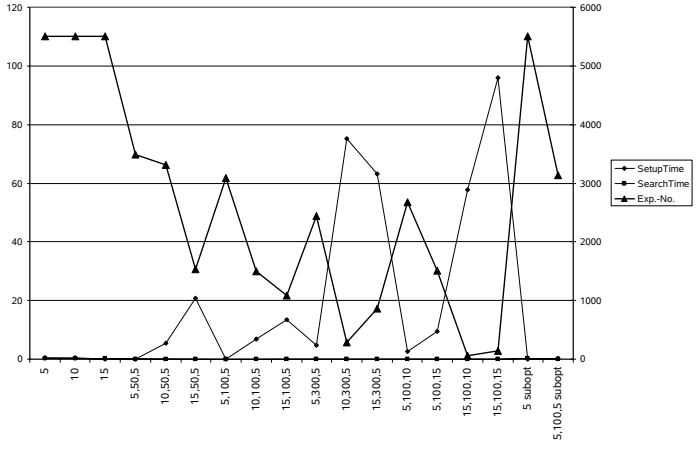

Fig. 5. Explicit pattern databases in Logistics

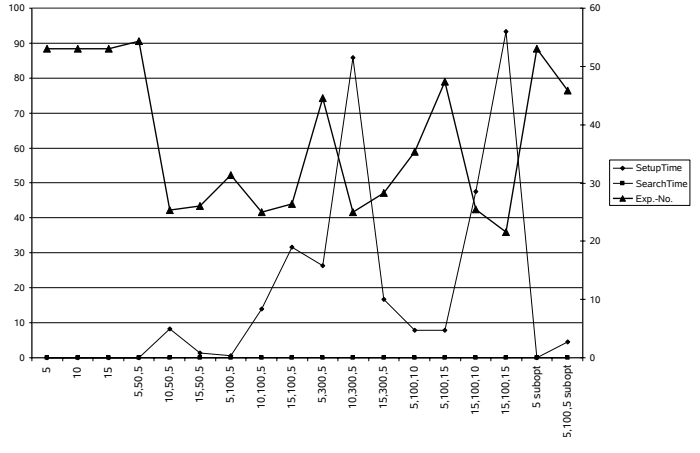

Fig. 6. Explicit pattern databases in DriverLog

The aim was to learn about the parameter setting of genetic algorithms in planning domains, as they are known to be sensitive to parameter selection.

We use several runs applying the mean of the heuristic value as a comparison guideline. As the iterated tests were involved, we depict the change in the exploration efficiencies for some interesting domains only. The horizontal axes denote the choice of parameters as follows.

Label	Meaning
5	Bin Packing with a memory threshold of 2^5 abstract states
5,50,5	GA with a memory threshold of 2^5 abstract states, 50 epocs, 5 genes

In Fig. 5 the setup time, the search time (in seconds, left label) and the number of expansions (right label) for different genetic parameters settings in the *Logistics*

domain are shown. As we can see, even when including the entire construction time for all pattern databases, with genetic algoirthms there is some substantial gain compared to ordinary bin packing. As a general finding we observe that for a better guidedance, longer construction time is needed. In some cases, however, the construction time is so large that there is no gain in exploration. In the *DriverLog* domain (Fig. 6) the influence is present, but less apparent.

Automatically selecting the parameters of the genetic algorithm in such a way that the pre-computation efforts are acceptable with number of expansions that is small enough remained a challenge. In the next set of experiments, we try to scale-up the approach by using BDDs.

4.2 Symbolic Pattern Databases

For experimenting with symbolic pattern databases, we choose various problems from international planning competition (IPC-2, IPC-3, IPC-5).[11] For the construction of symbolic pattern databases we choose a population size of 5, and a number of 20 epocs (resulting in at most 100 pattern databases to be constructed and evaluated; some of them were eliminated due to size and state variable constraints). The random seed was fixed for all experiments.

The initial population the genetic algorithm is computed as follows. We first randomize the order of variables in the state vector. Next, we apply the bin-packing strategy. [12] The search algorithm we applied is symbolic A* search with full duplicate elimination. We have added the heuristic estimates.

In Table 1 symbolic exploration results for comparing greedy bin-packing with genetic pattern selection in the benchmark problems are shown. Headings read as follows: 2^l is the abstract state space size limit; the searching time t_s is compared to the total running time [13] In other words, the setup time $t - t_s$ for the genetic construction covers the time for computing *all* pattern databases during the optimization process. The additional time for pattern optimization contributes to the gain in the quality of the heuristic estimate, measured in $\overline{h}$, the mean heuristic estimate of the first (greedy bin-packing) or best surviving (genetic algorithm) pattern.

As the Logistics and DriverLog domains were less complex in the explicit than in the symbolic case, we have recognize that the application of symbolic pattern databases pays off (cf. Fig. 1). We also observe that pattern optimization generally leads to much better mean heuristic values and to smaller search times. When scaling the problems the savings in search dominate the additional workload during construction and take over to the total search time.

[11] For domains from IPC-4 good exploration results of symbolic pattern databases are already known [10].

[12] As there is no unshuffled bin packing result, invoking search without optimization can produce better results than with optimization.

[13] Total time includes the parsing efforts of the planning problem and pattern database construction. Time for grounding the domain is not counted as we apply an individual but same program to both strategies.

Table 1. Symbolic A* search with and without genetic optimized pattern databases

		Greedy Bin Packing					GA-Optimization				
problem	2^l	length	images	$\overline{h}$	t_s	t	length	images	$\overline{h}$	t_s	t
logistics-4-1	10	**19**	63	**9.28**	0.39	**0.77**	**19**	63	8.57	**0.37**	0.79
logistics-6-1	20	**14**	42	**21.9**	0.39	**0.77**	**14**	39	20.34	**0.30**	1.01
logistics-8-1	20	**44**	166	26.32	11.98	19.92	**44**	44	**29.51**	**5.7**	**1.42**
logistics-10-1	30	-	-	-	-	-	**42**	351	**38.82**	**33.78**	**85.69**
logistics-12-1	30	-	-	-	-	-	**69**	826	**51.49**	**138.02**	**498.76**
blocks-9-0	30	**30**	79	8.86	0.47	52.51	**30**	358	**20.03**	**8.89**	**19.4**
blocks-10-0	40	-	-	-	-	-	**34**	692	**25.15**	**8.53**	**34.94**
blocks-11-0	40	-	-	-	-	-	**32**	1,219	**24.20**	**49.30**	**58.44**
blocks-12-0	40	-	-	-	-	-	**34**	942	**25.36**	**101.95**	**104.55**
zeno-2	10	**6**	6	**6.37**	**0.17**	**0.55**	**6**	14	3.83	0.19	0.57
zeno-4	10	**8**	19	**5.74**	0.27	0.73	**8**	14	3.83	**0.19**	**0.57**
zeno-6	20	**11**	24	6.6	0.64	**1.21**	**11**	14	**8.58**	**0.58**	1.51
zeno-10	25	-	-	-	-	-	**22**	23	**15.7**	**43.12**	**190.56**
zeno-11	25	-	-	-	-	-	**14**	37	**15.06**	**15.11**	**833.16**
driverlog-9	25	**22**	109	12.9	86.76	87.59	**22**	107	**15.3**	**52.46**	**72.25**
driverlog-11	25	-	-	-	-	-	**19**	110	**10.67**	**34.48**	**44.60**
driverlog-13	35	-	-	-	-	-	**26**	143	**13.7**	**553.01**	**778.03**
openstack-1	20	**23**	96	3.71	**0.51**	**1.29**	**23**	116	**5.06**	0.60	3.04
openstack-3	20	**23**	96	3.71	**0.51**	**1.29**	**23**	110	**5.40**	0.60	3.02
openstack-6	30	**20**	65	4.99	70.38	**96.17**	**20**	39	**5.44**	**52.42**	216.19
openstack-7	30	-	-	-	-	-	**20**	38	**6.88**	**31.05**	**484.19**
pipesworld-2	20	**12**	91	6.53	0.77	**2.82**	**12**	82	**7.01**	**0.23**	4.72
pipesworld-4	20	**11**	34	4.75	4.37	6.44	**11**	50	**6.63**	**1.55**	**4.62**
pipesworld-6	20	**10**	23	5.44	3.31	5.44	**10**	29	**7.33**	**0.95**	**4.82**
pipesworld-8	20	**11**	25	6.12	55.08	60.07	**11**	29	**7.57**	**6.79**	**12.58**
pipesworld-10	53	-	-	-	-	-	**19**	203	**10.97**	**45.30**	**97.27**

5 Conclusion

We have seen a flexible attempt to optimize pattern database exploration prior to the overall search process using genetic algorithms[14]. The approach optimizes the partition of multi-variate variables into disjoint patterns. We showed that pattern optimization is essential and optimization with genetic algorithms can increase not only the search time, but also the total run time. While the greedy bin packing strategy often runs out of memory, improved pattern selection with genetic algorithm scales better and can find solutions where bin packing fails.

Given the time and space efficiency of symbolic search, we could construct and evaluate many large pattern databases in a limited amount of time. Driven

[14] To the author's knowledge, optimization of patterns has not been considered before. We are aware some (unpublished) work on the automated generation of pattern databases by Holte and Hernádvölgyi. Their approach enumerates *all* possible (unsubsumed) pattern partitions that are not subsumed by already considered ones.

by the theory of search tree prediction, we have chosen the mean heuristic value as a fitness function. For the evaluation of each chromosome we have computed an entire set of pattern databases. Faster construction of larger databases favors a symbolic construction, and the exploration gains obtained in the experiments are promising. The encoding of pattern partition in a 2D gene allows experts to reason on the structure of good patterns and to perform pattern fine-tuning.

Constructing the pattern databases for each fitness evaluation consumes a considerable amount of time, especially if pattern databases become large. Future work will address *learning* of the fitness functions in smaller instances for bootstrapping in a genetic algorithm for larger instances. We also plan to consider alternative optimization methods as the search efficiency varies a lot in different runs. This suggests to apply *randomized local search* with *random restarts* [32].

Acknowledgements. Stefan Edelkamp is supported by DFG in the projects *Heuristic Search* (Ed 74/3) and *Directed Model Checking* (Ed 74/2).

References

1. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 203–213 (2001)
2. Brie, A.H., Morignot, P.: Genetic planning using variable length chromosomes. In: ICAPS, pp. 320–329 (2005)
3. Bryant, R.E.: Symbolic manipulation of boolean functions using a graphical representation. In: ACM/IEEE Design Automation Conference, pp. 688–694 (1985)
4. Clarke, E.M., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems 16(5), 1512–1542 (1994)
5. Culberson, J.C., Schaeffer, J.: Pattern databases. Computational Intelligence 14(4), 318–334 (1998)
6. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press, Cambridge (2005)
7. Edelkamp, S.: Planning with pattern databases. In: ECP, pp. 13–24 (2001)
8. Edelkamp, S.: Symbolic pattern databases in heuristic search planning. In: AIPS, pp. 274–293 (2002)
9. Edelkamp, S.: Taming numbers and durations in the model checking integrated planning system. Journal of Artificial Intelligence Research 20, 195–238 (2003)
10. Edelkamp, S.: External symbolic heuristic search with pattern databases. In: ICAPS, pp. 51–60 (2005)
11. Edelkamp, S., Lluch-Lafuente, A.: Abstraction in directed model checking. In: ICAPS-Workshop on Connecting Planning Theory with Practice (2004)
12. Edelkamp, S., Reffel, F.: OBDDs in heuristic search. In: KI, pp. 81–92 (1998)
13. Felner, A., Alder, A.: Solving the 24 puzzle with instance dependent pattern databases. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, pp. 248–260. Springer, Heidelberg (2005)
14. Felner, A., Meshulam, R., Holte, R.C., Korf, R.E.: Compressing pattern databases. In: AAAI, pp. 638–643 (2004)
15. Felner, A., Zahavi, U., Schaeffer, J., Holte, R.: Dual lookups in pattern databases. In: IJCAI, pp. 103–108 (2005)

16. Fikes, R., Nilsson, N.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2, 189–208 (1971)
17. Gaschnig, J.: A problem similarity approach to devising heuristics: First results. In: IJCAI, pp. 434–441 (1979)
18. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. STTT 6(2), 117–127 (2004)
19. Hansen, E.A., Zhou, R., Feng, Z.: Symbolic heuristic search using decision diagrams. In: Koenig, S., Holte, R.C. (eds.) SARA 2002. LNCS (LNAI), vol. 2371, Springer, Heidelberg (2002)
20. Hansson, O., Mayer, A., Valtora, M.: A new result on the complexity of heuristic estimates for the A* algorithm (research note). Artificial Intelligence 55, 129–143 (1992)
21. Haslum, P., Bonet, B., Geffner, H.: New admissible heuristics for domain-independent planning. In: AAAI, pp. 1163–1168 (2005)
22. Haslum, P., Geffner, H.: Admissible heuristics for optimal planning. pp. 140–149 (2000)
23. Helmert, M.: A planning heuristic based on causal graph analysis. In: ICAPS, pp. 161–170 (2004)
24. Hernádvölgyi, I.T.: Automatically Generated Lower Bounds for Search. PhD thesis, University of Ottawa (2003)
25. Hoffmann, J., Nebel, B.: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research 14, 253–302 (2001)
26. Holland, J.: Adaption in Natural and Artificial Systems. PhD thesis, University of Michigan (1975)
27. Holte, R.C., Grajkowski, J., Tanner, B.: Hierarchical heuristic search revisited. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, pp. 121–133. Springer, Heidelberg (2005)
28. Holte, R.C., Hernádvögyi, I.T.: A space-time tradeoff for memory-based heuristics. In: AAAI (1999)
29. Holte, R.C., Newton, J., Felner, A., Meshulam, R., Furcy, D.: Multiple pattern databases. In: ICAPS, pp. 122–131 (2004)
30. Holte, R.C., Perez, M.B., Zimmer, R.M., Donald, A.J.: Hierarchical A*: Searching abstraction hierarchies. In: AAAI, pp. 530–535 (1996)
31. Jensen, R.M., Bryant, R.E., Veloso, M.M.: SetA*: An efficient BDD-based heuristic search algorithm. In: AAAI, pp. 668–673 (2002)
32. Junghanns, A.: Pushing the Limits: New Developments in Single-Agent Search. PhD thesis, University of Alberta (1999)
33. Klein, D., Manning, C.: A* parsing: Fast exact Viterbi parse selection. In: Human Language Technology Conference of North American Chapter of the Association for Computational Linguistics (2003)
34. Knoblock, C.A.: Automatically generating abstractions for planning. Artificial Intelligence 68(2), 243–302 (1994)
35. Korf, R.E.: Finding optimal solutions to Rubik's Cube using pattern databases. In: AAAI, pp. 700–705 (1997)
36. Korf, R.E., Felner, A.: Chips Challenging Champions: Games, Computers and Artificial Intelligence. In: chapter Disjoint Pattern Database Heuristics, pp. 13–26. Elsevier, Amsterdam (2002)
37. Korf, R.E., Reid, M., Edelkamp, S.: Time Complexity of Iterative-Deepening-A*. Artificial Intelligence 129(1–2), 199–218 (2001)
38. Korf, R.E., Zhang, W., Thayer, I., Hohwald, H.: Frontier search. Journal of the ACM 52(5), 715–748 (2005)

39. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an ai planning heuristic for directed model checking. In: Valmari, A. (ed.) Model Checking Software. LNCS, vol. 3925, Springer, Heidelberg (2006)
40. Mostow, J., Prieditis, A.E.: Discovering admissible heuristics by abstracting and optimizing. In: IJCAI, pp. 701 – 707 (1989)
41. Muslea, I.: A general-propose AI planning system based on genetic programming. In: Genetic Programming Conference (Late Breaking Papers), pp. 157–164 (1997)
42. Pearl, J.: Heuristics. Addison-Wesley, London (1985)
43. Qian, K., Nymeyer, A.: Heuristic search algorithms based on symbolic data structures. In: ACAI, pp. 966–979 (2003)
44. Qian, K., Nymeyer, A.: Guided invariant model checking based on abstraction and symbolic pattern databases. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 497–511. Springer, Heidelberg (2004)
45. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
46. Schroedl, S.: An improved search algorithm for optimal multiple sequence alignment. Journal of Artificial Intelligence Research 23, 587–623 (2005)
47. Silver, D.: Cooperative pathfinding. In: Conference on Artificial Intelligence and Interactive Digital Entertainment, pp. 117–122 (2005)
48. Spector, L.: Genetic programming and AI planning systems. In: AAAI, pp. 1329–1334 (1994)
49. Valtorta, M.: A result on the computational complexity of heuristic estimates for the A* algorithm. Information Sciences 34, 48–59 (1984)
50. Wall, M.: GAlib – A C++ Library of Genetic Algorithm Components. Massachusetts Institute of Technology (2005)
51. Westerberg, H., Levine, J.: Optimising plans using genetic programming. In: ECP, page Poster (2001)
52. Zhou, R., Hansen, E.: Space-efficient memory-based heuristics. In: AAAI, pp. 677–682 (2004)
53. Zhou, R., Hansen, E.: External-memory pattern databases using structured duplicate detection. In: AAAI (2005)