Automatic construction of admissible heuristics for classical cost-optimal planning problems

Eric Andrews
Seminar: Heuristic Search, Fall 2013
Department of Computer Science, University of Helsinki

Abstract

Many interesting problems, such as finding the cost-efficient sequence of actions for an autonomous robot, can be cast as classical single-agent planning tasks, in which the aim is to find a plan that solves the problem optimally. The search for a plan is often performed with state space search methods, which benefit from a guiding heuristic. This seminar report surveys some state-of-the-art methods for automatically deriving such heuristics based on the description of the planning task at hand.

1 Introduction

When solving single-agent planning problems using best-first search algorithms, it is useful to have a informed heuristic that guides the search through the search space so that node expansions are minimized and the heuristic remains efficiently computable. These two criteria are often at odds.

Coming up with heuristics that balance the trade-off between performance and informedness has typically been a manual endeavor requiring domain-dependent knowledge about the planning problem on hand [3]. Not surprisingly, there is value in studying methods that can automatically generate good heuristics for any given arbitrary planning problem.

In this paper we will examine some state-of-the-art methods that have been not only theoretically laid out, but also implemented and empirically evaluated against challenging benchmark problems. A common theme appearing in these methods, is the abstraction or "relaxation" of the original problem into a new problem that can be solved quickly, and which serves as a basis for constructing the heuristic of interest.

Two of the presented methods are based on pattern databases, which by now, are a common method for automatically constructing domain-independent heuristics. The last presented method borrows from the model-checking community and is a novel approach to building heuristics.

2 Background

To start off, we define precisely the kind of planning problems we are considering and the kind of solutions (plans) we are searching for them. An example is presented in order for the reader to intuitively grasp the types of problems under consideration.

2.1 Planning task model

A planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star, c \rangle$, where $\mathcal{V}$ is a finite set of *state variables*. Each state variable $v \in \mathcal{V}$ has an associated finite domain D_v. A *state* is a complete variable assignment over $\mathcal{V}$, formally $\langle w_1, w_2, ..., w_n \rangle$ where $w_i \in D_i$ and $n = |\mathcal{V}|$, assuming the variables of $\mathcal{V}$ are numbered in some way. The set of operators $\mathcal{O}$ consists of operators $o = \langle pre; eff \rangle$, where *pre* and *eff* are partial variable assignments that define the preconditions and effects the operator has. For an operator to be *applicable* to some state s, the precondition *pre* has to be consistent with s, and the resulting *successor state* is obtained by updating s with the effects *eff*. The cost of applying an operator is given by the function $c : \mathcal{O} \rightarrow \mathbb{R}^+_0$, so $c(o) > 0$ for all $o \in \mathcal{O}$ [3], [5], [7].

A *plan* is an applicable sequence of operators that leads from the *initial state* s_0 to a state that is consistent with the *goal* $s_\star$, which is a partial variable assignment. The cost of a plan is the sum of its operators' costs. The solution we are interested in is the *optimal plan*, which, as its name implies, is the plan with minimum cost [3], [5], [7].

The "classical" planning task model defined here has been studied extensively in planning literature, and it is compatible with real world automated planners like STRIPS (Stanford Research Institute Problem Solver).

2.1.1 Example

$$\begin{aligned}
\mathcal{V} &= \{A, B, C\} \\
D_v &= \{0, 1, 2, 3\} \text{ for } v \in \{A, B\} \\
D_C &= \{0, 1\} \\
s_0 &= \{A = 0, B = 0, C = 0\} \\
s_\star &= \{A = 3, B = 2\} \\
\mathcal{O} &= \{\text{act}\} \cup \{\text{inc}_k | k \in \{0, 1, 2\}\} \cup \{\text{copy}_k | k \in D_A\} \\
&\quad \text{where act} = \langle C = 0;\ C = 1\rangle \\
&\quad \text{inc}_k = \langle A = k, C = 1;\ A = k + 1, C = 0\rangle \\
&\quad \text{copy}_k = \langle A = k, C = 1;\ B = k, C = 0\rangle \\
c(o) &= 1 \text{ for all } o \in \mathcal{O}
\end{aligned}$$

Figure 1: Example Planning Task (stylistically adapted from [5])

An example of a simple planning task is given in Figure 1. There are three variables A, B, C of which the latter has a smaller domain than the previous two. The initial state starts with all variables being zero and the goal we are aiming for is $A = 3, B = 2$ and C can be any value in its domain. In the context of this particular planning task, variable C acts as a sort of activator boolean that is necessary to be "on" (=1) in order to increment A with operator inc_k or copy the value of A into B with operator copy_k. The optimal plan for this problem is $\langle \text{act}, \text{inc}_0, \text{act}, \text{inc}_1, \text{act}, \text{copy}_2, \text{act}, \text{inc}_2\rangle$ with a cost of 8.

The example presented here is very simplified and is not a fair demonstration of the complexity of problems that can be solved using this model. Sokoban, (n^2-1)-Puzzles and a variety of real world logistical problems can be modeled using this framework. [1]

3 Methods

Let us now consider the problem of automatic construction of an admissible heuristic for any given planning task. The methods presented here are based on building an abstraction of the original planning task which is easier to solve in terms of memory use and computation time. This abstraction is then used as the heuristic.

There are two main questions that the methods need to answer in order to be able to derive heuristics. What are the abstractions like (the abstraction model), and with that in mind, how do we generate, evaluate and choose between viable abstractions.

3.1 Pattern database approaches

We start off by exploring the pattern database approaches to abstraction. First we look into seminal work done by Haslum et al. [3], after which we will see how further work [5] attempts to improve on this by means of linear programming. Thirdly and finally, in Section 3.2, we consider a quite different approach to the two main questions [7].

3.1.1 Abstraction, patterns, pattern databases

An *abstraction* of a planning task Π is obtained by considering a subset of its state variables $P \subseteq \mathcal{V}$ called a *pattern*. In the process, we remove from the preconditions and effects of operations, and from the initial state and goal, all variables not in P. This results in a smaller planning task, whose states we call *abstract states*, and we denote by $h^P(s)$ the minimum cost of reaching a goal from the abstract state corresponding to the original state s [3], [5].

As we have only relaxed constraints, intuitively it is clear that $h^P(s)$ works as a lower bound for the cost of reaching a goal from s in the original state space. Thus h^P is an admissible (and consistent) heuristic [3].

If we abstract the planning task in Figure 1 with the pattern $P = \{A, B\}$, leaving out C, the following happens. The set of state variables reduces to $\{A, B\}$. The starting state becomes $\{A = 0, B = 0\}$ and goal state remains unchanged as it didn't contain C to begin with. The new operators are: act $= \langle ; \rangle$, $\text{inc}_k = \langle A = k; A = k + 1\rangle$ and $\text{copy}_k = \langle A = k; B = k\rangle$. The optimal plan for this new planning task is then $\langle \text{inc}_0, \text{inc}_1, \text{copy}_2, \text{inc}_2\rangle$ with a cost of 4.

A *pattern database* [2] (PDB) is a table of precomputed cost-to-goal information for all abstract states for some pattern. PDB heuristics, which we will be concerned with, use these tables to report heuristic estimates. It is desirable from a computational stance that the number of abstract states (bounded above by $\prod_{v \in P} |D_v|$) stay small. Otherwise a large cost-to-goal table needs to be computed and stored [3], [5].

[1]A vast collection of tasks can be found at the IPC-2011 (International Planning Competition 2001) homepage `http://www.plg.inf.uc3m.es/ipc2011-deterministic/`

3.1.2 Pattern collections and the canonical heuristic

The approach by Haslum et al. [3] considers a collection of patterns rather than just a single large pattern. The problem with the latter is that the size of the abstract state space grows quickly as more variables are included. This makes calculating and storing the corresponding PDB infeasible in many cases. To understand how patterns can be combined into collections that can be used as heuristics, some definitions are in order.

An admissible heuristic h *dominates* another h' iff $h(s) \geq h'(s)$ for all states s. Evidently, if A and B are patterns such that $A \subseteq B$, then h^B dominates h^A. Furthermore, h^A acts as an admissible heuristic for the (abstract) planning task of B.

Given two PDB heuristics h^A and h^B, both are dominated by the admissible heuristic $h(s) = \max\left(h^A(s), h^B(s)\right)$. In the case where the set of operations that affect some variable in A is disjoint from the set of operations which affect any variable in B, we say that patterns A and B are (pairwise) additive. Additivity ensures that $h(s) = h^A(s) + h^B(s)$ is an admissible heuristic, and this heuristic dominates the maximum shown earlier.

It is desirable of course to be able to utilize the additivity relationship among patterns for stronger heuristics. For a collection of patterns $C = \{P_1, ..., P_k\}$, however, it is rarely the case that all patterns would be pairwise additive. So we try the best we can with the *canonical heuristic*, which is a combination of PDB heuristics resulting in a strong, admissible heuristic. Let A be the collection of all maximal (in terms of set inclusion) additive subsets of C. The canonical heuristic function is then

$$h^C(s) = \max_{S \in A} \sum_{P \in S} h^P(s). \tag{1}$$

3.1.3 iPDB procedure

Armed with the knowledge of the abstraction model, it is time to delve into how to actually find and choose the pattern collections of interest. The question we address is: given a planning problem and limited amount of memory that may be allocated to the PDBs, how do we find and choose a collection of patterns that satisfy the memory requirements while giving the best search performance?

Haslum et al. [3] view the problem as a discrete optimization problem and devise a hill-climbing algorithm to find a local optimum, as the true optimum requires an exhaustive search due to the memory limit [3]. The search space used in the hill-climbing consists of states that represent pattern collections. The neighborhood of a state S_C consists of pattern collections that can be made from the pattern collection C of S_C by a certain modification.

The algorithm starts off with a collection consisting of one pattern per goal variable (variable that appears in the goal), each containing only that variable. The main loop of the algorithm constructs several new collections C' from the current collection C, by selecting a pattern $P \in C$ and a variable $v \notin P$, and setting $C' = C \cup \{P \cup \{v\}\}$. Every collection C' whose patterns' PDBs' memory use stays below the memory limit is considered a neighbor.

In the next step, the best neighbor C' is set as the collection C for the next iteration. The main loop continues to carry on, until at some point, no neighbor satisfies the size limit or the improvement in heuristic quality is insignificant. Note that because $C \subseteq C'$, the heuristic quality cannot decrease in between iterations.

Given that we expand the neighborhood of C and have several choices of C', how do we choose the one that generates the strongest heuristic? In Haslum et al. [3] it is motivated that we can use a relative measure that predicts the improvement of C' with respect to C.

The *counting approximation* draws a uniform random sample of m states $\{s_1, ...s_m\}$ from the original planning task at hand (using random walks), and measures the ratio for which $h^C(s_i) < h^{C'}(s_i)$ where $i \in [1, m]$. Therefore a collection C' is ranked high, if it is able to improve the heuristic estimate for many of the states. The authors [3] also introduce a more accurate predictor, which takes into account the number of states in the search tree within some cost bound.

Along with the steps explained above, Haslum et al. [3] introduce some pruning criteria and optimizations based on causal relationships of the state variables that make the hill-climbing procedure feasible in practice.

3.1.4 Linear programming method

Let us ignore the canonical heuristic for a moment, and reason through a stronger way of combining a collection of patterns into an additive heuristic [5].

For each operator $o \in \mathcal{O}$ in a given planning task Π, introduce a variable X_o, which is the total cost contributed by the operator o in some (fixed) optimal plan. In the example given in Figure 1, variable X_{act} would equal 4, as operator act has a cost of 1 and is called 4 times in the optimal plan. Clearly $X_o \geq 0$ for all operators, and the cost of the optimal plan can be represented as $\sum_{o \in \mathcal{O}} X_o$.

Suppose P is a pattern of planning task Π. The equivalent heuristic h^P being admissible, it must hold that $h^P(s) \leq \sum_{o \in \mathcal{O}} X_o$. This bound can be further tightened by realizing that operators o not affecting any variable in P do not contribute to the cost of the optimal plan in the abstract task. Therefore $h^P(s) \leq \sum_{o \in \mathcal{O}'} X_o$, where $O' = \{o = \langle pre, \mathit{eff} \rangle \in \mathcal{O} \mid \mathit{eff}$ affects variables in $P\}$.

There is still room for some reduction in the number of variables X_o. This is done in order to make the soon presented linear program feasible in practice [5]. Assume we have two (or more) operators o_1 and o_2 and a pattern collection C. We can combine the variables into a single variable $X_{o_{1,2}} = X_{o_1} + X_{o_2}$ if o_1 and o_2 affect the same subset of patterns in C. Combining variables as described above, leads to a partition $\mathcal{O}/\sim$ on the respective operators $\mathcal{O}$.

The *post-hoc optimization heuristic* [5], $h^{\mathrm{PhO}}(s)$, for a planning task Π, a pattern collection $C = \{P_1, ..., P_k\}$, and a state s of Π, can be estimated by the objective value of the following linear program:

$$\begin{aligned} \text{Minimize} \sum_{[o] \in \mathcal{O}/\sim} X_{[0]} & \quad \text{subject to} \\ \sum_{[o] \in \mathcal{O}/\sim, [o] \text{ affects } P_i} X_{[0]} \geq h^{P_i}(s) & \quad \text{for all } i \in \{1, ..., k\} \\ X_{[0]} \geq 0 & \quad \text{for all } [o] \in \mathcal{O}/\sim \end{aligned}$$

To solve this linear program, the PDBs of the patterns in C must be known. After calculating and plugging in the heuristic estimate of reaching a goal from state s for all patterns in C, or $h^{P_i}(s)$ for all $P_i \in C$, we can solve the linear program to attain a strong heuristic estimate that utilizes all of the patterns.

The admissibility of the heuristic is guaranteed by the individual patterns it is composed from, due to all accounted costs being justified by the admissibility of some component heuristic [5].

Why h^{PhO} dominates the canonical heuristic is proved in detail by Pommerening et al. [5]. The intuition is that the linear program can be viewed from a maximization perspective by considering its *dual program*. The objective function of the dual program is similar to that of Equation 1 (canonical heuristic function), except that instead of choosing the maximum, a coefficient (≤ 1) is put in front of each heuristic summation. The coefficients are constrained by the operators, and the objective is to maximize the objective function.

The post-hoc optimization heuristic can therefore be seen as the LP relaxation of the integer program formulation of the canonical heuristic. Rather than having to select a single 1 coefficient with the rest being 0, we can instead adjust the coefficients arbitrarily. Consequently h^{PhO} dominates h^C as there are more coefficient configurations that can be chosen.

The hill-climbing procedure (iPDB) presented earlier can be used in conjunction with the newly presented heuristic. However, according to Pommerening et al [5], a more effective method is to systematically generate and include all patterns up to a given size in the pattern collections. Further pruning criteria based on causal relationships of state variables is then needed in order to prevent the collections from becoming too large and to exclude useless patterns.

3.2 Cartesian abstraction approach

The approach presented by Seipp and Helmert [7] is based on a method called Counterexample-guided abstraction refinement (CEGAR) which has been explored previously [1] in model-checking literature.

In the context of model-checking, suppose we have a concurrent system and we want to verify whether a certain property holds in it. The idea of CEGAR is to start off with a coarse abstraction of the system, in which we search for an error trace that violates the property of interest. If and when we find one, we check if that error trace generalizes to the actual system, and if not, the abstraction is refined just enough so as to guarantee that the same error trace will never be encountered again. From there the process repeats. If an error trace does generalize, we have found out that property does not hold in the system.

Before going into the details of the algorithm, and especially how it is applied to the construction of domain-independent heuristics, we will look at the abstraction model considered by Seipp and Helmert [7].

3.2.1 Cartesian abstraction

A transition system is a tuple $\mathcal{T} = \langle S, L, T, s_0, S_\star \rangle$ consisting of a finite set of states S, a finite set of transition labels L, a set of transitions $T \subset S \times L \times S$, an initial state s_0, a set of goal states $S_\star \subseteq S$, and a cost function $c : L \to \mathbb{R}_0^+$ assigning each label an associated cost.

A planning task Π *induces* a transition system as follows. The states of Π (complete variable assignments over $\mathcal{V}$) become the states S. The operators $\mathcal{O}$ become the labels L. Initial state is kept as it is. Goal states $S_\star$ are all states that are consistent with the partial variable assignment $s_\star$ of Π. Transitions T are all the possible relations that can be between the states of Π with operators in $\mathcal{O}$.

Suppose we have a planning task Π and the transition system $\mathcal{T}$ induced from it. Let's define an equivalence relation $\sim$ on their states called the *abstract relation*. This relation partitions the states into equivalence classes which we consider the abstract states. Notationally $[s]_\sim$ denotes the equivalence class to which state s belongs to in the abstraction.

Using the relation $\sim$ we can modify $\mathcal{T}$ into an *abstract transition system* that has the following differences compared to the original. The set of states is now $\{[s]_\sim | s \in S\}$, initial state is $[s_0]_\sim$, goal states are $\{[s_\star]_\sim | s_\star \in S_\star\}$. Finally, the transitions become $\{\langle [s]_\sim, l, [s']_\sim \rangle \mid \langle s, l, s' \rangle \in T\}$.

This resulting abstract transition system can then be used as an admissible (and consistent) heuristic. The heuristic function $h^\sim(s)$ is the minimum cost from $[s]_\sim$ to some goal state in $S_\star$.

As was mentioned earlier, the CEGAR algorithm works by making minor modifications to the abstraction in order to prevent certain traces from reoccurring. Unfortunately pattern databases do not support granular modification, as each time a variable is added to a pattern, the number of possible abstract states at least doubles.[2] This is why an alternative abstraction is proposed, one that supports more fine-grained refinement.

A set of states is *Cartesian* if it is of the form $A_1 \times A_2 \times ... \times A_n$, where $A_i \subset D_{v_i}$ for all $i \in [1, n]$, where D_{v_i} is the domain of state variable v_i. A *Cartesian abstraction* is a specific kind of abstract transition system. One in which every abstract state, or equivalence class, can be represented as a Cartesian set.

Just to be clear, it is not trivial for a set of states to be Cartesian. For example, the set consisting of $\langle 0,0,0 \rangle$ and $\langle 1,1,1 \rangle$ can't be represented in Cartesian form. The closest thing is $\{0,1\} \times \{0,1\} \times \{0,1\}$, but this entails other possibilities like $\langle 0,1,0 \rangle$.

3.2.2 Example

Let's use the planning task presented earlier in Figure 1 as an example. The transition system induced from this task is visualized in Figure 2.

In total there are 32 states in the transition system, of only which a portion is shown for convenience. The initial state $s_0 = 0,0,0$ is in the lower left corner. Using the edges of the graph, the aim is to get into either goal state in the set $S_\star = \{3\} \times \{2\} \times \{0,1\}$. The alert reader may have noticed by now that what the transition system really amounts to, is an explicit representation of the state space of the planning task.

[2]A variable has at least two values, e.g. $\{0,1\}$. Thus if there were previous n possible combinations of state variables, now there are at least $2n$.

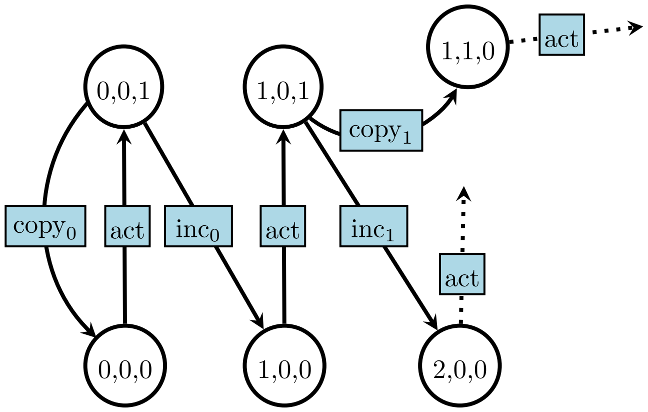

Figure 2: Visualization of transition system induced from running example. Only a region of the state space is shown. Each node i, j, k corresponds to state $A = i$, $B = j$ and $C = k$.

An example of a Cartesian abstraction of this particular transition system is given in Figure 3. Notice how each of the original 32 states are captured by either abstract state but not both. A Cartesian abstraction partitions the original state space into non-overlapping, non-empty subsets. An additional implied requirement stemming from the definition is that these subsets need be represented as Cartesian sets. The presented abstraction is actually not very useful, since both of its abstract states are considered goals, making its heuristic estimates always 0.

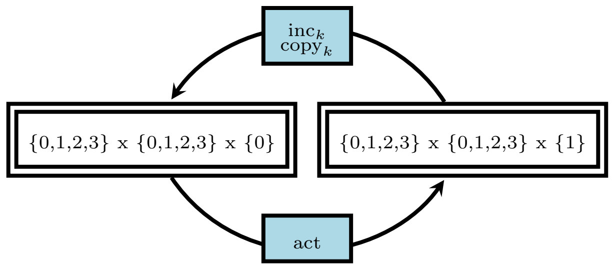

Figure 3: An example of one possible Cartesian abstraction of the transition system given in Figure 2. There are actually several edges going from right to left, but they have been labeled as a single edge for convenience.

3.2.3 Refinement algorithm

The pseudocode for the iterative refinement process is given in Algorithm 1. The algorithm starts off with initializing the abstract transition system $\mathcal{T}'$ to be the trivial Cartesian abstraction containing only a single all-capturing abstract state, namely $D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$. Note that the Cartesian abstraction $\mathcal{T}'$ is repre-

Algorithm 1 Refinement Loop (slightly altered [7])

```
1: $\mathcal{T}' \leftarrow$ TrivialAbstraction()
2: while not Terminate() do
3:     $\tau' \leftarrow$ FindOptimalPlan($\mathcal{T}'$)
4:     if $\tau'$ is undefined then
5:         return task is unsolvable
6:     end if
7:     $\phi \leftarrow$ FindFlaw($\tau'$)
8:     if there is now flaw in $\phi$ then
9:         return plan extracted from $\tau'$
10:    end if
11:    Refine($\mathcal{T}', \phi$)
12: end while
13: return $\mathcal{T}'$
```

sented as an explicit graph at all times.

The algorithm keeps on refining $\mathcal{T}'$ until a time or memory limit is hit, after which the current $\mathcal{T}'$ can be used in building a heuristic.

The main loop functions as follows. In Line 3 we compute an optimal solution to the abstract transition system $\mathcal{T}'$. If none is found, then it can be safely said that the original problem is neither solvable as there are only more constraints there. In Line 7 we attempt to apply the operators of the found plan τ' to the original planning task. In the unlikely scenario that this works, the original task has actually been solved using the abstract one. Most likely, however, a flaw will be found, which we then use to refine the abstraction $\mathcal{T}'$.

The flaws that can be found are listed below.

1. An operator is not applicable at some state s_{i-1}.
2. Operators were applicable, but end state s_n is not a goal.
3. At some point, $[s_i] \neq [s'_i]$, where s_i is the state of the original planning task and $[s'_i]$ the abstract state of the abstract planning task after applying the i:th operation.

In each of the three cases, $\mathcal{T}'$ is refined by splitting up the problematic abstract state $[s']$ into two, according to rules explained next, respectively numbered.

1. Split $[s_{i-1}]$ into those states in which the operator applies, and those in which it does not.
2. Split $[s_n]$ into those states that are goals, and those which are not.
3. Split $[s_{i-1}]$ into those states that cannot lead to any state in $[s'_i]$ after applying the i:th operation, and vice versa.

After the appropriate split has been decided, the old state is replaced with two new states. This requires some "rewiring" of transitions. For each incoming and outgoing transition of the removed state, we need to decide to which of the new states it re-connects to.

The Cartesian sets are closed under the presented splits above. Hence it is guaranteed that $\mathcal{T}'$ is a Cartesian abstraction at all times. From a performance perspective, splitting and rewiring can be done in $O(\sum_{v\in\mathcal{V}} D_v)$ [7].

4 Comparison

An empirical evaluation of performance, common to each of the presented papers, was to see how many instances of some planning task type the methods could solve in some time and memory bounds. This included both the resources required to construct the heuristic and to solve the task using best-first search.

The oldest and first presented of the three methods, iPDB , was able to solve many easy-to-normal Sokoban puzzles and a majority of 15-puzzles in experiments conducted by Haslum et al. [3]. Sokoban especially is a planning problem that has been difficult even for domain-specific solvers. 24-puzzles, while solvable by domain-specific solvers, were out of reach for domain-independent solvers at least back in 2007.

The empirical results by Pommerening et al. [5] of using the post-hoc optimization heuristic instead of the canonical heuristic are interesting. When using the hill-climbing (iPDB) procedure, canonical heuristic fairs better. The opposite is true with systematic pattern generation. Generally though, the post-hoc fairs better than the canonical heuristic. However even when considering the post-hoc optimization heuristic, the clear winner is the state-of-the-art LM-cut heuristic [4], based on landmarks and delete relaxation.

The CEGAR approach loses to iPDB in terms of number of solved tasks in experiments conducted by Seipp and Helmert [7]. However the results, which compare CEGAR to other approaches as well, hint that there may be a certain robustness to CEGAR. It is seldom the best or the worst performer, staying in the "middle ground" most of the time. Also compared to iPDB, CEGAR's heuristic estimates grows more smoothly with respect to number of abstract states.

Just recently there has been further development of the CEGAR approach that considers, similarly to the pattern database approaches, multiple patterns instead of just one [6]. The experimental results look promising with the new CEGAR solving 475 tasks compared to 407 with old CEGAR and 442 with iPDB.

A final point should be made about the nature of evaluating these methods empirically. Alongside science, there is significant competition and engineering that goes into this area of research as well. This means that older algorithms, like iPDB, may be more polished than newer candidates. For example, the version of iPDB pitted against CEGAR has been enhanced with further optimizations [8] not present in the original paper. Of course progress in hardware also expands the set of problems that can be solved with these methods.

5 Conclusion

In this seminar report, the planning task formalism was introduced, and an overview was given of some techniques for automatically constructing admissible heuristics for planning tasks. Constructing a heuristic for a planning task, and afterwards searching for an optimal plan using best-first search (e.g. A*) with the heuristic is, in many cases, faster than searching with a bad or no heuristic at all.

Stating which heuristic construction technique is best is hard, because there are trade-offs to be considered. For example, how many instances of a planning task type can be solved? In what time and memory constraints? How elegant or theoretically-sound is the technique? Or more generally, what combination of generation, evaluation and abstraction leads to the best results? To the author of this seminar report, CEGAR seemed to be the most elegant, and research on it looks promising.

From what I gathered, this is a very active area of research, as there were plenty of papers written on this subject this year alone. One key question in this line of research is, what kind of abstractions of the planning task result in good heuristics? Just as importantly, how should one actually generate and evaluate these abstractions? I found both to be very interesting questions, and there seems to be room for new answers to be discovered.

References

[1] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement". In: *Computer aided verification*. Springer. 2000, pages 154–169.

[2] Joseph C. Culberson and Jonathan Schaeffer. "Pattern databases". In: *Computational Intelligence* 14.3 (1998), pages 318–334.

[3] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. "Domain-independent construction of pattern database heuristics for cost-optimal planning". In: *AAAI*. Volume 7. 2007, pages 1007–1012.

[4] Malte Helmert and Carmel Domshlak. "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*. 2009, pages 162–169.

[5] Florian Pommerening, Gabriele Röger, and Malte Helmert. "Getting the Most Out of Pattern Databases for Classical Planning". In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. 2013, pages 2357–2364.

[6] Jendrik Seipp and Malte Helmert. "Additive Counterexample-Guided Cartesian Abstraction Refinement". In: *Late-Breaking Developments in the Field of Artificial Intelligence - Papers Presented at the Twenty-Seventh AAAI Conference on Artificial Intelligence*. 2013, pages 119–121.

[7] Jendrik Seipp and Malte Helmert. "Counterexample-guided Cartesian abstraction refinement". In: *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013). AAAI Press.* 2013, pages 347–351.

[8] Silvan Sievers, Manuela Ortlieb, and Malte Helmert. "Efficient Implementation of Pattern Database Heuristics for Classical Planning". In: *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SOCS)*. 2012, pages 105–111.