Explicit-State Abstraction: A New Method for Generating Heuristic Functions

Malte Helmert*
Albert-Ludwigs-Universität Freiburg
Freiburg, Germany
helmert@informatik.uni-freiburg.de

Patrik Haslum†
NICTA & Australian National University
Canberra, Australia
Patrik.Haslum@nicta.com.au

Jörg Hoffmann
University of Innsbruck (STI)
Innsbruck, Austria
joerg.hoffmann@sti2.at

Abstract

Many AI problems can be recast as finding an optimal path in a discrete state space. An abstraction defines an admissible heuristic function as the distances in a smaller state space where arbitrary sets of states are "aggregated" into single states. A special case are pattern database (PDB) heuristics, which aggregate states iff they agree on the state variables inside the pattern. Explicit-state abstraction is more flexible, explicitly aggregating selected pairs of states in a process that interleaves composition of abstractions with abstraction of the composites. The increased flexibility gains expressive power: sometimes, the real cost function can be represented concisely as an explicit-state abstraction, but not as a PDB. Explicit-state abstraction has been applied to planning and model checking, with highly promising empirical results.

Introduction

Many problems in AI, and computer science in general, can be recast as the problem of finding an optimal (i. e., shortest, or, more generally, least cost) path in a discrete state space. Examples include planning (Ghallab, Nau, and Traverso 2004), model checking (Clarke, Grumberg, and Peled 2000), diagnosis (Grastien et al. 2007) and PCFG parsing (Klein and Manning 2003). The state spaces explored in these applications are implicitly defined directed graphs, whose nodes are assignments to a set of state variables and whose edges correspond to state transitions (as defined by actions, program statements, events or grammar rules).

A highly successful method for solving such problems is the use of admissible heuristic functions, which give for every state a lower bound on the cost of reaching a solution state, in search algorithms such as A* (Pearl 1984). A useful heuristic function must be efficiently computable (low order polynomial time) as well as accurate (provide tight lower bounds). The heuristic function must also be constructed automatically from the instance description.

Abstractions provide a very general way of defining heuristic functions. An abstraction is a mapping that reduces the size of the state space by aggregating several states into one. The cost of reaching a solution state in the abstract state space is a lower bound on the same cost in the original state space. To make this approach practical, we must answer two questions: How to represent the mapping, and distances in the abstract space, in a way that is compact and permits efficient indexing? And how to choose a mapping that will yield an accurate heuristic?

Traditional answers to these questions simplify matters by considering a restricted class of abstractions. In particular, *pattern databases* (PDBs) have been explored in depth and shown to be very useful in several search problems (Culberson and Schaeffer 1998; Edelkamp 2001). The abstraction underlying a PDB ignores completely all but a subset of the state variables (known as the "pattern"): states that do not differ on the chosen variables are aggregated in the abstract space. This corresponds to a *projection* onto the variables in the pattern, which makes the mapping easy to represent. Sophisticated methods for choosing patterns have been developed (Haslum et al. 2007), and the corresponding heuristics can be compactly stored (Edelkamp 2002).

Yet, compact and efficient representation can be achieved also for a much more general class of abstractions, herein called *explicit-state abstractions* (Dräger, Finkbeiner, and Podelski 2006; Helmert, Haslum, and Hoffmann 2007). The distinguishing feature of these abstractions is that they explicitly select particular pairs of states to aggregate. The state space of the problem is viewed as the synchronized product of its projections onto the single state variables. Each such projection is an abstract space and the product is computed by iteratively composing two abstract spaces, replacing them with their product. (Note that the projection onto a subset of variables, i. e., a PDB, may also be viewed in this way.) While in a PDB the size of the abstract space is controlled by limiting the number of compositions, the key to managing the size of the abstraction in the more general setting is to *interleave* the process of composition with abstraction of the intermediate composites. Every time two abstract spaces are composed,

*This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See http://www.avacs.org/ for more information.

†NICTA is funded by the Australian Government, as represented by the Department of Broadband, Communications and the Digital Economy, and the Australian Research Council, through the ICT Centre of Excellence program.
Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

pairs of states are aggregated until the size of the resulting abstract space is below a given bound. The accuracy of the resulting heuristic often depends crucially on the order in which composites are formed and the choice of abstract states to aggregate. A few different strategies have been tried (Dräger, Finkbeiner, and Podelski 2006; Helmert, Haslum, and Hoffmann 2007), but exploring this question in greater depth remains an open topic.

Importantly, the increased flexibility of explicit-state abstractions leads to a gain in expressive power. There exist problems for which the real cost function can be represented concisely as an explicit-state abstraction, but not as a PDB. In fact, at least five widely used planning benchmark domains have this property (Helmert, Haslum, and Hoffmann 2007). In a nutshell, the reason is that explicit-state abstractions can concisely represent information involving all state variables, while any single pattern can involve only a very limited (at most logarithmic) number of variables.

The next two sections introduce explicit-state abstractions, in a generic form based on labelled transition systems. We point out the representational advantages over PDBs, and outline the application in planning and model checking.

Transition Systems and Abstractions

We use a standard notion of finite transition systems:

Definition 1 ***Transition systems.***
*A **transition system** is a 5-tuple $\mathcal{T} = \langle S, L, T, s_0, S_\star \rangle$ where S is a finite set of **states**, L is a finite set of **transition labels**, $T \subseteq S \times L \times S$ is a set of (labelled) **transitions**, $s_0 \in S$ is the **start state**, and $S_\star \subseteq S$ is the set of **solution states**.*

*A path from s_0 to any $s_\star \in S_\star$ following the transitions of $\mathcal{T}$ is a **solution** for $\mathcal{T}$. A solution is **optimal** iff the length of the path is minimal.*

*We say that $\mathcal{T}$ has **state variables** if there is a set V and for each $v \in V$ a finite domain $\mathcal{D}_v$, and the state set of $\mathcal{T}$ consists of all functions s on V with $s(v) \in \mathcal{D}_v$, for all v.*

As stated, transition systems usually capture the state space of a more concise formal model (e. g. a planning task or a program). The variables V correspond to the data store of that formal model. Transitions are labelled with the entity (e. g. the planning action or the program statement) enabling them. These labels are needed for synchronizing the product of several transition systems.

Definition 1 allows for transition systems without state variables because abstractions are also transition systems, with possibly more complex states. In particular, abstractions may aggregate several states into one:

Definition 2 ***Abstractions and abstraction heuristics.***
*An **abstraction** of a transition system $\mathcal{T} = \langle S, L, T, s_0, S_\star \rangle$ is a function $\mathcal{A}$ on S. The **abstract transition system** is $\mathcal{T}^\mathcal{A} = \langle \{\mathcal{A}(s) \mid s \in S\}, L, T^\mathcal{A}, \mathcal{A}(s_0), \{\mathcal{A}(s_\star) \mid s_\star \in S_\star\} \rangle$ where $T^\mathcal{A} = \{\langle \mathcal{A}(s), l, \mathcal{A}(s') \rangle \mid \langle s, l, s' \rangle \in T\}$. The **abstraction heuristic** $h^\mathcal{A}$ is the function which assigns to each state $s \in \mathcal{S}$ the length of the shortest path, in $\mathcal{T}^\mathcal{A}$, from $\mathcal{A}(s)$ to any $\mathcal{A}(s_\star)$ s.t. $s_\star \in S_\star$.*

If $\mathcal{A}$ is injective – if it does not merge any states – then $\mathcal{T}^\mathcal{A}$ and $\mathcal{T}$ are isomorphic. In any case, $\mathcal{A}$ is a homomorphism from $\mathcal{T}$ to $\mathcal{T}^\mathcal{A}$.[1] It is easy to see that, hence, $h^\mathcal{A}$ is an admissible (consistent, in fact) heuristic function for forward search on $\mathcal{T}$. The question is, how to represent $\mathcal{A}$ and $h^\mathcal{A}$? As stated, pattern databases answer this question in a restricted class of abstractions:

Definition 3 ***Projections.***
*Let $\mathcal{T} = \langle S, L, T, s_0, S_\star \rangle$ be a transition system with state variables V. Let $V' \subseteq V$. A **projection** of $\mathcal{T}$ onto V', written as $\pi_{V'}$, is an abstraction $\mathcal{A}$ where $\mathcal{A}(s) = \mathcal{A}(s')$ iff $s(v) = s'(v)$ for all $v \in V'$.*

The PDB simply "ignores" the variables $V \setminus V'$, making $\mathcal{A}$ trivial to represent, and enabling efficient storage of $h^\mathcal{A}$ (Edelkamp 2002). However, more flexible abstractions can be elegantly constructed, and can be quite beneficial.

Explicit-State Abstractions

Arbitrary abstractions can be generated by creating an explicit representation of $\mathcal{T}$ and iteratively aggregating pairs of states. Of course, relying on an explicit representation of $\mathcal{T}$ is not viable: we must abstract already while constructing $\mathcal{T}$ to ensure that the size of the construction remains feasible.

Several methods are conceivable. We choose to construct $\mathcal{T}$ as the synchronized product of its *atomic projections*, i. e., of $\mathcal{T}^{\pi_{\{v\}}}$ for $v \in V$. The construction is an iteration of steps replacing two transition systems with their synchronized product, and in between these steps we abstract by aggregating pairs of states. Note that the latter is exactly the most general abstraction method outlined above: it is made feasible by applying it in an iterated fashion, on transition systems much smaller than $\mathcal{T}$.

We now specify the method precisely. The synchronized product of two transition systems is defined as follows:

Definition 4 ***Synchronized product.***
*Let $\mathcal{T}' = \langle S', L, T', s'_0, S'_\star \rangle$ and $\mathcal{T}'' = \langle S'', L, T'', s''_0, S''_\star \rangle$ be transition systems. The **synchronized product** of $\mathcal{T}'$ and $\mathcal{T}''$ is defined as $\mathcal{T}' \otimes \mathcal{T}'' = \langle S, L, T, s_0, S_\star \rangle$, where $S = S' \times S''$, $\langle (s', s''), l, (t', t'') \rangle \in T$ iff $\langle s', l, t' \rangle \in T'$ and $\langle s'', l, t'' \rangle \in T''$, $s_0 = (s'_0, s''_0)$, and $S_\star = S'_\star \times S''_\star$.*

If $\mathcal{T}$ is a transition system with state variables V, and $\mathcal{A}', \mathcal{A}''$ are homomorphic abstractions of $\mathcal{T}$, then $T^{\mathcal{A}'} \otimes T^{\mathcal{A}''}$ is a homomorphic abstract transition system for $\mathcal{T}$, provided $\mathcal{A}'$ and $\mathcal{A}''$ are *orthogonal*, meaning there is no variable v that distinguishes states in both $\mathcal{A}'$ and $\mathcal{A}''$, and provided $\mathcal{T}$ satisfies an additional condition, which may be stated in several ways: one is that for any $U, U' \subseteq V$, if $\langle \pi_U(s), l, \pi_U(t) \rangle \in T^{\pi_U}$ and $\langle \pi_{U'}(s), l, \pi_{U'}(t) \rangle \in T^{\pi_{U'}}$, then $\langle \pi_{U \cup U'}(s), l, \pi_{U \cup U'}(t) \rangle \in T^{\pi_{U \cup U'}}$, i. e., if a transition exists in two different projections, it also exists in the combined projection, and similarly, if $\pi_U(s_\star)$ and $\pi_{U'}(s_\star)$ are abstract goal states in the respective projections, then $\pi_{U \cup U'}(s_\star)$ must be an abstract goal state in $\pi_{U \cup U'}$. We call a transition system with these properties *factored*. As hinted earlier, if $\mathcal{T}$ is factored, then $\mathcal{T}$ is isomorphic with

[1] A more general definition of abstractions would allow $\mathcal{T}^\mathcal{A}$ to be non-homomorphic, with additional transitions and solution states; herein, we consider homomorphic abstractions only.

generic algorithm compute-abstraction($\mathcal{T}$, N):
  abs := $\{\mathcal{T}^{\pi_{\{v\}}} \mid v \in V\}$
  while $|$*abs*$| > 1$:
    Select $\mathcal{T}_1, \mathcal{T}_2 \in$ *abs*.
    Shrink $\mathcal{T}_1$ and/or $\mathcal{T}_2$ until *size*$(\mathcal{T}_1) \cdot$ *size*$(\mathcal{T}_2) \leq N$.
    abs := (*abs* $\setminus \{\mathcal{T}_1, \mathcal{T}_2\}) \cup \{\mathcal{T}_1 \otimes \mathcal{T}_2\}$
  return the only element of *abs*

Figure 1: Computing an abstraction, with size bound N. (The size of a transition system is the number of states.)

$\bigotimes_{v \in V} \mathcal{T}^{\pi_v}$ and the outcome of the construction algorithm is a homomorphic abstraction of $\mathcal{T}$.

The algorithm is shown in Figure 1: It maintains a pool of transition systems, initially consisting of all atomic projections. Repeatedly, one of two possible operations is performed:

- Two transition systems are *composed*, replacing them with their synchronized product.
- A transition system is *shrunk* by replacing it with an abstraction of itself.

Time and space requirements are kept under control by enforcing a size limit, specified as an input parameter N. Each computed transition system, including the final result, contains at most N states. If there are more than N states in the product of two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ (i. e., if the product of their state counts exceeds N), either or both of the transition systems are abstracted by a sufficient amount before they are composed. As indicated earlier, this is done by iteratively aggregating pairs of states s and s'.

There are two important choice points left unspecified in the generic algorithm in Figure 1:

- *Composition strategy*: Which transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ to select from the current pool?
- *Shrinking strategy*: Which transition systems to abstract and which pairs of states s and s' to aggregate?

We refer to the combination of a particular composition and shrinking strategy as an *abstraction strategy*. This strategy is of paramount importance since it determines the quality of the resulting heuristic. It depends on the application domain – and even on the particular problem instance considered – what a good abstraction strategy is. So far, only first methods were devised for planning and model checking. We will get back to these below.

An interesting general observation can be made regarding the shrinking strategy. Think about which states s and s' should be aggregated, with the aim of minimizing loss of precision in the resulting heuristic function. If s is close to the start state and s' is close to a solution state, then identifying them introduces a shortcut in the abstract space. Thus, it is a good idea to use an *h-preserving* shrinking strategy: aggregate s and s' only if their h-values, i. e., their respective distances to the nearest solution states, are identical. Indeed, if $\mathcal{T}'$ results from $\mathcal{T}$ by such a strategy, both transition systems represent the same heuristic function. (Still, $\mathcal{T}$ may contain information not contained in $\mathcal{T}'$, leading to a difference in heuristic quality when computing the synchronized product of $\mathcal{T}$ or $\mathcal{T}'$ with another transition system.)

Representational Power

Two important observations, made by Helmert et al. (2007), relate explicit-state abstractions to pattern databases. First, h-preserving shrinking strategies "automatically capture" the benefits of additive patterns. Second, there exist families of transition systems for which the perfect heuristic h^* – the real solution distance – can be represented concisely as a flexible abstraction heuristic, but not as a PDB heuristic.

Regarding the first observation, two projections $\pi_{V'}$ and $\pi_{V''}$ are additive iff $h^{\pi_{V'}} + h^{\pi_{V''}}$ is admissible. Exploiting such additivity is the key to the success of many pattern database approaches (Felner, Korf, and Hanan 2004). Now, say $\pi_{V'}$ and $\pi_{V''}$ are additive; say $\mathcal{A}'$ is an h-preserving abstraction of $\pi_{V'}$, say $\mathcal{A}''$ is an h-preserving abstraction of $\pi_{V''}$, and say $\mathcal{B}$ is an h-preserving abstraction of $\mathcal{A}' \otimes \mathcal{A}''$. Then $h^{\mathcal{B}}$ *dominates the sum of the pattern database heuristics,* $h^{\pi_{V'}} + h^{\pi_{V''}}$. Hence, for every sum of concisely representable additive PDB heuristics, there exists a dominating explicit-state abstraction heuristic that also has a concise representation. So additivity is captured "automatically" in the sense that no special construction is needed for reaping it. Of course, it may still be important to take additivity into account in the abstraction strategy.

Regarding the second observation, a suitable family of transition systems is given by the planning benchmark domain "Gripper", where n balls must be transported from room A to room B using a robot with two hands ("grippers"). A perfect explicit-state abstraction heuristic can be obtained as follows: First, compose variables for the robot position and the grippers, without any aggregation. Then include variables for all balls, in any order, aggregating any two states iff they agree on the status of robot and grippers, as well as on the numbers of balls in each room. The resulting heuristic is perfect and is computed in polynomial time. In contrast, there exists no polynomial-sized family of pattern databases which can guarantee a heuristic value of at least $(\frac{2}{3} + \varepsilon)h^*$, for any $\varepsilon > 0$: any single pattern may contain only a logarithmic number of ball variables, and the patterns are not additive if more than one of them contains the robot position variable. Hence, the robot moves, constituting $\frac{1}{3}$ of the required actions, are considered for only a logarithmic number of balls.

There are at least four other widely used planning benchmark domains with a similar separation. Intuitively, in difference to pattern databases, explicit-state abstractions can concisely represent exponentially many equivalent options.

Applications

Explicit-state abstraction has so far been put to use in cost-optimal planning (Helmert, Haslum, and Hoffmann 2007) and model checking (Dräger, Finkbeiner, and Podelski 2006); we detail those results and outline some other potential application areas.

Planning

There is an obvious correspondence between state spaces of planning tasks and transition systems (though if actions have conditional effects, some care is needed to ensure the transition system is factored). Helmert et al. (2007) implement a "linear" composition strategy, which decides on some order of the atomic projections and then iteratively composes them in that order. The order is determined by causal dependencies between the state variables. Provided N is large enough, the shrinking strategy is h-preserving, and also g-preserving meaning s and s' are aggregated only if their distance to the start state is equal. The strategy prefers to aggregate states with high $g + h$ value, the intuition being that the A* algorithm is less likely to expand a state with high $g + h$ value than one with low $g + h$ value. With this strategy, plugging the explicit-state abstraction heuristic into A*, one obtains a planner that is highly competitive with – and sometimes superior to – the state-of-the-art in optimal sequential planning.

Alternative shrinking strategies were not explored in sufficient depth to obtain a conclusive picture. What can be said is that preserving h values, and preferring to merge states with high $g + h$ value, is important. In the linear composition strategy, ordering the atomic projections in a variation of the causal order did not have much impact; non-linear composition strategies were not investigated.

It remains a challenge how to choose a good value for N. From empirical observations, it appears that it suffices to fix one value per domain, though different domains require very different values (from $N = 1000$ to $N = 200000$).

Model Checking

Explicit-state abstraction was first explored in model checking of automata networks, where the idea suggests itself: the atomic projections are the single automata, the synchronized product is the synchronized execution of two automata, and the transition system is the overall state space of the network. Sabnani et al. (1989) had shown how to conservatively reduce automata in between synchronization steps, without losing any precision; Dräger et al. (2006) instead reduced non-conservatively by aggregating states, and hence obtained explicit-state abstraction heuristics.

Dräger et al. (2006) use a non-linear composition strategy, which composes pairs of abstractions based on state transitions with shared labels. A pair of abstractions is preferred if those transitions lead to states with low solution distance. The intention is to get rid of synchronization close to solutions. This method was empirically shown to be superior to a random composition strategy. The shrinking strategy is h-preserving and prefers to aggregate states with high h-value, which is shown to be superior to aggregating states with low h-value. No alternative abstraction strategies were explored.

Since heuristic search does not have a long tradition in model checking, a comparison to alternative approaches is difficult there. The implementation of Dräger et al. outperforms depth-first search and a fairly simplistic heuristic. The choice of N was made by hand, and may be less critical than in planning: Dräger et al. explore only $N = 2500$ and $N = 10000$, the latter performing best in all but one case.

Other Applications

Explicit-state abstraction can in principle be applied to any problem reducible to the problem of finding an optimal path in a discrete state space. Grastien et al. (2007) approach the problem of "optimistic" diagnosis of discrete-event systems in this way, posing the question as "Does there exist a normal (non-faulty) path that is consistent with observations?". Klein and Manning (2003) use A* to find the maximum-likelihood parse of a sentence in a probabilistic CFG, and propose a variety of heuristics for guiding it. (Some of those heuristics are based on abstraction; their abstraction mappings, however, are fixed.)

Conclusion

Explicit-state abstractions generalize pattern databases by explicitly selecting pairs of states to aggregate, rather than aggregating all states that agree on the state variables of a given pattern. The generalization significantly increases expressive power, and first empirical results in planning and model checking are promising. Developing the technique further, and applying it in other areas of AI, remains an exciting area of research.

References

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 2000. *Model Checking*. The MIT Press.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed model checking with distance-preserving abstractions. In *Proc. SPIN-2006*, 19–34.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. ECP 2001*, 13–24.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Proc. AIPS 2002*, 274–283.

Felner, A.; Korf, R.; and Hanan, S. 2004. Additive pattern database heuristics. *JAIR* 22:279–318.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Grastien, A.; Anbulagan; Rintanen, J.; and Kelareva, E. 2007. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proc. AAAI 2007*, 305–310.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. AAAI 2007*, 1007–1012.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS 2007*, 176–183.

Klein, D., and Manning, C. D. 2003. A* parsing: Fast exact Viterbi parse selection. In *Proc. HLT-NAACL 2003*, 40–47.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Sabnani, K. K.; Lapone, A. M.; and Uyar, M. Ü. 1989. An algorithmic procedure for checking safety properties of protocols. *IEEE Transactions on Communications* 37(9):940–948.