Exhibiting Knowledge in Planning Problems to Minimize State Encoding Length

Stefan Edelkamp and Malte Helmert

Institut für Informatik
Albert-Ludwigs-Universität, Am Flughafen 17, D-79110 Freiburg, Germany,
e-mail: {edelkamp,helmert}@informatik.uni-freiburg.de

Abstract. In this paper we present a general-purposed algorithm for transforming a planning problem specified in Strips into a concise state description for single state or symbolic exploration.
The process of finding a state description consists of four phases. In the first phase we symbolically analyze the domain specification to determine constant and one-way predicates, i.e. predicates that remain unchanged by all operators or toggle in only one direction, respectively.
In the second phase we symbolically merge predicates which lead to a drastic reduction of state encoding size, while in the third phase we constrain the domains of the predicates to be considered by enumerating the operators of the planning problem. The fourth phase combines the result of the previous phases.

1 Introduction

Single-state space search has a long tradition in AI. We distinguish memory sensitive search algorithms like A* [12] that store the explored subgraph of the search space and approaches like IDA* and DFBnB [15] that consume linear space with respect to the search depth. Especially on current machines memory sensitive algorithms exhaust main memory within a very short time.

On the other hand linear search algorithms explore the search tree of generating paths, which might be exponentially larger than the underlying problem graph. Several techniques such as transposition tables [21], finite state machine pruning [23], and heuristic pattern databases [3] have been proposed. They respectively store a considerable part of the search space, exhibit the regular structure of the search problems, and improve the lower bound of the search by retrieving solutions to problem relaxations. Last but not least, in the last decade several memory restricted algorithms have been proposed [4,22]. All memory restricted search approaches cache states at the limit of main memory.

Since finite state machine pruning is applicable only to a very restricted class of symmetric problems, single-state space search algorithms store millions of fully or partially defined states. Finding a good compression of state space is crucial. The first step is to efficiently encode each state; if we are facing millions of states we are better off with a small state description length.

The encoding length is measured in bits. For example one instance to the well-known Fifteen Puzzle can be compressed to 64 bits, 4 bits for each tile.

Single-state algorithms have been successful in solving "well-informed domains", i.e. problems with a fairly elaborated lower bound [13,16], for good estimates lead to smaller parts of the search tree to be considered. In solving one specific problem, manually encoding the state space representations can be devised to the user. In case of AI planning, however, we are dealing with a family of very different domains, merely sharing the same, very general specification language. Therefore planners have to be general-purposed. Domain-dependent knowledge has either to be omitted or to be inferred by the machine.

Planning domains usually have large branching factors, with the branching factor being defined as the average number of successors of a state within planning space. Due to the resulting huge search spaces planning resists almost all approaches of single-state space search. As indicated above automated finite state pruning is generally not available although there is some promising research on symmetry leading to good results in at least some domains [10].

On the other hand, domain-independent heuristic guidance in form of a lower bound can be devised, e.g. by counting the number of facts missing from the goal state. However, these heuristics are too weak to regain tractability. Moreover, new theoretical results in heuristic single-state search prove that while finite state machine pruning can effectively reduce the branching factor, in the limit heuristics cannot [5,17]. The influence of lower bounds on the solution length can best be thought of as a decrease in search depth. Therefore, even when incorporated with lower bound information, the problem of large branching factors when applying single-state space searching algorithms to planning domains remains unsolved. As a solution we propose a promising symbolic search technique also favoring a small binary encoding length.

2 Symbolic Exploration

Driven by the success of model checking in exploring search spaces of 10^{20} states and beyond, the new trend in search is reachability analysis [19]. Symbolic exploration bypasses the typical exponential growth of the search tree in many applications. However, the length of the state description severely influences the execution time of the relevant algorithms. In symbolic exploration the rule of thumb for tractability is to choose encodings of not much more than 100 bits.

Edelkamp and Reffel have shown that and how symbolic exploration leads to promising results in solving current challenges to single-agent search such as the Fifteen Puzzle and Sokoban [6]. Recent results show that these methods contribute substantial improvements to deterministic planning [7].

The idea in symbolically representing a set S is to devise a boolean function ϕ_S with input variables corresponding to bits in the state description that evaluates to true if and only if the input a is the encoding of one element s in S. The drawback of choosing boolean formulae to describe ϕ_S is that satisfiability checking is NP-complete. The unique representation with binary decision diagrams (BDDs) can grow exponentially in size, but, fortunately, this characteristic seldom appears in practice [2].

BDDs allow to efficiently encode sets of states. For example let $\{0,1,2,3\}$ be the set of states encoded by their binary value. The characteristic function of a single state is the minterm of the encoding, i.g. $\phi_{\{0\}}(x) = \overline{x_1} \wedge \overline{x_2}$. The resulting *BDD* has two inner nodes. The crucial observation is that the *BDD* representation of S increases by far slower than $|S|$. For example the *BDD* for $\phi_{\{0,1\}} = \overline{x_1}$ consists of one internal node and the *BDD* for $\phi_{\{0,1,2,3\}}$ is given by the 1-sink only.

An operator can also been seen as an encoding of a set. In contrast to the previous situation a member of the transition relation corresponds to a pair of states (s', s) if s' is a predecessor of s. Subsequently, the transition relation T evaluates to 1 if and only if s' is a predecessor of s. Enumerating the cross product of the entire state space is by far too expensive. Fortunately, we can set up T symbolically by defining which variables change due to an operator and which variables do not.

Let S_i be the set of states reachable from the start state in i steps, initialized by $S_0 = \{s\}$. The following equation determines ϕ_{S_i} given both $\phi_{S_{i-1}}$ and the transition relation: $\phi_{S_i}(s) = \exists s' \; (\phi_{S_{i-1}}(s') \wedge T(s', s))$. In other words we perform breadth first search with *BDDs*. A state s belongs to S_i if it has a predecessor in the set S_{i-1} and there exists an operator which transforms s' into s. Note that on the right hand side of the equation ϕ depends on s' compared to s on the left hand side. Thus, it is necessary to substitute s with s' in the *BDD* for ϕ_{S_i}. Fortunately, this substitution corresponds to a simple renaming of the variables.

Therefore, the key operation in the exploration is the *relational product* $\exists v (f \wedge g)$ of a variable vector v and two boolean functions f and g. Since existential quantification of one boolean variable x_i in the boolean function f is equal to disjunction $f|_{x_i=0} \vee f|_{x_i=1}$, the quantification of v results in a sequence of subproblem disjunctions. Although computing the relational product is NP-hard in general, specialized algorithms have been developed leading to an efficient determination for many practical applications.

In order to terminate the search we test, if a state is contained in the intersection of the symbolic representation of the set S_i and the set of goal states G. This is achieved by evaluating the relational product $\mathit{goalReached} = \exists x \; (\phi_{S_i} \wedge \phi_G)$. Since we enumerated $S_0, \ldots, S_{i-1}$ in case *goalReached* evaluates to 1, i is known to be the optimal solution length.

3 Parsing

We evaluate our algorithms on the AIPS'98 planning contest problems[1], mostly given in Strips [8]. An operator in Strips consists of pre- and postconditions. The latter, so-called effects, divide into an add list and a delete list.

Extending Strips leads to ADL with first order specification of conditional effects [20] and PDDL, a layered planning description domain language. Although symbolic exploration and the translation process described in this paper are not restricted to Strips, for the ease of presentation we will keep this focus.

[1] `http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html`.

A PDDL-given problem consists of two parts. In the domain specific part, predicates and actions are defined. A predicate is given by its name and its parameters, and actions are given by their names, parameters, preconditions, and effects. One example domain, Logistics, is given as follows[2].

```
(define (domain logistics-strips)
  (:predicates (OBJ ?obj) (TRUCK ?tru) (LOCATION ?loc) (AIRPLANE ?plane) (CITY ?city)
               (AIRPORT ?airport) (at ?obj ?loc)  (in ?obj ?obj) (in-city ?obj ?city))
(:action LOAD-TRUCK
  :parameters (?obj ?tru ?loc)
  :precondition (and (OBJ ?obj) (TRUCK ?tru) (LOCATION ?loc) (at ?tru ?loc) (at ?obj ?loc))
  :effect (and (not (at ?obj ?loc)) (in ?obj ?tru)))
(:action UNLOAD-TRUCK
  :parameters (?obj ?tru ?loc)
  :precondition (and (OBJ ?obj) (TRUCK ?tru) (LOCATION ?loc) (at ?tru ?loc) (in ?obj ?tru))
  :effect (and (not (in ?obj ?tru)) (at ?obj ?loc)))
(:action DRIVE-TRUCK
  :parameters (?tru ?loc-from ?loc-to ?city)
  :precondition  (and (TRUCK ?tru) (LOCATION ?loc-from) (LOCATION ?loc-to) (CITY ?city)
                   (at ?tru ?loc-from) (in-city ?loc-from ?city) (in-city ?loc-to ?city))
  :effect (and (not (at ?tru ?loc-from)) (at ?tru ?loc-to))) ... )
```

The problem specific part defines the objects to be dealt with and describes initial and goal states, consisting of a list of facts (instantiations to predicates).

```
(define (problem strips-log-x-1)
   (:domain logistics-strips)
   (:objects package6 package5 package4 package3 package2 package1
             city6 city5 city4 city3 city2 city1
             truck6 truck5 truck4 truck3 truck2 truck1 plane2 plane1
             city6-1 city5-1 city4-1 city3-1 city2-1 city1-1
             city6-2 city5-2 city4-2 city3-2 city2-2 city1-2)
   (:init (and (OBJ package1) (OBJ package2) ... (at package6 city3-1) .. ))
   (:goal (and (at package6 city1-2) ... )))
```

Using current software development tools, parsing a PDDL specification is easy. In our case we applied the Unix programs *flex* and *bison* for lexically analyzing the input and parsing the result into data structures. We used the standard template library, STL for short, for handling the different structures conveniently. The information is parsed into vectors of predicates, actions and objects. All of them can be addressed by their name or a unique numeric identifier, with STL maps allowing conversions from the former ones to the latter ones. Having set up the data structures, we are ready to start analyzing the problem.

4 Constant and One-Way Predicates

A *constant predicate* is defined as a predicate whose instantiations are not affected by any operator in the domain. Since Strips does not support types, constant predicates are often used for labeling different kinds of objects, as is the case for the `TRUCK` predicate in the Logistics domain. Another use of constant predicates is to provide persistent links between objects, e.g. the `in-city` predicate in the Logistics domain which associates locations with cities. Obviously, constant predicates can be omitted in any state encoding.

[2] Dots (...) are printed if source fragments are omitted.

Instantiations of *one-way* predicates do change over time, but only in one direction. There are no one-way predicates in the Logistics domain; for an example consider the Grid domain, where doors can be opened with a key and not be closed again. Thus `locked` and `open` are both one-way predicates. Those predicates need to be encoded only for those objects that are not listed as open in the initial state. In PDDL neither constant nor one-way predicates are marked and thus both have to be inferred by an algorithm. We iterate on all actions, keeping track of all predicates appearing in any effect lists. Constant predicates are those that appear in none of those lists, one-way predicates are those that appear either as add effects or as delete effects, but not both.

5 Merging Predicates

Consider the Logistics problem given above, which serves as an example for the remaining phases. There are 32 objects, six packages, six trucks, two airplanes, six cities, six airports, and six other locations. A naive state encoding would use a single bit for each possible fact, leading to a space requirement of $6 \cdot 32 + 3 \cdot 32^2 = 3264$ bits per state, since we have to encode six unary and three binary predicates. Having detected constant predicates, we only need to encode the `at` and `in` predicates, thus using only $2 \cdot 32^2 = 2048$ bits per state. Although this value is obviously better, it is far from being satisfying.

A human reader will certainly notice that it is not necessary to consider all instantiations of the `at` predicate independently. If a given package p is at location a, it cannot be at another location b at the same time. Thus it is sufficient to encode *where* p is located, i.e. we only need to store an object number which takes only $\lceil \log 32 \rceil = 5$ bits per package. How can such information be deduced from the domain specification? To tell the truth, this is not possible, since the information does not only depend on the operators themselves but also on the initial state of the problem. If the initial state included the facts (`at` p a) as well as (`at` p b), then p could be at multiple locations at the same time.

However, we can try to prove that the number of locations a given object is at cannot increase over time. For a given state, we define $\#at_2(p)$ as the number of objects q for which the fact (`at` p q) is true. If there is no operator that can increase this value, then $\#at_2(p)$ is limited by its initial value, i.e. by the number of corresponding facts in the initial state. In this case we say that `at` is *balanced* in the second parameter. Note that this definition can be generalized for n-ary predicates, defining $\#pred_i(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n)$ as the number of objects p_i for which the fact (`pred` $p_1 \ \dots \ p_n$) is true. If we knew that `at` was balanced in the second parameter, we would be facing one of the following situations:

- $\#at_2(p) = 0$: We have to store no information about the location of p.
- $\#at_2(p) = 1$: The location of p can be encoded by using an object index, i.e. we need $\lceil \log o \rceil$ bits, where o denotes the number of objects p can be assigned to in our problem.
- $\#at_2(p) > 1$: In this case, we stick to naive encoding.

So can we prove that the balance requirement for `at` is fulfilled? Unfortunately we cannot, since there are some operators that increase $\#at_2$, namely the `UNLOAD-TRUCK` operator. However, we note that whenever $\#at_2(p)$ increases, $\#in_2(p)$ decreases, and vice versa. If we were to merge `at` and `in` into a new predicate `at+in`, this predicate would be balanced, since $\#(at + in)_2 = \#at_2 + \#in_2$ remains invariant no matter what operator is applied.

We now want to outline the algorithm for checking the balance of $\#pred_i$ for a given predicate *pred* and parameter i: For each action a and each of its add effects e, we check if e is referring to predicate `pred`. If so, we look for a corresponding delete effect, i.e. a delete effect with predicate `pred` and the same argument list as e, except for the i-th argument which is allowed to be (and normally will be) different. If we find such a delete effect, it balances the add effect, and there is no need to worry.

If there is no corresponding delete effect, we search the delete effect list for any effect with a matching argument list (again, we ignore parameter i), no matter what predicate it is referring to. If we do not find such an effect, our balance check fails. If we do find one, referring to predicate `other`, then we recursively call our algorithm with the merged predicate `pred+other`. Note that "matching argument list" does not necessarily mean that `other` takes its arguments in the same order as `pred`, which makes the actual implementation somewhat more complicated.

It is even possible to match `other` if that predicate takes one parameter less than `pred`, since parameter i does not need to be matched. This is a special case in which $\#other_i(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)$ can either be 1 or 0, depending on whether the corresponding fact is true or not, since there is no parameter p_i here. Examples of this situation can be found in the Gripper domain, where `carry ?ball ?gripper` can be merged with `free ?gripper`.

If there are several candidates for `other`, all of them are checked, maybe proving balance of different sets of merged predicates. In this case, all of them are returned by the algorithm. It is of course possible that more than two predicates are merged in order to satisfy a balance requirement since there can be multiple levels of recursion. This algorithm checks the i-th parameter of predicate `pred`. Executing it for all predicates in our domain and all possible values of i and collecting the results yields an exhaustive list of balanced merged predicates.

In the case of the Logistics domain, our algorithm exhibits that merging `at` and `in` gives us the predicate `at+in` which is balanced in the second parameter. Looking at the initial facts stated in the problem specification, we see that we can store the locations of trucks, airplanes and packages by using six bits each, since $\#(at + in)_2$ evaluates to one for those objects, and that we do not need to encode anything else, since the other objects start off with a count of zero.

Note that $\lceil \log 32 \rceil = 5$ bits are not sufficient for encoding locations at our current level of information, since we not only have to store the index of the object we are referring to, but also which of the two predicates `at` or `in` is actually meant. Thus our encoding size can be reduced to $(6 + 6 + 2) \cdot 6 = 84$ bits, which is already a reasonable result and sufficient for many purposes.

6 Exploring Predicate Space

However, we can do better. In most cases it is not necessary to allow the full range of objects for the balanced predicates we have detected, since e.g. a package can only be at a location or in a vehicle (truck or airplane), but never at another package, in a location, and so on.

If a fact is present in the initial state or can be instantiated by any valid sequence of operators, we call it *reachable*, otherwise it is called *unreachable*.

Many facts can be proven to be unreachable directly from the operators themselves, since actions like `LOAD-TRUCK` require the object the package is put into to be a truck. However, there are some kinds of unreachable facts we do not want to miss that cannot be spotted that way.

For example, `DRIVE-TRUCK` can only move a truck between locations in the same city, since for a truck to move from a to b, there must be a city c, so that (`in-city` a c) and (`in-city` b c) are true. Belonging to the same city is no concept that is modeled directly in our Strips definition.

For those reasons, we do not restrict our analysis to the domain specification and instead take the entire problem specification into account. What we want to do is an exploration of predicate space, i.e. we try to enumerate all instantiations of predicates that are reachable by beginning with the initial set of facts and extending it in a kind of breadth-first search.

Note that we are exploring predicate space, not search space. We do not store any kind of state information, and only keep track of which facts we consider reachable. Thus, our algorithm can do one-side errors, i.e. consider a fact reachable although it is not, because we do not pay attention to mutual exclusion of preconditions. If a fact f can be reached by an operator with preconditions g and h, and we already consider g and h reachable, then f is considered reachable, although it might be the case that g and h can never be instantiated at the same time. This is a price we have to pay and are willing to pay for reasons of efficiency. Anyway, if we were able to decide reliably if a given combination of facts could be instantiated at the same time, there would hardly remain any planning problem to be solved. We tested two different algorithms for exploring predicate space, *Action-Based Exploration* and *Fact-Based Exploration*.

6.1 Action-Based Exploration

In the action-centered approach, the set of reachable facts is initialized with the facts denoted by the initial state. We then instantiate all operators whose preconditions can be satisfied by only using facts that we have marked as reachable, marking new facts as reachable according to the add effect lists of the instantiated operators. We then again instantiate all operators according to the extended set of reachable facts. This process is iterated until no further facts are marked, at which time we know that there are no more reachable facts.

Our implementation of the algorithm is somewhat more tricky than it might seem, since we do not want to enumerate all possible argument lists for the operators we are instantiating, which might take far too long for difficult problems (there are e.g. $84^7 \approx 3 \cdot 10^{13}$ different possible instantiations for the `drink` operator in problem Mprime-14 from the AIPS'98 competition).

To overcome this problem, we apply a backtracking technique, extending the list of arguments one at a time and immediately checking if there is an unsatisfied precondition, in which case we do not try to add another argument. E.g., considering the `LOAD-TRUCK` operator, it is no use to go on searching for valid instantiations if the `?obj` parameter has been assigned an object o for which (`OBJ` o) has not been marked as reachable.

There is a second important optimization to be applied here: Due to the knowledge we already have accumulated, we know that `OBJ` is a constant predicate and thus there is no need to dynamically check if a given object satisfies this predicate. This can be calculated beforehand, as well as other preconditions referring to constant predicates.

So what we do is to statically constrain the domains of the operator parameters by using our knowledge about constant and one-way predicates. For each parameter, we pre-compute which objects possibly could make the corresponding preconditions true. When instantiating operators later, we only pick parameters from those sets. Note that due to this pre-computation we do not have to check preconditions concerning constant unary predicates at all during the actual instantiation phase.

For one-way predicates, we are also able to constrain the domains of the corresponding parameters, although we cannot be as restrictive as in the case of constant predicates. E.g., in the Grid example mentioned above, there is no use in trying to open doors that are already open in the initial state. However, we cannot make any assumption about doors that are closed initially.

There are two drawbacks of the action-based algorithm. Firstly, the same instantiations of actions tend to be checked multiple times. If an operator is being instantiated with a given parameter list, it will be instantiated again in all further iterations. Secondly, after a few iterations, changes to the set of reachable facts tend to become smaller and less frequent, but even if only a single fact is added to the set during an iteration, we have to evaluate all actions again, which is bad. Small changes should have less drastic consequences.

6.2 Fact-Based Exploration

Therefore we shift the focus of the exploration phase from actions to facts. Our second algorithm makes use of a queue in which all facts that are scheduled to be inserted into the set of reachable facts are stored. Initially, this queue consists of the facts in the initial state, while the set of reachable facts is empty. We then repeatedly remove the first fact f from the queue, add it to the set of reachable facts and instantiate all operators whose preconditions are a subset of our set of reachable facts *and include* f. Add effects of these operators that are not yet stored in either the set of reachable facts or the fact queue are added to the back of the fact queue. This process is iterated until the fact queue is empty.

```
(5 bits) package6
  at  city6-1 city5-1 city4-1 city3-1 city2-1 city1-1
      city6-2 city5-2 city4-2 city3-2 city2-2 city1-2
  in  truck6 truck5 truck4 truck3 truck2 truck1 plane2 plane1
...
(5 bits) package1
  at  city6-1 city5-1 city4-1 city3-1 city2-1 city1-1
      city6-2 city5-2 city4-2 city3-2 city2-2 city1-2
  in  truck6 truck5 truck4 truck3 truck2 truck1 plane2 plane1
(1 bit) truck6
  at  city6-1 city6-2
...
(1 bit) truck1
  at  city1-1 city1-2
(3 bits) plane2
  at  city6-2 city5-2 city4-2 city3-2 city2-2 city1-2
(3 bits) plane2
  at  city6-2 city5-2 city4-2 city3-2 city2-2 city1-2
```

Table 1. Encoding the Logistics problem 1-01 with 42 bits.

Although it does not look as if much was gained at first glance, this algorithm is a big improvement to the first one.

The key difference is that when instantiating actions, only those instantiations need to be checked for which f is one of the preconditions, which means that we can *bind* all parameters appearing in that precondition to the corresponding values of f, thus reducing the number of degrees of freedom of the argument lists. Of course, the backtracking and constraining techniques mentioned above apply here as well. The problem of multiple operator instantiations does not arise. We never instantiate an operator that has been instantiated with the same parameter list before, since we require f to be one of the preconditions, and in previous iterations of the loop, f was not regarded a reachable fact.

Returning to our Logistics problem, we now know that a package can only be at a location, in a truck or in an airplane. An airplane can only be at an airport, and a truck can only be at a location which must be in the same city as the location the truck started at.

7 Combining Balancing and Predicate Space Exploration

All we need to do in order to receive the encoding we are aiming at is to combine the results of the previous two phases. Note that these results are very different: While predicate space exploration yields information about the facts themselves, balanced predicates state information about the *relationship* between different facts. Both phases are independent of each other, and to minimize our state encoding, we need to combine the results.

In our example, this is simple. We have but one predicate to encode, the `at+in` predicate created in the merge phase. This leads to an encoding of 42 bits (cf. Table 1), which is the output of our algorithm. However, there are cases in which it is not obvious how the problem should be encoded. In the Gripper domain (constant predicates omitted) the merge step returns the balanced predicates `at-robby`, `carry+free`, and `at+carry`; `at-robby` is an original operator, while `carry+free` and `at+carry` have been merged.

We do not need to encode each of the merged predicates, since this would mean encoding `carry` twice. If we had already encoded `carry+free` and now wanted to encode the `at+carry` predicate for a given object x, with n facts of the type (`at` x y) and m facts of the type (`carry` x y), we would only need $\lceil \log(n+1) \rceil$ bits for storing the information, since we only have to know which of the `at`-facts is true, or if there is no such fact. In the latter case, we know that some fact of the type (`carry` x y) is involved and can look up which one it is in the encoding of `carry+free`. However, encoding `at+carry` first, thus reducing the space needed by `carry+free` is another possibility for encoding states, and is in fact the better alternative in this case. Since we cannot know which encoding yields the best results, we try them out systematically.

Although there is no need for using heuristics here since the number of conflicting possibilities is generally very small, we want to mention that as a rule of thumb it is generally a good idea to encode predicates that cover the largest number of facts first.

Predicates that are neither constant nor appear in any of the balanced merge predicates are encoded naively, using one bit for each possible fact. Those predicates are rare. In fact, in the considered benchmark set we only encountered them in the Grid domain, and there only for encoding the `locked` state of doors which obviously cannot further be compressed.

8 Experimental Results

In this section we provide data on the achieved compression to the state descriptions of the AIPS'98 planning competition problems. The problem suite consists of six different Strips domains, namely *Movie*, *Gripper*, *Logistics*, *Mystery*, *Mprime*, and *Grid*. In Table 1 we have exemplarily given the full state description for the first problem in the Logistics suite. The exhibited knowledge in the encoding can be easily extracted in form of *state invariants*, e.g. a package is either a location in a truck or in an airplane, each truck is restricted to exactly one city, and airplanes operate on airports only.

Table 2 depicts the state description length of all problems in the competition. Manually encoding some of the domains and comparing the results we often failed to devise a smaller state description length by hand.

Almost all of the execution time is spent on exploring predicate space. All the other phases added together never took more than a second of execution time. The time spent on exploring predicate space is not necessarily lost. When symbolically exploring planning space using BDDs, the operators need to be instantiated anyway for building the transition function, and if we keep track of all operator instantiations in the exploration phase this process can be sped up greatly.

	Movie		Gripper		Logistics		Mystery		Mprime		Grid	
problem	bits	sec	bits	sec	bits	sec	bits	sec	bits	sec	bits	sec
1-01	6	<1	11	<1	42	<1	28	<1	32	<1		
1-02	6	<1	15	<1	56	<1	117	<1	121	<1		
1-03	6	<1	19	<1	98	<1	77	<1	89	<1		
1-04	6	<1	23	<1	115	<1	50	<1	63	<1		
1-05	6	<1	27	<1	35	<1	86	<1	96	<1		
1-06	6	<1	31	<1	174	<1	148	<1	179	<1		
1-07	6	<1	35	<1	95	<1	82	<1	126	1		
1-08	6	<1	39	<1	254	1	90	<1	142	<1		
1-09	6	<1	43	<1	184	<1	83	<1	93	<1		
1-10	6	<1	47	<1	162	<1	291	<1	315	<1		
1-11	6	<1	51	<1	104	1	52	1	61	1		
1-12	6	<1	55	<1	195	<1	42	<1	56	<1		
1-13	6	<1	59	<1	287	1	291	<1	323	1		
1-14	6	<1	63	<1	282	1	320	1	346	1		
1-15	6	<1	67	<1	144	<1	184	<1	210	1		
1-16	6	<1	71	<1	205	<1	90	<1	120	<1		
1-17	6	<1	75	<1	190	<1	188	<1	202	<1		
1-18	6	<1	79	<1	270	1	112	1	160	1		
1-19	6	<1	83	<1	256	<1	129	<1	163	<1		
1-20	6	<1	87	<1	264	2	144	<1	169	2		
1-21	6	<1			300	1	205	<1	230	<1		
1-22	6	<1			530	3	234	1	283	1		
1-23	6	<1			166	<1	157	<1	186	<1		
1-24	6	<1			336	<1	229	<1	263	1		
1-25	6	<1			343	5	23	<1	26	<1		
1-26	6	<1			382	3	67	<1	86	<1		
1-27	6	<1			604	4	63	<1	67	<1		
1-28	6	<1			818	<1	38	<1	41	<1		
1-29	6	<1			566	1	74	<1	86	<1		
1-30	6	<1			470	8	109	<1	117	1		
2-01					26	<1			120	<1	67	<1
2-02					28	<1			84	<1	83	<1
2-03					39	<1			269	<1	93	<1
2-04					64	<1			129	<1	107	<1
2-05					63	<1			46	<1	139	1

Table 2. Length of state encodings and elapsed time of the AIPS'98 benchmark set. The data was generated on a Sun Ultra Sparc Station.

9 Related Work and Conclusion

There is some work in literature dealing with reformulation of planning problems. However, research mainly concentrates on inferring state invariants instead of minimizing the state description length.

Fox and Long, for example, have contributed several suggestions that have been implemented in the planner *Stan* (for STate ANalysis) [18]. The project is based on Graphplan [1] and uses a variety of techniques to exhibit domain-dependent information. In this context the automatic inference of state invariants is important. The pre-processor *Tim* (Type Inference Module) explores planning domains in order to find typings of untyped parameters [9]. The information is found by an algorithm starting with a projection of actions to their parameters establishing so-called *properties*, i.e. predicates together with the argument position filled by the objects. Given the properties and operators, transition rules are inferred (i.e. $\texttt{on}_1 \rightarrow \texttt{on}_1$ in Logistics) which constitute a finites state machine corresponding to the property exchanges. Types are found by exploration of membership patterns starting with the initial set.

Given the inferred type specification, in an additional analysis step three major state invariants can be found: *identity invariants*, *membership invariants* and *unique state invariants*. E.g in *Blocks World* we have $\forall x, y, z$ $\texttt{on}(y,x) \wedge \texttt{on}(z,x) \rightarrow y = z$, $\forall x\ \exists y$ $\texttt{on}(y,x) \vee \texttt{clear}(x)$, and $\forall x\ \neg(\exists y$ $\texttt{on}(y,x) \wedge \texttt{clear}(x))$ Furthermore, *Tim* has been extended to infer *cardinality constraints* such as $|\{x|\texttt{at-robot}(x)\}| = 1$ in *Gripper*. Tim is sound but not complete, i.e., it will find correct invariants but not all of them. Very recent unpublished work by Fox and Long focus *Mobile Analysis*, which constructs maps of locations that can be navigated by a mobile through an operator schema that gives the mobility.

The problem of finding state invariants is also addressed by Gerevini and Schubert [11]. Their planner *Discoplan* discovers two kinds of invariance rules, *single-valued* and *implicative constraints*. For example we have $\texttt{on}(x,y) \wedge y \neq z \Rightarrow \neg\texttt{on}(x,z)$ and $\texttt{on}(x,y) \wedge y \neq \texttt{table} \Rightarrow \neg\texttt{clear}(y)$ in *Blocks World*. While *Tim* improves explorations in *Graphplan*, in case of *Discoplan* the invariants improve satisfiability planning such as in *Satplan* [14]. Satplan itself is closely related to our approach of symbolically exploring planning space with BDDs, since both algorithms rely on a specification of the problem with boolean formulae.

The information gathered by *Tim* and *Discoplan* can be compared to our approach of balanced predicates and constraining the domains of predicates, since the presented algorithms exhibit domain-dependent knowledge leading to problem invariants as shown in the given example. As highlighted above the encoding in *Logistics* apparently give *identity invariants membership invariants* and *unique state invariants* as well as some *single-valued* and *implicative constraints*. Even *cardinality constraints* can be extracted from the encodings.

We conjecture that it is possible to extract the same invariants as in Tim and Discoplan from our encodings and that the knowledge inferred by our algorithms is more detailed, but there is an extraction process required to obtain the invariants from the encodings and to prove the assertion. On the other hand we think that the binary encoding length is probably the best performance measure to compare the inferred knowledge of different precompilers.

Literature reveals that an information gathering phase prior to search takes time. Through the efficiency of our approaches the time spent on these efforts is by far shorter than the time needed for constructing the transition function and the symbolic search phase itself. Automatically inferring problem-dependent knowledge in planning problems is challenging but an inevitable necessity for current state space search engines. The paper contributes efficient new algorithms based on symbolical manipulation and search. The promising results of BDD-based exploration according to the achieved encodings are given in [7]. We conclude that our approach to automatically infer compressed state descriptions mainly tailored to symbolic exploration reflects current research and could have a strong impact on current planning systems.

Acknowledgments: Thanks to F. Reffel for helpful discussions concerning the symbolic exploration phase. S. Edelkamp and M. Helmert are partially supported by DFG in project Ot-11/3 entitled *Heuristic Search and Its Application in Protocol Verification.*

References

1. A. Blum and M. Furst. Fast planning through planning graph analysis. In *IJ-CAI'95*, pages 1636–1642, 1995.
2. R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *ACM/IEEE Design Automation*, pages 688–694, 1985.
3. J. C. Culberson and J. Schaeffer. Searching with pattern databases. In *Proceedings of the Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in AI*, volume 1081 of *LNAI*, pages 402–416, 1996.
4. J. Eckerle and S. Schuierer. Efficient memory-limited graph search. *Lecture Notes in Computer Science*, 981:101–112, 1995.
5. S. Edelkamp and R. E. Korf. The branching factor of regular search spaces. In *AAAI-98*, pages 299–304, 1998.
6. S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *KI-98*, volume 1504 of *LNAI*, pages 81–92, Berlin, 1998. Springer.
7. S. Edelkamp and F. Reffel. Deterministic state space planning with BDDs. Technical Report 120, Computer Science Departement, University of Freiburg, 1999.
8. R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving.
9. M. Fox and D. Long. The automatic inference of state invariants in TIM. *JAIR*, 9:367–421, 1998.
10. M. Fox and D. Long. The detection and exploration of symmetry in planning problems. Technical Report 1/99, Department of Computer Science, University of Durham, 1999.
11. A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *AAAI-98*, pages 905–912. AAAI Press, 1998.
12. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on SSC*, 4:100, 1968.
13. A. Junghanns and J. Schaeffer. Single agent search in the presence of deadlocks. In *AAAI-98*, pages 228–233, 1998.
14. H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *AAAI-96*, pages 1194–1201, 1996.
15. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
16. R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI-97*, pages 700–705, 1997.
17. R. E. Korf. Complexity analysis of admissible heuristic search. In *AAAI-98*, pages 305–310, 1998.
18. D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *JAIR*, 10:87–115, 1998.
19. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.
20. E. Pednould. ADL: Exploring the middleground between Strips and situation calculus. *KR-89*, pages 324–332, 1989.
21. A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.
22. S. Russell. Efficient memory-bounded search methods. In *ECAI-92*, pages 1–5. Wiley, 1992.
23. L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI-93*, pages 756–761, 1993.